Mit Ebook auf DVD

Klaus Löffelmann

Entwicklerbuch

Microsoft

Visual Basic 2010

Grundlagen, Techniken, Profi-Know-how

Microsoft Press

Klaus Löffelmann, Sarika Calla Purohit

Microsoft Visual Basic 2010 – Das Entwicklerbuch

Klaus Löffelmann, Sarika Calla Purohit

Microsoft Visual Basic 2010 – Das Entwicklerbuch

Microsoft Press

Klaus Löffelmann, Sarika Calla Purohit: Microsoft Visual Basic 2010 – Das Entwicklerbuch
Microsoft Press Deutschland, Konrad-Zuse-Str. 1, 85716 Unterschleißheim
Copyright © 2010 by O'Reilly Verlag GmbH & Co. KG

Das in diesem Buch enthaltene Programmmaterial ist mit keiner Verpflichtung oder Garantie irgendeiner Art verbunden. Autor, Übersetzer und der Verlag übernehmen folglich keine Verantwortung und werden keine daraus folgende oder sonstige Haftung übernehmen, die auf irgendeine Art aus der Benutzung dieses Programmmaterials oder Teilen davon entsteht. Die in diesem Buch erwähnten Software- und Hardwarebezeichnungen sind in den meisten Fällen auch eingetragene Marken und unterliegen als solche den gesetzlichen Bestimmungen. Der Verlag richtet sich im Wesentlichen nach den Schreibweisen der Hersteller.

Das Werk, einschließlich aller Teile, ist urheberrechtlich geschützt. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Verlags unzulässig und strafbar. Das gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
12 11 10

ISBN 978-3-86645-535-1, PDF-eBook-ISBN 978-3-86645-389-0

© 2010 O'Reilly Verlag GmbH & Co. KG
Balthasarstraße 81, 50670 Köln
Alle Rechte vorbehalten

Korrektorat: Kristin Grauthoff, Lippstadt
Fachlektorat und Übersetzung: Thomas Irlbeck, München
Satz: Silja Brands, Uta Berghoff, ActiveDevelop, Lippstadt (http://www.ActiveDevelop.de)
Layout: Gerhard Alfes, mediaService, Siegen (www.media-service.tv)
Umschlaggestaltung: Hommer Design GmbH, Haar (www.HommerDesign.com)
Gesamtherstellung: Kösel, Krugzell (www.KoeselBuch.de)

Inhaltsverzeichnis

Vorwort XIX

Einleitung XXIII
Danksagungen XXIV
Support XXVI

Teil A – Der Einstieg in Sprache und Entwicklungs-Werkzeuge 1

1 Einführung 3
Einführung 4
Was brauchen Sie, um mit diesem Buch zu arbeiten? 6
Die Begleitdateien zum Buch und das E-Book 6

2 »Symbolischer Allzweckbefehlscode für Anfänger« 7
Visual Studio das erste Mal starten 8
Konsolenanwendungen (Console Applications) 10
Anwendung starten 12
Anatomie eines (Visual Basic-)Programms 14
Programmstart mit der Main-Methode 16
Methoden mit und ohne Rückgabewerte 19
Definieren von Methoden ohne Rückgabewerte mit Sub 19
Definieren von Methoden mit Rückgabewerten mit Function 20
Deklaration von Variablen 20
Nullables 23
Ausdrücke und Variablendefinitionen 24
Gleichzeitiges Deklarieren und Definieren von Variablen 25
Komplexe Ausdrücke 25
Boolesche Ausdrücke 26
Was sind Objekte im Unterschied zu »normalen« Datentypen? 27
Ableiten von Objekten und abstrakten Objekten 28
Eigenschaften 29
Typliterale zur Typfestlegung von Konstanten 30
Typsicherheit 32
Lokaler Typrückschluss 35
Arrays und Auflistungen 37

Programmcode bedingt ausführen ... 39
If … Then … Else … ElseIf … End If ... 39
Logische Operatoren And, Or, Xor, Not ... 40
Vergleichsoperatoren, die boolesche Ergebnisse zurückliefern ... 43
Select … Case … End Select ... 45
If-Operator und IIf-Funktion ... 46
Schleifen ... 47
For … Next-Schleifen ... 47
For Each-Schleifen ... 50
Do … Loop- und While … End While-Schleifen ... 51
Exit – Vorzeitiges Verlassen von Schleifen ... 53
Continue – Vorzeitige Schleifenwiederholung ... 53
Verkürzter Zugriff auf Objekteigenschaften und -methoden mit With … End With ... 53
Gültigkeitsbereiche (Scope) von Variablen ... 54
Die Operatoren += und –= und ihre Verwandten ... 56
Die Bitverschiebeoperatoren << und >> ... 57
Fehlerbehandlung im Code ... 58
Elegantes Fehlerabfangen mit Try/Catch/Finally ... 60

3 Einführung in .NET Framework ... 65
Was ist .NET und aus was besteht es? ... 66
Was ist eine Assembly? ... 67
Was ist ein Namespace? ... 67
Was versteckt sich hinter CLR (Common Language Runtime) und CLI (Common Language Infrastructure)? ... 71
Was ist die FCL (Framework Class Library) und was die BCL (Base Class Library)? ... 72
Was ist das CTS (Common Type System)? ... 73
Was ist CIL/MSIL (Microsoft Intermediate Language) und wozu dient der JITter? ... 73

4 Rundgang durch Visual Studio ... 75
Einführung ... 76
Visual Studio das erste Mal starten – das Profil auswählen ... 77
Die Startseite – der Ausgangsort für Ihre Entwicklungsaufgaben ... 79
Mit der Entwicklung beginnen – neue Projekte erstellen ... 82
Aus früheren Visual Studio-Versionen auf Visual Studio 2010 migrieren ... 85
Mit Visual Studio 2003-2008 erstellte Projekte aktualisieren ... 85
Visual Basic 6.0-Anwendungen auf Visual Studio 2010 aktualisieren ... 87
Anwendungen für mehrere .NET Framework-Versionen (Multizielversionen) unter Visual Studio 2010 ... 87
Die Geschichte der Multizielversionen ... 88
Die .NET Framework-Zielversion für Anwendungen ändern ... 91
Wissenswertes über Multizielversionen ... 95
Grenzen der Multizielversionen ... 95

Ein näherer Blick auf die neue und verbesserte WPF-basierte IDE ... 97
Bildschirmressourcen verwalten und effizient nutzen ... 98
Beständigkeit des Fensterlayouts ... 103
Code: navigieren, suchen und nachvollziehen ... 107
Die Hervorhebungsfarbe ändern ... 110
Regionen und Gliederungen ... 111
Architektur-Explorer ... 112
Sequenzdiagramme ... 113
Klassendiagramme ... 114
Bottom-up-Programmierung ... 114
Generate From Usage ... 117
Visual Studio erweitern ... 118
Visual Studio-Erweiterungen verwalten ... 119
Erweiterungsarten ... 120

5 Einführung in Windows Forms – Designer und Code-Editor am Beispiel ... 121
Das Fallbeispiel – der DVD-Hüllen-Generator »Covers« ... 122
Das »Pflichtenheft« von Covers ... 123
Erstellen eines neuen Projekts ... 126
Gestalten von Formularen mit dem Windows Forms-Designer ... 128
Positionieren von Steuerelementen ... 128
Häufige Arbeiten an Steuerelementen mit Smarttags erledigen ... 132
Dynamische Anordnung von Steuerelementen zur Laufzeit ... 133
Automatisches Scrollen von Steuerelementen in Containern ... 143
Selektieren von Steuerelementen, die Sie mit der Maus nicht erreichen ... 145
Festlegen der Tabulatorreihenfolge (Aktivierreihenfolge) von Steuerelementen ... 146
Über die Eigenschaften Name, Text und Caption ... 148
Einrichten von Bestätigungs- und Abbrechen-Funktionalitäten für Schaltflächen in Formularen ... 151
Hinzufügen neuer Formulare zu einem Projekt ... 151
Wie geht's weiter? ... 154
Namensgebungskonventionen für Steuerelemente in diesem Buch ... 154
Funktionen zum Layouten von Steuerelementen im Designer ... 155
Tastaturkürzel für die Platzierung von Steuerelementen ... 157
Der Code-Editor ... 157
Die Darstellung im Editor auf die richtige Größe einstellen ... 158
Viele Wege führen zum Code-Editor ... 159
IntelliSense – Ihr stärkstes Zugpferd im Coding-Stall ... 159
Automatische Vervollständigung von Struktur-Schlüsselwörtern und Codeeinrückung ... 162
Fehlererkennung im Code-Editor ... 163
XML-Dokumentationskommentare für IntelliSense bei eigenen Objekten und Klassen ... 167
Hinzufügen neuer Codedateien zum Projekt ... 171

Code umgestalten (Refactoring) ... 173
Die Bibliothek der Codeausschnitte (Code Snippets Library) ... 176
Speichern von Anwendungseinstellungen mit dem Settings-Designer ... 180

6 Einführung in Windows Presentation Foundation ... 189
Was ist Windows Presentation Foundation? ... 190
Was ist so neu an WPF? ... 191
25 Jahre Windows, 25 Jahre gemalte Schaltflächen ... 192
Wie WPF Designer und Entwickler zusammenbringt ... 203
XAML: Extensible Application Markup Language ... 205
Ereignisbehandlungsroutinen in WPF und Visual Basic ... 212
Die XAML-Syntax im Überblick ... 213
ImageResizer – das praktische WPF-Beispiel ... 216

7 Die essenziellen .NET-Datentypen ... 255
Numerische Datentypen ... 256
Numerische Datentypen deklarieren und definieren ... 257
Delegation numerischer Berechnungen an den Prozessor ... 258
Hinweis zur CLS-Konformität ... 260
Die numerischen Datentypen auf einen Blick ... 261
Tabellarische Zusammenfassung der numerischen Datentypen ... 268
Rundungsfehler bei der Verwendung von Single und Double ... 269
Besondere Funktionen, die für alle numerischen Datentypen gelten ... 272
Spezielle Funktionen der Fließkommatypen ... 276
Spezielle Funktionen des Wertetyps Decimal ... 278
Der Datentyp Char ... 278
Der Datentyp String ... 279
Strings – gestern und heute ... 280
Strings deklarieren und definieren ... 280
Prüfen, ob ein String »leer« ist – String.IsNullOrWhiteSpace ... 281
Automatische Zeichenwiederholungen durch den String-Konstruktor (String$-Funktionalität aus VBA) ... 281
Einem String Zeichenketten mit Sonderzeichen zuweisen ... 283
Speicherbedarf von Strings ... 284
Strings sind unveränderlich ... 284
Speicheroptimierung von Strings durch .NET Framework ... 285
Ermitteln der String-Länge ... 286
Ermitteln von Teilen eines Strings oder eines einzelnen Zeichens ... 286
Angleichen von String-Längen ... 287
Suchen und Ersetzen ... 287
Algorithmisches Auflösen eines Strings in Teile ... 291
Iterieren durch einen String ... 292
StringBuilder vs. String – wenn es auf Geschwindigkeit ankommt ... 293

Der Datentyp Boolean ... 297
Konvertieren von und in numerische Datentypen ... 297
Konvertierung von und in Strings ... 298
Der Datentyp Date ... 298
Rechnen mit Zeiten und Datumswerten – TimeSpan ... 299
Bibliothek mit brauchbaren Datumsrechenfunktionen ... 300
Zeichenketten in Datumswerte wandeln ... 303
.NET-Äquivalente primitiver Datentypen ... 306
GUIDs ... 308
Konstanten und Nur-Lesen-Felder (Read-only-Feldvariablen) ... 310
Konstanten ... 310
Nur-Lesen-Felder ... 311

Teil B – Objektorientiertes Programmieren ... 313

8 Einführung in die objektorientierte Programmierung ... 315
Einführung in Klassen und Objekte ... 318
Miniadresso – die prozedurale Variante ... 318

9 Klassentreffen ... 321
Was ist eine Klasse? ... 322
Klassen mit New instanziieren ... 324
New oder nicht New – wieso es sich bei Objekten um Verweistypen handelt ... 325
Nothing ... 329
Nothing als Null-Referenz-Zeiger ... 329
Nothing in Zuweisungen für Standardwerte ... 330
Klassen verwenden ... 331
Wertetypen ... 332
Wertetypen mit Structure erstellen ... 333
Wertetypen durch Zuweisung klonen ... 334

10 Klassencode entwickeln ... 335
Eigenschaften ... 336
Zuweisen von Eigenschaften ... 338
Ermitteln von Eigenschaften ... 339
Viel Tipparbeit sparen mit automatisch implementierten Eigenschaften ... 341
Eigenschaften mit Parametern ... 342
Default-Eigenschaften (Standardeigenschaften) ... 343
Öffentliche Variablen oder Eigenschaften – eine Glaubensfrage? ... 345
Der Konstruktor einer Klasse – bestimmen, was bei New passiert ... 347
Parametrisierte Konstruktoren ... 349
Hat jede Klasse einen Konstruktor? ... 351
Klassenmethoden mit Sub und Function ... 356

Überladen von Methoden, Konstruktoren und Eigenschaften ... 357
Methodenüberladung und optionale Parameter ... 359
Gegenseitiges Aufrufen von überladenen Methoden ... 362
Gegenseitiges Aufrufen von überladenen Konstruktoren ... 363
Überladen von Eigenschaftenprozeduren mit Parametern ... 364
Zugriffsmodifizierer für Klassen, Methoden und Eigenschaften – Gültigkeitsbereiche definieren ... 365
Zugriffsmodifizierer bei Klassen ... 365
Zugriffsmodifizierer bei Prozeduren (Subs, Functions, Properties) ... 366
Zugriffsmodifizierer bei Variablen ... 366
Unterschiedliche Zugriffsmodifizierer für Eigenschaften-Accessors ... 367
Statische Elemente ... 368
Statische Methoden ... 369
Statische Eigenschaften ... 371
Über mehrere Codedateien aufgeteilter Klassencode – das Partial-Schlüsselwort ... 371
Partieller Klassencode bei Methoden und Eigenschaften ... 372

11 Vererben von Klassen und Polymorphie ... 373
Wiederverwendbarkeit von Klassen durch Vererbung (Inheritance) ... 374
Initialisierung von Feldvariablen bei Klassen ohne Standardkonstruktoren ... 384
Überschreiben von Methoden und Eigenschaften ... 386
Überschreiben vorhandener Methoden und Eigenschaften von .NET Framework-Klassen ... 388
Polymorphie ... 389
Ein erstes einfaches Beispiel zur Polymorphie ... 392
Polymorphie in der Praxis ... 395
Polymorphie und der Gebrauch von Me, MyClass und MyBase ... 408
Abstrakte Klassen und virtuelle Prozeduren ... 409
Eine Klasse mit MustInherit als abstrakt deklarieren ... 410
Eine Methode oder Eigenschaft einer abstrakten Klasse mit MustOverride als virtuell deklarieren ... 411
Schnittstellen (Interfaces) ... 412
Unterstützung bei abstrakten Klassen und Schnittstellen durch den Editor ... 419
Schnittstellen, die Schnittstellen implementieren ... 425
Einbinden mehrerer Schnittstellen in eine Klasse ... 426
Die Methoden und Eigenschaften von Object ... 427
Ausgeben der String-Repräsentation eines Objekts mit ToString ... 427
Prüfen auf Gleichheit von Objekten mit Object.Equals oder dem Is/IsNot-Operator ... 427
Equals, Is und IsNot im praktischen Einsatz ... 430
Übersicht über die Eigenschaften und Methoden von Object ... 431
Shadowing (Überschatten) von Klassenprozeduren ... 432
Shadows als Unterbrecher der Klassenhierarchie ... 433
Sonderform »Modul« in Visual Basic ... 437
Singleton-Klassen und Klassen, die sich selbst instanziieren ... 438

12 Entwickeln von Wertetypen ... 441
Erstellen von Wertetypen mit Structure am praktischen Beispiel ... 442
Verhalten der Parameterübergabe mit ByVal und ByRef steuern ... 448
Konstruktoren und Standardinstanziierungen von Wertetypen ... 449
Wann Werte-, wann Referenztypen verwenden? ... 451
Gezieltes Zuweisen von Speicherbereichen für Strukturmember mit den StructLayout- und FieldOffset-Attributen ... 451

13 Typenumwandlung (Type Casting) und Boxing von Wertetypen ... 455
Konvertieren von primitiven Typen ... 456
Konvertieren von und in Zeichenketten (Strings) ... 458
Konvertieren von Strings mit den Parse- und ParseExact-Methoden ... 458
Konvertieren in Strings mit der ToString-Methode ... 459
Abfangen von fehlschlagenden Typkonvertierungen mit TryParse oder Ausnahmehandlern ... 460
Casten von Referenztypen mit DirectCast ... 461
TryCast – herausfinden, ob gecastet werden kann ... 462
IsAssignableFrom – Test auf erlaubtes Casting ohne zusätzliche Objektvariable ... 463
Boxing von Wertetypen und primitiven Typen ... 463
Übrigens: Was DirectCast nicht kann ... 466
Wann wird und wann wird nicht geboxt? ... 466
Wertänderung von durch Schnittstellen geboxten Wertetypen ... 467

14 Beerdigen von Objekten – Dispose, Finalize und der Garbage Collector ... 471
Der Garbage Collector – die Müllabfuhr in .NET ... 474
Wie der Garbage Collector arbeitet ... 476
Die Geschwindigkeit der Objektbereitstellung ... 478
Finalize ... 479
Wann Finalize nicht stattfindet ... 481
Dispose ... 484
Implementierung eines hochauflösenden Timers als IDisposable ... 485
Unterstützung durch den Visual Basic-Editor beim Einfügen eines Disposable-Patterns. 491
Gezieltes Freigeben von Objekten mit Using ... 494

15 Eigene Operatoren für benutzerdefinierte Typen ... 497
Einführung in Operatorenprozeduren ... 498
Vorbereitung einer Struktur oder Klasse für Operatorenprozeduren ... 499
Implementierung von Operatoren ... 503
Überladen von Operatorenprozeduren ... 505
Implementierung von Vergleichsoperatoren ... 506
Implementierung von Typkonvertierungsoperatoren mit dem Operator CType ... 506

Implementieren von Wahr- und Falsch-Auswertungsoperatoren ... 508
Problembehandlungen bei Operatorenprozeduren ... 510
Aufgepasst bei der Verwendung von Referenztypen ... 510
Übersicht der implementierbaren Operatoren ... 511

16 Ereignis, Delegaten und Lambda-Ausdrücke ... 513
Konsumieren von Ereignissen mit WithEvents und Handles ... 515
Auslösen von Ereignissen ... 518
Events können nicht vererbt werden – der Umweg über Onxxx ... 519
Ereignisparameter zur Verfügung stellen ... 521
Die Quelle des Ereignisses: Sender ... 521
Nähere Informationen zum Ereignis: EventArgs ... 523
Delegaten ... 525
Delegaten an Methoden übergeben ... 531
Lambda-Ausdrücke ... 534
Einzeilige Ausdrucks- und mehrzeilige Anweisungs-Lambdas ... 535
Dynamisches Anbinden von Ereignissen mit AddHandler ... 538
Implementieren eigener Event Handler ... 547

17 Enums ... 553
Bestimmung der Werte der Aufzählungselemente ... 555
Bestimmung der Typen der Aufzählungselemente ... 556
Ermitteln des Elementtyps zur Laufzeit ... 556
Konvertieren von Enums ... 556
In Zahlenwerte umwandeln und aus Werten definieren ... 557
In Strings umwandeln und aus Strings definieren ... 557
Flags-Enums (Flags-Aufzählungen) ... 558
Abfrage von Flags-Aufzählungen ... 559

18 Generics (Generika) und generische Auflistungen ... 561
Einführung ... 562
Generics: Verwenden einer Codebasis für verschiedene Typen ... 562
Lösungsansätze ... 563
Standardisierung der Codebasis eines Typs durch den Einsatz von Generics ... 565
Beschränkungen (Constraints) ... 568
Beschränkungen für generische Typen auf eine bestimmte Basisklasse ... 568
Beschränkungen auf Klassen, die bestimmte Schnittstellen implementieren ... 573
Beschränkungen auf Klassen, die über einen Standardkonstruktor verfügen ... 576
Beschränkungen auf Wertetypen ... 576
Kombinieren von Beschränkungen und Bestimmen mehrerer Typparameter ... 577

19 Typen für Fortgeschrittene ... 579
Nullable Wertetypen ... 580
Aufpassen beim Umgang mit booleschen Ausdrücken auf Basis von Nullables! ... 583
Besonderheiten bei Nullable beim Boxen ... 584
Unterschied zwischen Nothing und Nothing als Standardwert ... 586
Generische Delegaten ... 586
Action(Of T) ... 587
Function(Of T) ... 590
Tuple(Of T) ... 590
Typvarianz ... 591
Erweiterungsmethoden ... 595
Das Haupteinsatzgebiet von Erweiterungsmethoden ... 596
Erweiterungsmethoden zur Vereinfachung von Auflistungsinitialisieren ... 598

Teil C – Programmieren von und mit Datenstrukturen aus .NET Framework ... 599

20 Arrays und Auflistungen ... 601
Grundsätzliches zu Arrays ... 602
Initialisierung von Arrays ... 603
Änderung der Array-Dimensionen zur Laufzeit ... 604
Die Magie von ReDim ... 604
Wertevorbelegung von Array-Elementen im Code ... 606
Typrückschluss bei der Verwendung von Array-Initialisierern ... 607
Mehrdimensionale und verschachtelte Arrays ... 609
Die wichtigsten Eigenschaften und Methoden von Arrays ... 610
Implementierung von Sort und BinarySearch für eigene Klassen ... 613
Verwenden von Lambdas mit Arrays-Methoden ... 617
Enumeratoren ... 618
Benutzerdefinierte Enumeratoren durch Implementieren von IEnumerable ... 619
Grundsätzliches zu Auflistungen (Collections) ... 622
Initialisierung von Auflistungen ... 626
Vereinfachung von Auflistungsinitialisierern durch Erweiterungsmethoden ... 626
Wichtige Auflistungen von .NET Framework, die Sie kennen sollten ... 628
ArrayList – universelle Ablage für Objekte ... 628
Typsichere Auflistungen auf Basis von CollectionBase ... 631
Hashtables – für das Nachschlagen von Objekten ... 634
Anwenden von Hashtables ... 635
Verwenden eigener Klassen als Schlüssel (Key) ... 644
Enumerieren von Datenelementen in einer Hashtable ... 647
DictionaryBase ... 648
Queue – Warteschlangen im FIFO-Prinzip ... 648
Stack – Stapelverarbeitung im LIFO-Prinzip ... 650
SortedList – Elemente ständig sortiert halten ... 651

Generische Auflistungen (Generic Collections) ... 653
List(Of)-Auflistungen und Lambda-Ausdrücke ... 655
KeyedCollection – Schlüssel/Wörterbuch-Auflistung mit zusätzlichen Index-Abrufen ... 660
Elementverkettungen mit LinkedList(Of) ... 663

21 Serialisierung ... 665
Einführung in Serialisierungstechniken ... 666
Serialisieren mit SoapFormatter und BinaryFormatter ... 669
Flaches und tiefes Klonen von Objekten ... 674
Universelle DeepClone-Methode ... 677
Serialisieren von Objekten mit Zirkelverweisen ... 680
Serialisierung von Objekten unterschiedlicher Versionen beim Einsatz von BinaryFormatter- oder SoapFormatter-Klassen ... 682
XML-Serialisierung ... 683
Prüfen der Versionsunabhängigkeit der XML-Datei ... 688
Serialisierungsfehler bei KeyedCollection ... 688

22 Attribute und Reflection ... 693
Genereller Umgang mit Attributen ... 694
Einsatz von Attributen am Beispiel von ObsoleteAttribute ... 695
Die speziell in Visual Basic verwendeten Attribute ... 696
Einführung in Reflection ... 697
Die Type-Klasse als Ausgangspunkt für alle Typenuntersuchungen ... 699
Klassenanalysefunktionen, die ein Type-Objekt bereitstellt ... 700
Objekthierarchie von MemberInfo und Casten in den spezifischen Info-Typ ... 704
Ermitteln von Eigenschaftswerten über PropertyInfo zur Laufzeit ... 704
Erstellung benutzerdefinierter Attribute und deren Erkennen zur Laufzeit ... 705
Ermitteln von benutzerdefinierten Attributen zur Laufzeit ... 709

Teil D – Entwicklungsvereinfachungen in Visual Basic 2010 ... 711

23 My als Abkürzung zu Framework-Funktionen verwenden ... 713
Die Visual Basic 2010-Vereinfachungen am Beispiel DotNetCopy ... 715
DotNetCopy mit /Autostart und /Silent-Optionen ... 719
Die prinzipielle Funktionsweise von DotNetCopy ... 721
Der My-Namespace ... 722
Formulare ohne Instanziierung aufrufen ... 724
Auslesen der Befehlszeilenargumente mit My.Application.CommandLineArgs ... 726
Gezieltes Zugreifen auf Ressourcen mit My.Resources ... 727
Anlegen und Verwalten von Ressourcenelementen ... 728
Abrufen von Ressourcen mit My.Resources ... 728

Internationalisieren von Anwendungen mithilfe von Ressourcendateien und dem My-Namespace ... 730
Vereinfachtes Durchführen von Dateioperationen mit My.Computer.FileSystem ... 733
Verwenden von Anwendungseinstellungen mit My.Settings ... 736

24 Das Anwendungsframework ... 739
Die Optionen des Anwendungsframeworks ... 741
Windows XP Look and Feel für eigene Windows-Anwendungen –Visuelle XP-Stile aktivieren ... 741
Verhindern, dass Ihre Anwendung mehrfach gestartet wird –Einzelinstanzanwendung erstellen ... 741
Eigene Einstellungen beim Herunterfahren automatisch sichern – My.Settings beim Herunterfahren speichern ... 741
Bestimmen, welcher Benutzer-Authentifizierungsmodus verwendet wird ... 742
Festlegen, wann eine Anwendung »zu Ende« ist – Modus für das Herunterfahren ... 742
Einen Splash-Dialog beim Starten von komplexen Anwendungen anzeigen – Begrüßungsbildschirm ... 742
Eine Codedatei implementieren, die Anwendungsereignisse (Starten, Beenden, Netzwerkzustand, generelle Fehler) behandelt ... 742

Teil E – Language Integrated Query – LINQ ... 747

25 Einführung in Language integrated Query (LINQ) ... 749
Wie »geht« LINQ prinzipiell ... 751
Erweiterungsmethoden als Basis für LINQ ... 754
Die Where-Methode ... 755
Die Select-Methode ... 756
Anonyme Typen ... 757
Typrückschluss für generische Typparameter ... 757
Kombinieren von LINQ-Erweiterungsmethoden ... 760
Vereinfachte Anwendung von LINQ – Erweiterungsmethoden mit der LINQ-Abfragesyntax ... 761

26 LINQ to Objects ... 763
Einführung in LINQ to Objects ... 764
Der Aufbau einer LINQ-Abfrage ... 764
Performance von LINQ-Abfragen ... 770
Kombinieren und verzögertes Ausführen von LINQ-Abfragen ... 771
Parallelisieren von LINQ-Abfragen mit AsParallel ... 773
Faustregeln für das Erstellen von LINQ-Abfragen ... 776
Kaskadierte Abfragen ... 776
Gezieltes Auslösen von Abfragen mit ToArray oder ToList ... 777

Verbinden mehrerer Auflistungen zu einer neuen ... 778
Implizite Verknüpfung von Auflistungen ... 778
Explizite Auflistungsverknüpfung mit Join ... 780
Gruppieren von Auflistungen ... 781
Gruppieren von Listen aus mehreren Auflistungen ... 782
Group Join ... 783
Aggregatfunktionen ... 784
Zurückliefern mehrerer verschiedener Aggregierungen ... 785
Kombinieren von gruppierten Abfragen und Aggregatfunktionen ... 785

27 LINQ to XML ... 787
Einführung in LINQ to XML ... 788
XML-Dokumente verarbeiten – damals und heute ... 789
XML-Literale – XML direkt im Code ablegen ... 790
Einbetten von Ausdrücken in XML-Literalen ... 791
Erstellen von XML mithilfe von LINQ ... 791
Abfragen von XML-Dokumenten mit LINQ to XML ... 793
IntelliSense-Unterstützung für LINQ To XML-Abfragen ... 794

28 LINQ to Entities – Programmieren mit Entity Framework ... 797
Voraussetzungen zum Test der Beispiele ... 799
Download und Installation von SQL Server 2008 Express Edition with Advanced Services ... 799
Installation der AdventureWorks Beispieldatenbanken ... 807
Prinzipielle Funktionsweise eines Entity Data Model (EDM) ... 809
LINQ to Entities – ein erstes Praxisbeispiel ... 811
Nachträgliches Ändern des Entitätscontainernamens ... 816
Abfrage von Daten eines Entitätsmodells ... 818
Abfrage von Daten mit LINQ to Entities-Abfragen ... 819
Wie Abfragen zum Datenprovider gelangen – Entity SQL (eSQL) ... 820
Anpassen des Namens der Entitätenmenge ... 820
Generierte SQL-Anweisungen unter die Lupe nehmen ... 821
Lazy Loading und Eager Loading in Entity Framework ... 823
Anonymisierungsvermeidung bei Abfragen in verknüpften Tabellen ... 827
Kompilierte Abfragen ... 829
Daten verändern, speichern, einfügen und löschen ... 830
Datenänderungen mit SaveChanges an die Datenbank übermitteln ... 831
Einfügen von verknüpften Daten in Datentabellen ... 832
Daten aus Tabellen löschen ... 834
Concurrency-Checks (Schreibkonfliktprüfungen) ... 836
Datenmodell aus Datenbank aktualisieren ... 838
Model First-Design ... 839

Vererbung im konzeptionellen Datenmodell ... 845
T-SQL-Befehle im Objektkontext direkt ausführen ... 849
Arbeiten mit gespeicherten Prozeduren ... 850
Ausblick ... 852

Teil F – Anwendungsausführung parallelisieren ... 853

29 Programmieren mit der Task Parallel Library (TPL) ... 855
Einführung in die Technik des Threading ... 856
Die verschiedenen Möglichkeiten, Threads zu starten, im Schnelldurchlauf ... 862
Verwenden der Thread-Klasse ... 864
Asynchrones Aufrufen von Delegaten ... 866
Verwenden der Task-Klasse ... 867
Direktes Verwenden eines Thread des Threadpools ... 868
Zugriff auf Steuerelemente von Threads ... 868
Parallelisierung mit Parallel.For und Parallel.ForEach ... 873
Parallel.For ... 874
Parallel.ForEach ... 879
CancellationToken – Exit For für Parallel.For und Parallel.ForEach ... 881
Vermeiden von Fehlern beim Parallelisieren von Schleifen ... 885
Arbeiten mit Tasks ... 888
Warten auf das Ende von Tasks – WaitOne, WaitAny und WaitAll ... 891
Task mit und ohne Rückgabeparameter ... 892
Verhindern, dass die Benutzeroberfläche beim Warten auf Tasks blockiert ... 895
Vorzeitiges Abbrechen von Tasks mit CancellationToken ... 897
Threads synchronisieren ... 902
Synchronisieren von Threads mit SyncLock ... 904
Mehr Flexibilität in kritischen Abschnitten mit der Monitor-Klasse ... 906
Synchronisieren von beschränkten Ressourcen mit Mutex ... 911
Wie geht es weiter? ... 914

Stichwortverzeichnis ... 915

Vorwort

Von Lisa Feigenbaum
Microsoft Visual Studio 2010 Community Manager

Visual Basic 2010 ist eine sehr aufregende Version der Visual Basic-Sprache, deren Versionsnummer mit der Version 10 sogar schon zweistellig wird. Das ist eine phänomenale Errungenschaft für eine Programmiersprache und demonstriert den großen Nutzen, den sie den Entwicklern Jahr um Jahr zur Verfügung stellte. Visual Basic galt immer als Werkzeug erster Wahl, um Microsoft Plattformen verfügbar zu machen und auf einfache Weise benutzen zu können. Während sich die eigentlichen Techniken und Geräte über die Zeit weiterentwickelt haben, ist die Grundideologie von Visual Basic stets dieselbe geblieben. Mit den Visual Basic-Versionen 1.0 und 3.0 hat es angefangen: Diese Versionen haben die Vorgehensweise bei der Anwendungsentwicklung unter Windows revolutioniert, indem sie sie Entwicklern in einer bis dahin nicht gekannten Weise zugänglich gemacht hat. Versionen 4 bis 6 sorgten dann im weiteren Verlauf für eine stark vereinfachte Vorgehensweise beim Entwickeln von Komponenten durch COM, OLE-Automatisierung und ActiveX. Und schließlich, ab Visual Basic 7, befähigte die Sprache Entwickler, die Common Language Runtime und viele weitere .NET Framework-Technologien zu verwenden und ihnen zum Durchbruch zu verhelfen.

Dieses Buch enthält viele Beispiele dafür, indem es zeigt, wie man die Visual Basic Datentypen von .NET Framework richtig einsetzt, wie man Language Integrated Query (LINQ), die Windows Presentation Foundation (WPF) oder auch die Task Parallel Library (TPL) richtig verwendet. Gerade LINQ hatte einen besonderen Einfluss auf die Sprache, stellt es doch einen vereinheitlichenden Weg dar, um auf Daten von Objekten, von XML-Dokumenten oder von relationalen Datenquellen zuzugreifen. Eines der revolutionärsten Features die dabei im Rahmen von LINQ vorgestellt wurden, sind XML-Literale, die Visual Basic vor allen anderen Sprachen zum produktivsten Werkzeug für den Umgang mit XML machen.

Ein Blick in die Zukunft lässt drei große Entwicklungsrichtungen erkennen, die Visual Basic schon jetzt und wohl auch in der Zukunft beeinflussen werden: Die deklarative, die dynamische und die parallelisierende Programmierung. Die deklarative Entwicklung befähigt Entwickler, dem Programm direkt mitzuteilen, *was* es zu tun hat, also in detaillierten Schritten festzulegen, *wie* der Compiler die entsprechenden Befehle dazu generieren soll. Diese Vorgehensweise war schon immer das Entwurfsprinzip von Visual Basic, bei der wir bestrebt sind, die Ausdrucksstärke der Sprache zu erhöhen, sodass Sie »mehr durch weniger Code sagen können«. Einige Beispiele dafür waren in Visual Basic 9 LINQ oder der lokale Typrückschluss bei der Variablendeklaration. Visual Basic 10 führt neue, ähnlich effiziente Techniken mit mehrzeiligen Lambda-Funktionen, Array-Literalen, Auflistungsinitialisierern, automatisch implementierten Eigenschaften und der impliziten Zeilenfortführung ein – auf all dies geht dieses Buch ein.

Dynamische Programmierung ist eine weitere Disziplin, die das Design von Visual Basic beeinflusst hat. Spätes Binden (Late Binding) ist ein wichtiges Feature, das Visual Basic zu einer wichtigen Sprache in der Office- und COM-Entwicklung gemacht hat. In Visual Basic 10 haben wir die Unterstützung für das späte Binden dahingehend erweitert, dass es nunmehr mit anderen dynamischen Typumgebungen zusammenarbeiten kann – etwa mit JavaScript oder IronPython. Ermöglicht wurde das durch die DLR, die Dynamic Language Runtime, die erstmals mit .NET Framework 4.0 zur Verfügung steht. Und schließlich gibt es die parallelisierende Programmierung, als Trend, den man nicht übersehen darf, und der viele generelle Vorge-

hensweisen in der Entwicklung beeinflusst. Ob Ihre Anwendung auf einer Maschine mit mehreren Prozessor-Cores oder mehreren Prozessoren, in einer geclusterten Umgebung in Ihren Räumlichkeiten, in einer verteilten Umgebung in der Cloud oder sogar auf einem einfachen Ein-Prozessor-Core-Rechner mit sehr Eingabe-/Ausgabe-lastigen Anforderungen läuft – parallelisierende Programmierung kann Ihnen stets behilflich sein, die Ausführungsgeschwindigkeit zu verbessern. .NET Framework 4.0 stellt Ihnen einige leistungsstarke Werkzeuge für die parallelisierende Programmierung zur Verfügung, wie beispielsweise die *Task Parallel Library* oder auch Parallel LINQ. Teil F dieses Buchs zeigt, wie Sie diese Technologien in Visual Basic verwenden können.

Visual Basic ist eine lebhafte Umgebung, und wir laden Sie herzlich ein, in Visual Studio 2010 einzutauchen! Ganz gleich, ob sie bereits vorherige Versionen von Visual Basic oder andere Sprachen zum objektorientierten Programmieren (OOP) verwendet haben, oder OOP absolutes Neuland für Sie ist: Dieses Buch stellt Ihnen die Informationen zur Verfügung, damit Sie schnell produktiv arbeiten können. Es erklärt die Programmierkonzepte Visual Basic, Visual Studio und.NET Framework von Grund auf, und errichtet damit ein starkes Wissensfundament. Für den erfahreneren Leser taucht dieses Buch tief in diese Themen ein, und es stellt obendrein auch Abschnitte zur Verfügung, die verraten, was neu im 2010er Release von Visual Studio ist. Das Buch deckt eine ganze Reihe verschiedener Themen ab, von denen einige technologiespezifisch (wie beispielsweise WPF), andere anwendungsunabhängig (wie die Garbage Collection oder das Thema Serialisierung) sind. Dieses Buch verschafft Ihnen ein solides Basiswissen, das es Ihnen ermöglicht, Anwendungen für jede Plattform, auf die Visual Studio abzielen kann, zu entwickeln – wie SharePoint, das Web oder für die Cloud.

Als Visual Studio Community Manager genieße ich es immer, Mitglieder der Visual Studio Community persönlich kennen zu lernen. Bei einem der ersten Male, als Klaus mir schrieb, zitierte er ein Motto, das er von seiner Großmutter gelernt hatte: »Der schlimmste Versuch ist der, den du nie gemacht hast!«. Zu diesem Zeitpunkt wusste ich, dass er eine sehr ambitionierte Persönlichkeit war! Klaus schreibt seit über 20 Jahren Computerbücher – zum Commodore 16, Commodore 64, Commodore 128, Atari ST, Amiga, zu Visual Basic 1, Visual Basic 3, Visual Basic 4, Visual Basic 5, Visual Basic 6, Visual Studio 2003, Visual Studio 2005, Visual Studio 2008 und jetzt Visual Studio 2010. Ich habe Klaus bei unterschiedlichen Anlässen in den verschiedenen Städten auf der ganzen Welt getroffen: Antwerpen, Berlin und Seattle. Sein erster Trip nach Seattle war im Rahmen des Microsoft Most Valuable Professional (MVP) Summits. Das MVP-Programm zeichnet Microsoft Technologie-Experten für ihren Einfluss auf die Community aus, und Klaus wurde dies aufgrund seiner großen Unterstützung, die er der Community durch das Schreiben von Büchern, Durchführen von Webcasts und dem Überarbeiten von Inhalten des Microsoft Developer Networks zukommen ließ, zuteil. Klaus besuchte Seattle das zweite Mal, um dieses Buch zu schreiben und empfand es als große Inspiration, an dem Platz zu sein, an dem »the Magic happens«. Ich nahm Klaus mit auf eine Tour durch die Microsoft-Büros, und auf dieser Tour lernte er weitere Kollegen des Visual Basic-Produkt-Teams kennen. Es war ein sehr aufregender Besuch, und Klaus zeigte auch hier seine Begeisterung für Visual Basic.

Ich war erfreut, den Kontakt zwischen Sarika und Klaus herzustellen, die sich als großartige Partnerin beim Schreiben dieses Buchs erwies. Als Testleiterin des Microsoft Visual Studio Professional Teams verfügt Sarika über eine umfangreiche Expertise auf dem Gebiet Visual Studio. Sarika startete Ihre Karriere bei Microsoft im Jahre 2002, und sie hat seitdem an Visual Studio 2003, Visual Studio 2005, Visual Studio 2008 und Visual Studio 2010 mitgearbeitet. Während dieser Zeit war sie umfassend in die Weiterentwicklung der Visual Basic-Sprache und der Benutzeroberfläche von Visual Studio eingebunden. Als Testleiterin wendet Sarika viel Zeit darauf, darüber nachzudenken, wie Entwickler Visual Studio und .NET Framework verwenden, um Client-, Web- oder Anwendungen auf Basis anderer Plattformen zu erstellen. Sie tritt auch zu verschiedenen Zeitpunkten während des Produktentwicklungszyklus mit der Community in Kontakt, um Feedback zu bekommen, sich über Fehler und Probleme berichten zu lassen und Kunden neue Versionen zu präsentieren.

Sarikas und Klaus' Hintergrund waren die Voraussetzungen, dieses gut durchdachte Buch schreiben zu können. Sarika hat beinahe ein Jahrzehnt damit verbracht, an der Visual Studio IDE zu arbeiten – ein signifikanter Bereich an Aufwand in diesem Release. Die Visual Studio 2010-Benutzeroberfläche wurde komplett auf Basis der Windows Presentation Foundation umgeschrieben und ermöglicht dem Benutzer eine viel facettenreichere Bedienung von Visual Studio selbst, von den zusätzlichen Erweiterungsmöglichkeiten durch Produkte von Drittherstellern ganz zu schweigen. Sarika teilt ihr Wissen zu diesem Thema im Kapitel 4 »Rundgang durch Visual Studio«. Klaus' Erfahrungen basieren auf vielen Jahren Arbeit als Technologieberater in Visual Basic .NET-Projekten und Visual Basic 6-Migrationen. Er arbeitete mit vielen Entwicklerteams an den unterschiedlichsten Projekten, und weiß daher, welche Konzepte für Entwickler erfahrungsgemäß besondere Herausforderungen beim Erlernen sind. Er verfügt über eine echte Leidenschaft, die brauchbarsten Informationen für seine Leser zusammenzustellen und sie so aufzubereiten, das sie leicht zu verstehen sind. Obwohl das Schreiben eines Buchs ein aufwendiges Unterfangen sein kann, handelt Klaus es mit Enthusiasmus. Er hat einen großen Sinn für Humor, der in seinem Schreibstil immer wieder durchscheint, sodass es Spaß macht, seinen Gedankengängen zu folgen.

Wie es uns Klaus' Oma also raten würde: Probieren Sie das Arbeiten mit diesem Buch aus! Ich bin mir sicher, dass Ihnen die 10. Version von Visual Basic dabei helfen wird, Ihre Software-Entwicklungsprojekte mit größerer Leichtigkeit und viel mehr Produktivität als jemals zuvor angehen zu können.

Lisa Feigenbaum
Community Program Manager
Microsoft Visual Studio
August 2010

Einleitung

Sobald mich jemand fragt, »Sag mal, was machst du eigentlich beruflich«, fange ich oft an, zu stammeln und fühle mich für den Moment richtig unwohl. Das liegt wohl daran, dass ich mein erstes Buch – das Grafikbuch zum Commodore 128 – bereits mit 16 Jahren geschrieben habe, und damals war es uncool, freakig, geekig, nerdig und was-weiß-ich-nicht-noch-ig sich überhaupt mit Computern zu beschäftigen, geschweige denn darüber zu schreiben, wie man sie programmiert. Und wahrscheinlich hat sich das bei mir so sehr eingebrannt, dass mich die richtige Antwort auch noch heute ein wenig in Verlegenheit bringt. »Ich schreibe Fachbücher über Softwareentwicklung« klingt auch irgendwie, hmm, abgehoben, oder?

Naja, mittlerweile ist es ja nicht mehr ganz so schlimm, denn es ist ja nicht nur beim Schreiben geblieben: unsere kleine Firma ActiveDevelop (aber immerhin sind wir schon zu zehnt!) im einzigen Lippstädter Bürohochhaus produziert nicht nur Texte über Softwareentwicklung, sondern *praktiziert* auch Softwareentwicklung. Und hilft anderen Unternehmen, Software zu entwickeln, deren Teams auf den aktuellen Stand der Technik zu bringen, oder auch Software beispielsweise von Visual Basic 6 zu Visual Basic .NET (oder, ungerne und oft unnötig, zu C#) zu migrieren. Oh, genau, da sich hier gerade die Gelegenheit ergibt: Falls Sie sich selbst im deutschen und/oder englischsprachigen Raum befinden, und kompetenten Support rund um .NET, Schulungen und Projekt-Coaching in den Sprachen Visual Basic und C#, Unterstützung bei Lokalisierungen und ein enorm motiviertes Team mit guten Verbindungen benötigen (hey, meine Co-Autorin arbeitet immerhin im Visual Studio Team! ;-), dann wissen Sie ja, wo Sie uns finden können: *info@activedevelop.de.*

Da das Schreiben über das Thema Softwareentwicklung aber von jeher eine Art Drang gewesen ist, gibt es neben diesem Job auch immer mal wieder ein neues Buch; meistens natürlich, wenn sich bei Microsoft in Sachen neuen Produkten und Versionen etwas tut. Anders ist es wohl nicht zu erklären, dass das Buch, das Sie gerade in den Händen halten, das ca. 30ste ist, das ich geschrieben oder an dem ich mitgeschrieben habe. Und obwohl ich es immer noch spannend wie zu 64er-Zeiten finde, mich in neue Technologien hineinzuarbeiten, und ich heutzutage immer öfters die Möglichkeit dazu bekomme, schon früh Beta-Versionen der neusten Microsoft Software zu erhalten, ist es mit dem Schreiben doch ein wenig – sagen wir mal – routinierter und »normaler« geworden. Jedenfalls bis zum letzten Buch, dem Visual Basic 2008 Entwicklerbuch.

Mit diesem Buch erlebe ich allerdings als Fachbuchautor eine Art zweiten Frühling, denn ich habe mir beim Schreiben einen Wunsch erfüllt, von dem ich als Jugendlicher schon geträumt habe. In meiner Vorstellung wollte ich unbedingt ein Programmierbuch über eine Microsoft Programmiersprache in der Stadt Microsofts schreiben, in Seattle. Und genau das tat ich. Ich flog die rund 9000 Kilometer von Frankfurt über Vancouver nach Seattle, wo ich fast 4 Wochen blieb und konnte große Teile sozusagen am »Originalschauplatz« schreiben: das meiste in einem Appartement in Belletown, aber ich schrieb auch im Japanischen Restaurant Blue Fin im Northgate Mall (sehr empfehlenswert, Flat-Rate-Sushi in sehr großer Auswahl!), auf dem Boing Air Field (nein, nicht direkt *auf* dem Air Field), in den hunderten Starbucks in und rund um Seattle, am Pear mit Blick auf den Pudget Sound, am und im Pike Place Market, in Pubs mit Blick auf den Washington Lake, in der Cafeteria zwischen Building 41 und 42 auf dem Microsoft Campus, auf den Bänken vor Building 41 – in dem die Compiler Teams und meine Co-Autorin Sarika übrigens zuhause sind –

ich schrieb überall. Einmal fuhr ich sogar auf eine Halbinsel, die Visual Studio 2005 den Codenamen gab: Whidbey. Und auch dort schrieb ich. Und es machte einen riesen Spaß, zumal es für mich eine außerordentliche Ehre war und ist, diese vielen kompetenten Leute bei Microsoft kennenzulernen, die mir obendrein und trotz ihrer eigenen Arbeitslast jederzeit bei Fragen weitergeholfen haben. Ich bin immer noch unfassbar beeindruckt, wenn ich daran zurückdenke.

Mit Sarika Calla, der Co-Autorin dieses Buchs, hab ich meinen Ursprungstraum sogar noch toppen können, denn jemanden vom Visual Studio Team an Board zu haben, habe ich mir in meinen kühnsten Träumen nicht vorstellen können. Sarika hat sich unter anderem um die in Visual Studio völlig überarbeitete Benutzeroberfläche gekümmert, und das nicht nur in diesem Buch! Das ist auch der Grund, weswegen es wohl keinen Menschen auf dieser Welt gibt, der das besser könnte als sie: Sarika war für Visual Studio 2010 Testleiterin der neuen WPF-unterstützten Benutzeroberfläche. Und jetzt setzen wir noch einen drauf: Wenn alles nach Plan verläuft, wird es Sarikas Kapitel, das natürlich in Englisch entstand, nicht nur in Deutsch, sondern meine Kapitel auch auf Englisch geben. So gehen Verlag und Autoren davon aus, dieses Buch noch im November/Dezember 2010 auch in der englischen Fassung vorlegen zu können. Ich fand und finde das alles so cool und spannend, dass ich Ihnen das am Anfang dieses Buchs erstmal unbedingt erzählen wollte. Ach so, und falls wir uns mal persönlich treffen sollten: Sie wissen ja jetzt, was ich beruflich mache – fragen Sie mich lieber nicht! ☺

Danksagungen

von Sarika Calla:

Beim Schreiben dieses Buchs haben mir die verschiedensten Leute geholfen. Ich möchte mich bei Manish Jayaswal (Microsoft) und meinem Schwiegervater Mr. R. K. Purohit für die Unterstützung beim Lektorieren des Originalmanuskriptes bedanken. Ich bin Klaus Löffelmann, Co-Autor dieses Buchs, zu tiefem Dank verpflichtet, sowie Lisa Feigenbaum, die mir vom ersten Konzept bis zur letzten Zusammenfassung mit Rat und Tat zur Seite stand. Schließlich und nicht zuletzt gilt mein aufrichtiger Dank meinem Mann Bhanu, unseren Kindern Shubham und Soham, meinen Eltern (Dr. S. K. Calla & Dr. Sudha Calla), meiner Schwiegermutter (Mrs. Sharda Purohit), meinen Verwandten Surabhi, Nitesh, Dr.Veena, Dr.Rajesh, Yogi und meinen Freunden für ihre großartige Unterstützung und Ermutigungen.

von Klaus Löffelmann:

Als allererstes möchte ich mich ganz herzlich bei Lisa Feigenbaum bedanken. Sie hat einen enormen Zeitaufwand in das Konzept dieses Buchs gesteckt, und mir nicht nur geholfen, die wesentlichen Themen herauszuarbeiten, sondern die englische Version des Konzepts auch sprachlich perfekt zu machen. Lisa ist als Community Managerin auch die erste Ansprechpartnerin für uns MVPs, versorgt uns mit Informationen aus erster Hand, und ich bin mir sicher, im Namen aller MVPs sprechen zu dürfen, wenn ich sage: Wir MVPs sind wirklich froh, dass das so ist – Lisa, you rock!

Als nächstes möchte ich mich bei Sarika bedanken, die sich nicht nur bereit erklärt hat, viele Aspekte Ihres Wissens über die neue Visual Studio Oberfläche in diesem Buch preiszugeben, sondern mir auch in der Zeit des Schreibens mit Rat und Tat zur Seite stand. Neue Inhalte für ein Buch in dieser Größenordnung zu recherchieren, wenn man die Experten, die Visual Basic und Visual Studio entwickelt haben, direkt fragen kann, ist gigantisch! Vielen Dank, dass du mitgeschrieben hast!

Ganz besonders liegt es mir am Herzen, Ramona Leenings, unserer Fachinformatiker-Auszubildenen, meinen tiefen Dank auszudrücken. Ramona hat sich nicht nur rührend darum gekümmert, bestimmt über 90% der Grafiken dieses Buchs neu zu schießen (und es zu Recht streckenweise wie die Pest gehasst hat, es ist nämlich echt kein toller Job!), sondern sie hat auch viele Beispiele überarbeitet, konvertiert und – Achtung – die Texte des Praxisbeispiels in Kapitel 6 zu WPF geschrieben. Mit ihren 19 Lenzen gehört sie schon jetzt zu den Erste-Klasse-Entwicklern, sie hat ein Händchen für Ästhetik und Design, ist sehr ambitioniert, kann toll formulieren und ich vermute, dass die VB- und .NET-Community im Laufe der nächsten Jahre ihren Namen noch oft hören wird. Ramona, ich bin voll Fan!

Ebenfalls bedanken möchte ich mich bei unserem ActiveDevelop-Entwicklungsleiter Andreas Belke, der mich – als unser Datenbankexperte – wie schon beim letzten Mal an vielen Stellen des LINQ-Teils unterstützt hat, und mit dem es stets einen Riesenspaß macht, zu arbeiten. Außerdem ist Andreas, genau wie ich, großer Fan der Pacific Standard Time-Zeitzone, was uns sehr verbindet!

Danke auch an Dr. Marcus Pusch, unserem Robotik-Genie, dass er mir in der Zeit des Schreibens so konsequent den Rücken freigehalten hat. Vielen, vielen Dank dafür, das hat wirklich sehr geholfen, und dein neues Spielzeug hab ich Dir gerade auf deinen Schreibtisch gelegt.

Thomas Irlbeck wurde die wirklich nicht einfache Aufgabe zuteil, das Buch auf korrekten Inhalt und Plausibilität zu prüfen. Er hat diese Aufgabe mit Bravour bewältigt, und ich möchte soweit gehen zu sagen, dass es dieses Buch in dieser Qualität ohne ihn nicht geben würde. Es ist unglaublich, was für ein Auge für Details er hat, und sieht Unstimmigkeiten, die den Autoren noch beim 10. Mal durchlesen entgehen.

Silja und Uta sind unsere routinierten Typografieexpertinnen und haben schon allerhand Bücher für Microsoft Press gesetzt. Besonders viel »Freude« kommt bei den beiden natürlich immer auf, wenn Sie ein Buch von mir setzen müssen, und ihr Engagement, die 1. Druckfahne dieses Buchs pünktlich in der Druckerei zu haben, weiß ich sehr zu schätzen; das gilt gleichermaßen für Kristin Grauthoff, die dieses Buch Korrektur gelesen hat. Ihr drei: Diese entscheidende Woche UND das darauffolgende Wochenende komplett von sehr früh morgens bis abends spät durchzuarbeiten, um den Termin zu halten, fand ich nicht selbstverständlich. Vielen Dank für euren Einsatz!

Und schließlich bleiben noch meine Eltern. Ihr musstet mich in der Zeit des Buchschreibens ertragen, und ich spar mir zu persönliche Worte an dieser Stelle: Ihr bekommt alle drei stattdessen einen dicken Schmatzer auf die Wange.

Support

Es wurden alle Anstrengungen unternommen, um die Korrektheit dieses Buchs und des Begleitinhalts sicherzustellen.

Microsoft Press stellt unter der folgenden Internetadresse eventuell notwendige Korrekturen zu veröffentlichten Büchern zur Verfügung:

http://www.microsoft-press.de/support.asp

Sollten Sie Anmerkungen, Fragen oder Ideen zu diesem Buch haben, senden Sie diese bitte an eine der folgenden Adressen von Microsoft Press:

Postanschrift:
Microsoft Press
Betrifft: Visual Basic 2010 – Das Entwicklerbuch
Konrad-Zuse-Straße 1
85716 Unterschleißheim

E-Mail:
presscd@microsoft.com

Beachten Sie bitte, dass unter der oben angegebenen Adresse kein Produktsupport geleistet wird. Supportinformationen zu .NET Framework, zu Visual Basic .NET oder Visual Studio finden Sie auf der Microsoft-Produktsupportseite unter:

http://www.microsoft.com/germany/support

Teil A

Der Einstieg in Sprache und Entwicklungs-Werkzeuge

In diesem Teil:

Einführung 3
»Symbolischer Allzweckbefehlscode für Anfänger« 7
Einführung in .NET Framework 65
Rundgang durch Visual Studio 75
Einführung in Windows Forms – Designer und Code-Editor am Beispiel 121
Einführung in Windows Presentation Foundation 189
Die essenziellen .NET-Datentypen 255

Kapitel 1

Einführung

In diesem Kapitel:

Einführung 4
Die Begleitdateien zum Buch und das E-Book 6

Einführung

Hanover. Mit einem »N«. Nicht das in Niedersachsen, Deutschland, dem Zuhause der berühmten Computer-Messe CeBIT, nein, gemeint ist das Hanover in New Hampshire, in den Vereinigten Staaten von Amerika. 1769 wird hier das nunmehr neuntälteste College der USA gegründet, das Dartmouth College – ursprünglich übrigens als Schule für die Ureinwohner Amerikas. Keine 200 Jahre später, 1963 um genau zu sein, tun sich die Wissenschaftler John George Kemeny und Thomas Eugene Kurtz zusammen, und sie implementieren zusammen mit einigen Studenten eine Programmiersprache mit dem Ziel, für das damals sündhaft teure Computersystem ihrer Uni ein *Time-Sharing System* (eine Art Mehrbenutzersystem) zu erschaffen, das sogenannte *Dartmouth Time Sharing System*[1] oder kurz DTSS, um Studenten dessen Benutzung für die Entwicklung eigener Programme mit einer neuartigen Programmiersprache zu ermöglichen. Einzelne, kleine Computer gibt es nämlich damals noch nicht, viele Studenten einer Uni müssen um Rechenzeit eines großen aber für heutige Maßstäbe sehr leistungsschwachen Computersystems buhlen. Basis des neuen DTSS ist die brandneue Programmiersprache, die dabei entsteht. Ihr Name: BASIC, ein Apronym.[2] BASIC steht für **B***eginner* **A***ll-purpose* **S***ymbolic* **I***nstruction* **C***ode (*in etwa: *Symbolischer Allzweck-Befehlscode für Anfänger*), und als Wort übersetzt aus dem Englischen bedeutet es *grundlegend*. 1964 wird dieses System produktiv geschaltet und es verbleibt bis 1999 in Betrieb.

Das Konzept geht auf, und BASIC selbst findet nicht nur eine schnelle Verbreitung auf anderen Großrechnersystemen, sondern gewinnt enorm an Popularität. Und so ist es fast eine logische Konsequenz, dass der erste Heimcomputer, der Altair 8800 der Firma MITS,[3] (siehe Bild) ebenfalls über einen BASIC-Dialekt verfügen muss. Obwohl es mit Tiny BASIC[4] bereits 1974 eine BASIC-Implementierung für den Altair 8800 gibt, schafften es Bill Gates und Paul Allen mit Unterstützung von Monte Davidoff, nicht nur ein für die damalige Zeit extrem leistungsfähiges BASIC, das mehr als Tiny BASIC kann, in den kleinen Arbeitsspeicher des Altair zu quetschen, und den MITS-Boss Ed Roberts davon zu überzeugen, diese Sprache mit jedem Altair auszuliefern, sondern verhelfen damit auch ihrer gerade gegründeten Softwarefirma Micro-Soft[5] zum ersten kommerziellen und erfolgreichen Produkt. Aus dem BASIC für den Altair entsteht so das Microsoft BASIC, was zur Grundlage für die eingebauten BASIC-Dialekte der verschiedensten Heimcomputer wurde: Amiga BASIC, Applesoft BASIC, Commodore BASIC, IBM Cassette BASIC, GW-BASIC (*Ghee Whiz!* –

[1] *http://dtss.dartmouth.edu*

[2] Anders als beim Akronym gibt es beim Apronym bereits das gleichlautende Wort mit Beziehung zur Abkürzungsdefinition.

[3] *http://www.virtualaltair.com;* Nachruf von Bill Gates auf *http://tinyurl.com/yemv53r*

[4] *http://en.wikipedia.org/wiki/Tiny_BASIC*

[5] Micro-Soft wurde erst später in Microsoft umbenannt.

in etwa *Potz Blitz!*) oder QBasic sind nur einige der Derivate, die daraus entstanden sind – die älteren unter uns erinnern sich sicherlich noch daran. Haben wir einen wichtigen Dialekt vergessen? Ach ja, vielleicht nicht ganz unwichtig: Visual Basic, den Urururururenkel von Microsoft BASIC.

44 Jahre später sind wir bei Visual Basic 2010, und außer einigen Sprachkonstrukten wie `For/Next` oder `If/Then/Else` hat Visual Basic mit dem ursprünglichen Microsoft BASIC nicht mehr viel gemein. Wobei, so ganz richtig ist das nicht: Sicherlich versteht das BASIC eines Commodore 64 kaum noch etwas von Visual Basic 2010, aber probieren Sie mal aus, ob Sie das Programm der folgenden Grafik in das Hauptmodul einer Visual Basic 2010-Konsolenanwendung eingeben und ausführen können. Einschließlich der Zeilennummern.

Abbildung 1.1 So sah Microsoft BASIC in den 80ern aus: Hier das BASIC V2 auf dem Commodore 64

TIPP Verwenden Sie statt `PRINT;` lediglich `Console.Write` und statt `PRINT` (ohne Semikolon am Ende) verwenden Sie `Console.WriteLine`. Hinter die Zeilennummern setzen Sie einen Doppelpunkt. Falls Sie noch nie eine Konsolenanwendung erstellt haben, lesen Sie erst das nächste Kapitel, und gönnen sich anschließend nochmal den Spaß, dieses Beispiel auszuprobieren!

Sie werden überrascht sein!

Der Aufwand, das Microsoft BASIC aus den 80ern zu erlernen, war sicherlich übersichtlich. Soviel zu lernen, um alles aus Visual Basic 2010 herausholen zu können, bedarf im Gegensatz dazu sicherlich einiges an Willen, viel Zeit und Durchhaltevermögen – und ein geeignetes Buch, das Ihnen dabei hilft! ☺

Wir hoffen jedenfalls, dass unser Buch ausreichend geeignet ist und Ihnen deswegen beim Erlernen von Visual Basic 2010 mit Rat und Tat zur Seite zu stehen kann – auf Ihr Feedback sind wir sehr gespannt!

Was brauchen Sie, um mit diesem Buch zu arbeiten?

Anders als viele der Heimcomputer der 80er Jahre, haben heutige Computer kein aktuelles Visual Basic in ihrem Rom fest verdrahtet. Um *alle* Beispiele in diesem Buch nachzuvollziehen, benötigen Sie daher eine aktuelle Version von Visual Studio 2010 in der Standard oder Professional Edition, und die ist kostenpflichtig. Sie sollten diese Version in Erwägung ziehen, wenn Sie professionelle Softwareentwicklung machen möchten.

Die preiswerte Alternative: die Express Editions

Doch es gibt auch eine sehr viel preiswertere Alternative, die wahrscheinlich die bessere für die Hobbyisten oder die Gelegenheitsentwickler unter Ihnen darstellt. Microsoft bietet nämlich mit den so genannten Express Editions abgespeckte Versionen von Visual Studio an, die jeweils nur eine Programmiersprache beinhalten. So gibt es beispielsweise die C# Express Edition, die C++ Express Edition und natürlich auch die Visual Basic Express Edition. Sie können diese Versionen unter dieser Adresse herunterladen: *http://www.microsoft.com/germany/Express/*

Aber um es noch einmal zu wiederholen: Deutlich empfehlenswerter für eine professionelle Softwareentwicklung ist *Visual Studio 2010 Professional*, und deswegen sollten Sie auf lange Zeit gesehen dessen Einsatz in Erwägung ziehen. Die in der 2010er Version komplett überarbeitete Benutzeroberfläche von Visual Studio fördert Ihre Produktivität ungemein, und es wäre wie »Porsche fahren mit angezogener Handbremse«, wenn Sie nur mit dem Framework SDK oder der Express Edition von Visual Basic 2010 größere Entwicklungen durchführen wollten.

Die Begleitdateien zum Buch

Sie können die Begleitdateien unter *http://www.microsoft-press.de/support.asp?s100=&s110=535* oder unter *http://www.activedevelop.de/vb2010.aspx* herunterladen.

Kopieren Sie diese Dateien einfach in ein Verzeichnis Ihrer Wahl, damit Sie die dort enthaltenen Projekte an den Stellen, an denen es sinnvoll erscheint, anpassen, mit ihnen herumspielen und ausprobieren können. Wann immer Beispiele im Buch auftauchen, die zur Verdeutlichung von Zusammenhängen auf längere Codeabschnitte zurückgreifen, sehen Sie einen wie in diesem Beispiel gezeigten Hinweis auf die Begleitdateien sowie die Nennung des Unterverzeichnisses, in dem Sie das für das Beispiel benötigte Projekt finden.

Falls Sie dieses Buch als eBook erworben haben, können Sie die Begleitdateien unter

www.microsoft-press.de/support.asp?s110=535 oder *msp.oreilly.de/support/9783866455351/525* herunterladen.

Kapitel 2

»Symbolischer Allzweck-befehlscode für Anfänger«

In diesem Kapitel:

Visual Studio das erste Mal starten 8
Konsolenanwendungen (Console Applications) 10
Anatomie eines (Visual Basic-)Programms 14
Programmstart mit der Main-Methode 16
Methoden mit und ohne Rückgabewerte 19
Deklaration von Variablen 20
Ausdrücke und Variablendefinitionen 24
Was sind Objekte im Unterschied zu »normalen« Datentypen? 27
Eigenschaften 29
Typliterale zur Typfestlegung von Konstanten 30
Typsicherheit 32
Arrays und Auflistungen 37
Programmcode bedingt ausführen 39
Schleifen 47
Verkürzter Zugriff auf Objekteigenschaften und -methoden mit With ... End With 53
Gültigkeitsbereiche (Scope) von Variablen 54
Die Operatoren += und –= und ihre Verwandten 56
Fehlerbehandlung im Code 58

Merkwürdige Kapitelüberschrift, meinen Sie? Aber warum denn? Denn schließlich ist das die wörtliche Übersetzung von *beginners all-purpose symbolic instruction code*. Und nun nehmen Sie einmal jeden einzelnen Anfangsbuchstaben und setzen ihn zu einem Wort zusammen – heraus kommt: *Basic*. Als es 1964 von John George Kemeny und Thomas Eugene Kurtz am Dartmouth College entwickelt wurde (von den beiden stammt im Übrigen auch die Namensgebung), hatte es mit der Programmiersprache, wie wir sie heute als Visual Basic 2010 kennen, nur wenig, sehr, sehr wenig zu tun. Und von Objektorientierung ist Basic damals so weit entfernt gewesen wie Columbus am Ende seiner berühmten Entdeckungsreise von Hinterindien.

Dennoch gibt es immer noch grundlegende Sprachelemente in Form von grundsätzlichen Vorgehensweisen bei der Variablendeklaration und der Anwendung von Strukturbefehlen, die immer noch sehr »basicesque« in der ursprünglichen Definition von Basic sind, und über diese grundsätzlichen Sprachelemente wird Ihnen dieses Kapitel alles Wissenswerte erzählen.

Und nun verdrehen Sie nicht die Augen, und sagen, »Och, nö, kenn' ich doch alles schon!«, denn: Im ungünstigsten Fall kennen Sie wirklich bereits alles, was Sie hier im Folgenden beschrieben finden, und Sie klopfen sich nach dem Kapitel auf die Schulter, loben sich mit »Jo – ich bin gut, ich mach sofort mit Objektorientierung weiter!« Und dann widmen Sie sich hoch motiviert diesen sehr viel anspruchsvolleren Themen; oder Sie lesen die folgenden Absätze und ertappen sich vielleicht doch beim gelegentlichen »Wie, das geht auch?«-Sagen!

Doch eines sollte auch nicht unerwähnt bleiben: Dieses Kapitel soll kein Einsteigerbuch in Form von ausführlichen Erklärungen des Themas sein, und es fängt sicherlich auch nicht bei Adam und Eva an. Programmieren an sich sollte Ihnen nicht fremd sein, und die folgenden Abschnitte sollen Ihnen die Sprache Visual Basic zu »Rekapitulationszwecken« einerseits in Erinnerung rufen, Ihnen andererseits aber auch Unterschiede zu Basic-Dialekten aufzeigen, mit denen Sie vielleicht bislang gearbeitet haben – und all das in möglichst kompakter Form. Damit scheidet dieses Kapitel aber eigentlich für einen Zweck schon aus: Nämlich Visual Basic ohne Vorkenntnisse von der Pike auf zu lernen.

Visual Studio das erste Mal starten

Heutzutage in Visual Basic zu programmieren bedeutet, dass Sie sicherlich 99,999% der Zeit Ihrer Arbeit in Visual Studio verbringen. Die übrige Zeit sind Sie damit beschäftigt, Codedateien aus anderen Projekten zu suchen und sie in Ihrem aktuellen Projekt einzubinden oder Visual Studio nach einem Absturz neu zu starten, was aber glücklicherweise äußerst selten geworden ist.

Die IDE – die integrierte Entwicklungsumgebung – von Visual Studio 2010 stellt Ihnen die Hilfsmittel in einer Oberfläche zur Verfügung, mit der Sie Ihre Programme gestalten. Das sind nach Wichtigkeit sortiert:

- Der Visual Basic 2010-Compiler,[1] der dann aktiv wird, wenn Sie den Kompilierungsvorgang per Befehl (im Menü *Erstellen* oder entsprechende Symbolleiste)

[1] Kurz zum Hintergrund: Programme, die Sie schreiben, werden vom Compiler in einen Zwischencode, den so genannten MSIL übersetzt, der plattformunabhängig dann zur Laufzeit in Prozessorcode übersetzt wird. Mehr zu diesem Thema erfahren Sie im nächsten Kapitel. Der Vollständigkeit halber: Visual Studio stellt natürlich auch weitere Compiler für C++ oder C# bereit, doch soll uns das in diesem Zusammenhang weniger interessieren. Mit Visual Basic Express startet übrigens ein abgespecktes Visual Studio, das tatsächlich nur den Visual Basic-Compiler enthält. Es steht Ihnen aber natürlich frei, dieses mit Visual C# Express oder C++ Express zu ergänzen. Den Link auf die Express-Downloadseiten finden Sie unter *http://www.microsoft.com/germany/Express/*.

- Der Visual Studio-Editor, der Sie mit Syntaxhervorhebung (Syntax Highlighting), IntelliSense und anderen Highlights bei der Bearbeitung Ihres Programmquellcodes unterstützt
- Verschiedene Designer mit entsprechenden Toolfenstern, die Sie beim Aufbauen von Formularen oder anderen visuellen Objekten unterstützen
- Der Projektmappen-Explorer, der die Codedateien verwaltet und organisiert, aus denen Ihr Projekt besteht

Beim ersten Start von Visual Studio 2010 landen Sie allerdings nicht direkt in dieser integrierten Entwicklungsumgebung, sondern Sie sehen im Vorfeld erst folgenden Dialog:

Abbildung 2.1 Beim ersten Start des Programms bestimmen Sie, wie die Entwicklungsumgebung voreingestellt werden soll

In diesem Dialog bestimmen Sie, mit welchen Voreinstellungen die Entwicklungsumgebung konfiguriert werden soll. Vorschlagsweise wählen Sie an dieser Stelle *Allgemeine Entwicklungseinstellungen* aus.

TIPP *Allgemeine Entwicklungseinstellungen* ist die Einstellung, mit der die meisten Visual Studio 2005/2008- und 2010-Installationen voreingestellt werden. *Visual Basic-Entwicklungseinstellungen* enthalten spezielle Anpassungen, bei denen Fensterlayout, Befehlsmenüs und Tastenkombinationen mehr auf das schnelle Erreichen spezieller Visual Basic-Befehle angepasst sein sollen. Der Dialog zum Anlegen eines neuen Projekts ist beispielsweise nur auf Visual Basic-Projekte angepasst, um nicht zu sagen, beschnitten: Einige Optionen, wie das automatische Anlegen einer Projektmappe, sind schon beim Anlegen eines neuen Projekts automatisch ausgeblendet – Sie haben die Möglichkeit, Projekte namenlos zu erstellen und erst am Ende der Entwicklungssitzung unter einem bestimmten Namen zu speichern. Das gilt auch für die Befehle, die Sie über Pulldown-Menüs abrufen können: »Angepasst« bedeutet, dass viele Funktionen, die auch Visual Basic-Projekte betreffen könnten, einfach ausgeblendet sind.

Probieren Sie die für Sie am besten geeignete Variation aus. Falls Sie mit der hier gewählten Voreinstellung später nicht mehr zufrieden sind, finden Sie in Kapitel 4 die notwendigen Hinweise, wie Sie Einstellungen von Visual Studio zurücksetzen können.

Konsolenanwendungen (Console Applications)

Wenn Sie für Endbenutzer programmieren, dann entwickeln Sie mit großer Wahrscheinlichkeit Programme, die die grafische Benutzeroberfläche von Windows verwenden. Solche Anwendungen nennen sich im .NET-Jargon »Windows Forms-Anwendungen« oder kurz: »WinForms-Anwendungen«. Für den Benutzer ist das sicherlich der zurzeit einfachste Weg, ein Programm zu bedienen – für Lernzwecke für uns Entwickler hingegen ist dieser Projekttyp nicht unbedingt der geeignete, da das aufwändige Drumherum von Windows mit all seinen grafischen Elementen wie Schaltflächen, Fenstern, der Maussteuerung etc. vom Erlernen des jeweiligen Sprachelements ablenkt.

Unter .NET können Sie jedoch auch einen weiteren Projekttyp verwenden – die Älteren unter uns werden sich an ihn noch gerne erinnern –, und auch Administratoren von Servern wird er selbst in der heutigen Zeit kein Unbekannter sein. Diese so genannten Konsolenanwendungen sind Programme, die mit einer sehr minimalistischen Benutzeroberfläche an den Start gehen: Solche Programme werden direkt an der Windows-Eingabeaufforderung gestartet und auch ausschließlich durch die Tastatur gesteuert. Die einzige Schnittstelle zwischen Benutzer und Programm ist eine zeichenorientierte Bildschirmausgabe und die Tastatur.

Für die folgenden Abschnitte werden wir ausschließlich Konsolenanwendungen entwerfen. Sie erlauben – wie eingangs schon erwähnt – für den Entwickler den Blick aufs Wesentliche. Und die folgende Schritt-für-Schritt-Anwendung zeigt, wie Sie eine Konsolenanwendung anlegen, nachdem Sie Visual Studio gestartet haben:

1. Wählen Sie aus dem Menü *Datei* den Befehl *Neu* und weiter den Unterbefehl *Projekt*. Visual Studio zeigt anschließend ein Dialog, wie Sie ihn auch in Abbildung 2.2 erkennen können.
2. Unter *Installierte Vorlagen* öffnen Sie den Zweig *Visual Basic* und wählen *Windows* aus.
3. Im mittleren Fensterbereich wählen Sie *Konsolenanwendung*, etwa wie in Abbildung 2.2 zu sehen.
4. Geben Sie einen Namen für Ihr neues Projekt ein. Falls Sie möchten, dass ein ausdrückliches Projektmappen-Verzeichnis angelegt wird, achten Sie darauf, dass die Option *Verzeichnis für Lösung erstellen* aktiviert ist.

HINWEIS Visual Studio ist dafür ausgelegt, dass Ihre Projekte sehr komplex und umfangreich werden können. So umfangreich, dass bei einem einzigen Projekt die Übersichtlichkeit einerseits verloren gehen kann, andererseits die Aufteilung eines Projekts in Teilprojekte einige dieser Teilprojekte auch für andere Projekte wiederverwendbar macht. Aus diesem Grund legen Sie nicht nur Projekte, sondern in der Regel auch eine Projektmappe an (Visual Basic Express hat hier leicht andere Voreinstellungen, beherrscht aber Projektmappen grundsätzlich auch). In dieser Projektmappe befindet sich im einfachsten Fall also nur ein Projekt – Ihre Windows- oder Konsolenanwendung. Das Projektmappen-Verzeichnis macht dann Sinn, wenn Sie mehrere Projekte in Ihrer Projektmappe erwarten: Das können neben Ihrer eigentlichen Hauptanwendung beispielsweise auch weitere Klassenbibliotheken oder ganz andere Projekttypen wie z. B. Webdienste sein. Das Basisverzeichnis enthält in dem Fall lediglich die »Solution«-Datei – übrigens der englische Name für *Projektmappe* –, und diese trägt die Endung *.sln*.

Abbildung 2.2 Mit diesem Dialog, den Sie mit *Datei/Neu/Projekt* erreichen, erstellen Sie ein neues Projekt für eine Konsolenanwendung

5. Bestimmen Sie mit *Durchsuchen* den *Pfad*, unter dem Sie Ihr Projekt speichern möchten.
6. Klicken Sie anschließend auf *OK*.

Im Anschluss daran befinden Sie sich im Code-Editor von Visual Basic, in dem Sie zwischen den Befehlszeilen `Sub Main` und `End Sub` nun die folgenden Zeilen einfügen, sodass Ihr Ergebnis anschließend wie in Abbildung 2.3 aussehen sollte.

```
Sub Main()

    Dim geburtsjahr As Date
    Dim alter As Integer

    Console.Write("Bitte geben Sie Ihr Geburtsdatum ein (dd.mm.yyyy): ")
    geburtsjahr = CDate(Console.ReadLine())
    alter = (Now.Subtract(geburtsjahr)).Days \ 365
    Console.Write("Sie sind {0} Jahre alt", alter)
    Console.ReadKey()
End Sub
```

Abbildung 2.3 Die IDE von Visual Studio 2010 nach dem Anlegen einer neuen Konsolenanwendung und der Nacharbeitung einiger Codezeilen

Anwendung starten

Wenn Sie diese kleine Anwendung starten, indem Sie entweder F5 drücken, das Startsymbol in der Symbolleiste anklicken (▶) oder aus dem Menü *Debuggen* den Befehl *Debugging starten* wählen, fordert die Anwendung Sie auf, Ihr Geburtsdatum einzugeben. Und hier sehen Sie, wie sehr sich Konsolenanwendungen von Ihren gewohnten Windows-Anwendungen unterscheiden: Die Interaktion mit dem Programm erfolgt lediglich über die Tastatur sowie durch die Anzeige von Text auf dem Bildschirm – Abbildung 2.4 gibt Ihnen einen Eindruck von dieser (unserer ersten!) typischen Konsolenanwendung.

Abbildung 2.4 Eine typische Konsolenanwendung: Die Interaktion mit dem Benutzer erfolgt nur über die Tastatur und über reine Textausgaben

WICHTIG Die Geschwindigkeit, mit der eine Anwendung (ganz gleich ob Konsolenanwendung oder Windows Forms-Anwendung) abläuft, wenn Sie sie auf die gerade beschriebene Weise starten, entspricht lange nicht der, wie sie außerhalb der Visual Studio-Benutzerumgebung erreicht wird. Sie haben nämlich innerhalb der Visual Studio-IDE die Möglichkeit, die Anwendung mithilfe des automatisch angehängten Debuggers auf Fehler zu untersuchen. Dabei können Sie beispielsweise an bestimmten Zeilen Haltepunkte setzen (mit F9, wenn sich die Einfügemarke in der entsprechenden Zeile befindet), um Ihr Programm, wenn es einen Haltepunkt erreicht, automatisch anzuhalten und dann bestimmte Programmzustände zur Laufzeit zu untersuchen und das Programm dann Zeile für Zeile (F11) oder in Prozedurschritten (F10) abzuarbeiten. Dass Ihnen all diese Möglichkeiten zur Verfügung gestellt werden können, kostet Zeit bei der Programmausführung, wobei es keine Rolle spielt, ob Sie sie nun konkret nutzen oder nicht.

Möchten Sie Ihre Anwendung in der Geschwindigkeit laufen sehen, die sie unter normalen Bedingungen erreichen würde, starten Sie sie mit Strg + F5, oder wählen Sie aus dem Menü *Debuggen* den Befehl *Starten ohne Debugging*. Die Debugger-Funktionalität steht Ihnen dann natürlich nicht zur Verfügung.

TIPP Sie können sich übrigens die gesamte Debugging-Funktionalität auch durch eine Symbolleiste zur Verfügung stellen lassen. Dazu klicken Sie einfach mit der rechten Maustaste in der Visual Studio-IDE auf einen freien Bereich innerhalb einer Symbolleiste, um das Kontextmenü zu öffnen. Wählen Sie hier *Debuggen*, um die entsprechende Symbolleiste darzustellen, so wie in Abbildung 2.5 zu sehen.

Abbildung 2.5 Öffnen Sie das Kontextmenü mithilfe der rechten Maustaste über dem freien Bereich einer Symbolleiste, können Sie weitere Symbolleisten anzeigen. Die Symbolleiste Debuggen stellt Ihnen einfachen Zugang zu vielen Debugging-Funktionen bereit.

Anatomie eines (Visual Basic-)Programms

Die ersten Datenträger in der EDV waren nicht magnetischer Natur, also keine Disketten und keine Festplatten, es waren Lochstreifen. Lochstreifen liefen durch einen Lochstreifenleser und anhand der Anordnung der Löcher fanden so die zuvor gestanzten Bits und Bytes ihren Weg zurück in den Computer. Wussten Sie aber, dass die ersten lochstreifenähnlichen Gebilde in der Textilbranche zum Einsatz kamen? Dabei übrigens nicht in Form der ersten frühen Computer und nicht mit der Aufgabe, dort vielleicht Umsätze oder Kundeninformationen zu speichern, sie fanden vielmehr ihren Einsatz in Webstühlen. Auf hintereinander angeordneten Holzplättchen wurden Informationen für die Steuerung des Webstuhls festgehalten.

Und streng genommen entsprach schon diese Vorgehensweise auch der, wie man in der heutigen Zeit festhält, was ein Computer machen soll, und damit der Anatomie eines Programms, wie Sie es auch heute entwickeln:

1. Sie benötigen etwas, das Sie verarbeiten – nämlich die Daten (Wolle).
2. Sie benötigen Vorschriften, wie etwas verarbeitet wird – die Programmanweisungen (das Strickmuster).

Natürlich ist das eine starke Vereinfachung dessen, was Sie an Möglichkeiten haben, Ihre Programme zu gestalten, bzw. es lässt nicht unbedingt darauf schließen, welche unglaublichen und unfassbar vielen Möglichkeiten das sind. Auch unsere erste Anwendung schabt nur leicht die Spitze des Eisbergs an, und deswegen finden Sie auch noch mal eine etwas getunte Version des ersten Beispiels in visueller Form (Abbildung 2.6). Diese soll Ihnen die verschiedenen Aspekte der Anatomie eines typischen Visual Basic-Programms (wenn es sich auch immer noch nur um eine Konsolenanwendung handelt) ein wenig näher bringen.

```
Imports System.IO
Imports System.Data

Module Module1

    Sub Main()

        Dim eingabe As String

        Do
            Console.Write("Bitte geben Sie Ihr Geburtsdatum ein (dd.mm.yyyy): ")
            eingabe = Console.ReadLine

            If eingabe = "uhrzeit" Then
                Console.WriteLine("Beim nächsten Ton ist es: " & Now.ToShortTimeString)
                Beep()
                Exit Do

            ElseIf eingabe = "heute" Then
                Console.WriteLine("Heute ist {0}, der {1}", _
                                  Now.ToString("dddd"), _
                                  Now.ToString("dd. mmmm yyyy"))
                Console.WriteLine()

            Else
                If eingabe IsNot Nothing AndAlso eingabe.Length > 6 Then

                    Try

                        Dim geburtsjahr As Date
                        Dim alter As Integer

                        geburtsjahr = Date.Parse(eingabe)
                        alter = CalcAlter(geburtsjahr)
                        Console.Write("Sie sind {0} Jahre alt", alter)
                    Catch ex As Exception
                        Console.WriteLine("Fehler: Ihre Eingabe hatte " & vbNewLine & _
                                          "vielleicht nicht das richtige Datumsformat!")
                        Console.WriteLine(ex.Message)
                        Console.WriteLine()
                    End Try
                End If
            End If
        Loop While String.IsNullOrEmpty(eingabe)

    End Sub

    ''' <summary>
    ''' Diese Methode berechnet das Alter in Jahren
    ''' aus einem angegebenen Datum
    ''' </summary>
    ''' <param name="geburtsjahr">Das Geburtsdatum, das zur Altersberechnung dient</param>
    ''' <returns>Das Alter in Jahren.</returns>
    ''' <remarks>Keine besonderen Bemerkungen sind erforderlich</remarks>
    Function CalcAlter(ByVal geburtsjahr As Date) As Integer

        Dim retAlter As Integer
        Dim heute As Date = Date.Now
        Dim diffZumGeburtsdatum As TimeSpan

        diffZumGeburtsdatum = heute.Subtract(geburtsjahr)
        retAlter = diffZumGeburtsdatum.Days \ 365
        Return retAlter

    End Function

End Module
```

Importieren von Namespaces
Sonderform "Modul" in Visual Basic - im Prinzip eine Klasse mit ausschließlich statischen Elementen
Start der Hauptmethode "Main"
Variabledeklaration
Variablendefinition (hier: durch Einlesen von Tastatur)
Verteilen einer logischen Programmzeile auf mehrere physische Editorzeilen
Abfrage einer Eigenschaft
Kurzschlussauswertung
Geschachtelte Struktur für bedingte Programm-Ausführung
Try/Catch-Block zum Abfangen möglicher Fehler
Aufruf einer selbstdefinierten Methode mit Rückgabewert
Schleifenende mit Formulierung des Abbruchkriteriums
Ende der Hauptmethode
XML Comment zum Generieren von IntelliSense-Informationen
Methode mit Rückgabewert

Abbildung 2.6 Die Anatomie einer kleinen Visual Basic-Anwendung visualisiert

Diese Abbildung zeigt auch: Es stimmt zwar, dass eine Anwendung zum einen aus Daten und zum anderen aus Programmanweisungen besteht, doch gerade was die Programmanweisungen anbelangt, gibt es offensichtlich eine ganze Reihe unterschiedlicher Strukturen. Doch die sind sicherlich für Sie leichter zu verstehen, wenn Sie das Beispielprogramm auch einmal in Aktion gesehen haben, deswegen: Bevor wir uns mit der internen Funktionsweise des Programms beschäftigen und uns den einzelnen Komponenten der Sprache Basic widmen, werfen wir erst einmal einen Blick auf dessen Bedienung (Abbildung 2.7).

Abbildung 2.7 Die »Professional Edition« unseres Beispiels hat sogar noch ein paar mehr Features in petto als die erste »Light«-Version

Selbsterklärend, oder? Nach dem Start der Anwendung geben Sie entweder ein Datum (am besten ein Geburtsdatum) oder einen der Befehle **heute** oder **uhrzeit** ein. Das Programm reagiert wie in Abbildung 2.7 zu sehen darauf. Falls Sie ein Datum mit ungültigem Format eingeben, wird dieser Fehler erkannt, abgefangen und das Programm zeigt Ihnen einen entsprechenden Hinweis.

Und wie funktioniert diese zugegeben recht einfache Anwendung nun intern? Auch recht einfach, und die schematische Abbildung gibt Ihnen auch einen – zugegeben – etwas alternativen Eindruck, mit welchen Elementen diese Anwendung aufgebaut wird.

Programmstart mit der Main-Methode

Jede Konsolenanwendung startet mit einer bestimmten Methode, die (nicht nur) in Visual Basic den Namen *Main* trägt und (nur in Visual Basic) Anweisungen enthält, die zwischen `Sub` und `End Sub` eingeklammert sind. Eine `Sub` in Visual Basic definiert eine Methode ohne Rückgabewerte, und mindestens diese eine Methode `Main` benötigt eine Konsolenanwendung, damit Windows weiß, wo Ihr Programm startet. Methoden mit Rückgabewerten werden in Visual Basic übrigens mit `Function` definiert, ihr Rückgabewert dann mit `Return` zurückgegeben – der Abschnitt nach diesem Kasten erklärt das ein wenig genauer.

Die Geschichte bedeutender Programmiersprachen … und die Geschichte von Sub Main

Wir begegnen, wenn wir heute in der Geschäftswelt von Programmiersprachen sprechen, eigentlich nur zwei verschiedenen Konzepten: der prozedural-strukturierten sowie der objektorientierten Programmierung[2]. Prozedural bedeutet, man kann, wie es der nächste Abschnitt deutlich macht, Code wiederverwenden bzw. mehrfach benutzen, da er von verschiedenen Punkten innerhalb des Programms aufgerufen werden kann und stets an die Stelle zurückkehrt, von der aus er aufgerufen wurde. Ein anderes Wort für Methode ist übrigens Prozedur, und in einer solchen wird wiederverwendbarer Code platziert, weswegen sich diese Art des konzeptionellen Aufbaus eben prozedurale Programmierung nennt.

Der Terminus strukturierte Programmierung ergibt sich aus bestimmten Strukturkonstrukten, wie zum Beispiel `If … Then … Else`- oder `Switch`-Konstrukten. Und die objektorientierte Programmierung setzt da noch eine Abstraktionsebene drauf, doch das soll uns in diesem schönen grauen Kasten vorerst nicht interessieren. Uns soll hier beschäftigen: Wo nahm alles seinen Anfang?

Angefangen hat allerdings alles zu einer Zeit, zu der Entwickler noch die Prozessoren der damaligen Maschinen sozusagen zu Fuß in direktem Maschinencode programmieren mussten. Einem gewissen Herrn Backus ging das Entwickeln auf diese Weise irgendwann so sehr auf die Nerven, dass er mit der Bitte, ein Hilfsprogramm entwickeln zu dürfen, an seine Vorgesetzten herantrat. Der liebe Herr Backus arbeitete zu dieser Zeit – 1958 übrigens – auf einem 704 Mainframe-Computer, und seine Vorgesetzten waren wie er selbst Angestellte des Unternehmens, das diesen Computer baute: Die *International Business Machines Corporation*, besser bekannt unter ihrem abgekürzten Namen IBM.

Dieses Hilfsprogramm sollte dabei helfen, Computer mit Befehlen zu programmieren, die ein wenig mehr an die menschliche Sprache angelehnt sein sollten und die es darüber hinaus auch erlaubten, direkt mathematische Funktionen auszuwerten und zu berechnen. Der hochtrabende Name dieses Projekts: *The IBM Mathematical **For**mula **Tran**slation System*. Diesen Namen jedoch ständig auszusprechen hätte das Projekt vermutlich erst ein Jahr später fertig werden lassen, sodass John Warner Backus sich kurzerhand für einen Kosenamen aus einer Vermischung einzelner Silben entschied, und geboren war eine der ersten Hochsprachen, nämlich ForTran – natürlich hier nur zur Verdeutlichung der Silbenherkunft mit großem »T« geschrieben.

Alle Hochsprachen dieser Zeit verfolgten das so genannte imperative Konzept, bei dem Befehle nacheinander abgehandelt wurden. Strukturen gab es eigentlich noch gar keine, geschweige denn Prozeduren. Die wenigen Abweichungen vom normalen Programmverlauf ergaben sich, wenn zum Beispiel Sprungziele berechnet wurden – eine `GOTO`-Anweisung gab es immerhin –, oder man eines von drei angebbaren Sprungzielen mit einer `IF`-Anweisung erreichen konnte, je nachdem, ob der zu untersuchende Ausdruck kleiner, gleich oder größer als 0 war. Mit jeder weiteren Entwicklung evolvierten die Programmiersprachen ganz allmählich über verschiedene Fortran- und Algol-Versionen, bis sie schließlich bei CPL (*Combined Program Language*, etwa: *kombinierte Programmiersprache*) landeten, einer Programmiersprache, die versuchte, das Beste aus Fortran und Algol (den Sprachen für wissenschaftliche Anwendungen) sowie Cobol (der Sprache für finanzmathematische Anwendungen) zu vereinen. Dummerweise war das Ergebnis ein Monster, das zwar ideale Voraussetzungen hatte, selbst als Plattform für einen Sprachcompiler zu dienen, aber eben viel zu mächtig und träge war. ▶

[2] Aspektorientierte und dynamische Programmierung, aber auch funktionale Programmierung werden gerade, während diese Zeilen entstehen, immer mehr zu einem Thema; wirklich durchgesetzt haben sie sich aber bislang nicht. Im Auge behalten sollte man diese Programmierparadigmen auf jeden Fall, zumal mit Lambdas viele Aspekte der funktionalen Programmierung bereits Einzug in Visual Basic gehalten haben.

Martin Richards von der Cambridge University nahm sich deshalb CPL an (von der man auch sagt, die Abkürzung sei von *Cambridge Program Language* abgeleitet worden; denn in Cambridge wurde diese Programmiersprache auch entwickelt[3]). Er speckte sie ab und schuf damit die Sprache BCPL, die aber wiederum nur auf Mainframe-Computern richtig einsetzbar war, jetzt aber immerhin als Plattform für die weitere Compilerentwicklung taugte.

```
Übrigens: BCPL war die erste Programmiersprache, die Sprachstrukturen{
        naja, {
         sagen wir:
                  { deren Ansätze }
             }
    }
erstmalig mit geschweiften Klammern kennzeichnete.
```

Ken Thompson und Dennis Ritchie waren es dann, die sich in den berühmten Bell Laboratories so etwa gegen 1969 das Ziel setzten, BCPL tatsächlich auch auf Minicomputern zum Laufen zu bekommen. Da die damaligen Minicomputer natürlich auch Minimalisten in Sachen Hardware waren, war das ein ehrgeiziges Ziel, zu dem es nur einen Weg gab: *Alles* an Elementen musste dazu aus der Sprache verschwinden, was nicht absolut überlebenswichtig war, und die Jungs taten genau das und noch mehr – denn sie machten in ihrem Sparbemühen auch vor dem Namen nicht halt: Aus BCPL wurde so »B«, der – Sie ahnen es vielleicht – Papa von C und Opa von C++ – und letzten Endes auch C#. Übrigens: Im Tutorial zur B-Implementierung von B. W. Kernighan und Murray Hill heißt es gleich auf Seite 1 direkt unterhalb des Abschnitts 2 »Generelles Layout von Programmen«:

```
main( ) {
            statements
        }
newfunc(arg1, arg2) {
            statements
        }
fun3(arg) {
        more statements
          }
```

Und weiter:

»Alle B-Programme bestehen aus einer oder mehreren Funktionen, die den Funktionen oder Unterroutinen eines Fortran-Programms entsprechen beziehungsweise den Prozeduren von PL/I. Bei `Main` [im Beispiel] handelt es sich um eine solche Funktion, und in der Tat müssen alle B-Programme über eine `Main`-Funktion verfügen. Die Ausführung eines Programms beginnt bei der ersten Anweisung von `Main` und endet für gewöhnlich bei der letzten. `Main` sorgt normalerweise dafür, dass weitere Funktionen aufgerufen werden – sowohl die des Programms als auch andere aus Bibliotheken. Wie in [modernem] Fortran besteht mit der Übergabe von Parametern [bzw. Argumenten] eine Möglichkeit der Kommunikation…« ▶

[3] Um ganz genau zu sein: CPL war ein Gemeinschaftsprojekt der Cambridge University und der Computerabteilung der Londoner Universität, was wiederum zu Behauptungen führte, dass die ursprüngliche Abkürzung *Cambridge Programming Language* aus diesem Grund in *Combined Programming Language* abgeändert wurde und weniger aus der Kombination zweier Computersprachwelten.

Und nun wissen Sie nicht nur ein wenig mehr über die Evolution bedeutender Programmiersprachen der Neuzeit, sondern auch um die historische Bedeutung der `Main`-Funktion, die es schon in B vor über 35 Jahren gegeben und deren Name sich als Symbolbezeichner für den Einsprungspunkt zum Start eines Programms bis heute gehalten hat.

Methoden mit und ohne Rückgabewerte

Eine Methode `Main`, wie Sie sie gerade kennen gelernt haben, hat natürlich einen Sonderstatus innerhalb einer Anwendung – denn sie braucht mindestens einen benutzerdefinierten Typ, in dem sie »liegen« kann und definiert nur durch ihren Namen den Startpunkt der Anwendung. Ein benutzerdefinierter Typ ist dabei eigentlich eine Klasse – in Visual Basic gibt es aber auch den Sonderfall einer Klasse, ein Modul. Ein Modul ist im Prinzip nichts weiter als eine Klasse, die über Methoden verfügt, die immer da sind – man nennt diese auch *statische* Methoden.

Methoden sind Einheiten, in die Sie Ihr Programm aufgliedern. So wie das Betriebssystem die `Main`-Methode aufruft, können Sie in Ihren Programmen ebenfalls Methoden zum Aufruf bringen. Das können entweder Ihre selbst geschriebenen Methoden sein oder Methoden, die Sie aus Objekten aufrufen, die Ihnen .NET Framework zur Verfügung stellt.

Methoden kann man weitere Rahmenbedingungen zur Durchführung ihrer Aufgaben mitgeben – muss man aber nicht. Diese Rahmenbedingungen werden *Parameter* oder *Argumente* genannt. Ob eine Methode Parameter entgegennimmt oder nicht, bestimmt die Methode durch ihre *Signatur*. Diese legt fest, welche Parameter von welchem Typ in welcher Reihenfolge die Methode erwartet.

Die Schemaabbildung auf Seite 15 zeigt die Anwendung gleich mehrerer Methoden und wie Methoden im Programm definiert werden. `Console.WriteLine` ruft beispielsweise eine Methode der Klasse `Console` auf und bewirkt, dass das, was an Parametern übergeben wird, in einem bestimmten Format – ebenfalls durch Parameter bestimmt – im Konsolenfenster ausgegeben wird.

Definieren von Methoden ohne Rückgabewerte mit Sub

Methoden werden in Visual Basic auf zwei Arten definiert. Zum einen mit dem Schlüsselwort `Sub`. Das sind dann Methoden, die zwar Parameter entgegennehmen können, aber kein Ergebnis zurückliefern:

```
Sub MethodenName([Parameter1 As Typ1[, Parameter2 As Typ2]])
    [Anweisungen]
    [Exit Sub|Return]
End Sub
```

Wenn Sie die Ausführung einer Methode vorzeitig beenden wollen, können Sie das entweder mit `Exit Sub` oder `Return` erreichen – welche der beiden Anweisungen Sie verwenden, spielt dabei keine Rolle. Dieser Fall macht natürlich nur Sinn, wenn eine bestimmte Bedingung vorliegt, die Sie beispielsweise mithilfe des `If … Then … Else`-Konstrukts abprüfen können (näheres dazu erfahren Sie später im Laufe dieses Kapitels).

Definieren von Methoden mit Rückgabewerten mit Function

Methoden können auch ein Funktionsergebnis zurückliefern. In Visual Basic wird eine solche Methode mit dem Schlüsselwort `Function` definiert:

```
Function MethodenName([Parameter1 As Typ1[, Parameter2 As Typ2]]) as RückgabeTyp
    [Anweisungen]
    [Exit Sub]
    [Anweisungen]
    Return funktionsergebnis
End Sub
```

In der Abbildung auf Seite 15 sehen Sie ein solches Beispiel in der Methode `CalcAlter`; dieser Methode wird ein Geburtsdatum als Parameter übergeben, und sie liefert einen Ganzzahlwert – einen so genannten Integer – an die aufrufende Instanz zurück. Der Typ, den sie zurückliefert, wird durch `As Integer` am Ende der Methodendefinition bestimmt.

Bei Methoden, die Rückgabewerte zurückliefern, kommt `Return` eine andere Bedeutung zu als bei Methoden, die keine zurückliefern. `Return` bestimmt nicht nur, *dass* die Methode an genau der Stelle beendet ist, sondern auch, *welchen* Wert die Methode zurückliefern soll.

Deklaration von Variablen

Unsere Anwendung *deklariert* als erste Zeile ihrer `Main`-Methode eine so genannte Variable, in der sie für die Dauer ihres »Lebens« Daten vorhalten kann. Die Zeile

```
Dim eingabe As String
```

legt fest, dass die Variable `eingabe` Inhalte vom Typ `string` aufnehmen kann.

`String` bedeutet dabei, dass `eingabe` Zeichenketten (engl. *character strings*) oder, mit anderen Worten, beliebigen Text aufnehmen kann. `String` stellt also den Typen dar. Würden Sie als Typ `Integer` angeben, dann könnte die Variable `eingabe` Zahlen zwischen ungefähr -2 Milliarden und +2 Milliarden speichern, aber auch nur ganze Zahlen, keine Brüche. Und mit `Double` als Typ auch Brüche, deren Wertebereich allerdings mit größerer Genauigkeit kleiner wird, weswegen man auch von Fließkommazahlen spricht, da sich das Dezimalkomma verschieben kann.

WICHTIG In .NET (nicht nur in Visual Basic) ist die Typsicherheit übrigens sehr wichtig, denn die Verwendung falscher Typen ist – wenn eine Programmiersprache das zulässt – eine sehr häufige Fehlerquelle. Wenn Sie ein Datum beispielsweise im deutschen Datumsformat – als Zeichenkette – ausschreiben und sortieren, dann ist dabei zum Beispiel der 11.10.2002 größer als der 10.11.2005 – denn die 11 ist größer als die 10. Als Datumswert betrachtet ist das zweite Datum natürlich »größer« als das erste, da es der jüngere Wert ist. Man muss sich das in etwa nach der Anzahl vergangener Zeiteinheiten vorstellen. Bei einem weiter zurückliegenden Datum sind seit dem Jahre 0 weniger Sekunden vergangen als beim heutigen Datum, das einen höheren Sekundenwert aufweist. Ergo ist das heutige Datum »größer«. Das Sortieren als Zeichenkette führt also zu einem völlig anderen Ergebnis als das Sortieren nach Datum – Typsicherheit wird deshalb in jeder .NET-Sprache groß geschrieben.

Die verschiedenen Typen, die Sie in .NET Framework und damit auch in Ihren Visual Basic-Programmen verwenden können, finden Sie übrigens in Kapitel 7 beschrieben.

HINWEIS Dass Variablen in Visual Basic mit dem Schlüsselwort `Dim` »vertypisiert« werden – man nennt diesen Vorgang korrekterweise *Deklarieren von Variablen* –, hat übrigens historische Gründe. In den ersten Basic-Versionen war es ursprünglich möglich, Variablen »einfach so« zu verwenden, ohne zuvor etwas zu Typ und über zukünftige Namenskonsistenz zu definieren. Variablen wurden erst zur Laufzeit des Programms aus dem Kontext ihrer Zuweisung angelegt und typisiert. Das galt aber nicht für so genannte Arrays, Variablenfelder also, die man mithilfe eines oder mehrerer Indexe verwenden konnte, um beispielsweise innerhalb von Schleifen mit Zählvariablen auf die einzelnen Arrayelemente zugreifen zu können (und natürlich auch immer noch kann – denn Arrays sind weiterhin zentraler Bestandteil aller .NET-Sprachen). Hier war es schon damals erforderlich, die Dimensionen und Grenzen eines Arrays (Ober-, Untergrenze, Anzahl Dimensionen) zu definieren, und deswegen leitet sich der Befehl für das Einrichten der Dimensionen eines Arrays bis heute auch für die Deklaration normaler Variablen ab.

Durch die reine Deklaration einer Variablen hat sie allerdings noch keinen Wert bzw., falls es etwas wie einen Grundzustand der Variablen gibt, wurde ihr dabei automatisch dieser Wert zugewiesen – das ist der Wert 0 für alle numerischen Datentypen, `false` für boolesche Variablen und *nichts* für Zeichenketten. Dass »nichts« dabei wirklich ein Wert bzw. ein Variablenzustand sein kann (und der Computer damit philosophisch betrachtet einen Gegensatz in sich selbst schafft, nämlich: »der Zustand *ist* nichts«), sei nur am Rande erwähnt und zunächst einfach hingenommen. Übrigens: Sie finden zu nichts wirklich gute Grundlagen in Wikipedia. Entschuldigung – ich meinte natürlich: Sie finden zu »Nichts« wirklich gute Grundlagen in Wikipedia.

Insgesamt unterscheidet .NET Framework (oder genauer gesagt, das **C**ommon **T**ype **S**ystem, das verschiedene Datentypen .NET Framework-weit reguliert) zwei verschiedene Datentypenarten: Wertetypen und Referenztypen. Ganz essenzielle, direkt in .NET Framework eingebaute und hochoptimierte Wertetypen sind beispielsweise `Integer`, `Long`, `Single`, `Double`, `Decimal`, `Date` und `String`, und da sie einerseits so tief in .NET Framework verankert und andererseits so hochoptimiert sind und auf keinem anderen Typ basieren,[4] werden sie auch *primitive Datentypen* genannt.

Sie benötigen so viele verschiedene Datentypen, um dem Bedürfnis des Speicherns der verschiedenen Arten von Daten gerecht werden zu können – Sie würden schließlich Ihre Hemden auch nicht unbedingt in einer Schublade lagern und Ihre Socken im Schrank aufhängen; das würde funktionieren, wäre aber nicht unbedingt praktisch. Und so gibt es eben eine Anzahl an primitiven Datentypen, die für die unterschiedlichsten Aufgaben geeignet sind:

[4] Rein technisch gesehen ist das nicht ganz richtig – vielleicht haben Sie schon einmal die richtige Aussage gehört, dass alle Datentypen auf `Object` basieren. Konzeptionell stimmt die Aussage aber allemal, denn die primitiven Datentypen erfahren durch die Common Language Runtime eine Sonderbehandlung, auch wenn sie auf `ValueType` und damit auf `Object` basieren.

- `Byte`, `Short`, `Integer`, `Long`, `SByte`, `UShort`, `UInteger`, `ULong`: speichern numerische Werte ohne Nachkommastellen – die Konstante *Pi* ließe sich also mit ihnen nicht mal annähernd speichern, bei »3« wäre Schluss, die »,1415926« etc. würde unter den Tisch fallen. Diese so genannten Integer-Datentypen (*Integer*, engl. für *ganzzahlig*) unterscheiden sich nur im Wertebereich und in der Frage, wie viel Platz sie zum Speichern benötigen. `Long` beispielsweise benötigt 8 Bytes, deckt aber dafür den gigantischen Wertebereich von etwa -9 Trillionen bis +9 Trillionen ab. `Byte` hingegen benötigt nur – wer hätte das gedacht – nur ein Byte, dafür gibt es auch nur Werte zwischen 0 und 255.[5] Auf einem Byte als Größeneinheit basiert dann die ganze Datenspeicherung in der Computerei: Ein `Short` besteht aus 2 Bytes, und damit kann er den Wertebereich von -32768 bis +32767 abdecken. Bei einem `UShort` bedeutet das *U* vor dem Datentypnamen, dass er *unsigned* ist, also kein Vorzeichen aufweist. Damit verschiebt sich der Datenbereich von 0 auf 65535 – negative Werte sind nicht drin bei Integer-Datentypen, die mit »U« beginnen. Der am häufigsten gebrauchte Integer-Datentyp heißt dann auch `Integer`: Er besteht aus 4 Bytes, was 32 Bit entspricht, und 32 Bits kann ein 32-Bit-basierter Prozessor naturgemäß am schnellsten verarbeiten. Wenn Sie in Ihrem Programm zählen müssen, dann geht das deswegen mit Integer-Datentypen meistens an schnellsten.
- `Single` und `Double`: Diese benötigen 4 und 8 Bytes und können auch Nachkommastellen darstellen. Bei den Typen handelt es sich um Fließkommatypen, was bedeutet, dass Genauigkeit auf Kosten von Wertebereichgröße geht. Und: Rundungsfehler sind bei diesen Datentypen an der Tagesordnung, deswegen sollten Sie sie nicht für finanzmathematische Anwendungen verwenden, lieber für die Berechnung von Grafiken oder Ähnlichem, wo es auf extrem hohe Geschwindigkeit, aber nicht auf extrem hohe Genauigkeit ankommt.
- `Decimal`: ist für hohe Genauigkeit besser geeignet. Rundungsfehler, wie sie bei der Umrechnung vom 2er- ins 10er-System auftreten können, sind hier ausgeschlossen, da die Werte intern anders als bei `Single` oder `Double` gespeichert werden. Dafür kann der Prozessor diesen Datentyp nicht direkt berechnen, sodass er viel langsamer als die anderen beiden berechnet wird. Aber für Finanzkrams immer noch schnell genug!
- `Date`: speichert Datumswerte, also Kalenderdaten. Intern ist das eine 8 Byte große Fließkommazahl, wobei die Vorkommastellen das eigentliche Datum (Tag, Monat, Jahr) und die Nachkommastellen die Uhrzeit speichern. Kapitel 7 erzählt genaueres zu diesem Datentyp.
- `Boolean`: ist ganz einfach, entspricht nämlich einem Bit und speichert 0 und 1 und damit *Falsch* und *Wahr*.
- `Char`: speichert ein Zeichen. In der Neuzeit sind pro Zeichen 2 Byte fällig, denn mit 256 verschiedenen Zeichen kommen wir schon längst nicht mehr klar.
- `String`: speichert Zeichenketten, also Texte. Streng genommen sind Strings so genannte *Arrays* (Felder) von `Char`-Werten.

---

[5] Dass dieser Wert für Normalsterbliche so willkürlich »krumm« erscheint, hat historische Gründe. Die kleinste Informationseinheit, die ein Computer verarbeiten kann, ist ein Bit: Ein Strom fließt, oder er fließt eben nicht, das entspricht *Wahr* oder *Falsch* oder *0* oder *1*. Nur mit 0 und 1 ist jedoch noch kein Blumentopf zu gewinnen, und so – erzählt man – fügte der Herr Werner Buchholz während der Designphase eines der ersten IBM-Großrechner ein paar Bits (zunächst 6) hintereinander und machte damit aus einem Biss*chen* (Bit) einen *Biss* (Bite). Da Bite und Bit jedoch sehr oft verwechselt wurden, wurde im Laufe der Zeit aus *Bite* das *Byte*. Und da 6 Bits wertemäßig oftmals auch in den frühen Jahren einen nicht ausreichend großen Wertebereich abdeckten (2^6 =64, das entspricht dem Wertebereich von 0-63) setzten sich 8 Bits im Laufe der Zeit als Basisgrößeneinheit für ein Byte durch.

Und neben den primitiven Wertetypen gibt es dann noch die »normalen« Wertetypen, die aus primitiven Datentypen zusammengesetzt werden. Der Typ `Size` beispielsweise besteht aus 2 Integer-Werten und stellt Methoden und Eigenschaften bereit, um die Größe von Objekten zu bestimmen und damit zu rechnen. `Location`, als weiteres Beispiel, dient zum Speichern von Koordinaten.

Die eigentliche Definition einer Variablen erfolgt durch die Zuweisung eines Ausdrucks – der nächste Abschnitt »Nullables« weiß mehr darüber.

Nullables

Alle Wertetypen können auch als so genannte *Nullables* deklariert werden. Ein Nullable zeichnet sich dadurch aus, dass er auch *keinen* Wert gespeichert haben kann, was bei einem Wertetyp standardmäßig nicht möglich ist.

Ein Beispiel: Wenn Sie folgende Variable deklarieren, können Sie sie direkt verwenden – das liegt in der Natur von Wertetypen; Sie brauchen ihnen in Visual Basic keinen initialen Wert zuzuweisen:

```
Dim count As Integer ' Count ist 0
count = count + 1    ' Geht, count ist jetzt 1
```

Den Zustand, dass `count` im Beispiel kein Wert zugewiesen wurde, kann der Integer-Wertetyp nicht unterscheiden. Erst, wenn eine Variable als *Nullable* von einem bestimmten Typ definiert wurde, ist das möglich, und das geschieht, indem man ihrem Typnamen (`Integer` im Beispiel) ein Typliteral hinten anstellt – das Fragezeichen:

```
Dim nullableCount As Integer? ' Count ist jetzt Nothing
nullablecount = 0             ' Initialwert zuweisen.
nullablecount = nullablecount + 1             ' Geht, count ist jetzt 1
```

Das eigentlich Interessante ist, dass Sie mit der Eigenschaft `HasValue` einen Nullable-Wertetyp darauf hin überprüfen können, ob er einen Wert aufweist oder nicht, wie das folgende Beispiel zeigt:

```
        Dim nullableCount As Integer?
        nullableCount = 0
        nullableCount = nullableCount + 1

        Dim anotherNullable As Integer?

        Console.WriteLine("Hat nullableCount einen Wert: " & nullableCount.HasValue.ToString)
        Console.WriteLine("Hat anotherNullable einen Wert: " & anotherNullable.HasValue.ToString)
```

Dieser Beispielcode gibt Folgendes im Konsolenfenster aus:

```
Hat nullableCount einen Wert: True
Hat anotherNullable einen Wert: False
```

Sie können den Wert eines Nullables löschen, indem Sie ihm `Nothing` zuweisen:

```
nullableCount = Nothing
```

Nullables lösen *keinen* Fehler aus, wenn Sie versuchen auf ihren Wert zuzugreifen, um beispielsweise damit zu rechnen. In diesem Fall bleibt das Ergebnis »Nichts, unbekannt, ich weiß nicht, was drin steht«, kurz `Nothing`, und man könnte den folgenden Beispielcode so erklären:

```
Dim anotherNullable As Integer?         ' Hat keinen Wert bzw. Wert ist unbekannt.
anotherNullable = anotherNullable + 1   ' und dabei bleibt es.
Console.WriteLine("anotherNullable ist: " & anotherNullable.ToString)
```

Die Ausgabe lautet:

```
anotherNullable ist:
```

Es wird also buchstäblich nichts ausgegeben. Beim Versuch der Addition sieht Visual Basic, dass in `anotherNullable` nichts drin steht, es ist nicht definiert, und Visual Basic interpretiert diesen Zustand mit »ich weiß nicht, was sich in `anotherNullable` befindet«. Und so reagiert es im Weiteren: Wenn es nicht weiß, was in der Variablen drinsteht, und ein Wert auf sie aufaddiert werden soll, dann weiß Visual Basic wieder nicht, was das Ergebnis ist. Und damit ist es wieder `Nothing`.

HINWEIS Nullables finden insbesondere bei Datenbankanwendungen ihren Einsatz. Es ist üblich, Datenbanktabellen zu definieren, bei denen die Spalten einiger Zeilen keine Werte aufweisen. Dabei ist es ein Unterschied, ob also ein Feld einer Tabelle den Wert 0 speichert oder keinen Wert eingetragen hat. Nullables in .NET helfen, solche Zustände widerspiegeln zu können. Ausführliches zum Thema Nullables erfahren Sie in Kapitel 19.

Ausdrücke und Variablendefinitionen

Sowohl für numerische Berechnungen als auch für Datumsberechnungen oder Zeichenkettenoperationen ist es wichtig, die Funktionsweise von Ausdrücken verstanden zu haben. Im Beispiel in der Abbildung auf Seite 15 ist die Zeile

```
Eingabe = Console.ReadLine()
```

ein `String`-Ausdruck. Die Methode `ReadLine` liefert hier eine Zeichenkette zurück (in diesem Fall eine, die der Benutzer über die Tastatur eingegeben hat) und weist das Ergebnis an die Variable `Eingabe` zu.

Ein anderes Beispiel für einen Ausdruck aus dem Beispiel in der Grafik auf Seite 15 ist die statische `Parse`-Methode des `Date`-Datentyps, und mit der Zuweisung des Ausdrucks

```
geburtsjahr = Date.Parse(eingabe)
```

wird die Zeichenkette, die das Datum repräsentieren soll, in einen echten Datumstyp umgewandelt – denn wir erinnern uns: Der richtige Typ an dieser Stelle ist relevant; ein Computer sortiert eine Zeichenkette, die zufälligerweise einen Datumswert darstellt, anders als ein »echtes« Datum (siehe auch Kasten »Wichtig« auf Seite 20). Deswegen müssen wir den `Date`-Datentyp anweisen, die Zeichenkette zu parsen (Zeichen für Zeichen zu durchlaufen und dabei zu analysieren) und zu versuchen, die Zeichen als Datum zu interpretieren und entsprechend umzuwandeln.

Gleichzeitiges Deklarieren und Definieren von Variablen

Deklaration und Definition von Variablen können übrigens in einem Rutsch erfolgen. Anstelle von

```
Dim weihnachten10 As Date
weihnachten10 = #12/24/2010#
```

können Sie also auch Folgendes schreiben:

```
Dim weihnachten10 As Date = #12/24/2010#
```

HINWEIS Der `As`-Part hinter einer Variablendeklaration ist übrigens innerhalb von Prozeduren (zum Beispiel innerhalb von Methoden, die mit `Sub` oder `Function`) definiert werden, ab Visual Basic 2008 nicht mehr erforderlich, wenn Deklaration und Definition in einem Rutsch erfolgen. Sobald die Option *lokaler Typrückschluss* (mehr dazu im entsprechenden Abschnitt »Lokaler Typrückschluss« ab Seite 35), kann der Visual Basic-Compiler bei einer Ausdruckszuweisung während einer Deklaration anhand des Typs des Ausdrucks den Typ für die Deklaration erkennen. Anstelle also Definition und Deklaration einer `String`-Variablen mit

```
Dim sweetZoe as String = "Zoe Dröge"
```

reicht es auch

```
Dim sweetZoe = "Zoe Dröge"
```

zu schreiben. Der Compiler erkennt nämlich, dass der konstante Ausdruck »Zoe Dröge« ein String ist und zieht bei der »Vertypisierung« den entsprechenden Rückschluss.

Komplexe Ausdrücke

Das Ergebnis bzw. der Rückgabewert einer Methode kann natürlich auch als Operand eines Operators dienen und so die Rückgabewerte mehrerer Methoden miteinander in einem komplexen Ausdruck kombinieren. Nur der einfacheren Verständlichkeit halber ist die Methode aus dem Ausgangsbeispiel:

```
    Function CalcAlter(ByVal geburtsjahr As Date) As Integer

        Dim retAlter As Integer
        Dim heute As Date = Date.Now
        Dim diffZumGeburtsdatum As TimeSpan
```

```
        diffZumGeburtsdatum = heute.Subtract(geburtsjahr)
        retAlter = diffZumGeburtsdatum.Days \ 365
        Return retAlter

    End Function
```

so formuliert, dass ein Ergebnis in mehreren Zwischenschritten errechnet wird. Natürlich könnte man den Wert, der sich in der vorletzten Zeile in `retAlter` befindet und der mit Return zurückgegeben wird, nicht nur an einem Stück berechnen, sondern auch zurückgeben. Das sähe dann so aus:

```
    Function CalcAlter(ByVal geburtsjahr As Date) As Integer

        Return Date.Now.Subtract(geburtsjahr).Days \ 365

    End Function
```

Das Ergebnis ist natürlich dasselbe.

Numerische Ausdrücke zu verstehen, fällt nicht sonderlich schwer – in der 10. Klasse sind wir mit ihnen spätestens das erste Mal konfrontiert worden. Zeichenkettenausdrücke verhalten sich da ähnlich. Wenn wir verstehen, dass Deklaration, Ausdrucksberechnung und anschließende Definition von

```
Dim dbl As Double
dbl = 5
dbl = dbl +5 * 10
```

für `dbl` 55 ergeben (Hierarchieregel: »Potenz vor Klammer vor Punkt vor Strich« wird bei der Auswertung natürlich beachtet!), dann können wir auch einfach nachvollziehen, wieso

```
Dim str As String
str = "-> Angela Wördehoff <-"
str = str.Substring(3, 16)
```

»Angela Wördehoff« als Wert für die Variable `str` ergibt, und wieso

```
Dim übermorgen As Date
übermorgen = Date.Now
übermorgen = übermorgen.AddDays(2)
```

immer zwei Tage ab heute ergibt.

Boolesche Ausdrücke

Numerische Ausdrücke, Ausdrücke, die Datumswerte berechnen sowie String-Ausdrücke sind vergleichsweise einfach zu lesen und zu verstehen – zu sehr sind sie an typische Formeln von Kurvendiskussionen angelehnt, die wir alle in der 9. oder 10. Klasse kennen gelernt haben. Wie würden Sie allerdings den folgenden Ausdruck lesen?

```
Dim var = 5 = 5
```

Ist diese Zeile gültig, und falls ja, von welchem Typ ist `var` und welchen Wert trägt `var`?

Zum Ersten: Ja, diese Codezeile ist gültig. Zum Zweiten: Sie definiert eine Variable vom Typ `Boolean`. Variablen dieses Typs haben wie bereits erwähnt keinen außergewöhnlich großen Zahlenbereich, denn sie können nur zwei Zustände wiedergeben: `True` und `False`.

Und jetzt betrachten wir die Formel da oben noch mal ein wenig analytischer: 5 = 5 ist eine wahre Aussage. *Wahr* heißt auf englisch `True`, das Ergebnis des Ausdrucks 5=5 ist also `True`. Da der Gleichheitsoperator (nicht der Zuweisungsoperator, der eine Variable definiert: gleiches Operatorzeichen, anderer Kontext, andere Bedeutung!) immer ein boolesches Ergebnis zurückliefert, wird die Variable `var` aus diesem Grund auch durch lokalen Typrückschluss als `Boolean` definiert, und würden Sie anschließend Ihr Ergebnis ausdrucken, zum Beispiel durch die Anweisung

```
Console.WriteLine(var.ToString)
```

ergäbe das auch den »Wert« `True`.

Boolesche Variablen bzw. boolesche Ausdrücke sind deswegen so wichtig, weil sie als Argumente in Anweisungen für die bedingte Programmabarbeitung oder auch zum Testen eines Schleifenabbruchkriteriums eingesetzt werden. Bei einer `If`-Abfrage zum Beispiel wird der zwischen `If` und `End If` stehende Codeblock dann ausgeführt, wenn das Ergebnis des hinter `If` stehenden booleschen Ausdrucks `True` ist, ansonsten eben nicht. Und gibt es in diesem Konstrukt noch einen `Else`-Zweig, dann wird dieser ausgeführt, wenn das Ergebnis `False` ist – der Abschnitt »If … Then … Else … ElseIf … End If« ab Seite 39 zeigt nochmals detaillierter, wie das funktioniert.

Was sind Objekte im Unterschied zu »normalen« Datentypen?

Objekte sind Dinge. Irgendetwas. Was man anfassen oder sonst wie beschreiben kann. Und das ist beim Programmieren nicht anders. Objekte sind damit eines der abstraktesten Gebilde, neben den Threads vielleicht so das abstrakteste Gebilde, was man bei der Programmierung kennt.

Um zu erklären, was ein Objekt ist, beschreibt man am besten, was es nicht ist. Eine `Integer`-Variable beispielsweise ist kein Objekt. Eine Datumsvariable auch nicht. Eine boolesche Variable nicht. Ein Datentyp namens `Point`, in der eine Position zum Zeichnen festgelegt wird, ist auch kein Objekt. Das Gebilde, auf dem man zeichnet, der Inhalt eines Fensters, einer PictureBox oder ein Druckerkontext, ist hingegen ein Objekt. Ein Pinsel, mit dem gezeichnet wird, ist ein Objekt. Eine Schaltfläche ist ein Objekt. Eine TextBox, in der Sie in einer Windows- oder Webanwendung Texte eingeben, ist auch ein Objekt. Ein ToolTip ist eines, eine TCP/IP-Verbindung kann auch ein Objekt sein, ein Objekt nämlich, das diese Verbindung steuert. Beim String wird es schwierig. Ist eine Zeichenkette ein Objekt? Prinzipiell ja, aber per Definition nicht.

Sie merken: Objekte und primitive Datentypen haben schon etwas gemeinsam. Beide speichern Daten und beide reglementieren den Zugang zu diesen Daten. Aber Objekte sind in der Regel komplexer, sie brauchen mehr Speicherplatz und sie verfügen über mehrere Methoden, um etwas mit diesem Objekt zu machen (eine TextBox fokussieren, eine Verbindung aufbauen, eine Schaltfläche in den Vordergrund holen). Sie verfügen auch über komplexere Eigenschaften (den Text einer TextBox festlegen oder ermitteln, die Puffergröße einer TCP/IP-Verbindung bestimmen oder abfragen, ein Menü ein- und ausschalten und damit seinen »Eingeschaltet«-Zustand festlegen oder ermitteln, die Hintergrundfarbe einer Schaltfläche bestimmen oder abfragen usw.). Und: Objekte können Ereignisse auslösen (eine Schaltfläche wurde *angeklickt*, der Text einer TextBox *geändert*, in einer offenen Verbindung wurden Daten *empfangen* usw.).

Einfache Variablentypen wie `Integer`, `Double`, `Date` oder `Boolean` kann man deswegen direkt nach der Deklarierung verwenden. Die .NET Framework-Infrastruktur – die Common Language Runtime (CLR) um genau zu sein – kümmert sich unter anderem darum, dass bei diesen so genannten primitiven Datentypen der entsprechende Speicherplatz auf dem Prozessorstack[6] reserviert ist. Eine Variable in Ihrem Programm ist also mit einer Speicheradresse auf dem Prozessorstack verbunden.

Komplexere Objekte passen da nicht drauf. Ein einziges vielleicht – aber für Ihr Vorhaben, einen Bilderorganizer zu programmieren, werden Sie wohl mehr als nur ein Objekt gleichzeitig im Speicher halten müssen. Und deswegen muss für diese komplexen Objekte Speicher reserviert werden. Dieser zu reservierende Speicher nennt sich in .NET-Sprech übrigens *Managed Heap* – wörtlich: *verwalteter Haufen*. Managed Heap deswegen, weil Sie sich nicht darum sorgen müssen, dass andere Datenstrukturen den Speicherbereich für Ihre Bilder verletzen können. Und weil Sie sich auch nicht darum kümmern müssen, den Speicherbereich wieder freizugeben, wenn Sie das Objekt nicht mehr benötigen.

Es gilt als Faustregel:

- **Wertetypen** Müssen mit wenig Speicherplatz auskommen, sind aber extrem schnell
- **Referenztypen** Können viel Speicherplatz verwenden, sind aber langsamer (aber immer noch recht schnell)

Ableiten von Objekten und abstrakten Objekten

Objekte unter .NET kennen aber noch eine Besonderheit: Sie bauen aufeinander auf. Das ist mit Objekten im täglichen Leben nicht anders: Ein Gefäß zum Beispiel ist ein abstraktes Objekt, das wir vielleicht generell klassifizieren, aber nicht konkretisieren können. Eine Milchtüte ist sicherlich ein Gefäß, aber ein Gefäß muss nicht zwangsläufig eine Milchtüte sein. Ein Gefäß kann auch eine Flasche oder ein Eimer oder ein Fass sein. Die Schnittmenge bestimmter Eigenschaften macht all diese Objekte zu einem Gefäß, und deswegen kann man sagen, dass sämtliche flüssigkeithaltende Objekte wie Milchtüte, Flasche, Fass oder Eimer von Gefäß abgeleitet sind.

[6] Ein spezieller Speicherbereich zum Ablegen kurzfristiger Informationen, auf den der Prozessor bzw. ein laufender Prozess mithilfe des Prozessors extrem schnell zugreifen kann.

Bei der Programmierung funktioniert das ebenfalls. Sie können die Vorlage für ein Objekt erstellen – so etwas nennt man übrigens Klasse –, aber diese Vorlage nur für weitere Vorlagen verwenden. Und nur die Unterschiede fließen dann in die Ableitungen ein und machen – durch weitere Eigenschaften und Methoden – aus einer abstrakten Klasse *Gefäß* eben die Klasse *Flasche* oder die Klasse *Milchtüte* oder die Klasse *Fass*. Und dann benutzen Sie diese neue Vorlage, um ein konkretes Fass ins Leben zu rufen – es zu instanziieren. Sie instanziieren dann ein Objekt aus einer Klasse.

Auch da gibt es wieder eine Analogie: Förmchen und Sandkuchen. Ein Förmchen ist eine Klasse, das Instanziieren der Sand, den Sie vom Sandhaufen, also vom »Sand Heap«, nehmen. Sie haben nur ein Förmchen, also eine Klasse, können aber durch das Einfüllen des Sandes in das Förmchen unzählige Objekte herstellen. In .NET-Sprech würde man sagen: »Sie reservieren auf dem Managed Heap Speicher, damit eine Klasse in ein Objekt instanziiert werden kann«[7].

Das ist übrigens auch der Grund, weswegen Objekte zum Neuanlegen – zum Instanziieren – das `New`-Schlüsselwort benötigen. Wenn Sie also eine neue Schaltfläche einem Windows-Formular hinzufügen wollen, dann sind die folgenden Zeilen dafür notwendig:

```
'Neues Button-Objekt instanziieren
Dim t As New System.Windows.Forms.Button

'Text-Eigenschaft setzen
t.Text = "Neue Schaltfläche"

'Dem Formular hinzufügen
meinFormular.Controls.Add(t)

'Schaltfläche fokussieren
t.Focus()
```

Klassen und Objekte sind ein so umfangreiches Thema, dass ihnen ein ganzer Buchteil gewidmet ist. Mit dieser ungefähren Vorstellung wird es Ihnen aber zunächst erst mal leichter fallen, in den vielen Beispielen bis dorthin Klassen und Objekte verwenden zu können und sie vor allen Dingen von primitiven Datentypen zu unterscheiden.

Eigenschaften

Objekte, aber auch Werte können Eigenschaften haben. Eigenschaften beschreiben Zustände, die man ermitteln oder verändern kann, im Gegensatz zu Methoden, die eine Aufgabe erfüllen sollen. Doch Sie merken, auch im normalen Sprachgebrauch kann man diese beiden Dinge nicht so wirklich trennen, und so sind die Eigenschaften den Methoden sehr ähnlich: Das Setzen einer Eigenschaft entspricht zunächst dem Aufruf einer Methode ohne Rückgabewerte, das Auslesen einer Eigenschaft einem Funktionsaufruf an eine Methode, die ein Ergebnis zurückliefert.

[7] In »Sandkasten-Sprech« übersetzt würde das heißen: »Sie nehmen Sand aus dem Sandhaufen, damit ein Förmchen daraus einen Sandkuchen bilden kann«.

Die Länge einer Zeichenkette beispielsweise ist eine Eigenschaft. Wenn eine `String`-Variable folgendermaßen deklariert und definiert wurde

```
Dim einName as String = "Angela Wördehoff"
```

dann können Sie über eine entsprechende Eigenschaft die Länge, also die Anzahl der Zeichen, dieses Strings ermitteln:

```
Dim längeDesNamens As Integer = einName.Length
```

Einige Eigenschaften, wie beispielsweise das aktuelle Datum (`Date.Now`) oder die Länge eines Strings (`stringVar.Length`), kann man nur lesen – deshalb nennen sich solche Eigenschaften auch dann Nur-Lesen-Eigenschaften.

Andere Eigenschaften kann man lesen und schreiben – so können Sie beispielsweise durch die `Enabled`-Eigenschaft eines Steuerelements – etwa einer Schaltfläche – bestimmen, ob es im Fenster anwählbar ist oder nicht:

```
einButton.Enabled = True      ' jetzt kann man ihn bedienen
einButton.Enabled = False     ' jetzt nicht mehr.

'Eigenschaft abfragen und reagieren:
If einButton.Enabled Then
   'Hier steht der Code, der ausgeführt werden soll, wenn der Button enabled ist.
End If
```

Typliterale zur Typfestlegung von Konstanten

Typliterale – meiner Meinung nach sollten diese »Typerzwingungsliterale für Konstanten« heißen – dienen dazu, eine Konstante dazu zu zwingen, ein bestimmter Typ zu sein.

Sie haben das bereits in einigen Beispielen kennen gelernt. Wenn Sie einer Variablen eine Zeichenkette zuordnen wollen, definieren Sie sie zum einen vom Typ `String`, zum anderen packen Sie aber das, was Sie zuweisen möchten, in Anführungszeichen:

```
Dim EinText As String = "Steht in Anführungszeichen"
```

Bei einer numerischen Variablen machen Sie das nicht:

```
Dim Pi as Double = 3.1415926
```

Wenn Sie eine numerische Konstante dazu zwingen wollen, eine Zeichenkette zu sein, damit Sie diese, ohne sie umwandeln zu müssen, an eine `String`-Variable zuweisen können, dann hüllen Sie sie wieder in Anführungszeichen ein. Das Anführungszeichen (oder *die* Anführungszeichen in diesem Fall) dienen also dazu, aus einer Zahl eine Zeichenkette zu machen:

```
Dim PiAlsText As String="3.141592657"
```

Natürlich können Sie mit `PiAlsText` nicht rechnen – in diesem Fall ist die Variable `PiAlsText` eine Zeichenkette wie oben `EinText`, in der nur *zufällig* eine Zeichenfolge steht, die wir Menschen auch als Zahl interpretieren könnten. Aus Sicht des Computers gibt es allerdings keinen prinzipiellen Unterschied zwischen

```
"3.1415926"
```

und

```
"Hallo, schönes Wetter heute!"
```

In den .NET-Versionen von Visual Basic (also in allen Versionen seit Visual Basic 2002) beschränken sich Typliterale nicht nur auf Zeichenketten – es gibt sie auch für andere Datentypen. Um beispielsweise eine Datumskonstante an eine Variable vom Typ `Date` zuzuweisen, verwenden Sie das Nummernzeichen (#), etwa wie im folgenden Beispiel:

```
Dim KlausGeburtstag As Date=#07/24/1969#
```

Dabei müssen Sie darauf achten, dass Sie das Datum im amerikanischen Format angeben, also erst den Monat, dann den Tag und schließlich die Jahreszahl.

Neben dem Anführungszeichen bei Strings ist dies das einzige weitere, das eine Konstante quasi einklammert. Andere Typliterale werden der Konstante einfach nachgestellt. In einigen Fällen bestehen diese übrigens nicht nur aus einem Buchstaben sondern aus zweien.

Tabelle 2.1 zeigt, wie Sie Konstanten mit Typliteralen definieren und gibt Beispiele. Sollte es Variablentypzeichen für einen bestimmten Typ der Tabelle geben, finden Sie diese ebenfalls in der Tabelle angegeben.

Typename	Typzeichen	Typliteral	Beispiel
Byte	–	–	`Dim var As Byte = 128`
SByte	–	–	`Dim var As SByte = -5`
Short	–	S	`Dim var As Short = -32700S`
UShort	–	US	`Dim var As UShort = 65000US`
Integer	%	I	`Dim var% = -123I` oder `Dim var As Integer = -123I`
UInteger	–	UI	`Dim var As UInteger = 123UI`
Long	&	L	`Dim var& = -123123L` oder `Dim var As Long = -123123L`
ULong	–	UL	`Dim var As ULong = 123123UL`
Single	!	F	`Dim var! = 123.4F` oder `Dim var As Single = 123.4F`
Double	#	R	`Dim var# = 123.456789R` oder `Dim var As Double = 123.456789R`
Decimal	@	D	`Dim var@ = 123.456789123D` oder `Dim var As Decimal = 123.456789123D`
Boolean	–	–	`Dim var As Boolean = True`
Char	–	C	`Dim var As Char = "A"C` ▶

Typename	Typzeichen	Typliteral	Beispiel
Date	–	#dd/MM/yyyy HH:mm:ss# oder #dd/mm/yyyy hh:mm:ss am/pm#	`Dim var As Date = #12/24/2008 04:30:15 PM#`
Object	–	–	In einer Variable vom Typ `Object` kann jeder beliebige Typ gekapselt (»geboxt«) werden oder mit dieser referenziert werden
String	$	"Zeichenkette"	`Dim var$ = "Zeichenkette"` oder `Dim var As String = "Zeichenkette"`

Tabelle 2.1 Typliterale und Variablentypzeichen der primitiven Datentypen ab Visual Basic 2005

Typsicherheit

In .NET gilt der Grundsatz der Typsicherheit. Typsicher bedeutet, dass Sie unterschiedliche Typen bei Zuweisungen nicht »einfach so« mischen können Die folgende Zeile wird dann erst gar nicht kompiliert, sondern mit einer Fehlermeldung quittiert:

```
Dim einAndererString As String
einAndererString = 1.23
```

Abbildung 2.8
Typsicherheit unter .NET verlangt, dass implizit nur gleiche oder sichere Typen einander zugewiesen werden können

HINWEIS Nicht vorhandene Typsicherheit wie in Visual Basic 6.0 oder Visual Basic for Applications (für die Makroprogrammierung unter Microsoft Word oder Excel beispielsweise) wird in Visual Basic .NET standardmäßig simuliert. Sie können und sollten für alle neuen Projekte dieses Standardverhalten ändern, indem Sie unter *Extras/Optionen* und dann im Zweig *Projekte und Projektmappen* unter *VB Standard* alle Häkchen setzen.

Damit bestimmen Sie die Einstellungen für alle *zukünftigen* Projekte. Um die Einstellung für das *gerade geöffnete* Projekt vorzunehmen, wählen Sie aus dem Menü *Projekte* den vorletzten Menüpunkt *Eigenschaften* und die Registerkarte *Kompilieren*. Stellen Sie die Aufklappfelder aller Kompilierungsoptionen auf *Ein*.

Die Autokorrektur für intelligentes Kompilieren, die Sie öffnen können, wenn Sie auf den kleinen roten Balken in der Unterschlängelung klicken (Abbildung 2.8), zeigt Ihnen, was das Problem ist: Implizit, also ohne weiteres Zutun (oder anders gesagt »von selbst«) können Sie unter .NET im Grunde genommen nur gleiche Typen zuweisen oder Typen, die zwar unterschiedlich sind, bei denen eine implizite Konvertierung also auf jeden Fall sicher ist. Und was heißt »auf jeden Fall sicher«? Sicher ist es, wenn Sie einen `Integer`-Typ einem `Long`-Typ zuweisen:

```
Dim einLong As Long
Dim einInt As Integer = 10000
einLong = einInt
```

In keiner .NET-Sprache würde der Compiler hier was zu meckern haben, denn bei dieser Typkonvertierung kann nichts schief gehen. `Integer` deckt in jedem Fall einen viel kleineren Zahlenbereich als `Long` ab, und deswegen ist hier eine implizite Konvertierung gefahrlos möglich.

Umgekehrt sieht es schon wieder anders aus, wie Abbildung 2.9 zeigt.

Abbildung 2.9
Während »kleinere« Typen sicher in »größere« konvertiert werden können, ist das umgekehrt nicht typsicher und deswegen auch nicht implizit gestattet

Bei einer Konvertierung von `Long` zu `Integer` können Informationen verloren gehen, deswegen stuft .NET diese Art der Konvertierung als nicht typsicher ein. Natürlich können Sie eine derartige Konvertierung vornehmen, nur eben nicht implizit. Sie müssen .NET explizit mitteilen, dass Sie sich sozusagen »der Gefahr bewusst sind«, und entsprechende Aktionen durchführen, um die Konvertierung vornehmen zu können. Die Autokorrektur für intelligentes Kompilieren macht Ihnen auch direkt einen entsprechenden Vorschlag: »Setzen Sie `CInt` (etwa: ***Convert to Integer – in Integer konvertieren***) ein und zeigen Sie dem Compiler damit, dass Sie sich der Konvertierung von `Long` in `Integer` bewusst sind.«

Und vielleicht ahnen Sie jetzt auch schon, wozu die Typliterale in Visual Basic dienen, die Sie im vorherigen Abschnitt kennen gelernt haben. Typsicherheit muss auch für Konstanten gelten. Und damit muss es auch einen Weg geben, eine Konstante dazu zu bringen, ein bestimmter Typ zu sein. Und eben genau das geschieht mit Typliteralen. Funktioniert das Folgende implizit?

```
Dim einChar As Char
Dim einString As String = "Hallo"
einChar = einString
```

Nein. Denn dabei würde »`allo`« auf der Strecke bleiben. Ein `Char`-Typ kann nur ein einzelnes Unicode-Zeichen und keinen ganzen `String` aufnehmen. Auch wenn dieser Codeausschnitt folgendermaßen lauten würde:

```
Dim einChar As Char = "H"
```

`String` ist `String`, und `Char` ist `Char`. Nur in Anführungszeichen definieren Sie eben einen `String`, unabhängig davon, wie viele Zeichen der hat. Und selbst wenn es passen würde (so wie in dieser Zeile), wäre die Typsicherheit dennoch verletzt. Sie müssen aus dem »H« einen `Char`-Typ machen, und das erreichen Sie durch das Hintenanstellen eines Typliterals, wie Abbildung 2.10 zeigt.

Abbildung 2.10 Mit Typliteralen zwingen Sie Konstanten in einen bestimmten Typ

Das gilt natürlich nicht nur für `String` und `Char`, sondern auch für die unterschiedlichen numerischen Datentypen. Der folgende Codeausschnitt zeigt ein paar Beispiele, die demonstrieren, wann der Einsatz von Typliteralen Sinn macht:

```
'Fehler: Von Double nach Decimal geht nicht implizit.
Dim decimal1 As Decimal = 1.2312312312312
'Hier ist es ein Decimal durch das D am Ende:
Dim decimal2 As Decimal = 1.2312312312312D

Dim decimal3 As Decimal = 9223372036854775807   ' OK.
' Überlauf - Ohne Typliteral ist es implizit ein
' Long-Wert, und in diesem Fall liegt er außerhalb seines Wertebereichs:
Dim decimal4 As Decimal = 9223372036854775808
' Mit dem Typliteral "D" wird Decimal als Konstante erzwungen, und es passt:
Dim decimal5 As Decimal = 9223372036854775808D  ' Kein Überlauf.
```

```
'Fehler: Ohne Typliteral ist's wieder implizit ein Long,
'aber außerhalb des Long-Wertebereichs:
Dim ushort1 As ULong = 9223372036854775808
' Mit Typliteral wird Decimal als Konstante erzwungen, und es passt:
Dim ushort2 As Decimal = 9223372036854775808UL  ' Kein Überlauf mehr.
```

Lokaler Typrückschluss

Seit Visual Basic 2008 ist es erlaubt, dass die Typfestlegung von Variablen auch aufgrund ihrer initialen Zuweisungen festgelegt wird. Ganz eindeutig wird das beispielsweise an der folgenden Zuweisung:

```
Dim blnValue = True
```

Wenn Sie einer primitiven Variable den Wert `True` zuweisen und dazu noch Typsicherheit definiert ist, dann muss es sich bei der Variablen einfach um den booleschen Datentyp handeln. Genauso ist das bei

```
Dim strText = "Eine Zeichenkette."
```

`strText` *muss* eine Zeichenkette sein – das bestimmt die Zuweisung. Anders ist es bei numerischen Variablen. Hier muss man wissen, dass durch Zuweisung einer ganzen Zahl an eine bislang noch nicht typbestimmte Variable der `Integer`-Typ definiert wird und durch Zuweisung einer Fließkommazahl der `Double`-Typ. Doch diese Standardtypen von Konstanten gab es vorher schon – letzten Endes bestimmen die Konstanten mit ihren Typliteralen, welchen Typ sie darstellen.

```
Dim einInteger = 100 ' Integer, einfache Fließkommalosezahl definiert Integerkonstante
Dim einShort = 101S  ' Short, weil das Typliteral S eine Short-Konstante bestimmt
Dim einSingle = 101.5F  ' Single, weil das Typliteral F eine Single-Konstante bestimmt
```

WICHTIG Lokaler Typrückschluss funktioniert übrigens nur auf Prozedurebene, nicht auf Klassenebene (deswegen auch die Bezeichnung »lokaler« Typrückschluss).

Gesteuert wird der lokale Rückschluss übrigens durch die Anweisung `Option Infer`, der als Parameter `Off` oder `On` übernimmt (von engl. *Inference*, etwa: *der Rückschluss*). Standardmäßig ist der lokale Typrückschluss eingeschaltet. Sie können ihn also durch die entsprechende Anweisung

```
Option Infer Off
```

direkt am Anfang der Codedatei eben nur für die Klassen und Module dieser Codedatei oder aber global für das ganze Projekt ausschalten (oder eben anschalten). Dazu öffnen Sie das Kontextmenü des Projekts (nicht der Projektmappe!) im Projektmappen-Explorer und wählen den Menüeintrag *Eigenschaften* aus (Abbildung 2.11).

Abbildung 2.11 Öffnen Sie das Kontextmenü des Projekts im Projektmappen-Explorer ...

Wählen Sie anschließend den Menüpunkt *Eigenschaften*, und legen Sie auf der Registerkarte *Kompilieren* im Dialog, den Sie jetzt sehen, die entsprechende Option für `Option Infer` fest (Abbildung 2.12).

Abbildung 2.12 ... und machen Sie die gewünschten Einstellungen auf der Registerkarte Kompilieren

Arrays und Auflistungen

Auf zwei Arten zur Speicherung von Daten ist dieses Kapitel bereits eingegangen:

- Wertetypen und primitive Wertetypen zur Speicherung von schnellen, aber kleinen Datenstrukturen wie Datumswerte, Ganzzahlwerte, Fließkommawerte, Wahr/Falsch-Werte oder auch Positionen, Größenangaben oder Zeichen und Zeichenketten
- Referenztypen zur Speicherung großer Datenmengen, wie etwa Steuerelemente oder Bilder

Beide Typenarten werden innerhalb eines Programms durch Variablen abgesprochen. Doch dabei stellt sich in einigen Fällen ein Problem: Wie muss man vorgehen, wenn man in Abhängigkeit des Werts einer Variablen einen anderen Wert in einer Tabelle nachschlagen möchte?

Angenommen, Sie haben 5 Namen in Variablen gespeichert, was ja erst einmal kein großes Problem darstellt:

```
Dim Name1 As String = "Lisa Feigenbaum"
Dim Name2 As String = "Sarika Calla"
Dim Name3 As String = "Ramona Leenings"
Dim Name4 As String = "Amanda Silver"
Dim Name5 As String = "Tanja Gelo"
```

Jetzt möchten Sie aber auf einen Namen zugreifen, dessen Nummer in einer Variablen gespeichert ist – so wie in Abbildung 2.13 zu sehen.

```
Sub ArrayTest()
    Dim Name1 As String = "Lisa Feigenbaum"
    Dim Name2 As String = "Sarika Calla"
    Dim Name3 As String = "Ramona Leenings"
    Dim Name4 As String = "Amanda Silver"
    Dim Name5 As String = "Tanja Gelo"

    Dim NameNo As Integer = 3
    Console.WriteLine(Name NameNo)
End Sub
```

Komma, ")" oder eine gültige Ausdrucksfortsetzung erwartet.

Abbildung 2.13 Möchte man mit einer Variable eine andere quasi wie in einer Liste nachschlagen, kommt man mit normalen Variablennamen nicht weiter

Die Lösung zum Problem sind hier so genannte Arrays. Ein Array speichert nicht nur einen Wert unter einem Namen, sondern so viele Werte, wie Sie angeben. Mit einem Indexwert können Sie dann bestimmen, auf welchen der Werte Sie zugreifen möchten:

```
Dim Namen(0 To 4) As String
Namen(0) = "Lisa Feigenbaum"
Namen(1) = "Sarika Calla"
Namen(2) = "Ramona Leenings"
Namen(3) = "Beth Messi"
Namen(4) = "Tanja Gelo"

Dim NameNo As Integer = 3
Console.WriteLine(Namen(NameNo))
```

WICHTIG Achten Sie darauf, dass – anders als zum Beispiel in VBA oder damals in Visual Basic 6.0 – Arrays in .NET immer 0-basiert sind, das erste Element eines Arrays also die Nummer 0 hat. Sie geben in Visual Basic bei der Dimensionierung eines Arrays immer die Obergrenze und nicht die Anzahl der Elemente an (wie es beispielsweise in C# gemacht wird). Sie können bei der Dimensionierung auch auf die Kurzform zurückgreifen. Die in Fettschrift gesetzte Zeile im oben stehenden Listing könnte also auch folgendermaßen lauten:

```
Dim Namen(4) As String
```

TIPP Arrays können auch mit so genannten Array-Initialisierern definiert werden. Dabei werden die Elemente direkt mit angegeben, und das spart Tipparbeit:

```
Dim Namen() As String = {"Lisa Feigenbaum",
                         "Sarika Calla",
                         "Ramona Leenings",
                         "Beth Messi",
                         "Tanja Gelo"}

Dim NameNo As Integer = 3
Console.WriteLine(Namen(NameNo))
```

Die Obergrenze dürfen Sie dabei nicht bei der Definition des Arrays spezifizieren, da sie sich aus der Anzahl der Elemente ergibt, die Sie in geschweiften Klammern übergeben.

Falls Sie zum Zeitpunkt der ersten Verwendung noch nicht genau sagen können, wie viele Elemente ein Array haben soll, dann empfiehlt es sich, eine Auflistung zu verwenden. Auflistungen fügen Sie mithilfe der `Add`-Methode neue Elemente hinzu, etwa wie im folgenden Beispiel zu sehen:

```
Dim AndereNamen As New List(Of String)
AndereNamen.Add("Lisa Feigenbaum")
AndereNamen.Add("Sarika Calla")
AndereNamen.Add("Ramona Leenings")
AndereNamen.Add("Beth Messi")
AndereNamen.Add("Tanja Gelo")

'Auf die dynamische Liste können Sie,
'wie bei Arrays, auch per Index zugreifen:
Dim NameNo As Integer = 3
Console.WriteLine(AndereNamen(NameNo))
```

Diese Ausführungen sollen zunächst über Arrays und Auflistungen genügen. Ihnen ist zum späteren Zeitpunkt ein ganzes Kapitel gewidmet, und im Kontext von Schleifen und an anderer Stelle werden sie Ihnen ebenfalls noch begegnen.

Programmcode bedingt ausführen

Der `Boolean`-Datentyp wird in der Regel bei der Auswertung von Entscheidungen benötigt. Mit ihm können Sie in Abhängigkeit seines Werts steuern, ob Programmcode ausgeführt wird oder nicht. Dafür verwenden Sie die `If`-, `Case- [Is]`, `While`- oder `Until`-Anweisungen.

Die `IIf`-Funktion steuert zwar nicht den Programmablauf, sollte aber auf Grund ihrer Ähnlichkeit zur `If`-Anweisung ebenfalls in diesem Zusammenhang erwähnt werden. Sie liefert ein Funktionsergebnis auf Basis des booleschen Werts, der ihr übergeben wird. Ist der übergebene Wert `True`, wird das erste mögliche Funktionsergebnis zurückgeliefert, ist er `False`, das zweite.

If ... Then ... Else ... ElseIf ... End If

Die `If`-Anweisung haben Sie höchstwahrscheinlich schon Hunderte Male angewendet und wissen deshalb aus dem Effeff, wie sie funktioniert. Der Vollständigkeit halber möchte ich sie dennoch ein wenig genauer unter die Lupe nehmen:

In der einfachsten Form wird bei der `If`-Anweisung der Code ausgeführt, der zwischen `If` und `End If` positioniert wird, wenn der hinter `If` stehende boolesche Ausdruck `True` wird. Obwohl Basic seit Jahren im Einsatz ist, gibt es immer noch Entwickler, die das Konzept von Vergleichen mit booleschen Ausdrücken nicht verinnerlicht und Schwierigkeiten beim Verständnis folgender Konstrukte haben:

```
locBoolean = True
If locBoolean Then
      'Wird nur ausgeführt, wenn locBoolean True ist.
End If
```

Einige Entwickler verstehen nicht, wieso hier nicht der folgende Alternativausdruck zum Einsatz kommen muss:

```
locBoolean = True
If locBoolean = True Then
      'Wird nur ausgeführt, wenn locBoolean True ist.
End If
```

Tatsache ist, dass der Ausdruck

```
locBoolean = True
```

in dieser Bezeichnung keine Besonderheit der `If`-Anweisung ist, sondern im Prinzip eine ganz normale Funktion. Wenn `locBoolean` den Wert `True` hat, ist der gesamte Ausdruck natürlich ebenfalls wieder `True`. Die Anweisung `If` macht nichts weiter, als den dahinter stehenden booleschen Wert zu untersuchen und die nachstehenden Anweisungen nur dann auszuführen, wenn der Wert `True` war. Aus diesem Grund braucht der Wert nicht noch zusätzlich durch Eingreifen des Programmierers überprüft zu werden; das wäre redundant. Denn das Gleichheitszeichen ist hier ja der Vergleichsoperator! Wenn man `locBoolean` durch seinen augenblicklichen Wert ersetzt, lautet die Überprüfung:

```
If True = True Then ' (Wenn wahr wahr ist, dann)
```

was man ziemlich selbstsicher durch `If True Then …` oder `If locBoolean Then …` ersetzen kann (wenn es wahr ist, dann …).

Es ist in Basic (nicht nur in Visual Basic) aber in der Tat verwirrend, dass Zuweisungsoperator und Vergleichsoperator mit denselben Zeichen angewandt werden. Der Ausdruck

```
Dim locBoolean = "Klaus" = "Uwe"
```

ist aber natürlich gültig. Das erste Gleichheitszeichen fungiert hier als Zuweisungsoperator, das zweite als boolescher Vergleichsoperator. In diesem Beispiel nimmt `locBoolean` den Wert `False` an, weil die Zeichenkette »Klaus« nicht »Uwe« entspricht. Der Vergleichsoperator hat vor dem Zuweisungsoperator die höhere Priorität. Anderenfalls käme es bei diesem Beispiel auch zu einem Typkonvertierungsfehler.

Aber auch der andere Weg ist nicht wirklich überzeugend, C++ benutzt beispielsweise `=` für die Zuweisung, `==` für den Vergleich. Das ist aber alles andere als intuitiv. Allein 5% – 8% aller Fehler in C++-Programmen gehen nämlich auf diese Verwechselung zurück. Und das ist viel häufiger als die Anzahl der Fehler, die durch die Falschbenutzung des `=`-Zeichens in Visual Basic passieren.

Dem `If`-Codeblock kann auch ein `Else`-Codeblock folgen, der ausgeführt wird, wenn der boolesche Ausdruck hinter `If` den Wert `False` annahm. Darüber hinaus können Sie mit `ElseIf` weitere Auswertungen in das `If`-Konstrukt einschieben. Der Codeblock hinter dem letzten `Else`-Codeblock wird, falls vorhanden, nur dann ausgeführt, wenn keine der Bedingungen der einzelnen `If`- bzw. `ElseIf`-Sektionen `True` ergaben. Ein Beispiel:

```
locString1 = "Klau's, und lass Dich nicht erwischen"
locString2 = "Klaus*"
locBoolean = (locString1 Like locString2)   ' ergibt False; Like prüft auf Ähnlichkeit (s.u.)

If locBoolean Then
    'Schachteln geht natürlich auch:
    If locString2 = "Klaus" Then
        Console.WriteLine("Namen gefunden!")
    Else
        Console.WriteLine("Keinen Namen gefunden!")
    End If
ElseIf Now = #12:00:00 PM# Then
    Console.WriteLine("Mittag!")
ElseIf Now = #12:00:00 AM# Then
    Console.WriteLine("So spät noch auf?")
Else
    Console.WriteLine("Es ist irgendwann sonst oder locString1 war nicht wie locString1...")
End If
```

Logische Operatoren And, Or, Xor, Not

Visual Basic kennt eine Reihe von logischen Operatoren, die sowohl auf numerische als auch auf boolesche Datentypen angewendet werden können. Letztere sind insbesondere von Interesse, weil sich damit zum Beispiel im `If`-Konstrukt fast natürlichsprachliche Bedingungen für Programmabläufe formulieren lassen.

Geht man davon aus, dass die beiden numerischen Werte, die vom booleschen Typ dargestellt werden können, nämlich 0 und 1, *Wahr* und *Falsch* entsprechen, lassen sich damit Verknüpfungen formulieren, in etwa wie: »Wenn Aussage1 *und* Aussage2 wahr sind, dann...« oder »Wenn entweder Aussage1 *oder* Aussage2 wahr ist, dann«.

Die wichtigsten sind:

- `And`: führt eine logische Und-Verknüpfung durch. Beide Aussagen müssen wahr sein, damit der gesamte Ausdruck wahr ist:

Aussage 1	Aussage 2	Ergebnis
Falsch (0)	Falsch (0)	Falsch (0)
Wahr (1)	Falsch (0)	Falsch (0)
Falsch (0)	Wahr (1)	Falsch (0)
Wahr (1)	Wahr (1)	Wahr (1)

- `Or`: Führt eine logische Oder-Verknüpfung durch: Mindestens eine der Aussagen muss wahr sein, damit der gesamte Ausdruck wahr ist:

Aussage 1	Aussage 2	Ergebnis
Falsch (0)	Falsch (0)	Falsch (0)
Wahr (1)	Falsch (0)	Wahr (1)
Falsch (0)	Wahr (1)	Wahr (1)
Wahr (1)	Wahr (1)	Wahr (1)

- `Xor`: Führt eine logische exklusive Oder-Verknüpfung durch. Nur wenn genau *eine* der beiden Aussagen wahr ist, ist der Ausdruck wahr, sonst ist er falsch:

Aussage 1	Aussage 2	Ergebnis
Falsch (0)	Falsch (0)	Falsch (0)
Wahr (1)	Falsch (0)	Wahr (1)
Falsch (0)	Wahr (1)	Wahr (1)
Wahr (1)	Wahr (1)	Falsch (0)

- `Not`: Negiert die Aussage. Wenn Sie wahr ist, wird sie falsch, war sie falsch, wird sie wahr:

Aussage	Ergebnis
Falsch (0)	Wahr (1)
Wahr (1)	Falsch (1)

Beispiele:

Prüfen, ob sich bei einer Menüauswahl einer Konsolenanwendung ein eingegebenes Zeichen in einem bestimmten Bereich befunden hat:

```
Sub ConditionCheck()

    'Optionen ausgeben und Zeichen von der Tastatur lesen.
    Console.Write("Welche Funktion möchten Sie ausführen (1-9, 0 oder 'ende' zum Beenden): ")
    Dim eingabe = Console.ReadLine

    'Bei Eingabe von "0" oder "ende",
    If eingabe = "0" Or eingabe.ToUpper = "ENDE" Then
        'Methode beenden.
        Exit Sub
    End If

    'Wenn das gedrückte Zeichen (String-Länge=1) größergleich "1" und kleinergleich "9" ist...
    If eingabe.Length = 1 And eingabe >= "1" And eingabe <= "9" Then
        '...dann war es eine gültige Auswahl, ...
        Console.WriteLine("Diese Funktion ist möglich!")
        '...anderenfalls...
    Else
        '...nicht.
        Console.WriteLine("Sie haben eine falsche Auswahl getroffen.")
    End If
End Sub
```

HINWEIS Die logischen Operatoren lassen sich nicht nur auf boolesche Werte, sondern auch auf andere numerische Datentypen anwenden. Dabei werden die Bits, aus denen sich die Werte intern zusammensetzen, miteinander verknüpft. Beispiel:

```
13 And 7
```

ergibt 5, denn im Dualsystem geschrieben wird folgende Operation durchgeführt:

```
    1101 (13)
And 0111 (07)
-------------
    0101 (05)
```

Jedes Bit des Ausgangswerts wird in diesem Fall mit jedem Bit des zweiten Werts mit `And` verknüpft – und daraus resultiert dann anschließend das Ergebnis.

Vergleichsoperatoren, die boolesche Ergebnisse zurückliefern

Visual Basic kennt die folgenden so genannten Vergleichsoperatoren, die zwei Ausdrücke miteinander vergleichen und ein boolesches Ergebnis zurückliefern:

- `Ausdruck1 = Ausdruck2`: Prüft auf gleich; liefert `True` zurück, wenn beide Ausdrücke gleich sind
- `Ausdruck1 > Ausdruck2`: Prüft auf größer; liefert `True` zurück, wenn `Ausdruck1` größer als `Ausdruck2` ist
- `Ausdruck1 < Ausdruck2`: Prüft auf kleiner; liefert `True` zurück, wenn `Ausdruck1` kleiner als `Ausdruck2` ist
- `Ausdruck1 >= Ausdruck2`: Prüft auf größer/gleich; liefert `True` zurück, wenn `Ausdruck1` größer oder gleich `Ausdruck2` ist
- `Ausdruck1 <= Ausdruck2`: Prüft auf kleiner/gleich; liefert `True` zurück, wenn `Ausdruck1` kleiner oder gleich `Ausdruck2` ist
- `Ausdruck1 <> Ausdruck2`: Prüft auf ungleich; liefert `True`, wenn `Ausdruck1` nicht `Ausdruck2` entspricht
- `Ausdruck1 Is [Ausdruck2|Nothing]`: Prüft auf Gleichheit eines Objektverweises (nur auf Referenztypen anwendbar); liefert `True` zurück, wenn `Ausdruck1` auf den gleichen Datenspeicherbereich wie `Ausdruck2` zeigt. Wenn `Ausdruck1` keinem Speicherbereich zugewiesen ist (definierte Objektvariable aber kein instanziiertes Objekt), liefert der Vergleich durch `Is` mit `Nothing` den booleschen Wert `True` zurück.
- `Ausdruck1 IsNot [Ausdruck2|Nothing]`: Prüft auf Ungleichheit eines Objektverweises (nur auf Referenztypen anwendbar); liefert `True` zurück, wenn `Ausdruck1` auf einen anderen Datenspeicherbereich als `Ausdruck2` zeigt. Wenn `Ausdruck1` auf einen gültigen Speicherbereich mit Instanzdaten verweist, liefert der Vergleich durch `IsNot` mit `Nothing` den booleschen Wert `True` zurück.
- `String1 Like String2`: Prüft auf Ähnlichkeit zweier Strings; ein Mustervergleich kann den Vergleich flexibilisieren. Bei vorliegender Gleichheit der beiden Strings nach bestimmten Regeln[8] wird `True` zurückgeliefert, anderenfalls `False`.

Die folgenden Codezeilen demonstrieren den Einsatz der Vergleichsoperatoren:

```
Dim locString1 As String = "Uwe"
Dim locString2 As String = "Klaus"

locBoolean = (locString1 = locString2)      ' Ergibt False.
locBoolean = (locString1 > locString2)      ' Ergibt True.
locBoolean = (locString1 < locString2)      ' Ergibt False.
locBoolean = (locString1 >= locString2)     ' Ergibt True.
locBoolean = (locString1 <= locString2)     ' Ergibt False.
locBoolean = (locString1 <> locString2)     ' Ergibt True.
locBoolean = (locString1 Is locString2)     ' Ergibt False.

locString2 = "Uwe"
String.Intern(locString2)                  ' Ergibt jetzt True, da beide
locBoolean = (locString1 Is locString2)     ' Stringobjekte auf einen Bereich zeigen. (siehe Kap. 7)

locString1 = "Klau's, und lass Dich nicht erwischen"
locString2 = "Klau*"
locBoolean = (locString1 Like locString2)   ' Ergibt True.
```

[8] Genaueres erfahren Sie in MSDN unter dem Schlagwort *Like-Operator*.

Kurzschlussauswertungen mit OrElse und AndAlso

Betrachten Sie den folgenden Codeblock:

```
'Kurzschlussauswertung beschleunigt den Vorgang.
If locChar < "0" OrElse locChar > "9" Then
    locIllegalChar = True
    Exit For
End If
```

Auffällig ist hier das Schlüsselwort `OrElse`. Es gibt ein weiteres, das nach dem gleichen Prinzip funktioniert: `AndAlso`. Beide entsprechen den Befehlen `Or` bzw. `And`, und auch sie dienen dazu, boolesche Ausdrücke logisch miteinander zu verknüpfen und auszuwerten – nur viel schneller. Ein Beispiel aus dem täglichen Leben soll das verdeutlichen:

Wenn Sie sich überlegen, ob Sie einen Regenschirm zu einem Spaziergang mitnehmen, da es vielleicht regnet *oder auch (*`or else`*)* zumindest sehr verhangen ausschaut, dann machen Sie sich – berechtigterweise – schon keine Gedanken mehr, wie der Himmel aussieht, wenn Sie bereits festgestellt haben, *dass* es regnet. Sie brauchen das zweite Kriterium also gar nicht zu prüfen – der Schirm muss mit, sonst wird's nass! Genau das macht `OrElse` (bzw. `AndAlso`), und man nennt diese Vorgehensweise »Kurzschlussauswertung«.

Gerade bei Objekten bzw. dem Aufruf von Methoden können Ihnen Kurzschlussauswertungen helfen, Ihre Programme sicherer zu machen, wie das folgende Beispiel zeigt:

```
Private Sub btnAndAlsoDemo_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
Handles btnAndAlsoDemo.Click
    Dim einString As String = "Klaus ist das Wort, mit dem diese Zeile beginnt"
    If einString IsNot Nothing AndAlso einString.Substring(0, 5).ToUpper = "KLAUS" Then
        MessageBox.Show("Die Zeichenfolge beginnt mit Klaus!")
    End If

    If einString IsNot Nothing And einString.Substring(0, 5).ToUpper = "KLAUS" Then
        MessageBox.Show("Die Zeichenfolge beginnt mit Klaus!")
    End If
End Sub
```

Wie erwartet, zeigt dieser Programmcode zweimal einen Meldungstext an. Denn `einString` hat in beiden Fällen einen Inhalt, und beide Male beginnt die Zeichenfolge (es ist ja auch dieselbe) mit »Klaus«. Doch ersetzen Sie nun die Zeile

```
    Dim einString As String = "Klaus ist das Wort, mit dem diese Zeile beginnt"
```

durch

```
    Dim einString As String = Nothing
```

und lassen Sie das Programm ein weiteres Mal laufen. Das Ergebnis zeigt die folgende Abbildung:

```
Private Sub btnAndAlsoDemo_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
    Dim einString As String = Nothing
    If einString IsNot Nothing AndAlso einString.Substring(0, 5).ToUpper = "KLAUS" Then
        MessageBox.Show("Die Zeichenfolge beginnt mit Klaus!")
    End If

    If einString IsNot Nothing And einString.Substring(0, 5).ToUpper = "KLAUS" Then
        MessageBox.Show("Die Zeichenfolge beginnt mit Klaus!")
    End If
End Sub
Class
```

NullReferenceException wurde nicht behandelt.

Der Objektverweis wurde nicht auf eine Objektinstanz festgelegt.

Hinweise zur Fehlerbehebung:

Erstellen Sie eine Objektinstanz mit dem Schlüsselwort "neu".

Überprüfen Sie, ob das Objekt NULL ist, bevor Sie die Methode aufrufen.

Für diese Ausnahme die allgemeine Hilfe aufrufen.

Weitere Hilfeinformationen online suchen...

Aktionen:

Details anzeigen...

Ausnahmedetail in die Zwischenablage kopieren

Abbildung 2.14 *AndAlso* dient Ihnen bei kombinierten Abfragen auf *Nothing* und Verwendungen von Instanzmethoden

Hier wird die Arbeitsweise von `AndAlso` deutlich. Die erste Abfrage funktioniert, weil der zweite, durch `AndAlso` verknüpfte Teil `einString.Substring(0, 5).ToUpper = "KLAUS"` schon gar nicht mehr abgehandelt wird. Das Objekt `einString` war nämlich `Nothing`, und bei der Verwendung von `AndAlso` interessieren alle folgenden Prüfteile nicht mehr.

In der zweiten Variante nur mit `And` wird, obwohl es in diesem Zusammenhang keinen Sinn macht, der zweite Part sehr wohl abgehandelt. Aber eben weil `einString` den Wert `Nothing` aufweist, können Sie seine Instanzfunktionen (`SubString`, `ToUpper`) nicht verwenden; das Programm bricht mit einer `NullReferenceException` ab.

Select ... Case ... End Select

Sie können, wie im vorherigen Abschnitt zu sehen war, `ElseIf` zur Optionsanalyse verwenden, wenn Sie mehrere boolesche Ausdrücke in einem Rutsch prüfen und darauf mit der Ausführung entsprechender Programmcodes reagieren müssen. Mit einem `Select`-Konstrukt geht das allerdings sehr viel eleganter. `Select` bereitet einen Ausdruck für einen Vergleich mit booleschem Ergebnis vor; der eigentliche Vergleich findet dann durch verschiedene `Case`-Anweisungen, aber mindestens eine `Case`-Anweisung statt, hinter denen jeweils ein entsprechendes Vergleichsargument vom gleichen Typ (oder implizit konvertierbar) folgen muss. `Case Else` kann optional für die Ausführung von Anweisungen herangezogen werden, wenn keine der hinter `Case` angegebenen Bedingungen zutraf. `End Select` schließt das Konstrukt ab. Andererseits führt `Select` auch keine weiteren Überprüfungen durch, wenn einmal eine Bedingungsprüfung positiv verlief.

Bei der Bedingungsprüfung prüft `Case` ohne Zusatz auf Gleichheit. Durch Verwenden des `Is`-Schlüsselwortes können Sie auch andere Vergleichsoperatoren verwenden. Das folgende Beispiel soll den Umgang verdeutlichen:

```
Dim locString1 as String = "Miriam"

Select Case locString1
   Case "Miriam"
      Console.WriteLine("Treffer")
   Case Is > "N"
      Console.WriteLine("Name kommt nach 'M' im Alphabet")
```

```
   Case Is < "M"
      Console.WriteLine("Name kommt vor 'M' im Alphabet")

   Case Else
      Console.WriteLine("Name beginnt mit 'M'")

   'Case Like "Miri"
      'Das funktioniert nicht!!!

End Select
```

Allerdings werden hier Vergleichsoperation und bedingte Ausführung in einem Abwasch gemacht, sodass das folgende Konstrukt nicht funktioniert:

```
        'Das funktioniert so nicht!!!
Select Case locBoolean

   Case
Console.WriteLine("War wahr!")

End Select
```

Der Compiler meckert hier zurecht.

If-Operator und IIf-Funktion

`If` gibt es nicht nur als Strukturanweisung (`If … Then … Else`), sondern auch als Funktion – und zwar in zwei Versionen. Die vorhin bereits kurz erwähnte `IIf`-Funktion nimmt 3 Parameter entgegen und liefert in Abhängigkeit des Ergebnisses des ersten booleschen Ausdrucks entweder den ersten oder den zweiten Ausdruck als Ergebnis zurück:

```
Dim c As Integer
'Liefert 10 zurück
c = CInt(IIf(True, 10, 20))
'Liefert 20 zurück
c = CInt(IIf(False, 10, 20))
```

`IIf` hat einen großen Nachteil: Wenn Sie die `IIf`-Funktion (schreibt sich mit zwei »i«) verwenden, müssen Sie das Ergebnis in den Typ casten, der der Zuweisung entsprach, da die `IIf`-Funktion nur `Object` zurücklieferte, also beispielsweise:

```
Dim c As Integer
'Liefert 10 zurück
c = CInt(IIf(True, 10, 20))
```

Das geht seit Visual Basic 2008 einfacher, denn das Schlüsselwort `If` (mit einem »i«) ist für den Gebrauch als Operator erweitert worden:

```
Dim c As Integer
'Liefert 20 zurück
c = If(False, 10, 20)
```

Noch weniger Schreibaufwand verursacht das, wenn Sie den `If`-Operator mit lokalem Typrückschluss kombinieren:

```
'Liefert 20 zurück
Dim c = If(False, 10, 20)
```

Das Mischen von verschiedenen Typen bei der Typrückgabe bringt den Compiler allerdings ins Straucheln, wie in Abbildung 2.15 zu sehen.

Abbildung 2.15 If als Ersatz für IIf funktioniert nur dann, wenn Sie dem Compiler die Chance geben, die entsprechenden Typen eindeutig zu ermitteln

Schleifen

Schleifen dienen dazu, die in ihnen enthaltenen Befehle und Anweisungen immer wieder auszuführen, so lange, bis ein bestimmter Schleifenende-Zustand (die so genannte *Abbruchbedingung*) erreicht wurde. In der Abbildung auf Seite 15 sehen Sie ein Beispiel für eine Schleife in Form eines `Do … Loop`-Konstrukts, das dafür sorgt, dass der Anwender wiederholt Geburtsdatumsangaben bzw. Befehlseingaben durchführen kann, und zwar so oft, bis er in der Eingabeaufforderung einfach ↵ drückt, die Variable `eingabe` also leer bleibt und die Abbruchbedingung für die Schleife erreicht ist, die hinter dem `Loop`-Schlüsselwort formuliert wurde.

Neben `Do … Loop`-Schleifen kennt Visual Basic übrigens auch `For … Next`-Schleifen, die alle inneren Anweisungen so oft wiederholen, bis eine Zählvariable einen bestimmten Wert über- oder unterschritten hat, `For Each … Next`-Schleifen, die alle inneren Anweisungen so oft wiederholen, wie es bestimmte Elemente in einer Objektauflistung gibt, sowie `While … End While`-Schleifen, die ihre inneren Anweisungen so oft wiederholen, wie eine bestimmte Abbruchbedingung noch gilt, die direkt hinter `While` angegeben wird. Gibt es übrigens keine Abbruchbedingung, spricht man von einer Endlosschleife – das Schleifeninnere läuft dann so lange, bis Sie Ihre Stromrechnung nicht mehr bezahlen. (Wobei das streng genommen auch schon wieder eine Abbruchbedingung darstellt.)

For … Next-Schleifen

`For … Next`-Schleifen erlauben Ihnen, eine Reihe von Anweisungen des Codeblocks, den sie einschließen, eine bestimmte, vordefinierte Anzahl oft zu wiederholen. Die Syntax für eine solche Konstruktion lautet wie folgt (Elemente in eckigen Klammern sind optional, was heißt, dass sie nicht in jedem `For … Next`-Schleifenkonstrukt vorkommen *müssen*):

```
For zählvariable [As datentyp] = start To ende [Step schrittweite]
    [Anweisungen]
    [Exit For]
    [Continue For]
Next [zählvariable]
```

Die `zählvariable` der `For`-Anweisung ist zentraler Bestandteil des gesamten Schleifenkonstrukts und deswegen nicht optional – Sie müssen sie angeben. `datentyp` bestimmt, von welchem numerischen Typ (zum Beispiel `Integer` oder `Long`) `zählvariable` sein soll, kann aber auch weggelassen werden, wenn für Ihr Projekt der lokale Typrückschluss eingeschaltet ist oder wenn diese Variable bereits zuvor definiert war. Letzteres ist wichtig, wenn die Variable auch nach dem Verlassen des Schleifenkonstrukts gültig sein und ihr Inhalt weiterverwendet werden soll.

WICHTIG Theoretisch könnten Sie auch Fließkommavariablentypen wie `Double` und `Single` verwenden, aber darauf sollten Sie verzichten. Durch Rundungsungenauigkeiten, die bei der Umwandlung vom internen Dual- ins Dezimalsystem Natur der Sache sind, können Sie die Abfrage nach Grenzwerten nicht präzisieren. So könnte eine Schleife vielleicht einmal zu oft wiederholt werden, obwohl Sie es nicht erwarten, oder ein gezieltes Prüfen auf einen bestimmten Wert könnte durch Rundungsgenauigkeiten fehlschlagen – der Wert 63,0000001 ist zwar dicht dran an 63, aber eben nicht 63. Eine Prüfung der Variable auf 63 per `If`-Abfrage würde in diesem Fall fehlschlagen.

Falls Sie dennoch nicht auf Fließkommazahlen in Schleifen verzichten können, ziehen Sie in Betracht, den genaueren, aber sehr viel langsameren `Decimal`-Typ zu verwenden oder testen Sie nach bestimmten Werten in Bereichen, etwa:

```
If doubleVar >= 10.3 And doubleVar < 10.4 Then
    [Anweisungen]
End If
```

`start` und `ende` sind erforderliche Angaben und bestimmen, mit welchem Wert `zählvariable` initial zugewiesen soll und damit, wo das Zählen startet, sowie den Grenzwert, wie weit gezählt werden soll. Standardmäßig wird in 1er-Schritten gezählt, Sie können aber durch den Parameter `schrittweite` genau diese angeben.

Mit `Exit For` können Sie eine `For … Next`-Schleife vorzeitig verlassen – Sinn ergibt diese Anweisung deshalb auch nur, wenn Sie sie in einer `If`-Abfrage innerhalb der Schleife platzieren. `Exit For` beendet die Schleife und sorgt dafür, dass das Programm mit der Ausführung der Anweisungen hinter `Next` fortfährt.

Die Angabe von `Next` ist wiederum erforderlich, denn es begrenzt die `For`-Schleife im Programmablauf nach unten, bildet also die zweite Grenze der Schleife.

TIPP Das `For … Next`-Schleifenkonstrukt sollten Sie dann verwenden, wenn Sie von vorneherein wissen, wie viele Schleifendurchläufe zu erwarten sind. Haben Sie zum Beispiel ein Array mit einer bestimmten Anzahl an Elementen, die Sie verarbeiten möchten, dann eignet sich dazu eine `For … Next`-Schleife ausgezeichnet:

```
Dim Namen() As String = {"Lisa Feigenbaum",
                         "Sarika Calla",
                         "Ramona Leenings",
                         "Tanja Gelo"}

'Mit For-Schleife durchlaufen und ausgeben:
For index As Integer = 0 To Namen.Length - 1
    Console.WriteLine(Namen(index))
Next
```

In Fällen, bei denen Sie im Vorfeld nicht wissen können, wie viele Durchläufe notwendig sein werden, sollten Sie lieber die `Do … Loop`-Schleife verwenden, die Sie weiter unten beschrieben finden.

Dieses Beispiel gibt Folgendes auf dem Bildschirm aus:

```
Beth Messi
Lisa Feigenbaum
Sarika Calla
Ramona Leenings
Beth Messi
Tanja Gelo
```

Beachten Sie bei diesem Beispiel, dass die Elemente eines Arrays oder eine Auflistung immer 0-basiert sind. Deswegen wird als oberer Grenzwert für das Durchlaufen des Arrays von der Größe des Arrays (Anzahl der Elemente im Array wird mit `Length` ermittelt) der Wert 1 abgezogen.

Wenn eine `Next`-Anweisung einer äußeren Schachtelungsebene vor der `Next`-Anweisung einer inneren Ebene auftritt, beanstandet der Compiler das, und Sie sehen eine entsprechende Meldung in der Fehlerliste. Der Compiler kann diesen Überlappungsfehler einerseits nur erkennen, wenn Sie in jeder `Next`-Anweisung die entsprechende Zählvariable angeben, nimmt aber andererseits die jeweils richtige Zählvariable an, wenn Sie keine explizit benennen. Die letzte Variante ist deswegen oftmals die bessere.

HINWEIS Der Wert von `schrittweite` kann bei ganzen Zahlen positiv oder negativ sein. Sollte die Schrittweite negativ sein, müssen Sie natürlich die Start- und Endwerte der Schleife entsprechend anpassen – `startwert` muss dann größer sein als der Endwert, damit es überhaupt zu mehreren Schleifendurchläufen kommt. Falls kein Wert für `schrittweite` angegeben ist, wird wie bereits erwähnt standardmäßig 1 verwendet. Auch wichtig: Falls `start` größer als `ende` ist, wird nicht automatisch -1 als Schrittweite angenommen. Sie müssen dann `schrittweite` in jedem Fall mit angeben, damit mehrere Schleifendurchläufe durchgeführt werden.

WICHTIG `start`, `ende` und `schrittweite` können nicht nur konstante Werte sein, sondern auch berechnet und komplexe Ausdrücke:

```
For tagesZähler= 1 To (Now.Date - New Date(Now.Year, Now.Month, 1)).TotalDays
    (Anweisungen)
Next
```

In diesem Beispiel gilt wie für alle Ausdrücke: Sie werden nur ein einziges Mal ausgewertet, wenn die Schleife ihre Arbeit aufnimmt. Das ist wichtig zu wissen, sowohl für Performanceüberlegungen (Konstrukte in Form des Beispiels verlangsamen den Ablauf nicht, eben weil sie nur einmal ausgewertet werden) als auch für den Ablauf: Sie können den Grenzwert, der zum Schleifenende führt, nicht verändern, während die Schleife läuft. Ebenfalls wichtig: Sie sollten die Zählvariable nicht noch an anderer Stelle innerhalb der Schleife manipulieren, weil das zu unvorhergesehenen Effekten führen bzw. die Fehlersuche innerhalb von Algorithmen, die in Schleifen laufen, schwierig werden lassen kann.

Verschachteln von For-Schleifen

For-Schleifen lassen sich ineinander verschachteln, was bedeutet, dass innerhalb einer For-Schleife eine weitere For-Schleife ablaufen kann. Damit multiplizieren sich die Gesamtdurchläufe der innersten Schleife, wenn alle Schleifen konstante Werte verwenden. Ein Beispiel:

```
For count1 As Integer = 1 To 10
    For count2 As Integer = 1 To 10
        For count3 As Integer = 1 To 10
            ' Die Anweisungen hier werden insgesamt 10*10*10=1000 Mal durchlaufen.
        Next count3
    Next
Next count1
```

TIPP Sie müssen die zählvariable bei einer Next-Anweisung nicht unbedingt angeben. Allerdings verbessert das die Lesbarkeit Ihres Programms unter Umständen bei häufigen und gerade bei tiefen Verschachtelungen. Aber:

For Each-Schleifen

Mithilfe einer For Each-Schleife wiederholen Sie Ihre inneren Anweisungen für jedes Element der angegebenen Auflistung oder des Arrays. Die generelle Anwendung von For Each lautet:

```
For Each element [As datentyp] In auflistung
    [Anweisungen]
    [Exit For]
    [Anweisungen]
    [Continue For]
Next [element]
```

Die Variable element stellt dabei die Variable dar, die die Schleife zum Durchlaufen der Elemente der Auflistung verwendet – jedes Element der Auflistung wird dabei für jeden Schleifendurchgang an die Variable element zugewiesen. Entsprechend der Anzahl der Elemente, die sich in der Auflistung oder dem Arrays auflistung befinden, werden dann die inneren Anweisungen in der Schleife wiederholt ausgeführt.

Optional können Sie mit Exit For die Schleifendurchläufe vorzeitig beenden – weswegen es Sinn ergibt, Exit For in einem gesonderten If-Block einzuschließen.

Die Angabe von Next ist erforderlich, denn es beendet die Definition der For-Schleife; Sie müssen aber element hinter Next nicht unbedingt angeben.

HINWEIS Mehr zu Auflistungen finden Sie in Kapitel 20; dort wird in diesem Zusammenhang auch noch intensiver auf die Funktionsweise von For Each-Schleifen eingegangen.

Das folgende Beispiel zeigt, wie Sie mithilfe einer `For Each`-Schleife durch das `String`-Array des letzten Beispiels hindurchiterieren und die Inhalte aller Elemente dieses Arrays in das Konsolenfenster ausgeben:

```
Dim Namen() As String = {"Lisa Feigenbaum",
                         "Sarika Calla",
                         "Ramona Leenings",
                         "Beth Messi",
                         "Tanja Gelo"}

'Mit ForEach-Schleife durchlaufen und ausgeben:
For Each name In Namen
    Console.WriteLine(name)
Next
```

TIPP In manchen Fällen ist es notwendig zu wissen, wann das letzte Element einer Liste erreicht ist. In diesem Fall können Sie `For Each` nicht verwenden, sondern müssen auf `For … Next` zurückgreifen:[9]

```
'Letztes Element erfährt Sonderbehandlung:
For index As Integer = 0 To Namen.Length - 1
    If index < Namen.Length - 1 Then
        Console.WriteLine(Namen(index))
    Else
        Console.WriteLine("und last, but not least: " & Namen(index))
    End If
Next
```

Dieser Code gibt Folgendes im Konsolenfenster aus:

```
Lisa Feigenbaum
Sarika Calla
Ramona Leenings
Beth Messi
und last, but not least: Tanja Gelo
```

Do … Loop- und While … End While-Schleifen

`Do … Loop` und `While … End` wiederholen Anweisungen im Rumpfinneren entweder *so lange*, wie eine Bedingung `True` ist bzw. *bis* die Bedingung `True` wird. Dabei wird im Zusammenhang mit `Do` das Schlüsselwort `Until` verwendet, um anzuzeigen, dass eine Bedingung `True` werden muss, und Sie verwenden `While`, um die Schleife so lange laufen zu lassen, *wie* eine Bedingung `True` ist.

`While` und `Until` können Sie dabei sowohl hinter den Schleifenanfang (`Do`) als auch an das Schleifenende (`Loop`) platzieren.

[9] Denkbar wäre natürlich auch, eine zusätzliche Zählvariable in der `For Each`-Schleife zu verwenden – aber das ist eher überflüssig, denn dann können Sie gleich `For … Next` verwenden.

Mit `Do While … Loop` erreichen Sie dasselbe wie mit `While … End While` – für dasselbe Ergebnis verwenden Sie hier nur – aus historischen Gründen – eine andere Syntax. Damit haben Sie folgende prinzipielle Variationsmöglichkeiten, ein solches Schleifenkonstrukt mit den entsprechenden Abbruchbedingungen aufzubauen:

```
Do {While|Until} Bedingung
    [Anweisungen]
    [Exit Do]
    [Anweisungen]
Loop
```

oder

```
Do
    [Anweisungen]
    [Exit Do]
    [Anweisungen]
Loop {While|Until} Bedingung
```

oder

```
While condition
    [Anweisungen]
    [Exit While]
    [Anweisungen]
End While
```

Sowohl bei `While … End While` als auch bei `Do … Loop` dienen boolesche Variablen bzw. boolesche Ausdrücke als Abbruchbedingungen. Die folgenden Codezeilen zeigen ein paar Beispiele für den Einsatz dieser Schleifentypen:

```
        Dim Namen() As String = {"Lisa Feigenbaum",
                                 "Sarika Calla",
                                 "Ramona Leenings",
                                 "Beth Messi",
                                 "Tanja Gelo"}

        'Die Namen einmal in aufsteigender Reihenfolge ...
        Dim index As Integer = 0
        Do While index < Namen.Length
            Console.WriteLine(Namen(index))
            index += 1
        Loop

        Console.WriteLine("-------------------------")
        Console.WriteLine("      und rückwärts:")
        Console.WriteLine("-------------------------")

        '... und in absteigender Reihenfolge ausgeben.
        index = Namen.Length - 1
        While index > -1
            Console.WriteLine(Namen(index))
            index -= 1
        End While
```

Diese Codezeilen bewirken, dass Folgendes auf dem Bildschirm ausgegeben wird:

```
Lisa Feigenbaum
Sarika Calla
Ramona Leenings
Beth Messi
Tanja Gelo
-------------------------
      und rückwärts:
-------------------------
Tanja Gelo
Beth Messi
Ramona Leenings
Sarika Calla
Lisa Feigenbaum
```

Exit – Vorzeitiges Verlassen von Schleifen

Die `Exit`-Anweisung veranlasst, dass das Programm die Schleife abbricht und seine Ausführung hinter dem Schleifenende fortsetzt – bei einer `For`-Schleife setzt die Programmausführung also nach der `Next`-Anweisung fort, bei einer `Do`-Schleife hinter `Loop`, bei `While` hinter `End While`. Möglicherweise möchten Sie eine Schleife beenden, wenn Sie eine Bedingung feststellen, die das Fortsetzen des Durchlaufs unnötig oder unmöglich macht, zum Beispiel ein fehlerhafter Wert oder eine Aufforderung zum Beenden. Wenn Sie eine Ausnahme in einer `Try … Catch … Finally`-Anweisung abfangen (siehe Abschnitt »Fehlerbehandlung im Code« ab Seite 58), lässt sich beispielsweise am Ende des `Finally`-Blocks `Exit For` verwenden.

Sie können eine beliebige Anzahl von `Exit`-Anweisungen an jeder Stelle einer Schleife einfügen. `Exit` wird häufig nach der Auswertung einer Bedingung verwendet, zum Beispiel in einer `If … Then … Else`-Struktur.

Continue – Vorzeitige Schleifenwiederholung

Sie können auch eine Schleife vorzeitig »loopen« lassen – in einer `For … Next`-Schleife also in bestimmten Fällen das `Next` quasi vorziehen. Das funktioniert mit `Continue`. Sie können `Continue` natürlich auch bei allen anderen Schleifentypen einsetzen.

Verkürzter Zugriff auf Objekteigenschaften und -methoden mit With ... End With

Mit `With … End With` können Sie eine Reihe von Elementen (Eigenschaften, Methoden) eines Objekts innerhalb eines Codeblocks immer wiederverwenden, ohne den Namen des Objekts immer wieder neu nennen zu müssen. Wenn der vollqualifizierte Name für die entsprechende Objektvariable sehr lang ist, kann so durch die Verwendung von `With … End With` nicht nur Tipparbeit gespart, sondern auch die Leistung verbessert werden. Außerdem senken Sie obendrein das Risiko einer falschen Schreibweise eines seiner Elemente.

Wenn Sie beispielsweise mehrere verschiedene Eigenschaften eines einzelnen Objekts verwenden möchten, fügen Sie die Anweisungen für die Eigenschaftenzuweisungen in eine `With … End With`-Struktur ein. Sie müssen dann nicht in jeder Eigenschaftenzuweisung auf das Objekt verweisen, sondern es genügt ein einziger Verweis auf das Objekt, indem Sie dem Eigenschaftennamen lediglich einen Punkt voranstellen (Abbildung 2.16).

Abbildung 2.16 In einem With … End With-Block greifen Sie mit dem Punkt auf die Liste der Methoden und Eigenschaften des Objekts zu, das hinter With angegeben wurde

HINWEIS Denken Sie daran, dass auch `With … End With` eine Struktur bildet. Variablen, die zwischen `With` und `End With` definiert werden, haben daher nur in diesem Gültigkeitsbereich Bestand. Der folgende Abschnitt hält mehr zu diesem Thema bereit.

Gültigkeitsbereiche (Scope) von Variablen

Bei Variablen, die auf Prozedurebene deklariert wurden (also innerhalb von `Sub`- und `Function`-Methoden oder – dazu kommen wir später noch – auch `Property`-Prozeduren), gibt es Auswirkungen auf ihren Gültigkeitsbereich. Während in VBA und Visual Basic 6.0 beispielsweise noch Variablen innerhalb von Prozeduren überall ab dem Zeitpunkt ihrer Deklaration gelten, sind sie in .NET-Versionen von Visual Basic ab dem Zeitpunkt ihrer Deklaration nur innerhalb des Codeblocks gültig, in dem sie definiert sind. »Codeblock« in diesem Zusammenhang bedeutet im Prinzip irgendetwas, was Code in irgendeiner Form einklammern kann – das können beispielsweise `If … Then … Else`-Abfragen, `For … Next`- oder `Do … Loop`-Schleifen, aber auch `With`-Blöcke sein. Es gilt dabei die Regel: Jede Strukturanweisung, die dazu führt, dass der Editor den dazwischen platzierten Code einrückt, begrenzt automatisch den Gültigkeitsbereich von Variablen, die in diesem Block definiert sind. Ein Beispiel zeigt Abbildung 2.17.

```
Option Strict On

Module Module1

    Sub Main()

        For zähler As Integer = 0 To 10
            Dim fünfGefunden As Boolean
            If zähler = 5 Then
                fünfGefunden = True
            End If
        Next
        If fünfGefunden Then
            Der Name "fünfGefunden" wurde nicht deklariert.
            Debug.Print("Die Zahl 5 kam in der Zahlenreihe vor!")
        End If
    End Sub

End Module
```

Abbildung 2.17 Variablen sind nur in dem Strukturblock gültig, in dem sie deklariert wurden

Wie in der Abbildung zu sehen, versucht das Programm beim zweiten Mal, auf `fünfGefunden` zuzugreifen, nachdem der Strukturbereich und damit auch der Gültigkeitsbereich der Variable verlassen wurde – und das führt zu einem Fehler.

Wollten Sie `fünfGefunden` in jedem Strukturblock zugreifbar machen, müssten Sie folgende Änderungen vornehmen:

```
Sub Main()

    Dim fünfGefunden As Boolean

    For zähler As Integer = 0 To 10
        'Dim fünfGefunden As Boolean
        If zähler = 5 Then
            fünfGefunden = True
        End If
    Next

    If fünfGefunden Then
        Debug.Print("Die Zahl 5 kam in der Zahlenreihe vor!")
    End If
End Sub
```

Die Änderungen sind hier im Listing durch Fettschrift gekennzeichnet.

Diese Regel für Gültigkeitsbereiche führt dazu, dass Variablen innerhalb mehrerer Strukturblöcke mehrfach unter gleichem Namen deklariert werden können, wie das folgende Beispiel zeigt:

```
Sub Main()
    For zähler As Integer = 0 To 10
        Dim fünfGefunden As Boolean
        If zähler = 5 Then
            fünfGefunden = True
        End If
    Next
    For zähler As Integer = 0 To 10
        Dim fünfGefunden As Boolean
```

```
        If zähler = 5 Then
            fünfGefunden = True
        End If
    Next
End Sub
```

Hier wird die Variable `fünfGefunden` innerhalb einer Prozedur zweimal deklariert – dennoch meldet Visual Basic keinen Fehler, weil sich beide Deklarationen in unterschiedlichen Gültigkeitsbereichen befinden.

```
Option Strict On

Module Module1

    Sub Main()

        For zähler As Integer = 0 To 10
            Dim fünfGefunden As Boolean
            If zähler = 5 Then
                fünfGefunden = True
            End If
            For zweiterZähler As Integer = 0 To 10
                Dim fünfGefunden As Boolean
                    Die Variable "fünfGefunden" verbirgt eine Variable in einem einschließenden Block.
                If zähler = 5 Then
                    fünfGefunden = True
                End If
            Next
        Next

    End Sub

End Module
```

Abbildung 2.18 Variablen eines übergeordneten Gültigkeitsbereichs dürfen solche eines untergeordneten nicht verbergen

Allerdings: Das Deklarieren von Variablen gleichen Namens in einem Gültigkeitsbereich, der einen anderen kapselt, funktioniert natürlich nicht, denn Variablen in einem übergeordneten Gültigkeitsbereich sind ohnehin immer von einem untergeordneten aus zugreifbar. Eine entsprechende Fehlermeldung würde, falls Ihnen das passiert, dann so ausschauen, wie in Abbildung 2.18 zu sehen.

Die Operatoren += und –= und ihre Verwandten

Mit += und –= ist Visual Basic um Operatoren für numerische Berechnungen und bei &= auch für Stringverkettungen reicher geworden – das folgende Beispiel (dem nächsten Abschnitt entnommen) verdeutlich ihre Verwendung:

```
    Do
        If (hilfswert And 1) = 1 Then
            ergebniswert += wert1
        End If
        wert1 = wert1 << 1
        hilfswert = hilfswert >> 1
    Loop Until hilfswert = 0
```

Es gibt andere Operatoren in Visual Basic, die das ebenfalls können – die folgende Tabelle zeigt, welche das sind:

Operation	Kurzform	Beschreibung
`var = var + 1`	`var += 1`	Den Variableninhalt um eins erhöhen
`var = var – 1`	`var –= 1`	Den Variableninhalt um eins reduzieren
`var = var * 2`	`var *= 2`	Den Variableninhalt verdoppeln (mit zwei multiplizieren)
`var = var / 2`	`var /= 2`	Den Variableninhalt halbieren (durch zwei dividieren – Fließkommadivision)
`var = var \ 2`	`var \= 2`	Den Variableninhalt ohne Rest halbieren (durch zwei dividieren – Integerdivision)
`var = var ^ 3`	`var ^= 3`	Den Variableninhalt mit drei potenzieren
`varString = VarString & "Ramona"`	`varString &= "Ramona"`	An den Inhalt des Strings *varString* die Zeichenkette »Ramona« anhängen

Tabelle 2.2 Kurzformen von Operatoren in Visual Basic

HINWEIS Auch wenn die Verwendung der Kurzformen weniger Tipparbeit bereitet – eine schnellere Codeausführung bewirkt sie nicht.

Also ganz egal, ob Sie

```
intvar = intvar + 1
```

oder

```
intvar += 1
```

schreiben – der Compiler wird aus beiden Angaben den gleichen Code erzeugen.

Die Bitverschiebeoperatoren << und >>

Zusätzlich zu den genannten Operatoren gibt es seit Visual Basic 2003 die so genannten Bitverschiebeoperatoren, mit denen die einzelnen Bits von Integer-Werten schrittweise nach links oder nach rechts verschoben werden können. Ohne im Detail auf die Funktionsweise des Binärsystems eingehen zu wollen: Eine Verschiebung der Bits eines Integer-Werts nach links verdoppelt seinen Wert, eine Verschiebung nach rechts teilt seinen Wert ohne Rest durch 2.

Aus binär 101 (dezimal 5) wird durch Rechtsverschiebung also binär 10 (dezimal 2) und durch Linksverschiebung binär 1010 (dezimal 10).

Der folgende Code demonstriert einen Multiplikationsalgorithmus auf Bitebene. Die Älteren unter uns erinnern sich vielleicht noch an ihre Commodore 64-Zeiten. Dort galt das Anwenden solcher Algorithmen in Assembler zur täglichen Praxis!

```
Private Sub MultiplikationMitBitverschiebung
    Dim wert1, wert2, ergebniswert, hilfswert As Integer
    wert1 = 10
    wert2 = 5
    ergebniswert = 0
    hilfswert = wert2

        'Dieser Algorithmus funktioniert so, wie Sie
        'auch im Dezimalsystem "zu Fuß" multiplizieren:
        '
        '(10)   (5)
        '1010 * 101 =
        '------------
        '      1010 +
        '     0000  +
        '    1010
        '------------
        '    110010 = 50

        'Die "5" wird dazu bitweise nach rechts verschoben,
        'um ihr rechts äußeres Bit zu testen. Ist es gesetzt,
        'wird der Wert von 10 zunächst addiert, und dann sein
        'kompletter "Bitinhalt" um eine Stelle nach links verschoben;
        'ist es hingegen nicht gesetzt, passiert gar nichts.
        'Dieser Vorgang wiederholt sich so lange, bis alle
        'Bits von "5" verbraucht sind - die Variable hilfswert,
        'die diesen Wert verarbeitet, also 0 geworden ist.
        'Für eine Multiplikation sind also gerade so viele
        'Additionen nötig, wie Bits im zweiten Wert gesetzt sind.
    Do
        If (hilfswert And 1) = 1 Then
            ergebniswert += wert1
        End If
        wert1 = wert1 << 1
        hilfswert = hilfswert >> 1
    Loop Until hilfswert = 0
    Console.WriteLine("Das Ergebnis lautet:" & ergebniswert)
End Sub
```

Fehlerbehandlung im Code

Mir persönlich war die Fehlerbehandlung in den alten Visual Basic-Versionen, die bis heute in .NET funktionieren, immer ein Gräul. Trat in einer sehr langen Routine im fertigen Programm ein Fehler auf, war es vergleichsweise schwierig oder mit großem Aufwand verbunden, die genaue Stelle des Fehlers zu lokalisieren. Zwar konnten Sie mit der Systemvariablen `Erl` die Zeile in der Fehlerbehandlungsroutine anzeigen lassen, in der der Fehler aufgetreten war, aber dazu mussten Sie manuell vor jede Zeile eine Zeilennummer setzen – selbst für »damalige« Verhältnisse war das doch eine eher vorsintflutliche Vorgehensweise.

Das geht heute viel, viel einfacher – auch wenn das Visual Basic Team die ursprüngliche Verfahrensweise auch noch zulässt. Schauen wir uns abermals das Vorher und das Nachher an – hier am Beispiel einer kleinen VBA-Routine, die eine Datei in eine `String`-Variable liest, oder dies zumindest versucht. Achten Sie im Listing auf die Kommentare, die mit den Ziffern den Programmverlauf im Falle eines Fehlers kennzeichnen.

Achtung! Visual Basic 6.0/VBA-Code!

```
Private Sub Command1_Click()

    Dim DateiNichtGefundenFlag As Boolean
    On Local Error GoTo 1000

    Dim ff As Integer
    Dim meinDateiInhalt As String
    Dim zeilenspeicher As String
    ff = FreeFile
    '1: Hier tritt der Fehler auf
    Open "C:\EineTextdatei.TXT" For Input As #ff
    '3: dann wieder hier hin
    If DateiNichtGefunden Then
        '4: um dann diese Meldung auszugeben.
        MsgBox ("Die Datei existiert nicht")
    Else
        'Dieser Block wird nur ausgeführt,
        'wenn alles OK war.
        Do While Not EOF(ff)
            Line Input #ff, zeilenspeicher
            meinDateiInhalt = meinDateiInhalt & zeilenspeicher
        Loop
        Close ff
        Debug.Print meinDateiInhalt
    End If
    'Und das ist auch nötig, damit das
    'Programm nicht in die Fehlerroutine rennt.
    Exit Sub

    '2: Hier springt dann das Programm hin
1000 If Err.Number = 53 Then
        DateiNichtGefunden = True
        Resume Next
    End If

End Sub
```

Erstaunlich, oder? Um einen simplen Fehler abzufangen, muss sich das Programm kreuz und quer durch den Programmcode quälen, und – einem Steinbock im Himalaja gleich – hin und her springen, was das Zeug hält. Und wir fangen hier gerade mal einen einzigen Fehler ab!

Das noch Unangenehmere ist: Prinzipiell ist dieses `On Error GoTo`-Monster auch noch in allen VB.NET-Derivaten möglich, wenn auch alles andere als nötig – wie Sie gleich noch sehen werden. Das Einlesen einer Textdatei funktioniert hier zwar ein wenig anders, da es `Open` und `Close` in dieser VB6-Form nicht mehr gibt. Aber darum geht es an dieser Stelle auch gar nicht.

WICHTIG Der einzige Unterschied, jedenfalls was die Fehlerbehandlung anbelangt: Zeilennummern müssen, wie andere Sprungmarken, mit Doppelpunkten versehen werden. Wollte man also das obige Programm in VB.NET übersetzen, käme vielleicht dieses (leider) mögliche Ergebnis dabei heraus:

```
Private Sub btnDateiLesenDotNet_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
                                Handles btnDateiLesenDotNet.Click
    'WICHTIG:
    'Um auf die IO-Objekte zuzugreifen, muss "Imports System.IO"
    'am Anfang der Codedatei stehen!

    Dim DateiNichtGefundenFlag As Boolean
    'Selbe Blödsinn hier!
    On Error GoTo 1000

    Dim meinDateiInhalt As String
    '1: Wenn hier ein Fehler auftritt
    Dim dateiStromLeser As New StreamReader("C:\EineTextdatei.txt")
    '3: um das durch Resume Next hier wieder zu landen
    If DateiNichtGefundenFlag Then
        '4: und den Fehler abzufangen
        MessageBox.Show("Datei wurde nicht gefunden!")
    Else
        'Trat kein Fehler auf, wird dieser Block ausgeführt
        meinDateiInhalt = dateiStromLeser.ReadToEnd()
        dateiStromLeser.Close()
        'Und der Dateiinhalt ausgegeben.
        Debug.Print(meinDateiInhalt)
    End If
    Exit Sub
    '2: Fährt der Programmablauf hier fort
1000: If Err.Number = 53 Then
        DateiNichtGefundenFlag = True
        Resume Next
    End If
End Sub
```

Elegantes Fehlerabfangen mit Try/Catch/Finally

Es gibt aber eine viel, viel elegantere Möglichkeit in VB.NET, Fehlerbehandlungen zu implementieren. Im Gegensatz zu `On Error GoTo` erlaubt die Struktur `Try/Catch/Finally` Fehler an den Stellen zu behandeln, an denen sie auch tatsächlich auftreten können. Schauen Sie sich zur Verdeutlichung das folgende Beispiel an, das die auf `Try/Catch` adaptierte Version des obigen Beispiels enthält:

```
Private Sub btnDateiLesenDotNet_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
Handles btnDateiLesenDotNet.Click
    'WICHTIG:
    'Um auf die IO-Objekte zuzugreifen, muss "Imports System.IO"
    'am Anfang der Codedatei stehen!

    Dim meinDateiInhalt As String
    Dim dateiStromLeser As StreamReader

    Try
        'Diese folgenden Befehle probieren (try=versuchen, ausprobieren).
        dateiStromLeser = New StreamReader("C:\meineTextdatei.txt")
        meinDateiInhalt = dateiStromLeser.ReadToEnd()
```

```
        dateiStromLeser.Close()
        Debug.Print(meinDateiInhalt)
    Catch ex As FileNotFoundException
        'Hier werden nur FileNotFoundExceptions abgefangen
        MessageBox.Show("Datei wurde nicht gefunden!" & vbNewLine & _
                        vbNewLine & "Der Ausnahmetext lautete:" & vbNewLine & ex.Message, _
                        "Ausnahme", MessageBoxButtons.OK, MessageBoxIcon.Error)
    Catch ex As Exception
        'Hier werden alle anderen bis zu diesem Zeitpunkt nicht
        'behandelten Ausnahmen abgefangen
            'Hier werden alle anderen bis zu diesem Zeitpunkt nicht
            'behandelten Ausnahmen abgefangen
            MessageBox.Show("Beim Verarbeiten der Datei trat eine Ausnahme auf!" & vbNewLine & _
                            "Die Ausnahme war vom Typ:" & ex.GetType.ToString & vbNewLine & _
                            vbNewLine & "Der Ausnahmetext lautete:" & vbNewLine & ex.Message, _
                            "Ausnahme", MessageBoxButtons.OK, MessageBoxIcon.Error)
    End Try
End Sub
```

Was passiert hier? Alle Anweisungen, die zwischen der `Try`- und der ersten `Catch`-Anweisung platziert sind, befinden sich in einer Art »Versuchsmodus«. Tritt bei der Ausführung einer dieser Anweisungen ein Fehler auf, springt das Programm automatisch in den »zutreffenden« `Catch`-Block. Und was bedeutet dabei »zutreffend«?

Das Ausnahmenabfangen ist nicht nur auf einen Ausnahmetyp beschränkt

Wenn Sie in den Editor einfach nur die Zeichenfolge »`Try`« eingeben und anschließend ↵ drücken, fügt dieser automatisch den folgenden Block ein:

```
Try

Catch ex As Exception

End Try
```

`Exception` lautet die in der Vererbungshierarchie zuoberst stehende Klasse, deren Instanzen (hier durch `ex` referenziert) ausnahmslos alle Ausnahmen abfangen können. Doch es könnte, wie im Beispielcode zu sehen, sinnvoll sein, zwischen den vielen verschiedenen Ausnahmetypen zu differenzieren. Ihr Programm soll nämlich vielleicht auf eine Ausnahme, die durch eine nicht vorhandene Datei ausgelöst wird, anders reagieren, als auf eine Datei, die zwar vorhanden, aber gerade mit einem anderen Programm verarbeitet wird.

Sie können das am Beispielcode nachvollziehen. Wenn Sie diesen starten, wird eine Ausnahme ausgelöst, die sich `FileNotFoundException` nennt. Ein `Catch` mit diesem Klassennamen als Argument fängt nur Ausnahmen dieses Typs ab. Dementsprechend wird der darunter stehende Code ausgeführt, und der Code sorgt dafür, dass ein entsprechender Dialogs angezeigt wird.

Abbildung 2.19 Im Beispielcode wird diese Meldung ausgegeben, wenn die Datei nicht vorhanden war, also eine *FileNotFoundException* vorlag

Wenn Sie nun aber auf Laufwerk C: eine Textdatei mit diesem Namen anlegen, und diese Datei anschließend beispielsweise in Microsoft Word öffnen, dann produziert die Zeile

```
dateiStromLeser = New StreamReader("C:\meineTextdatei.txt")
```

abermals eine Ausnahme – doch dieses Mal keine vom Typ `FileNotFoundException` sondern vom Typ `IOException`:

Abbildung 2.20 Jetzt ist die Datei zwar vorhanden, aber durch Word bereits geöffnet

Und wieso wird diese Ausnahme durch `Catch as Exception` abgefangen, obwohl sie vom Typ `IOException` ist? Das liegt daran, dass `IOException` auf `Exception` basiert. In der objektorientierten Programmierung können durch Vererbung aus Klassen erweiterte Klassen entstehen, aus diesen wiederum nochmals spezialisierte, und so fort. Genau so ist das bei Ausnahmeklassen. `Exception` selbst ist die Basisklasse. Davon abgeleitet ist die Ausnahmeklasse `SystemException`. Und davon wiederum ist `IOException` abgeleitet. Da wir im Beispiel aber `IOException` nicht gesondert behandeln, wird der `Catch`-Block angesteuert (so vorhanden), der zumindest eine der Basisklassen der aufgetretenen Ausnahmeklassen repräsentiert. Bei der Verwendung von `Exception` sind Sie dabei immer auf der sicheren Seite, denn auf dieser basieren *alle* anderen Ausnahmeklassen. Und so wird zwar die Ausnahme `FileNotFoundException` gesondert behandelt, aber bei allen anderen Ausnahmen der `Catch`-Block ausgeführt, der mit `Exception` selbst arbeitet.

WICHTIG Sie können in einem `Try-Catch`-Block so viele `Catch`-Zweige implementieren, wie Sie möchten, und damit alle denkbaren Ausnahmetypen gezielt abfangen. Achten Sie dabei aber darauf, dass Sie die spezialisierteren Ausnahmetypen nach oben stellen, und die, auf denen diese basieren, erst weiter unten kommen. Anderenfalls handeln Sie sich Ärger mit dem Compiler ein, der das schlichtweg nicht zulässt. Denn eine Basisausnahmeklasse würde in diesem Fall eine spezialisierte Ausnahme schon längst verarbeitet haben, bevor der `Catch`-Block mit der spezialisierteren Ausnahme zum Zuge käme. Und diesen »Un-Sinn« unterbindet der Compiler von vornherein:

```
Try
    'Diese folgenden Befehle probieren (try=versuchen, ausprobieren).
    dateiStromLeser = New StreamReader("C:\eineTextdatei.txt")
    meinDateiInhalt = dateiStromLeser.ReadToEnd()
    dateiStromLeser.Close()
    Debug.Print(meinDateiInhalt)
Catch ex As Exception
    'Hier werden alle anderen bis zu diesem Zeitpunkt nicht
    'behandelten Ausnahmen abgefangen
    MessageBox.Show("Beim Verarbeiten der Datei trat eine Ausnahme auf!" & vbNewLine & _
                    "Die Ausnahme war vom Typ:" & ex.GetType.ToString & vbNewLine & _
                    vbNewLine & "Der Ausnahmetext lautete:" & vbNewLine & ex.Message, _
                    "Ausnahme", MessageBoxButtons.OK, MessageBoxIcon.Error)
Catch ex As FileNotFoundException
Der Catch-Block wird niemals erreicht, da "System.IO.FileNotFoundException" von "System.Exception" erbt.efangen
    MessageBox.Show("Datei wurde nicht gefunden!" & vbNewLine & _
                    vbNewLine & "Der Ausnahmetext lautete:" & vbNewLine & ex.Message, _
                    "Ausnahme", MessageBoxButtons.OK, MessageBoxIcon.Error)
End Try
```

Abbildung 2.21 Catch-Blöcke mit spezialisierteren Ausnahmeklassen müssen vor den Basisausnahmeklassen platziert werden, sonst meckert der Compiler

Und wozu dient Finally?

Programmcode, den man in einem `Finally`-Block platziert, *wird in jedem Fall ausgeführt*. Auch dann, wenn ein Fehler auftritt, mit `Catch` abgefangen wird und die Anweisung ergeht, die gesamte Prozedur beispielsweise mit `Return` *eigentlich* noch im `Catch`-Block zu verlassen.

Denn normalerweise ist ja `Return` der letzte Befehl in einer Prozedur, ganz gleich, wo `Return` platziert wurde. Nach `Return` ist Schicht. Doch in einigen Fällen ist das nicht sinnig, gerade wenn es um Fehlerbehandlungen geht, und deswegen bildet `Finally` in diesem Fall die goldene Ausnahme.

Angenommen, Sie lesen aus einer Datei, und diese Datei haben Sie dazu vorher geöffnet. Beim Öffnen der Datei ist zwar kein Fehler aufgetreten, aber beim Lesen – vielleicht ist der Fall eingetreten, dass Sie über das Dateiende hinaus gelesen haben, und nun sind Zeilen, die Sie verarbeiten wollen, leer (also `Nothing`). Diesen Fall (das Abfangen einer Null-Referenz) haben Sie mit einem entsprechenden `Catch`-Block behandelt, und da es nichts Weiteres zu tun gibt, möchten Sie nach der Anzeige einer Fehlermeldung Ihre Leseroutine mit `Return` direkt aus dem `Catch`-Block heraus verlassen. Das Problem: die Datei ist noch geöffnet, und Sie sollten sie schließen. Mit `Finally` lässt sich ein solcher Vorgang elegant implementieren.

Das folgende Beispiel simuliert einen solchen Prozess. Es liest »`c:\eineTextdatei.txt`« zeilenweise aus und wandelt die einzelnen Zeilen in Großbuchstaben um, bevor es sie zu einem Gesamttextblock zusammenführt. Doch da es viel zu viele Zeilen liest, funktioniert die `ToUpper`-Methode des Strings, die die Zeile in Großbuchstaben umwandeln soll, irgendwann nicht mehr, denn die `ReadLine`-Funktion greift ins Leere und liefert `Nothing` zurück:

```
Private Sub btnTryCatchFinally_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
Handles btnTryCatchFinally.Click
    'WICHTIG:
    'Um auf die IO-Objekte zuzugreifen, muss "Imports System.IO"
    'am Anfang der Codedatei stehen!

    Dim meinDateiInhalt As String
    Dim dateiStromLeser As StreamReader
```

```
    Try
        'Diese folgenden Befehle probieren (try=versuchen, ausprobieren).
        dateiStromLeser = New StreamReader("C:\eineTextdatei.txt")
        Dim locZeile As String
        meinDateiInhalt = ""
        'Jetzt lesen wir zeilenweise aber viel zu viele Zeilen,
        'und schießen daher irgendwann über das Ende der Datei hinaus:
        Try
            ' Wenn Ihre Datei "C:\eineTextdatei" nicht gerade
            ' 1001 Zeilen enthält, knallt es hier, denn:
            For zeilenzähler As Integer = 0 To 1000
                locZeile = dateiStromLeser.ReadLine()
                'locZeile ist jetzt Nothing, und dann kann
                'die Konvertierung in Großbuchstaben nicht mehr
                'funktionieren.
                locZeile = locZeile.ToUpper
                meinDateiInhalt &= locZeile
            Next
        Catch ex As NullReferenceException
            MessageBox.Show("Die Zeile konnte nicht in umgewandelt werden, weil sie leer war!")
            ' Return? Aber die Datei ist doch noch geöffnet!!!
            Return
        Finally
            ' Egal! Auch bei Return im Catch-Block wird Finally in jedem Fall noch ausgeführt!
            dateiStromLeser.Close()
        End Try
        'Aber diese Zeile wird nur im Erfolgsfall abgearbeitet:
        Debug.Print(meinDateiInhalt)
    Catch ex As FileNotFoundException
        'Hier werden nur FileNotFoundExceptions abgefangen
        MessageBox.Show("Datei wurde nicht gefunden!" & vbNewLine &
                        vbNewLine & "Der Ausnahmetext lautete:" & vbNewLine & ex.Message,
                        "Ausnahme", MessageBoxButtons.OK, MessageBoxIcon.Error)
    Catch ex As Exception
        'Hier werden alle anderen bis zu diesem Zeitpunkt nicht
        'behandelten Ausnahmen abgefangen
        MessageBox.Show("Beim Verarbeiten der Datei trat eine Ausnahme auf!" & vbNewLine &
                        "Die Ausnahme war vom Typ:" & ex.GetType.ToString & vbNewLine &
                        vbNewLine & "Der Ausnahmetext lautete:" & vbNewLine & ex.Message, _
                        "Ausnahme", MessageBoxButtons.OK, MessageBoxIcon.Error)
    End Try
End Sub
```

TIPP Sie können diesen Vorgang übrigens gut nachvollziehen, indem Sie einen Haltepunkt mit F9 in der Zeile setzen, die `Return` beinhaltet, und das Programm starten. Wenn Sie anschließend mit F11 schrittweise durch das Programm steppen, werden Sie feststellen, dass nach dem `Return` der Code im `Finally`-Block noch abgehandelt wird.

Kapitel 3

Einführung in .NET Framework

In diesem Kapitel:

Was ist .NET und aus was besteht es? 66
Was ist eine Assembly? 67

.NET bzw. .NET Framework gehören in der IT-Branche zu den am wenigsten verstandenen Begriffen. Und daran ist Microsoft nicht ganz unschuldig, denn offensichtlich haben die Marketingmeister diesen Begriff seinerzeit ebenfalls so gründlich nicht verstanden, dass sie ihn für alles und jeden Zweck verwendet haben, der ihnen nur in den Sinn kommen konnte, als sich die erste Version von .NET Framework ihrer ersten Einsatzfertigkeit näherte.

So hieß die erste Version von Windows Server 2003 noch .NET Server, und das Einzige, was man hier in Sachen .NET finden konnte, war ein vorinstalliertes .NET Framework. Das UI (*User Interface*, *Benutzerschnittstelle*) des Betriebssystems selbst stützte sich mit keiner Codezeile auf .NET Framework, und außer, dass .NET Framework in der damaligen Version 1.1 vorinstalliert war, existierten nur die Internet Information Services, die auf Basis dieser .NET Framework-Version das Hosten von ASP.NET-Websites zuließ. Immerhin.

Fragte man zu der damaligen Zeit, was .NET eigentlich sei und aus welchen Komponenten es bestehe, bekam man alle möglichen Antworten, und nicht unbedingt welche, die der Wahrheit sehr nahe kamen.

O.k. – dann wollen wir damit mal beginnen, was also was ist .NET?

Was ist .NET und aus was besteht es?

.NET ist eine Sammlung verschiedener Technologien, die auf einer gemeinsamen Infrastruktur beruhen und die Softwareentwicklern die Möglichkeit bieten, sichere, stabile, einfach zu pflegende, auf mehreren Plattformen lauffähige Software sowohl webbrowserorientiert als auch als Smart Client zu entwickeln.

Es gibt eine ganze Menge Begriffe im Zusammenhang mit .NET Framework, mit denen gerade auf Fachtagungen, Entwicklerkonferenzen und unter Entwicklerkollegen gerne herumgeworfen wird – oft ohne dass den Werfern die Bedeutung so richtig klar zu sein scheint. Selbst erfahrenen Programmierern erschließt sich der Hintergrund bestimmter .NET-Fachbegriffe auf Anhieb nicht ohne Weiteres, und die Erfahrung zeigt, dass es auch eine ganze Menge Entwickler gibt, die bestenfalls über Halbwissen verfügen und von denen man entweder nur Halbwahrheiten oder sogar völlig falsche Informationen bekommt.

Und dann gibt es natürlich noch eine besonders erwähnenswerte Eigenschaft der ganzen .NET Framework-Plattform, die das Kompilieren von Anwendungen betrifft. Im Gegensatz zu herkömmlichen Programmiersprachen produziert nämlich keiner der .NET-Sprachen-Compiler, und damit auch nicht der Visual Basic-Compiler, direkt vom Prozessor ausführbare Maschinenbefehle, sondern einen Zwischencode, der erst zur Laufzeit der Anwendung in nativen Prozessorcode umgesetzt wird. Damit werden Anwendungen einerseits plattformunabhängig und andererseits können Optimierungen auf das jeweilige System vorgenommen werden, was nicht möglich wäre, würde das Programm schon im Vorfeld in Maschinensprache übersetzt.

Die folgenden Abschnitte sollen deshalb die wichtigsten .NET-Fachbegriffe und -Technologien vorstellen und kurz erklären. Und keine Angst: Auch wenn einen die vielen Abkürzungen und Akronyme anfangs verwirren, so ist das dahinter stehende Konzept genial, schlüssig und im Grunde genommen auch ganz einfach zu verstehen. Und spätestens, wenn Sie die folgenden Abschnitte gelesen haben, werden Sie einen roten Faden erkennen, der sich elegant durch das ganze .NET-Konzept zieht.

Was ist eine Assembly?

Wenn Sie ein Programm unter Windows schreiben, dann wandelt ein Compiler im einfachsten Fall Ihren Quellcode schlussendlich in eine ausführbare .EXE-Datei um. Bei größeren Projekten bietet es sich an, Funktionen, die von einzelnen Teilprogrammen immer wieder verwendet werden, in so genannten DLLs[1] zusammenzufassen und von außen aufzurufen – einfach um Redundanz durch doppelte Funktionen zu vermeiden und letztendlich Platz zu sparen. Eine DLL kann man sich als eine Art Bibliothek (daher auch der Name) vorstellen, die zwar alleine nicht lauffähig ist, die in ihr enthaltenen Funktionen aber beliebig vielen anderen EXE-Dateien zur Verfügung stellt.

Unter .NET funktioniert das im Großen und Ganzen genauso. Ein entscheidender Unterschied ist jedoch, wie die .EXE-Dateien bzw. DLLs tatsächlich vorliegen (dazu liefert der kommende *JITter*-Abschnitt tiefere Einblicke) und mit welchem Oberbegriff diese bezeichnet werden: nämlich als *Assembly*.

Eine ausführbare .NET-EXE-Datei ist eine Assembly. Eine .NET-DLL ist auch eine Assembly. Eine Assembly stellt also im Grunde genommen nichts weiter als eine direkt (EXE) oder indirekt (DLL) ausführbare Einheit dar, die Programmcode enthält. Eine Assembly ist folglich zunächst die kleinste auslieferbare Einheit einer .NET Programmierung.

Hinter dem Konzept von Assemblys verstecken sich zugegebenermaßen noch kompliziertere Konzepte und weitaus mehr Möglichkeiten. Doch für den täglichen Umgang mit .NET-Technologien ist die hier gegebene Beschreibung völlig ausreichend.

Was ist ein Namespace?

Namespaces dienen in erster Linie zur thematischen Ordnung von Klassen innerhalb von Assemblys. Namespaces haben überhaupt keinen Einfluss auf den Namen einer Assembly, sondern nur auf den voll qualifizierten Namen einer Klasse oder Struktur *innerhalb* einer Assembly. Namespaces haben ferner keinen Einfluss auf Dateinamen (anders als Assemblys), sie sind also eine abstrakte Größe.

Die Definition von Namespaces dient also in erster Linie genau zu dem, wozu Kapitel in einem Buch dienen. Würden die einzelnen Abschnitte eines Buchs nur »lose im Raum« stehen, litte die Übersicht darunter ganz gewaltig. Genauso verhält es sich bei Objekten. Der Name des Objekts `AdressenDetails` beispielsweise sagt nur ungefähr etwas darüber aus, welchem Zweck es dient. Befindet sich das Objekt im Namespace `MeineFibu.Lieferanten`, so ist eine schon größere Ordnung hergestellt – das Wiederfinden des »richtigen« Objekts wird einfacher und auch die Möglichkeit der Existenz eines weiteren Objekts namens `AdressenDetails` ist ohne Mehrdeutigkeitsprobleme denkbar, wenn es sich in einem anderen Namespace wie `MeineFibu.Kunden` befindet.

Namespaces können sich durchaus über mehrere Assemblys erstrecken. Es ist also denkbar, zwei Klassen des Namespaces `MeineFibu.Lieferanten` in einer Assembly und zwei weitere Klassen des gleichen Namespaces in einer anderen Assembly unterzubringen. Namespaces und Assemblys sind, was das anbelangt, voneinander völlig unabhängig.

---

[1] *Dynamic Link Libraries*, etwa: *dynamisch verbindbare Bibliotheken.*

Die FCL, die alle Klassen von .NET Framework enthält (genaue Erklärung folgt), macht von Namespaces regen Gebrauch. Klassen für die Formularsteuerung befinden sich beispielsweise im Namespace `System.Windows.Forms`. Dieser Namespace ist aber nicht nur in der Assembly `System.Windows.Forms` definiert, in der sich die meisten und wichtigsten Objekte für die Formularsteuerung befinden. Hier ist es nur durch die Entscheidung der Microsoft-Programmierer so geschehen, dass Namespace und Assembly gleiche Namen tragen. Das muss aber, wie schon gesagt, nicht so sein: Andere Objekte des gleichen Namespaces (`System.Windows.Forms`) sind beispielsweise auch in der Assembly *System.dll* vorhanden.

HINWEIS Namespaces ermöglichen zwar einerseits das »saubere« Definieren von Klassen mit gleichem Klassennamen; dabei kann es unter Umständen aber auch passieren, dass ein bereits importierter Namespace eine Klasse oder Struktur des `System`-Namespaces verdeckt und Sie deswegen nicht mehr darauf zugreifen können. Das `Global`-Schlüsselwort sorgt dann für Abhilfe.

Einbinden von Namespaces und Assemblys in Codedateien und Projekten am Praxisbeispiel

Und nachdem wir uns die Theorie um Namespaces und Assemblys einverleibt haben, schauen wir uns an, wie das Ganze in der Praxis aussieht. Dazu folgendes Szenario: Wir erstellen eine Konsolenanwendung (*Datei/Neu/Projekt* und dann aus dem Dialog, der jetzt erscheint, unter *Visual Basic* und *Windows* die Vorlage *Konsolenanwendung* auswählen. Nennen Sie das Beispielprojekt etwa **NamespaceDemo**), von der aus wir eine `MessageBox`, also einen typischen Windows-Nachrichtendialog aufrufen wollen.

Wenn Sie versuchen, den Namen der `MessageBox`-Klasse einzugeben, quittiert der Compiler diesen Versuch mit einer Fehlermeldung, etwa wie in Abbildung 3.1 zu erkennen.

```
Module1                                                    Main
1  Module Module1
2
3      Sub Main()
4
5          MessageBox.Show("Ein Nachrichtenfeld anzeigen")
6          "MessageBox" wurde nicht deklariert. Auf das Objekt kann aufgrund seiner Schutzstufe möglicherweise nicht zugegriffen werden.
7      End Sub
8
9  End Module
10
```

Abbildung 3.1 Eine Methode kann nur auf die Objekte zugreifen, die ihre eigene Assembly selbst definiert oder die in Assemblys liegen, auf die sie verweist

Dass der Compiler `MessageBox` nicht kennt, liegt einfach daran, dass die `MessageBox`-Klasse in einer Assembly von .NET Framework definiert wird, die standardmäßig nicht in Konsolenanwendungen eingebunden ist. Ihre Aufgabe ist es in solchen Fällen deshalb, einen Verweis auf die Assembly zu setzen, die die entsprechende Klasse enthält – in unserem Fall ist das die Assembly *System.Windows.Forms.dll*.

1. Bewegen Sie dazu den Mauszeiger auf den Projektnamen (*nicht* den Projektmappen-Namen; Positionieren Sie den Mauszeiger also in Verbindung mit einer Projektmappe, die ein einzelnes Projekt enthält, auf den *zweiten* Eintrag von oben).

2. Drücken Sie die rechte Maustaste, um das Kontextmenü zu öffnen.

Abbildung 3.2 Sie fügen Ihrem Projekt einen Verweis auf eine zusätzliche Assembly hinzu, indem Sie im Projektmappen-Explorer das Kontextmenü öffnen und Verweis hinzufügen anklicken...

3. Wählen Sie den Eintrag *Verweis hinzufügen* aus (Abbildung 3.2).
4. Im Dialog, der jetzt erscheint, wählen Sie auf der ersten Registerkarte den Eintrag *System.Windows.Forms* aus, wie in Abbildung 3.3 zu sehen.

Abbildung 3.3 In diesem Dialog wählen Sie die Assembly aus, auf die Ihr Projekt (und damit die aktuelle Assembly) verweisen soll

5. Klicken Sie auf *OK*, um den Dialog zu verlassen.

An der Fehlermeldung hat sich nichts geändert, aber immerhin haben Sie jetzt die Möglichkeit, auf die `MessageBox`-Klasse zuzugreifen. Der Namespace vieler Klassen lautet nämlich oftmals genauso wie die Assembly selbst – auf jedem Fall ist das in unserem Beispiel so. Wenn Sie also den `MessageBox`-Klassennamen durch seinen voll qualifizierten Namen austauschen (das ist der, der die Namespace-Angabe mit enthält), ist die Fehlermeldung schon einmal ausgemerzt, wie Abbildung 3.4 zeigt.

```
Module Module1

    Sub Main()

        System.Windows.Forms.MessageBox.Show("Ein Nachrichtenfeld anzeigen")

    End Sub

End Module
```

Abbildung 3.4 Mit dem voll qualifizierten Namen können Sie eine Klasse in einer per Verweis referenzierten Assembly *immer* erreichen!

Natürlich würden Sie sich die Finger wund tippen, müssten Sie beim Verwenden einer Methode der Klassen einer externen Assembly ständig deren voll qualifizierten Namen ausschreiben, zumal Sie in der Regel nicht nur eine Klasse, sondern viele Klassen einer eingebundenen Assembly verwenden. Daher erlaubt es die `Imports`-Anweisung, einen Namespace für eine Codedatei global zu importieren. Wie diese funktioniert, zeigt Abbildung 3.5.

```
Imports System.Windows.Forms
Module Module1

    Sub Main()

        MessageBox.Show("Ein Nachrichtenfeld anzeigen")

    End Sub

End Module
```

Abbildung 3.5 Mit der *Imports*-Anweisung binden Sie einen Namespace global für eine Codedatei ein

Aber auch das kann auf Dauer zu einer zusätzlichen Tippbelastung führen, deswegen kennt Visual Basic auch die Möglichkeit, Namespaces global für ein Projekt einzubinden:

1. Dazu öffnen Sie abermals, wie unter Abbildung 3.2 gezeigt, das Kontextmenü des Projekts.
2. Wählen Sie aber dieses Mal den Menüpunkt *Eigenschaften* aus.
3. Im Dialog, den Visual Studio jetzt als Registerkartendokument öffnet, wechseln Sie auf die Registerkarte *Verweise*. Auf dieser Registerkarte finden Sie einerseits die Verweise auf alle Assemblys, die Ihrem Projekt hinzugefügt wurden, und deren Klassen Sie deswegen verwenden können, sowie andererseits eine Liste mit den vorhandenen und vorimportierten Namespaces, die diese Assemblys zur Verfügung stellen. Die Assemblys, vor denen Sie ein Häkchen sehen (Abbildung 3.6), sind vorimportiert – auf deren Klassen können Sie also ohne Weiteres zugreifen. Damit ersparen Sie sich die entsprechenden `Imports`-Anweisungen am Anfang der betroffenen Codedateien.

Abbildung 3.6 In den Projekteigenschaften finden Sie auf der Registerkarte *Verweise* eine Referenz aller Assemblys sowie der Namespaces, die diese Assemblys zur Verfügung stellen. Die angeklickten Namespaces werden automatisch vorimportiert

Was versteckt sich hinter CLR (Common Language Runtime) und CLI (Common Language Infrastructure)?

Die *Common Language Runtime* besteht unter anderem aus der Basisassembly von .NET Framework (die Assembly selbst nennt sich *mscorlib.dll*). Sie bildet den unteren Layer, sozusagen das Fundament, auf dem alle anderen Objekte von .NET Framework ruhen. Diese Assembly wird als *Base Class Library* (*BCL*; siehe unten) bezeichnet. Diese BCL ist aber nur ein Teil der CLR, denn Letztere hat weitere, genauso wichtige Aufgaben. So stellt sie beispielsweise die *JITter*-Funktionalität (siehe unten), die dafür sorgt, dass .NET-Assemblys, die zunächst in der *Intermediate Language* vorliegen (M[S]IL)[2], kurz vor der Ausführung in nativen Assemblercode kompiliert werden (*JIT* steht für ***J**ust **i**n **T**ime*, etwa: *genau rechtzeitig*).

Eine ebenfalls sehr wichtige Rolle übernimmt die CLR in Form der .NET Framework-eigenen »Müllabfuhr«, dem so genannten *Garbage Collector*, den sie ebenfalls implementiert. Anders als bei C oder C++ müssen Sie sich bei der Entwicklung mit .NET Framework nicht um das Aufräumen Ihres Datenmülls kümmern, um Objekte also, die Sie verwendet haben und die Sie ab einem bestimmten Zeitpunkt nicht mehr benötigen. Der Garbage Collector stellt selbstständig fest, ob ein Objekt in Ihrem Programmcode noch benötigt (referenziert) wird und entsorgt es, wenn es nicht mehr verwendet wird.

[2] Einige Microsoftler verwenden die Abkürzung *MSIL*, einige *MIL* und einige nur *IL*.

Schließlich sorgt die CLR mithilfe ihrer *Execution Engine*, die sich in der DLL *mscoree.dll* verbirgt, dass Ihre .NET-Programme überhaupt zur Ausführung kommen. Aufgabe der Execution Engine ist es, die vergleichsweise neue .NET-Technologie zusammen mit dem Vorgang des Just-in-Time-Kompilierens an die Startvorgänge eines herkömmlichen Programms unter Windows anzupassen.

Die CLR selbst basiert auf der so genannten CLI (der *Common Language Infrastructure*) – »der Spezifikation eines internationalen Standards für das Erstellen von Entwicklungs- und Programmausführungsumgebungen«, einem Standard, der definiert, dass verschiedene Programmiersprachen (respektive der Code, den sie generieren) und verschiedene Bibliotheken nahtlos zusammenarbeiten können. Dieses Konzept ermöglicht es, dass Sie unter .NET Projekte entwickeln und dabei mit verschiedenen Programmiersprachen gleichzeitig arbeiten können. So wäre es beispielsweise denkbar, finanztechnische Programmbibliotheken in Visual Basic zu formulieren und diese von Ihrer eigentlichen Windows-Anwendung aufzurufen, die Sie in C# entwickelt haben.

Was ist die FCL (Framework Class Library) und was die BCL (Base Class Library)?

Erst einmal der Ursprung für einen Haufen falscher Erklärungen. Es gibt Experten, die meinen, die FCL und BCL wären dasselbe. Und ist das richtig? Nein. Andere differenzieren schon detaillierter und erklären die BCL zur Oberkategorie der CLR. Haben sie Recht? Nein.

Die BCL ist *Teil* der Common Language Runtime und enthält alle Basisobjekte, die Sie während Ihres Entwicklungsalltags benötigen. Sie definiert unter anderem alle primitiven Datentypen und implementiert damit das *Common Type System* (CTS; siehe unten) in Form von verwendbaren Objekten und Typen, von denen sich die meisten im `System`-Namespace bzw. in der Assembly *mscorlib.dll* befinden. Objekte und Methoden aus der BCL werden Sie beim Entwickeln unter .NET wohl am häufigsten verwenden. Egal, ob Sie eine Variable eines primitiven Datentyps verwenden, große Datenmengen in Arrays speichern, mit Zeichenketten hantieren oder reguläre Ausdrücke verwenden – die dafür benötigten Klassen und Methoden befinden sich alle in der BCL.

Die Base Class Library ist weitestgehend plattformunabhängig formuliert; sie ist standardisiert (denn sie basiert auf der durch die ECMA zertifizierten CLI), und deswegen ist ihr Quellcode gegenwärtig in der Version 1.0 im Rahmen der freien CLI-Implementierung namens *Rotor* auch frei verfügbar und vergleichsweise leicht portierbar. Lauffähige Implementierungen der CLI gibt es derzeit unter MacOS. FreeBDS und – natürlich – Windows.[3] Das heißt im Klartext:

- Die BCL ist das CLI-Pendant unter Windows und damit Teil der CLR
- Die FCL fasst *alle* .NET-Funktionsbereiche unter einem Namen zusammen

[3] Das Projekt »Mono« unter Linux geht übrigens noch einen ganzen Schritt weiter und implementiert eine komplette FCL, die in vielen Bereichen sogar kompatibel zur Microsoft FCL ist.

Was ist das CTS (Common Type System)?

Das CTS bildet eine Richtlinie für die Implementierung von Datentypen und Datentypenkonzepten unter der CLI. Um es mit den bisher vorgestellten Akronymen zu sagen: Die BCL der CLR setzt das CTS unter Beachtung der CLI um (wenn das kein Satz zum Angeben ist! Vor der Benutzung im Rahmen romantischer Kontakte zur Darstellung der eigenen Fähigkeiten ist aber dringend abzuraten).

Im Klartext heißt das: Das Common Type System definiert, wie Typen deklariert, verwendet und in der CLR gemanagt werden, und sie spielt darüber hinaus einen wichtigen Part bei der Integration der verschiedenen .NET-Sprachen durch die CLR. Sie realisiert das durch unbedingte Typsicherheit (Sie können nicht ohne Weiteres einer Integervariablen eine Zeichenkette zuweisen) und garantiert eine hohe Performance bei der Codeausführung. Schließlich definiert sie einen festen Satz an Richtlinien, denen die verschiedenen .NET-Sprachen folgen müssen, und sie stellt damit sicher, dass Objekte, die in der einen .NET-Sprache entwickelt worden sind, sich in einer anderen .NET-Sprache verwenden lassen.

Dies ist ein gewaltiger Fortschritt gegenüber früheren Technologien von Microsoft (wie zum Beispiel COM) zum Datenaustausch zwischen unterschiedlichen Programmen und Programmierumgebungen. Wenn früher ein Visual Basic-Programm (unter Visual Basic 6.0) eine Zeichenfolge (`string`) an eine Anwendung übergeben wollte, die beispielsweise in C/C++ geschrieben worden war (was sofort der Fall war, wenn man in Visual Basic Funktionen von Windows nutzen wollte), musste man sehr genau verstehen, was man tat – eine Zeichenfolge in C ist etwas anderes als eine Zeichenfolge in Visual Basic, sonst kam es zu merkwürdigen Effekten bis hin zum Absturz der Anwendung.

Durch das Konzept des Umsetzens dieser Richtlinien ergibt sich »ganz nebenbei« ein weiterer, wirklich nicht unwesentlicher Vorteil, den Sie im folgenden Abschnitt beschrieben finden.

Was ist CIL/MSIL (Microsoft Intermediate Language) und wozu dient der JITter?

Auch bedingt durch das Konzept und die Reglementierung, die das CTS vorschreibt, übersetzen die Compiler der verschiedenen .NET-Programmiersprachen ihren Quellcode nicht direkt in Maschinensprache, der vom Prozessor eines Computers verstanden wird. Vielmehr werden Ihre Programme zunächst in eine Zwischensprache umgewandelt – in die so genannte *Common Intermediate Language* oder kurz: *CIL*). »Zwischensprache« deshalb, weil sie einerseits schon abstrakter und eine Ebene höher als eine Prozessormaschinensprache angesiedelt ist, andererseits dennoch viel prozessornäher als eine vollwertige Hochsprache konzipiert ist, wie beispielsweise Visual Basic oder C#. Auch die CIL beschreibt wieder in Anlehnung an die CLI das grundsätzliche Konzept und stellt Technologieträger dar; die konkrete Technologie, die von Microsoft zur Auslieferung für die Windows-Plattform kommt, nennt sich *MSIL* (*Microsoft Intermediate Language*).

Eine Assembly enthält also in der Regel keine Befehle, die ein Prozessor direkt verstehen könnte, sondern Programmcode, aus dem erst der so genannte *Just-in-Time-Compiler* (der JITter) zur Laufzeit das eigentliche Maschinenprogramm erzeugt. Das hört sich zunächst nach einem möglichen Performanceproblem an, ist es aber nicht.[4] Der JITter ist so optimiert, dass er nur jeweils die Methoden einer Klasse kompiliert, die

[4] O.k., na gut, sagen wir: In den seltensten Fällen. O.k., na gut, sagen wir: In der Regel nicht.

als Nächstes benötigt werden. Und einiges an Zeit, das zunächst durch das Kompilieren der Methoden geopfert werden muss, fahren Ihre Programme schlussendlich wieder ein, da der JITter viel effizienteren Code als jeder andere Compiler erzeugen kann, denn:

Der JITter kennt den Computer, auf dem der zu kompilierende Code laufen wird. Ein herkömmlicher Compiler kennt ihn nicht, denn der Computer, auf dem eine Anwendung entwickelt und auch schon kompiliert wird, ist üblicherweise ein ganz anderer als der, auf dem diese laufen wird. Der JITter kann also auf die Besonderheiten eines Prozessors eingehen. Erkennt er beispielsweise, dass sich in dem Computer, auf dem eine .NET-Anwendung laufen soll, ein Pentium 4-Prozessor befindet, kann er den zu erzeugenden Code für diesen Prozessor optimieren. Ein herkömmlicher Compiler jedoch muss immer Maschinencode erzeugen, der nur durchschnittlich gut optimiert ist, bei dem aber gewährleistet ist, dass er auf allen kompatiblen Prozessoren (P3, P4, Athlon etc.) funktioniert. Es gibt eine Vielzahl weiterer Optimierungsmöglichkeiten für den JITter, auf die ich an dieser Stelle nicht näher eingehen will.

HINWEIS Eine Ausnahme bildet übrigens die FCL selbst. Sie ist vorkompiliert, sodass nicht die Notwendigkeit besteht, auch noch das komplette .NET Framework (oder zumindest die Teile, die eine Anwendung benötigt) bei jedem Start der Anwendung zu *JITten*. Damit könnte der Eindruck entstehen, dass die FCL, anders als Ihre eigenen Programme, vielleicht nicht optimal auf Ihr System »eingestellt« ist – denn wäre sie schon bei Microsoft hausintern vorkompiliert, könnte sie ja unmöglich auf Ihr System optimiert sein. Dem ist aber nicht so, denn: Möglicherweise ist Ihnen im Laufe Ihrer Arbeit mit .NET schon aufgefallen, dass die Installation des zig MB großen .NET Framework selbst auf einem flotten Computer zwar immer noch schnell vonstatten geht, aber auffällig länger als erwartet dauert. Das liegt daran, dass die FCL bei ihrer Installation nicht nur in die entsprechenden Verzeichnisse *kopiert*, sondern quasi dort hinein*kompiliert* wird. Die Installationsdateien der FCL im .NET Framework-Setup liegen nämlich als MSIL vor. Erst wenn Sie sie auf einem Computer installieren, werden sie in nativen Maschinencode übersetzt und dann – optimiert auf Ihren Computer – in den Zielverzeichnissen eingerichtet.

Und einen weiteren Vorteil hat diese Vorgehensweise, denn: Programme, die Sie unter .NET Framework entwickeln, werden prozessorunabhängig. Eine Anwendung, die Sie mit dem Zieltyp *Any* erstellen (etwa: *zielt ab auf eine beliebige durch .NET unterstützte Architektur*) läuft auch prozessorunabhängig. Entwickeln Sie also ein Projekt mit dem Zieltyp *Any*, wird es beispielsweise unter einem 32-Bit-Windows XP laufen, aber auf einem 64-Bit-Windows Vista auch mit der vollen 64-Bit-Unterstützung.

Kapitel 4

Rundgang durch Visual Studio

In diesem Kapitel:

Einführung 76
Visual Studio das erste Mal starten – das Profil auswählen 77
Die Startseite – der Ausgangsort für Ihre Entwicklungsaufgaben 79
Aus früheren Visual Studio-Versionen auf Visual Studio 2010 migrieren 80
Anwendungen für mehrere .NET Framework-Versionen (Multizielversionen) unter Visual Studio 2010 80
Ein näherer Blick auf die neue und verbesserte WPF-basierte IDE 80
Code: navigieren, suchen und nachvollziehen 80
Regionen und Gliederungen 80
Architektur-Explorer 80
Generate From Usage 80
Visual Studio erweitern 80

Einführung

Die Intention dieses Kapitels ist es, Ihnen einen Überblick über die integrierte Entwicklungsumgebung von Visual Studio 2010 zu geben, die kurz als IDE (*Integrated Development Environment*) bezeichnet wird. Obwohl die Funktionalität der IDE derart mächtig ist, dass sie ein eigenes Buch rechtfertigen würde, sollten Sie am Ende dieses Kapitels sehr gut mit Visual Studio 2010 zurechtkommen, dessen Hauptfunktionen und Stärken kennen und in der Lage sein, weitere Funktionalitäten selbstständig zu erkunden und anzuwenden.

Anstatt alle Elemente der Benutzeroberfläche (auch UI für *User Interface* genannt) herunterzubeten, liegt das Motto des Kapitels darin, die wesentlichen zu beschreiben, die Entwickler für ihre Arbeit benötigen. Dies beinhaltet:

- Ganz am Anfang die geeigneten Entwicklungseinstellungen auswählen
- Neue Projekte erstellen und bestehende Projekte auf die neue Visual Studio-Version aktualisieren
- Die Produktivität mit der mächtigen WPF-basierten Visual Studio-Benutzeroberfläche erhöhen
- Durch den Code bewegen, diesen durchsuchen und ihn nachvollziehen
- Test-gesteuerte Entwicklung
- Visual Studio erweitern / Erweiterungen von Fremdherstellern anwenden

In diesem Kapitel wird reichlich Gebrauch von Tasten und Tastenkombinationen gemacht. Die Tasten in diesem Kapitel basieren dabei auf den *Visual Basic-Entwicklungseinstellungen* und können daher von Ihrer Tastenbelegung abweichen, falls Sie andere Einstellungen verwenden. Bei den hier angegebenen Tastenkombinationen bedeutet ein Pluszeichen (+), dass Sie die Tasten gleichzeitig betätigen.

TIPP Wenn Sie im Internet Tipps zu Tastenkombinationen für das Aufrufen bestimmter Funktionen finden, sind diese unter Umständen für das englische Tastaturlayout gedacht. Beispielsweise schalten Sie bei IntelliSense zwischen den Registerkarten mit Alt < und Alt > um. Die Tastenpositionen selbst bleiben dieselben auf dem Deutschen Keyboard, aber durch das andere Layout ergeben sich die Tasten Alt , und Alt . . Aus diesem Grund ist es manchmal nützlich, das englische Tastaturlayout zu kennen, um die deutsche Entsprechung finden zu können.

In diesem Kapitel finden sich überaus viele Bildschirmabbildungen, um die unterschiedlichen Benutzeroberflächenelemente zu erklären.

Visual Studio das erste Mal starten – das Profil auswählen

Wenn Sie Visual Studio 2010 das erste Mal starten, erhalten Sie einen Dialog (Abbildung 4.1), der Ihnen die Möglichkeit gibt, Ihre Entwicklungseinstellungen auszuwählen.

Abbildung 4.1 Visual Studio 2010 das erste Mal starten

Um das Verhalten der IDE für ein bestimmtes Entwicklungsprojekt zu optimieren, unterstützt Visual Studio das Konzept der Einstellungen. Diese Einstellungen (die auch als Profil bezeichnet werden) legen benutzerspezifische Anpassungen an der IDE fest, konkret das Fensterlayout, das Ein- und Ausblenden von bestimmten Werkzeugleisten und Menüeinträgen, Tastenbelegungen, die Filterung und Anordnung von Projektvorlagen und noch mehr. Als Entwickler können Sie diejenigen Einstellungen auswählen, die mit der Form von Entwicklungsaktivitäten in Übereinstimmung stehen, die Sie am häufigsten nutzen, und an Ihre Anforderungen anpassen. Die frisch geänderten Einstellungen können dann gespeichert werden oder sogar zur gemeinsamen Nutzung freigegeben werden. Wenn Sie unsicher sind, sind die *Allgemeinen Entwicklungseinstellungen* eine gute Wahl für den Anfang. Wenn Sie die Softwareentwicklung mit Visual Basic beginnen, ist es wahrscheinlich am besten, mit den *Visual Basic-Entwicklungseinstellungen* zu starten. Nichtsdestotrotz versteckt diese Einstellung einige UI-Elemente und -Funktionalität, die Sie möglicherweise später benötigen. Daher werden zwar zu Lernzwecken und für Hobbyprogrammierer die *Visual Basic-Entwicklungseinstellungen* empfohlen, aber wenn Sie Visual Basic professionell einsetzen, sind die *Allgemeinen Entwicklungseinstellungen* wahrscheinlich besser geeignet für Sie.

Wenn Sie ein Upgrade aus einer früheren Version von Visual Studio durchführen, haben Sie zudem die Möglichkeit, die Einstellungen aus der ursprünglichen Version zu übernehmen, indem Sie die Option *Vorherige Einstellungen* verwenden.

Um die Einstellungen zu ändern, wählen Sie *Einstellungen importieren und exportieren* aus dem *Extras*-Menü, wie in Abbildung 4.2 zu sehen.

Abbildung 4.2 Um die Einstellungen der Visual Studio-IDE zurückzusetzen, verwenden Sie den Menüeintrag *Einstellungen importieren und exportieren*

Dieser Vorgang ruft einen Assistenten auf, mit dem Sie das Importieren benutzerdefinierter Einstellungen, das Exportieren der aktuellen Umgebungseinstellungen oder das Zurücksetzen der Einstellungen auf die Standardeinstellungen durchführen können. Dies erweist sich als nützlich, da Entwickler häufig die Programmiersprache oder zwischen Client- und Webentwicklung wechseln.

TIPP Wenn Einstellungen geändert werden, bietet der Assistent eine Option an, mit der sich die aktuellen Einstellungen speichern lassen. Diese Sicherung kann dann verwendet werden, wenn Sie in das ursprüngliche Projekt zurückkehren und wieder mit den früheren Einstellungen arbeiten möchten. Die Einstellungen werden dabei standardmäßig in einer XLM-Datei namens *CurrentSettings-<Datum>.vssettings* im Verzeichnis *Eigene Dokumente\Visual Studio 2010\Settings* gespeichert (Dateiname und Verzeichnis können beliebig angepasst werden). Aktualisierungen in der Datei finden jedoch unabhängig von diesem Dialog immer dann statt, wenn eine Einstellung in der IDE geändert wird, etwa eine Symbolleiste verschoben wird, die Farbeinstellungen für Texte oder Kommentare geändert werden usw.

Die Einstellungen werden schichtenförmig angewendet. Eine Einstellungsdatei kann sowohl Informationen über sämtliche IDE-Einstellungen aufweisen als auch nur einer Teilmenge davon. Das bedeutet, dass Sie mehrfache *.Vssettings*-Dateien importieren können, wobei die neue Datei dann nur jeweils diejenigen Einstellungen verändert, die in der neuen Datei definiert sind. Eine Beispieldatei mit Einstellungen findet sich in den Begleitdateien zu diesem Buch im Kapitelverzeichnis.

Die Startseite – der Ausgangsort für Ihre Entwicklungsaufgaben

Nachdem Sie nun Ihre Entwicklungseinstellungen ausgewählt haben, stellt das erste Dokument, das in der Visual Studio-IDE geöffnet ist, die Startseite dar. Die Startseite ist eine Art Willkommensseite, die Direktzugriffsmöglichkeiten zum Erstellen neuer Projekte, zum Öffnen bestehender Projekte, zum Abrufen von Hilferessourcen sowie von Informationen über neue Updates von Visual Studio anbietet. Abbildung 4.3 zeigt die verschiedenen Bereiche der Startseite.

Abbildung 4.3 Die Startseite von Visual Studio

Die Startseite ist in drei Hauptbereiche aufgeteilt: den Befehlsbereich, der Befehle zum Anlegen neuer Projekte sowie zum Öffnen bestehender Projekte aufweist, die Liste mit den zuletzt geöffneten Projekten (die leer ist, wenn Sie Visual Studio das erste Mal überhaupt starten) und einen registerartigen Inhaltsbereich mit den Registerkarten *Erste Schritte*, *Leitfaden und Ressourcen* sowie *Aktuelle Nachrichten* (Letztere erlaubt das Abonnieren eines RSS-Feeds). Der Befehlsbereich beinhaltet die Befehle *Neues Projekt* und

Projekt öffnen. Die Liste *Zuletzt geöffnete Projekte* bietet Links zu den zuletzt verwendeten Projekten an. Ein Klick auf einen Link öffnet das jeweilige Projekt in Visual Studio. Ein Rechtsklick auf einen Link erlaubt es, eine der Optionen aus Tabelle 4.1 auszuwählen.

Projekt öffnen	Lädt die Projektmappe in die IDE
Enthaltenen Ordner öffnen	Startet den Windows-Explorer und öffnet das Verzeichnis, in dem sich die Projektmappendatei befindet
Aus Liste entfernen	Entfernt den Eintrag aus der Liste *Zuletzt geöffnete Projekte*. Bei diesem Vorgang wird nur der Link aus der Liste herausgenommen, nicht aber das Projekt selbst vom Datenträger gelöscht.

Tabelle 4.1 Optionen für Projekte aus der Liste *Zuletzt geöffnete Projekte* auf der Startseite

Wird der Mauszeiger auf einen Eintrag in der Liste *Zuletzt geöffnete Projekte* bewegt, wird der Eintrag hervorgehoben und mit einem Pinnnadelsymbol versehen. Ein Klick auf das Symbol »pinnt« das Projekt auf die Liste, sodass es dauerhaft in der Liste bleibt, auch wenn inzwischen andere Projekte geöffnet und geschlossen werden und das Projekt daher nach einiger Zeit aus der Liste verschwinden würde. Ein Klick auf ein Pinnnadelsymbol eines Projekts hebt den gepinnten Status wieder auf. Das Symbol verschwindet und das Projekt wird wie andere Projekte behandelt. In Abbildung 4.3 ist *String - Split* ein gepinntes Projekt. Sie können auch das Kontextmenü des Projekts (dazu das Projekt mit der rechten Maustaste anklicken) verwenden, um das Projekt aus der Liste zu entfernen. Die Anzahl der Projekte, die in der Liste *Zuletzt geöffnete Projekte* angezeigt werden, kann individuell angepasst werden. Um die Anzahl zu ändern (10 ist die Vorgabe), rufen Sie *Extras/Optionen/Umgebung/Allgemein* auf und geben Sie den gewünschten Wert im Textfeld *in den Listen der zuletzt geöffneten Elemente angezeigte Elemente* ein (Abbildung 4.4). Der Wert kann sich dabei im Bereich von 1 bis 24 bewegen.

Abbildung 4.4 Die Einträge der Liste *Zuletzt geöffnete Projekte konfigurieren*

Der Inhaltsbereich beheimatet die Registerkarten *Erste Schritte*, *Leitfaden und Ressourcen* sowie *Aktuelle Nachrichten*. Die Registerkarte *Erste Schritte* zeigt eine Liste mit Hilfethemen, Websites, technischen Artikeln und anderen Ressourcen, die Ihnen helfen können, Ihre Produktivität zu erhöhen und etwas über die Features von Visual Studio zu lernen. Was in der Auflistung angeboten wird, hängt von der Auswahl in der Kategorienleiste unterhalb der Registerkartenleiste ab. Klicken Sie entsprechend auf *Willkommen*, *Windows*, *Internet*, *Cloud*, *Office*, *SharePoint* oder *Daten*. Ein Klick auf ein Symbol innerhalb der Registerkarte *Erste Schritte* zeigt eine Beschreibung der Ressource an, zusammen mit einer Auflistung von Links, die zur jeweiligen Ressource gehören. Die Registerkarte *Leitfaden und Ressourcen* stellt eine allgemeinere Hilfe zu den Themen Codeerstellung und Programmierung zur Verfügung und verhält sich ansonsten vom Aufbau her wie die Registerkarte *Erste Schritte*. Sie enthält unter anderem Kategorien für den Entwicklungsprozess, MSDN-Ressourcen und zusätzliche Tools. Die Registerkarte *Aktuelle Nachrichten* bietet ausgewählte Artikel aus dem eingestellten Newsfeed. Sie können auf einen anderen Newsfeed schalten, indem Sie die Adresse in dem Textfeld *RSS-Feed* oberhalb der Artikelliste ändern.

Wie sich Visual Studio beim Start verhält, kann über *Extras/Optionen/Umgebung/Start* festgelegt werden (Abbildung 4.5).

Abbildung 4.5 Die Startseite konfigurieren

Obwohl die Startseite nützlich ist, schließen Entwickler die Startseite in den meisten Fällen, wenn das jeweilige Projekt erst einmal geöffnet wurde. Dies kann auch automatisch erreicht werden, indem das Kontrollkästchen *Seite nach dem Laden des Projekts schließen* auf der Startseite unten aktiviert wird. Es gibt hier ein weiteres Kontrollkästchen (*Seite beim Start anzeigen*), mit dem gesteuert wird, ob die Startseite beim Aufruf von Visual Studio angezeigt werden soll oder nicht.

Mit der Entwicklung beginnen – neue Projekte erstellen

Wenn Sie ein neues Projekt erstellen möchten, rufen Sie den Dialog *Neues Projekt* auf. Dieser Dialog kann über die Startseite, über das Menü *Datei/Neues Projekt,* mit dem Symbol oder per Strg+N erreicht werden. Abbildung 4.6 zeigt den Dialog *Neues Projekt*.

Abbildung 4.6: Der Dialog *Neues Projekt*

Der Dialog *Neues Projekt* kategorisiert die installierten Vorlagen und ermöglicht die Erstellung neuer Projekte. Projektvorlagen sind die Dateien, die Informationen über die Visual Studio-Version, das Ziel-.NET Framework, die bei der Projekterstellung hinzuzufügenden Verweise und Importe und noch mehr enthalten. Um ein neues Projekt zu erstellen, wählen Sie einfach eine Vorlage aus der Liste aus, versehen das Projekt und die Projektmappe mit einem Namen, geben einen Speicherort ein und klicken auf *OK*.

Die Vorlagen werden auf Basis der .NET Framework-Version gefiltert, die im Listenfeld im oberen mittleren Bereich des Dialogs ausgewählt ist. Der Inhalt des Listenfelds hängt von den .NET Framework-Versionen ab, die auf dem Computer installiert sind. Auf die verschiedenen .NET Framework-Versionen wird im Abschnitt »Anwendungen für mehrere .NET Framework-Versionen (Multizielversionen) unter Visual Studio 2010« (Seite 80) später in diesem Kapitel noch eingegangen.

Da es inzwischen eine große Anzahl an Vorlagen gibt und die Liste zudem weiter im Wachsen begriffen ist, erweist sich die Suchfunktionalität des Dialogs *Neues Projekt* als nützlich. Um nach Vorlagen zu suchen, verwenden Sie das Textfeld *Installierte Vorlagen suchen* in der rechten oberen Ecke des Dialogs. Ähnlich wie beim Windows-Explorer und Internet Explorer setzt Strg+E die Einfügemarke in das Suchfeld.

Die Suche geht über die Titel und Beschreibungen der Vorlagen in allen Kategorien. Groß- und Kleinschreibung wird dabei ignoriert, ebenso wird auch innerhalb von Begriffen gesucht. Werden mehrere Suchbegriffe verwendet, werden diese mit einem logischen *Und* verknüpft. Das bedeutet, dass der Suchbegriff *proj test VB* dazu führt, dass alle Vorlagen gefunden werden, die Begriffe aufweisen (im Titel und der Beschreibung), die sämtliche Teilstrings in einer beliebigen Reihenfolge enthalten. Der Dialog kann auch dazu verwendet werden, um nach Onlinevorlagen und von Benutzern erstellten Vorlagen zu suchen und diese zu installieren.

Um Projekte zu erweitern, indem Projektelemente hinzugefügt werden, verwenden Sie den Dialog *Neues Element hinzufügen*. Die Projektelementvorlagen im Dialog werden basierend auf dem aktuellen Projekt und der .NET Framework-Zielversion gefiltert. Analog zu den Projektvorlagen speichern Elementvorlagen Informationen über Dateien, Verweise und Importe, die in das Projekt aufgenommen werden, wenn das Projektelement dem Projekt hinzugefügt wird. Der Dialog *Neues Element hinzufügen* (Abbildung 4.7) ist in puncto Funktionalität dem Dialog *Neues Projekt* sehr ähnlich.

Abbildung 4.7 Dialog *Neues Element hinzufügen*

Vorlagen verwalten

Dieser Abschnitt betrifft diejenigen unter Ihnen, die eigene Vorlagen erstellen oder die installierten Vorlagen in einer anderen Weise als der standardmäßigen Kategorisierung organisieren möchten. Trifft dies nicht auf Sie zu, können Sie diesen Abschnitt überspringen.

Vorlagendateien müssen an einem Ort untergebracht werden, den Visual Studio erkennt. Nur dann erscheinen die Vorlagen in den Dialogen *Neues Projekt* und *Neues Element hinzufügen*. Sie können eigene Unterkategorien für Vorlagen erstellen, wobei diese Unterkategorien dann auch in der Benutzeroberfläche erscheinen. Standardmäßig durchsucht Visual Studio zwei Orte nach Projekt- und Elementvorlagen. Wenn eine komprimierte Datei, die eine *.Vstemplate*-Datei enthält, an diesen Orten vorhanden ist, erscheint die Vorlage in den Dialogen *Neues Projekt* und *Neues Element hinzufügen*. Die zusammen mit Visual Studio installierten Vorlagen werden an den folgenden Orten untergebracht (die Lokalsierung ist 1031 für Deutsch und 1033 für Englisch):

- \<Visual Studio-Installationsverzeichnis>\Common7\IDE\ItemTemplates\<Programmiersprache>\<Lokalisierung>\
- \<Visual Studio-Installationsverzeichnis>\Common7\IDE\ProjectTemplates\<Programmiersprache>\<Lokalisierung>\

Eigene Visual Basic-Projekt- und -Elementvorlagen werden hier untergebracht:

- \Eigene Dokumente \Visual Studio 2010\Templates\ProjectTemplates\Visual Basic\
- \ Eigene Dokumente\Visual Studio 2010\Templates\ItemTemplates\Visual Basic\

Die Orte für eigene Vorlagen lassen sich per *Extras/Optionen/Projekte und Projektmappen/Allgemein* verändern. Die Kategorien in den Dialogen *Neues Projekt* und *Neues Element hinzufügen* spiegeln die Verzeichnisstruktur an den Orten mit den installierten und eigenen Vorlagen wider. Sie können diese Verzeichnisstrukturen verändern, um Ihre Vorlagen auf die gewünschte Weise zu organisieren. Beachten Sie, dass neue Kategorien nicht auf Programmiersprachenebene, sondern nur innerhalb der einzelnen Programmiersprachen erzeugt werden können.

Auf ähnliche Weise können installierte Vorlagen organisiert werden, indem Unterverzeichnisse im Programmiersprachenordner erzeugt werden. Diese Unterverzeichnisse erscheinen in den Dialogen *Neues Projekt* und *Neues Element hinzufügen* als virtuelle Ordner innerhalb jeder Programmiersprache.

Änderungen an der Organisation der Vorlagen machen sich nicht unmittelbar bemerkbar. Um die Änderungen wirksam werden zu lassen, gehen Sie folgendermaßen vor:

1. Schließen Sie alle Instanzen von Visual Studio.
2. Starten Sie die *Visual Studio-Eingabeaufforderung (2010)* aus dem Startmenü als Administrator.
3. Geben Sie **devenv /setup** ein.
4. Starten Sie Visual Studio. Die Änderungen sollten nun in den Dialogen *Neues Projekt* und *Neues Element hinzufügen* sichtbar sein.

Vorlagen ändern

Um die Vorlagen zu ändern, extrahieren Sie einfach die spezifische ZIP-Datei, die zu Ihrer Vorlage gehört, die Sie ändern möchten, machen Sie die entsprechenden Änderungen und komprimieren Sie alle Dateien wieder.

Aus früheren Visual Studio-Versionen auf Visual Studio 2010 migrieren

Lassen Sie uns nun die Gangart wechseln und den Prozess besprechen, wie Projekte und Projektmappen, die unter früheren Versionen von Visual Studio erstellt wurden, in Visual Studio 2010 übernommen werden.

Mit Visual Studio 2003-2008 erstellte Projekte aktualisieren

Der Prozess des Migrierens von Projekten, die unter einer früheren Version von Visual Studio erstellt wurden, auf eine neuere Version von Visual Studio wird als *Aktualisieren* (oder *Upgrade)* bezeichnet. Wenn ein unter Visual Studio 2003 - Visual Studio 2008 erstelltes Projekt in Visual Studio 2010 geöffnet wird, ermittelt Visual Studio die .NET Framework-Zielversion des zu aktualisierenden Projekts. Um die Stetigkeit des Entwicklungsprozesses zu bewahren, behält Visual Studio 2010 die .NET Framework-Zielversion für zu aktualisierende Projekte bei. Dieses Verhalten unterscheidet sich jedoch von der Express-Version von Visual Studio, bei der Projekte auf die Zielversion 4.0 von .NET Framework aktualisiert werden. Es ist klar, dass dies nur funktionieren kann, wenn die Zielversionen von .NET Framework auf Ihrem Computer installiert sind. Sie müssen .NET 3.5 SP1 installiert haben, um als Zielversion .NET Framework 2.0 - 3.5 anvisieren zu können. Wenn die angepeilte .NET Framework-Zielversion sich nicht auf Ihrem Computer befindet, bietet der Konvertierungs-Assistent folgende relevante Möglichkeiten an (Abbildung 4.8):

- Die benötigte .NET Framework-Version herunterzuladen (mittlere Option),
- die .NET Framework-Zielversion der Projekte auf 4.0 zu ändern (obere Option).

Abbildung 4.8 Ein Projekt aktualisieren, wenn sich die .NET Framework-Zielversion nicht auf dem Computer befindet[1]

[1] Der Begriff *erneut* in der ersten Option ist nicht ganz glücklich, da es sich um eine Änderung (Neuausrichtung) handelt und nicht um eine erneute (wiederholte) Auswahl derselben Einstellung. Im englischen Original heißt es retarget, was etwa »neu einstellen« oder »neu anvisieren« bedeutet.

Der Upgradeprozess aktualisiert Projektdateien und Projektmappendateien auf die Visual Studio-Version 2010 (interne Versionsnummer 10.0). Er entschärft auch Bezeichner in den Codedateien, die sich mit neu eingeführten Schlüsselwörtern überschneiden. Wenn zum Beispiel ein Visual Studio 2003-Projekt auf Visual Studio 2010 aktualisiert wird, werden sämtliche Bezeichner namens *Global, Using, IsNot, Partial, Custom, UShort, Uinteger, ULong* und *Of* in eckige Klammern [] gesetzt und damit »escapet«. Bei der Aktualisierung eines Projekts, das »überholte«, also veraltete APIs verwendet oder Codemuster, die in neueren Versionen von Visual Studio zu Warnungen führen, wird die Fehlerliste beim Aktualisierungsprozess mit »veraltet«-Hinweisen und anderen Warnungen gefüllt.

Dinge, die beim Konvertieren von Projekten nach Visual Studio 2010 zu beachten sind

- Nachdem eine Projektmappe oder eines der darin enthaltenen Projekte konvertiert wurde, ist es nicht länger möglich, diese unter früheren Versionen von Visual Studio zu bearbeiten, zu kompilieren und zu starten
- Wenn die Projektmappe oder das Projekt unter Quellcodeverwaltung steht, wird sie bzw. es während der Konvertierung automatisch ausgecheckt
- Die Erstellung einer Sicherungskopie (diese Option ist im Konvertierungs-Assistenten verfügbar) ist stets eine sichere Variante
- In Verbindung mit Microsoft Office-Projekten modifiziert der Konvertierungs-Assistent Ihr Projekt, um die Zielversion von Microsoft Office, die Sie installiert haben, zu adressieren. Wenn Sie die Microsoft Office-Version, welche dieses Projekt anvisiert, nicht ändern möchten, brechen Sie den Assistenten ab und deaktivieren das Kontrollkästchen *Immer Upgrade auf installierte Version von Office durchführen* im Dialog *Extras/Optionen/Office-Tools/Projektupgrade*.
- Die Aktualisierung von *Intelligente Geräte*-Projekte wird derzeit nicht von Visual Studio 2010 unterstützt
- Um die Konvertierung einer großen Anzahl an Projektmappen oder Projekten zu automatisieren, können Sie eine Stapeldatei erstellen, welche die Befehlszeilen-Upgradetools verwenden. Das folgende Skript aktualisiert eine Projektmappe namens *MyProject.sln:*

```
devenv "MyProject.sln" /upgrade
```

  Eine Sicherungskopie wird dabei automatisch im aktuellen Ordner erzeugt.

- Das Konvertierungsprotokoll enthält hilfreiche Informationen für den Fall, dass die Projekte (bzw. die Projektmappe) nicht reibungslos aktualisiert werden konnten. Tabelle 4.2 enthält eine Auflistung von Fehlermeldungen, die beim Scheitern des Konvertierungsvorgangs auftreten können.

Fehlermeldung	Bedeutung
Fehler beim Erstellen der Sicherungsdatei	Es kann keine Sicherungskopie der Projektmappendatei erzeugt werden. Überprüfen Sie, ob bereits eine Datei mit demselben Namen vorhanden ist.
Fehler beim Erstellen einer Sicherungskopie der Projektdatei	Es kann keine Sicherungskopie der Projektdatei erzeugt werden. Überprüfen Sie, ob bereits eine Datei mit demselben Namen vorhanden ist.
Datei kann nicht geschrieben werden	Die Datei ist schreibgeschützt oder befindet sich auf einer schreibgeschützten Netzwerkfreigabe. Dateien, die der Quellcodeverwaltung unterliegen und die exklusiv von einem anderen Benutzer ausgecheckt wurden, sind schreibgeschützt. ▶

Fehlermeldung	Bedeutung
Interner, nicht schwerwiegender Fehler	Es ist ein Fehler in Visual Studio aufgetreten. Um zu versuchen, Visual Studio zu reparieren, wählen Sie im Dialog *Programm deinstallieren oder ändern* (*Systemsteuerung/Programme und Funktionen*) das installierte Produkt über einen Klick mit der rechten Maustaste aus und klicken dann auf *Deinstallieren/ändern* im Kontextmenü.
Die Projektdatei kann nicht zum Aktualisieren geöffnet werden	Die Projektdatei kann nicht geöffnet werden. Überprüfen Sie, ob die Projektdatei von einer anderen Anwendung geöffnet und damit gesperrt wurde.
Projektdatei kann nicht analysiert werden	Die Projektdatei ist korrupt oder stellt eine Version dar, die nicht aktualisiert werden kann
Die aktualisierte Projektdatei kann nicht gespeichert werden	Es ist möglicherweise ein interner Fehler aufgetreten
Die Webprojektkomponente ist nicht installiert. Projekt kann nicht aktualisiert werden	Das Projekt ist ein Webprojekt. Die einzige Express Edition, die zur Konvertierung von Webprojekten verwendet werden kann, ist die Visual Web Developer Express Edition.

Tabelle 4.2 Mögliche Fehlermeldungen beim Konvertierungsvorgang auf Visual Studio 2010

Visual Basic 6.0-Anwendungen auf Visual Studio 2010 aktualisieren

Visual Studio 2010 unterstützt keine direkte Aktualisierung von Visual Basic 6.0-Anwendungen. Die Aufgabenstellung kann aber unter Zuhilfenahme zweier alternativer Varianten durchgeführt werden:

- Laden Sie die kostenlose Visual Basic 2008 Express Edition herunter und aktualisieren Sie die Visual Basic 6.0-Projekte in das Visual Studio 2008-Format. Nachdem die Projekte nach Visual Studio 2008 konvertiert wurden, lassen sie sich über den Konvertierungs-Assistenten wie oben beschrieben auf Visual Studio 2010 aktualisieren.
- Verwenden Sie entsprechende Tools von Fremdherstellern, es gibt dabei sowohl kostenlose als auch leistungsfähigere kostenpflichtige Versionen dieser Tools, die abhängig von den Projektanforderungen verwendet werden können. ArtinSoft ist ein Beispiel für eine Firma, die ein Tool für diese Aufgabe anbietet.

Die zweite Variante wird auch von Microsoft bevorzugt.

Anwendungen für mehrere .NET Framework-Versionen (Multizielversionen) unter Visual Studio 2010

Häufig werden Softwareentwickler und große Firmen daran gehindert, neuere Versionen von Visual Studio einzusetzen, bevor nicht deren Kundenstamm auf das dazugehörige neue .NET Framework umsattelt. Die Herausforderung wird umso größer, wenn einige Firmen oder Personen Anwendungen entwickeln, die auf verschiedene .NET Framework-Versionen abzielen. In Visual Studio 2002 bis Visual Studio 2005 bestand die einzige Möglichkeit darin, nebeneinander mehrfache Versionen von Visual Studio zu installieren und mehrfache Kopien der Anwendung zu pflegen. Das Multizielversionen (Multi-targeting)-Feature von Visual Studio 2010 erlaubt Entwicklern, Anwendungen zu erstellen, welche .NET Framework 2.0, 3.0, 3.5 und 4.0 sowie unterschiedliche Profile innerhalb dieser .NET Framework-Versionen anvisieren. Der Hauptvorteil von Multizielversionen liegt darin, dass Sie Visual Studio 2010 einsetzen können, um neue Projekte zu

erstellen und zu entwickeln, die auf ältere Versionen von .NET Framework abzielen (etwa 2.0, 3.0 und 3.5). Der Einsatz von Multizielversionen in Visual Studio 2010 ermöglicht es ebenso, dass Sie die Entwicklung von Projekten fortsetzen, die unter älteren Versionen von Visual Studio wie Visual Studio 2005 und Visual Studio 2008 erstellt wurden, und dennoch von den neuesten und den ausgezeichneten Tools von Visual Studio 2010 profitieren.

Die Geschichte der Multizielversionen

Visual Studio unternahm das erste Mal in der Version 2008 einen Schritt Richtung Multizielversionen, aber es war nur ein Minischritt. Auch wenn es Ihnen Visual Studio 2008 erlaubte, verschiedene Versionen von .NET Framework 2.0, 3.0 und 3.5 anzuvisieren, blieb die zugrunde liegende Common Language Runtime (CLR) für all diese .NET Framework-Versionen dieselbe, nämlich .NET CLR 2.0. Sie können sich das wie ein zwiebelartiges Modell vorstellen, bei dem die CLR 2.0 den Kern bildet und .NET Framework 2.0, .NET Framework 3.0 und .NET Framework 3.5 Schichten darstellen, die jeweils übereinander gelegt sind (Abbildung 4.9).

Abbildung 4.9 Schematische Darstellung der Multizielversionen bei Visual Studio 2008

Die Herausforderung für Multizielversionen wuchs mit dem Erscheinen des neuen .NET 4.0 Framework für Visual Studio 2010, das auf der .NET CLR-Version 4.0 läuft. Diese Version der CLR überschreibt keine früheren Versionen, sondern installiert sich parallel zu früheren Versionen der .NET CLR.

Visual Studio ging einen weiteren Schritt mit Visual Studio 2010 nach vorne, indem es die Adressierung von .NET Framework-Versionen ermöglicht, die mehrere Versionen der CLR (CLR 2.0 und CLR 4.0) überspannen (Abbildung 4.10).

Auch die Aufnahme von Silverlight-Projektvorlagen in Visual Studio 2010 bedeutete, dass Visual Studio 2010 die Fähigkeit bekommen sollte, Projekte zu erstellen, die als Ziel Silverlight 2.0 bis Silverlight 4.0 abdecken.

Abbildung 4.10 Schematische Darstellung von Multizielversionen bei Visual Studio 2010

Multizielversionen in Visual Studio 2010 werden von der IDE gut unterstützt. Es kamen einige Features dazu, die bei der Arbeit an einem Projekt, das eine frühere Version von .NET Framework anvisiert, behilflich sind. Nachfolgend finden Sie eine Auflistung einiger der Filterungen, die in diesem Zusammenhang von Visual Studio zur Verfügung gestellt werden. Diese Filterungen gewährleisten, dass eine Anwendung, die auf eine bestimmte .NET Framework-Version abzielt, ordnungsgemäß bei den Endanwendern funktioniert, also auf Computern, auf denen gegebenenfalls nur die jeweilige .NET Framework-Zielversion installiert ist.

- Elemente werden in den Dialogen *Neues Projekt*, *Neues Element hinzufügen*, *Verweis hinzufügen* und *Dienstverweis hinzufügen* gefiltert, um die Auswahl von Elementen zu verhindern, die in der .NET Framework-Zielversion nicht verfügbar sind
- Benutzerdefinierte Steuerelemente werden in der Toolbox gefiltert, um solche Steuerelemente zu entfernen, die in der Zielversion nicht verfügbar sind, und die neueste Version von Steuerelementen anzuzeigen, falls mehrere Steuerelemente für die anvisierte Zielversion vorhanden sind
- IntelliSense wird ebenso gefiltert, um die Verwendung von Sprachmerkmalen zu verhindern, die der Zielversion nicht bekannt sind. Da die Filterung auf den referenzierten Assemblys für die .NET Framework-Zielversion basiert, gibt es aber auch Ausnahmen.
- Eigenschaften im Eigenschaften-Fenster werden einer Filterung unterzogen, damit die Eigenschaften, die der Zielversion nicht bekannt sind, weggelassen werden
- Auch Menüoptionen, die in der Zielversion nicht verfügbar sind, werden durch Filterung aussortiert
- Die Kompilierung und der Build-Erstellungsprozess erfolgen mit der dazugehörigen Version des Compilers oder mit dem 4.0-Compiler unter Verwendung der entsprechenden Compiler-Schalter

Multizielversionen für neue Projekte

Visual Studio 2010 erlaubt Ihnen bei der Erstellung einer neuen Anwendung, zwischen .NET Framework 2.0, 3.0, 3.5 und 4.0 als Zielversion zu wählen. Der Dialog *Neues Projekt* hat hierzu eine Auswahlliste mit den jeweiligen Zielversionen von .NET Framework, die auf Ihrem Computer installiert sind (Abbildung 4.11).

Abbildung 4.11 Die Zielversion von .NET Framework für ein neues Projekt auswählen

HINWEIS In den Express Editionen von Visual Studio ist das Merkmal Multizielversionen im Dialog *Neues Projekt* nicht verfügbar.

Die .NET Framework-Zielversion für Anwendungen ändern

Bei einem bestehenden Projekt können Sie die Zielversion von .NET Framework oder eines .NET Framework-Profils in den Projekteigenschaften ändern. Hierfür gehen Sie folgendermaßen vor:

1. Öffnen Sie die Projekteigenschaften, indem Sie zum Beispiel doppelt auf den Zweig *My Project* im Projektmappen-Explorer klicken.
2. Wechseln Sie auf die Registerkarte *Kompilieren*.
3. Klicken Sie auf die Schaltfläche *Erweiterte Kompilierungsoptionen*.
4. Wählen Sie die .NET Framework-Zielversion aus dem Listenfeld *Zielframework (alle Konfigurationen)* aus (Abbildung 4.12). Beachten Sie, dass das Listenfeld nur die auf dem Computer installierten .NET Framework-Versionen anbietet und eine Option aufweist, mit der sich .NET Framework-Versionen aus dem Internet installieren lassen (*Andere Frameworks installieren*). Letztere öffnet .NET Framework Developer Center im Webbrowser.

Wenn die .NET Framework-Version eines Projekts geändert wird, führt Visual Studio sämtliche erforderlichen Anpassungen an den Verweisen und Konfigurationsdateien durch.

Abbildung 4.12 Die .NET Framework-Zielversion eines Projekts ändern

Erstellen Sie beispielsweise eine neue Windows Forms-Anwendung mit der .NET Framework-Zielversion 3.5, besitzen alle Verweise die Runtime-Version 2.0.50727 (Abbildung 4.13). Aktualisieren Sie nun das Projekt auf Visual Studio 2010. Jetzt werden die Versionen der Verweise auf v4.0.30319 geändert (Abbildung 4.14).

Abbildung 4.13 Eigenschaften der Datei *System.dll* bei einem Projekt, das die .NET Framework-Version 3.5 anvisiert

Abbildung 4.14 Eigenschaften der Datei System.dll bei einem Projekt, das die .NET Framework-Version 4.0 anvisiert

Profile verstehen

Beim Öffnen des Listenfelds für die .NET Framework-Zielversion haben Sie sicher die Einträge *.NET Framework 4 Client Profile* und *.NET Framework 3.5 Client Profile* bemerkt.

Ein .NET Framework-Profil ist eine Untermenge von .NET Framework, die eine eingeschränkte Menge an Bibliotheken und Leistungsmerkmalen aufweist. Der Hauptvorteil von Profilen liegt darin, dass die Größe des Verteilungspakets einer Anwendung reduziert wird, indem nur die relevanten Assemblys aufgenommen werden. Außerdem stellt dies eine Hilfe dar, da Sie sich keine Gedanken machen müssen, was das Installieren von Patches angeht, wenn bestimmte Assembly(versionen) nicht in dem auf Ihrem Computer befindlichen Profil enthalten sind.

.NET Framework 4 Client Profile enthält die Leistungsmerkmale, die zur Entwicklung einer Clientanwendung wie Windows Forms, WPF usw. benötigt werden, lässt aber Features wie ASP.NET und .NET Framework Data Provider for Oracle aus. Eine Anwendung, die auf .NET Framework 4 Client Profile abzielt, weist ein kleineres Redistributionspaket auf, bei dem nur die Minimalausstattung der Client-Assemblys auf dem Computer des Endbenutzers installiert wird, ohne dass die vollständige Version von .NET Framework 4 erforderlich ist.

Eine vollständige Liste an Verweisassemblys, die in .NET Framework 4 Client Profile enthalten sind, finden Sie unter »Assemblies in the .NET Framework Client Profile« unter *http://msdn.microsoft.com/en-us/library/ff462634.aspx*.

In dem oben erwähnten Listenfeld haben Sie sicher auch bemerkt, dass .NET Framework 3.5 zudem ein Clientprofil aufweist und sich vielleicht gefragt, wo der Unterschied zwischen .NET Framework 3.5 Client Profile und .NET Framework 4.0 Client Profile sei. .NET Framework Client Profile wurde mit .NET Framework 3.5 SP1 eingeführt, um die Verteilung und Installation von .NET Framework zu verbessern. Tabelle 4.3 stellt die Unterschiede zwischen .NET Framework 3.5 SP1 Client Profile und .NET Framework 4 Client Profile gegenüber. Beachten Sie, dass die Version 4.0 vielseitiger ist und besser unterstützt wird als die Version 3.5.

.NET Framework 3.5 SP1-Client Profile	.NET Framework 4 Client Profile
Nur Webinstallation	Lokales Paket und Webinstallation
Unterstützt nur Microsoft Windows XP SP2 oder SP3 und x86- Architekturen, auf denen keine frühere Version von Microsoft .NET Framework installiert ist	Alle Plattformen und Prozessorarchitekturen, die von .NET Framework 4 unterstützt werden, mit Ausnahme von IA64
Eigener Eintrag *Deinstallieren/ändern* in *Programme und Funktionen* (*Systemsteuerung*). Wenn die vollständige Version von .NET Framework installiert wird, wird der Client in *Programme und Funktionen* ersetzt. Eine Rückkehr zum alten Stand ist dann nicht möglich	Teil von .NET Framework. Dabei besteht .NET Framework aus den Komponenten .NET Framework 4 Client Profile und .NET Framework 4 Extended, für die in *Programme und Funktionen* (*Systemsteuerung*) jeweils ein gesonderter Eintrag vorhanden ist
Windows Update führt ein Upgrade auf die vollständige Version von .NET Framework durch	Unabhängige Komponente Kann getrennt verwaltet werden Benötigt die Komponente .NET Framework 4 Extended von .NET Framework nicht

Tabelle 4.3 Unterschiede zwischen den beiden .NET Framework-Clientprofilen

Das Clientprofil ist Vorgabe für Clientprojekte

Beim Erstellen einer neuen Clientanwendung, die auf .NET Framework 4.0 abzielt (Windows Forms-, WPF-, Microsoft Office-, Workflow-, Konsolenanwendung etc.), ist als .NET Framework-Zielversion standardmäßig *.NET Framework 4 Client Profile* eingestellt. Bei Nicht-Clientanwendungen wie Web- und Verteilungsanwendungen ist das Ziel standardmäßig auf *.NET Framework 4* (also das vollständige Profil) gesetzt. Da Profile nicht im Dialog *Neues Projekt* angeboten werden, können diese nur über die Projekteigenschaften geändert werden.

Wissenswertes über Multizielversionen

Die entscheidenden Fragen, die mir beim Verstehen von Multizielversionen durch den Kopf gingen, sind die folgenden:

- Wie gewährleistet Visual Studio, dass die Anwendung nur gegen eine erlaubte, festgelegte Konfiguration ausgeführt wird?

  Die Antwort liegt in der zusätzlichen Datei *App.config,* die den .Exe-Projektvorlagen hinzugefügt wird und konkrete Profil- und CLR-Runtime-Versionen vorgibt. Es wird dabei sichergestellt, dass Anwendungen, die in der Datei *App.config* als *NETFX4 Full* spezifiziert sind, nicht auf einem NETFX4-Client gestartet werden können. Fehlt die Datei *App.config*, läuft die Anwendung ausschließlich unter dem Clientprofil.

- Welcher Compiler wird im Falle einer kleineren Zielversion (*Down Targeting*) bei der Kompilierung meines Projekts verwendet?

  Für Nicht-Webanwendungen wird auf den .NET 4.0-Compiler zurückgegriffen, um alle .NET Framework-Zielversionen abzudecken. Der 4.0-Compiler wurde im Rahmen eines Wechsels der Sprachversion eingeführt, die es erlaubt, IL-kompatiblen Code in Verbindung mit verschiedenen .NET Framework-Zielversionen zu erzeugen. Der .NET 4.0-Compiler ermöglicht den Einsatz neuerer Sprachfeatures, die für *Down Targeting* verwendet werden.

Grenzen der Multizielversionen

Das Multizielversionen-Feature hat trotz seiner unbestrittenen Fähigkeiten einige Schwächen oder, anders ausgedrückt, einiges an Potenzial für Verbesserungen:

Keine Roundtrip-Unterstützung Roundtrip (auf Deutsch: *hin und zurück*) ist die Fähigkeit, eine aktuelle oder ältere Version von Visual Studio verwenden zu können, um eine Plattform zu adressieren, die von beiden Visual Studio-Versionen unterstützt wird. Mit Roundtrip könnten Sie zum Beispiel Projekte aus einer älteren Version von Visual Studio in einer neueren IDE laden, ohne dass die Notwendigkeit einer Konvertierung bestünde. Somit ließe sich nebeneinander an alten und aktualisierten Projekten arbeiten. Das Fehlen der Roundtrip-Unterstützung hat zur Folge, dass Unternehmen, die einige Entwickler haben, die mit Visual Studio 2008 arbeiten, und andere, die Visual Studio 2010 einsetzen, ihre Projektmappen separat aufheben müssen. Die Quelldateien können dagegen projektmappenübergreifend verteilt werden, solange nicht die Entwurfszeit-Kompatibilität gefährdet ist, auf die im Folgenden eingegangen wird.

Gefährdete Entwurfszeit-Kompatibilität Mit dem Einsatz des .NET 4.0-Compilers in der IDE erhalten Sie sowohl eine neue Toolunterstützung für Visual Studio 2010 als auch die Möglichkeit, neue Sprachfeatures, die von der BCL (Base Class Library) für die .NET Framework-Zielversion unterstützt werden, zu verwenden. Die eine Kehrseite der Möglichkeit der Verwendung von Visual Studio 2010-Features für kleinere Zielversionen mündet in einem potenziellen Problem bei Quelldateien, die in Verbindung mit älteren Versionen nicht entwurfszeitkompatibel sind. Dies kann bei einem Quellcode, der projektübergreifend unterschiedliche .NET Framework-Versionen adressiert, zu Problemen führen. Dazu ein Beispiel: Der folgende Code, der in eine beliebige Klasse eingefügt wird, lässt sich unter Verwendung von Visual Studio 2010 für sämtliche Versionen .NET Framework 2.0 und höher kompilieren:

```
Property test As String
Sub DesignTimeCompatDemo()
        Dim query = Aggregate process In System.Diagnostics.Process.GetProcesses()
                    Into Average(process.Threads.Count) ' Beachten Sie: kein explizites
                    ' Zeilenfortsetzungszeichen verwendet
End Sub
```

Der obige Code verwendet die Leistungsmerkmale implizite Zeilenfortsetzung und selbstimplementierenden Eigenschaften, die mit der Visual Studio 2010 eingeführt wurden. Daher ist die erzeugte Assembly runtimekompatibel. Wenn Sie jedoch die gleiche Quellcodedatei nehmen und versuchen, diese unter den .NET-Versionen 3.5 oder 2.0 des Compilers zu kompilieren, werden Sie Fehler erhalten, da der 3.5-Compiler Sprachmerkmale nicht erkennt und die entsprechenden Stellen als ungültigen Code behandelt. Diese Einschränkung wird als fehlende Entwurfszeit-Kompatibilität von Quelldateien bezeichnet. Um diesen Quelldateien zu einer Entwurfszeit-Kompatibilität zu verhelfen, dürfen Sie nur Features verwenden, die im 3.5-Compiler vorhanden sind.

Keine optimale Unterstützung bei gemischten Zielen innerhalb einer Projektmappe In Verbindung mit den meisten Anwendungen wird der überwiegende Teil der Entwickler eine einzelne .NET Framework-Zielversion für alle Projekte einer Projektmappe verwenden. Wenn Ihre Anwendung jedoch mehrfache .NET Framework-Zielversionen innerhalb einer Projektmappe benötigt, müssen Sie einen Kompromiss in der IDE eingehen. Bei Visual Basic-Projekten mit Verweisen zwischen Projekten, die verschiedene .NET Framework-Zielversionsgruppen adressieren:

- 4.0 Client Profile, 4.0 Full Profile
- 2.0, 3.0, 3.5 Client Profile, 3.5 Full Profile
- .NET Framework-Zielversionsgruppen

werden diese Verweise als Dateiverweise behandelt. Haben Sie beispielsweise zwei Projekte in Ihrer Projektmappe, wobei das erste die .NET Framework-Version 4.0 adressiert und das zweite die .NET Framework-Version 3.5, wird der Projektverweis zwischen den beiden Projekten in einen Dateiverweis konvertiert. Die Einschränkung von Dateiverweisen liegt darin, dass es keine Unterstützung für Live-IntelliSense, Live-Editor-Unterringelungen, das Umbenennen von und Suchen nach allen projektübergreifenden Verweisen und die Funktion *Gehe zu Definition* bei Symbolen (gemeint sind Symbole im Sinne von Verweisen), die beide Projekte überspannen, gibt. Das ist ein ähnliches Verhalten, wie wenn Sie Projekte in Ihrer Projektmappe haben, die auf verschiedenen Programmiersprachen basieren, etwa Visual Basic und C#. Wenn Sie einen Verweis zwischen diesen .NET Framework-Gruppen hinzufügen, erhalten Sie die Meldung aus Abbildung 4.15. Die gute Nachricht ist, dass wenn Sie dann die .NET Framework-Zielversionen der Projekte so ändern, dass sie zusammenpassen, die Dateiverweise automatisch in projektbezogene Verweise konvertiert werden.

Abbildung 4.15 Meldung, wenn Projekte verschiedenen .NET Framework-Zielversionsgruppen unterliegen und untereinander verweisen

Verwendung von Fremdhersteller-Benutzer-Steuerelementen Der Dialog *Verweis hinzufügen* deaktiviert Systemassemblys, die nicht zu Ihrer .NET Framework-Zielversion gehören. (Systemassemblys sind *.dll*-Dateien, die Teil einer .NET Framework-Version sind.) Dies hilft, zu vermeiden, dass Verweise auf Assemblys hinzugefügt werden, die sich nicht auf Ihrem Zielsystem befinden. Wenn Sie Ihre Projektdatei ändern, sodass Verweise enthalten sind, die zu einer neueren .NET Framework-Version gehören als die, die von Ihrem Projekt momentan adressiert wird, wird der Verweis nicht aufgelöst. Zudem können Sie keine Steuerelemente hinzufügen und verwenden, die von solch einem Verweis abhängen. Diese Fähigkeit der Begrenzung von Benutzer-Steuerelementen verhindert außerdem das Hinzufügen von Steuerelementen, die unter Verwendung der .NET Framework 4.0 Full-Zielversion in Projekten erstellt wurden, in Projekten, welche die .NET Framework 4.0 Client Profile-Zielversion adressieren (welche die Vorgabe für neue Windows Forms- und WPF-Anwendungen darstellt). Dieses Problem kann gelöst werden, indem die .NET Framework-Zielversion Ihres Projekts auf eine Version umgestellt wird, die diesen Verweis enthält. Verwenden Sie hierfür die Projekteigenschaften wie weiter oben beschrieben. Eine weitere Möglichkeit besteht in der Erstellung einer benutzerdefinierten Vorlage für Clientprojekte, die auf das vollständige .NET Framework abzielt. Eine derartig modifizierte Windows Forms-Vorlage ist in den Begleitdateien des Buchs enthalten (in den Beispieldateien zu diesem Kapitel). Um die .NET Framework-Zielversion einer Vorlage zu ändern, passen Sie die dazugehörige *.Vbproj*-Datei in der Vorlage an.

Ein näherer Blick auf die neue und verbesserte WPF-basierte IDE

Wenn Sie die Marketingkampagnen für Visual Studio 2010 verfolgt haben, haben Sie sicher die Botschaft »neue, verbesserte WPF-IDE« gehört und haben sich vielleicht gefragt, was dies für einen Hintergrund hat. Während der Entwicklung von Visual Studio 2010 wurde ein besonderer Akzent gesetzt, um zu gewährleisten, dass Visual Studio ein modernes *Look and Feel* bekommt und dass einige neue und aufkommende Benutzerflächenelemente eingebaut werden. Die Notwendigkeit, solche Erweiterungsmöglichkeiten in Visual Studio einzubauen, wurde betont. In Anbetracht des mittlerweile etwas betagtem Codebestandes der Visual Studio-Komponenten konnte dies nur angegangen werden, indem der Code einem Upgrade unterzogen und indem dieser im stärkeren Maße komponentenorientiert gemacht wurde, was durch eine Trennung der Business-Logik von der Präsentationsschicht erzielt wurde. Als Technologie führt WPF naturgemäß zu dieser getrennten Präsentationsarchitektur, die auch häufig als *Model, View, Controller* oder *Model, View, Presenter* bezeichnet wird. Als Nebeneffekt der Integration von WPF wurde Visual Studio ein hervorragendes Beispiel dafür, dass große und komplexe Anwendungen erfolgreich mit WPF erstellt werden können. Obwohl alle wichtigen Benutzeroberflächenkomponenten wie die Visual Studio-Shell, die Startseite,

der Code-Editor, das Fensterverwaltungssystem usw. von der WPF bewältigt werden, gibt es einige Benutzeroberflächenelemente wie den Objektkatalog, die Klassenansicht usw. , die auf Windows Forms basieren und innerhalb eines WPF-Controls gehostet werden. Dies ist eine sehr interessante Mischung von Technologien. Wenn Sie sehen möchten, welche Teile der IDE in WPF geschrieben sind, können Sie *Snoop* verwenden (*http://snoopwpf.codeplex.com*), das ein WPF-Spy-Tool ist und mit dem sich die visuelle WPF-Struktur von Visual Studio untersuchen lässt (Abbildung 4.16).

Abbildung 4.16 Schnüffeln muss nicht immer anrüchig sein, das Tool *Snoop* hat seine Nase in der Visual Studio 2010-IDE

Bildschirmressourcen verwalten und effizient nutzen

Mit der Verlagerung der Programmier- und Designparadigmen hin zu einem modularen Ansatz wird das Anordnen mehrerer Code- und Designerdateien sowie Toolfenster zu einer immer individuelleren Geschichte. Visual Studio 2010 hat einen Schritt hin in Richtung mehr Anpassbarkeit unternommen, um IDE-Fenster innerhalb eines Bildschirms oder über mehrere Bildschirme verteilt anzuordnen.

Fenster teilen Stellen Sie sich vor, Sie haben eine große Visual Basic-Codedatei und würden gerne zwei unterschiedliche Teile derselben Datei zur gleichen Zeit sehen; die Unterstützung geteilter Fenster ist das passende Feature dafür. Gehen Sie dazu folgendermaßen vor:

1. Klicken Sie auf das Teilensymbol in der rechten oberen Ecke des Codefensters (in Abbildung 4.17 zum besseren Auffinden hervorgehoben).
2. Ziehen Sie die Fensterteilung bei gedrückt gehaltener Maustaste nach unten an die gewünschte Position.

Alternativ wählen Sie *Teilen* aus dem *Fenster*-Menü.

Abbildung 4.17 Optionen für die Verwaltung von Dokumentfenstern

Abbildung 4.18 zeigt, wie die Datei *Form1.vb* in zwei Fensterbereiche geteilt wurde, einer zeigt das `Button1_click`-Ereignis, während der andere das `Form1_Load`-Ereignis präsentiert.

Abbildung 4.18 Horizontal geteilte Fenster

Jeder Bereich des geteilten Fensters kann einen anderen Teil des Codes zeigen und besitzt Fähigkeiten zum Scrollen und Zoomen. Um den Fokus zwischen den geteilten Fenstern zu wechseln, klicken Sie entweder innerhalb des zu fokussierenden Fensterbereichs oder drücken Sie F6 bzw. ⇧+F6. Beide Fensterbereiche haben eigene *Identitäten* in dem Sinn, dass bei der Bearbeitung des Codes in einem Fensterbereich der andere Bereich nicht mitscrollt. Hingegen sind allgemeine Elemente wie Listenfelder und das Anspringen der Fehlerkorrektur via Fehlerliste mit dem aktiven Fensterbereich verdrahtet. Erweiterte oder wieder geschlossene Codeblöcke/-regionen sind in allen Fensterbereichen gleich. Wenn ein Codeblock in einem Fensterbereich erweitert (eingeblendet) wird, wird er also in allen Fensterbereichen erweitert dargestellt. Abbildung 4.19 zeigt verschiedene Hervorhebungen von Verweisen, die in den beiden Fensterbereichen der Datei *Form1.vb* ausgelöst wurden.

Abbildung 4.19 Hervorgehobene Verweise (hier: *myvar*, *Prop*, jeweils grau unterlegt) in geteilten Fenstern

Horizontale und vertikale Registerkartengruppe

Die Dokumentfenster können horizontal oder vertikal angeordnet werden. Die entsprechenden Optionen stehen über das *Fenster*-Menü zur Verfügung. Abbildung 4.20 zeigt horizontal angeordnete Dokumente, Abbildung 4.21 vertikal angeordnete.

Abbildung 4.20 Horizontal angeordnete Dokumentfenster

Abbildung 4.21 Vertikal angeordnete Dokumentfenster

Unterstützung mehrerer Bildschirme und abdockbare Fenster

Bildschirme wurden bezahlbar und viele Entwickler verwenden zwei oder mehr Bildschirme für ihre tägliche Arbeit. Visual Studio 2010 erlaubt Entwicklern, ihre Code- und Entwurfsfenster über mehrere Bildschirme auszubreiten, indem es ermöglicht, Dokument- und Toolfenster aus der IDE herauszuziehen. Dazu klicken Sie das Registerkartensymbol des Fensters an (die Stelle, an welcher der Dateiname oder der Name des jeweiligen Elements vermerkt ist), ziehen Sie es aus dem Visual Studio-Fenster heraus und legen es an der gewünschten Stelle ab. Dies kann auch erreicht werden, indem das Fenster mit [⊞] + Pfeiltasten bewegt wird. Um das Fenster wieder anzudocken, klicken Sie die Titelleiste des Fensters an, ziehen Sie das Fenster zurück in den IDE-Bereich und richten die Titelleiste an der Registerkartengruppe der anderen Fenster aus. Beim Wiederandocken erscheint ein Diamant-Führungssymbol (Abbildung 4.22), das sich verwenden lässt, um beim korrekten Ausrichten des Fensters eine optische Kontrolle zu haben. Zum Wiederandocken können Sie auch [Strg] gedrückt halten und die Titelleiste des Fensters doppelt anklicken.

Abbildung 4.22 Das Diamant-Führungssymbol beim Bewegen des Fensters *Fehlerliste*

Beständigkeit des Fensterlayouts

Die Dokument-Bildschirmposition wird in dem Visual Studio-Speicherbereich für jeden Benutzer auf dem lokalen Computer gespeichert. Das bedeutet, wenn Sie an der gleichen Projektmappe an verschiedenen Computern arbeiten, können Sie gesonderte Fensterpositionen auf verschiedenen Computern aufrechterhalten. Die Konsequenz ist zudem, dass jeder Benutzer sein Fensterlayout beibehalten kann, wenn an Projekten im Team unter Verwendung der Quellcodeverwaltung gearbeitet wird. Ändert sich die Anzahl der an den Computer angeschlossenen Bildschirme, erkennt Visual Studio, ob ein zweiter Bildschirm verfügbar ist oder nicht. Wenn nicht, verschiebt es die dem zweiten Bildschirm zugewiesenen Dokumente auf den ersten. So ist sichergestellt, dass Sie keine Dokumente »verlieren«.

Gebräuchliche Szenarien Obwohl das Dokumentlayout eine individuelle Angelegenheit ist, finden Sie unten einige Fälle, bei denen die Fensterfunktionen von Visual Studio 2010 Ihnen sicher helfen, die Produktivität zu erhöhen.

- Betrachten Sie Entwurf- und Codedatei gleichzeitig
- Betrachten Sie Anwendungs- und Testtool-Fenster gleichzeitig
- Betrachten Sie mehrere Codedateien gleichzeitig

Normale Ansicht In der normalen bzw. standardmäßigen Ansicht werden verschiedene Dokumentfenster als Registerkartengruppe angeordnet (Abbildung 4.23).

```
Alarmgeber.vb   frmMain.vb   PolarCoordinate.vb
(Allgemein)                  (Deklarationen)
Imports System.Collections.ObjectModel

Public Structure PolarCoordinate
    Private myLength_X As Single
    Private myLength_Y As Single
    Private myAngle As Single
    Private myOffset As PointF
    Private myCartesian As PointF
    Private myOriginalOffset As PointF
    Private myOriginalAngle As Single

    Sub New(ByVal Offset As PointF, ByVal Length As Single, ByVal Angle As Single)
        myLength_X = Length
        myLength_Y = Length
        myAngle = Angle
        myOriginalAngle = Angle
        myOffset = Offset
        myOriginalOffset = Offset
        recalculate()
    End Sub

    Sub New(ByVal Offset As PointF, ByVal Length_X As Single, ByVal Length_Y As Single, ByVal Angle As Single)
        myLength_X = Length_X
        myLength_Y = Length_Y
        myAngle = Angle
        myOriginalAngle = Angle
        myOffset = Offset
        myOriginalOffset = Offset
        recalculate()
    End Sub

    ''' <summary>
    ''' Liefert die Polarkoordinate als Kartesische Koordinate zurück
    ''' </summary>
    ''' <value></value>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public ReadOnly Property Cartesian() As PointF
100 %
```

Abbildung 4.23 Angedockte Ansicht von Dokumentfenstern

Fenster verschieben In Visual Studio 2010 können Sie Fenster so darstellen, dass sie beliebig (pixelweise) verschiebbar sind. Die Fenster schwimmen (*floating Windows Style*) gewissermaßen innerhalb und außerhalb der IDE (Abbildung 4.24).

Abbildung 4.24 Darstellung von verschiebbaren Fenstern

Fenster andocken Darüber hinaus können Sie auch das Fenster andocken, indem Sie die Titelleiste eines Fensters bei gedrückt gehaltener Maustaste an eine Position ziehen. Dabei wird das Diamant-Führungssymbol eingeblendet, welches das Andocken von Fenstern erleichtert. Abbildung 4.25 zeigt die Möglichkeit, die Datei *PolarCoordinate.vb* entweder in der Mitte, oben, unten, links oder rechts anzudocken.

Abbildung 4.25 Das Diamant-Führungssymbol erleichtert das Andocken eines Dokumentfensters

Neben dem Andocken innerhalb der IDE erlaubt es Visual Studio 2010 Ihnen zudem, auch AUSSERHALB der IDE anzudocken. Fenster außerhalb der IDE lassen sich maximieren, um den gesamten Bildschirmplatz auszunutzen, oder nebeneinander bzw. untereinander in verschiedenen Konfigurationen platzieren. Wie Sie bislang gesehen haben, sind die Kombinationsmöglichkeiten beim Fensterlayout quasi unendlich, und es liegt beim Entwickler, die maximale Produktivität herauszuholen.

Code: navigieren, suchen und nachvollziehen

RAD (Rapid Application Development) ist ein Muss in der heutigen Ära der Softwareentwicklung. Das bringt die Notwendigkeit guter Tools hervor, mit denen sich auf einfache Weise im Code navigieren, in diesem suchen und sich dieser nachvollziehen lässt. In diesem Abschnitt werden einige Leistungsmerkmale von Visual Studio 2010 beleuchtet, die Entwicklern ein produktives Suchen und Navigieren im Code ermöglichen.

Navigieren zu

Diese Funktion erlaubt Ihnen, nach Symbolen und Dateien innerhalb einer Projektmappe zu suchen und diese anzusteuern. Zum Aufruf des Dialogs *Navigieren zu* wählen Sie *Bearbeiten/Navigieren zu* oder verwenden Sie die neue Tastenkombination Strg+,. Der Dialog (Abbildung 4.26) bietet eine schnelle und inkrementelle Suche quer durch Visual Basic-, C#- und C++-Projekte der Projektmappe. *Inkrementell* bedeutet, dass die Suchergebnisse bereits während des Eintippens des Suchbegriffs ständig, quasi live, aktualisiert werden. Jeder Tastendruck filtert die Ergebnisse neu, und wenn Zeichen entfernt werden, wird die Ergebnisliste ebenso neu gefiltert. Um an die Position eines gefundenen Symbols zu navigieren, klicken Sie doppelt auf einen Ergebniseintrag (oder drücken Sie ↵ in Verbindung mit dem derzeit ausgewählten Eintrag in der Ergebnisliste). Wenn die Datei nicht bereits im Code-Editor geöffnet ist, wird diese automatisch geöffnet und das Symbol im Code hervorgehoben.

Abbildung 4.26 Dialog *Navigieren zu*

Intelligente Suchfunktionen Der Dialog *Navigieren zu* unterstützt einige nette Fuzzy-Suchfunktionen, die es Ihnen erlauben, intelligente Filter- und Suchvorgänge vorzunehmen, ohne dass Sie den genauen Namen des Elements, nach dem Sie suchen, kennen müssen. Diese Suchcharakteristika werden im Folgenden beschrieben:

- Es werden auch Teilstrings berücksichtigt. Das bedeutet, dass Begriffe am Anfang, am Ende oder irgendwo in der Mitte gefunden werden. Der Suchbegriff *name* findet zum Beispiel die Symbole ***Name**Ermitteln*, *Erzeuge**Name**Neu* und myVor***name***.
- Mehrfache Teilstringsuche: Um Elemente zu finden, die zwei verteilte Zeichensequenzen enthalten, geben Sie beide Sequenzen getrennt durch ein Leerzeichen ein. Beachten Sie, dass die Reichenfolge dieser Sequenzen keine Rolle spielt. Beispielsweise findet *adresse neu* Symbole wie ***Adresse**nGenerierenNeu* und ***Neu**eEMailAdresse**Anlegen***. Als Ersatz für das Leerzeichen kann auch das Jokerzeichen »*« verwendet werden.
- Eine Pascal Case-Suche wird unterstützt. Typen und Member innerhalb von .NET Framework verwenden das Namenentwurfsmuster *Pascal Case* – was bedeutet, dass der erste Buchstabe jedes Teilwortes bei einem Typen oder Member großgeschrieben wird. Der Dialog *Navigieren zu* erlaubt es Ihnen optional, diese Pascal Case-Konvention zu verwenden, um auf schnelle Art und Weise Typen zu filtern. Geben Sie einfach die (Groß-)Buchstaben ein, mit denen die einzelnen Teilnamen eines Typen oder Members beginnen. Daraufhin werden automatisch die Begriffe gefunden, die der Pascal Case-Namenskonvention entsprechen. *GWD* findet zum Beispiel Typen und Member, die Teilwörter aufweisen, die mit G, W und D in der angegebenen Reihenfolge beginnen, etwa *GetWeatherData*. Bei der Pascal Case-Suche ist die Großschreibung bei der Eingabe wichtig, Sie müssen also im Beispiel *GWD* schreiben, *gwd* würde nicht funktionieren.
- Wenn der Suchstring Großbuchstaben enthält, wird bei der Suche Groß- und Kleinschreibung beachtet, andernfalls nicht
- Die Suchergebnisse weisen in der Projektmappe vorkommende Symboldefinitionen und Dateinamen auf, aber keine Namespaces und lokalen Variablen

Zwischen Codeblöcken navigieren

Um zwischen Methoden zu navigieren, verwenden Sie die Menüeinträge *Nächste Methode* und *Vorherige Methode* im *Bearbeiten*-Menü oder Strg+↑ bzw. Strg+↓.

Hervorheben und Navigieren zwischen Symbolverweisen

Das Hervorheben von Verweisen ist ein sehr einfaches und bequemes Konzept, das hilft, ein konzentriertes Codestück nachzuvollziehen. Es ist ein Feature, das man sich als »Finde alle Verweisstellen innerhalb des geöffneten Dokuments« vorstellen kann. Angenommen, es wäre notwendig herauszufinden, wo überall in einer Datei eine spezifische Überladung einer Methode aufgerufen wird. Mit der *Verweise hervorheben*-Funktion kann dies mit einem Klick erreicht werden. Platzieren Sie die Einfügemarke einfach in der spezifischen Methode (Deklaration oder Aufruf), woraufhin die Deklaration und alle Aufrufstellen für diese Überladung hervorgehoben werden. Um sich einen Gesamtüberblick zu verschaffen, müssen Sie nur noch nach oben oder unten scrollen. Für das Navigieren zum nächsten oder vorangehenden Verweis drücken Sie dabei Strg+⇧+↑ bzw. Strg+⇧+↓. Mit der Einschränkung, dass das Feature nicht in XML Doc-Abschnitten und XAML arbeitet, ist es ein nützliches Merkmal, um durch die Struktur einer Codedatei zu navigieren. In Abbildung 4.27 werden Verweise der Methode `DrawString` hervorgehoben.

Abbildung 4.27 Das Feature *Verweise hervorheben* im Einsatz

Eine weitere Erweiterung dieses Features liegt in der Hervorhebung von Elementen, die den Codefluss eingrenzen: Module, Klassen, Schleifenkonstrukte, `try … catch`-Blöcke usw. In Abbildung 4.28 wurde ein `try … catch`-Block hervorgehoben.

Abbildung 4.28 Einen `Try … Catch`-Block hervorheben

HINWEIS Damit das normale Tippen nicht beeinträchtigt wird, wird die *Verweise hervorheben*-Funktion erst 750 ms, nachdem die Einfügemarke auf das Symbol gesetzt wurde, aktiviert.

Der Wirkungsbereich des *Verweise hervorheben*-Features ist dateibasiert. Jedoch gibt es hier noch einige Punkte anzumerken: Wenn mehrere Dateien geöffnet sind, hat jede Ansicht ihre eigene Verweishervorhebung. Das bedeutet, dass in einer Datei die Methode `DrawString` hervorgehoben sein kann und in der anderen die Methode `ShowDialog`. Das gilt auch dann, wenn mehrere Instanzen ein und derselben Datei zur gleichen Zeit geöffnet sind. Jede Ansicht der Datei besitzt dann ihre eigene Verweishervorhebung. Für geteilte Fenster gilt das Gleiche wie für mehrfache Fenster. Demzufolge kann es unterschiedliche Hervorhebungen in jedem Fensterteilbereich geben.

Die Hervorhebungsfarbe ändern

Die Hervorhebungen werden standardmäßig in Grau angezeigt. Die Hervorhebungsfarbe kann über *Extras/Optionen/Umgebung/Schriftarten und Farben* konfiguriert werden. Klicken Sie zunächst auf den Eintrag *Hervorgehobener Verweis* im Listenfeld *Elemente anzeigen*. Ändern Sie dann die Farbe unter *Elementhintergrund* nach Ihren Vorstellungen. In Abbildung 4.29 wurde die Farbe auf Pink gesetzt. Eine Anpassung der Farbe unter *Elementvordergrund* bewirkt, dass die Farbe des Rahmens, mit dem die Hervorhebung umrandet wird, geändert wird.

Abbildung 4.29 Die Farbe der Hervorhebungen ändern

Verweise hervorheben-Funktion abschalten

Um die Hervorhebung von Verweisen abzuschalten, wählen Sie *Extras/Optionen/Text-Editor/Basic/VB-spezifisch* und schalten Sie das Kontrollkästchen *Markierung von Verweisen und Schlüsselwörtern aktivieren* ab (Abbildung 4.30). Diese Einstellung wirkt sich dauerhaft auf alle IDE-Sitzungen aus.

Abbildung 4.30 Die *Verweise hervorheben*-Funktion aktivieren und deaktivieren

Regionen und Gliederungen

Im Falle großer Codedateien ist es oft sehr hilfreich, einige Codeblöcke auszublenden, um die Struktur der Datei zu sehen. Um diese gewünschte Komprimierung einer Datei zu erreichen, können in Visual Studio Regionen und Gliederungs-Menüfunktionen verwendet werden.

Regionen lassen sich dabei durch Eingabe von `#Region "Name der Region"` einleiten. Eine Region endet mit `#End Region`. Die Regionen können mit den Gliederungssymbolen »+« und »-« am linken Rand eingeblendet (also die Struktur erweitert) und ausgeblendet werden. Abbildung 4.31 zeigt eine eingeblendete und eine ausgeblendete Region.

```
Class test
private members
#Region "public members"
    Public Sub MyPublicMethod()

    End Sub
    Public Function MyFunction1() As Boolean
        Return False
    End Function
#End Region
End Class
```

Abbildung 4.31 Regionen und die dazugehörigen Gliederungssymbole

Neben den Regionen gibt es verschiedene andere Gliederungsfunktionen, die über das Kontextmenü und das Visual Studio-Menü *Bearbeiten/Gliedern* zur Verfügung stehen. Markieren Sie gegebenenfalls zunächst den gewünschten Codebereich. Das Kontextmenü erscheint nach einem Rechtsklick; die entsprechenden Einträge finden sich als Untereinträge von *Gliedern*.

- **Aktuelles Element umschalten** – blendet den derzeit markierten Text aus. Es muss ein Text markiert sein, damit dieser Befehl ausgeführt werden kann. Tastenbefehl: Strg+M und dann Strg+H. Diese Funktion kann verwendet werden, um Regionen zu simulieren. Sie erreichen dann etwas Vergleichbares wie zum Beispiel bei einem Methodenkörper, obwohl der aus- und einzublendende Bereich beliebigen Umfang haben kann und keinen Codeblock im eigentlichen Sinne darstellen muss.
- **Gliederungserweiterung umschalten** – kehrt den aktuellen aus- oder eingeblendeten Zustand im innersten Gliederungsbereich, in dem sich die Einfügemarke befindet, um, wenn Sie sich in einem verschachtelten Abschnitt befinden. Tastenbefehl: Strg+M und ein weiteres Mal Strg+M.
- **Alle Gliederungen umschalten** – legt für alle Prozeduren den gleichen ausgeblendeten oder eingeblendeten Zustand fest. Wenn einige Regionen eingeblendet und andere ausgeblendet sind, werden die ausgeblendeten Regionen eingeblendet. Tastenbefehl: Strg+M und dann Strg+L.
- **Gliederung entfernen** – entfernt alle Gliederungsinformationen für das gesamte Dokument. Alle Prozeduren sind dann sichtbar und die Gliederungssymbole werden entfernt, aber der zugrunde liegende Code bleibt unangetastet. Tastenbefehl: Strg+M und dann Strg+P.
- **Gliederung in aktuellem Element entfernen** – entfernt die Gliederungsinformationen für die momentan ausgewählte, benutzerdefinierte Region. Tastenbefehl: Strg+M und dann Strg+U.
- **Nur Definitionen anzeigen** – blendet die Member aller Typen dergestalt aus, dass alle zu einem Typ gehörenden Member deutlich zu erkennen sind. Tastenbefehl: Strg+M und dann Strg+O. Entspricht dem Befehl *Block reduzieren* mit dem Unterschied, dass die Umgebung automatisch sämtlichen Code im Dokument durchläuft, in den einzelnen Prozeduren Regionen erstellt und diese dann ausblendet.

Architektur-Explorer

In Visual Studio Ultimate können Sie mit dem Architektur-Explorer Ihren Code in Gestalt von vertikalen Bereichen oder »Scheiben« darstellen. Diese Visualisierung ermöglicht das Verstehen des Codes, indem Knoten für Knoten in der Codehierarchie nach unten navigiert wird. Man spricht hierbei auch von *drill-down*, zu Deutsch etwa *auf den Grund gehen*. Der Architektur-Explorer lässt sich über *Architektur/Fenster/Architektur-Explorer* oder aus einem Sequenzdiagramm aufrufen. In Abbildung 4.32 wird die *Klassenansicht* verwendet, um bis hinunter zu den Aufrufen, die eine bestimmte Methode (`btnDruckAufStandarddrucker_Click` im Beispiel) durchführt, zu navigieren.

Abbildung 4.32 Der Architektur-Explorer

Sequenzdiagramme

Ein Bild sagt mehr als tausend Worte. Visual Studio 2010 unterstützt die spontane Erzeugung von Sequenzdiagrammen einer ausgewählten Methode. Zur Erzeugung klicken Sie die gewünschte Methodendeklaration mit der rechten Maustaste an und wählen *Sequenzdiagramm generieren* aus dem Kontextmenü. Daraufhin wird ein Dialog angezeigt, über den die Diagrammerzeugung beeinflusst werden kann. Konkret legen Sie fest, was genau visualisiert werden soll und wie hoch die Aufruftiefe sein soll.

Abbildung 4.33 zeigt ein Beispiel für ein Sequenzdiagramm, das die Aufrufhierarchie, eine Schleife und eine `If`-Abfrage der Methode `Start_DedicatedTaskParallelFor` aus dem Projekt *ImageResizer* veranschaulicht. Das erzeugte Sequenzdiagramm kann bearbeitet werden und es gibt auch Navigationsfunktionen. Dazu stehen entsprechende Befehle im Kontextmenü des Sequenzdiagramms zur Verfügung.

Abbildung 4.33 Ein Sequenzdiagramm

Klassendiagramme

Ein Klassendiagramm erlaubt es Ihnen auf visuelle Art und Weise Klassen, Strukturen, Enums, Methoden, Schnittstellen, Delegaten und mehr zu erstellen, zu bearbeiten und zu verwalten. Um die Klassendiagrammansicht zu öffnen, klicken Sie auf das Symbol *Klassendiagramm anzeigen* im Projektmappen-Explorer. Das Symbol ist in Abbildung 4.34, das ein Beispiel für ein Klassendiagramm zeigt, zum besseren Auffinden hervorgehoben.

Abbildung 4.34 Ein Klassendiagramm

Bottom-up-Programmierung

IntelliSense ist eines der am meisten verwendeten Features und hat den Haupteinfluss darauf, wie schnell der Code von der Hand geht. In den vorangegangenen Versionen des Visual Basic-Editors basierte IntelliSense auf dem Top-down-Paradigma (von oben nach unten) beim Schreiben von Code. In Verbindung mit dem neuen Programmierparadigma *testgesteuerte Entwicklung* wurde IntelliSense erweitert und unterstützt

jetzt auch den Bottom-up-Programmierstil (von unten nach oben). Da dies eine Erweiterung der bestehenden IntelliSense-Funktionalität darstellt, besitzt IntelliSense in Visual Studio 2010 zwei Modi: den bisherigen *Beendigungsmodus* und den neuen *Vorschlagsmodus*. Zwischen diesen beiden Modi kann per [Strg]+[Alt]+[ ] oder über *Bearbeiten/IntelliSense/Beendigungsmodus umschalten* gewechselt werden.

Beendigungsmodus (*erst deklarieren*) Wenn Sie vorangegangene Versionen von IntelliSense benutzt haben, bedeutet dies für Sie, dass IntelliSense wie gewohnt arbeitet. In diesem Modus zeigt IntelliSense die Elemente, die im Code vom Aufrufort aus gesehen deklariert sind, und erlaubt Ihnen ausschließlich, eine dieser Deklarationen auszuwählen. Abbildung 4.35 zeigt IntelliSense in diesem Modus.

Abbildung 4.35 IntelliSense im bekannten Beendigungsmodus

Vorschlagsmodus (erst anwenden)

In diesem Modus haben Sie die Flexibilität, Codeelemente bereits verwenden zu können, bevor sie deklariert werden. Beachten Sie hierfür Abbildung 4.36. Dort weist IntelliSense einen zusätzlichen Eintrag auf, der den vom Benutzer eingetippten Elementnamen anzeigt.

Abbildung 4.36 IntelliSense im neuen Vorschlagsmodus

Neben der Möglichkeit, ein anderes Programmparadigma verwenden zu können, gibt es eine ganze Reihe weiterer hilfreicher Funktionen, die in IntelliSense neu in Visual Studio 2010 aufgenommen wurden. Einige davon werden im Folgenden aufgelistet:

Teilstringfilterung anstelle von Präfixfilterung Eine hervorragende Funktionalität von IntelliSense war immer schon, dass die Liste bereits während des Eintippens gefiltert wird. Dies hilft, die infrage kommenden Einträge immer weiter auf das Element zu reduzieren, das der Benutzer auswählen und verwenden will. Dies hilft jedoch nicht weiter, wenn sich der Benutzer unsicher ist, mit welchen Zeichen der Name beginnt,

nach dem er Ausschau hält. Mit Visual Studio 2010 basiert die Filterung von IntelliSense nun auf Teilstrings. Beginnt der Benutzer beispielsweise, **dir** im Methodenkörper einzutippen, zeigt IntelliSense alle Verzeichnisverwaltungsmethoden an, also auch `ChDir` (Abbildung 4.37).

Abbildung 4.37 Die Filterung von IntelliSense arbeitet jetzt basierend auf Teilstrings

Pascal Case-Filterung in IntelliSense

Pascal Case-Filterung bezeichnet die Fähigkeit, nach einer Phrase aus mehreren Teilwörtern über eine einzelne Abkürzung zu suchen. Das abgekürzte Wort muss streng aus Großbuchstaben bestehen, um der IntelliSense-Engine verstehen zu geben, dass Pascal Case gemeint ist.

Beispiel: Bei Eingabe von **Dim x as DC** bietet IntelliSense das in Abbildung 4.38 Gezeigte an und nicht etwa `DataRowCollection`.

Abbildung 4.38 IntelliSense unterstützt nun die Pascal Case-Filterung

HINWEIS Um zwischen den Registerkarten *Allgemein* und *Alle* im IntelliSense-Fenster umzuschalten, kann auch `Alt`+`,` und `Alt`+`.` verwendet werden. Eine weitere Hilfe: Wenn Sie `Strg` gedrückt halten, wird die IntelliSense-Liste halbtransparent dargestellt, sodass der Code durchschimmert und damit lesbar ist.

Wenn Sie frühere Versionen von Visual Studio verwendet haben, haben Sie möglicherweise eine Veränderung beim Verhalten von IntelliSense in Verbindung mit Enums festgestellt. In früheren Versionen war die Enum-Liste von IntelliSense flach und enthielt nur die Enum-Member, aber in Visual Studio 2010 gibt es zwei Registerkarten: *Allgemein* und *Alle*, wobei Erstere die Enum-Member anbietet und Letztere eine allgemeine Liste. Dies kann in einigen Fällen den Tippfluss stören. Dem Visual Basic-Team ist dies bewusst und es ist wahrscheinlich, dass dieses Problem zusammen mit Enum-ähnlichen Listen (zum Beispiel Boolean, Farben) in zukünftigen Versionen angegangen wird.

Generate From Usage

Das Feature *Generate From Usage*, auf Deutsch *beim Einsatz generieren*, ermöglicht es Ihnen, Klassen und Member an dem Ort, an dem auf sie zugegriffen werden soll, zu generieren. Sie können eine Stub[2] für jedes Element (Klasse, Konstruktor, Methode, Eigenschaft, Feld oder Enum), das Sie verwenden möchten, aber noch nicht definiert haben, generieren, ohne dass Sie dazu die aktuelle Position im Code verlassen müssen. Dies minimiert Unterbrechungen in Ihrem Arbeitsfluss. Dieses Feature wurde als Erweiterung der Fehlerkorrekturfunktionalität in Visual Basic hinzugefügt. Es wird aufgerufen, indem der Mauszeiger auf den Smarttag (das schmale rote Rechteck) bewegt, dann auf das rote, eingekreiste Ausrufezeichen geklickt und schließlich der jeweilige Eintrag ausgewählt wird. Alternativ kann auch das dazugehörige Element in der Fehlerliste doppelt angeklickt werden. Eine dritte Möglichkeit besteht in einem Druck auf ⇧+Alt+F10 und eine vierte über Strg+..

Das Generate From Usage-Fetaure kann verwendet werden, um die nachfolgend angegebenen Codeelemente abhängig vom Kontext zu erzeugen:

- *Eigenschaftsstub generieren*
- *Feldstub generieren*
- *Methodenstub generieren*
- *Klasse generieren*
- *Schnittstelle generieren*
- *Neuen Typ generieren* (für eine Klasse, Struktur, Schnittstelle, einen Enum, Delegaten oder ein Modul)

Typen anpassen mit dem Dialog Neuen Typ generieren

Die Option *Neuen Typ generieren* bietet zusätzliche Flexibilität beim Erzeugen einer Klasse, Struktur, Schnittstelle, eines Enums, Delegaten oder Moduls. Dabei werden Optionen angeboten, mit denen sich festlegen lässt, um was für ein Element es sich handelt, welcher Zugriffsmodifizierer eingesetzt wird sowie ob das zu erstellende Element in eine bestehende Projektdatei (in der aktuellen Projektmappe) aufgenommen oder dafür eine neue Datei erstellt werden soll. Die konkreten Optionen entnehmen Sie Tabelle 4.4. Der Dialog *Neuen Typ generieren* ist in Abbildung 4.39 zu sehen.

[2] Sprich: »stap«. Ein stellvertretender, meist kurzer Code, der später durch den endgültigen ersetzt wird.

Abbildung 4.39 Der Dialog *Neuen Typ generieren*

Option	Auswahlmöglichkeiten
Zugriff	Standard (es wird kein Zugriffsmodifizierer eingesetzt), `Friend` oder `Public`
Art	`Class`, `Structure`, `Interface`, `Enum`, `Delegate` oder `Module`
Speicherort/Projekt	Aktuelles Projekt oder ein anderes Projekt in der Projektmappe
Speicherort/Dateiname	Eine neue Datei erstellen mit Angabe des Dateinamens oder zu vorhandener Datei hinzufügen

Tabelle 4.4 Auswahlmöglichkeiten des Dialogs *Neuen Typ generieren*

Wenn der Typ in einer neuen Datei generiert wird, enthält die neue Datei die standardmäßigen Importe für Klassen, die auch bei Verwendung des Dialogs *Neues Element hinzufügen* erzeugt werden. Wird der Typ in einem anderen Projekt der Projektmappe generiert, werden den Projekten automatisch Projektverweise und Importe hinzugefügt.

Visual Studio erweitern

Eine der leistungsfähigsten Funktionalitäten, die in Visual Studio 2010 eingeführt wurde, ist die Erweiterbarkeit der IDE. Die WPF-Architektur hat zu leicht erweiterbaren Hooks (spezielle Schnittstellen) in den UI-Elementen von Visual Studio geführt. Beispielsweise können die Startseite und das IntelliSense-Fenster leicht angepasst werden, um individuelle Anforderungen zu erfüllen. Ebenso lassen sich neue Refactoring-Funktionen (mehr zum Refactoring in Kapitel 5) einfach im Visual Basic-Editor hinzufügen. Inzwischen sind zahlreiche freie Erweiterungen online verfügbar, einschließlich Productivity Power Tools von Microsoft.

Die Startseite enthält einige hilfreiche Links unter dem Abschnitt *Erweitern von Visual Studio*. Da das Erstellen von Erweiterungen eine Community-Akivität ist, kann es Erweiterungen geben, die in Konflikt stehen mit anderen.

Visual Studio-Erweiterungen verwalten

Angesichts der hohen Anzahl verfügbarer Erweiterungen (1.045 im August 2010), die auch noch weiter zunimmt, wurde es eine Notwendigkeit, dass sich die Erweiterungen einsehen, nach ihnen suchen und diese auswählen (jeweils auch online) sowie lokal installierte Erweiterungen wieder entfernen lassen. Visual Studio 2010 besitzt hierfür den *Erweiterungs-Manager* (Abbildung 4.40), mit dem Sie Visual Studio-Erweiterungen hinzufügen, entfernen, aktivieren und deaktivieren können. Um den Erweiterungs-Manager zu öffnen, wählen Sie *Extras/Erweiterungs-Manager*.

Abbildung 4.40 Dialog *Erweiterungs-Manager*

Das Fenster des Erweiterungs-Managers ist in drei Bereiche aufgeteilt. Der linke Bereich kategorisiert die Erweiterungen nach installierten Erweiterungen, neuen Erweiterungen aus dem Onlinekatalog und für bereits installierte Erweiterungen verfügbare Updates. Im mittleren Bereich werden die einzelnen Erweiterungen zur Auswahl angeboten. Die Auswahl hängt von der im linken Bereich angeklickten Kategorie ab. Die Liste kann nach verschiedenen Kriterien sortiert und gefiltert werden. Der rechte Bereich bietet Informationen zur ausgewählten Erweiterung an. Wenn die jeweilige Erweiterung noch nicht installiert ist, können Sie diese herunterladen. Ist diese dagegen bereits installiert, lässt sie sich deaktivieren, aktivieren oder deinstallieren.

Der Erweiterungs-Manager kann Erweiterungen von der *Visual Studio Gallery* (*http://visualstudiogallery.msdn.microsoft.com/de-de/*) der MSDN-Website installieren. Diese Erweiterungen können Pakete, Vorlagen oder andere Komponenten sein, die zusätzliche Funktionalität in Visual Studio bringen.

Erweiterungsarten

Der Erweiterungs-Manager unterstützt Erweiterungen im VSIX-Package-Format, das Projektvorlagen, Elementvorlagen, Toolbox-Elemente, Managed Extension Framework (MEF)-Komponenten und VSPackages enthalten kann. Darüber hinaus ist der Erweiterungs-Manager auch in der Lage, MSI-basierte Erweiterungen zu laden und zu installieren, kann diese aber nicht aktivieren und deaktivieren. Visual Studio Gallery enthält sowohl VSIX- als auch MSI-Erweiterungen. Visual Studio unterstützt weiterhin Verteilungstechnologien wie Visual Studio Content Installer (VSI) und Microsoft Installer (MSI), aber nicht durch den Erweiterungs-Manager. Das VSI-Format kann für Makros, Add-Ins, Codeausschnitte und einige andere Visual Studio-Erweiterungsarten eingesetzt werden. Das MSI-Format, das im großen Umfang für Anwendungen Verwendung findet, lässt sich auch für Erweiterungen nutzen. Sowohl *.vsi*- als auch *.msi*-Dateien bestehen aus kompletten Paketen, die verteilt und per Doppelklick installiert werden können.

HINWEIS Viele weitere Features der IDE (Windows Forms-Designer, Code-Editor, WPF-Designer und der XAML-Editor) werden per Learning by Doing in Kapitel 5 und 6 erklärt.

Kapitel 5

Einführung in Windows Forms – Designer und Code-Editor am Beispiel

In diesem Kapitel:

Das Fallbeispiel – der DVD-Hüllen-Generator »Covers« 122
Gestalten von Formularen mit dem Windows Forms-Designer 128
Der Code-Editor 157

Wenn es darum geht, Windows Forms-Anwendungen, die auch unter dem Modenamen »Smart Clients« gehandelt werden, zu entwickeln, dann sind der Formular-Designer und der Code-Editor die Werkzeuge, die Sie neben den Debugging-Werkzeugen wohl am häufigsten verwenden werden. Darum sei ihnen auch ein eigenes Kapitel gewidmet.

Nun gibt es zwei nahe liegende Ansätze, die wichtigen Werkzeuge einer Entwicklungsumgebung zu erklären: eine theoretische und eine praktische Vorgehensweise. Die theoretische gibt Ihnen einen schnellen Überblick, was für Möglichkeiten es bei dem einen oder anderen Werkzeug gibt, macht Sie aber nicht praxissicher. Die praktische bringt ungleich mehr: Vielleicht unterfordert Sie das Beispiel aus Entwicklersicht ein wenig, das Sie dazu am besten von vorne bis hinten durchexerzieren müssen; aber auf diese Weise lernen Sie am schnellsten die Feinheiten und Möglichkeiten der Werkzeuge kennen, und Sie sind viel besser für die Praxis gerüstet.

Aus diesem Grund habe ich mich dazu entschieden, die Vorstellung der Möglichkeiten von Formular-Designer und Code-Editor an einem konkreten Beispiel vorzuführen. Das Beispiel geht sogar noch ein wenig über die thematischen Kapitelgrenzen hinaus und liefert Ihnen schon jetzt die eine oder andere Info zu völlig anderen Themen, die mir aber schon zu diesem Zeitpunkt im Kontext sinnvoll erscheinen. Natürlich werden fast alle Punkte im Laufe des Buchs noch weiter vertieft – und denken Sie daran, dass es in erster Linie darum geht, Ihnen schnellstmöglich zu so vielen Kenntnissen zu verhelfen, dass Sie – auch ohne das Buch zunächst komplett lesen und verstehen zu müssen – schon bald Ihre ersten kleinen Projekte realisieren können.

Das Fallbeispiel – der DVD-Hüllen-Generator »Covers«

Es ist schon eindrucksvoll, wie Zeitschriften, Zeitungen, Tankstellen, Buchclubs und andere Institutionen inzwischen um die Gunst ihrer Kunden werben. Auf diese Weise bin ich in den Genuss einer DVD-Sammlung gekommen, die sich sehen lassen kann und für die ich vergleichsweise wenig Geld investiert habe. So liegen beispielsweise fast allen großen Fernsehzeitschriften immer mal wieder DVDs einfach so als »Goody« bei – klasse Spielfilme, mehrsprachig, untertitelt und in überragender Qualität, quasi zum Nulltarif. Doch was verständlicherweise fehlt, ist jeweils ein ordentliches Cover, mit dessen Hilfe man diese DVD wie ein Buch in den Schrank stellen und in meterlangen Regalen – so die Idee – im Bedarfsfall auch wieder finden könnte. Das gilt umso mehr für die Cover von Filmen, die man beispielsweise von Pay-TV-Sendern aufgenommen oder sogar selbst gedreht hat. Klar – es gibt die unterschiedlichsten Cover-Designer für wenig Geld auf dem Markt; und jeder, der mit seinem Computer einen CD- oder DVD-Brenner erworben hat, wird ohnehin ein solches Programm sein Eigen nennen können, denn die meisten Brennprogramme, die den Brennern beiliegen, verfügen über teilweise recht leistungsfähige Designer.

Abbildung 5.1 Ein ausgedrucktes Inlay schneiden Sie einfach entlang der Markierungslinien aus und stecken es in das Plastikcover – hier ein Beispiel

Diese allerdings können in meinen Augen und für meine Belange viel zu viel. Wenn ich Zeit hätte, könnte ich damit bestimmt auch Cover gestalten, die vielleicht sogar fast genauso gut sind wie manches Original. Doch habe ich nie oder selten Zeit, und wenn doch, werde ich sicherlich meine Zeit nicht mit aufwändigem Coverdesign verbringen. Was ich brauche – was viele andere vielleicht auch haben wollen –, ist ein Programm, mit dem ich Titel, Schauspieler und Kurzbeschreibung eintippen, vielleicht noch ein Bild auswählen kann und das mir daraus dann ein Cover zaubert. Das geht schnell und reicht völlig.

Das »Pflichtenheft« von Covers

Ein solches Programm sollte folgendermaßen funktionieren:

- Das Programm sollte einen einfachen, aber dynamisch vergrößerbaren Dialog anbieten, in dem man die Grunddaten des Films in simplen Texteingabefeldern erfassen kann: Filmtitel, Schauspieler, Kurzbeschreibung und ein Bild, das auf der Frontseite des Covers abgedruckt werden soll. Abbildung 5.1 zeigt, wie so was in etwa im Ergebnis ausschauen kann; die Abbildungen 5.2 und 5.3 vermitteln Ihnen einen Eindruck von der Bedienung des Programms.

Abbildung 5.2 Dazu soll »Covers« in der Lage sein: Aus einer simplen Erfassung der Filmdaten druckt es buchstäblich auf Knopfdruck ...

Abbildung 5.3 ... das Inlay für das Filmcover

- Wenn Sie das Programmfenster von »Covers« vergrößern oder verkleinern, sollen sich alle Steuerelemente automatisch an die Größe des Formulars anpassen. Die Steuerelemente wachsen nach unten im Verhältnis zur Vergrößerung des Formulars mit; beim Verkleinern werden auch die Steuerelemente wieder entsprechend niedriger. Vergrößern Sie das Formular nach rechts, wandern die Schaltflächen auf der rechten Seite quasi mit dem rechten Fensterrand mit. Die anderen Steuerelemente, die für die Dateneingaben vorhanden sind, vergrößern sich dementsprechend.
- Sie können ein Bild in das Formular laden, das dann später automatisch auf die kompletten Ausmaße der Covervorderseite skaliert wird. Sollte das Bild nicht in das Bildelement auf dem Formular passen, erscheinen automatisch Bildlaufleisten, mit denen sich innerhalb des Bildes scrollen lässt.
- Mit der Schaltfläche *Eingaben löschen* setzen Sie alle Eingaben wieder zurück. Diese Schaltfläche ist praktisch, wenn Sie direkt hintereinander mehrere Cover drucken wollen.
- Das Programm soll sich alle Eingaben »merken«. Wenn Sie das Programm verlassen und neu starten, sollen sämtliche zuletzt vorgenommenen Eingaben oder Bilddefinitionen wieder so vorzufinden sein, wie sie vor dem letzten Beenden des Programms vorhanden waren.
- Wenn Sie Daten in das Formular eingetragen haben, gelangen Sie durch *Inlay drucken* zur Druckvorschau, die Ihnen das Inlay auf dem Bildschirm so darstellt, wie es später auch gedruckt werden soll. Auch dieses Vorschaufenster soll sich dynamisch vergrößern lassen.
- Vor dem Drucken sollen Sie die Möglichkeit haben, einen Drucker, den Sie verwenden wollen, auszuwählen und einzustellen. Der Drucker soll das Inlay aber immer unabhängig von den Einstellungen im Querformat drucken.

Die folgende Schritt-für-Schritt-Anleitung, die sich über mehrere Abschnitte erstreckt, zeigt, wie Sie diese Software von A–Z mit Visual Studio 2010 und Visual Basic 2010 erstellen. Sie werden erstaunt sein, wie wenig Code dazu nötig ist – das eigentliche Drucken macht dabei noch den größten Teil aus.

BEGLEITDATEIEN Natürlich ist das vollständige Projekt in den Begleitdateien zum Buch enthalten – Sie sollten aber in diesem Fall besser vollständig auf sie verzichten und kleinere Codepassagen wirklich abtippen, um ein Gefühl und besseres Verständnis für den Code-Editor zu erlangen. Die vergleichsweise umfangreiche Druckroutine wäre allerdings doch ein wenig zu viel des Guten, und Sie finden sie deswegen als reine Textdatei im Verzeichnis

```
...\VB 2010 Entwicklerbuch\Kapitel 05\Covers\Druckroutine für Covers.txt
```

Erstellen eines neuen Projekts

Und los geht's. Wir beginnen so einfach wie möglich: mit dem Erstellen eines neuen Projekts. Dazu starten Sie Visual Studio, falls Sie es noch nicht getan haben.

HINWEIS Um den Speicherort des Projekts, wie im Folgenden beschrieben, schon beim Anlegen festlegen zu können, rufen Sie den *Optionen*-Dialog über den Menübefehl *Extras/Optionen* auf. Stellen Sie dann sicher, dass das Kontrollkästchen *Neue Projekte beim Erstellen speichern* im Bereich *Projekte und Projektmappen/Allgemein* aktiviert ist.

1. Um ein neues Projekt anzulegen, klicken Sie auf der *Startseite* von Visual Studio im linken oberen Bereich auf *Neues Projekt*. Abbildung 5.4 hilft Ihnen bei der Orientierung.

Abbildung 5.4 Wenn Sie ein neues Projekt erstellen wollen, rufen Sie die entsprechende Funktion direkt aus der Startseite auf

2. Im anschließend erscheinenden Dialog wählen Sie in der Baumstruktur unter *Installierte Vorlagen* den Zweig *Visual Basic/Windows*. Im großen Listenfeld entscheiden Sie sich für *Windows Forms-Anwendung*. Geben Sie unter *Name* den Namen Ihres neuen Projekts ein – für unser Beispiel wählen Sie **Covers**. Unter *Ort* bestimmen Sie das Verzeichnis, in dem alle Projektdateien gespeichert werden sollen. *Durchsuchen* bringt Sie hierfür zu einem Dialog, der Ihnen bei der Bestimmung des Verzeichnisses helfen kann. Stellen Sie sicher, dass das Kontrollkästchen neben *Verzeichnis für Lösung erstellen* aktiviert ist. Abbildung 5.5 hilft Ihnen bei der Orientierung. Klicken Sie anschließend auf *OK*.

TIPP Wenn Sie mehrere Projekte in einer Projektmappe vereinen möchten – beispielsweise eine Anwendung in mehrere Schichten (Datenschicht, Geschäftslogik, Benutzeroberfläche, Steuerelemente etc.) aufteilen und diese Schichten in verschiedenen Klassenbibliotheken unterbringen –, können Sie Ihr Windows Forms-Anwendungsprojekt schon jetzt in einer Projektmappe integrieren. Die Projektmappe bekommt dann ein eigenes Verzeichnis; das darin enthaltene Windows Forms-Projekt liegt in einem Verzeichnis unterhalb des Projektmappen-Verzeichnisses. In diesem Fall aktivieren Sie das Kontrollkästchen *Verzeichnis für Lösung erstellen* und tragen Sie im Eingabefeld *Projektmappenname* den Namen der Projektmappe ein.

Abbildung 5.5 Mit diesem Dialog richten Sie eine neue Windows Forms-Anwendung ein

Gestalten von Formularen mit dem Windows Forms-Designer

Wir beginnen unser Fallbeispiel mit dem Entwurf des Formulars. Da wir die Toolbox zu diesem Zweck häufig gebrauchen werden, macht es Sinn, diese ständig im Vordergrund zu haben. An der linken Seite der Visual Studio-IDE finden Sie eine »eingefahrene« Registergruppe, bestehend aus dem Server-Explorer und der Toolbox.

Bewegen Sie den Mauszeiger auf das Toolbox-Symbol, worauf sich die Toolbox in den Vordergrund schiebt, und klicken Sie auf das Pin-Symbol im Fenstertitel (das Symbol in der rechten oberen Ecke der Toolbox; vergleiche Abbildung links). Dadurch bleibt die Toolbox, wo sie ist, und verschwindet nicht wieder automatisch im Hintergrund, wenn der Mauszeiger aus ihr herausbewegt wird.

Damit viel Platz für Experimente bleibt, vergrößern Sie das vorhandene Formular, sodass es einen Großteil der Arbeitsfläche einnimmt.

Positionieren von Steuerelementen

Wie in Abbildung 5.2 (Seite 124) zu sehen, verfügt das Formular über drei Schaltflächen, mit denen verschiedene Funktionen ausgelöst werden. Diese Schaltflächen werden wir als Erstes auf dem Formular positionieren.

1. Dazu klicken Sie in der Toolbox auf das *Button*-Symbol. Bewegen Sie den Mauszeiger in das Formular und ziehen Sie eine Schaltfläche ungefähr in der Größe auf, wie sie den Schaltflächen in Abbildung 5.2 entspricht.
2. Ziehen Sie zwei weitere Schaltflächen irgendwo im Formular auf – Größe und Position der Schaltflächen sind dabei zunächst völlig egal. Wir werden gleich die entsprechenden Funktionen kennen lernen, mit denen wir die Schaltflächen angleichen können.

TIPP Wenn Sie ein Steuerelement, wie zum Beispiel die Schaltfläche in der Toolbox, auswählen und dann auf die Formularoberfläche an eine gewünschte Position einfach klicken, wird das Steuerelement in der Windows-Standardgröße eingefügt.

3. Per Ziehen und Ablegen verschieben Sie nun die Schaltfläche mit der Aufschrift *Button1* in die rechte, obere Ecke des Formulars. Wenn Sie in der oberen, rechten Ecke ankommen, werden Sie merken, dass die Schaltfläche an einer bestimmten Ecke »anschnappt«, und Sie können zwei Hilfslinien erkennen (Abbildung 5.6), die die Schaltfläche auf Abstand halten.

Abbildung 5.6 Ausrichtungslinien helfen Ihnen bei der Positionierung von Steuerelementen am Formularrand oder untereinander

Ausrichtungslinien (Guidelines) und die Margin/Padding-Eigenschaften von Steuerelementen

Jedes Steuerelement, das Sie im Formular positionieren können, basiert auf einem Grundsteuerelement namens `Control`. Dieses Steuerelement stellt nicht nur einen Grundpool an Funktionalität zur Verfügung, auf denen andere Steuerelemente beispielsweise für das Anzeigen ihrer eigentlichen Inhalte (eine Schaltfläche muss andere Inhalte darstellen als ein `CheckBox`-Steuerelement) aufbauen können, sondern stellt auch eine grundsätzliche Designerfunktionalität zur Verfügung (der folgende graue Kasten liefert genauere Infos zu diesem Thema).

Das ist der Grund, wieso jedes Steuerelement, das Sie im Formular platzieren können, automatisch und ausnahmslos über eine `Margin`-Eigenschaft verfügt. Diese Eigenschaft bestimmt, wie groß der Abstand zwischen einem Steuerelement und einem anderen Steuerelement sein wird, wenn Sie bei dessen Platzierung die durch die Ausrichtungslinien vorgegebenen Abstände akzeptieren (oder mit anderen Worten: ab welchem Abstand die Ausrichtungslinien »anspringen« und das Steuerelement bei diesem Abstand »einschnappt«).

Wenn Steuerelemente als »Träger« anderer Steuerelemente fungieren – ein solches Steuerelement nennt man übrigens Container-Steuerelement bzw. `ContainerControl` –, kommt eine weitere Eigenschaft ins Spiel: die `Padding`-Eigenschaft. Diese Eigenschaft bestimmt, wie viel Abstand zusätzlich zum äußeren Rand eingehalten werden soll, wenn ein Steuerelement innerhalb eines beinhaltenden Steuerelements (also Container-Steuerelement) angeordnet wird.

HINWEIS Formulare selbst machen dabei offensichtlich noch eine Ausnahme: Beim Anordnen von Steuerelementen am Rand von Formularen werden zusätzlich ein paar Pixel Abstand eingehalten, wenn die Ausrichtungslinien dargestellt werden, möglicherweise um dem Microsoft Style-Guide für das Gestalten von Formularen zu entsprechen.

TIPP Das Einschnappen geschieht übrigens nicht nur an Stellen, die den durch die `Margin`- bzw. die `Padding`-Eigenschaften festgelegten Abständen entsprechen. Visual Studio stellt Ihnen das Einschnappen auch zur Verfügung, um Steuerelemente an Basislinien Ihrer Textzeilen exakt auszurichten, wenn diese nebeneinander stehen. Das ist häufig der Fall, wenn Sie ein `Label`-Steuerelement dafür verwenden, ein `TextBox`-Steuerelement im Formular zu beschriften, wie in Abbildung 5.7 zu sehen.

Abbildung 5.7 Gerade beim Beschriften von TextBox-Steuerelementen mit Label-Steuerelementen helfen Ihnen Ausrichtungslinien, da die Steuerelemente auch an den Textbasislinien einschnappen

TIPP Entwickler, die lieber mit der Rasterfunktionalität arbeiten, die sie aus Visual Studio 2003 bzw. Visual Basic 6.0 kennen, müssen darauf auch in Visual Studio 2010 nicht verzichten. Im *Optionen*-Dialog von Visual Studio, den Sie über das *Extras*-Menü erreichen, können Sie auf der Registerkarte *Windows Forms-Designer* die *LayoutMode*-Eigenschaft von *SnapLines* auf *SnapToGrid* ändern. Möchten Sie zwar im Layoutmodus von Visual Studio 2010 arbeiten, aber ohne die Einschnapp-Funktionalität auskommen, setzen Sie im gleichen Dialog die *SnapToGrid*-Eigenschaft auf *False*.

Wem »gehört« eigentlich der Formular-Designer?

Rein rechtlich gesehen natürlich Ihnen – schließlich haben Sie Visual Studio und damit auch den Formular-Designer erworben. Aber das meine ich gar nicht. Die eigentliche Frage lautet: In welchem Programmteil des gesamten Visual Studio/.NET Framework-Komplexes ist die Designer-Funktionalität eigentlich untergebracht? Die Antwort mag Sie überraschen, denn: Es ist nicht so, wie man vielleicht vermuten würde, dass Visual Studio komplett selbst für die Bearbeitung der Steuerelemente im Designer zuständig wäre. Visual Studio fungiert im Grunde nur als Ereignis-Weiterreich-Mechanismus an die betroffene Komponente. Ein Steuerelement besteht nämlich grundsätzlich aus zwei Teilen: dem Steuerelement, das die eigentliche Funktionalität zur Verfügung stellt, die Sie zur Laufzeit Ihres Programms benötigen, und einer weiteren Komponente, dem Designer. Ja genau. Jede Komponente verfügt streng genommen über ihren eigenen Designer. Aber natürlich haben nicht Hunderte von Entwicklern Hunderte verschiedene Designer programmiert. Genauso, wie `Control` alle Grundfunktionalitäten für ein neues, eigentliches Steuerelement bereitstellt, das diese lediglich weiter ausbaut, gibt es auch eine weitere Komponente namens `ControlDesigner`, die alle Design-Grundfunktionalitäten beinhaltet. Visual Studio spielt dabei nur die Rolle eines Gastgebers und stellt das Umfeld, damit ein solcher Designer Platz zum Austoben hat. Im Fachterminus lautet das: »Visual Studio ist der *Designer Host*«. Wird ein neues Steuerelement geschaffen und bringt es keinen eigenen Designer mit (komplett neu oder auf `ControlDesigner` basierend, spielt dabei keine Rolle), verwendet Visual Studio eben einfach `ControlDesigner` selbst für dieses Steuerelement – und dessen Basisfunktionen erlauben zumindest das Verschieben, Kopieren, »Einschnappen« oder andere Funktionen, die Sie beim Aufbau eines Formulars benötigen. Für diejenigen unter Ihnen, die sich ein wenig mit den .NET-Assemblys auskennen und die es interessiert: Die Designer aller Steuerelemente von .NET befinden sich in der Assembly *System.Design.Dll*.

Angleichen von Größe und Position von Steuerelementen

Nun befinden sich alle drei Steuerelemente wild platziert im Formular – Ziel ist es aber, dass sie alle drei nicht nur wie an einer Schnur ausgerichtet, bündig untereinander stehen, sondern dass sie auch alle drei die gleiche Größe haben. Es gäbe die Möglichkeit, diese Änderungen manuell vorzunehmen, indem Sie die Steuerelemente solange »zurechtzuppeln« und verschieben, bis sie passen. Dank der Ausrichtungslinienfunktionalität ist das kein großer Akt, aber immerhin ein Akt.

Es geht auch einfacher: Sie können mehrere Steuerelemente markieren und anschließend Größen- und Positionsoperationen durchführen, wobei sich dann alle Steuerelemente an dem zuerst markierten Steuerelement orientieren. Wenn Sie die Steuerelemente markiert haben (wie gesagt: das zuerst markierte ist immer das Referenzsteuerelement), bietet Ihnen Visual Studio bestimmte Funktionen an, die Sie im Abschnitt »Funktionen zum Layouten von Steuerelementen im Designer« ab Seite 155 beschrieben finden.

Selektieren mehrerer Steuerelemente und Bestimmen des Referenzsteuerelements

Um mehrere Steuerelemente im Formular gleichzeitig zu selektieren, klicken Sie das erste Steuerelement ganz normal an und dann nacheinander alle weiteren Steuerelemente bei gedrückt gehaltener ⇧-Taste. Oder Sie ziehen mit der Maus einen Rahmen um die Steuerelemente, die selektiert werden sollen. Das Referenzsteuerelement für bestimmte Funktionen, die Sie in Tabelle 5.4 ab Seite 155 beschrieben finden, ist dasjenige, welches als Erstes selektiert wurde, wenn Sie die Steuerelemente bei gedrückter ⇧-Taste auswählen. Möchten Sie das Referenzsteuerelement nach der Selektion der Steuerelemente ändern, klicken Sie es einfach an (dabei halten Sie die ⇧-Taste *nicht* mehr gedrückt). Anders als bei vielen anderen Selektionsverfahren unter Windows, wird dabei die Selektion nicht aufgehoben und das neu angeklickte Element nicht ausgewählt, sondern Sie ändern wirklich nur das Referenzsteuerelement. Möchten Sie die Selektion komplett aufheben, klicken Sie einfach in einen freien Bereich innerhalb des Formulars oder auf ein zuvor nicht selektiertes Steuerelement.

TIPP Das Referenzsteuerelement erkennen Sie übrigens daran, dass es mit weißen Anfasspunkten versehen ist, während alle anderen markierten Steuerelemente über schwarze Anfasspunkte verfügen.

1. Für unser Beispiel selektieren Sie auf diese Weise zunächst alle Schaltflächen im Formular. Wichtig dabei ist, dass die Schaltfläche in der linken, oberen Ecke zum Referenzsteuerelement wird.

HINWEIS Für den folgenden Schritt müssen Sie unter Umständen die benötigte Symbolleiste einschalten. Klicken Sie dazu mit der rechten Maustaste auf einen freien Bereich neben einer schon eingeblendeten Symbolleiste, und wählen Sie aus dem Kontextmenü, das jetzt erscheint, den Eintrag *Layout*.

2. Wählen Sie aus der Symbolleiste die Funktion *Größe angleichen* – die Tabelle im Abschnitt »Funktionen zum Layouten von Steuerelementen im Designer« ab Seite 155 hilft Ihnen, das richtige Symbol dafür zu finden.

Abbildung 5.8 Hier sehen Sie, dass sich alle Schaltflächen wild platziert im Formular befinden – mit Ausnahme der ersten, die ihre endgültige Position und Größe bereits hat ...

3. Klicken Sie anschließend auf das Symbol *Links ausrichten*, um die Schaltflächen linksbündig untereinander auszurichten. Damit stehen anschließend alle drei Schaltflächen genau ausgerichtet untereinander.
4. Die Abstände zwischen den Schaltflächen passen Sie dann entweder durch Verschieben der Schaltflächen mit der Maus per Ziehen und Ablegen und der Unterstützung durch die Ausrichtungslinien an, oder Sie verwenden die Funktion *Vertikalen Abstand angleichen* und so oft die Funktionen *Vertikalen Abstand vergrößern* bzw. *Vertikalen Abstand verkleinern*, bis Sie mit dem Ergebnis zufrieden sind.

Abbildung 5.9 ... und hier nach Anwenden der Funktionen Größe angleichen, Links ausrichten, Vertikalen Abstand angleichen sowie Vertikalen Abstand verkleinern

Im günstigsten Fall mussten Sie die Schaltflächen nach dem Aufziehen im Formular nicht einmal mehr »anpacken«. Innerhalb von Sekunden konnten Sie sie mithilfe der Funktionen der *Layout*-Symbolleiste so anpassen, dass sie sich nun mit sauberem Layout im Formular befinden.

Häufige Arbeiten an Steuerelementen mit Smarttags erledigen

Es gibt komplexe Steuerelemente, wie beispielsweise das `TableLayoutPanel` (das wir für den nächsten Schritt unseres Beispiels benötigen), die für das bequeme Einstellen bestimmter Verhaltensweisen so genannte Smarttags anbieten. Smarttags, wie in Abbildung 5.10 zu sehen, führen zu einer Liste mit Kontextaufgaben, die an das jeweilige Steuerelement gekoppelt sind und die Ihnen eine Art Abkürzung zu den Eigenschaften des Steuerelements bieten und gleichzeitig eine komfortablere Einstellung dieser Eigenschaften ermöglichen.

Abbildung 5.10 Ein Klick auf den Smarttag eines Steuerelements öffnet das Aufgaben-Fenster, das Funktionen zum Erledigen der häufigsten Aufgaben anbietet

1. Für unser Beispiel wählen Sie aus der Toolbox per Klick das `TableLayoutPanel` aus. Ziehen Sie dieses Steuerelement im Formular in etwa in der Größe auf, dass es Abbildung 5.10 entspricht.
2. Sofort öffnet sich die hinter dem Smarttag stehende Liste mit Kontextaufgaben. Sie können diese Kontextaufgabenliste jederzeit mit einem Klick auf den Smarttag öffnen und schließen.

HINWEIS Nicht jedes Steuerelement verfügt über einen Smarttag mit einer dahinter stehenden Liste mit Kontextaufgaben, aber wenn ein Steuerelement einen Smarttag besitzt, dann steht Ihnen diese Liste jederzeit zur Verfügung.

Dynamische Anordnung von Steuerelementen zur Laufzeit

Denken Sie mal einige Jahre zurück und dabei an den Aufwand, den es noch in Visual Basic 6.0 machte, eine Benutzeroberfläche wie den Windows-Explorer zur implementieren. Sie mussten alleine mehrere Mannstunden dafür opfern, den Code dafür zu entwickeln, die Steuerelemente dynamisch neu anzuordnen, falls der Benutzer des Programms zur Laufzeit das Fenster vergrößerte oder verkleinerte.

Seit den .NET-Versionen von Visual Basic ist das sehr viel einfacher geworden. Geschickt umgesetzt müssen Sie nicht eine einzige Zeile Code schreiben, um Steuerelemente in Abhängigkeit einer Formulargrößenänderung neu zu positionieren und auszurichten. Und ab Visual Basic 2008 ist dies noch ungleich bequemer geworden, denn es stellt neue Container-Steuerelemente bereit, die ausschließlich diesem Zweck dienen. Folgende Features stehen Ihnen in Visual Basic 2010 zur Verfügung, mit denen Sie diese Problematik – wie gesagt ohne Programmierung – in den Griff bekommen:

- **`Anchor`-Eigenschaft:** erlaubt das Verankern jeder Seite eines Steuerelements auf dem Formular. Vergrößern oder verkleinern Sie das Formular, dann »wandern« die Seiten des Steuerelements im gleichen Verhältnis hinter den Seiten des Formulars her, an denen es verankert ist.
- **`Dock`-Eigenschaft:** erlaubt das Andocken eines Steuerelements an einen Formularrand oder an ein bereits gedocktes Steuerelement. Dabei wird dafür gesorgt, dass das Steuerelement an den gedockten Seiten immer zum Formularrand oder zum ersten schon gedockten Steuerelement aufschließt.
- **`SplitContainer`-Steuerelement:** stellt einen Doppelcontainer für weitere Steuerelemente zur Verfügung. Dessen Besonderheit ist, dass sich die Größe eines Containerteils auf Kosten des anderen Containerteils vergrößern lässt. Denken Sie als Beispiel an den Windows-Explorer. In der Mitte können Sie einen vertikalen Trennbalken mit der Maus packen und nach links oder rechts verschieben – dementsprechend vergrößert (bzw. verkleinert) sich die linke `TreeView` mit den Verzeichnissen und anderen Hauptelementen (wie Systemsteuerung, Mobile Geräte etc.), während sich die Elementeliste im rechten Containerteil, die durch eine `ListView` realisiert wird, verkleinert (bzw. vergrößert).
- **`TableLayoutPanel`-Steuerelement:** stellt einen Container für weitere Steuerelemente zur Verfügung. Die Besonderheit ist, dass dieser Container mehrere Steuerelemente in einer Tabelle anordnet, deren Zeilen und Spalten sich in einstellbaren Verhältnissen vergrößern, wenn das `TableLayoutPanel`-Steuerelement selbst sich ebenfalls vergrößert oder verkleinert. Damit wird es möglich, dass nicht nur bestimmte Steuerelemente sich vergrößern, wenn sich das Formular vergrößert, sondern dass Steuerelemente im Falle einer Vergrößerung oder Verkleinerung des Formulars in bestimmten Verhältnissen anwachsen oder sich verkleinern.

- **`FlowLayoutPanel`-Steuerelement:** ist am besten mit einem Texteditor mit aktiviertem Wortumbruch zu erklären. Wenn ein Wort nicht mehr in eine Zeile passt, wird es automatisch in die nächste Zeile umbrochen. Beim `FlowLayoutPanel` ist dies genauso, jedoch nicht, was Worte betrifft, sondern Steuerelemente. Sie ordnen Steuerelemente nicht an festen Positionen, sondern nebeneinander bzw. untereinander an. Wenn sich das Formular vergrößert oder verkleinert, versucht das `FlowLayoutPanel`, möglichst viele Steuerelemente beispielsweise in eine Zeile zu bekommen. Passen sie nicht mehr in eine Zeile, da es zu eng wird im Formular, springen sie wie beim Wortumbruch des Editors automatisch in die nächste Zeile.

Die folgenden Abschnitte demonstrieren die wichtigsten dieser Features an unserem Fallbeispiel.

Verankern von Steuerelementen mit der Anchor-Eigenschaft

Die `Anchor`-Eigenschaft, die jedem Steuerelement anheim ist, löst das wichtigste, da häufigste aller Probleme auf schnelle und elegante Weise. Sie dient dazu, die Seiten von Steuerelementen mit den Seiten eines Formulars oder des sie beinhaltenden Container-Steuerelements zu verankern. Jede verankerte Seite des Steuerelements behält dabei den Abstand zum Rand seines Containers, wenn dieser sich vergrößert oder verkleinert, was zur Folge hat, dass das Steuerelement (wenn es an zwei gegenüberliegenden Seiten verankert ist) sich entsprechend vergrößert oder seine Position verändert (wenn es an nur jeweils einer Seite verankert ist).

Für unser Beispiel werden wir im Folgenden die drei Schaltflächen so neu verankern, dass sie hinter dem rechten Formularrand mitwandern, wenn der Benutzer das Formular in x-Richtung vergrößert oder verkleinert. Das `TableLayoutPanel`, das später die Eingabefelder, Beschriftungen und das Bild enthält, wird an allen vier Seiten verankert, damit es in allen Richtungen dynamisch mit dem Formular mitwächst und seinerseits wieder dafür sorgt, dass sich später alle in ihm enthaltenen Steuerelemente proportional anpassen.

1. Markieren Sie alle drei Schaltflächen.
2. Suchen Sie im Eigenschaftenfenster nach der `Anchor`-Eigenschaft. Öffnen Sie das Listenfeld, und klicken auf eines des vier länglichen Rechtecke am Rand (die ein Kreuz formen), um die Verankerungspositionen festzulegen. Aktive Verankerungen werden dabei grau dargestellt, ein Klick auf einen solchen grauen Bereich schaltet die jeweilige Verankerung wieder ab. Orientieren Sie sich dabei am besten an Abbildung 5.11.

TIPP Sie müssen wie hier im Beispiel nicht Eigenschaften verschiedener Steuerelemente, die Sie mit gleichen Werten belegen möchten, nacheinander setzen. Selektieren Sie stattdessen die Steuerelemente, für die der gleiche Wert einer Eigenschaft gelten soll, und weisen Sie mit dem Eigenschaftenfenster diese Eigenschaft allen selektierten Steuerelementen in einem Rutsch zu. Visual Studio findet bei mehreren selektierten Steuerelementen übrigens automatisch den größten gemeinsamen Nenner in Sachen vorhandene Eigenschaften, zeigt also bei mehreren selektierten Steuerelementen nur diejenigen Eigenschaften im Eigenschaftenfenster an, über die alle selektierten Steuerelemente verfügen.

Abbildung 5.11: Setzen Sie die Anchor-Eigenschaft von ursprünglich *Top, Left* auf *Top, Right* bewirkt das, dass die Steuerelemente immer den gleichen Abstand zum oberen und rechten Formularrand behalten und beim Vergrößern des Formulars nach rechts quasi hinter dem Rand herlaufen

3. Vergrößern Sie als Nächstes das `TableLayoutPanel` so, dass es mit den drei betroffenen Seiten möglichst nah an den Formularrändern liegt.
4. Setzen Sie die `Anchor`-Eigenschaft des `TableLayoutPanel` so, dass es an alle vier Seiten gebunden ist.

Abbildung 5.12 Setzen Sie die Anchor-Eigenschaft eines Steuerelements für alle vier Seiten, vergrößert sich das Steuerelement in alle Richtungen proportional mit dem Formular

Nachdem Sie alle Einstellungen vorgenommen haben, können Sie schon jetzt im Entwurfsmodus das Formular vergrößern und verkleinern, und Sie werden feststellen, dass sich bereits jetzt alle Steuerelemente an die jeweils neuen Formularausmaße anpassen.

Proportionales Anpassen von Steuerelementen an Formulargrößenänderungen mit dem TableLayoutPanel

Wenn Sie das `TableLayoutPanel` in Abbildung 5.12 betrachten, sehen Sie, dass es aus insgesamt vier Zellen besteht. Diese Zellen dienen, wie beispielsweise das `Panel`, als Container für jeweils ein weiteres Steuerelement. Das Besondere: Wenn sich das `TableLayoutPanel` an sich vergrößert bzw. verkleinert, vergrößern bzw. verkleinern sich die in ihm enthaltenen Zellen im Verhältnis. Bei geschickter Anordnung von Steuerelementen in den Zellen reichen Sie die verhältnismäßige Vergrößerung oder Verkleinerung an diese weiter. Ihre Formulare wachsen damit dynamisch, wenn mehr Platz auf dem Bildschirm zur Verfügung steht, und nutzen diesen damit wesentlich besser aus.

Einstellen der vorhandenen Spalten und Zeilen des TableLayoutPanel

Standardmäßig, also wenn Sie ein `TableLayoutPanel` das erste Mal im Formular aufziehen, besteht es aus vier Zellen: nämlich aus zwei Spalten und zwei Zeilen. Um die Anzahl an Spalten oder Zeilen zu erhöhen, verwenden Sie entweder – wie bei allen Steuerelementen – das Eigenschaftenfenster, oder Sie greifen auf die Kontextaufgabenliste zurück, die Sie über den Smarttag des Steuerelements erreichen.

1. Für unser Beispiel klicken Sie auf den Smarttag des sich bereits im Formular befindlichen `TableLayoutPanel` und klicken anschließend auf *Zeilen und Spalten bearbeiten*. Orientieren Sie sich dafür an Abbildung 5.13.
2. Öffnen Sie das Listenfeld *Anzeigen* und wählen Sie das Element *Zeilen* aus.

Abbildung 5.13 Wählen Sie aus der Kontextaufgabenliste des `TableLayoutPanel`, die Sie durch Klick auf seinen Smarttag öffnen, den Eintrag *Zeilen und Spalten bearbeiten* und unter *Anzeigen* das Element *Zeilen*, um die Anzahl und Eigenschaften der Zeilen einzustellen

3. Klicken Sie zweimal auf *Hinzufügen*, um dem `TableLayoutPanel` zwei zusätzliche Zeilen hinzuzufügen.

Abbildung 5.14 In diesem Dialog konfigurieren Sie die Größenverhältnisse der einzelnen Zellen

4. Klicken Sie auf die erste Zeile der Zeilenliste, und stellen Sie den *Größentyp* auf *Absolut*. Geben Sie **25** Pixel als Höhe der ersten Zeile ein. Orientieren Sie sich dabei an Abbildung 5.14. Sie bestimmen damit, dass sich diese Zeile nicht dynamisch vergrößert, sondern stets eine Höhe von 25 Pixeln aufweist. Da sich in dieser Zeile später das Eingabefeld für den Filmtitel befindet, der nur einzeilig eingegeben werden soll, braucht sich das Eingabefeld nicht in y-Richtung zu vergrößern, egal, wie groß der Abstand nach unten auch wird. Vielmehr macht es durch diese Einstellung sogar Sinn, den Abstand klein zu halten, damit anderen Zeilen, die Steuerelemente enthalten werden und die sich vergrößern sollen, mehr Platz für ihre Inhalte zur Verfügung steht.

WICHTIG Wenn Sie eine Wertänderung vorgenommen haben, drücken Sie NICHT [↵], sondern klicken Sie, falls Sie weitere Größentypen oder Werte ändern möchten, einfach auf die nächste Zeile in der Liste, deren Einstellungen Sie ändern möchten. Denn [↵] bewirkt das Auslösen der *OK*-Schaltfläche, und damit verschwindet der Dialog vom Bildschirm; Sie müssen den Dialog dann erneut aufrufen.

HINWEIS Wenn Sie einen Mix aus absoluten und prozentualen Größentypen verwenden, dann werden vom maximal zur Verfügung stehenden Platz (egal ob für Zeilen in der Höhe oder Spalten in der Breite) die absoluten Pixelangaben abgezogen. Der Rest des zur Verfügung stehenden Platzes wird zwischen den anderen Zellen so aufgeteilt, wie es den prozentualen Werteangaben entspricht.

5. Für *Row2* (Zeile 2) belassen Sie den *Größentyp* auf *Prozent*, geben als Wert allerdings **25**% ein.

6. Für *Row3* und *Row4* ändern Sie den *Größentyp* auf *Prozent* und geben als Wert **25**% und **50**% ein.

7. Wählen Sie anschließend aus dem Listenfeld *Anzeigen* das Element *Spalten*. Klicken Sie die Zeile *Column1* (Spalte 1) an, setzen Sie den *Größentyp* auf *Absolut* und geben Sie **120** als feste Pixelbreite ein.
8. An *Column2* (Spalte 2) nehmen Sie ebenfalls eine kleine Änderung vor, indem Sie die Prozentangabe für diese Spalte auf **100**% einstellen.

   Mit diesen Einstellungen haben Sie nun erreicht, dass die linke Spalte, die die immer gleich lautenden Beschriftungen enthalten wird, stets eine Breite von 120 Pixel behält, während sich die rechte Spalte dynamisch vergrößern oder verkleinern kann.
9. Sie können den Dialog nun mit ↵ bestätigen und damit schließen, da Sie ihn jetzt nicht mehr benötigen.
10. Klappen Sie die noch geöffnete Kontextaufgabenliste durch Klick auf den Smarttag des `TableLayoutPanel` auch wieder zu.

Das Ergebnis sollte anschließend in etwa wie in Abbildung 5.15 ausschauen.

Abbildung 5.15 So sollte das vorläufige Ergebnis ausschauen, nachdem Sie die Zeilen- und Spaltenparameter der Zellen bestimmt haben

Anordnen von Steuerelementen in den Zellen eines TableLayoutPanel

Ein Blick auf das Ergebnis in Abbildung 5.2 (Seite 124) offenbart, was als Nächstes auf uns zukommt: die Anordnung der `TextBox`- und `Label`-Steuerelemente für die Eingabe der Daten sowie deren Beschriftung. Zum Platzieren der Steuerelemente fügen Sie diese per Ziehen und Ablegen aus der Toolbox in die Zellen des `TableLayoutPanel` ein. Abbildung 5.16 hilft Ihnen bei diesem Vorgang.

Abbildung 5.16 Fügen Sie zum Platzieren der Steuerelemente diese einfach per Ziehen und Ablegen in die jeweiligen Zellen des TableLayoutPanel ein

1. Ziehen Sie drei `Label`-Elemente in die drei linken oberen Zellen.
2. Anschließend ziehen Sie drei `TextBox`-Elemente in die drei rechten oberen Zellen.
3. In die linke untere Zelle fügen Sie ein `Panel`-Steuerelement ein, das später als Träger für die Auslassungsschaltfläche (…) zur Wahl der Grafikdatei sowie für ein weiteres `Panel` und ein darin geschachteltes `PictureBox`-Steuerelement dient.

Anschließend sollte sich das Ergebnis in etwa wie in Abbildung 5.17 gestalten.

Abbildung 5.17 Nach dem Einfügen aller Steuerelemente sollten Sie in etwa dieses Ergebnis vorfinden

HINWEIS Wozu dient das `Panel` in der linken, unteren Ecke (wir benötigen dort schließlich ein Bild und eine Auslassungsschaltfläche für die Auswahl des Bildes)? Das verhält sich folgendermaßen: In jeder Zelle eines `TableLayoutPanel` darf sich nur ein Steuerelement befinden. Das bedeutet jedoch nicht, dass dieses eine Steuerelement nicht als Container für andere Steuerelemente fungieren kann. Auf diese Weise stehen Ihnen Tür und Tor offen für komplexe Gebilde, die jedoch, je komplexer und verschachtelter sie werden, einen entscheidenden Nachteil haben: Das Neuanordnen solcher Gebilde zur Laufzeit kostet Zeit – gerade bei langsamen Grafikkarten, weil natürlich die Inhalte aller Steuerelemente bei der kleinsten Größenänderung aktualisiert werden müssen. Sie sollten sich eigentlich mit maximal einem Container pro Zelle begnügen und die Anzahl der Steuerelemente in Containern pro Zelle wirklich so weit wie möglich begrenzen. Nur in Sonderfällen – die folgenden Abschnitte demonstrieren einen solchen – sollten Sie eine weitere Verschachtelungstiefe (im Sinne von »Container enthält Container enthält die eigentlichen Steuerelemente«) in Erwägung ziehen.

Verankern von Steuerelementen im TableLayoutPanel

Nun ist die Anordnung, wie Sie sie in Abbildung 5.17 sehen können, noch nicht besonders elegant. Die Beschriftungen »hängen« in der linken, oberen Ecke, und die `TextBox`-Elemente erstrecken sich nicht über die volle Breite des Formulars. Das zu ändern, hängt wieder buchstäblich an der `Anchor`-Eigenschaft der einzelnen Steuerelemente.

1. Setzen Sie die `Anchor`-Eigenschaft des oberen `Label`-Steuerelements auf *Left, Right*.
2. Setzen Sie die `Anchor`-Eigenschaften der verbleibenden `Label`-Steuerelemente auf *Top, Left, Right*.

3. Damit die `Label`-Elemente einen schöneren Eindruck machen, selektieren Sie alle, und …
4. … setzen Sie alle `BorderStyle`-Eigenschaften auf *Fixed3D*.
5. Setzen Sie alle `TextAlign`-Eigenschaften der `Label`-Steuerelemente auf *MiddleRight*.
6. Setzen Sie die `Anchor`-Eigenschaft des oberen `TextBox`-Steuerelements auf *Left, Right*.
7. Bevor wir die `Anchor`-Eigenschaften der zwei unteren `TextBox`-Steuerelemente ändern können, müssen wir diese für den mehrzeiligen Betrieb einstellen. Nur dann kann eine `TextBox` den gesamten zur Verfügung stehenden Bereich einer Zelle ausnutzen und an allen vier Seiten ohne Abstand verankert werden. Selektieren Sie dazu beide unteren `TextBox`-Steuerelemente.
8. Stellen Sie die `Multiline`-Eigenschaft der TextBox auf `True`. Damit diese TextBox automatisch eine Bildlaufleiste erhält, falls ein Benutzer mehr Zeilen eingibt, als sichtbarer Platz zur Verfügung steht, setzen Sie die `Scrollbars`-Eigenschaft auf `Vertical`.
9. Selektieren Sie nun zusätzlich das `Panel`, indem Sie die Taste `Strg` gedrückt halten und auf das Steuerelement klicken.
10. Stellen Sie jetzt die `Anchor`-Eigenschaft aller selektierten Steuerelemente auf *Top, Bottom, Left, Right*. Das Ergebnis sollte nun ausschauen wie in Abbildung 5.18.

Abbildung 5.18 Nach dem Verankern aller Steuerelemente sollte das Formular wie hier ausschauen

Verbinden von Zeilen oder Spalten des TableLayoutPanel

Das `TableLayoutPanel` fungiert neben seiner schon bekannten Funktionalität auch als so genannter *Property Extender*.

Was ist ein Property Extender?

Ein Steuerelement oder eine Komponente, die als Property Extender (etwa *Eigenschaftenerweiterer*) ausgelegt ist, ergänzt Steuerelemente, die im gleichen Container wie sie selbst liegen, oder Steuerelemente, die sie beinhaltet (wenn es sich bei dem Property Extender selbst um ein Container-Steuerelement handelt), um weitere Eigenschaften.

Eine solche Komponente schaut sich dabei an, welche Steuerelemente, die in ihrer Reichweite liegen, für die Erweiterung durch neue Eigenschaften, die im Kontext Sinn machen, infrage kommen, und stattet diese Steuerelemente mit neuen Eigenschaften aus. Diese Eigenschaften existieren aber dann nur für diese erreichbaren Steuerelemente, da sich diese neuen Eigenschaften grundsätzlich auf irgendeine Interaktion mit dem Property Extender-Steuerelement beziehen sollten.

Steuerelemente außerhalb der Reichweite des Property Extenders oder Steuerelemente in anderen Formularen sind davon nicht betroffen.

Aus diesem Grund haben alle Steuerelemente, die sich innerhalb einer Zelle eines `TableLayoutPanel` befinden, vier weitere Eigenschaften: `Column`, `ColumnSpan`, `Row` und `RowSpan`. Mit `Column` und `Row` wird festgelegt, in welcher Zeile und Spalte sich das Steuerelement befindet. Interessanter sind `ColumnSpan` und `RowSpan`: Diese bestimmen, über wie viele Zeilen und Spalten sich das Steuerelement erstrecken soll.

Das `Panel`, das später `PictureBox` und Auslassungsschaltfläche beinhalten soll, erstreckt sich bei genauerer Betrachtung (Abbildung 5.2 auf Seite 124) über die komplette Spaltenbreite. Aus diesem Grund sollte seine `ColumnSpan`-Eigenschaft auf 2 (die Anzahl an Spalten im `TableLayoutPanel`) gesetzt werden.

1. Heben Sie dazu als Erstes die aktuelle Selektion der anderen Steuerelemente auf, damit Sie nicht versehentlich die `ColumnSpan`-Eigenschaften der zuletzt selektierten Steuerelemente mit neu setzen (ist mir gerade passiert). Klicken Sie dazu auf einen freien Bereich im Formular.
2. Selektieren Sie das `Panel` und setzen Sie die `ColumnSpan`-Eigenschaft auf 2.
3. Für weitere Aufgaben, die in den folgenden Abschnitten besprochen werden, fügen Sie innerhalb des `Panel` eine Schaltfläche mit vergleichsweise kleinen Ausmaßen sowie ein weiteres `Panel` ein.
4. Platzieren Sie dabei die Schaltfläche in der rechten oberen Ecke des sie umschließenden `Panel`. Kontrollieren Sie die `Location`-Eigenschaft, die die linke obere Ecke des Steuerelements widerspiegelt, dahingehend, dass der y-Wert nicht negativ, sondern am besten 0 ist. Damit liegt die Schaltfläche ganz oben an.

Abbildung 5.19 Die untere Zeile des `TableLayoutPanel` enthält nun ein Panel, das sich über zwei Spalten erstreckt. Es beinhaltet ein weiteres Panel sowie eine Schaltfläche zur späteren Bilddateiauswahl.

5. Vergrößern Sie das zweite `Panel`, das Sie gerade eingefügt haben, dergestalt, dass es möglichst ohne Abstand an den äußeren Rändern des umgebenden `Panel` anliegt. Die Eigenschaften `Location` und `Size` können Ihnen beim Feintuning helfen.
6. Setzen Sie die `Anchor`-Eigenschaft der Schaltfläche auf *Top, Right*.
7. Setzen Sie die `Anchor`-Eigenschaft des zweiten `Panel` auf *Top, Bottom, Left, Right*.

Sie sollten anschließend ein Ergebnis etwa wie das in Abbildung 5.19 sehen.

Automatisches Scrollen von Steuerelementen in Containern

Vielleicht stellen Sie sich die Frage, wieso wir neben die Schaltfläche ein weiteres `Panel` und dort nicht direkt die `PictureBox` platziert haben. Die Antwort zeigt sich wie von selbst, wenn Sie nochmals einen Blick auf Abbildung 5.2 (Seite 124) werfen: Der Bildausschnitt bei Coverbildern, die größenmäßig nicht in die `PictureBox` passen, soll scrollbar sein. Leider stellt die `PictureBox` eine solche Funktionalität nicht zur Verfügung.

Abbildung 5.20 In diesem Beispiel befinden sich alle Steuerelemente in einem Panel, das als Container fungiert. Da nicht alle Steuerelemente in den sichtbaren Ausschnitt passen und die *AutoScroll*-Eigenschaft des `Panel` gesetzt ist, lässt sich der dargestellte Ausschnitt zur Entwurfs- und Laufzeit mit automatisch zur Verfügung gestellten Bildlaufleisten einstellen

Allerdings bieten Container-Steuerelemente eine Funktionalität, mit der wir sehr nah an das herankommen, was wir erreichen wollen. Wenn Sie die `AutoScroll`-Eigenschaft eines Container-Steuerelements auf `True` setzen, dann lassen sich die in ihm enthaltenen, aber durch einen zu kleinen sichtbaren Ausschnitt ausgeblendeten Steuerelemente mit den automatisch erzeugten Bildlaufleisten in das Sichtfenster des Containers »einrollen«. Abbildung 5.20 demonstriert dieses Verhalten recht anschaulich.

Dieses Verhalten nutzen wir nun aus: Die `PictureBox` erlaubt eine Einstellung, die das `PictureBox`-Steuerelement an die Größe des Bildes anpasst – diese Eigenschaft lautet `SizeMode` –, und sie muss zu diesem Zweck auf `AutoSize` gestellt werden. Befindet sich die `PictureBox` in einem `Panel`, dessen `AutoSize`-Eigenschaft gesetzt ist, und wird ein Bild geladen, das die `PictureBox` dazu zwingt, sich so weit zu vergrößern, dass sie nicht mehr in das `Panel` passt, bekommt das `Panel` ohne weiteres Zutun Bildlaufleisten. Wenn nun nur das `Panel` über einen Rahmen verfügt und die `PictureBox` keine weiteren sichtbaren Abgrenzungen aufweist, dann schaut es für den Benutzer so aus, als ließe sich das Bild in der `PictureBox` scrollen – in Wirklichkeit scrollt aber das `Panel` die ganze `PictureBox`. Ihr Vorteil: Sie haben nicht nur gerade mehrere Tage Entwicklungsarbeit gespart,[1] Sie brauchten sogar auch noch *nicht einmal eine einzige* Zeile Code dazu zu schreiben. Gehen wir es also an:

[1] Kein Scherz: Ein Steuerelement mit scrollbarem Inhalt zu entwickeln, ist wirklich keine triviale Sache.

Abbildung 5.21 Die geschachtelte Kombination aus einem *Panel*, einem weiteren Panel und einer *PictureBox* stellt später die Scrollfunktionalität zur Verfügung – so sollte das vorläufige Ergebnis ausschauen

1. Selektieren Sie das `Panel` neben der Auslassungsschaltfläche, sofern dieses noch nicht selektiert ist.
2. Setzen Sie dessen `BorderStyle`-Eigenschaft auf `FixedSingle`.
3. Setzen Sie dessen `AutoScroll`-Eigenschaft auf `True`.
4. Ziehen Sie eine `PictureBox` im `Panel` auf.
5. Damit es mit der Maus nicht zu filigran wird: Setzen Sie die `Location`-Eigenschaft der `PictureBox` »zu Fuß« auf `0;0`.

HINWEIS Denken Sie daran, dass sich Koordinaten eines Steuerelements immer auf dessen unmittelbaren Container beziehen.

6. Setzen Sie die `SizeMode`-Eigenschaft der `PictureBox` auf `AutoSize`.

Das Ergebnis sollte nun in etwa dem in Abbildung 5.21 entsprechen.

Selektieren von Steuerelementen, die Sie mit der Maus nicht erreichen

Bei solchen Verschachtelungen, wie Sie sie im vorherigen Abschnitt kennen gelernt haben, wird es ziemlich schwierig, bestimmte Steuerelemente mit der Maus zu selektieren – Sie treffen bisweilen einfach nicht mehr eine Begrenzungslinie des Steuerelements, das Sie eigentlich erreichen wollten, und wählen unbeabsichtigt ein ganz anderes aus.

In diesem Fall selektieren Sie einfach bewusst ein anderes Steuerelement. Drücken Sie dann so oft ⇆, bis Sie das Steuerelement erreicht haben, das Sie auch tatsächlich selektieren wollten. Und falls das auch nicht mehr hilft:

Selektieren von Steuerelementen mit dem Eigenschaftenfenster

Das Eigenschaftenfenster gibt Ihnen in der oberen Zeile Information über das derzeit selektierte Steuerelement. Bei dieser Infozeile handelt es sich allerdings um ein Listenfeld. Wählen Sie aus dieser Liste ein Steuerelement oder eine Komponente aus, wird diese im Formular-Designer selektiert. Hier erkennen Sie, wie wichtig es ist, den Steuerelementen sprechende und selbsterklärende Namen zu geben: Sonst wünschen wir Ihnen viel Spaß zum Beispiel bei der Suche nach der gewünschten `TextBox` in der Liste unter *TextBox1*, *TextBox2* etc.

Abbildung 5.22 Mithilfe des Eigenschaftenfensters können Sie jedes Steuerelement im Designer selektieren

Festlegen der Tabulatorreihenfolge (Aktivierreihenfolge) von Steuerelementen

Die Tabulatorreihenfolge – die Microsoft-Terminologie lautet »Aktivierreihenfolge« – bestimmt, welches Steuerelement jeweils als Nächstes aktiviert wird, wenn der Benutzer es vorzieht, mit der Tastatur zu arbeiten, und mit ⇆ zum jeweils nächsten Element springen will.

Abbildung 5.23 Nach dem Aufrufen von Ansicht/Aktivierreihenfolge können Sie die Reihenfolge festlegen, mit der Steuerelemente zur Laufzeit per (i) zu erreichen sind. Das Formular muss dazu selektiert sein.

Mit Visual Studio 2010 ist das ruck, zuck erledigt. Um beim Beispiel zu bleiben:

1. Klicken Sie auf die Titelzeile des Formulars, um es zu selektieren.
2. Wählen Sie aus dem Menü *Ansicht* den Befehl *Aktivierreihenfolge*. Das Formular ändert seine Darstellung, etwa wie in Abbildung 5.23 zu sehen.
3. Klicken Sie nacheinander die Steuerelemente in der Reihenfolge an, wie sie durch ⇆ zur Laufzeit des Programms aktiviert werden sollen. Beziehen Sie dabei auch die Container mit ein; etwas kniffelig wird es beim Bestimmen des ersten Steuerelements – dem `TableLayoutPanel` –, dessen Aktivierreihenfolgenordinalnummer, wie in Abbildung 5.23 zu sehen, etwas in den Hintergrund gerät (sie liegt hinter der Beschriftung *3,0*). Klicken Sie am besten auf den äußersten Rand der linken oberen Ecke, um es »zu erwischen«. Orientieren Sie sich dabei am Ergebnis, das Sie in Abbildung 5.24 sehen können.

TIPP Wenn Sie sich im Aktivierreihenfolgenmodus befinden und den Mauszeiger über ein Steuerelement bewegen, wird es zur besseren Orientierung mit einem dicken, gerasterten Rahmen gekennzeichnet. In Abbildung 5.24 sehen Sie das für *Panel1*.

Abbildung 5.24 Die Ordinalnummern aller Steuerelemente, die Sie bereits bestimmt haben, werden mit einem weißen Hintergrund eingefärbt. Drücken Sie (Esc), wenn Sie mit der Zuweisung fertig sind.

4. Nachdem Sie die letzte Schaltfläche angeklickt haben, drücken Sie Esc, um den Aktivierreihenfolgenmodus zu verlassen.

TIPP Die Aktivierreihenfolge wird durch die `TabIndex`-Eigenschaft eines Steuerelements festgelegt. Sie können die Aktivierreihenfolge daher auch ändern, indem Sie die `TabIndex`-Eigenschaften der Steuerelemente manuell anpassen. Es erfolgt allerdings keine automatische Neunummerierung. Wenn Sie zwischen dem Steuerelement mit dem TabIndex 3 und dem mit TabIndex 4 ein weiteres Steuerelement einfügen wollen, müssen Sie ab dem Steuerelement mit dem TabIndex 4 alle TabIndex-Einträge um 1 erhöhen. Hoffentlich hat Ihr Formular nicht 36 Steuerelemente.

Der eigentliche Aufbau des Formulars samt seiner kompletten Größenanpassungsfunktionslogik ist damit abgeschlossen. Rekapitulieren Sie: Für das, was Sie schätzungsweise in der letzten Stunde erreicht haben, hätten Sie noch in Visual Basic 6.0 sehr, sehr lange programmieren müssen. Zusammen mit einem `PictureBox`-Steuerelement, das eine Scrollfunktionalität für seinen Inhalt angeboten hätte, vielleicht sogar mehrere Tage.

Über die Eigenschaften Name, Text und Caption

Was für das Beispiel auf Designer-Seite jetzt noch fehlt, sind die Beschriftungen (für die Steuerelemente, die es betrifft) und die Namen der Steuerelemente, unter denen Sie sie beim Programmieren »erreichen« können. Diese Dinge werden mit den Eigenschaften `Name` (für den Namen eines Steuerelements, mit dem Sie es beim Programmieren ansprechen) und `Text` (für Beschriftungen) festgelegt.

TIPP Ausnahmslos jedes Steuerelement verfügt über eine `Text`-Eigenschaft, da diese auch in `Control` implementiert ist, auf das jedes Steuerelement aufbaut. Zwar gibt es Steuerelemente, die ihre `Text`-Eigenschaft nicht im Eigenschaftenfenster offen legen, dennoch ist die `Text`-Eigenschaft bei diesen vorhanden.

WICHTIG Die `Caption`-Eigenschaft, wie Sie sie von Visual Basic 6.0 beispielsweise für das Festlegen der Beschriftung von Schaltflächen und `Label`-Steuerelementen kennen, gibt es in .NET NICHT mehr. Auch hier übernimmt, wie es unter Visual Basic 6.0 beispielsweise auch schon immer beim `TextBox`-Steuerelement der Fall war, die `Text`-Eigenschaft diese Aufgabe.

Und noch ein …

TIPP Wenn Sie `Text`- und `Name`-Eigenschaften zuweisen, sollten Sie das nicht wechselweise, sondern nacheinander machen. Bestimmen Sie also zuerst alle `Name`-Eigenschaften und dann alle `Text`-Eigenschaften. Der Grund: Wenn Sie das erste Steuerelement angeklickt und dann beispielsweise die `Text`-Eigenschaft im Eigenschaftenfenster festgelegt haben, brauchen Sie anschließend einfach nur das nächste Steuerelement anzuklicken und einfach drauflos zu tippen. Visual Studio merkt sich, dass Sie beim letzten Gebrauch des Eigenschaftenfensters die `Text`-Eigenschaft gesetzt hatten, und leitet in dem Moment, in dem Sie nach der Selektion eines neuen Steuerelements zu tippen begonnen haben, die Tastatureingaben automatisch an die zuletzt verwendete Eigenschaft im Eigenschaftenfenster weiter.

Schnellzugriffstasten durch die Text-Eigenschaft bestimmen

Die `Text`-Eigenschaft übernimmt bei vielen Steuerelementen übrigens noch eine weitere Funktion: Buchstaben, vor die Sie das &-Zeichen stellen, werden als Schnellzugriffstasten markiert. Sie erkennen die Schnellzugriffstasten eines Steuerelements daran, dass diese unterstrichen dargestellt werden. Zur Laufzeit befähigen Sie den Benutzer Ihres Programms dann, dieses Steuerelement anzusteuern oder – bei Schaltflächen beispielsweise – auszulösen, indem er die Schnellzugriffstaste in Verbindung mit `Alt` betätigt. Wenn Sie zum Beispiel eine `TextBox` über diese Schnellzugriffstaste anspringen wollen, sollten Sie ein `Label`-Steuerelement in der Aktivierreihenfolge VOR die TextBox stellen und dort die Schnellzugriffstaste durch das &-Zeichen festlegen. Da `Label` (also reine Beschriftungen) unter Windows standardmäßig nicht angesprungen werden, wird die nachfolgende `TextBox` angesteuert. Unten im Beispiel sehen Sie diese Vorgehensweise.

Auf diese Weise bestimmen Sie zunächst die `Text`-Eigenschaften aller `Label`- und `Button`- (Schaltflächen-) Steuerelemente. Orientieren Sie sich dabei am besten an Abbildung 5.25 und an Tabelle 5.1.

Abbildung 5.25 Nehmen Sie bei der Zuweisung von Beschriftungen und Schnellzugriffstasten an die Steuerelemente diese Abbildung zu Hilfe

Steuerelement	Name (Name-Eigenschaft)	Beschriftung (Text-Eigenschaft)
Label (links, oben)	lblNameDesFilms	"&Name des Films"
Label (links, Mitte)	lblSchauspieler	"&Schauspieler"
Label (links, unten)	lblKurzbeschreibung	"&Kurzbeschreibung"
Button (rechts, oben)	btnOK	"&OK"
Button (rechts, Mitte)	btnInlayDrucken	"Inlay &drucken"
Button (rechts, unten)	btnEingabenLöschen	"Eingaben &löschen"
TextBox (Mitte, oben)	txtNameDesFilms	"" ((Leerstring – Löschen Sie die Eingabe))
TextBox (Mitte, Mitte)	txtSchauspieler	""
TextBox (Mitte, unten)	txtKurzbeschreibung	""
Schaltfläche (neben der PictureBox)	btnCoverbildWählen	"..."
PictureBox	picCoverbild	((nicht anwendbar))

Tabelle 5.1 Verwenden Sie diese *Name*- und *Text*-Eigenschaften für die Steuerelemente im Formular

Sie erkennen, dass ich bei der Benennung von Steuerelementen nach einem bestimmten Schema vorgehe, indem ich im Namen mit einer Drei-Buchstaben-Abkürzung kenntlich mache, um was für ein Steuerelement es sich handelt. Ob Sie für eigene Namensgebungen von Steuerelementen diese Konvention adaptieren oder nicht, bleibt Ihnen natürlich überlassen. Viele Entwickler machen das, einige nicht – Microsoft sagt, man soll es *nicht* tun. Sollten Sie sich dafür entscheiden – im Abschnitt »Namensgebungskonventionen für Steuerelemente« ab Seite 154 finden Sie eine Tabelle, die die Konventionen für die wichtigsten Steuerelemente auflistet.

HINWEIS Auch das Formular selbst verfügt über eine `Text`-Eigenschaft. In diesem Fall bestimmt sie den Text, der in der Titelzeile des Formulars erscheinen soll.

1. Selektieren Sie das Formular.
2. Bestimmen Sie im Eigenschaftenfenster als `Text`-Eigenschaft einen Text, der Ihnen geeignet scheint – beispielsweise »Der VB-Entwicklerbuch-Hüllen-Inlay-Generator« oder Ähnliches.

Einrichten von Bestätigungs- und Abbrechen-Funktionalitäten für Schaltflächen in Formularen

Wie Sie Schnellzugriffstasten einrichten, haben Sie im letzten Abschnitt bereits erfahren. Es gibt zwei Tasten bei der Formularbedienung, denen eine besondere Funktionalität zukommt und die Sie mit diesem Verfahren allerdings nicht definieren können: [↵] (in der Regel für die *OK*-Schaltfläche), um einen Dialog zu bestätigen, und [Esc], um einen Dialog wieder zu verlassen, ohne dass die Eingaben übernommen werden.

Visual Basic 6.0-Entwickler werden bei den Schaltflächeneigenschaften vergeblich nach diesen Einstellungsmöglichkeiten suchen – sie wurden in .NET nämlich in das Formular verlagert. Das macht letzten Endes auch mehr Sinn, schließlich kann nur jeweils eine Schaltfläche für das Abbrechen und eine weitere für das Bestätigen eines Dialogs infrage kommen.

Diese Aufgaben übernehmen nun zwei Formulareigenschaften namens `AcceptButton` und `CancelButton`. Beim Setzen dieser Eigenschaften öffnen Sie ein Listenfeld, das alle Schaltflächenelemente des Formulars enthält, die für die jeweilige Aufgabe infrage kommen.

Für unser Beispiel weichen wir ein wenig vom Standard ab. Wir legen die Abbrechen-Funktionalität auf die *OK*-Schaltfläche und die Bestätigen-Funktionalität auf die *Inlay drucken*-Schaltfläche. Das macht mehr Sinn für den Benutzer, denn er verlässt ja schließlich auch mit *OK* den Dialog. Ein versehentliches Betätigen von [↵] führt damit nicht zum Dialogende (und damit zum Beenden des Programms).

1. Selektieren Sie das Formular im Designer.
2. Setzen Sie die `AcceptButton`-Eigenschaft auf die Schaltfläche `btnInlayDrucken`.
3. Setzen Sie die `CancelButton`-Eigenschaft auf die Schaltfläche `btnOK`.

Hinzufügen neuer Formulare zu einem Projekt

Eine Smart Client-Anwendung besteht in den wenigsten Fällen aus nur einem Formular. In unserem Fall ist das genauso wenig der Fall. Wir benötigen ein weiteres Formular, das später die Druckvorschau enthält und mit dem man den Druckvorgang auslösen kann. Diesem Formular werden dann später zur Laufzeit die zu druckenden Daten übergeben – die Funktion des Druckens und der Vorschaudarstellung soll es dann völlig autonom übernehmen.

HINWEIS Trennen Sie sich schon beim Designen der Formulare Ihrer Anwendung von der prozeduralen Denkweise, falls Sie es noch nicht getan haben. Es zeugt von einem schlechten Programmierstil, wenn Sie quasi von außen, also beispielsweise aus einem anderen Formular, einem anderen Modul oder einer anderen Klasse auf die Elemente eines Formulars zugreifen. Stellen Sie besser nur wenige öffentliche Methoden im Formular bereit, die ausschließlich dazu da sind, Parameter entgegenzunehmen, und überlassen Sie dem Formular selbst die Auswertung dieser Parameter. Sie sollten – auch in Vorbereitung auf objektorientierte Programmierung – jedes Formular als eine kleine, in sich geschlossene Anwendung betrachten, der Sie lediglich Daten übergeben können und Resultate von diesem wiederbekommen.

WICHTIG Was im Formular passiert (das Setzen oder Abfragen von Eingabefeldern, das Auswerten von Benutzeraktionen) sollte einzig und allein der Formularklasse vorbehalten sein! Vermeiden Sie es, Inhalte von Benutzersteuerelementen von anderen Modulen, Klassen oder Formularen aus direkt zu manipulieren!

Um dem Projekt ein weiteres Formular hinzuzufügen, verfahren Sie wie folgt:

1. Öffnen Sie im Projektmappen-Explorer das Kontextmenü des Projekts (nicht der Projektmappe!) mit der rechten Maustaste. Falls Sie keine Projektmappe für Ihr Projekt sehen, öffnen Sie den Dialog *Extras/Optionen*, und wählen Sie im Bereich *Projekte und Projektmappen/Allgemein* die Option *Projektmappe immer anzeigen*.

   Wählen Sie *Hinzufügen* und *Windows Form*.

Abbildung 5.26 Mithilfe des Eigenschaftenfensters können Sie jedes Steuerelement im Designer selektieren

2. Im Dialog, der jetzt erscheint, belassen Sie den vorgegebenen Namen zunächst mit *Form2.vb* so, wie er ist.
3. Klicken Sie auf *Hinzufügen*.

4. Vergrößern Sie das neue Formular, das jetzt sofort angezeigt wird, dergestalt, dass es genug Platz für weitere Steuerelemente bereitstellt. Orientieren Sie sich am besten dazu an Abbildung 5.27.
5. Suchen Sie in der Toolbox in der Sektion *Drucken* (die Sie im Bedarfsfall öffnen müssen) nach dem `PrintPreviewControl`. Ziehen Sie es in ausreichender Größe im Formular auf. Stellen Sie dann die einzelnen Seiten des Steuerelements mithilfe der Ausrichtungslinien so ein, dass sie alle den gleichen Abstand zum oberen, linken und rechten Formularrand haben.
6. Setzen Sie die `Anchor`-Eigenschaft dieses Steuerelements auf *Top, Bottom, Left, Right*.
7. Fügen Sie zwei Schaltflächen in das Formular nebeneinander ein. Setzen Sie die `Anchor`-Eigenschaften beider Schaltflächen auf *Bottom, Right*.
8. Weisen Sie `Name`- und `Text`-Eigenschaften der Steuerelemente mithilfe von Tabelle 5.2 zu.

Steuerelement	Name (Name-Eigenschaft)	Beschriftung (Text-Eigenschaft)
PrintPreviewControl	((Behalten Sie den vorgegebenen Namen bei.))	–
Linker Button	btnOK	&OK
Rechter Button	btnDrucken	&Drucken
Formular	((Behalten Sie den vorgegebenen Namen bei.))	Covervorschau zeigen und Cover drucken

Tabelle 5.2 *Name*- und *Text*-Eigenschaften für die Steuerelemente im zweiten (Druck-)Formular

9. Speichern Sie das komplette Projekt, indem Sie entweder aus dem Menü *Datei* den Menüeintrag *Alle Speichern* oder das entsprechende Symbol aus der Symbolleiste verwenden.

Abbildung 5.27 So soll das Druckformular im Designer endgültig aussehen

Wie geht's weiter?

Der Design-Teil unseres kleinen Softwareprojekts ist abgeschlossen – jetzt geht es darum, die entsprechenden Steuerelemente mit Programmlogik zu füttern. Die beiden folgenden Abschnitte sollen Ihnen als Referenzen zum Nachschlagen dienen – verlieren Sie ruhig einen Blick darauf, und wenn Sie dann bereit sind, mit dem Programmieren zu beginnen und die Software fertig zu stellen, fahren Sie mit dem Abschnitt »Der Code-Editor« ab Seite 157 fort.

Namensgebungskonventionen für Steuerelemente in diesem Buch

Microsoft erklärt ausdrücklich, dass die Namensgebung von Objektvariablen keinerlei Hinweise darauf enthalten sollte, welchen Typ sie darstellen. Diverse Stil- und Sicherheitsüberprüfungswerkzeuge für Quellcode (beispielsweise FxCop, das bei den »größeren« Versionen von Visual Studio in den Projekteigenschaften implementiert ist) würden das bei der Quellcodeanalyse sogar anmahnen.

Eine Funktion von IntelliSense, so das Argument, reicht aus, um den Typ einer Objektvariablen eindeutig zu ermitteln – Sie brauchen lediglich den Mauszeiger auf die entsprechende Objektvariable zu positionieren, um den Typ der Variablen als ToolTip anzeigen zu lassen.

Für die Beispiele in diesem Buch habe ich mich dennoch dazu entschlossen, zumindest bei der Benennung von Objektvariablen, die Steuerelemente referenzieren, auf die Microsoft-Empfehlung zu verzichten. Ich finde, dass dies das Lesen von Listings auf Papier erleichtert und zum besseren Verständnis beiträgt. IntelliSense steht Ihnen nämlich nur im Code-Editor von Visual Studio zur Verfügung – aber wer weiß: Vielleicht arbeitet Microsoft schon an einer Papierversion von IntelliSense?

Komponente	Präfix-/Namenkombination
Label	lblName
Button	btnName oder cmdName[2]
TextBox	txtName
CheckBox	chkName
RadioButton	optName[3] oder rbtName
GroupBox	grpName
PictureBox	picName
Panel	pnlName
ListBox	lstName
ComboBox	cmbName
ListView	lvwName
TreeView	tvwName

Tabelle 5.3 Ein Vorschlag für Namenskonventionen bei der Namensgebung von Steuerelementen

[2] Von `CommandButton` – so hieß eine Schaltfläche offiziell unter Visual Basic 6.0. Unter .NET ist dies aber nicht mehr üblich; man sieht es nur noch hier und da bei Visual Basic 6.0-Portierungen.

[3] Von `OptionButton`, der Name der Optionsschaltfläche unter Visual Basic 6.0.

Funktionen zum Layouten von Steuerelementen im Designer

Tabelle 5.4 stellt die Funktionen gegenüber, die sich hinter den Symbolen der Symbolleiste *Layout* befinden. Abbildung 5.28 zeigt die Ausgangspositionierung der drei Steuerelemente im Formular. In der Tabelle finden Sie in der rechten Spalte das Ergebnis der Anordnung, nachdem Sie auf das entsprechende Symbol geklickt haben.

Abbildung 5.28 Die Ausgangsanordnung der Schaltflächen für die folgende Tabelle. Achten Sie darauf, dass sich bestimmte Funktionen an der oberen rechten Schaltfläche orientieren

Symbol	Funktion	Ergebnis nach Anwendung auf Steuerelemente in Abbildung
	Links ausrichten	
	Zentriert	
	Rechts ausrichten	
	Oben ausrichten	

Symbol	Funktion	Ergebnis nach Anwendung auf Steuerelemente in Abbildung
	Mittig ausrichten	
	Unten ausrichten	
	Breite angleichen	
	Höhe angleichen	
	Größe angleichen	
	Horizontalen Abstand angleichen	
	Horizontalen Abstand vergrößern	- n/a -
	Horizontalen Abstand verkleinern	- n/a -
	Horizontalen Abstand entfernen	- n/a -
	Vertikalen Abstand angleichen	
	Vertikalen Abstand vergrößern	- n/a -
	Vertikalen Abstand verkleinern	- n/a - ▶

Symbol	Funktion	Ergebnis nach Anwendung auf Steuerelemente in Abbildung
	Vertikalen Abstand entfernen	- n/a -
	Horizontal im Formular zentrieren	- n/a -
	Vertikal im Formular zentrieren	- n/a -
	In den Vordergrund	- n/a -
	In den Hintergrund	- n/a -

Tabelle 5.4 Symbole, mit denen Sie erweiterte Funktionalitäten des Ausgabefensters steuern

Tastaturkürzel für die Platzierung von Steuerelementen

Taste	Aufgabe
Pfeiltasten	Verschiebt das ausgewählte Steuerelement in die entsprechende Richtung
Strg + Pfeiltasten	Verschiebt das ausgewählte Steuerelement in die entsprechende Richtung, wobei das Steuerelement automatisch an der jeweils nächsten Ausrichtungslinie zum Stehen kommt
⇧ + Pfeiltasten	Vergrößert oder verkleinert das ausgewählte Steuerelement pixelweise
Strg + ⇧ + Pfeiltasten	Vergrößert oder verkleinert das Steuerelement so, dass es mit der entsprechenden Seite an der jeweils nächsten Ausrichtungslinie anliegt
Alt	Schaltet die Ausrichtungslinienfunktion so lange aus, wie die Taste beim Verschieben von Steuerelementen mithilfe der Maus gedrückt bleibt

Tabelle 5.5 Tastaturkürzel für die Platzierung von Steuerelementen

Der Code-Editor

Der Code-Editor ist mit Sicherheit das Element in Visual Studio, mit dem Sie die mit Abstand meiste Zeit verbringen werden. Dabei ist er über die Jahre zu einem wirklich genialen Werkzeug avanciert, das Ihnen so viel Arbeit abnehmen kann, dass das Arbeiten auch nach Jahren noch Spaß macht und flüssig von der Hand geht.

Das ist aber erst dann der Fall, wenn Sie alle seine Feinheiten und auch die eine oder andere Tücke kennen und zu beherrschen wissen. Und dabei geht es schon los mit der Wahl der richtigen Schriftart, die Ihnen ein ermüdungsfreies Arbeiten ermöglicht:

Die Darstellung im Editor auf die richtige Größe einstellen

In Visual Studio 2010 können Sie die Inhalte der Dokumente beinahe nach Belieben vergrößern und verkleinern – besondere Vorkehrungen zur Auswahl der richtigen Schrift, wie es noch in Visual Basic 2010 der Fall war, brauchen Sie dazu gar nicht zu treffen.

Am einfachsten ist das, wenn Sie mit der Maus arbeiten: Halten Sie die Taste Strg gedrückt, und bewegen Sie das Mausrad, wenn Sie sich im Editor befinden – dann vergrößert und verkleinert sich die Schrift mit der Drehung des Rades.

Abbildung 5.29 Der Text im Editor lässt sich in Visual Studio 2010 (und in den meisten anderen Dokumentenfenstern) beliebig vergrößern und verkleinern

Alternativ können Sie auch direkt einen Vergrößerungsfaktor in das entsprechende Textfeld (siehe Pfeil in der Grafik) eingeben, oder die ComboBox öffnen und den Zoom-Faktor auswählen.

Mit der seit Visual Studio 2010 standardmäßig eingestellten Schrift arbeitet es sich recht angenehm – finden jedenfalls die Autoren. Falls Sie eine generell andere Schrift zum Editieren verwenden wollen, wählen Sie aus dem Menü *Extras* den Befehl *Optionen*, und wechseln Sie zu *Schriftarten und Farben*. Im den entsprechenden Auswahlfeldern können Sie nun die Schriftgröße und Farbe bestimmen, mit der Sie standardmäßig arbeiten möchten.

HINWEIS Die eingestellte Größe für ein Editor-Fenster gilt nur solange, wie das Editorfenster geöffnet ist. Beim Schließen und nächsten Öffnen wird wieder die Ausgangsschriftgröße verwendet.

Viele Wege führen zum Code-Editor

Es gibt die verschiedensten Möglichkeiten, den Code-Editor ins Leben zu rufen. Der einfachste Weg, Module oder reine Klassendateien zu bearbeiten, läuft natürlich über den Projektmappen-Explorer: Ein Doppelklick auf eine Klassen- oder eine Moduldatei öffnet die Datei im Editor direkt. Um den Klassencode eines Formulars einzusehen, müssen Sie das Formular im Projektmappen-Explorer selektieren und anschließend auf das Symbol *Code anzeigen* klicken, das Sie in der oberen Zeile des Projektmappen-Explorers finden (der ToolTip hilft Ihnen, das richtige Symbol zu erhaschen). Folgende Möglichkeiten gibt es, den Code-Editor ins Leben zu rufen:

- Doppelklick auf eine reine Klassen- oder Moduldatei im Projektmappen-Explorer
- Bei ausgewählter Formularklasse ein Klick auf das Symbol *Code anzeigen* im Projektmappen-Explorer
- Bei einem Compiler-Fehler: Doppelklick auf die entsprechende Fehlermeldung im Ausgabefenster
- Bei einem Compiler-Fehler: Doppelklick auf die entsprechende Fehlermeldung in der Fehlerliste
- Bei Kommentaren in der Aufgabenliste: Doppelklick auf den entsprechenden Kommentar
- Doppelklick auf ein Element in einem Designer. Ein Doppelklick auf eine Schaltfläche in einem Formular bringt Sie beispielsweise zur bereits vorhandenen Ereignisbehandlung für diese Schaltfläche oder öffnet den Editor für das Formular und fügt den Coderumpf für die Ereignisbehandlungsroutine ein.
- Nach dem Auftreten eines Fehlers während der Ausführung einer Anwendung im Debug-Modus

IntelliSense – Ihr stärkstes Zugpferd im Coding-Stall

Das Konzept von IntelliSense erspart Ihnen beim Entwickeln die meiste Zeit. Warum? IntelliSense liefert Ihnen alle denkbaren Informationen über Objekte und Sprachelemente, die Sie gerade in Bearbeitung haben.

Zur Demonstration implementieren wir nun die ersten Codezeilen unseres Beispiels.

1. Öffnen Sie *Form1* im Designer, indem Sie doppelt auf die Formulardatei im Projektmappen-Explorer klicken.
2. Klicken Sie anschließend doppelt auf die Schaltfläche *OK*. Visual Studio bringt Sie daraufhin zum Code-Editor für den Klassencode von `Form1` und stellt automatisch den Funktionsrumpf für die Ereignisbehandlung der *OK*-Schaltfläche zur Verfügung. Das ist die Methode, die aufgerufen und ausgeführt wird, wenn der Benutzer zur Laufzeit auf die *OK*-Schaltfläche klickt.
3. In die Zeile zwischen `Private Sub…` und `End Sub` geben Sie nun **me.** ein. Sobald Sie den Punkt getippt haben, öffnet sich eine Liste mit allen Elementen, die für das Objekt anwendbar sind (in diesem Fall ist `me` das Formular selbst, da wir uns in der Klassendatei *Form1* befinden; die genaue Bedeutung lernen Sie in den Klassenkapiteln noch kennen – wir wollen uns an dieser Stelle auf die Editorfähigkeiten konzentrieren). Diese Liste nennt sich »Vervollständigungsliste«.
4. Um das Formular bei einem Klick auf *OK* zu schließen, wollen wir die `Close`-Methode verwenden. Geben Sie nun die ersten Buchstaben von `Close` weiter ein, springt der Auswahlbalken irgendwann auf die gesuchte Methode (Abbildung 5.30).

HINWEIS Sie werden feststellen, dass IntelliSense Ihnen nicht nur eine vollständige Elementliste für das Objekt liefert, sondern auch eine Kurzbeschreibung des jeweils ausgewählten Elements als ToolTip – ebenfalls in Abbildung 5.30 zu erkennen.

5. Sie brauchen den Methodennamen nun gar nicht weiter einzugeben. Sobald der richtige Methodenname markiert ist, drücken Sie [↵] – der Code-Editor fügt die restlichen Buchstaben des Methodennamens dann automatisch ein und stellt die Einfügemarke in die nächste Zeile.

TIPP Möchten Sie hinter dem Methodennamen keine neue Zeile beginnen, drücken Sie [Strg]+[↵]. Damit vervollständigen Sie lediglich den Methodennamen. Oder [Leerzeichen], dann erscheint die Methode gefolgt von einem Leerschritt.

Die erste Funktionalität des Programms haben Sie damit schon implementiert: Starten Sie das Programm testweise mit [F5], und probieren Sie die *OK*-Schaltfläche aus!

Abbildung 5.30 Der Punkt nach einem Objektnamen öffnet dank IntelliSense die Vervollständigungsliste, die eine Übersicht liefert, welche Elemente für das Objekt zur Anwendung kommen können

Filtern von Elementen in der Vervollständigungsliste

Wie Sie in Abbildung 5.30 erkennen können, verfügt die von IntelliSense angezeigte Elementliste über zwei Registerzungen, mit denen Sie die Elemente nach Wichtigkeit filtern können. Welche Elemente dabei »wichtig« sind, bestimmt Microsoft – schweigt sich aber über das Selektionsverfahren aus.

Zitat der Onlinehilfe: »Auf der standardmäßig aktivierten Registerkarte *Allgemein* werden Elemente angezeigt, die am häufigsten zum Vervollständigen der geschriebenen Anweisung verwendet werden. Auf der Registerkarte *Alle* sind alle für die automatische Vervollständigung verfügbaren Elemente aufgeführt, einschließlich der Elemente auf der Registerkarte *Allgemein*.«

Anzeigen der Parameterinfo von Elementen

1. Für das nächste IntelliSense-Feature werden wir die Funktion »Eingaben löschen« implementieren.
2. Wechseln Sie mit [Strg]+[⇆] erneut in die Designerdarstellung des Formulars *Form1*.
3. Klicken Sie doppelt auf die Schaltfläche *Eingaben löschen*, um den Funktionsrumpf für die Ereignisbehandlungsroutine dieser Schaltfläche einzufügen.
4. Beginnen Sie einzugeben:

```
Dim locDr As DialogResult
```

Sie werden feststellen, dass Ihnen IntelliSense nach dem Schreiben des Schlüsselworts `As` auch wieder die Vervollständigungsliste anbietet. Tippen Sie so viel vom Wort `DialogResult`, bis diese Enum in der Liste erscheint und markiert ist. Drücken Sie anschließend ↵.

5. Schreiben Sie in die nächste Zeile

```
locDr=
```

Auch hier wird IntelliSense wieder aktiv. Diesmal zeigt es Ihnen alle möglichen Member der Enum `DialogResult` an, da es davon ausgeht, dass Sie der Variablen `locDr` eines ihrer Elemente zuweisen wollen. Ignorieren Sie die Liste aber einfach in diesem Fall und schreiben Sie weiter (das `Show` brauchen Sie dabei dank Vervollständigungsfunktion auch nicht komplett selbst zu schreiben).

```
MessageBox.Show(
```

In dem Moment, in dem Sie die Klammer tippen, zeigt Ihnen IntelliSense eine vollständige Parameterinfo der `MessageBox.Show`-Methode (Abbildung 5.31), mit der Sie übrigens ein Meldungsfeld auf dem Bildschirm darstellen können.

```
Private Sub btnEingabenLöschen_Click(ByVal sender As System.Object, ByVal e A
    Dim locDr As DialogResult
    locDr=MessageBox.Show(
End Sub
Class
```

▲ 2 von 21 ▼ Show(**text As String**, caption As String) As System.Windows.Forms.DialogResult
Zeigt ein Meldungsfeld mit dem angegebenen Text und der angegebenen Beschriftung an.
text: *Der im Meldungsfeld anzuzeigende Text.*

Abbildung 5.31 Bei Methoden, die über Parameter verfügen, zeigt IntelliSense alle Parameter mitsamt Erklärungen der Parameter an. Der jeweils aktuelle Parameter, den Sie gerade eingeben, ist in Fettschrift gekennzeichnet.

Mehrzeilige Befehlszeilen, impliziter Zeilenumbruch und die Parameterinfo

Viele von Ihnen wissen bereits aus Visual Basic 6.0-Zeiten oder von Erfahrungen mit VBScript, dass Sie eine logische Codezeile in Visual Basic der besseren Übersicht wegen mithilfe des Unterstrichs (»_«; engl. *Underscore*) auf mehrere physische Zeilen im Editor verteilen konnten. Am Ende der (physischen) Zeile fügten Sie dazu ein Leerzeichen, gefolgt von einem »_« -Zeichen, ein. Die eigentliche (logische) Zeile schreiben Sie dann ganz normal weiter, als hätten Sie nie einen Zeilenumbruch eingefügt.

TIPP Ab Visual Basic 2010 ist das nicht mehr erforderlich, in den meisten Fällen umbrechen Sie die Zeile einfach, und schreiben in der nächsten Zeile weiter. Das funktioniert nach einem Komma, nach Klammer auf, nach einem Punkt – aber nicht in allen Fällen.

```
Dim locDr As DialogResult
locDr = MessageBox.Show("Sind Sie sicher?", "Eingabe löschen",
                        MessageBoxButtons.YesNo,
                        MessageBoxIcon.Question,
                        MessageBoxDefaultButton.Button2)
```

Abbildung 5.32 Ab Visual Basic 2010 brauchen Sie in vielen Fällen das Zeilenfortsetzungszeichen nicht mehr bemühen

Vor dem Handels-Schlüsselwort beispielsweise müssen Sie nach wie vor den Unterstrich zum Umbrechen verwenden. Im Folgenden üben wir den Zeilenumbruch deshalb auch mit dem Unterstrich, denn er führt in jedem Fall zum Ziel. In vielen Fällen können Sie ihn ab Visual Basic 2010 weglassen – probieren Sie einfach aus, welche Methode für Sie am besten funktioniert!

Um beim Beispiel zu bleiben:

1. Ergänzen Sie die Zeile (die ja noch nicht vollständig eingegeben wurde), um

```
"Sind Sie sicher?", "Eingaben löschen?", _
```

   und drücken anschließend [↵]. Nach dem Zeilenumbruch ist die Parameterinfo verschwunden. Um die Parameterinfo auch in der neuen Zeile wieder darzustellen, drücken Sie einfach [Strg]+[⇧]+[ ].

2. Geben sie nun den restlichen Teil der logischen Befehlszeile ein.

```
MessageBoxButtons.YesNo, _
MessageBoxIcon.Question, MessageBoxDefaultButton.Button2)
```

Automatische Vervollständigung von Struktur-Schlüsselwörtern und Codeeinrückung

1. Für die Komplettierung der Methode zum Löschen der Eingabefelder geben Sie die folgende Zeile ein:

```
If locDr = Windows.Forms.DialogResult.Yes Then
```

   Sobald Sie [↵] nach dem Eingeben dieser Zeile betätigt haben, fügt der Editor automatisch ein entsprechendes `End If` zwei Zeilen darunter ein und platziert die Einfügemarke zwischen den beiden Zeilen.

2. Wenn Sie nun die restlichen Zeilen

```
txtKurzbeschreibung.Text = ""
txtNameDesFilms.Text = ""
txtSchauspieler.Text = ""
picCoverbild.Image = Nothing
myBilddateiname = ""
```

   dazwischen eingeben, werden Sie feststellen, dass egal in welcher Spalte Sie zu tippen beginnen, die Zeilen sich immer der von `If … End If` vorgegebenen Struktur anpassen und entsprechend eingerückt formatiert werden.

HINWEIS Das funktioniert auch bei geschachtelten Strukturen; Sie behalten auf diese Weise immer den Überblick, in welcher Strukturverschachtelungsebene Sie sich gerade befinden.

Fehlererkennung im Code-Editor

Visual Basic verfügt über einen so genannten Hintergrundcompiler (*Background Compiler*). Dieser Hintergrundcompiler leistet einiges an Vorarbeit für den eigentlichen Compiler, der ja erst dann aktiv wird, wenn Sie ein Projekt erstellen (und das führt dazu, dass Anwendungen, die Sie in Visual Basic entwickeln, wesentlich kürzere Turn-Around-Zeiten[4] aufweisen, als die, die Sie in anderen .NET-Sprachen entwickeln. Dieser Hintergrundcompiler spart Ihnen eine ganze Menge Zeit beim Entwickeln, denn im Gegensatz zu anderen Sprachen wie C++ oder C# (oder auch zum alten Visual Basic 6.0) entdeckt der Hintergrundcompiler syntaktische Fehler bereits, nachdem Sie eine Codezeile vollständig eingegeben haben.[5]

Einfache Fehlerkennzeichnung im Editor

Wenn Sie die letzten Änderungen am Code aufmerksam nachvollzogen haben, bemerkten Sie sicherlich einen Fehler in der letzten Zeile, die Sie eingegeben haben. Dieser Fehler war auch gar nicht schwer zu bemerken, denn schließlich hat der Code-Editor diese Zeile gekennzeichnet, wie in Abbildung 5.33 zu sehen.

```
  If locDr = Windows.Forms.DialogResu
      txtKurzbeschreibung.Text = ""
      txtNameDesFilms.Text = ""
      txtSchauspieler.Text = ""
      picCoverbild.Image = Nothing
      myBilddateiname = ""
  End
d Sub
```

Der Name "myBilddateiname" wurde nicht deklariert.

Abbildung 5.33 Fehler, die der Hintergrundcompiler feststellt, werden direkt im Editor noch vor dem eigentlichen Kompilierungsvorgang beim Erstellen des Projekts gekennzeichnet

Diese betroffene Variable, die wir später im gesamten Formular für die Speicherung des Dateinamens benötigen, wurde nicht deklariert. Also machen wir das als Nächstes.

Editorunterstützung bei Fehlern zur Laufzeit

Direkt unterhalb der Klassendefinition fügen Sie die im Folgenden in Fettschrift formatierte Zeile ein:

```
Public Class Form1

    Private myBilddateiname As Integer
```

Sie sehen, dass der Fehler nun nicht mehr durch Unterschlängelung markiert ist.[6]

---

[4] Als »Turn-Around-Zeit« (etwa: »Wenden-Zeit«, wie das Wenden beim Autofahren) bezeichnet man die Zeitspanne, die vom Starten des Compilers über das Kompilieren eines Projekts bis zum eigentlichen Anwendungsstart vergeht.

[5] Hintergrundcompiler gibt es zwar auch in diesen anderen Sprachen, aber sie sind lange nicht so konsequent wie in Visual Basic 2010, und bei diesen anderen Sprachen passiert es oft, dass der eigentliche Compiler erst beim Erstellen des Projekts syntaktische Fehler oder nicht deklarierte Variablen findet. In Visual Basic 2010 passiert das äußerst selten – quasi nie.

[6] Kleiner Hinweis am Rande: Bevor Sie nun eine E-Mail schreiben, da Sie glauben, einen Fehler gefunden zu haben – warten Sie erst die nächsten Absätze ab! ;-)

Nun probieren wir das Programm in seinem derzeitigen Zustand aus.

1. Starten Sie das Programm mit F5
2. Geben Sie ein paar Zeilen in die Eingabefelder an
3. Klicken Sie auf *Eingaben löschen*
4. Bestätigen Sie das Meldungsfeld mit *Ja*

Statt die Eingabefelder zu löschen, bricht das Programm mit einer Meldung ab (Abbildung 5.34).

Abbildung 5.34 Tritt während der Anwendungsausführung im Debug-Modus ein Fehler (eine so genannte »Ausnahme« – engl. *Exception*) auf, ruft Visual Studio den Editor auf, unterlegt die betroffene Stelle gelb und zeigt einen entsprechenden Hinweis an

Fehler, die zur Laufzeit in einer .NET-Anwendung auftreten und dafür verantwortlich sind, dass ein Programm nicht fortgesetzt werden kann, werden als Ausnahmen (engl. *Exception* – sprich: »Ixäpschen«) bezeichnet. In diesem Fall ist uns ein Fehler beim Deklarieren der Variablen `myBilddateiname` passiert,[7] der schließlich zu dieser Ausnahme geführt hat. .NET versuchte zur Laufzeit, den Leerstring der Variablen zuzuweisen, die aber dummerweise als `Integer`-Variable deklariert wurde – das führte zu einer `InvalidCastException` (etwa: *Ausnahme wegen ungültiger Typkonvertierung*).

[7] O.k., o.k. – mir, nicht Ihnen.

Abbildung 5.35 Mithilfe des Ausnahmedetail-Dialogs erfahren Sie Genaueres über die Umstände, die zur Ausnahme führten

Wenn Sie in einem solchen Fall genauere Informationen zum Ausnahmenumstand benötigen, klicken Sie unten im Dialog bei *Aktionen* auf *Details anzeigen*. Der Editor zeigt Ihnen anschließend einen weiteren Dialog (Abbildung 5.35), mit dem Sie diese erweiterten Informationen abrufen können.

1. Für das weitere Nachvollziehen des Beispiels klicken Sie im Dialog auf *OK*.
2. Schließen Sie den darunter liegenden Dialog mit einem Klick auf die Schließen-Schaltfläche in der rechten oberen Ecke.
3. Beenden Sie das Debuggen: Wählen Sie dazu aus dem Menü *Debuggen* den Befehl *Debugging beenden* oder klicken Sie in der Symbolleiste auf das entsprechende Symbol zum Beenden des Debuggens.

Fehlerverbesserungsvorschläge des Editors bei Typkonflikten – erzwungene Typsicherheit

Passiert wäre das nicht, wenn wir in unserer Anwendung von vornherein Typsicherheit erzwungen hätten. In diesem Fall hätten wir den Fehler bereits bemerkt, noch bevor wir das Programm gestartet hätten – der Editor hätte uns darauf aufmerksam gemacht.

Seit Visual Basic 2002 kennt Visual Basic die Anweisung `Option Strict [On|Off]`. Schalten Sie `Option Strict` mit `On` ein, sorgt schon der Hintergrundcompiler dafür, dass Sie nur gleiche Typen zuweisen können – oder optional strikt dafür sorgen, dass eine saubere Typkonvertierung (beispielsweise von `Integer` zu `String`) stattfindet.

Sobald Sie die Anweisung

```
Option Strict On
```

ganz oben in den Code einfügen (noch vor der Klassendefinitionsanweisung `Class Form1`) und anschließend zurück zur betroffenen Zeile scrollen, sehen Sie, dass nicht nur der vermutlich zur Ausnahme führende Teil unterschlängelt ist, sondern am Ende auch ein längliches Rechteck (rote Umrandung, gelb gefüllt) aufweist.

Smarttags im Editor von Visual Basic

Einen Smarttag erkennen Sie zunächst als kleines, längliches Rechteck (man könnte es auch für einen Unterstrich halten) in einer Codezeile. Ein Smarttag erscheint dann im Code-Editor, wenn Visual Basic der Auffassung ist, dass es in der betreffenden Zeile noch irgendetwas zu verbessern oder anzumerken gibt.

Bewegen Sie den Mauszeiger auf diesen Smarttag, also das kleine Rechteck, verwandelt er sich in ein kleines Symbol mit der Form eines Ausrufezeichens, das Ihnen je nach Fall ganz verschiedene Arten von Unterstützung anbietet – nicht nur Hilfestellungen bei Typkonflikten, wie hier im Beispiel zu sehen.

Autokorrektur für intelligentes Kompilieren

In vielen Fällen verbirgt sich hinter dem Smarttag ein Dialog, der Ihnen einen Korrekturvorschlag für den erkannten »Fehler« unterbreitet – etwa, wie in der Abbildung zu sehen. Solche Hilfedialoge, die hinter Smarttags stecken, nennt Microsoft übrigens »*Autokorrektur für intelligentes Kompilieren*«. Falls die Autokorrektur für intelligentes Kompilieren Recht behält, und es sich tatsächlich um den vermuteten Fehler handelt, brauchen Sie noch nicht einmal die Tastatur zu bemühen: Klicken Sie einfach auf den blauen Korrekturvorschlag, und der Editor nimmt die Korrektur im Programmcode vor.

WICHTIG In unserem Beispiel greift die Autokorrektur für intelligentes Kompilieren an der Stelle des Fehlers leider nicht, weil es sich um einen Folgefehler handelt. Aber Sie sehen, dass wir auf jeden Fall einen kompletten Turn-Around-Lauf gespart hätten, wenn `Option Strict` von vornherein eingeschaltet gewesen wäre. Aus diesem Grund sollten Sie `Option Strict` projektweit einschalten und sich die Anweisung vor jeder Codedatei sparen. Außerdem sollten Sie die Visual Studio-Optionen so einstellen, dass `Option Strict` grundsätzlich beim Anlegen jedes neuen Projekts eingeschaltet ist. Wie das geht, zeigt der folgende graue Kasten.

Um den Fehler zu beheben, ändern Sie die Zeile

```
Private myBilddateiname As Integer
```

einfach in

```
Private myBilddateiname As String
```

Erzwungene Typsicherheit (Option Strict) projektweit einstellen

Sie können dafür sorgen, dass Typsicherheit grundsätzlich in einem Projekt erzwungen wird, ohne dass Sie dazu `Option Strict On` über jede Codedatei schreiben müssen. Dazu öffnen Sie im Projektmappen-Explorer das Kontextmenü des entsprechenden Projekts (nicht der Projektmappe!) und wählen *Eigenschaften*. Die Projekteigenschaften öffnen sich nun als Dokumentenfester in der Dokumentenregisterkartengruppe, und Sie können auf der Registerkarte *Kompilieren* `Option Strict` global mit *On* einschalten.

Erzwungene Typsicherheit für alle folgenden neuen Projekte

Möchten Sie, dass diese Einstellung grundsätzlich gilt, wenn Sie ein neues Projekt anlegen, müssen Sie den Optionsdialog von Visual Studio bemühen. Rufen Sie ihn mit *Extras/Optionen* auf, und wählen Sie den Bereich *Projekte und Projektmappen*. Auf der Registerkarte *VB-Standard* nehmen Sie die Einstellungen für `Option Strict` vor.

XML-Dokumentationskommentare für IntelliSense bei eigenen Objekten und Klassen

Wie Ihnen IntelliSense beim Finden der richtigen Klassen, Objekte, Methoden und anderer Elemente bei der Entwicklung von Anwendungen helfen kann, haben Sie bereits kennen gelernt. Doch damit ist noch lange nicht das Ende der Fahnenstange erreicht.

Zur Demonstration müssen wir ein wenig mehr vorbereitenden Aufwand betreiben. Dazu implementieren wir im Folgenden eine Methode, die eine Bilddatei aus einer Datei in ein `Image`-Objekt lädt. Auch hier soll die eigentliche Funktionsweise nicht von primärem Interesse sein – es geht schließlich immer noch um die Erarbeitung der Funktionalitäten der Code-Editor-Fähigkeiten.

1. Fügen Sie zu diesem Zweck die folgenden Zeilen in den Klassencode von *Form1* ein:

```
Function CoverbildAusDateinamen(ByVal CoverbildDateiname As String) As Image
    Dim locImage As Image
    If CoverbildDateiname IsNot Nothing AndAlso CoverbildDateiname <> "" Then
        locImage = Image.FromFile(CoverbildDateiname)
        Return locImage
    End If
    Return Nothing
End Function
```

2. Wechseln Sie mit F7 zum Entwurfsmodus (zum Designer-Dokumentenfenster). Übrigens: Mit F7 wechseln Sie zwischen Designer- und Codedarstellung eines Formulars.
3. Klicken Sie doppelt auf die Auslassungsschaltfläche neben der `PictureBox`, um den Code-Editor zu öffnen und die Ereignisbehandlungsroutine für diese Schaltfläche zu bearbeiten.
4. Fügen Sie in die Stub[8] (in den Funktionsrumpf, also zwischen `Private Sub btnCoverbildWählen …` und `End Sub`) folgende Zeilen ein:

[8] Sprich: »stap«.

```
        Dim locOfd As New OpenFileDialog

        With locOfd
            locOfd.CheckFileExists = True
            locOfd.DefaultExt = "*.bmp"
            locOfd.Filter = "JPeg-Bilder (*.jpg)|*.jpg|Windows Bitmap (*.bmp)|*.bmp|Alle Dateien
(*.*)|*.*"

            Dim locDr As DialogResult = locOfd.ShowDialog()
            If locDr = Windows.Forms.DialogResult.Cancel Then
                Return
            End If

            myBilddateiname = locOfd.FileName
        End With
```

5. Zwischen den letzten beiden Zeilen ergänzen Sie nun eine weitere Zeile, die Sie noch nicht komplett eingeben:

```
picCoverbild.Image = CoverbildAusDateinamen(
```

Sobald Sie die Klammer getippt haben, sehen Sie, dass IntelliSense auch bei selbst geschriebenen Methoden (und anderen Elementen wie Eigenschaften etc.) greift. Doch schauen Sie sich Abbildung 5.36 an. Fällt Ihnen etwas auf?

Abbildung 5.36 Auch bei selbst geschriebenen Methoden funktioniert IntelliSense – natürlich fehlen dabei zunächst noch die Erklärungen

Genau wie bei eingebauten Methoden wird IntelliSense zwar aktiv – doch die sonst so hilfreichen Erklärungen finden wir hier natürlich nicht. Woher sollen sie auch stammen! Noch in Visual Basic 2003 war ohne Tools von Drittherstellern in Sachen IntelliSense bei selbst entwickelten Methoden an dieser Stelle Schluss. Doch mit Visual Basic 2010 ist das anders, wie Sie gleich sehen werden:

1. Bevor Sie nämlich die Zeile nun komplettieren, positionieren Sie die Einfügemarke oberhalb der Zeile

```
Function CoverbildAusDateinamen(ByVal CoverbildDateiname As String) As Image
```

2. Tippen Sie drei Hochkommas (⇧+#).
3. Sobald Sie das dritte Hochkomma getippt haben, wird der Code-Editor aktiv und greift Ihnen unter die Arme (Abbildung 5.37).

```
''' <summary>
''' |
''' </summary>
''' <param name="CoverbildDateiname"></param>
''' <returns></returns>
''' <remarks></remarks>
Function CoverbildAusDateinamen(ByVal CoverbildDateiname As String) As Image
    Dim locImage As Image
    If CoverbildDateiname IsNot Nothing AndAlso CoverbildDateiname <> "" Then
        locImage = Image.FromFile(CoverbildDateiname)
        Return locImage
    End If
    Return Nothing
End Function
```

Abbildung 5.37 Der Editor fügt ein XML-Skelett oberhalb der Methode ein, in der Sie Ihre Dokumentation nur zu ergänzen brauchen

4. Für die folgenden Schritte orientieren Sie sich an Abbildung 5.38. Geben Sie zwischen `<summary>` und `</summary>` eine Funktionsbeschreibung ein.

```
''' <summary>
''' Lädt ein Bild, so vorhanden, aus einer Datei und liefert das Bild als
''' Image (oder Nothing) zurück
''' </summary>
''' <param name="CoverbildDateiname">Name der Coverbild Datei</param>
''' <returns>Image Objekt, das das Bild des angegebenen Datei enthält</returns>
''' <remarks>Erstellt von Klaus Löffelmann</remarks>
Function CoverbildAusDateinamen(ByVal CoverbildDateiname As String) As Image
    Dim locImage As Image
    If CoverbildDateiname IsNot Nothing AndAlso CoverbildDateiname <> "" Then
        locImage = Image.FromFile(CoverbildDateiname)
        Return locImage
    End If
    Return Nothing
End Function
```

Abbildung 5.38 Komplettieren Sie die XML-Tags etwa nach dieser Vorlage.

5. Zwischen `<param name="CoverbildDateiname">` und `</param>` geben Sie die Bedeutung des Parameters *CoverbildDateiname* an.

HINWEIS Sollten Sie es zu einem späteren Zeitpunkt mit Methoden oder Eigenschaften zu tun haben, die mehrere Parameter entgegennehmen, werden an dieser Stelle weitere `param`-Tags aufgelistet, zwischen denen Sie die Beschreibungen der weiteren Parameter platzieren können.

6. Zwischen `<returns>` und `</returns>` geben Sie die Bedeutung des Rückgabewerts an.

7. Optional geben Sie zwischen `<remarks>` und `</remarks>` eine Bemerkung ein.

HINWEIS Außer den hier von uns benutzten Tags wie `<returns>` und `<remarks>` sind noch weitere möglich. Schauen Sie in die Hilfe unter dem Stichwort »Empfohlene XML-Tags für Dokumentationskommentare« nach.

8. Wenn Sie alle Eingaben abgeschlossen haben, kehren Sie zur noch nicht vollständig eingegebenen Zeile

```
picCoverbild.Image = CoverbildAusDateinamen(
```

zurück. Löschen Sie alle Zeichen, bis nur noch »Cov« von »CoverbildAusDateinamen(« übrig bleibt, und drücken Sie anschließend [Strg]+[ ]. Sie sehen nun, dass die Funktion nicht nur in der Vervollständigungsliste zu sehen ist (das war sie zuvor auch schon), sondern nunmehr auch die Funktionsbeschreibung enthält, die Sie mithilfe der XML-Tags angegeben haben (Abbildung 5.39).

Abbildung 5.39 Wenn Sie Ihre Methoden und Eigenschaftenprozeduren mit entsprechenden XML-Tags versehen haben, zeigt IntelliSense sowohl eine Funktionsbeschreibung ...

9. Drücken Sie [Strg]+[↵], um den Funktionsnamen zu vervollständigen.

10. Tippen Sie die Klammer, wird IntelliSense wieder aktiv (Abbildung 5.40). Sie sehen, dass IntelliSense nun nicht nur den Parameternamen und den Parametertyp, sondern auch die Parameterdokumentation zeigt.

Abbildung 5.40 ... als auch entsprechende Parametererklärungen an

11. Vervollständigen Sie die Zeile, sodass diese komplett wie folgt lautet:

```
picCoverbild.Image = CoverbildAusDateinamen(myBilddateiname)
```

HINWEIS Übrigens: So ganz nebenbei haben Sie mit nur ein paar Zeilen Code die komplette Bilddarstellung im Formular implementiert – und dank der Einstellungen, die Sie zuvor im Designer vorgenommen haben, läuft die Bildausschnittswahl mit Bildlaufleiste ebenfalls schon. Den Beweis dazu können Sie selbst antreten: Starten Sie das Programm mit [F5], klicken Sie auf die Auslassungsschaltfläche und wählen Sie im Dialog, der jetzt gezeigt wird, eine Bilddatei aus.

Hinzufügen neuer Codedateien zum Projekt

Für unser Beispiel benötigen wir eine Datenstruktur, mit der die Eingaben, die der Benutzer im Hauptformular vorgenommen hat, an das Druckformular übergeben werden. Das Programm verwendet also diese Datenstruktur hier im Beispiel, um die einzelnen Datenfelder in dieser Datenstruktur – nennen wir Sie `CoverInhalt` – zunächst zwischenzuspeichern und sie dann in einem Rutsch an das Druckformular zu übergeben.

Auf diese Weise sind die beiden Aufgaben – Datenerfassung und Drucken – sauber voneinander getrennt. Der Hauptdialog sorgt in eigener Regie für die Datenerfassung und der Druckdialog selbstständig für das Drucken der Daten. Der Hauptdialog muss so nicht auf den Druckdialog direkt zugreifen und dort irgendwelche Variablen manipulieren, sondern sagt dem Druckdialog nur mithilfe einer einzelnen öffentlichen Funktion, wie dieser das Cover in der Vorschau darstellen oder auf einem Drucker ausgeben soll.

Eleganterweise bringen wir diese Datenstruktur in einer eigenen Codedatei unter: der Klassendatei, die den gleichen Namen wie die Datenstruktur (die Klasse) selbst bekommen soll: `CoverInhalt`.

HINWEIS Visual Basic 6.0-Programmierer kennen Datenstrukturen der einfachsten Ausführung in Form von benutzerdefinierten Typen. Ohne den Klassen- oder Umstiegsteil des Buchs vorwegzunehmen: Die einfachste vorstellbare Klasse, wie Sie sie gleich kennen lernen werden, entspricht in etwa der Definition eines neuen Typen mit `Type`.

Abbildung 5.41 So fügen Sie eine neue Klassendatei zum Projekt hinzu

1. Um eine neue Klassendatei zu einem Projekt hinzuzufügen, verfahren Sie auf fast genau dieselbe Art und Weise, wie Sie es beim Hinzufügen einer Formular-Klassendatei schon getan haben. Rufen Sie das Kontextmenü des Projekts *Covers* (nicht der Projektmappe!) im Projektmappen-Explorer auf.
2. Wählen Sie *Hinzufügen/Klasse*.
3. Im jetzt erscheinenden Dialog geben Sie den Namen der Klassendatei (ohne Dateiendung) ein – für unser Beispiel **CoverInhalt**.
4. Klicken Sie auf *Hinzufügen* oder drücken Sie [↵].

   Der Code-Editor wird geöffnet; die Klassendefinition ist bereits vorgegeben. Tragen Sie nun zwischen `Public Class CoverInhalt` und `End Class` die folgenden Zeilen ein:

```
Public FilmTitel As String
Public Schauspieler As String
Public Beschreibung As String
Public Coverbild As Image
```

Mit der Einführung dieser neuen Klasse haben wir die Möglichkeit, die Daten aus dem Hauptformular auszulesen und dem Druckformular zu übergeben. Genau das werden wir als Nächstes implementieren.

TIPP Damit dieser Abschnitt für Sie nicht in ein endloses Getippe ausartet, werden wir es uns mit dem Code für das Drucken einfach machen und diesen aus einer Textdatei in das Formular hineinkopieren. Sie finden den Code deswegen als reine Textdatei: ...\VB 2010 Entwicklerbuch\Kapitel 05\Covers\Druckroutine für Covers.txt.

1. Öffnen Sie die Textdatei *Druckroutine für Covers.txt* mit Notepad von Windows.
2. Drücken Sie [Strg]+[A] (alles markieren) und [Strg]+[C] (in die Zwischenablage kopieren).
3. Schließen Sie Notepad.
4. Wechseln Sie zurück zu Visual Studio.
5. Klicken Sie im Projektmappen-Explorer auf *Form2.vb* und anschließend in der Symbolleiste des Projektmappen-Explorers auf das Symbol *Code anzeigen*.
6. Drücken Sie [Strg]+[A] (alles markieren) und [Strg]+[V] (Zwischenablageinhalt einfügen). Damit ist der Code für das Druckformular vollständig. Die genaue Funktionalität soll uns an dieser Stelle noch nicht interessieren; es würde bedeuten, zu viele Themen vorwegnehmen zu müssen, deren ausführliche Erklärung für ein genaueres Verständnis erforderlich wäre.
7. Speichern Sie alle Änderungen und schließen Sie das Dokument *Form2.vb*.

Um den Code zu implementieren, der eine neue Instanz der Klasse *CoverInhalt* bildet, sie mit Daten aus den Eingabefeldern füttert und die Klasse zur Weiterverarbeitung dem Druckformular übergibt, verfahren Sie wie folgt:

1. Wechseln Sie mit [Strg]+[⇆] zum Entwurfsmodus von *Form1.vb* (also zu *Form1.vb [Entwurf]*).
2. Klicken Sie doppelt auf die Schaltfläche *Inlay drucken*, um dafür zu sorgen, dass die Stub für die Ereignisbehandlungsroutine von `btnInlayDrucken_Click()` im Code eingefügt wird.
3. Tippen Sie den folgenden Code ein (die Kommentare dienen nur zur Groberklärung des Codes und müssen natürlich nicht mit eingegeben werden):

```
'CoverInhalt-Klasse in ein Objekt instanziieren
Dim locCoverInhalt As New CoverInhalt

'Dem CoverInhalt-Objekt die Daten zuordnen
locCoverInhalt.FilmTitel = txtNameDesFilms.Text
locCoverInhalt.Schauspieler = txtSchauspieler.Text
locCoverInhalt.Beschreibung = txtKurzbeschreibung.Text
locCoverInhalt.Coverbild = picCoverbild.Image

'Die Druckvorschau aufrufen und das CoverInhalt-Objekt
'mit den Daten übergeben.
Dim locCoverDruckenForm As New Form2
locCoverDruckenForm.DialogDarstellen(locCoverInhalt)
```

Code umgestalten (Refactoring)

Grundlegende Refactoring-Funktionalitäten wurden in Visual Studio 2008 eingeführt. Visual C# 2008 wurde dabei leider in viel größerem Umfang mit entsprechenden Werkzeugen bedacht als Visual Basic 2008.[9] Im Grunde genommen gab es in Visual Basic 2008 nur eine einzige Funktion, die das automatische Umgestalten von Code erlaubt – das Umbenennen von Namen von Methoden, Eigenschaften, Objekten oder Ereignissen – in Visual Basic 2010 ist immerhin noch die »Consume-First-Funktionalität« hinzugekommen, die Sie im vorherigen Kapitel schon kennen gelernt haben.

Was bedeutet Refactoring genau?

Stellen Sie sich vor, Sie entwickeln eine relativ umfangreiche Funktion, und Sie bemerken nach einer Weile, dass nur diese eine Funktion bereits aus 1000 Zeilen Code besteht. Sie müssen (oder sollten zumindest) sich dann eingestehen, dass Sie das Problem, das Sie lösen wollen, besser auf mehrere Unterfunktionen verteilt hätten. Aber dazu ist es ja nicht zu spät. Wenn Sie aus einem Teil der Funktion nun eine neue Unterfunktion generieren, dann betreiben Sie bereits aktives Refactoring.

Am schlimmsten ist es bei solchen Aktionen, wenn sich Namen, die Sie beispielsweise Methoden gegeben haben, als falsch oder nicht ausreichend aussagekräftig entpuppen. Gerade wenn Sie im Team arbeiten, sind Sie auf dieses Problem sicherlich schon das ein oder andere Mal gestoßen. Und jeder, der eine Funktion aus solchen Gründen umbenennen musste, weiß, was die Umbenennerei für eine Arbeit macht, weil die Funktion doch viel öfter referenziert wird, als man es eigentlich erwartet hätte.

Suchen und Ersetzen kommt für diesen Zweck auch nicht in Frage, denn stellen Sie sich vor: Sie möchten eine Eigenschaft namens »bindControl« in »BoundControl« umbenennen. Es gibt aber auch eine Methode namens »UnbindControl«, an die Sie aber überhaupt nicht mehr denken und die Sie aus Versehen und ohne es zu merken, ebenfalls umbenennen, nämlich in »UnboundControl«.

Noch ernster wird es bei der Umbenennung von Variablen, die Sie in verschiedenen Gültigkeitsbereichen mehrfach verwenden. Wenn Sie beispielsweise eine Methode in zwei Klassen implementiert haben, dann möchten Sie unter Umständen, dass sich nur der Name der Methode in einer Klasse ändert – und überall dort, wo Sie ihn verwendet haben. Mit Suchen und Ersetzen wäre das fehlerfrei zu tun fast ein Ding der Unmöglichkeit.

---

[9] Es gibt allerdings eine kostenfreie Version des Add-Ins Refactor!, die Sie sich unter *http://tinyurl.com/66u4la* herunterladen können.

Hier kommt eines der beiden Refactoring-Werkzeuge von Visual Basic 2010 ins Spiel – das Umbenennen-Werkzeug. Das Umbenennen von Namen über das Refactoring bezieht sich immer nur auf das Element, das Sie umbenennen, und es benennt nicht nur das Element, sondern auch alle Referenzen um. Um das auszuprobieren, machen Sie Folgendes:

1. Suchen Sie im Code von *Form1.vb* die Funktion *CoverbildAusDateinamen*.
2. Klicken Sie mit der rechten Maustaste auf den Funktionsnamen, und aus dem Kontextmenü, das der Editor nun öffnet, wählen Sie *Umbenennen*.
3. Im Dialog, der jetzt dargestellt wird, geben Sie einen neuen Namen für die Funktion ein – beispielsweise **GetCoverImage** (Abbildung 5.42).

Abbildung 5.42 Mit dem Umbenennen beispielsweise einer Methode (einer Funktion) ändern Sie nicht nur deren Namen, sondern auch alle ihre Referenzen

4. Klicken Sie auf *OK*.
5. Bewegen Sie die Einfügemarke zur Funktion `btnCoverbildWählen_Click`. Sie werden hier erkennen, dass nicht nur der Name der Funktion, sondern auch die (zugegebenermaßen einzige) Referenz auf die Funktion geändert wurde, wie im Listing der Funktion fett markiert zu sehen:

```
    Private Sub btnCoverbildWählen_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
Handles btnCoverbildWählen.Click
        Dim locOfd As New OpenFileDialog

        With locOfd
            locOfd.CheckFileExists = True
            locOfd.DefaultExt = "*.bmp"
            locOfd.Filter = "JPeg-Bilder (*.jpg)|*.jpg|Windows Bitmap (*.bmp)|*.bmp|Alle Dateien
               (*.*)|*.*"

            Dim locDr As DialogResult = locOfd.ShowDialog()
            If locDr = Windows.Forms.DialogResult.Cancel Then
                Return
            End If

            myBilddateiname = locOfd.FileName
            picCoverbild.Image = GetCoverImage(myBilddateiname)
        End With
    End Sub
```

Code umgestalten (Klassennamenanpassung) beim Umbenennen von Projektdateien oder Objekteigenschaften

Solange Sie Ihre Klassendateien oder Formulardateien genauso nennen wie die Klassen oder Formularklassen, die diese speichern, funktioniert das Umbenennen hier genauso – mit dem Unterschied, dass Sie auch die physische Datei umbenennen, die den eigentlichen Klassencode enthält.

Probieren Sie es aus:

1. Öffnen Sie das Kontextmenü von *Form2.vb* im Projektmappen-Explorer.
2. Wählen Sie *Umbenennen*.
3. Geben Sie einen neuen Namen für die Formulardatei ein – beispielsweise **frmCoverDrucken.vb**.

HINWEIS Sie benennen dabei eine physische Datei um. Geben Sie deswegen die Dateiendung unbedingt mit ein!

4. Drücken Sie ↵. Beantworten Sie eine möglicherweise erscheinende Sicherheitsabfrage entsprechend.
5. Bewegen Sie die Einfügemarke zur Funktion `btnInlayDrucken_Click` im Code von *Form1*. Sie sehen, dass die letzten Zeilen dieser Funktion nunmehr folgendermaßen lauten:

```
'Die Druckvorschau aufrufen und das CoverInhalt-Objekt
'mit den Daten übergeben.
Dim locCoverDruckenForm As New frmCoverDrucken
locCoverDruckenForm.DialogDarstellen(locCoverInhalt)
```

6. Woran das liegt, sehen Sie, wenn Sie den Code von »ehemals« *Form2.vb* (und nunmehr *frmCoverDrucken.vb*) öffnen. Durch das Umbenennen der Datei wurde auch der Klassenname in `frmCoverDrucken` geändert und dementsprechend die geänderte Referenz, die Sie sich gerade in der Funktion `btnInlayDrucken_Click` angeschaut haben.

HINWEIS Bei umfangreichen Projekten mit vielen Formulardateien oder Klassendateien kann sich das Refactoring von Klassen- und Formularklassen durch das Umbenennen Ihrer Codedatei als störend erweisen. Sie können das »Dateinamen-Umbenennen-Refactoring« im *Optionen*-Dialog von Visual Studio ausschalten. Rufen Sie diesen Dialog dazu aus dem *Extras*-Menü auf, und wählen Sie das Register *Windows Forms-Designer*. Wie in Abbildung 5.43 zu sehen, setzen Sie die `EnableRefactoringOnRename`-Eigenschaft auf `False`.[10]

[10] Die in Visual Studio 2010 enthaltenen Tools zum Refactoring gehen auf die Werkzeuge einer Firma namens Developer Express zurück. Wenn Sie das vollständige Tool zum Refactoring benutzen wollen, können Sie dies unter *http://tinyurl.com/5z32yt* als Testversion oder kostenpflichtige Vollversion beziehen.

Abbildung 5.43 Hier schalten Sie das Refactoring durch Umbenennen von Codedateien aus

Die Bibliothek der Codeausschnitte (Code Snippets Library)

Unser Beispielprogramm kann sich bis zu diesem Zeitpunkt schon sehen lassen. Allerdings läuft es noch nicht wirklich fehlerfrei und ist nicht gegen »groben Unfug« geschützt. Warum? Nun, Sie könnten beispielsweise versuchen, statt einer Bilddatei eine Textdatei als Coverbild zu laden. Einen solchen Versuch quittiert das Programm direkt mit einer Ausnahme – einer Laufzeitfehlermeldung –, die es aber nicht abfängt (Abbildung 5.44).

Abbildung 5.44 Das Laden einer kleinen Textdatei führt zu einer »Zu wenig Speicher«-Ausnahme? Das lässt Platz für Vermutungen, wird aber wohl an den internen Grafikfiltern liegen, die Textbytes fälschlicherweise als Größenangaben für eine zu ladende Grafik interpretieren. So tritt der Fehler bereits beim Reservieren des Speichers für die Grafik und nicht erst beim Lesen der »falschen« Bytefolgen auf.

In einem professionellen Programm darf so etwas natürlich nicht passieren – schon gar nicht, wenn die ausgelöste Ausnahme, wie hier im Beispiel, den Benutzer auf eine völlig falsche Fährte lockt (siehe Bildunterschrift).

Schön wäre es überdies, wenn dieses Fehlverhalten nicht nur nicht zum Abbruch des Programms führen, sondern den Fehler einer zuständigen Stelle obendrein noch melden würde. Zum Beispiel, indem es versucht, eine E-Mail an einen Administrator zu schicken.

Ich würde Ihnen gerne erklären, wie das funktioniert. Aber wissen Sie was? Ich kann es nicht. Ich müsste dazu stundenlang recherchieren, und ob ich die Infos zur Programmierung eines solchen Features selbst dann überhaupt finden würde, wäre fraglich… ;-)

Aber wissen Sie noch was: Das muss ich auch gar nicht. Denn Visual Basic 2010 verfügt über eine Codeausschnittsbibliothek, die für jeden Geschmack etwas Passendes bereitstellt. Zum Beispiel, wie man einen Coderumpf zum Abfangen eines Fehlers implementiert. Und das geht so:

1. Schließen Sie zunächst den Dialog mit der Ausnahme, der immer noch auf dem Bildschirm zu sehen sein sollte.
2. Stoppen Sie das Programm mit einem Klick auf das *Debugging beenden*-Symbol (der ToolTip des Symbols hilft Ihnen, das richtige zu finden).
3. Bewegen Sie die Einfügemarke in die Methode `GetCoverImage`, und zwar so, dass sie sich genau am Anfang dieser Zeile befindet:

```
locImage = Image.FromFile(CoverbildDateiname)
```

   Rufen Sie mit der rechten Maustaste das Kontextmenü auf und wählen Sie *Ausschnitt einfügen*.
4. Sie sehen nun eine Liste mit Ordnern, die die verschiedenen Codeausschnitt-Oberbegriffe enthalten. Klicken Sie auf *Codemuster - If, For, Each, Try Catch, Property usw.*
5. Klicken Sie doppelt auf *Fehlerbehandlung (Ausnahmen)*.

Abbildung 5.45 Beim Einfügen eines Codeausschnitts (Code Snippet) sehen Sie die verschiedenen Kategorien hierarchisch nebeneinander stehen. Ein ToolTip gibt Ihnen genauere Infos zum ausgewählten Ausschnitt. Achten Sie dabei auch auf die dort ausgewiesene Verknüpfungszeichenfolge für »das nächste Mal«!

6. In der Liste, die sich daraufhin öffnet, klicken Sie doppelt auf *Try…Catch…End Try-Anweisung* (Abbildung 5.45).

HINWEIS Merken Sie sich am besten dabei gleich die Verknüpfung, die der ToolTip anzeigt, für das nächste Mal, wenn Sie den Codeausschnitt häufiger benötigen. Sie können diese Verknüpfung dann später verwenden, um mit weniger Aufwand den Codeausschnitt einzufügen.

Der Editor fügt nun den kompletten Rumpf zum Abfangen eines Fehlers ein, den Sie im Prinzip nur ein wenig umgestalten müssen, etwa wie in den folgenden Codezeilen zu sehen:

```
        If CoverbildDateiname IsNot Nothing AndAlso CoverbildDateiname <> "" Then
            Try
                'Versuche das hier ohne Fehler, und...
                locImage = Image.FromFile(CoverbildDateiname)
                '...wenn kein Fehler auftrat, liefere das Ergebnis zurück:
                Return locImage
            Catch ex As Exception
                'Beim Auftreten eines Fehlers landet man hier.

            End Try
        End If
```

HINWEIS Denken Sie dabei auch daran, `ApplicationException` in `Exception` umzuwandeln, damit nicht nur Ausnahmen vom Typ `ApplicationException`, sondern alle denkbaren Ausnahmen abgefangen werden können, so wie es Kapitel 2 schon beschrieben hat.

Mit dieser Codeänderung haben wir den Fehler auf alle Fälle schon mal abgefangen. Jetzt müssen wir im `Catch`-Block nur noch dafür sorgen, auf ihn auch entsprechend zu reagieren – zum Beispiel, indem wir eine E-Mail an einen zuständigen Administrator versenden.

Einfügen von Codeausschnitten mithilfe von Verknüpfungen

Auch dafür kommen uns die Codeausschnitte zu Hilfe. In der Kategorienliste *Sonstiges - Konnektivität, Sicherheit, Workflow*, Unterkategorie *Konnektivität und Netzwerk*, findet sich ein Eintrag namens *E-Mail-Nachricht erstellen*, der übrigens die Verknüpfungszeichenfolge *conEmail* trägt. Wenn Sie, wie in diesem Fall, die Verknüpfungszeichenfolge kennen, brauchen Sie sie lediglich an die Stelle im Code einzufügen und ⇥ zu drücken, an der der ganze Ausschnitt erscheinen soll. Und so wird …

```
    Function GetCoverImage(ByVal CoverbildDateiname As String) As Image
        Dim locImage As Image
        If CoverbildDateiname IsNot Nothing AndAlso CoverbildDateiname <> "" Then
            Try
                'Versuche das hier ohne Fehler, und...
                locImage = Image.FromFile(CoverbildDateiname)
                '...wenn kein Fehler auftrat, liefere das Ergebnis zurück:
                Return locImage
            Catch ex As Exception
                'Beim Auftreten eines Fehlers landet man hier.
                conEmail
            End Try
        End If
        Return Nothing
    End Function
```

… nach dem Drücken von ⇥ hinter *conEmail* die Funktion wie in Abbildung 5.46 zu sehen, abgeändert.

```
    If CoverbildDateiname IsNot Nothing AndAlso CoverbildDateiname <> "" Then
        Try
            'Versuche das hier ohne Fehler, und...
            locImage = Image.FromFile(CoverbildDateiname)
            '...wenn kein Fehler auftrat, liefere das Ergebnis zurück:
            Return locImage
        Catch ex As Exception
            'Beim Auftreten eines Fehlers landet man hier.
            Return Nothing

            Dim message As New MailMessage("sender@address", "from@address", "Subject", "Message Text")
            Dim emailClient As New SmtpClient("Email Server Name")
            emailClient.Send(message)

        End Try
```

Abbildung 5.46 Der eingefügte Codeausschnitt ist hier zu sehen. Mit ⇥ springen Sie bequem von Parameter zu Parameter und ändern diese in einem Rutsch.

Sie brauchen nun nur mit ⇥ von Parameter zu Parameter zu springen und die entsprechend gültigen Werte einzutragen, bis sich im `Catch`-Block beispielsweise folgendes Bild ergibt:

```
.
.
.
        Catch ex As Exception
            'Beim Auftreten eines Fehlers landet man hier.

            Dim message As New MailMessage("covers@loeffelmann.de", "klaus@loeffelmann.de", _
                            "Fehler bei der Programmausfühung", ex.Message)
            Dim emailClient As New SmtpClient("192.168.0.1")
            emailClient.Send(message)
        End Try
.
.
.
```

HINWEIS Damit – und das sei nur der Vollständigkeit halber erwähnt – dieses Beispiel auf Ihrem System laufen kann, müssen Sie natürlich einen entsprechend konfigurierten SMTP-(Mail-)Server im Netzwerk zur Verfügung haben. Tragen Sie dann die für Sie gültigen Daten anstelle der hier im Listing abgedruckten E-Mail-Daten ein, auch die hier angegebene TCP/IP-Nummer (192.168.0.1) müssen Sie durch die TCP/IP-Nummer oder den Hostnamen (zum Beispiel smtp.web.de) Ihres SMTP-Servers ersetzen. Falls Sie auf SMTP-Server zugreifen müssen, die keine offene Relay-Funktion unterstützen,[11] ergänzen Sie im Bedarfsfall noch folgende Zeile (fett im folgenden Listingauszug), mit der Sie die Anmeldeinformationen übergeben.

[11] Ein Weiterleiten von und an beliebige E-Mail-Adressen ohne Anmeldung sollte bei den meisten SMTP-Servern (hoffentlich!) nicht gegeben sein, es sei denn, sie erlauben das Relaying aufgrund Verwendung der integrierten Sicherheit in Active Directory-Netzwerken. In diesen Fällen übernimmt die Anmeldung an das Netzwerk beim Starten von Windows auch das implizite Anmelden am SMTP-Server im Bedarfsfall.

```
.
.
.
        Catch ex As Exception
            'Beim Auftreten eines Fehlers landet man hier.

            Dim message As New MailMessage("covers@loeffelmann.de", "klaus@loeffelmann.de",
                            "Fehler bei der Programmausfühung", ex.Message)
            Dim emailClient As New SmtpClient("192.168.0.1")
            emailClient.Credentials = New Net.NetworkCredential("Username", "Passwort")
            emailClient.Send(message)
        End Try
.
.
.
```

Speichern von Anwendungseinstellungen mit dem Settings-Designer

Viele Anwendungen müssen beim Beenden Einstellungen speichern. Früher war das ein vergleichsweise großer Aufwand, denn Anwendungen mussten sich komplett selbst um die so genannte Serialisierung[12] ihrer Einstellungsdaten kümmern.

Programmierte Methoden innerhalb der Anwendung hatten dafür zu sorgen, die wichtigen Daten so aufzubereiten, dass sie im richtigen Format gespeichert werden konnten. Das Konvertieren in das richtige Format (beispielsweise numerische Werte in Zeichenketten beim Serialisieren oder Zeichenketten in numerische Werte beim Deserialisieren) stellte dabei den größten Aufwand dar, weil falsche Typkonvertierungen (Datum liegt als Zeichenkette vor, es wurde aber beispielsweise versucht, die Zeichenkette in eine numerische Variable zu konvertieren) in vergleichsweise großem Aufwand abgefangen und ausgeschlossen werden mussten.

In .NET 4.0 bzw. Visual Studio 2010 geht das ungleich einfacher. Interaktiv können Sie mit einem speziellen Designer Einstellungsvariablen einrichten, auf die Sie dann von Ihrer Anwendung aus zugreifen und diese speziell zum Speichern von Anwendungseinstellungen verwenden können. Und das Tolle: Eine Visual Basic-Anwendung kümmert sich automatisch darum, dass die Inhalte dieser Variablen, wenn Sie es wünschen, beim Programmende gesichert und beim nächsten Programmstart automatisch wieder gestartet werden.

Einrichten von Settings-Variablen

Der Designer zum Einrichten der Settings-Variablen verbirgt sich in den Projekteigenschaften.

1. Um diesen Designer also aufzurufen, wählen Sie aus dem Kontextmenü des Projekts *Covers* (nicht der Projektmappe!) im Projektmappen-Explorer den Menüeintrag *Eigenschaften*.
2. Wechseln Sie auf die Registerkarte *Einstellungen*.

[12] Serialisierung nennt man den Vorgang, bei dem ein Objekt, das innerhalb einer Anwendung Daten im Hauptspeicher speichert, diese durch einen Datenstrom in einen Zielspeicher (seriell) ablegt, entweder zu dem Zweck, die Daten für die Weiterverwendung durch ein Objekt gleicher »Bauart« bereitzustellen oder dauerhaft auf einem Datenträger zu speichern. Beim entgegengesetzten Vorgang entsteht aus einem Datenstrom wieder die ursprüngliche Instanz des Objekts – dieser Vorgang entspricht also dem Laden der Daten in den Hauptspeicher. Mehr zum Thema Serialisierung finden Sie im gleichnamigen Kapitel dieses Buchs.

3. Für unser Beispielprogramm möchten wir, dass die Eingaben, die der Benutzer in den Texteingabefeldern zur Laufzeit vorgenommen hat, zum Programmende gespeichert und, wenn das Programm erneut startet, wieder geladen und in die Eingabefelder geschrieben werden. Unsere erste Settings-Variable nennen wir daher `LetzterFilmname`, und sie soll vom Typ `String` sein, da sie Zeichenketten speichert.
4. Klicken Sie, wie in Abbildung 5.47 zu sehen, in die erste Namens-Zelle der Settings-Tabelle und geben Sie als Variablennamen **LetzterFilmname** ein. Drücken Sie ⇥.

Abbildung 5.47 Den Settings-Designer (Einstellungs-Designer) finden Sie in den Projekteigenschaften, die Sie aus dem Kontextmenü des Projekts erreichen

5. In der nächsten Spalte würden Sie im Listenfeld den gewünschten Variablentyp auswählen, was Sie in diesem Fall jedoch nicht machen müssen, da `String` für Texteingaben bereits der passende ist. Drücken Sie daher einfach ⇥.
6. Im *Bereich* wählen Sie aus, ob die Settings-Variable schreibgeschützt oder durch das Programm veränderbar sein soll. *Benutzer* wählen Sie, wenn Sie den Variableninhalt zur Laufzeit verändern wollen und der neue Inhalt nach Programmende auch gesichert werden soll; *Anwendung* wählen Sie, wenn es sich um eine Konstante handeln soll, die nur Sie zur Entwurfszeit einstellen können, die man aber zur Laufzeit nicht verändern darf. Lassen Sie auch hier die Einstellung *Benutzer* so, wie sie ist.
7. Das Feld *Wert* lassen Sie frei. Hier könnten Sie einen Standardwert einfügen, den die Settings-Variable beim ersten Start der Anwendung unter dem Benutzerkonto eines Benutzers haben würde.
8. Drücken Sie ⇥, um in die nächste Zeile zu gelangen, die vom Settings-Designer automatisch angelegt wird.

9. Auf diese Weise legen Sie weitere Variablen (alle vom Typ `String` und dem Bereich *Benutzer*) namens `LetzterSchauspieler`, `LetzteBeschreibung` und `LetzterCoverbildDateiname` an.
10. Klicken Sie auf das Symbol *Ausgewählte Elemente speichern*, um die Änderungen zu übernehmen.

Verwenden von Settings-Variablen im Code

1. Wechseln Sie nun mit [Strg]+[⇥] zum Entwurfsmodus von *Form1.vb*.
2. Klicken Sie irgendwo doppelt ins Formular – am besten unterhalb der drei Schaltflächen, damit Sie nicht versehentlich doch ein anderes Steuerelement erwischen.
3. Platzieren Sie die Einfügemarke zwischen `Private Sub Form1_Load` und `End Sub`.
4. Beginnen Sie zu schreiben:

```
txtKurzbeschreibung.Text = My.Settings.
```

5. In dem Moment, in dem Sie den Punkt hinter `My.Settings` getippt haben, wird IntelliSense aktiv und Sie bekommen, etwa wie in Abbildung 5.48 zu sehen, alle Settings-Variablen aufgelistet, die Sie gerade eingerichtet haben.

Abbildung 5.48 Den Zugriff auf die Settings-Variablen nehmen Sie über My.Settings – IntelliSense hilft Ihnen im Code-Editor anschließend beim Finden der richtigen Settings-Variablen

6. Wählen Sie für diesen Fall aus der Vervollständigungsliste `LetzteBeschreibung` aus.
7. Ergänzen Sie nach diesem Schema den Code um folgende Zeilen, sodass sich folgender Gesamtcodeblock ergibt:

```
    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) _
Handles MyBase.Load
        txtKurzbeschreibung.Text = My.Settings.LetzteBeschreibung
        txtNameDesFilms.Text = My.Settings.LetzterFilmname
        txtSchauspieler.Text = My.Settings.LetzterSchauspieler

        Dim locImage As Image = GetCoverImage(My.Settings.LetzterCoverbildDateiname)
        picCoverbild.Image = locImage
    End Sub
```

HINWEIS Sie werden übrigens feststellen, dass IntelliSense in der Vervollständigungsliste so lange, wie Sie noch keine Buchstaben zur genauen Identifizierung des zu vervollständigenden Wortes eingegeben haben, immer das Element in der Liste markiert, das Sie am häufigsten verwendet haben. Wenn Sie also beispielsweise zum dritten Mal `My` und anschließend den Punkt eingegeben haben, wird `Settings` automatisch selektiert.

Was macht `Form1_Load` nun genau? Durch den Zusatz `Handles Mybase.Load` wird bestimmt, dass `Form1_Load` dann automatisch aufgerufen wird, wenn .NET Framework das Formular darstellt – und das ist beim Programmstart der Fall. `Form1_Load` macht dann nichts weiter, als die Programmeinstellungen, die sich bereits in den `Settings`-Variablen befinden, in die Textfelder zu übertragen. Da Bilder übrigens nicht direkt in Settings-Variablen gespeichert werden können, bedienen wir uns eines Tricks: Wir speichern einfach den letzten bekannten Dateinamen und versuchen das Bild beim Programmstart einfach wieder aus der gleichen Quelle zu laden. Da unsere Coverbild-Ladefunktion Fehler dabei abfangen kann, können wir uns die Eventualität leisten, dass das Bild nicht mehr an seiner ursprünglichen Stelle zu finden ist.

Damit das Konzept aufgeht, benötigen wir nun noch den umgekehrten Fall: Wenn das Formular geschlossen wird, müssen die in den Textfeldern befindlichen Texte sowie der Bilddateiname in die Settings-Variablen übertragen werden, damit dafür gesorgt werden kann, dass die Inhalte der Settings-Variablen (und damit der Feldinhalte) beim Programmende gesichert werden.

Der beste Zeitpunkt, genau dafür zu sorgen, ist, wenn das Formular im Begriff ist, sich zu schließen. Zu diesem Zeitpunkt sind die Inhalte der Steuerelemente nämlich noch alle erhalten. Dieses Ereignis trägt den Namen `FormClosing`, und die entsprechende Ereignisbehandlungsroutine werden wir im Folgenden implementieren:

1. Wählen Sie mit Strg+⇆ das Formular *Form1.vb* im Entwurfsmodus aus.
2. Klicken Sie auf die Titelzeile des Formulars, um es zu selektieren.
3. Klicken Sie im Eigenschaftenfenster auf das Symbol *Ereignisse* (das Blitz-Symbol), um die Ereignisse anzeigen zu lassen.
4. In der Rubrik *Verhalten* finden Sie das `FormClosing`-Ereignis; auf diesen Eintrag klicken Sie nun doppelt.
5. Der Editor hat nun die Stub für das `FormClosing`-Ereignis eingefügt, das Sie nur noch um die entsprechenden Codezeilen zu ergänzen brauchen:

```
Private Sub Form1_FormClosing(ByVal sender As System.Object, _
ByVal e As System.Windows.Forms.FormClosingEventArgs) Handles MyBase.FormClosing
    My.Settings.LetzteBeschreibung = txtKurzbeschreibung.Text
    My.Settings.LetzterFilmname = txtNameDesFilms.Text
    My.Settings.LetzterSchauspieler = txtSchauspieler.Text
    My.Settings.LetzterCoverbildDateiname = myBilddateiname
End Sub
```

Verknüpfen von Settings-Werten mit Formular- oder Steuerelementeigenschaften

Settings-Werte können übrigens auch dazu verwendet werden, Eigenschaften von Formularen und/oder deren Steuerelementen zu speichern. Dabei werden die entsprechenden Eigenschaftenwerte sogar direkt an die Settings-Werte gebunden. Und was haben Sie davon?

Ein Beispiel: Angenommen Sie möchten, dass zur Laufzeit Ihres Programms beim Öffnen eines Formulars dieses automatisch an der letzten Position dargestellt wird. In diesem Fall könnten Sie Settings-Werte zur Speicherung der `Location`-Eigenschaft verwenden, die die Position des Formulars bestimmt. Sie müssten innerhalb des `Load`-Ereignishandlers des Formulars die Werte für die entsprechende Eigenschaft aus den Settings lesen und sie umgekehrt beim Schließen des Formulars in `FormClosing` wieder in die Settings übernehmen.

Doch diesen Vorgang können Sie auch automatisieren – Sie binden die entsprechende Eigenschaft einfach an einen Settings-Wert, und das funktioniert folgendermaßen:

1. Öffnen Sie `Form1` durch Doppelklick auf die entsprechende Klassendatei im Projektmappen-Explorer.
2. Klicken Sie auf die Titelleiste des Formulars, um das Formular selbst zu selektieren.
3. Suchen Sie im Eigenschaftenfenster (auf das Zurückschalten auf die Eigenschaftenliste achten!) nach dem Eintrag *(ApplicationSettings)*, und öffnen Sie den Zweig durch Klick auf das davor stehende Dreiecksymbol.
4. Klicken Sie auf den Eintrag *Location* und öffnen Sie das Listenfeld, wie in Abbildung 5.49 zu sehen.

Abbildung 5.49 An dieser Stelle steuern Sie das Binden von Formulareigenschaften an die Anwendungseinstellungen (Application Settings)

5. In der Liste, die sich nun öffnet, klicken Sie auf *(Neu)*.
6. Visual Studio öffnet nun einen weiteren Dialog, in dem Sie, ohne in die Anwendungseinstellungstabelle wechseln zu müssen, direkt einen neuen Settings-Wert mit dem für die Eigenschaft automatisch richtigen Typ einrichten können. Geben Sie dazu zunächst eine Standardposition für das Formular unter `DefaultValue` ein (denken Sie daran, die beiden Zahlen mit einem Semikolon und nicht mit einem Komma zu trennen!), und bestimmen Sie ferner den Namen für den Settings-Wert – beispielsweise `HauptformularPosition`.

Abbildung 5.50 An dieser Stelle steuern Sie das Binden von Formulareigenschaften an die Anwendungseinstellungen (Application Settings)

7. Beenden Sie den Dialog mit *OK*.

Wenn Sie das Programm nun starten, öffnet sich das Hauptformular des Programms beim ersten Mal an Position 200; 200. Verschieben Sie daraufhin das Formular und schließen es, dann wird es beim nächsten Start automatisch wieder an der Stelle erscheinen, an der es sich beim letzten Beenden befand.

HINWEIS Die Größe des Formulars können Sie leider nicht auf diese Art und Weise an Settings-Werte binden. Die `Size`-Eigenschaft ist aus .NET Framework-internen Gründen nämlich nicht an die Anwendungseinstellungen bindbar. Zwar ließe sich die Formulargröße auch indirekt durch die `ClientSize`-Eigenschaft an einen Settings-Wert binden, doch wenn Sie das machen, werden bei der Wiederherstellung der Größe die `Anchor`-Einstellungen der anderen Steuerelemente nicht korrekt berücksichtigt. In diesem Fall haben Sie also nur die Möglichkeit, wie in einem der vorherigen Absätze schon beschrieben, die Eigenschaftenzuweisung in den Ereignisbehandlungsroutinen `FormLoad` und `FormClosing` manuell zu durchzuführen, wobei Sie für die Größe des Formulars auf die eben erwähnte `Size`-Eigenschaft zurückgreifen.

Und wo werden die Settings-Daten abgelegt?

Im persönlichen Anwendungsdatenverzeichnis unter *Lokale Einstellungen* auf dem Windows-Installationslaufwerk in weiteren Unterordnern in Abhängigkeit von Ihrem Benutzernamen sowie dem in den Assembly-Infos hinterlegten Firmennamen und dort in weiteren Unterverzeichnissen, die sich aus Laufzeitumgebung (Debug- oder Nicht-Debug-Modus), dem Anwendungsnamen und der Anwendungsversion ergeben. Alles klar?

Oder besser, da verständlicher: In meinem Fall lautet das Verzeichnis:

```
C:\Dokumente und Einstellungen\loeffel.ACTIVEDEVELOP\Lokale
Einstellungen\Anwendungsdaten\AndereFirma\Covers.vshost.exe_Url_ohvclojvckpztoiykwlhs3ba2acq2v4i\1.0.0.0
```

Warum?

- Mein Anmeldename in unserem Active Directory-Netzwerk lautet *loeffel*. Unsere Domäne heißt *ActiveDevelop* (mein Geburtsdatum ist übrigens der 24.07.1969, aber Sie werden sich mit diesem Wissen dennoch nicht bei uns anmelden können …).
- Als Firmennamen habe ich im Assembly-Infodialog *Andere Firma* eingegeben. Diesen Dialog erreichen Sie, indem Sie aus dem Kontextmenü des Projekts im Projektmappen-Explorer *Eigenschaften* aufrufen. Wählen Sie das Register *Anwendung* (das ist die vorgewählte Eigenschaftenseite), und klicken Sie auf die Schaltfläche *Assemblyinformationen*. Das vorvorletzte Verzeichnis im Pfad entsteht aus dem Namen, den Sie unter *Firma* bestimmt haben.
- Das nächste Verzeichnis resultiert aus dem Programmnamen. Es entsteht aus dem Laufzeitprogrammnamen, der variieren kann. Wenn Sie das Programm im Debug-Modus starten, ist es nämlich nicht das Programm selbst, das gestartet wird, sondern ein Host-Prozess, der dafür sorgt, dass Sie den Programmcode auch während des Debuggens verändern können[13] und dass die Geschwindigkeit beim Debuggen einigermaßen erträglich bleibt. Dieser Hostprozess hat als Anwendungsnamen eine Kombination aus dem eigentlichen Anwendungsnamen (*Covers*) und *.vshost.exe*. Diesem Block wird die Zeichenkette *Url_* sowie eine weitere Kennung angehängt – deren genaue Bedeutung beim Entstehen dieser Zeilen noch nicht zu ermitteln war.

[13] Dabei handelt es sich übrigens um das viel diskutierte Edit & Continue-Feature, das es seit Visual Basic 2002 nicht mehr und erst ab Visual Basic 2008 wieder gibt.

- Schließlich folgt die Versionsnummer des Programms als weiteres Unterverzeichnis, in dem sich schließlich die *user.config*-Datei befindet – eine XML-Datei, die die eigentlichen Einstellungen speichert, wie im folgenden Beispiellisting zu sehen:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <userSettings>
        <Covers.My.MySettings>
            <setting name="LetzterFilmname" serializeAs="String">
                <value>Terminator III - Rise of the Machines</value>
            </setting>
            <setting name="LetzterSchauspieler" serializeAs="String">
                <value>Arnold Schwarzenegger, Kristanna Loken, Nick Stahl, Claire Danes</value>
            </setting>
            <setting name="LetzteBeschreibung" serializeAs="String">
                <value>Kristanna versucht einen Kranwagen einzuparken, was nicht wirklich klappt.
                    Arnie
                       hilft nach, legt ihn aber dann aufs Dach.</value>
            </setting>
            <setting name="LetzterCoverbildDateiname" serializeAs="String">
                <value>C:\Dokumente und Einstellungen\All Users\Dokumente\
                       Eigene Bilder\Beispielbilder\Arnie.jpg</value>
            </setting>
            <setting name="PrintFormPosition" serializeAs="String">
                <value>303, 123</value>
            </setting>
        </Covers.My.MySettings>
    </userSettings>
</configuration>
```

HINWEIS Benutzerdaten und Anwendungsdaten, die aus Settings-Einstellungen hervorgehen, werden übrigens nicht in den gleichen Konfigurationsdateien gespeichert. Wenn Sie im Settings-Designer unter *Bereich* den Eintrag *Anwendung* gewählt und damit eine Nur-Lesen-Settings-Eigenschaft bestimmt haben, die für alle Benutzer gilt, werden diese Einstellungen in der Datei *appname.exe.config* gespeichert – wobei *appname* dem Namen der Anwendung Ihres Projekts entspricht. Diese Datei befindet sich wiederum im *bin*-Unterverzeichnis des Verzeichnisses, das dem Namen der Konfiguration für die Erstellung des Projekts entspricht. Standardmäßig gibt es die Konfigurationseinstellungen *Debug* und *Release* – die sich in ihren Parametern zunächst nicht unterscheiden, außer, dass das Kompilat durch die unterschiedlichen Konfigurationsnamen auch in unterschiedlichen Unterverzeichnissen gespeichert wird.

Haben Sie also beispielsweise Ihr Projekt im Hauptverzeichnis von Laufwerk »D:« unter dem Namen *Covers* erstellt, finden Sie die ausführbaren Dateien (und auch die *appname.exe.config*) im Verzeichnis *d:\covers*. Mehr zu den Konfigurationseinstellungen finden Sie im anschließenden grauen Kasten.

Über die Konfigurationseinstellungen »Debug« und »Release« sowie die Geschwindigkeiten der Codeausführung

Gerade Entwickler, die ihre ersten Gehversuche mit der Visual Studio-IDE absolvieren, neigen anfangs dazu, das Konfigurationsmanagement von Projekt-Compilereinstellungen misszuverstehen. Sie stellen oft fest, dass die Geschwindigkeit der Codeausführung in der *Debug*-Konfigurationseinstellung wohl nicht der echten entsprechen kann, und sie sind dann enttäuscht, wenn auch das Umstellen auf *Release* keine Geschwindigkeitsrekorde erzielt. ▶

Aber das kann es auch gar nicht. Der Geschwindigkeitsunterschied an dieser Stelle rührt nämlich daher, dass der Debugger von Visual Studio beim Starten der Anwendung an den Prozess angehängt wird, was ihn spürbar verlangsamt. Die vordefinierten Konfigurationseinstellungen *Debug* und *Release* steuern hingegen den Optimierungsgrad des fertigen Kompilats. Im Release-Mode werden die Debug-Symbole nicht mitgeneriert, und erst das ist die Voraussetzung, den ausführbaren Code geschwindigkeitsmäßig optimieren zu können. Sie können aber in beiden Versionen den Debugger anhängen, was die Ausführung allerdings immer so verlangsamt, dass Sie die Optimierungen gar nicht mehr spüren.

Wenn Sie wissen wollen, wie schnell Ihr Programm später beim Kunden wirklich zu laufen in der Lage ist, dann starten Sie es über das Menü *Debuggen* und den Befehl *Starten ohne Debuggen* – oder drücken Sie einfach die Tastenkombination Strg + F5.

Mit den Konfigurationseinstellungen, legen Sie hingegen beispielsweise fest, auf welche Plattformen das Kompilat abzielen soll (x86, egal welcher Prozessor – »Any« etc.), welche Projekte innerhalb einer Projektmappe kompiliert werden sollen oder nicht und Ähnliches. Und im Übrigen auch, in welchen Unterverzeichnissen des Projekts die ausführbaren Dateien generiert werden.

WICHTIG Sobald Sie ein Programm im Debug-Modus (also mit F5) starten, wird es immer langsamer laufen, als wenn Sie es ohne zu debuggen starten (mit Strg + F5).

Herzlichen Glückwunsch!

Sie haben soeben Ihr erstes funktionsfähiges (und wie ich finde auch recht brauchbares) Windows .NET Framework 4.0-Programm fertig gestellt. Und nun: Viel Spaß beim Arbeiten mit Covers. Testen Sie es! Produzieren Sie damit die Hüllen Ihrer Urlaubsvideos. Drucken Sie, bis der Drucker qualmt!

Kapitel 6

Einführung in Windows Presentation Foundation

In diesem Kapitel:

Was ist Windows Presentation Foundation? 190
Was ist so neu an WPF? 191
Wie WPF Designer und Entwickler zusammenbringt 203
Die XAML-Syntax im Überblick 213
ImageResizer – das praktische WPF-Beispiel 216

Das Auge isst mit. Diesen Spruch habe ich mir von meiner Mutter schon anhören müssen, als ich als kleiner Junge den Tisch decken musste – na ja, sagen wir: entsprechende Versuche veranstaltet habe, diesen ihren ästhetischen Bedürfnissen entsprechend zu decken. Mein eigener Grundsatz »Form follows function«, in leicht abgeänderter Version im Deutschen eher als »Funktion vor Design« bekannt, widersprach diesem Grundsatz meiner Ma und war – ich gebe es offen zu – sicherlich auch Beweggrund, mich seit Visual Basic 4.0 eher auf Windows Forms-Anwendungen als auf Webanwendungen zu spezialisieren. Und mich im Übrigen auch nicht als Innenarchitekt zu versuchen. Viele Webanwendungen lassen sich sicherlich grafisch und designtechnisch ansprechend, auf alle Fälle aber sehr viel ansprechender als Windows Forms-Anwendungen designen. Leider aber nur von den richtigen Leuten, zu denen ich zugegeben leider nicht gehöre.

Was ist Windows Presentation Foundation?

Und dann brachte Microsoft mit .NET Framework 3.0 das Grafikframework *Windows Presentation Foundation* heraus. Erklärtes Ziel war es eigentlich, und daraus leitet sich auch der Name ab, eine Technik zu schaffen, mit der Informationen in grafisch vollendeter Form vor allem präsentiert werden können. Aber nicht nur. WPF, wie Windows Presentation Foundation gerne abgekürzt genannt wird, sollte auch dazu dienen, diese Grafik als Benutzerschnittstelle zu nutzen, also um mit dem Benutzer zu interagieren – mit Maus, mit Stift oder, wie es mit Windows 7, dem Nachfolger von Windows Vista der Fall ist, auch mit fingerberührungsempfindlichen Displays auf Notebooks[1] oder speziellen Tischdisplays.

Und das bedeutet für uns Windows Forms-Programmierer: umdenken, und zwar *heftigst*, wie wir Westfalen sagen. Der deutsche Spruch passt nicht mehr, denn nunmehr gilt: *Funktion ist Design*. Mit dem englischen Idiom sind wir immer noch gut aufgehoben, denn es ändert sich nichts an »Form follows function«.

WPF unterlag in den letzten zwei Jahren einem kleinen, von keinem und dann wieder von allen vorangetriebenen Evolutionsprozess. Denn obwohl Microsoft WPF ursprünglich niemals als designierten Nachfolger von Windows Forms konzipiert hatte, zeichnet es sich immer mehr ab, dass genau das passiert: Viele der immer noch umfangreichen Visual Basic 6.0- und C++-COM-Anwendungen müssen auf eine neue Plattform migriert werden, da viele dieser Anwendungen unter Vista, Windows 7 und Windows Server 2008 erhebliche Probleme bereiten und unter den immer häufiger anzutreffenden 64-Bit-Versionen von Windows überhaupt nicht mehr funktionieren. Die einzige Lösung dazu lautet: Die alten Anwendungen müssen auf bzw. zu .NET migriert werden; Visual Basic 6.0 und VBA vorzugsweise auf .NET-Versionen von Visual Basic. Und während dieses Migrationsprozesses, während also die Konzeptionen für die beste Vorgehensweise einer solchen Umstellung von den Tausenden verschiedenen Entwicklerteams, die es da draußen gibt, ausgearbeitet wird, stellt sich diesen erfahrungsgemäß immer wieder die gleiche Frage: Wie sollen die neuen Frontends und Benutzeroberflächen gestaltet werden? In diesem Prozess kommen immer mehr Teams zu dem Schluss, dass sich WPF *durchaus* auch als UI-Ersatz[2] für Windows Forms eignet. Die langfristigen Benefits, die sich daraus ergeben, scheint man als wichtiger zu erachten als die Nachteile und den erheblichen Zusatzaufwand, den man vor allen Dingen zurzeit noch bei der Ablösung von Windows Forms durch WPF hinnehmen muss. Und dieser Zeitaufwand liegt vor allem an folgendem:

---

[1] Demo gefällig? YouTube hilft, und *http://tinyurl.com/6gaoln* zeigt wo.

[2] UI: Abkürzung für User Interface, zu Deutsch *Benutzeroberfläche*.

- Der Einarbeitungsaufwand in die Technik von WPF ist ganz erheblich. Das gilt vor allen Dingen für Teams, die aus der Visual Basic 6.0-Welt kommen und die sich ohnehin erst an das objektorientierte Programmierkonzept gewöhnen müssen. Und aufgepasst: Die Rede ist dabei von objekt*orientiert* – viele Entwickler, die selbst wirklich alles aus Visual Basic 6.0 herausgeholt haben und dessen Tücken und Fallen aus dem Effeff kennen, haben dennoch – konzeptbedingt – bestenfalls objekt*basiert* programmieren können. Das ist ein großer Unterschied, gerade was den bestmöglichen Aufbau einer Anwendung anbelangt, der in .NET-Versionen von Visual Basic – objektorientiert – zu einem komplett anderen Ansatz führen kann, bei dem im Vergleich zur Ausgangsanwendung kein Stein auf dem anderen verbleibt.
- Der Aufbau von Formularen, oder das, was dem am nächsten kommt, unterliegt einem ganz anderen Konzept als in Windows Forms. Es gibt hier – ähnlich wie bei HTML zur Beschreibung von Webseiten – eine an XML angelehnte Sprache namens XAML (sprich: gsämmel), die dafür zuständig ist. Hier ist viel Lernaufwand erforderlich, insbesondere dann, wenn einem schon das Konzept von Dialekten wie HTML, die den Seitenaufbau beschreiben, eher fremd ist.
- Windows Forms verwendet einen Designer, der altbewährt ist. Selbst der .NET-Designer für Windows Forms-Anwendungen hat bereits fünf Jahre auf dem Buckel, und schon der allererste Designer dieser .NET-Version basierte konzeptionell weitestgehend auf dem, was zuletzt in Visual Basic 6.0 zu finden war. Der WPF-Designer hat – wohlwollend ausgedrückt – selbst in .NET 4.0 noch einiges an Verbesserungspotenzial, und so sollten Sie sich darauf einrichten, dass Sie viele Dinge, die Sie unter Windows Forms natürlich interaktiv mit Designerunterstützung machen würden, für WPF direkt in XAML eintippen müssen. Während Sie die Ergebnisse Ihrer Arbeit anfangs oft mit positivem Erstaunen kommentieren werden, sollten Sie sich allerdings nicht darüber wundern, dass Ihnen auf dem Weg dahin sicherlich der eine oder andere Fluch entgleiten wird.

HINWEIS Wenn Sie professionelle WPF-Anwendungen entwickeln wollen, werden Sie um den Einsatz einer speziell auf WPF abgestimmten Designersoftware nicht herumkommen. Ihr Name: Expression Blend – und es gibt sie inzwischen in der vierten Version. Der unverständliche Nachteil: Expression Blend ist NICHT in Visual Studio 2010 enthalten – Sie müssen es wohl oder übel dazukaufen. Spätestens hier bietet sich der Kauf eines so genannten MSDN-Abos an: Schon in der MSDN Professional Edition dieses Abonnements sind sowohl Visual Studio 2010 Professional als auch Expression Blend 4 enthalten. Und natürlich erhalten Sie dazu alle Betriebssysteme, Server und die gängigsten Anwendungsprogramme aus dem Hause Microsoft für Entwicklungszwecke obendrein. Mehr zum Thema MSDN erfahren Sie unter *http://tinyurl.com/3a9dkm7*.

Was ist so neu an WPF?

Ein guter Freund sagte mir einst, er hasse an vielen Computerbüchern, dass sie bestimmte Sachverhalte nicht einfach Punkt für Punkt an simplen, einfachen Beispielen erklären. Und er bezog sich damit indirekt auch auf die Rohfassung des Textes, den Sie gerade lesen. An dieser Stelle, so seine Meinung, seien schon zwei Seiten ins Land gegangen, und er wisse immer noch nicht, wie man eine einfache Linie mit WPF auf dem Bildschirm malt – das könne doch nicht so schwer sein. Und da hat er vielleicht irgendwie Recht, gleichzeitig aber auch nicht verstanden, was WPF ist. Denn seine Frage impliziert schon, dass er meint, dass die Rendering[3]-Engine von WPF so funktionieren würde wie das, was er kenne, sodass beispielsweise aus einer Linie ein Rechteck und aus einem Rechteck eine Schaltfläche entstünde. So war es aber nicht gedacht.

[3] Lustigerweise gibt es keine wirklich passende Übersetzung, die das Konzept des Renderings des »Entstehenlassens oder des Umsetzens« so ganz passend wiedergibt. Was gemeint ist: *http://tinyurl.com/34omj3v*

Dummerweise ist es nämlich völlig anders. Wenn Sie aber verstehen wollen, wie WPF funktioniert, sollten Sie etwas über die Entstehungsgeschichte von WPF wissen. Warum? Ganz einfach und als Metapher erzählt: Sonst geht es Ihnen wie diesem Zeitgenossen, der bislang nur das Waschbrett kannte. Und der sich beim erstmaligen Anblick der Waschmaschine fragt, wie man so etwas Unergonomisches bauen könne wie eine Waschmaschine, bei der die Waschreibe auf der Innenseite einer Trommel angeordnet sei und man zum Waschen der Wäsche ja in diese hineinkriechen müsse.

Das einfache Beispiel, das der vorhin erwähnte Freund verlangt, kann er bekommen. Doch nicht ohne den angekündigten Exkurs in Sachen *Geschichte von Windows* und wie dessen grafische Elemente bislang den Weg auf den Bildschirm fanden.

25 Jahre Windows, 25 Jahre gemalte Schaltflächen

Windows-Anwendungen sind mittlerweile seit mehr als 25 Jahren zugegen. Und sind wir ehrlich: Auch wenn es Windows 1.0 und 2.0[4] schon *gab*, wurde erst Windows 2.11 als Bundle-Version mit Adobe Pagemaker etwas bekannter. Und erst mit Windows 3.0 kam der Mainstream-Durchbruch dank neuem Look & Feel bei der Bedienung von Anwendungen, dessen grundsätzliches Konzept sich später allerdings selbst mit Windows 95 und Windows NT sowie über Windows 2000 und Windows XP hinweg im Wesentlichen kaum änderte.

Abbildung 6.1 Der Dialog zum Hinzufügen von Schriftarten in einem englischen Windows 3.0 ...

Sicherlich: Windows ist vor allen Dingen, was die auf Windows NT basierenden Versionen anbelangt, im Lauf der Zeit deutlich stabiler geworden; Windows 95 brachte genau wie Windows NT 4.0 viele neue grafische Elemente, die mit Windows XP nochmals bunter, moderner und mit einer Menge an Farbverläufen gekrönt wurden. Doch eines hat sich an und für sich seit der ersten Version von Windows nicht geändert: die Art und Weise, wie die Grafik ihren Weg auf die Mattscheibe findet.

[4] Mehr dazu liefert *http://www.winhistory.de/more/win1.htm.*

Abbildung 6.2 ... und in Windows Vista. Hier ist deutlich zu sehen, dass sich über all die Jahre wirklich nicht viel am Prinzip der Windows-Bedienung und des Renderings von Windows Forms-Dialogen geändert hat

Das geschieht nämlich im Grunde genommen bei 99% aller Programme mithilfe von GDI (*Graphics Device Interface*) von Windows. Diese API[5] beinhaltet Funktionen, mit der zweidimensionale Zeichenfunktionen (das Malen von Linien, Rechtecken, Kreisen und verschiedenen Pinselstärken, Formen und Mustern) möglich sind, die durch Grafikkarten unterstützt werden. *Unterstützung von Grafikkarten* bedeutet dabei, dass die Treiber der Grafikkarte bestimmte Befehle beispielsweise zum Zeichnen einer Linie nicht selbst durch Algorithmen und mithilfe des Prozessors umsetzen, sondern sie vielmehr an die Grafikkarte weiterreichen können, damit die Grafikkarte diese Linie selbst malt – natürlich viel, viel schneller, als es der Prozessor Ihres Computers jemals könnte. Und wie das in der Praxis ausschaute, daran soll uns das folgende kleine Beispiel noch einmal erinnern:

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2008 Entwicklerbuch\Kapitel 06\SimpleSampleGDIPlus
```

Öffnen Sie dort die Projektmappe (.SLN-Datei) *SimpleSampleGDIPlus*.

[5] *API* bedeutet *Application Programming Interface*, in etwa Anwendungsschnittstelle zum Programmieren.

Abbildung 6.3 Das Ergebnis nach einem Doppelklick ins Fenster – eine Reihe von aufgefächerten Linien zeigt eindrucksvoll den bei Fernsehmachern so gefürchteten Moiré[6]-Effekt

Die Figur, die dort gezeichnet wird (Abbildung 6.3), interessiert uns allerdings nur am Rande. Vielmehr soll Augenmerk darauf gelegt werden, *wie* die Linien auf den Bildschirm kommen und wie Windows dafür sorgt, dass sie dort verbleiben. Und dazu schauen wir uns erst einmal das kleine, zur Abbildung gehörige Listing an, das die Grafik beim Doppelklick ins Fenster bringt:

```
Public Class Form1

    Private mydoppelGeklickt As Boolean

    'OnLoad überschreiben - damit das Load-Ereignis behandeln, ...
    Protected Overrides Sub OnLoad(ByVal e As System.EventArgs)
        MyBase.OnLoad(e)
        '...und hier den Fenstertext setzen.
        Me.Text = "Doppelklicken Sie in das Fenster, um die Grafik darzustellen."
    End Sub

    'Wir hätten auch das Load-Ereignis behandeln können...
    Private Sub Form1_Load(ByVal sender As Object, ByVal e As System.EventArgs) _
                Handles Me.Load
        'aber warum ein Ereignis der Instanz in der Instanz behandeln,
        'die das Ereignis auslöst? Dann lieber an der Stelle eingreifen,
        'die das Ereignis auslösen, und das ist OnLoad.
    End Sub
    'Grafik wird ins Fenster gemalt, sobald der Anwender doppelklickt.
    Protected Overrides Sub OnDoubleClick(ByVal e As System.EventArgs)
        MyBase.OnDoubleClick(e)
        Dim g As Graphics = Graphics.FromHwnd(Me.Handle)
```

[6] Sprich: Moareh – mehr dazu zeigt *http://tinyurl.com/bt6zjf*.

```
        DrawDemo(g)
        mydoppelGeklickt = True
    End Sub

    'Und hier wird gemalt.
    Public Sub DrawDemo(ByVal g As Graphics)

        'Erstmal den Fensterinhalt löschen.
        g.Clear(Color.White)

        'tatsächliche Breite des Clientbereichs ermitteln und merken
        Dim tatsächlicheBreiteGemerkt = Me.ClientSize.Width
        Dim linienZähler = 0

        'Jeden 5. Pixel berücksichtigen
        For x = 0 To tatsächlicheBreiteGemerkt Step 5

            'Erst die Linie von links oben nach rechts unten malen.
            g.DrawLine(Pens.Black, 0, 0, x, Me.ClientSize.Height)

            'Und dann noch eine von rechts oben nach links unten.
            g.DrawLine(Pens.Black, tatsächlicheBreiteGemerkt, _
                               0, tatsächlicheBreiteGemerkt - x, _
                               Me.ClientSize.Height)

            'Linienzähler aktualisieren
            linienZähler += 2
        Next

        'Im Fenstertitel über die Anzahl gemalter Linien informieren.
        Me.Text = "Anzahl Linien im Fenster: " & linienZähler
    End Sub
End Class
```

Zum Malen von Inhalten in GDI und GDI+ (zu GDI+ etwas später mehr) ist dabei Folgendes zu sagen: Diese sind höchst volatil. Wenn Sie nämlich nicht selbst dafür sorgen, dass der Fensterinhalt, nachdem er zerstört wurde, weil etwas anderes über das Fenster geschoben wurde oder jemand das Fenster vergrößert oder verkleinert hat, restauriert wird, dann ist der Inhalt eben weg, wie Abbildung 6.4 anschaulich zeigt. Hier wurde das Fenster zunächst ein paar Pixel horizontal und vertikal vergrößert, und anschließend hat sich auch noch der der Windows-Taschenrechner über das Fenster geschoben.

Abbildung 6.4 Um das Aktualisieren des Fensterinhalts muss sich der Entwickler unter GDI oder GDI+ selbst kümmern. Tut er es nicht, gibt es beim Vergrößern oder beim Verdecken des Fensters Effekte, wie hier in der Abbildung zu sehen.

HINWEIS Ab Windows Vista müssen Sie die Aero-Darstellung ausschalten, um das ursprüngliche Verhalten von GDI nachvollziehen zu können. Unter Windows Vista und Windows 7 bekommt jede Anwendung bei Aero ohnehin einen dedizierten Videospeicherbereich zugewiesen, sodass sich Fenster aus Anwendungssicht gar nicht mehr überlappen können, und Sie dieses Verhalten auch gar nicht bemerken. Deswegen übrigens »wirkt« Vista auch soviel langsamer als Windows 7: Ganz vereinfacht ausgedrückt, sind Zugriff auf GDI-Funktionen unter Windows 7 besser optimiert und können sich nicht gegenseitig so sehr beeinflussen. Darüber hinaus sind Vista-GDI-Operationen nicht hardwareunterstützt, ab Windows 7 ist das wieder der Fall. Bei Windows Vista können sich Anwendungen beim Zugriff auf das GDI obendrein schon mal gegenseitig behindern, und wirken daher nicht so flüssig wie unter Windows 7. Die reine Performance von Anwendungen ist aber unter Windows Vista oftmals sogar schneller als unter Windows 7; die Bildschirmdarstellung von GDI-Operationen ist unter Windows 7 hingegen schneller, da hardwareunterstützt.

Damit solche Probleme wie in Abbildung 6.4 nicht passieren können, muss der Entwickler selbst Hand anlegen: Im `Paint`-Ereignis, das dann ausgelöst wird, wenn sich irgendetwas vor das Fenster legt und das Neuzeichnen des Inhalts beim Wiederhervorholen des Fensters erforderlich macht, sorgt er dafür, dass die Anwendung darüber informiert wird, dass der Fensterinhalt aktualisiert werden muss. Ein Ergänzen der folgenden Zeilen würde in unserem Beispiel also bereits für Abhilfe schaffen.

```
'Wird ausgelöst, wenn der Fensterinhalt zerstört wurde
 'und neugezeichnet werden muss
 Protected Overrides Sub OnPaint(ByVal e As System.Windows.Forms.PaintEventArgs)
     MyBase.OnPaint(e)

     'Nur wenn der Doppelklick bereits stattgefunden hat.
     If mydoppelGeklickt Then
         '
         DrawDemo(e.Graphics)
     End If
 End Sub
```

Gegebenenfalls muss er auch ein entsprechendes `Resize`-Ereignis behandeln, das ausgelöst wird, wenn sich die Fenstergröße durch Verkleinern oder Vergrößern des Fensters ändert.

Und dieses Prinzip des Darstellens von Dokumenteninhalten gilt für Windows Forms seit der ersten Windows-Version bis heute. Und nicht nur für Grafiken, die in Fenstern dargestellt werden, sondern für alles. Und dann wiederum doch nur für Grafiken, die in Fenstern dargestellt werden, denn vielen Leuten ist es überhaupt nicht bewusst, wieso Windows Windows heißt. Warum? Weil nicht nur Dokumenteninhalte jeglicher Form in Fenstern, die auch als solche zu erkennen sind, dargestellt werden. Vielmehr ist nämlich jedes Element in Windows – mit wenigen Ausnahmen – ein Fenster. Eine Schaltfläche ist ein Fenster. Eine ListBox ist ein Fenster. Ein geöffnetes Kombinationslistenfeld sind zwei Fenster – die TextBox ist eines und die Liste darunter ein weiteres. Streng genommen ist auch ein ToolTip oder eine Sprechblase des Systray (Infobereich) ein Fenster, und alle diese Windows-Klassen müssen dafür sorgen, dass sie ihren Inhalt neu malen, in dem Moment, wo dieser durch Überlappung mit einem anderen Fenster zerstört wurde. Die Grafikkarte hilft dabei hardwaremäßig nur wenig:

- Sie kopiert zwar Bildblöcke in rasender Geschwindigkeit.
- Sie zeichnet auch Grundlinienfiguren wie Linien, Rechtecke, Ellipsen und füllt diese mit Inhalten. Das macht sie bis heute aber leider nur, wenn die nativen Zeichenbefehle des Betriebssystems, genau gesagt von GDI, verwendet werden. Die .NET-Grafikbefehle werden von einer moderneren Grafikbibliothek umgesetzt: GDI+. Diese lässt leider auf vielen Grafikkarten die entsprechende Hardwareunterstützung vermissen, sodass sie zwar die deutlich komfortableren Grafikoperationen anbietet, diese sind aber *deutlich* langsamer als ihre GDI-Pendents.

HINWEIS Zwei Ausnahme betreffen einerseits das Rendern von Text unter .NET. Hier gibt es den `TextRenderer`, der tatsächlich als einziges direktes Grafikobjekt von GDI Gebrauch macht. Zum Zeichnen bestimmter UI-Elemente gibt es darüber hinaus auch die `ControlPaint`-Klasse im Namespace `System.Windows.Forms`, deren statische Methoden teilweise auch von GDI Gebrauch machen (zum Beispiel `DrawReversibleFrame` zum invertierten Zeichnen eines Rahmens oder `DrawReversibleLine` zum invertierten Zeichnen einer Linie).

Spielende Protagonisten

Grafikkarten können allerdings schon seit etlichen Jahren erheblich mehr als nur diese »alten Kamellen« in Sachen Grafikrendering. Das so genannte DirectX gibt es bereits seit Windows 95, und die Älteren unter uns erinnern sich sicherlich noch an den ersten Egoshooter *Wolfenstein 3D*, der zwar einen sehr umstrittenen Spielgegenstand lieferte, dessen Entwickler der Firma id Software aber nichtsdestotrotz demonstrierten, was eigentlich aus einem Bürocomputer der frühen 1990er-Jahre in Sachen Spiele und damit natürlich in Sachen

Grafik und Sound herauszuholen war. Das grundsätzliche Problem der damaligen Zeit war allerdings, dass für diese enorm hohe Performance eine Anwendung wirklich alleine und ohne störende »Mitbewerber« laufen musste. Und das war, obwohl Windows 3.11 schon als Quasi-Multitasking-Betriebssystem existierte, eben nur unter MS-DOS möglich. DirectX erst lieferte die nötigen Voraussetzungen, damit das – inzwischen war Windows 95 auf dem Markt – auch innerhalb eines Multitasking-Betriebssystems möglich wurde. Und auch hier war es wieder die Firma id Software, die letzten Endes – und auch erst mit der Version 3.0 von DirectX – durch die Portierung des Computerspiels Doom auf Windows 95 zum Geburtshelfer von DirectX wurde. Denn erst mit dieser Version gab es genügend Spieleentwickler, die DirectX als echte Multimedia-Alternative zum Stand-alone-Konzept von MS-DOS wahrnahmen, und DirectX gewann die nötige Akzeptanz.

Und gerade die Gamer sahen von diesem Zeitpunkt an atemberaubende Benutzeroberflächen für ihre heiß geliebten Spiele, die nur durch dreidimensionale, mit virtueller Realität ausgestattete KI-Systeme in Sciencefiction-Filmen übertroffen wurden. Mal ehrlich: Für welchen Windows-Benutzer der damaligen Zeit war der Film *Enthüllung* mit Michael Douglas und Demi Moore kein benutzeroberflächentechnischer Hochgenuss – wenn auch mit einem leicht bitteren Beigeschmack, nämlich zu wissen, dass man sich 45 Minuten später wieder am seit Jahren bekannten und deshalb recht eintönigen User Interface von Windows NT 4.0 SP4a der realen Welt einfinden musste.

Und genau dieses Problem sollte uns Windows-Benutzern dann noch die nächsten ca. zehn Jahre erhalten bleiben, sodass sich viele Benutzer und sicherlich nicht nur die Softwareentwickler unter uns sehnlichst wünschten, wenigstens eine kleine Bewegung in Richtung Spiele-UIs erleben zu dürfen. Doch wir sollten von Windows 2000, dessen zahlreichen Service Packs und auch von Windows XP und dessen leider weniger zahlreichen Service Packs enttäuscht bleiben. Doch dann kamen Windows Vista und .NET Framework 3.0.

Mit Windows Vista gab es nämlich auch endlich ein neues Grafiksystem, das die Darstellung von bestimmten Dingen über eine Funktionsbibliothek auf der Basis von DirectX erlaubte – genau das war Windows Presentation Foundation, kurz WPF genannt. WPF-basierende Anwendungen laufen am flüssigsten und am – wenn ich dieses Wort einmal kreieren darf – *hardwareunterstütztesten* ab Windows Vista. Mit einer besonderen Version kommen aber auch Windows XP-Benutzer in den Genuss, WPF-basierende Anwendungen einzusetzen, aber eben nicht in dieser hochwertigen Darstellungsvollendung wie der von Windows Vista.

Zum Zeitpunkt, zu dem diese Zeilen entstehen, ist die Multimedia-Entwicklergemeinde, was DirectX anbelangt, inzwischen bei der Version 11 angelangt. Diese beinhaltet die neusten und aufwändigsten Multimedia-Funktionsbibliotheken, die es jemals unter einer Windows Version gab. Und wer die Entwicklung von Windows Vista mitbekommen hat, der weiß nicht nur, dass DirectX seit Version 10 ausschließlich Vista-Benutzern und -Entwicklern vorbehalten ist, sondern auch, was es für ein unsäglicher Akt war, endlich in den Genuss stabiler NVIDIA- und ATI-Grafikkartentreiber für die volle DirectX 10-Unterstützung zu gelangen.

Theoretisch konnten wir Entwickler mit WPF, das – der Vollständigkeit halber erwähnt – übrigens schon auf Basis von DirectX 9 funktioniert, seit .NET Framework 3.0 bzw. Windows Vista auch anspruchsvolle Anwendungen entwickeln. Die Betonung liegt auf theoretisch, denn es gab ein kleines Problem: Es war zu diesem Zeitpunkt nämlich kein wirklich funktionierendes Tool verfügbar, mit dem die Entwicklung möglich gewesen wäre. Lediglich ein hastig zusammengefrickelter Aufsatz für Visual Studio 2005 musste zum Lernen und für die ersten Gehversuche herhalten; an eine richtige Entwicklung von WPF-Anwendungen war zu diesem Zeitpunkt noch nicht zu denken.

All diese Ereignisse der jüngeren WPF-Geschichte sorgten so dafür, dass wir erst mit Windows Vista SP1 und mit Visual Studio 2008 SP1 bzw. .NET Framework 3.5 SP1 so richtig in WPF loslegen konnten. Naja, vielleicht nicht sofort. Denn nachdem wir uns das bislang gültige Konzept noch mal in Erinnerung gerufen haben, wie Grafik mit GDI bzw. GDI+ ihren Weg auf den Bildschirm findet, und was man machen muss, damit eine Grafik auch persistent bleibt (oder zumindest so wirkt), wollen wir meinen Kollegen Marcus jetzt auch noch mal mit der notwendigen Demo versorgen, die zeigt, wie man in WPF einerseits Linien zeichnet, und wieso es andererseits um das Zeichnen von Linien mit WPF – so, wie wir es von GDI kennen – gar nicht gehen darf.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 06\SimpleSampleWPF
```

Öffnen Sie dort die Projektmappe (.SLN-Datei) *SimpleSampleWPF*.

Abbildung 6.5 Die WPF-Version des Linienzaubers sieht erst mal der GDI-Version recht ähnlich ...

Auf den ersten Blick scheint das Ergebnis sich von der GDI+-Version nicht sonderlich zu unterscheiden. Die Linien wirken aber irgendwie sauberer gezeichnet, und im Gegensatz zu GDI+ läuft das Zeichnen jedenfalls unter Vista ordentlicher, mit weniger Flimmern und subjektiv ein wenig schneller ab.

Erste wirkliche Prinzipunterschiede fallen aber bereits auf, wenn Sie das gleiche Spielchen mit diesem Fenster machen, wie wir es versuchsweise in Verbindung mit Abbildung 6.4 (Seite 196) gemacht haben. Jetzt geht beim Überlappen mit einem anderen Fenster nichts mehr verloren – auch wenn der eigentliche Malvorgang, anders als im ersten Beispiel, im `SizeChanged`-Ereignis des Fensters passiert und dementsprechend für das Neuzeichnen des Inhalts beim Vergrößern oder Verkleinern bereits gesorgt ist, wie das folgende Listing dieses Beispiels zeigt:

```
Imports System.Windows.Media.Animation

Class Window1

    Private Sub Window1_SizeChanged(ByVal sender As Object, ByVal e As
System.Windows.SizeChangedEventArgs) Handles Me.SizeChanged
        'meineLeinwand kommt aus der XAML-Definition
        'dort haben wir das vordefinierte Grid rausgeschmissen
        'und durch die Leinwand (Canvas) ersetzt.
        meineLeinwand.Children.Clear()

        'tatsächliche Breite der Leinwand ermitteln und merken
        Dim tatsächlicheBreiteGemerkt = meineLeinwand.ActualWidth
        Dim linienZähler = 0

        'Jeden 5. Pixel berücksichtigen
        For x = 0 To tatsächlicheBreiteGemerkt Step 5

            'Neues Linienobjekt anlegen und Parameter setzen
            Dim eineLinie As New Line()
            'Parameter ensprechend setzen
            eineLinie.Stroke = Brushes.Black
            eineLinie.X1 = 0
            eineLinie.Y1 = 0
            eineLinie.X2 = x
            eineLinie.Y2 = meineLeinwand.ActualHeight
            eineLinie.HorizontalAlignment = HorizontalAlignment.Left
            eineLinie.VerticalAlignment = VerticalAlignment.Center
            eineLinie.StrokeThickness = 1
            'Der Leinwand hinzufügen
            meineLeinwand.Children.Add(eineLinie)

            'Die entgegenlaufende Linie definieren
            eineLinie = New Line()
            'Wieder Parameter entsprechend setzen
            eineLinie.Stroke = Brushes.Black
            eineLinie.X1 = tatsächlicheBreiteGemerkt
            eineLinie.Y1 = 0
            eineLinie.X2 = tatsächlicheBreiteGemerkt - x
            eineLinie.Y2 = meineLeinwand.ActualHeight
            eineLinie.HorizontalAlignment = HorizontalAlignment.Left
            eineLinie.VerticalAlignment = VerticalAlignment.Center
            eineLinie.StrokeThickness = 1
            'Und der Leinwand hinzufügen
            meineLeinwand.Children.Add(eineLinie)

            'Linienzähler aktualisieren
            linienZähler += 2
        Next

        Me.Title = "Anzahl Linien im Fenster: " & linienZähler
    End Sub
    .
    .
    .
End Class
```

Tatsache ist: Anders als bei GDI oder GDI+ ist das Gezeichnete hier persistent. Und das wird umso deutlicher, wenn Sie abermals einen Doppelklick in die Grafik machen, denn dann sehen Sie, dass jede einzelne Linie beginnt, sich zu animieren, und damit einen Effekt generiert, der sich wirklich sehen lassen kann (Abbildung 6.6).

Abbildung 6.6 ...hinterlässt aber nach einem Doppelklick einen völlig anderen Eindruck, denn damit startet nämlich ein zwar einfacher Animationsvorgang, der aber jede einzelne der im Fenster enthaltenen 234 Linien betrifft!

Sie können nur ahnen, welchen Aufwand man in GDI+ bzw. GDI betreiben müsste, um diesen Animationseffekt so flimmerfrei wie hier im Beispiel hinzubekommen. Man müsste mit Doppelpufferung und Zwischenspeicherung des Anzeigebildes arbeiten, und man müsste die aktuellen Längen jeder einzelnen Linie zwischenspeichern und selbst kontrollieren. Nicht so in WPF:

```
Private Sub Window1_MouseDoubleClick(ByVal sender As Object, _
                      ByVal e As System.Windows.Input.MouseButtonEventArgs) Handles Me.MouseDoubleClick
        Dim count = 0

        'Funktioniert natürlich nur, da sich nur
        'Linien auf dem Canvas-Objekt befinden.
        For Each lineItem As Line In meineLeinwand.Children
            'Y2-Eigenschaft wird animiert, daher der Name.
            'Die Klasse DoubleAnimation animiert Eigenschaften
            'vom Typ Double.

            'Animiert wird von der aktuellen Höhe der Linie
            'bis zur halben Höhe der Linie.
            Dim Y2Animation As New DoubleAnimation(lineItem.ActualHeight, _
                         lineItem.ActualHeight / 2, TimeSpan.FromSeconds(3))

            'Hier wird festgelegt, dass die Animation automatisch wird,
            'wenn sie einmal komplett durchlaufen wurde.
            Y2Animation.RepeatBehavior = RepeatBehavior.Forever
```

```
            'Bestimmt, dass die DoubleAnimation hoch- und durch
            'AutoReverse=True anschließed wieder runterzählt.
            Y2Animation.AutoReverse = True

            'Die Animation für jede weitere Linie wird um
            '500 Millisekunden verzögert zur vorherigen begonnen,
            Y2Animation.BeginTime = New TimeSpan(500000 * count)

            'Animation starten ...
            lineItem.BeginAnimation(Line.HeightProperty, Y2Animation)
            count += 1
        Next
    End Sub
```

Jede einzelne Linie ist hier in WPF nicht nur eine simple Methode, die bewirkt, dass Veränderungen am Grafikspeicher vorgenommen werden, dessen Repräsentation dann wieder den Eindruck einer gemalten Linie bewirkt. Vielmehr ist eine Linie, wie wir sie hier verwenden, abgeleitet von einer Klasse, die schon unfassbar viel an Infrastruktur mitbringt. Die nicht nur dafür sorgt, dass eine Instanz dieser Klasse sich zu gegebener Zeit rendert und aktualisiert, sondern die sich auch in eine Animationsinfrastruktur einfügt und sich damit wie viele der meisten anderen Objekte, die WPF als »sichtbares Ding« kennt, also als so genanntes Visual, verhält und zu steuern vermag.

Abbildung 6.7 Eine TextBox, die auf eine animierte Stoffsimulation projiziert wird und dabei voll funktionsfähig bleibt – eine der eindrucksvollsten WPF-Demos derzeit, da mit klugem Kopf, aber minimalem Aufwand in die Tat umgesetzt

Und Sie haben hier wirklich nur die Spitze des Eisbergs gesehen. Zu was WPF seit dem Service Pack 1 von .NET 3.5 in der Lage ist, zeigt eine Demo von David Teitlebaum, die Sie unter *http://tinyurl.com/5kmpz7* einsehen können, auf Channel 9 übrigens sehr eindrucksvoll. Die Demodateien können Sie übrigens über David Teitlebaums Blog abrufen – sie sind allerdings leider nur in C# verfügbar.

Wie WPF Designer und Entwickler zusammenbringt

Wenn Sie bisher eine Benutzerschnittstelle entworfen und programmiert haben, dann wurden normalerweise Design und Logik schnell vermischt. In einer Windows Forms-Anwendung wird in einer Klasse, die von `System.Windows.Forms.Form` abgeleitet wird, in der Methode `InitializeComponent` das Aussehen eines Fensters in Form von Visual Basic-Code festgelegt. Nun hat ein Grafikdesigner mit Programmcode sehr wenig im Sinn. Der Grafiker benutzt diverse Werkzeuge, um schöne, bunte Grafiken zu erstellen. Er wird aber wohl kaum Visual Basic-Code schreiben wollen, um das Aussehen eines Fensters oder Steuerelements zu verändern. Andererseits ist es unmöglich, den logischen Teil der Applikation, also zum Beispiel die Datenzugriffe oder das Berechnen von Zahlen, mit einem Grafikwerkzeug durchzuführen.

Leider gibt es nur wenige Programmierer, die gleichzeitig auch hervorragende Designer sind. Das kann man natürlich auch anders herum betrachten: Es gibt wenige Topdesigner, die auch noch perfekt programmieren können.

Hier stoßen also zwei Welten aufeinander, für die es gilt, eine für beide Seiten akzeptable Brücke zu errichten. Auf der einen Seite befindet sich der Grafiker mit seinen Werkzeugen, die irgendetwas abliefern, das der Softwareentwickler auf der anderen Seite mit der Applikationslogik »unterfüttern« kann. Dabei sollte sich der Grafiker so wenig wie möglich mit Programmierung auseinandersetzen müssen, und der Entwickler sollte ohne größeres Wissen über die eingesetzten Designerwerkzeuge auskommen. Ziel ist letztendlich, dass ein Topdesigner zusammen mit einem Topsoftwareentwickler eine Topapplikation erstellt.

1. Beginnen wir im alten Stil und schreiben eine WPF-Applikation nur mit Visual Basic-Code. Wir führen zunächst einmal keine Trennung von Design und Code ein. Es handelt sich natürlich um eine weitere Version des bekannten *Hallo Welt!*-Programms. Die Vorgehensweise ist folgende:
2. Starten Sie Visual Studio 2010 und wählen Sie *Datei/Neu/Projekt* aus.
3. Im Dialog *Neues Projekt* wählen Sie als Programmiersprache *Visual Basic* aus. Im großen Listenfeld entscheiden Sie sich für *Leeres Projekt*.
4. Als Referenzen fügen Sie Verweise auf folgende Bibliotheken ein: `System`, `WindowBase`, `PresentationFramework`, `System.Xaml` und `PresentationCore`. Benutzen Sie hierzu im Projektmappen-Explorer durch Anklicken mit der rechten Maustaste den Befehl *Verweis hinzufügen*. Wählen Sie die angegebenen Referenzen im Dialog aus.
5. Fügen Sie nun im Projektmappen-Explorer eine neue Visual Basic-Codedatei vom Typ *Klasse* mit dem Namen »Hallo.vb« hinzu. Benutzen Sie wieder die rechte Maustaste und den Befehl *Hinzufügen/Neues Element*.
6. Tippen Sie nun den Code des nächsten Listings ein.
7. In den Projekt-Eigenschaften stellen Sie auf der Registerkarte *Anwendung* den folgenden *Anwendungstyp* ein: *Windows Forms-Anwendung*.

8. Führen Sie das Programm aus, indem Sie aus dem Menü *Debuggen* den Befehl *Starten ohne Debugging* aufrufen.

```
Imports System
Imports System.Windows

Namespace Hallo

    Public Class Hallo
        Inherits System.Windows.Window

        <STAThread()> _
        Shared Sub main()
            Dim w As New Window With {.Title = "Hallo, Welt!", .Width = "200", .Height = "200"}
            w.Show()

            Dim app As New Application
            app.Run()
        End Sub

    End Class
End Namespace
```

Wenn wir das Programm laufen lassen, erscheint ein ziemlich eintöniges Fenster mit dem Titel »Hallo, Welt!« (Abbildung 6.8).

Abbildung 6.8 Das erste WPF-Fenster

Schauen wir uns den Code des Beispiels nun etwas genauer an. Nach dem Einfügen der benötigten Namensräume aus den Referenzen definieren wir einen eigenen Namespace mit der Klasse `Hallo`. In WPF muss vor der statischen `Main`-Methode das Attribut `<STAThread()>` angewendet werden, um das Threading-Modell für die Applikation auf »Single-Threaded Apartment« zu setzen. Dies ist ein Relikt aus alten COM-Zeiten, aber es sorgt gegebenenfalls für eine Kompatibilität in die Welt des *Component Object Model*.

In der `Main`-Methode erzeugen wir zunächst ein `Window`-Objekt, welches über die Eigenschaft `Title` den Text für die Titelzeile erhält. Die Methode `Show` aus der `Window`-Klasse sorgt für die Darstellung des Fensters auf dem Bildschirm. Wenn wir die beiden letzten Codezeilen der `Main`-Methode nicht eingeben und unser Programm starten, dann werden wir das Fenster nur sehr, sehr kurz sehen. Die Applikation wird nämlich sofort wieder beendet. Hier kommen nun diese beiden letzten Zeilen ins Spiel. Im erzeugten `Application`-Objekt wird die Methode `Run` aufgerufen, um für dieses Hauptfenster eine Meldungsschleife (Message Loop)

zu erzeugen. Nun kann unser kleines Programm Meldungen empfangen. Das Fenster bleibt so lange sichtbar, bis diese Meldungsschleife beendet wird, zum Beispiel durch Anklicken der Schließenschaltfläche rechts oben in der Titelzeile des Fensters.

Nun haben wir in diesem Beispiel allerdings den Designer arbeitslos gemacht, denn wir haben als Softwareentwickler das Design im Visual Basic-Code implementiert. Eigentlich besteht das Design dieser Applikation nur aus einer Zeile:

```
Dim w As New Window With {.Title = "Hallo, Welt!", .Width = "200", .Height = "200"}
```

Alles andere möchte ich in diesem einfachen Beispiel einmal als Applikationslogik bezeichnen. Da ein Designer jedoch nicht mit Visual Basic-Code oder anderen Programmiersprachen arbeiten will, muss eine andere »Sprache« her, mit der man Benutzeroberflächen und Grafiken »deklarieren« kann.

In Windows Presentation Foundation wird hierzu die Extensible Application Markup Language (sprich: »gsämmel«) benutzt.

XAML: Extensible Application Markup Language

XAML ist, wie XML, eine hierarchisch orientierte Beschreibungssprache. Wir können mit XAML Benutzerschnittstellen oder Grafiken deklarieren. Kommen wir wieder zu unserem ersten Beispiel (Abbildung 6.8) zurück. Als Nächstes deklarieren wir nun den Designerteil mit XAML. Vorab jedoch noch ein kleiner Hinweis: Ein Designer wird natürlich den XAML-Code nicht »von Hand« eintippen, so wie wir es nun im nächsten Beispiel machen. Er wird stattdessen Werkzeuge verwenden, welche das erstellte Design schließlich als XAML-Code zur Verfügung stellen. In folgenden Listing können Sie nun den erforderlichen XAML-Code für unser erstes Beispielprogramm sehen:

```
<Window x:Class="Hallo2.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Hallo, Welt!"
    Height="250"
    Width="250"
       >
  </Window>
```

Was wir hier sehen, ist nicht gerade überwältigend. Diese wenigen Zeilen XAML-Code machen im Grunde genommen nichts anderes, als ein `Window`-Objekt zu deklarieren, den Titel des Fensters auf »Hallo, Welt!« zu setzen und die Größe des Fensters auf 250 × 250 Einheiten zu definieren. Hierzu werden die Eigenschaften `Title`, `Width` und `Height` auf die gewünschten Werte gesetzt.

Alle Eigenschaften in XAML werden mit Texten gefüllt. Darum müssen die Werte in Anführungszeichen gesetzt werden. Um XAML-Code zu nutzen, sollten Sie zwei Namensräume deklarieren:

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

Visual Studio 2010 fügt diese Namensräume automatisch in Ihre .NET Framework 4.0-Projekte in die XAML-Dateien ein.

Dieses Projekt wurde mit Visual Studio angelegt, und darum übernimmt Visual Studio auch den Rest der Arbeit. Im Hintergrund wird nämlich eine weitere Datei erzeugt, die Visual Basic-Code enthält und unter anderem ein `Application`-Objekt erstellt und die dazugehörige `Run`-Methode aufruft. Dieser, von Visual Studio erzeugte Code, soll uns aber gar nicht weiter interessieren. Wenn wir das Programm starten, werden wir wieder das gleiche Fenster wie im ersten Beispiel sehen.

Für das zweite Beispiel ist die Vorgehensweise mit Visual Studio 2010 folgende:

1. Starten Sie Visual Studio 2010 und wählen Sie *Datei/Neu/Projekt* aus.
2. Im Dialog *Neues Projekt* wählen Sie als Programmiersprache *Visual Basic* aus. Im großen Listenfeld entscheiden Sie sich für *WPF-Anwendung*.
3. Öffnen Sie nun im Projektmappen-Explorer die XAML-Datei mit dem Namen *MainWindow.xaml*.
4. Tippen Sie nun den Code des letzten Listings ein. Ändern Sie dabei den Code, welchen Visual Studio erzeugt hat, einfach entsprechend.
5. Führen Sie das Programm aus, indem Sie aus dem Menü *Debuggen* den Befehl *Starten ohne Debugging* aufrufen.

Nun haben wir die Deklaration der Benutzerschnittstelle in XAML hinterlegt. Dieser XAML-Code kann durch Grafikwerkzeuge erzeugt werden. Jetzt aber wieder zurück zum Programmieren. Wir kommen nun auf die Applikationslogik zurück. Im nächsten Beispiel wollen wir eine minimale Logik implementieren. Für die Benutzerschnittstelle verwenden wir wiederum XAML, für die Logik wird Visual Basic eingesetzt.

Das Beispiel soll zeigen, wie eine Benutzerschnittstelle und die Applikationslogik in einer WPF-Applikation zusammenarbeiten. Dazu deklarieren wir mitten im Fenster eine Schaltfläche in einer bestimmten Größe (siehe folgendes Listing). Das ist unsere Benutzerschnittstelle, die von einem Designer erstellt wurde:

```
<Window x:Class="Hallo3.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Hallo 3" Height="200" Width="200">
   <Grid>
       <Button Click="OnClick" Width="120" Height="40" Margin="28,58,21,64"
               >Bitte anklicken!</Button>
   </Grid>
</Window>
```

Innerhalb der Deklaration für das Hauptfenster wird ein weiteres WPF-Element definiert: die Schaltfläche. In XAML heißt dieses Element `Button`. Eine Schaltfläche hat natürlich Eigenschaften, die Sie setzen können. Hier wird die Breite (`Width`) der Schaltfläche mit 120 Einheiten angegeben und die Höhe (`Height`) mit 40 Einheiten. Der Text auf der Schaltfläche lautet: »Bitte anklicken!«.

Wenn der Benutzer des Programms auf die Schaltfläche klickt, dann soll ein Text ausgegeben werden. Sie ahnen, welcher Text? Natürlich »Hallo, Welt!«. Das ist die Applikationslogik, welche ein Softwareentwickler erstellt (siehe nächstes Listing). Die Verbindung zwischen dem `Button`-Element und dem Visual Basic-Code wird über ein Ereignis hergestellt. In XAML (vorheriges Listing) wird im `Button`-Element das `Click`-Ereignis benutzt, welches auf die Ereignismethode `OnClick` zeigt. Diese Methode wird nun in Visual Basic implementiert.

Dieses Projekt wurde ebenfalls mit Visual Studio 2010 erzeugt. Zunächst einmal werden nur die Namensräume `System` und `System.Windows` benötigt. Visual Studio erzeugt fast den gesamten Code aus dem folgenden Listing, wir müssen nur die Methode `OnClick` (im unteren Bereich) eingeben:

```
Imports System
Imports System.Windows

Namespace Hallo3
    Partial Public Class Window1
        Inherits System.Windows.Window

        ''' <summary>
        ''' Logik für dieses Beispiel
        ''' </summary>

        Public Sub New()
            InitializeComponent()
        End Sub

        Private Sub OnClick(ByVal sender As System.Object, ByVal e As  _
                            System.Windows.RoutedEventArgs)
            MessageBox.Show("Hallo, Welt!")
        End Sub
    End Class
End Namespace
```

Im Konstruktor der Fensterklasse wird die Methode `InitializeComponent` aufgerufen. Auch der Code dieser Methode wird in einer separaten Visual Basic-Codedatei von Visual Studio generiert. Der Visual Basic-Befehl, der hier zum Einsatz kommt, um die verschiedenen Dateien für den Compiler korrekt zusammenzuführen, heißt `Partial`. Der gesamte Code der Klasse kann auf mehrere Visual Basic-Dateien verteilt werden: Ein Teil der Klasse ist der Code aus dem Listing und der andere, unsichtbare Teil ist der von Visual Studio erzeugte Zusatzcode, der die Methode `InitializeComponent` und eine Art Abbildung des XAML-Codes in Visual Basic enthält. Beim Übersetzen werden beide Dateien zusammengefügt. Aus diesem Grund erhalten Sie auch eine Fehlermeldung vom Compiler, wenn Sie das Beispiel übersetzen, ohne vorher die Methode `OnClick` einzugeben.

Die Benutzerschnittstelle, die in XAML definiert wurde, wird übrigens nicht von einem XML-Interpreter abgearbeitet, also interpretiert. Das wäre sicherlich zu langsam. Auch XAML wird kompiliert. Letztendlich entsteht daraus ganz normaler MSIL-Code, wie wir ihn aus jedem .NET-Programm gewohnt sind. Dieser Zwischencode, der in der EXE-Datei steht, wird dann vom JIT-Compiler (*Just In Time*) zur Laufzeit des Programms in die Maschinensprache der jeweiligen Zielplattform übersetzt.

In die Ereignismethode werden zwei Parameter übergeben: Von Typ `Object` bekommen wir die Variable `sender`, die uns angibt, welches Objekt das Ereignis ausgelöst hat. Der Parameter `e` vom Typ `RoutedEventArgs` gibt uns zusätzliche Daten an, die zu dem jeweiligen Ereignis gehören. Diese Art der Ereignisprogrammierung kennen wir schon aus Windows Forms. In der Methode selbst wird dann einfach die Methode `MessageBox.Show` mit dem gewünschten Text aufgerufen. Das Ergebnis unserer Bemühungen zeigt Abbildung 6.9.

Abbildung 6.9 XAML und Logik werden ausgeführt

Wir sind aber noch nicht ganz am Ziel unserer Wünsche. Das ganze Programm nützt uns nichts, wenn wir die Objekte, die wir in XAML deklariert haben, nicht aus unserem Applikationscode manipulieren können. Die Schaltfläche *Bitte anklicken!* ist ein Element vom Typ `Button` und enthält diverse Eigenschaften und Methoden. Wie kann man diese nun aus dem Visual Basic-Code aufrufen?

Hierzu wollen wir das letzte Beispiel wieder ein bisschen ändern. Zunächst muss die Schaltfläche einen Namen bekommen. Dies erledigen wir durch Setzen der `Name`-Eigenschaft. Es handelt sich hierbei um die einzige Änderung am XAML-Code:

```
<Window x:Class="Hallo4.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Hallo 4" Height="200" Width="200">
         <Button Name="btn" Click="OnClick" Width="120" Height="40"
                 Margin="34,61,24,61">Bitte anklicken!</Button>
</Window>
```

Da die Schaltfläche nun einen Namen (`btn`) hat, können wir sie ganz normal aus dem Visual Basic-Code heraus ansprechen und die Eigenschaften neu setzen oder Methoden des Objekts `btn` aufrufen. Im Beispiel (nächstes Listing) ändern wir mit der Eigenschaft `Content` zunächst den Text, der auf der Schaltfläche ausgegeben wird. Schließlich werden die Fontgröße (`FontSize`) und die Vordergrundfarbe (`Foreground`) neu gesetzt. In allen Fällen wird vor der jeweiligen Eigenschaft der Name der Schaltfläche, der in XAML definiert wurde, angegeben:

```
Imports System
Imports System.Windows

Namespace Hallo4
    Partial Public Class Window1
        Inherits System.Windows.Window
        Public Sub New()
            InitializeComponent()
        End Sub

        Private Sub OnClick(ByVal sender As System.Object, _
                            ByVal e As System.Windows.RoutedEventArgs)
            MessageBox.Show("Hallo Welt!")
```

```
            ' Nach der Messagebox: Schaltfläche ändern
            With btn
                .Content = "Guten Tag!"
                .FontSize = 20
                .Foreground = Brushes.Red
            End With

        End Sub
    End Class
End Namespace
```

Abbildung 6.10 zeigt das Fenster nach dem Ausführen der Ereignismethode. Die Eigenschaften der Schaltfläche wurden entsprechend geändert, nachdem im `MessageBox`-Element die *OK*-Schaltfläche angeklickt wurde.

Abbildung 6.10 Darstellung des Fensters nach dem *MessageBox.Show*-Aufruf

Nun können wir »beide Richtungen« zwischen XAML und Code benutzen. Um aus XAML die Applikationslogik »aufzurufen«, verwenden wir Ereignisse. Diese stehen uns in den vielen WPF-Elementen in großer Anzahl zur Verfügung. Wir implementieren Ereignismethoden, welche die Applikationslogik enthalten. Um umgekehrt die in XAML deklarierten WPF-Elemente zur Laufzeit zu verändern, benutzen wir das ganz normale Objektmodell dieser Elemente und rufen die Eigenschaften und Methoden aus dem Programmcode auf. Die einzelnen Objekte werden durch ihre Namen identifiziert. Die in XAML deklarierten Eigenschaftswerte sind also die Initialeinstellungen bei der Darstellung der WPF-Elemente.

Wie eben bereits erwähnt wurde, stellt die XAML-Deklaration der Benutzerschnittstelle den Startzustand der Anwendung dar. Sie können allerdings, wenn erforderlich, die Startwerte sofort aus dem Programmcode ändern, indem Sie eine Methode für das Ereignis `Loaded` implementieren. Dies wird in einem Beispiel in den folgenden beiden Codelistings gezeigt.

Für das Hauptfenster mit dem Klassennamen `Window1` wird das Ereignis `Loaded` an die Methode `OnLoaded` gebunden. Der Code, der in der entsprechenden Visual Basic-Methode implementiert ist, wird nun direkt nach dem Konstruktoraufruf von `Window1` ausgeführt. Sobald das Fenster auf dem Bildschirm erscheint, enthält es bereits die vergrößerte Schaltfläche, da die beiden Eigenschaften `Width` und `Height` in der Methode `OnLoaded` entsprechend gesetzt wurden. Wird die Schaltfläche danach angeklickt, so wird die Ereignismethode `OnClick` aufgerufen und ausgeführt. Zunächst der XAML-Code:

```
<Window x:Class="Hallo5.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Hallo 5" Height="300" Width="300"
    Loaded="OnLoaded">
       <Button Name="btn" Click="OnClick" Width="120" Height="40"
               Margin="34,61,24,61">Bitte anklicken!</Button>
</Window>
```

Und der Visual Basic-Code:

```
Imports System
Imports System.Windows

Namespace Hallo5
    Partial Public Class Window1
        Inherits System.Windows.Window

        Public Sub New()
            InitializeComponent()
        End Sub

        Private Sub OnClick(ByVal sender As System.Object, _
                            ByVal e As System.Windows.RoutedEventArgs)
            MessageBox.Show("Hallo Welt!")

            ' Nach der Messagebox: Schaltfläche ändern
            With btn
                .Content = "Guten Tag!"
                .FontSize = 20
                .Foreground = Brushes.Red
            End With
        End Sub

        Private Sub OnLoaded(ByVal sender As System.Object, _
                             ByVal e As System.Windows.RoutedEventArgs)
            btn.Height = 80
            btn.Width = 150
        End Sub
    End Class
End Namespace
```

Abbildung 6.11 Nach der Ausführung der OnLoaded-Methode

WICHTIG Grundsätzlich können wir sagen, dass alles, was in XAML angegeben und deklariert werden kann, auch in Visual Basic mithilfe von Programmcode erzeugt werden kann. Umgekehrt gilt das allerdings nicht!

Wir können nun das Design der letzten Beispielanwendung ändern, ohne den Applikationscode zu modifizieren. Dazu wollen wir etwas ziemlich Verrücktes machen: Die Schaltfläche soll schräg im Fenster stehen und wir benötigen dazu eine so genannte Transformation. Benutzen Sie den entsprechenden XAML-Code mit der Transformation im Moment einfach so, wie in folgendem Listing aufgeführt:

```
<Window x:Class="Hallo5a.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Hallo 5a" Height="300" Width="300"
    Loaded="OnLoaded">
        <Button Name="btn" Click="OnClick" Width="120" Height="40" Margin="34,61,24,61">
            Bitte anklicken!
            <Button.RenderTransform>
                <SkewTransform AngleX="10" />
            </Button.RenderTransform>
        </Button>
</Window>
```

Die Logik, die in Visual Basic implementiert wurde (Ereignismethoden `OnLoaded`, `OnClick`), bleibt unverändert und wird für dieses Beispiel gar nicht mehr aufgelistet. Wir benutzen weiterhin den Code aus dem letzten Listing. Wenn Sie das Programm starten, sehen Sie zunächst die schräge Schaltfläche (Abbildung 6.12 links); die Funktionalität hinter dieser Schaltfläche ist jedoch unverändert geblieben. Ein Anklicken löst die `MessageBox.Show`-Methode aus und danach werden die Eigenschaften der nun schräg liegenden Schaltfläche wie bisher geändert (Abbildung 6.12 rechts).

Abbildung 6.12 Eine Applikation mit geändertem Design

Wir haben nun also eine Trennung zwischen dem Design der Applikation und der Logik erreicht. Der Softwareentwickler kann die Logik der Applikation ändern, ohne das Design (versehentlich) zu modifizieren. Umgekehrt kann ein Designer das Aussehen der Anwendung ändern, ohne dass der Programmcode ihm dabei ständig »in die Quere kommt«.

Ereignisbehandlungsroutinen in WPF und Visual Basic

Im Zusammenhang mit dem Designer möchte ich auf das »Verdrahten« von Ereignisbehandlungsroutinen in WPF-Anwendungen ein kleines bisschen näher eingehen. In Windows Forms-Anwendungen sind Visual Basic-Entwickler es gewohnt, Ereignisprozeduren per Doppelklick auf das entsprechende Steuerelement zu verdrahten – das vorherige Kapitel hat gezeigt, wie es geht. Das funktioniert in WPF-Anwendungen mit dem Designer prinzipiell ebenfalls. In beiden Fällen fügt Visual Basic dabei den Rumpf der Ereignisroutine mit einem geeigneten Namen ein, die automatisch die korrekte Signatur für das Behandeln des Ereignisses erhält. Das `Handles`-Schlüsselwort zeigt dem Visual Basic-Compiler an, dass hier die Infrastruktur für das Hinzufügen der Ereignisdelegatenliste in die `Form`- (bzw. – für WPF – `Window`-)Klasse eingefügt werden muss.

Bei WPF-Anwendungen gibt es in Ergänzung dazu auch eine Möglichkeit, die Sie auch bei der Entwicklung in jedem Fall vorziehen sollten, nämlich im XAML-Code selbst zu bestimmen, welche Ereignisprozedur für ein Ereignis eines bestimmten Objekts aufgerufen werden soll. Das funktioniert ähnlich einfach wie ein Doppelklick auf ein Steuerelement, wie Abbildung 6.13 zeigt.

Abbildung 6.13 Ereignisprozeduren lassen sich in WPF auch in Visual Basic-Programmen, anders als Sie es von Windows Forms-Anwendungen gewohnt sind, per XAML-Code »verdrahten«

Sie können den Namen des Ereignisses direkt in die XAML-Elementdefinition hineinschreiben, und IntelliSense hilft Ihnen anschließend, das Ereignis entweder mit einer bereits vorhandenen Methode (die natürlich über die entsprechende Signatur des Ereignisses verfügen muss) zu verknüpfen oder den Rumpf einer neuen Ereignisbehandlungsroutine einzufügen. In letzterem Fall würden Sie in der IntelliSense-Objekteliste einfach den ersten Eintrag <*Neuer Ereignishandler*> auswählen.

Die XAML-Syntax im Überblick

Dieser Abschnitt gibt Ihnen einen kurzen Überblick über die XAML-Syntax. Der XAML-Code soll auch mit normalem Visual Basic-Code verglichen werden. Viele Leser werden sicherlich schon Erfahrung mit XML gesammelt haben. Trotzdem soll hier mit einigen Beispielen die Syntax von XAML erläutert werden.

Eine XAML-Hierarchie beginnt immer mit einem Wurzelelement, einem `Window`- oder `Page`-Element. Dort werden die benötigten XML-Namensräume definiert.

Nun wird die Hierarchie der Benutzerschnittstelle deklariert. Alle Elemente, die in XAML benutzt werden können, existieren als normale Klassen in .NET Framework 4.0. Als Programmierer wissen wir, dass Klassen unter anderem Eigenschaften, Methoden und Ereignisse enthalten. Wir wollen nun betrachten, wie sich das in XAML verhält:

```
<Button Name="btnClear" Width="80" Height="25" Click="OnClear">Löschen</Button>
```

In der obigen XAML-Zeile wird ein Element vom Typ `Button` mit dem Namen `btnClear` deklariert. Zur Laufzeit wird also ein Objekt vom Typ `Button` erzeugt. Die Eigenschaften `Width` und `Height` werden gesetzt und außerdem wird das `Click`-Ereignis mit der Ereignismethode `OnClear` verbunden. Der Text auf der Schaltfläche wird über die Default-Eigenschaft des `Button`-Elements gesetzt. Das Gleiche könnten Sie mit folgendem Visual Basic-Code erreichen:

```
Dim btnClear As New Button
btnClear.Width = 80
btnClear.Height = 25
btnClear.Content = "Löschen"
AddHandler btnClear.Click, AddressOf OnClear
Me.Content = btnClear
```

Auf Grund des Inhaltsmodells von WPF müssen Sie in der letzten Visual Basic-Zeile die Schaltfläche als Inhalt in das übergeordnete Element einbringen. In XAML wird das einfach durch die Deklarationshierarchie erledigt:

```
<Window ...
  Width="300" Height="300">
  <!-- Die folgende Schaltfläche ist der Inhalt des Fensters -->
  <Button Click="OnClear">Test</Button>
</Window>
```

Die Ereignismethode `OnClear` wird im Visual Basic-Code implementiert und hat normalerweise folgende Kopfzeile:

```
        Public Sub OnClear(ByVal sender As System.Object, _
                           ByVal e As System.Windows.RoutedEventArgs)
```

Angehängte Eigenschaften (*attached properties*) werden Sie dann benutzen, wenn Sie auf Eigenschaften des übergeordneten Elements zugreifen müssen:

```
<Grid>
...
  <Button Grid.Row="0" Grid.Column="0">Button 1</Button>
  <Button Grid.Row="1" Grid.Column="0">Button 2</Button>
</Grid>
```

In den obigen Zeilen wird von den `Button`-Deklarationen aus auf die Eigenschaften `Row` und `Column` des äußeren `Grid`-Elements zugegriffen (mehr zum Thema *Grids* erfahren Sie später).

Oftmals werden Sie in XAML Elemente deklarieren, ohne deren Standorteigenschaft zu benutzen. In diesem Fall können Sie eine gekürzte Schreibweise benutzen. Statt

```
<TextBox Name="text" Width="100"></TextBox>
```

können Sie auch schreiben:

```
<TextBox Name="text" Width="100" />
```

Häufig müssen Sie einer Eigenschaft nicht einfach nur eine Zahl oder einen Text, sondern ein komplexes Element, welches wiederum eigene Eigenschaften besitzt, zuweisen. Als Beispiel wollen wir der Eigenschaft `RenderTransform` einer Schaltfläche (`Button`) ein Element vom Typ `RotateTransform` zuweisen, für welches die Eigenschaften `Angle`, `CenterX` und `CenterY` gesetzt werden sollen:

```
<Button Name="btnClear" Width="80" Height="25" Click="OnClear">
  Löschen
  <Button.RenderTransform>
    <RotateTransform Angle="25" CenterX="40" CenterY="12.5" />
  </Button.RenderTransform>
</Button>
```

Zunächst werden für die Schaltfläche selbst einige Eigenschaften »direkt« gesetzt (`Name`, `Width` …). In der `Button`-Klasse gibt es die Eigenschaft `RenderTransform`, der nun ein `RotateTransform`-Objekt zugewiesen werden soll. Darum wird innerhalb der Hierarchie das `RotateTransform`-Element deklariert, und die Eigenschaften `Angle`, `CenterX` und `CenterY` werden gesetzt. Der entsprechende Visual Basic-Code sieht folgendermaßen aus:

```
' Button-Element erzeugen
Dim btnClear As New Button
btnClear.Width = 80
btnClear.Height = 25
btnClear.Content = "Löschen"
AddHandler btnClear.Click, AddressOf OnClear

' Rotation erzeugen
Dim rot As New RotateTransform
rot.Angle = 25
rot.CenterX = 40
rot.CenterY = 12.5

' Rotation der Schaltfläche zuweisen
btnClear.RenderTransform = rot
Me.Content = btnClear
```

Diese Hierarchien können beliebig tief geschachtelt werden. Wie Sie am letzten Beispiel sehen können, ist XAML bei der Definition von Hierarchien meistens wesentlich einfacher und übersichtlicher als eine normale Programmiersprache.

Oft gibt es in den WPF-Objekten Eigenschaften, denen Sie mehrere Elemente eines Typs zuweisen können. In eine `ListBox` können Sie mehrere `ListBoxItem`-Elemente einfügen:

```
<ListBox>
  <ListBox.Items>
    <ListBoxItem>Test1</ListBoxItem>
    <ListBoxItem>Test2</ListBoxItem>
    <ListBoxItem>Test3</ListBoxItem>
  </ListBox.Items>
</ListBox>
```

Die drei `ListBoxItem`-Elemente werden in einem `Collection`-Objekt angelegt und der `ListBox`-Eigenschaft `Items` zugewiesen. In dem gezeigten Fall gibt es auch noch eine einfachere Schreibweise:

```
<ListBox>
  <ListBoxItem>Test1</ListBoxItem>
  <ListBoxItem>Test2</ListBoxItem>
  <ListBoxItem>Test3</ListBoxItem>
</ListBox>
```

Zum Vergleich auch hier der passende Visual Basic-Code zur Erzeugung der `ListBox`:

```
Dim lb As New ListBox
Dim item As New ListBoxItem
item.Content = "Test1"
lb.Items.Add(item)
item = New ListBoxItem
item.Content = "Test2"
lb.Items.Add(item)
item = New ListBoxItem
item.Content = "Test3"
lb.Items.Add(item)
Me.Content = lb
```

Mit XAML können Sie oft sehr einfach ganze Listen von Objekten deklarieren und zuweisen. Im folgenden Beispiel wird ein `MeshGeometry3D`-Element mit Daten initialisiert:

```
<MeshGeometry3D Positions="0,0,0 5,0,0 0,0,5"
                TriangleIndices="0 2 1"
                Normals="0,1,0 0,1,0 0,1,0" />
```

In diesem Beispiel wird die Eigenschaft `Positions` mit drei Elementen vom Typ `Point3D` initialisiert. Jedes `Point3D`-Element wird wiederum mit drei `Double`-Zahlen initialisiert, die jeweils durch Kommas getrennt in der Liste angegeben werden. Ganz ähnlich wird die Eigenschaft `Normals` gesetzt. Für die Eigenschaft `TriangleIndices` umfasst die Liste nur drei Zahlen, die einfach hinzugefügt werden. Der Visual Basic-Code zu diesem XAML-Beispiel sieht folgendermaßen aus:

```
Dim mesh As New MeshGeometry3D
mesh.Positions.Add(New Point3D(0.0, 0.0, 0.0))
mesh.Positions.Add(New Point3D(5.0, 0.0, 0.0))
mesh.Positions.Add(New Point3D(0.0, 0.0, 5.0))
```

```
mesh.TriangleIndices.Add(0)
mesh.TriangleIndices.Add(2)
 mesh.TriangleIndices.Add(1)
 mesh.Normals.Add(New Vector3D(0.0, 1.0, 0.0))
 mesh.Normals.Add(New Vector3D(0.0, 1.0, 0.0))
 mesh.Normals.Add(New Vector3D(0.0, 1.0, 0.0))
```

Wie Sie in diesem Beispiel leicht erkennen können, gibt es für die Eigenschaften `Positions`, `TriangleIndices` und `Normals` der `MeshGeometry3D`-Klasse immer eine `Add`-Methode, welche die Daten aus den XAML-Listenangaben korrekt verarbeitet und hinzufügt.

Mit diesen wenigen Beispielen haben wir die wichtigsten Syntaxelemente von XAML kennen gelernt und sollten in der Lage sein, die XAML-Hierarchien in diesem Buch zu lesen und zu verstehen. Mit etwas Übung werden Sie dann auch eigenen XAML-Code erstellen können.

HINWEIS Denken Sie daran, dass in Zukunft die XAML-Hierarchien nicht »von Hand« eingegeben, sondern von grafisch orientierten Werkzeugen generiert werden (wenn auch nicht gerade vom eingebauten WPF-Designer – Microsoft stellt aber weitere Werkzeuge zur Verfügung, über die Sie sich unter *http://www.microsoft.com/expression/* informieren können).

ImageResizer – das praktische WPF-Beispiel

Wie schon im vorherigen Kapitel soll auch in diesem das Motto gelten: Am besten funktioniert das Erlernen eines neuen Themenblocks und neuer Features von Visual Studio, wenn Sie sich ein Beispiel schnappen und dieses direkt ausprobieren. Und für WPF und die entsprechenden Tools haben wir – so meinen wir jedenfalls – uns etwas ganz Nettes einfallen lassen, das nicht nur ein typisches Beispiel darstellt, sondern dessen fertiges Ergebnis Sie später auch noch täglich nutzen können. Naja, viele von Ihnen jedenfalls:

Es beginnt mit Neugier – eine der stärksten Triebfedern des Menschen, welche seit dem Aufkommen der sozialen Netzwerke ein Freudenfeuerwerk nach dem anderen knallen lässt. Wenn ich meine Urlaubsbilder noch am gleichen Tag den neidischen Kollegen zuhause präsentieren will, ist das technisch kein Problem. Es gibt tausend Möglichkeiten, die Bilder im Internet zu verteilen, doch bergen sie alle ein Problem. Dank der Auflösung von vielen Millionen Pixel moderner Kameras sind die Bilder zwar gestochen scharf, doch das spiegelt sich auch in der Größe der Dateien wider. Und diese Tatsache ist die Grundlage der Idee für das Projekt dieses Kapitels.

Was kann dieses Kapitel Ihnen bieten? Es kann Ihnen einen schnellen, mühelosen und vor allem praktischen Einstieg in die WPF-Anwendungsentwicklung bieten. Wenn Sie bisher mit Windows Forms programmiert und diese geschätzt haben, so wird Ihnen dieses Kapitel helfen, alte Gewohnheiten – wenn auch nicht komplett – hinter sich zu lassen oder sie zumindest um WPF-Eigenarten und die des WPF-Designers zu ergänzen und damit die Tür zu einer neuen Welt zu öffnen.

Sie werden erfahren, wie Sie es schaffen, die Zeit für das manuelle Verkleinern der Bilder zu sparen und Ihre Kollegen dennoch neidisch zu machen. Das Ergebnis, der ImageResizer (Abbildung 6.14), ist ein WPF basiertes Projekt, welches uns mit möglichst effektiver Auslastung der Leistungsressourcen des Computers und minimalem Aufwand des Benutzers erlaubt, Bilder im Dateityp und in der Größe anzupassen.

Abbildung 6.14 Oberfläche des Programms

1. Starten Sie Visual Studio, falls Sie das noch nicht getan haben. Sie finden sich auf der Startseite wieder. Hier befindet sich alles, was Sie für einen schnellen Start brauchen: Zugriff auf Ihre letzten Projekte, die Möglichkeit, Projekte zu öffnen und neue Projekte anzulegen. Klicken Sie auf *Neues Projekt* und wählen Sie im daraufhin folgenden Dialog den Typ *WPF-Anwendung* aus. Geben Sie dem Projekt den Namen **ImageResizer**, und bestätigen Sie den Dialog mit *OK*.

Abbildung 6.15 Auf der Startseite ein neues Projekt anlegen

Standardmäßig besitzt eine neue WPF-Anwendung die Projektdateien *MainWindow* und *Application*. Dabei stellt *MainWindow* eine Klassendatei dar, die den Code des zurzeit noch einzigen Fensters der WPF-Anwendung enthält. Hier wird sich der größte Teil der Programmlogik abspielen, und wir werden unser Projekt durch weitere Projektdateien dieses Typs ergänzen. Dieser Dateityp kann sowohl XAML-Code als auch Programmcode (in einer genannten Code-Behind-Datei) beinhalten.

In *Application* können globale Einstellungen für das Projekt definiert werden, wie beispielsweise das Aussehen der Schaltflächen, aber auch Methoden, die unter anderem das Starten und Beenden der Anwendung betreffen. Hier steht also alles, was das Projekt im Ganzen betrifft. Auch für diesen Projektdateityp gilt, dass er XAML-Code sowie eine Code-Behind-Datei enthalten kann.

Visual Studio hat das Fenster *MainWindow* in der Entwurfsansicht geöffnet, sodass die Oberfläche bearbeitet werden kann. Die Entwurfsansicht teilt sich in den Designer und in den zugrunde liegenden Code, vergleichbar mit dem Entwurf einer Webseite: Unten steht gewissermaßen der HTML-Code und oben der Entwurf, wie er später im Webbrowser aussehen wird. Im Falle von WPF ist die Auszeichnungssprache, wie schon erwähnt, XAML, und der Designer übernimmt das Anzeigen des Fensters zur Laufzeit. Das Aussehen des Fensters wird durch Tags und deren Verschachtelung sowie Anordnung bestimmt.

Abbildung 6.16 WPF-Fenster im Designermodus

Der Code wie in Abbildung 6.16 zu sehen, lässt sich folgendermaßen übersetzen:

```
<Window x:Class="MainWindow">…</Window>
```

Durch die Tags `<Window>…</Window>` wird ein WPF-Fenster definiert, welches den Namen *MainWindow* trägt. Alles was zwischen diesen Tags definiert ist, befindet sich im Fenster.

```
<Window x:Class="MainWindow"
        Title="MainWindow" Height="350" Width="525">

</Window>
```

Hier werden drei Eigenschaften gesetzt: der Titel des Fensters (es heißt zunächst *MainWindow*), die Höhe (350 Pixel) und die Breite (525 Pixel).

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

Umgangssprachlich enthalten diese beiden Codezeilen die Grundlage für die Sprache, durch die das Fenster sein Aussehen erhält. Es werden zwei Namensräume registriert, in denen alle WPF-Objekte definiert sind. Dieses Prinzip sollten Sie sich dann merken, da Sie auf diese Art und Weise später Zugriff beispielsweise auf Ihre eigenen Steuerelemente im XAML-Code nehmen können. Sie registrieren die Assembly, in der Ihr Programm läuft, und geben ihr ein Präfix, unter dem sie zu erreichen ist.

```
xmlns:loc="clr-namespace:ImageResizer"
```

Nun sind alle Elemente der Assembly im Fenster über `<loc:NameDesElements></loc:NameDesElements>` zu erreichen und können so eingebunden werden.

```
<Window x:Class="MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MainWindow" Height="350" Width="525">
    <Grid>

    </Grid>
</Window>
```

Im Fenster geschachtelt ist ein *WPF*-`Grid`. Das `Grid` ist der einzige Inhalt eines Fensters oberster Ebene. Das ist ein großer Unterschied zwischen Windows Forms und WPF: Während Sie in Windows Forms in ein Form (was ja das Äquivalent zu einem Fenster in WPF bildet) beliebig viele Steuerelemente anordnen können, funktioniert das in WPF mit einem Fenster nicht. Jedes Fenster kann nur ein Steuerelement beherbergen und in 90% der Fälle besteht der Inhalt eines Fensters in WPF aus einem `Grid`. Das hat folgenden Grund: Das WPF-`Grid` ist das, wozu die HTML-Tabelle (`<table></table>`) jahrelang missbraucht wurde: Es teilt seine Fläche einerseits in Zeilen und Spalten auf und kann andererseits, im Unterschied zum Fenster selbst, untergeordnete Elemente anordnen.

HINWEIS Im Grunde genommen gibt es in WPF nur zwei Sorten von Steuerelementen. Die erste Art sind solche, die einen Inhalt (also ein weiteres Steuerelement) beinhalten können. *Inhalt* heißt auf Englisch *Content*, und deswegen haben solche Steuerelemente eine `Content`-Eigenschaft. Die zweite Sorte bilden Steuerelemente, die dazu dienen, andere Steuerelemente zu schachteln und in einem bestimmten Stil anzuordnen; sie fungieren also als *Container* für andere Steuerelemente, stellen selbst aber nichts visuell dar. Diese Trennung gibt es so in Windows Forms nicht. Hier kann jedes Steuerelement auch als Container fungieren (so und gerade auch das Form selbst), und daran muss man sich als Entwickler unter WPF erst einmal gewöhnen.

Übrigens, Sie können das einmal selbst ausprobieren: Löschen Sie zunächst das `Grid` im Fenster, und ziehen Sie eine Schaltfläche aus der Toolbox in das Fenster. Das funktioniert. Versuchen Sie es mit einer weiteren Schaltfläche, werden Sie scheitern – der Designer verweigert die Anordnung der zweiten Schaltfläche. Das Fenster selbst gehört nämlich auch zu den `Content`-Steuerelementen, das heißt, es kann nur ein einziges Steuerelement beherbergen. Damit *muss* Ihr einziges Steuerelement ein Container-Steuerelement sein, und das ist standardmäßig eben ein `Grid`-Steuerelement, weil es am flexibelsten layoutet werden kann.

Das bisher beschriebene Wissen über WPF soll als theoretische Grundlage reichen. Doch bevor Sie irgendetwas praktisch umsetzen, machen Sie sich eine Vorstellung davon, wie Ihre Oberfläche aussehen soll. Bitte unterschätzen Sie diesen Schritt nicht, er wird Ihnen beim Designen der Oberfläche mit WPF im Nachhinein viel Arbeit ersparen können.

Überlegen Sie sich:

- **Welche Informationen braucht das Programm und welche Steuerelemente resultieren daraus?**
  - Bilder-/Ordnerpfade müssen eingelesen werden:

    → TextBox, Button
  - Der Ausgabeordner muss eingelesen werden:

    → TextBox, Button
  - Eingabe eines Postfixes:

    → TextBox
  - Der Dateityp muss ausgewählt werden:

    → ComboBox
  - Die Auflösung muss eingelesen werden:

    → TextBox
  - Vorgang starten:

    → Button
- **Welche Steuerelemente brauche ich, um dem Benutzer Informationen anzuzeigen?**
  - Beschriftung der TextBoxen:

    → Label
  - Liste aller ausgewählten Bilder

    → ListBox

- **Welche zusätzlichen Steuerelemente wie Beschriftungen, Überschriften, Fortschrittsanzeigen und Ähnliches brauche ich?**
  - Überschrift im Fenster

    → TextBlock
  - Statusanzeige mit einem Fortschrittsbalken für den Bearbeitungsfortschritt und das Anzeigen des aktuellen Bildes

    → StatusBar, ProgressBar, TextBlock
- **Wie sollen sich die Steuerelemente verhalten, wenn der Benutzer das Fenster größer zieht?**
  - Die Überschrift und die StatusBar bleiben statisch. Auch die Eingabefelder sollen nicht wachsen, die Liste für die Dateinamen dennoch schon. Die ListBox soll sich dynamisch vergrößern.
- **Wie ordne ich die Steuerelemente am unmissverständlichsten an?**

Nachdem Sie sich die ersten vier Punkte klargemacht haben, sollten Sie sich im letzten Schritt Ihre Ideen bildlich vorstellen, was natürlich am besten mit Stift und Papier geht. Malen Sie ihr Fenster auf, wie es nachher aussehen soll. Und dann überlegen Sie sich, wie Sie den Entwurf am besten in WPF umsetzen können, zum Beispiel und am besten mithilfe eines `Grid`, welches Ihnen die Definition von Zeilen und Spalten erlaubt.

Einen Entwurfsvorschlag für die Aufteilung der Oberfläche des Projekts ImageResizer, an dem Sie sich im Folgenden orientieren sollten, zeigt Abbildung 6.17.

Abbildung 6.17 Die Oberfläche im Entwurf

Eine mögliche Aufteilung der Oberfläche in Zeilen und Spalten sehen Sie in Abbildung 6.18. Die Oberfläche ist in 6 Zeilen und 5 Spalten aufgeteilt.

Abbildung 6.18 Einteilung des Entwurfs der Oberfläche in Zeilen und Spalten

Um eine Oberfläche in WPF umzusetzen, gibt es prinzipiell zwei Möglichkeiten: den Designer und den Code. Ich persönlich würde Ihnen empfehlen, so viel wie möglich Code händisch zu definieren, weil daraus ein sauberer, pflegbarer und verständlicher Code resultiert, der sich auch später einfach debuggen lässt. Das mag am Anfang eher mühsam sein, doch es erspart einem die unter Umständen aufwändige Fehlersuche im vom Designer erzeugten Code. Sollten Sie eine komplizierte und verschachtelte Oberfläche im Designer konzipiert haben, kann das zur Laufzeit ein ungewolltes Ergebnis produzieren, dessen Ursache Sie dann im Code wieder finden müssen. Ein Designer ist universell und kann Ihre Ideen bis zu einem gewissen Grad umsetzen, jedoch vielleicht nicht auf die beste Art und Weise.

Definieren von Grid-Spalten und -Zeilen zum Anordnen von Steuerelementen

Trotzdem kann er lästige Tipparbeit übernehmen. Zum Beispiel beim Definieren der Zeilen und Spalten des `Grid`.

2. In der Entwurfsansicht des *MainWindow* in Visual Studio klicken Sie auf das Fenster und vergrößern es mithilfe des kleinen blauen Rechtecks rechts unten, bis es Ihnen passend erscheint.

Abbildung 6.19 Vergrößern Sie das Fenster, bis es Ihnen groß genug erscheint.

3. Anschließend klicken Sie auf das im Fenster befindliche `Grid` und bewegen den Mauszeiger an dessen blauen Rahmen entlang. Sie sehen eine Linie zum Definieren einer Zeile bzw. Spalte. Um eine Zeile oder Spalte Wirklichkeit werden zu lassen, suchen Sie sich die richtige Dimension der Zeile oder Spalte aus und klicken Sie an die Stelle.

Abbildung 6.20 Das Grid im Designer in Spalten und Zeilen aufteilen

4. Setzen Sie dieses Verfahren fort, bis Sie eine ähnliche Aufteilung wie in Abbildung 6.21 erreicht haben.

Abbildung 6.21 Grid nach Aufteilung in Zeilen und Spalten

Sie sehen: Die erste Spalte ist zu groß, die letzte etwas zu klein. Das Feintuning der Aufteilung kann nun sehr einfach und schnell im XAML-Code vorgenommen werden. Sie sehen, dass der WPF-Designer einiges an XAML-Code produziert hat:

```
<Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="45*" />
            <RowDefinition Height="163*" />
            <RowDefinition Height="54*" />
            <RowDefinition Height="57*" />
            <RowDefinition Height="55*" />
            <RowDefinition Height="41*" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="164*" />
            <ColumnDefinition Width="156*" />
            <ColumnDefinition Width="157*" />
            <ColumnDefinition Width="90*" />
            <ColumnDefinition Width="142*" />
        </Grid.ColumnDefinitions>
    </Grid>
```

Das Zeichen »*« neben den Werteangaben für die Zeilenhöhe bzw. Spaltenbreite bedeutet hier so viel wie »anteilig«. Die erste Zeile erhält anteilig »45« vom Ganzen, wohingegen die zweite mit »163 Teilen anteilig am Ganzen« eindeutig besser dran ist. »Anteilig« (also mit »*«) werden Spalten und Zeilen im Grid immer, wenn Sie sie mit dem WPF-Designer zusammenklicken. Möchten Sie feste Pixelbreiten, löschen Sie das Sternchen im XAML-Code einfach raus.

HINWEIS Prinzipiell geht das auch mit dem Designer (Abbildung 6.22), dauert aber unnötig lange, deswegen wollen wir hier im Beispiel nicht näher darauf eingehen.

Abbildung 6.22 Nur der Vollständigkeit halber erwähnt – feste Spalten- oder Zeilenmaße erreichen Sie auch über den Designer; nur sehr viel umständlicher und langwieriger

Im Entwurf liegt die Überschrift in der ersten `Grid`-Zeile und die Statusleiste in der letzten. Diese beiden Zeilen sollen statisch bleiben.

5. Platzieren Sie die Einfügemarke im Quellcode und ändern Sie die Angaben der Höhe in der ersten Zeile auf *40* und in der letzten Zeile auf *30*.
6. Die Eingabefelder liegen in den Zeilen drei, vier und fünf. Setzen Sie auch die Höhe dieser Zeilen auf *40*.
7. Es bleibt nur noch die Zeile mit der `ListBox`, die sich dynamisch vergrößern soll. Ändern Sie die Höhenangabe hier auf »*«. Nun bleibt alles bei seiner festen Größe und nur die `ListBox` mit den Dateinamen wächst später beim Vergrößern des Fensters dynamisch mit.

```
<Grid.RowDefinitions>
    <RowDefinition Height="40" />
    <RowDefinition Height="*" />
    <RowDefinition Height="40" />
    <RowDefinition Height="40" />
    <RowDefinition Height="40" />
    <RowDefinition Height="30" />
</Grid.RowDefinitions>
```

8. Wiederholen Sie die gleiche Vorgehensweise mit den Spalten. Überlegen Sie sich, welche Spalte gleich bleibend groß sein soll, und welche Spalten sich dynamisch vergrößern sollen. In der ersten Spalte befinden sich in der Entwurfsansicht die Textblöcke zur Beschriftung der Eingabefelder und in der letzten Spalte die Schaltflächen zum Bedienen des Programms. Diese beiden Spalten sollten sich nicht vergrößern, da weder die Schaltflächen noch die Textblöcke besonders hübsch oder sinnvoll sind, wenn diese breit gezogen dargestellt werden. Die restlichen Spalten sollen dynamisch mitwachsen. Ändern Sie die Werte der Breite in der ersten und letzten Spalte auf *120* und *140*, und gleichen Sie auch die dynamischen Werte wie unten stehend an:

```
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="120" />
            <ColumnDefinition Width="140*" />
            <ColumnDefinition Width="130*" />
            <ColumnDefinition Width="80*" />
            <ColumnDefinition Width="140" />
        </Grid.ColumnDefinitions>
```

Der Grundstein für die Oberflächengestaltung ist gelegt und die Oberfläche bereit für die Anordnung der Steuerelemente in der entstandenen Aufteilung.

WICHTIG Die erste Zeile eines `Grid` ist immer die 0, daraufhin folgt die 1. usw. Wenn Sie Steuerelementen eine Zeile des `Grid` zuweisen, denken Sie also immer daran, dass Sie von *null* an zählen müssen. Das Gleiche gilt natürlich auch für Spalten im `Grid`.

»Begin at the beginning«, the King said very gravely, »and go on till you come to the end: then stop!«[7]

Dieses Zitat von Lewis Carroll wird richtungsweisend bei der weiteren Vorgehensweise sein und nun befolgt, indem als Erstes die Überschrift ihren Platz auf der Oberfläche findet.

9. Bewegen Sie den Mauszeiger an den linken Rand der Visual Studio-IDE und blenden Sie durch Klicken des Toolbox-Symbols die Toolbox ein.

Abbildung 6.23 Ziehen eines *TextBlock* aus der Toolbox auf die Oberfläche

[7] Zu Deutsch: »Fange beim Anfang an«, sagte der König ernsthaft, »und lies, bis du ans Ende kommst, dann halte inne.«

10. Positionieren Sie per Ziehen und Ablegen, wie in Abbildung 6.23 zu sehen, einen `TextBlock` in der obersten Zeile des `Grid`. Der `TextBlock` sollte sich nun ungefähr an der Stelle befinden, wie sie Abbildung 6.24 zeigt. Klicken Sie den `TextBlock` mit der rechten Maustaste an und wählen Sie *Layout zurücksetzen/Alle*.

Abbildung 6.24 Positionierung des *TextBlock* und Anpassen des Layouts

11. Das Ergebnis ist ein `TextBlock`, der die Zelle vollständig ausfüllt. Ändern Sie rechts im Eigenschaftenfenster den Wert der Eigenschaft `Text` von *TextBlock* auf **ImageResizer**. Das Eigenschaftenfenster können Sie auch aufrufen, indem Sie das Steuerelement mit der rechten Maustaste anklicken und den Menüeintrag *Eigenschaften* auswählen.

Abbildung 6.25 Die Text-Eigenschaft im Eigenschaftenfenster ändern

12. Ändern Sie auch die Eigenschaften in der Kategorie *Text*, bis sie denen in Abbildung 6.26 entsprechen.

Abbildung 6.26 Anpassen der Eigenschaften der Kategorie *Text* im Eigenschaftenfenster

13. Die Überschrift soll sich in der Mitte der obersten Zeile befinden und sich über alle 5 Spalten erstrecken. Um dieses Ergebnis zu erreichen, müssen folgende Eigenschaften in der Kategorie *Layout* geändert werden: `HorizontalAlignment` erhält den Wert *Center*, `ColumnSpan` den Wert *5* und `Column` den Wert *0* (Abbildung 6.27).

Abbildung 6.27 Anpassen der Eigenschaften der Kategorie Layout

Der `TextBlock` befindet sich in Spalte 0, erstreckt sich über 5 Spalten und positioniert sich zentriert. All diese Eigenschaften finden sich auch im XAML-Code wieder. Der WPF-Designer hat XAML-Code generiert, der unsere Änderungen ausdrückt:

```
<TextBlock Grid.Column="0" Name="TextBlock1" Text="ImageResizer" FontSize="24" FontWeight="Bold"
HorizontalAlignment="Center" Grid.ColumnSpan="5" />
```

Als Nächstes wird die `ListBox` für die Anzeige der Dateinamen auf die Oberfläche gebracht. Dazu verwenden wir dieses Mal nicht den WPF-Designer, sondern, um zu trainieren und zu zeigen, wie es schneller und einfacher gehen kann, direkt den XAML-Editor:

14. Positionieren Sie die Einfügemarke im XAML-Code eine Zeile unterhalb der eben eingefügten `TextBox` und tippen Sie **<Lis**. Sie werden sehen, dass IntelliSense auch hier aktiv ist und Ihnen entsprechende Vorschläge macht (Abbildung 6.28).

Abbildung 6.28 IntelliSense lässt Sie auch im XAML- Editor nicht alleine

15. Wählen Sie aus der Liste `ListBox` und tippen Sie **>**. Visual Studio komplettiert Ihre Eingabe mit dem entsprechenden Endetag. Vervollständigen Sie die Zeile folgendermaßen:

```
<ListBox Name="FilenamesListBox" Grid.ColumnSpan="5"></ListBox>
```

Die `ListBox` ist im Code Behind nun durch den Namen *FilenamesListBox* ansprechbar – das, was Sie also als Äquivalent aus Windows Form kennen, wenn Sie Namen für Elemente vergeben, um sie aus dem Visual Basic-Code heraus anzusprechen, haben wir gerade direkt in XAML gemacht. Wie Sie den jeweiligen Code in der Code-Behind-Datei hinterlegen, werden wir später noch sehen.

Darüber hinaus haben wir mit der `Grid.ColumnSpan`-Eigenschaft im XAML-Code direkt mitbestimmt, dass sich die `ListBox` über die kompletten 5 definierten Spalten des `Grid` erstrecken soll. Jedoch gibt es noch ein Problem: Sie liegt zunächst noch mit dem `TextBlock` in einer Zeile, die Standardeinstellung für die Zeile ist nämlich 0. Steht also *keine* explizite Zeilenangabe im Code, befindet sich das Steuerelement in der obersten (der nullten) Zeile des `Grid`. Das Gleiche gilt für die Spalten. Zudem wäre es schön, wenn die `ListBox` nicht am Rand »kleben«, sondern einen gewissen Abstand nach außen einhalten würde. Die Eigenschaft `Margin` kümmert sich darum.

16. Ergänzen Sie die Codezeile um folgende Angaben:

```
<ListBox Name="FilenamesListBox" Grid.ColumnSpan="5" Grid.Row="1" Margin="4,4,4,4"></ListBox>
```

TIPP Anstelle, wie im Beispiel, alle vier Ränder für die `Margin`-Eigenschaft einzeln einzugeben, können Sie, falls die Werte alle gleich lauten, auch die verkürzte Notierung verwenden, also in etwa wie folgt:

```
<ListBox Name="FilenamesListBox" Grid.ColumnSpan="5" Grid.Row="1" Margin="4"></ListBox>
```

Das spart Zeit und Platz und trägt obendrein zur Übersichtlichkeit bei.

Das Resultat sollte nun in etwa wie in Abbildung 6.29 dargestellt aussehen.

Abbildung 6.29 Die Oberfläche nach dem Einfügen des *TextBlock* und der *ListBox*

Dieses Wissen reicht für die weitere Oberflächengestaltung aus. Sie können nun Steuerelemente sowohl per Designer auf die Oberfläche bringen als auch per Hand implementieren. Welche Vorgehensweise Sie im Folgenden bevorzugen, ist Ihnen natürlich selbst überlassen.

Als Nächstes sollen die Schaltflächen ihren Weg auf die Oberfläche finden. Jeder `Button` hat die Eigenschaft *Content*, deren Inhalt die Beschriftung der Schaltfläche vorgibt.

17. Fügen Sie drei Steuerelemente des Typ `Button` mit folgenden Eigenschaften ein:

Button-Eigenschaften				
Name	**Row**	**Column**	**Content**	**Margin**
AddImagesButton	2	4	Bilder hinzufügen...	4
ChooseOutputFolderButton	3	4	Ausgabeordner...	4
StartButton	4	4	Start!	4

Abbildung 6.30 Fügen Sie der Oberfläche drei Schaltflächen hinzu

Als Nächstes kümmern wir uns um die Beschriftungen der Elemente im Fenster. Zu diesem Zweck platzieren Sie vier Steuerelemente vom Typ `Label` im `Grid`. Die Beschriftung wird ebenfalls wieder mit der Eigenschaft `Content` festgesetzt. Die Eigenschaften für die vier Labels können Sie der folgenden Tabelle entnehmen oder aber auch aus dem Entwurf ableiten. Die Eigenschaft `Name` der Labels muss nicht gesetzt werden, da sie rein dekorative Zwecke erfüllen und später nicht vom Code aus verändert werden müssen. Der Designer gibt den Steuerelementen ungeachtet dessen standardmäßig immer einen Namen.

HINWEIS Falls Sie den Code für die Labels per Hand eingeben, denken Sie daran, dass der Standardwert für Zeilen und Spalten 0 ist und deswegen bei den ersten drei Labels die Spaltenangabe nicht mitgetippt werden muss.

Die Eigenschaften `HorizontalContentAlignment` und `VerticalContentAlignment` kümmern sich um die Positionierung des Inhalts *innerhalb* des Steuerelements (links, zentriert, rechts, oben, unten).

Content	Row	Column	HorizontalContentAlignment	VerticalContentAlignment
Neue X-Auflösung:	2	0	Right	Center
Ausgabeordner:	3	0	Right	Center
Postfix:	4	0	Right	Center
Neues Bildformat:	2	2	Right	Center

Abbildung 6.31 Fügen Sie der Oberfläche vier Labels hinzu

Um später zur Laufzeit die Eingaben des Benutzers entgegenzunehmen, fehlen noch Eingabefelder, und um die kümmern wir uns im Folgenden.

18. Fügen Sie dem Fenster drei Steuerelemente vom Typ `TextBox` mit folgenden Eigenschaften hinzu:

Name	Row	Column	Grid.ColumnSpan	Text	Margin	VerticalContentAlignment
X_ResTextBox	2	1	(Eigenschaft weglassen)	1024	4,6,0,6	Center
OutputFolderTextBox	3	1	3	(leer)	4,6	Center
PostfixTextBox	4	1	3	_klein	4,6	Center

Die Werte für den Außenabstand (`Margin`) sind hier anders angegeben als bisher. Ist nur eine Dezimalzahl spezifiziert, gilt der Wert für alle 4 Außenabstände – diese Eingabehilfe haben wir vorhin schon kennen gelernt; bei *zwei* Werten dagegen werden jeweils die Werte für den Abstand rechts und links bzw. oben und unten zusammengefasst: Die erste Zahl bezieht sich auf den Abstand der rechten und linken Seiten, die zweite auf den Abstand oben und unten. Sind alle 4 Angaben vorhanden, so wirken die Zahlen wie folgt auf die Seiten: links, oben, rechts, unten – also einmal im Uhrzeigersinn um das Steuerelement herum.

Abbildung 6.32 Fast fertig: Der Oberfläche fehlen jetzt nur noch zwei Elemente

Was noch fehlt, ist die Auswahlmöglichkeit für den Typ des Bildes.

19. Fügen Sie dafür den folgenden Code im XAML-Editor ein:

```
<ComboBox Grid.Column="3" Grid.Row="2" Name="FormatComboBox" Margin="4"
          VerticalContentAlignment="Center">
    <ComboBoxItem>.jpg</ComboBoxItem>
```

```
        <ComboBoxItem>.bmp</ComboBoxItem>
        <ComboBoxItem>.png</ComboBoxItem>
        <ComboBoxItem>.tif</ComboBoxItem>
    </ComboBox>
```

Was macht dieser Code? Er fügt eine `ComboBox` in die Zelle in Zeile 3 und Spalte 4 ein, nennt sie *FormatComboBox* und gibt ihr den Außenabstand 4. Der Inhalt wird angewiesen, sich zu zentrieren. Es werden 4 Listenelemente definiert, die dann zur Laufzeit zur Auswahl stehen – diese Vorgehensweise entspricht also der, wenn Sie im Code der `Items`-Auflistung der `ComboBox` Elemente mit der `Add`-Methode hinzufügen würden.

Um die Oberfläche zu komplettieren, fehlt nun noch die `StatusBar` in der letzten Zeile.

20. Ergänzen Sie diese wie folgt:

```
    <StatusBar Grid.ColumnSpan="5" Grid.Row="5" Name="StatusBar1">
        <StatusBarItem>
            <TextBlock Text="Gesamtfortschritt:"/>
        </StatusBarItem>
        <StatusBarItem>
            <ProgressBar Height="20" Width="150" Name="TotalProgressBar" />
        </StatusBarItem>
        <StatusBarItem>
            <TextBlock Text="Aktuelle Datei:" Name="CurrentFileTextBlock" />
        </StatusBarItem>
    </StatusBar>
```

Die `StatusBar` schachtelt hier drei untergeordnete Elemente, davon zwei Textblöcke und eine `ProgressBar`. Die `ProgressBar` namens *TotalProgressBar* wird zur Laufzeit den Fortschritt der Verkleinerung bzw. Umwandlung der Bilder anzeigen, und der TextBlock *CurrentFileTextBlock* wird darüber Auskunft geben, welche Datei gerade bearbeitet wird.

Damit ist die Oberfläche komplettiert und enthält nun unsere insgesamt 18 Steuerelemente. Und falls Sie XAML vorher noch nicht kannten, haben Sie nebenbei eine Menge über den Umgang mit dieser Auszeichnungssprache gelernt.

Abbildung 6.33 Die fertige Oberfläche

Nachträgliches Einfügen einer neuen Zeile

Nicht selten kommt nun die Stelle, wo der Fertigungsprozess Ihre Kreativität so angekurbelt hat, dass Ihnen noch eine (ultimative) Idee für die Oberflächengestaltung kommt. Wie wäre es beispielsweise mit einem Menü für den *ImageResizer*, mit einem Eintrag, mit dessen Hilfe das Programm beendet werden kann, alle Eingaben zurückgesetzt oder bestimmte Einstellungen geändert werden können? Doch leider ist auf unserem so toll durchgeplanten Fenster dafür nun leider kein Platz mehr. Alle Zeilen sind vollständig belegt und alle Steuerelemente müssten beim Einfügen einer neuen Zeile eine Position nach unten verschoben werden, was ziemlich aufwändig ist. Doch kein Problem ohne Lösung: Der Designer hilft in diesem Fall schneller als der XAML-Editor (einer der wenigen Fälle). Dabei gibt es einen kleinen Trick, den Sie im Folgenden nachvollziehen können:

21. Klicken Sie im Designer auf das `Grid` und platzieren Sie den Mauszeiger so auf dem blauen Rahmen, dass Sie mit der Linie die erste Zeile quasi halbieren, etwa wie in Abbildung 6.34 zu sehen.

Abbildung 6.34 Einfügen einer neuen Zeile: Schritt 1

22. Klicken Sie, um eine neue Zeile zu definieren. Sie werden sehen, dass der WPF-Designer die Definition der Zeilen angepasst hat. Die erste Zeile, die vormals die Höhe 40 hatte, ist nun in zwei Zeilen aufgeteilt, wobei Sie vielleicht die Mitte genauer getroffen haben als hier und nun über zwei Zeilen mit der Höhe 20 verfügen.

```
<Grid.RowDefinitions>
            <RowDefinition Height="17" />
            <RowDefinition Height="23" />
            <RowDefinition Height="*" />
            <RowDefinition Height="40" />
            <RowDefinition Height="40" />
            <RowDefinition Height="40" />
            <RowDefinition Height="30" />
        </Grid.RowDefinitions>
```

Auch hat der WPF-Designer die Zeilenangaben aller Steuerelemente automatisch für Sie angepasst.

Abbildung 6.35 Einfügen einer neuen Zeile: Die Zeilenangaben der darunter liegenden Zeilen wurden automatisch angepasst.

Das ging doch schnell, oder nicht? Das gleiche Prinzip lässt sich natürlich auch für Spalten anwenden. Passen Sie die neuen Zeilen in ihrer Höhe an, sodass die oberste Zeile den Wert *35* annimmt und die folgende den Wert *40*. Der Designer hat die Angaben der Position des `TextBlock` dahingehend geändert, dass seine vorherige Position im `Grid` unverändert ist:

```
<TextBlock Grid.Column="1" Name="TextBlock1" Text="ImageResizer" FontSize="24" FontWeight="Bold"
HorizontalAlignment="Center" Grid.ColumnSpan="2" Margin="159,0,37,0" Grid.RowSpan="2" />
```

Die Eigenschaft `RowSpan` hat nun den Wert *2*, das heißt, dass sich der `TextBlock` nun über beide Zeilen erstreckt. Gleichzeitig hat der Designer festgestellt, dass der `TextBlock` für eine Spalte zu breit ist und den Platz von zwei Spalten benötigt. Auch besitzt der `TextBlock` einen Abstand nach links und rechts, welchen der Designer in der Eigenschaft `Margin` festgehalten hat. Das ist ein Beispiel für Code des Designers, wie er im Buche steht. Der Designer hat den `TextBlock` in die zweite Spalte (also `Column="1"` wegen der nullbasierten Zählung) geschoben, ihm zwei Spalten Platz zugewiesen und ihm des Weiteren noch einen Außenabstand verpasst, der ihn genau dahin positioniert, wo er sich vor der Zeilenaufteilung auch befunden hat. Der `TextBlock` steht an der richtigen Position, und doch bleibt die Frage, ob diese Art der Positionierung nicht verwirrend oder umständlich ist, denn: Der `TextBlock` sollte sich eigentlich nur in der ersten Zeile befinden und sich in der Mitte *aller* 5 Spalten positionieren. Um das zu erreichen, ändern Sie den Code wie folgt:

- Löschen Sie die Angaben für `Margin`.
- Löschen Sie die Angaben für `RowSpan`.
- Ändern Sie die Angabe für `ColumnSpan` auf *5*.
- Ändern Sie die Angabe für `Column` auf *0*.
- Setzen Sie den Wert für `Row` auf *1*.

Das Ergebnis sieht damit in etwa wie folgt aus:

```
<TextBlock Grid.Column="0" Grid.Row="1" Grid.ColumnSpan="5" HorizontalAlignment="Center"
                   Name="TextBlock1" Text="ImageResizer" FontSize="24" FontWeight="Bold" />
```

Sie könnten einwenden, dass sich ein solches Verhalten als »Pingelei« bezeichnen ließe. Doch behalten Sie im Hinterkopf: Sollten Sie einmal gezwungen sein, den Code einer komplexen Benutzeroberfläche, die mit dem Designer erstellt wurde, per Hand zu ändern, werden Sie viel Verständnis für die Denkweise des Designers und somit die Strukturierung des Codes aufbringen müssen, um ihn und all seine Konsequenzen vollständig zu verstehen.

Einmal das Hauptmenü, bitte!

23. Um die eigentliche Idee des Menüs umzusetzen, fügen Sie im Code oberhalb des `TextBlock` mit der Überschrift folgenden Code hinzu.

```
<Menu Grid.ColumnSpan="5" Margin="5"></Menu>
```

Wer Menüs aus Windows Forms und dem dazugehörigen Designer schätzt, wird im WPF-Designer auch in der vorliegenden Version von Visual Studio 2010 eher enttäuscht werden. Das Menü lässt sich nur per Hand im XAML-Editor aufbauen; Designerunterstützung: Fehlanzeige. Das hat aber auch einen Vorteil: Das Aufbauen eines Menüs per Codeeingabe geht schnell von der Hand und ist außergewöhnlich flexibel, da seine Struktur jeglichem Kreativitätsanspruch standhält: Das Menü weist Menüelemente auf, die wiederum ebenfalls Menüelemente oder auch jedes andere Steuerelement enthalten können. Sie könnten also rein theoretisch einen Menüeintrag erstellen, der eine Combobox beinhaltet, die beim Öffnen abspielende Videos zeigt – das Konzept der `Context`-Eigenschaft macht es möglich! Menüelemente haben eine Eigenschaft namens `Header`, deren Wert im Normalfall den Text des Menüelements definiert, aber eben auch Bilder, Videos und jedes andere Steuerelement enthalten kann.

Der klassische Fall der Menüführung über den Menüeintrag *Datei* zum Befehl *Beenden* wird also umgesetzt so aussehen:

```
    <Menu Grid.ColumnSpan="5" Margin="5">
          <MenuItem Header="Datei">
              <MenuItem Header="Beenden"></MenuItem>
          </MenuItem>
      </Menu>
```

Abbildung 6.36 Das eben hinzugefügte Menü: Code und Ansicht im Designer

Das Element mit der Aufschrift *Beenden* ist also ein untergeordnetes Element des Elements mit der Aufschrift *Datei* und wird zur Laufzeit als Unterpunkt zu sehen sein. Probieren Sie es aus:

24. Ergänzen Sie den Code entsprechend dem oberen Listing, und starten Sie das Programm, indem Sie wahlweise F5 drücken oder aber über den Menüeintrag *Debuggen* den Befehl *Debugging starten* wählen.

Abbildung 6.37 Das Menü zur Laufzeit

Steuerelemente mit Ereignissen verknüpfen

Da schön auszusehen ja bekanntlich nicht reicht, geht es jetzt an die Funktionalität. Beenden Sie zunächst das Debugging über das »x«-Symbol im Fenster oder die entsprechende Schaltfläche in Visual Studio.

25. Platzieren Sie die Einfügemarke im Menüelement *Beenden* und geben Sie dem Menüelement den Namen *MenuItemBeenden*. Schreiben Sie dann weiter **Click=**. IntelliSense zeigt Ihnen alle verfügbaren Methoden, die sich bereits im Code Behind befinden und die mit der Signatur des `Click`-Ereignisses des `MenuItem` übereinstimmen und gibt Ihnen darüber hinaus die Möglichkeit, einen neuen Methodenrumpf für die Behandlung des `Click`-Ereignisses dort hinzuzufügen. Da das Fenster noch keine Methoden besitzt, sehen Sie für unser Beispiel lediglich die Möglichkeit, einen neuen Methodenrumpf einzufügen. Nehmen Sie diese Möglichkeit wahr, und klicken Sie doppelt auf `<Neuer Ereignishandler>` (Abbildung 6.38).

```
<Menu Grid.ColumnSpan="5" Margin="5">
    <MenuItem Header="Datei">
        <MenuItem Header="Beenden" Name="MenuItemBeenden" Click=""></MenuItem>
    </MenuItem>
</Menu>
```

<Neuer Ereignishandler>

Abbildung 6.38 Das *Click*-Ereignis des Menüelements mit einer Methode verknüpfen

26. Drücken Sie F7, um in die Code-Behind-Datei zu gelangen oder klicken Sie im Projektmappen-Explorer doppelt auf die Datei *MainWindow.xaml.vb*, um sie zu öffnen. Sie werden feststellen, dass Visual Studio den Methodenrumpf eingefügt hat.

27. Vervollständigen Sie den Methodenrumpf wie folgt, und starten Sie dann das Debugging:

```
Private Sub MenuItemBeenden_Click(ByVal sender As System.Object,
    ByVal e As System.Windows.RoutedEventArgs)
        Me.Close()
    End Sub
```

Wenn Sie nun zur Laufzeit auf das Menüelement *Beenden* klicken, wird das Fenster (und damit das Programm) geschlossen.

Laden der Standardeinstellungen

Der Ordner, in dem sich die bearbeiteten Bilder sehr wahrscheinlich wiederfinden sollen, ist der standardmäßige, für diesen Zweck vorgesehene Windows-Ordner *Eigene Bilder* bzw. *Bilder* (bzw. zur übersichtlicheren Strukturierung ein Unterordner davon) des angemeldeten Benutzers. Dieser Pfad soll beim Laden des Fensters schon voreingestellt sein. Auch soll standardmäßig das Bildformat *.jpg* ausgewählt sein.

Sie können Methoden auch direkt über den Code mit einem Ereignis verdrahten. Tippen Sie dazu den Methodenrumpf der Ereignisroutine und verknüpfen Sie diese mittels des Schlüsselworts `Handels` mit dem Ereignis Ihrer Wahl:

```
Private Sub MainWindow_Loaded(ByVal sender As Object,
                              ByVal e As System.Windows.RoutedEventArgs) Handles Me.Loaded

    End Sub
```

Diese Routine ist mit dem `Loaded`-Ereignis des Fensters verknüpft, was bedeutet, dass der Code ausgeführt wird, sobald das Fenster geladen ist – der ideale Zeitpunkt zum Voreinstellen einiger Werte.

HINWEIS Richtig geübte OOPler unter Ihnen werden übrigens vergeblich nach einer überschreibbaren Methode `OnLoaded` in der `Windows`-Klasse suchen. Wieso einige der ereignisauslösenden Methoden der `Windows`-Klasse (und ihrer Basisklassen – `Windows` ist von `FrameworkElement` abgeleitet) überschreibbar sind, und andere nicht, kann ich Ihnen auch nicht verraten. Tatsache ist, dass das `Loaded`-Ereignis nur über das Binden des eigentlichen Ereignisses erreichbar ist. Wichtig ist das zu wissen, wenn Sie es von Windows Forms gewohnt sind, sich beim Finden der Ereignisse auf die IntelliSense-Liste der überschreibbaren Methoden zu verlassen, die Sie erhalten, wenn Sie innerhalb einer Windows-Klasse **overrides** gefolgt von einem Leerzeichen in den Editor eintippen.

Übernehmen Sie weiterhin den oben stehenden Code, falls noch nicht geschehen, und vervollständigen Sie ihn wie folgt:

```
Class MainWindow

    Private Sub MainWindow_Loaded(ByVal sender As Object,
                                  ByVal e As System.Windows.RoutedEventArgs) Handles Me.Loaded

        'JPeg vorselektieren
        FormatComboBox.SelectedIndex = 0

        'Eigene Bilder + ImageConverter + Tagesdatum als Vorgabeverzeichnis bestimmen
        Dim myOutputFolder As String
        myOutputFolder = Environment.GetFolderPath(Environment.SpecialFolder.MyPictures)
        myOutputFolder &= "\ImageConverter\" & Now.ToString("yyyy-MM-dd")
        OutputFolderTextBox.Text = myOutputFolder
    End Sub

    Private Sub MenuItemBeenden_Click(ByVal sender As System.Object, ByVal e As
System.Windows.RoutedEventArgs)
        Me.Close()
    End Sub

End Class
```

28. Starten Sie das Programm mit F5, und der Pfad zu Ihrem Bilderordner sollte sich in der entsprechenden `TextBox` vorfinden (Abbildung 6.39).

Abbildung 6.39 Beim Laden des Fensters wird das Bildformat vorselektiert und der Ausgabeordner auf den benutzereigenen Bilderordner voreingestellt

Organisation ist alles: den Ausgabepfad frei definierbar machen

Falls der voreingestellte Ausgabeordner zugunsten von Übersicht und Ordnung geändert werden soll, muss ein geeigneter und benutzerfreundlicher Dialog her. Dieser Dialog wird uns von .NET Framework zur Verfügung gestellt, sodass der Aufwand auf ein paar Einstellungen beschränkt wird.

29. Verknüpfen Sie das `Click`-Ereignis der Schaltfläche *ChooseOutputFolderButton* mit einer entsprechenden Routine. Auf welche Weise, ob per `Handles`-Schlüsselwort oder via XAML-Code, das ist Ihnen überlassen. Im XAML-Code sieht das so aus:

```
<Button Name="ChooseOutputFolderButton" … Click="ChooseOutputFolderButton_Click"></Button>
```

Die Klasse, die den Dialog zur Verfügung stellt, heißt `FolderBrowserDialog`. Um sie instanziieren und verwenden zu können, muss allerdings ein wenig Vorarbeit geleistet werden, denn den Dialog gibt es (noch) nicht in der WPF-Funktionsbibliothek. Er ist aber unter Windows Forms verfügbar, und es ist durchaus legitim, uns Funktionalitäten für unsere Benutzeroberfläche bei den Windows Forms-Jungs »auszuborgen«. Und das geht so:

30. Klicken Sie zu diesem Zweck mit der rechten Maustaste auf das Projekt *ImageResizer* (nicht auf die Projektmappe!) und wählen Sie *Verweis hinzufügen* (Abbildung 6.40).

Abbildung 6.40 Um einen Verweis hinzuzufügen, klicken Sie mit der rechten Maustaste auf das Projekt und wählen den entsprechenden Menüeintrag aus

31. Suchen Sie nun im sich öffnenden Dialog auf der Registerkarte *.NET* die Assembly *System.Windows.Forms* (Abbildung 6.41). Wählen Sie diese aus, und bestätigen Sie den Dialog mit *OK*.

Abbildung 6.41 Binden Sie den Assembly-Verweis auf *System.Windows.Forms* ein

Zurück im Code des Fensters *MainWindow* sollen Klassen aus der Assembly verwendet werden, die wir gerade eingebunden haben. Um das möglich zu machen, tippen Sie folgenden Code ein. Dabei wichtig: Er muss noch vor der Klassendefinition stehen!

```
Imports System.Windows.Forms

Class MainWindow
        …
End Class
```

Da die Klasse, die uns die Ordnerauswahl auf dem Bildschirm anzeigen lässt, nun erreichbar ist, können wir Sie mit entsprechendem Code instanziieren und anschließend verwenden. Wir erlauben dem Benutzer, mit dem Wert der Eigenschaft `ShowNewFolderButton` neue Ordner im Dialog anzulegen, und geben dem Dialog über die Eigenschaft `Description` den Titel *Zielverzeichnis wählen*. Der Dialog wird mit der Methode `ShowDialog` aufgerufen und das Programm wartet so lange mit der Ausführung des weiteren Codes, bis der Dialog ein Ergebnis erzielt hat: Er wurde entweder abgebrochen oder bestätigt. Wird der Dialog mit *OK* bestätigt, ist der neue Ordner gewählt, und der Pfad kann im entsprechenden Eingabefeld angezeigt werden. Den Code dazu sehen Sie im Folgenden:

```
  Private Sub ChooseOutputFolderButton_Click(ByVal sender As System.Object,
                                             ByVal e As System.Windows.RoutedEventArgs)

        Dim folderBrowser = New FolderBrowserDialog
        folderBrowser.ShowNewFolderButton = True
        folderBrowser.Description = "Zielverzeichnis wählen"
        Dim dr = folderBrowser.ShowDialog
        If dr = Forms.DialogResult.OK Then
            OutputFolderTextBox.Text = folderBrowser.SelectedPath
        End If
    End Sub
```

Starten Sie einen Testlauf via F5.

Letzte Vorbereitungen treffen: Bilder hinzufügen

Die Bühne steht, doch die Protagonisten fehlen: die Bilder, die bearbeitet werden sollen. Beim Klicken auf die Schaltfläche soll ein Dialog das Hinzufügen der Bilder ermöglichen. Verdrahten Sie als Erstes das `Click`-Ereignis der Schaltfläche *AddImagesButton* mit einer Methode. Dort wird der altbekannte Datei-Auswahl-Dialog aufgerufen. Da dieser in zwei Versionen in verschiedenen Bibliotheken zur Verfügung steht, nämlich in Win32 und in Forms, muss er voll referenziert werden, um Verwechslungen auszuschließen:

```
Private Sub AddImagesButton_Click(ByVal sender As System.Object, _
                                  ByVal e As System.Windows.RoutedEventArgs) _
Handles AddImagesButton.Click

        'Wir verwenden den OpenFileDialog aus Win32, weil er für WPF "gemacht" ist -
        'müssen ihn aber voll qualifizieren, weil es ihn in .Forms und .Win32 gibt.
        Dim fod = New Microsoft.Win32.OpenFileDialog
    End Sub
```

Beim Aufrufen des Dialogs sind einige Einstellungen für den Programmablauf wichtig. Damit das Programm ordnungsgemäß laufen kann, müssen die Dateipfade und Dateien existieren sowie die Dateinamen gültig sein und vom Betriebssystem akzeptiert werden. Den Code dafür sehen Sie im Folgenden:

```
'nur existierende Dateien und Pfade dürfen gewählt werden.
fod.CheckPathExists = True
fod.CheckFileExists = True

'nur gültige Win32 Dateinamen werden akzeptiert
fod.ValidateNames = True
```

Um dem Benutzer die Handhabung angenehm zu gestalten, soll er mehrere Bilder gleichzeitig auswählen können:

```
'es dürfen mehrere Dateien ausgewählt werden
fod.Multiselect = True
```

Der Filter des Dialogs wird mit Bilddateitypen bestückt und standardmäßig soll der Dateityp *.jpg* ausgewählt sein:

```
'der Filter soll nach Bilddateitypen filtern
fod.Filter = "JPeg-Dateien (*.jpg; *.jpeg)|*.jpg; *.jpeg" &
            "|Bitmap-Dateien (*.bmp)|*.bmp" &
            "|TIFF-Dateien (*.tif;*.tiff)|*.tif;*.tiff" &
            "|PNG-Dateien (*.png)|*.png" &
            "|Alle Dateien (*.*)|*.*"

'der Filtert steht standardmäßig auf "*.jpg"
fod.DefaultExt = "*.jpg"
```

Ist der Dialog nun bestätigt, müssen die Pfade der ausgewählten Dateien als Elemente in die `ListBox`. Im nachfolgenden Code entdecken Sie ein neues Feature von Visual Basic 2010: mehrzeilige Anweisungs-Lambdas. Mehr zu diesen nützlichen neuen Helfern erfahren Sie übrigens in Kapitel 18. Hier wird für jedes Element in der Liste der gewählten Dateinamen die kleine nachfolgende Routine ausgeführt: Das Element wird der Elementauflistung der `ListBox` hinzugefügt:

```
'Den DialogResult des Dialogs speichern
Dim dr = fod.ShowDialog
'und auf Erfolg überprüfen
If dr = True Then
    Array.ForEach(fod.FileNames, Sub(filename)
                                     FilenamesListbox.Items.Add(filename)
                                 End Sub)
End If
```

Endspurt: das Verkleinern der Bilder

Nun stehen alle zur Ausführung benötigten Informationen zur Verfügung, und die eigentliche Bearbeitung der Bilder kann beginnen. Dazu macht das Programm folgende Schritte:

- Einlesen der Informationen
- Bearbeiten der Bilder
- Speichern der Bilder
- Anzeigen des Fortschritts für den Benutzer

Hier gibt es einen Haken: In den WPF-Libraries existiert, anders als in Windows Forms, keine Bild-Klasse, mit der man Bilder speichern könnte. Also, eigentlich jedenfalls nicht. Mit diesem Buch haben Sie gleichzeitig auch einen voll funktionsfähigen Workaround erworben, der es Ihnen ermöglicht, auch in WPF Bilder zu speichern. Die vom Autor geschriebene Bibliothek mit dieser und einigen anderen Funktionalitäten finden Sie begleitend im Unterverzeichnis *WriteableBitmapManager* der Projektmappe dieses Beispiels (*ImageResizer*). Sie haben natürlich die Möglichkeit, die Funktionalitäten daraus auch in Ihren anderen Projekten zu verwenden.

Sie werden darüber hinaus auch mit dem Nachvollziehen der folgenden Schritte lernen, wie Sie Programmbibliotheken einer Projektmappe prinzipiell hinzufügen und die entsprechenden Referenzen setzen, damit Sie sie von Ihren Projekten heraus verwenden können.

Hinzufügen eines Projekts zu einer bestehenden Projektmappe und Setzen der Verweise

32. Kopieren Sie das Projekt *WriteableBitmapManager* in ein Verzeichnis Ihrer Wahl, beispielsweise in das Projektverzeichnis des *ImageResizer*, das Sie standardmäßig in Ihrem Windows-Dokumenteordner im Verzeichnis *Visual Studio 2010\Projects* finden.

33. Wechseln Sie zu Visual Studio und klicken Sie im Projektmappen-Explorer die Projektmappe *ImageResizer* mit der rechten Maustaste an.

34. Klicken Sie auf *Hinzufügen/Vorhandenes Projekt* (Abbildung 6.42).

Abbildung 6.42 Hinzufügen eines vorhandenen Projekts zur Projektmappe

35. Navigieren Sie zu dem entsprechenden Ordner, wählen Sie die Projektdatei des Projekts *WriteableBitmapManager*, und bestätigen Sie den Dialog mit *Öffnen* (Abbildung 6.43).

Abbildung 6.43 Wählen Sie die Projektdatei des *WriteableBitmapManager*

Das Projekt befindet sich nun in der Projektmappe.

36. Verfahren Sie wie oben beschrieben, um dem Projekt *ImageResizer* einen Verweis auf das Projekt *WriteableBitmapManager* hinzuzufügen. Wählen Sie dafür im Dialog die Registerkarte *Projekte* und das Projekt *WriteableBitmapManager* und klicken Sie *OK*.

Abbildung 6.44 Einen Verweis auf das Projekt hinzufügen

37. Ergänzen Sie den Code im Fenster *MainWindow* um folgende (fett gedruckte) Zeile:

```
Imports System.Windows.Forms
Imports ActiveDevelop.Wpf.Imaging
Class MainWindow
…
End Class
```

Das Herzstück des Programms: die Funktionalität des Verkleinerns

Der Vorgang des Verkleinerns wird durch die Schaltfläche *StartButton* ausgelöst. Verdrahten Sie den `Click`-Event der Schaltfläche mit einer Methode. Für das Verkleinern der Bilder sind erst einmal folgende Informationen wichtig:

- Wie groß ist das neue Bild?
- Welchen Dateityp soll es erhalten?
- Welche Bilder sollen verkleinert werden?

Die Informationen dazu werden aus den Eingaben der Benutzer ermittelt. Geben Sie folgenden Code in den Methodenrumpf ein:

```
Dim myReduceToX = Integer.Parse(X_ResTextBox.Text)
Dim newExtension = DirectCast(FormatComboBox.SelectedItem, ComboBoxItem).Content.ToString
```

Initialisieren Sie die Fortschrittsanzeige in der `ProgressBar`, indem Sie den maximalen Fortschritt ermitteln und einen Zähler schaffen, der den bisher gemachten Fortschritt zwischenspeichert:

```
Me.TotalProgressBar.Maximum = FilenamesListbox.Items.Count
Dim progressCount = 0
```

Die folgenden Schritte müssen für jedes Bild ausgeführt werden:

- Den Namen des gerade bearbeiteten Bildes auf der Oberfläche anzeigen
- Den Dateinamen des Bildes in einer neuen Instanz der `FileInfo` Klasse speichern, die später Funktionalitäten zum Verwalten bzw. Bearbeiten zur Verfügung stellt
- Die Klasse `WriteableBitmapManager` instanziieren und anweisen, eine dem Bild entsprechende Kopie anzulegen
- Errechnen des Faktors, um den das Bild verkleinert werden soll
- Die Kopie des Bildes um den errechneten Faktor verkleinern, dabei die Zeit stoppen
- Umbenennen des Bildes entsprechend der Auswahl des Dateityps
- Speichern des Bildes mithilfe der `WriteableBitmapManager`-Instanz im angegebenen Ordner mit dem angegebenen Postfix
- Den Fortschrittszähler um eins erhöhen und den aktuellen Fortschritt in der `ProgressBar` anzeigen

Um die Funktionalität der `FileInfo`-Klasse nutzen zu können, muss noch folgender Namespace importiert werden:

```
Imports System.IO
```

Fügen Sie dem Projekt *ImageResizer* einen weiteren Verweis hinzu, indem Sie das Projekt mit der rechten Maustaste anklicken, *Verweis hinzufügen* auswählen und auf der Registerkarte *.NET* die Bibliothek *System.Drawing* auswählen. Umgesetzt in Code sieht die Prozedur nun so aus:

HINWEIS Der folgende Code, der sich in der `StartButton_Click`-Methode befindet, ist vielleicht ein wenig zu viel zum Abtippen. Wenn Sie Ihr Projekt bis hierhin nachgebaut haben, können Sie sich die Tipparbeit im Folgenden sparen und im Beispielverzeichnis dieses Kapitels die Textdatei *ImageResizer_SButtonClick.txt* öffnen, die den Code vollständig enthält und diesen von dort aus in die Ereignismethode kopieren.

```
Private Sub StartButton_Click(ByVal sender As System.Object, ByVal e As System.Windows.RoutedEventArgs)

        'neue Breite einlesen:
        Dim myReduceToX = Integer.Parse(X_ResTextBox.Text)
        'Dateityp einlesen
        Dim newExtension = DirectCast(FormatComboBox.SelectedItem, ComboBoxItem).Content.ToString

        'Zahl der Berechnungen ermitteln und der Fortschrittsanzeige zuweisen
        Me.TotalProgressBar.Maximum = FilenamesListBox.Items.Count
        'Zähler für bisher fertig gestellte Berechnungen = Fortschritt
        Dim progressCount = 0

        'Für jedes Bild ..
        For Each filenameItem As String In FilenamesListBox.Items
            'den Dateinamen anzeigen
            CurrentFileTextBlock.Text = filenameItem
            'und speichern
            Dim fileInfo As New FileInfo(filenameItem)

            'den Manager instanziieren und eine Kopie des Bildes anfertigen
            Dim currentPicture As New WriteableBitmapManager(filenameItem)
            Dim mySmallWbm = currentPicture.CreateCompatible(myReduceToX)
            'den Verkleinerungsfaktor errechnen
            Dim pixelFakt As Double = currentPicture.WriteableBitmap.PixelWidth / myReduceToX

            'Stoppuhr anschmeißen
            Dim sw = Stopwatch.StartNew
            'Das Bild verkleinern
            For y = 0 To mySmallWbm.WriteableBitmap.PixelHeight - 1
                For x = 0 To myReduceToX - 1
                    mySmallWbm.SetPixel(x, y,
                        currentPicture.GetPixel(CInt(Math.Truncate(pixelFakt * x)),
                                                CInt(Math.Truncate(pixelFakt * y))))
                Next
            Next

            'und aktualisieren
            mySmallWbm.UpdateBitmap()
            'Stoppuhr stoppen
            sw.Stop()

            'Den Dateityp anpassen
            Dim newFilename = fileInfo.Name.Replace(fileInfo.Extension, "")
            'den Ordner und das Postfix einbauen
```

```
            newFilename = OutputFolderTextBox.Text &
                          "\" & newFilename & PostfixTextBox.Text & newExtension

            'Speichern
            mySmallWbm.SaveBitmap(newFilename)

            'und den Fortschritt anzeigen
            progressCount += 1
            TotalProgressBar.Value = progressCount

        Next
    End Sub
```

Starten Sie mit F5 einen Testlauf des Programms.

Von der Holz- zur Businessklasse

Nach dem Verkleinern der Bilder mit dem Programm kann ich ohne hellseherische Fähigkeiten mit großer Wahrscheinlichkeit behaupten, dass Sie den Ausgabeordner gesucht haben und sich die verkleinerten Bilder angeschaut haben.

Um dem Programm den letzten Schliff an Bedienkomfort zu geben, wäre es nicht schön, die kleinen Bilder sofort als Vorschau zu sehen? Kein Problem. Klicken Sie zunächst mit der rechten Maustaste auf das Projekt *ImageResizer*, wählen Sie *Hinzufügen* und klicken Sie auf *Neues Element* (Abbildung 6.45).

Abbildung 6.45 Ein neues Element hinzufügen

Wählen Sie im daraufhin angezeigten Dialog den Eintrag *Fenster (WPF)* und geben Sie ihm den Namen **BitmapViewer.xaml** (Abbildung 6.46). Bestätigen Sie den Dialog mit *Hinzufügen*.

Abbildung 6.46 Ein neues Fenster hinzufügen

Das neue Fenster braucht nicht viel an Oberfläche. Fügen Sie in das `Grid` ein Steuerelement des Typs `ScrollViewer` ein und platzieren Sie darin ein `Image`. Setzen Sie die Eigenschaft `Stretch` des `Image` auf *None*, damit das Bild in seiner Originalgröße angezeigt wird, und geben Sie ihm den Namen **PreviewImage.**

Ziehen Sie das Fenster größer, bis es Ihnen groß genug erscheint, und setzen Sie die Startposition auf die Mitte des Bildschirms mithilfe der Eigenschaft `WindowStartupLocation` und dem Wert *CenterScreen*:

```
<Window x:Class="BitmapViewer"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="BitmapViewer" Height="448" Width="656" WindowStartupLocation="CenterScreen">
    <Grid>
        <ScrollViewer Name="ScrollViewer1" HorizontalScrollBarVisibility="Visible">
            <Image Name="PreviewImage" Stretch="None" />
        </ScrollViewer>
    </Grid>
</Window>
```

Wechseln Sie nun in die Code Behind-Datei, fügen Sie eine Methode namens *ShowPicture* ein, die als Übergabeparameter eine Variable vom Typ `WriteableBitmapManager` entgegennimmt sowie eine Variable vom Typ `String`. Implementieren Sie folgenden Code, um das Vorschaufenster zu komplettieren:

```
Public Sub ShowPicture(ByVal wbm As WriteableBitmapManager, ByVal statusText As String)
        PreviewImage.Width = wbm.WriteableBitmap.Width
        PreviewImage.Height = wbm.WriteableBitmap.Height
        PreviewImage.Source = wbm.WriteableBitmap
        Me.Title = statusText
    End Sub
```

Auch in diesem Fenster fehlt noch der Namespace für die Klasse `WriteableBitmapManager`.

TIPP Um fehlende Namespaces zu importieren, können Sie sich auch der Fehlerkorrektur bedienen, wie in Abbildung 6.47 zu sehen. Das erspart Ihnen nicht nur das Scrollen zum Anfang der Codedatei, sondern auch das Suchen nach dem entsprechenden Namespace-Namen, den es zu importieren gilt.

Abbildung 6.47 Visual Studio macht Ihnen Verbesserungsvorschläge bei auftretenden Fehlern, zum Beispiel bei fehlenden Verweisen

Fügen Sie die folgenden (fett gedruckten) Codezeilen in die Verkleinerungsroutine ein, um das Fenster aufzurufen, die einzelnen Bilder anzuzeigen und das Fenster wieder zu schließen:

```
    Private Sub StartButton_Click(ByVal sender As System.Object, ByVal e As _
    System.Windows.RoutedEventArgs)

        'Das Vorschaufenster instanziieren und anzeigen
        Dim picViewer As New BitmapViewer
        picViewer.Show()

        'neue Breite einlesen:
        Dim myReduceToX = Integer.Parse(X_ResTextBox.Text)
        'Dateityp einlesen
                …

        'Für jedes Bild ..
        For Each filenameItem As String In FilenamesListBox.Items
            'den Dateinamen anzeigen
            CurrentFileTextBlock.Text = filenameItem
```

```
                    …
        'und aktualisieren
        mySmallWbm.UpdateBitmap()
        'Stoppuhr stoppen
        sw.Stop()

        'das verkleinerte Bild anzeigen
        picViewer.ShowPicture(mySmallWbm, "Dauer der Konvertierung in ms: " &
                    sw.ElapsedMilliseconds.ToString("#,##0.00"))


            …
    Next

    picViewer.Close()

End Sub
```

Wenn Sie das Programm nun starten, werden Sie feststellen, dass das Vorschaufenster zwar ordnungsgemäß in seiner Größe vorhanden ist, jedoch als einziges Bild das als zuletzt konvertierte angezeigt wird. Wie kommt das?

Da das ganze Programm einzig und allein auf einem einzigen `Thread` läuft, ist es während der Berechnung so beschäftigt, dass es die Oberfläche vernachlässigt und keine Kapazität zum Aktualisieren der Oberfläche freigibt. Wenn man die Berechnung nun auf einen anderen `Thread` verlegt, wäre die Kapazität zum Aktualisieren der Oberfläche frei, doch das soll noch nicht Bestandteil dieses Kapitels sein: Ausführliche Informationen und Erklärungen über das Thema Threading finden Sie in Kapitel 29.

Fürs Erste lösen wir das Problem, indem wir gezielt die Windows-Nachrichtenschleife triggern, ihr also die Möglichkeit geben, aufgelaufene Nachrichten »mal eben« zu verarbeiten, obwohl wir ansonsten die gesamte Rechenleistung des UI-Threads blockieren. Nehmen Sie folgende Änderung vor:

```
            …
        picViewer.ShowPicture(mySmallWbm, "Dauer der Konvertierung in ms: " &
                    sw.ElapsedMilliseconds.ToString("#,##0.00"))

        'Böser Workaround - mehr dazu im Threading-Kapitel:
        System.Windows.Forms.Application.DoEvents()
            …
    Next

    picViewer.Close()

End Sub
```

Wie beim Speichern eines Bildes gibt es dafür auch bei WPF leider keine explizite Funktionalität, wohl auch, da das externe Triggern der Nachrichtenwarteschleife eher unter »Improvisation« und »Bitte, liebe Kinder, bitte nicht zuhause nachmachen« fällt. Wir leihen uns an dieser Stelle wieder eine Funktion aus der Windows Forms-Library, nämlich die `DoEvents`-Methode der `Application`-Klasse. Die Threading-Variante in Kapitel 29 zeigt dann später, wie es wirklich geht.

Kapitel 7

Die essenziellen .NET-Datentypen

In diesem Kapitel:

Numerische Datentypen 256
Der Datentyp Char 278
Der Datentyp String 279
Der Datentyp Boolean 297
Der Datentyp Date 298
.NET-Äquivalente primitiver Datentypen 306
Konstanten und Nur-Lesen-Felder (Read-only-Feldvariablen) 310

Sie wissen nun spätestens seit Kapitel 2, was Variablen sind und dass es unterschiedliche Typen gibt. Ein paar der .NET-Datentypen haben Sie in den dortigen Beispielen auch schon kennen gelernt. Allerdings fehlen bislang immer noch genaue Informationen über das Konzept einiger spezieller Datentypen, die in .NET fest integriert sind und die Sie nahezu ständig beim Entwickeln eigener Applikationen verwenden (müssen).

Das sind vor allem die schon erwähnten primitiven Datentypen wie `Integer`, `Double`, `Date`, `String` etc. Sie sind in den .NET-Sprachen von C# oder Visual Basic fest verankert, was Sie im Übrigen auch daran feststellen können, dass der Editor sie blau markiert, wenn Sie die Schlüsselwörter dieser Typen ausgeschrieben haben.

Unter primitive Datentypen fallen alle Typen, die in einer Programmiersprache unter .NET fest im Sprachschatz verankert sind. Das heißt genauer:

- **Konstanter Wert:** Ein konstanter Wert jedes primitiven Datentyps kann direkt angegeben werden. Die Angabe `123.324D` bezeichnet so beispielsweise einen Wert bestimmter Größe vom Typ `Decimal`.
- **Primitiver Datentyp als Konstante:** Es ist möglich, einen primitiven Datentyp als Konstante zu deklarieren. Wenn ein bestimmter Ausdruck ausschließlich als Konstante definiert wird (also beispielsweise beim Ausdruck `123.32D*2+100.23D`), kann er schon beim Kompilieren ausgewertet werden.
- **Funktionsdelegierung an den Prozessor:** Viele Operationen und Funktionen bestimmter primitiver Datentypen können von .NET Framework zur Ausführung direkt an den Prozessor delegiert werden. Dann ist keine Programmlogik erforderlich, um einen arithmetischen Ausdruck zu berechnen (beispielsweise eine Fließkommazahlendivision), der Prozessor kann das vielmehr selbstständig und damit natürlich sehr schnell. Dazu gehören die meisten der Operationen der Datentypen `Byte`, `Short`, `Integer`, `Long`, `Single`, `Double` und `Boolean`.

Numerische Datentypen

Für die Verarbeitung von Zahlen bietet Ihnen Visual Basic die primitiven Datentypen `Byte`, `Short`, `Integer`, `Long`, `Single`, `Double` und `Decimal` an. Neu seit Visual Basic 2005 waren darüber hinaus die Datentypen `SByte`, `UShort`, `UInteger` und `ULong`. Sie unterscheiden sich durch den Wertebereich, den sie abdecken können, die Präzision, mit der sie rechnen (Anzahl Nachkommastellen – auch Skalierung genannt) und den Speicherbedarf, den sie benötigen.

HINWEIS Die so genannten `Nullable`-Datentypen sind ebenfalls seit Visual Basic 2005 bekannt, und Sie haben sie schon kurz in der Einführung in Kapitel 2 kennen gelernt. `Nullable`-Datentypen verhalten sich wie ihre primitiven Datentypenpendants, haben aber auch die Möglichkeit, einen nicht definierten Zustand, nämlich *null*[1], *nichts* oder in Visual Basic `Nothing` widerzuspiegeln). Seit Visual Basic 2008 gliedern sich diese Datentypen auch in die Sprachsyntax von Visual Basic ein, und sie werden mit dem Fragezeichen als Typliteral definiert. Um einen `Integer`-Datentyp also als `Nullable` zu definieren, würde man wie folgt vorgehen:

```
Dim t As Integer?
```

Mehr zum Thema `Nullable`-Datentypen finden Sie in Kapitel 19.

[1] Nicht mit dem Wert *0* verwechseln! Null (englisch ausgesprochen *Nall*) kommt damit eher dem »Zustand« *nichts* nahe. Im Englischen würde man die Zahl 0 auch nicht als engl. *Null* (*Nall*), sondern eher als Zero oder Ought (*Oht* oder oft auch einfach *Oh* ausgesprochen) bezeichnen.

Numerische Datentypen deklarieren und definieren

Alle numerischen Datentypen werden wie alle primitiven Datentypen ohne das Schlüsselwort `New` deklariert; Zuweisungen von konstanten Werten können direkt im Programm erfolgen, wobei es bestimmte Markierungszeichen gibt, von welchem Typ ein konstanter Wert ist, der durch eine Ziffernfolge angegeben wird. Eine Variable vom Typ `Double` kann beispielsweise mit der Anweisung

```
Dim locDouble As Double
```

deklariert und sofort verwendet werden. Numerische Datentypen werden mit im Programmcode verankerten Konstanten definiert, indem Sie ihnen eine Ziffernfolge zuweisen, der im Bedarfsfall das Typliteral folgt, wie im folgenden Beispiel das `D` hinter der eigentlichen Konstanten:

```
locDouble = 123.3D
```

Genau wie bei anderen primitiven Datentypen können Deklaration und Zuweisung in einer Anweisung erfolgen. So lassen sich natürlich die beiden oben stehenden einzelnen Anweisungen durch die folgende ersetzen:

```
Dim locDouble As Double = 123.3D
```

Das hier gezeigte Beispiel gilt für alle anderen numerischen Datentypen äquivalent – wobei sich die Typliterale natürlich von Typ zu Typ unterscheiden können.

Typenname	Typzeichen	Typliteral	Beispiel
Byte	–	–	Dim var As Byte = 128
SByte	–	–	Dim var As SByte = -5
Short	–	S	Dim var As Short = -32700S
UShort	–	US	Dim var As UShort = 65000US
Integer	%	I	Dim var% = -123I oder Dim var As Integer = -123I
UInteger	–	UI	Dim var As UInteger = 123UI
Long	&	L	Dim var& = -123123L oder Dim var As Long = -123123L
ULong	–	UL	Dim var As ULong = 123123UL
Single	!	F	Dim var! = 123.4F oder Dim var As Single = 123.4F
Double	#	R	Dim var# = 123.456789R oder Dim var As Double = 123.456789R
Decimal	@	D	Dim var@ = 123.456789123D oder Dim var As Decimal = 123.456789123D
Boolean	–	–	Dim var As Boolean = True
Char	–	C	Dim var As Char = "A"C ▶

Typenname	Typzeichen	Typliteral	Beispiel
Date	–	#dd/MM/yyyy HH:mm:ss# oder #dd/mm/yyyy hh:mm:ss am/pm#	Dim var As Date = #12/24/2008 04:30:15 PM#
Object	–	–	In einer Variable vom Typ Object kann jeder beliebige Typ gekapselt (»geboxt«) oder mit dieser referenziert werden
String	$	"Zeichenkette"	Dim var$ = "Zeichenkette" oder Dim var As String = "Zeichenkette"

Tabelle 7.1 Typliterale und Variablentypzeichen der primitiven Datentypen in Visual Basic 2010

Delegation numerischer Berechnungen an den Prozessor

Die Eigenschaft, einige mathematische Operationen dem Prozessor direkt zu überlassen, zeigt das folgende Beispiel eindrucksvoll. Dazu müssen Sie wissen: Der `Decimal`-Typ wird, anders als `Double` oder `Single`, wegen seiner Rechengenauigkeit nicht alleine durch die Fließkommaeinheit des Prozessors, sondern durch entsprechenden Programmcode der Base Class Library berechnet.

HINWEIS Das folgende Beispiel steigt ein wenig tiefer ins System ein – Sie lernen dabei auch, wie Sie die Assembler- bzw. Maschinensprachenrepräsentation, also das konkrete Kompilat Ihres Programms, in der Form, wie es der Prozessor sieht, betrachten können. Interessant ist das allemal, und es hilft auch beim Verständnis, wie Daten vom Prozessor verarbeitet werden – notwendiges Wissen zum Entwickeln eigener Programme ist es allerdings nicht wirklich.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 07\Primitives01
```

Öffnen Sie dort die Projektmappe (.SLN-Datei) *Primitives01*.

Bevor Sie dann den folgenden Beispielcode ausführen, setzen Sie mit der Taste F9 einen Haltepunkt in der folgenden, fett markierten Codezeile:

```
Public Class Primitives
    Public Shared Sub main()
        Dim locDouble1, locDouble2 As Double
        Dim locDec1, locDec2 As Decimal

        locDouble1 = 123.434D
        locDouble2 = 321.121D
        locDouble2 += 1
        locDouble1 += locDouble2
        Console.WriteLine("Ergebnis der Double-Berechnung: {0}", locDouble1)

        locDec1 = 123.434D
        locDec2 = 321.121D
        locDec2 += 1
        locDec1 += locDec2
        Console.WriteLine("Ergebnis der Decimal-Berechnung: {0}", locDec1)

    End Sub
End Class
```

Wenn Sie dieses Programm anschließend starten, unterbricht das Programm in der Zeile, in der Sie zuvor den Haltepunkt gesetzt haben. Wählen Sie aus dem Menü *Debuggen/Fenster* den Punkt *Disassembly*. Mithilfe dieses Fensters können Sie sehen, was der JITter[2] aus dem zunächst in die IML kompilierten Programm gemacht hat:

```
       Dim locDouble1, locDouble2 As Double
       Dim locDec1, locDec2 As Decimal

        locDouble1 = 123.434D
00000055  movsd       xmm0,mmword ptr [000002E8h]
0000005d  movsd       mmword ptr [rsp+50h],xmm0
        locDouble2 = 321.121D
00000063  movsd       xmm0,mmword ptr [000002F0h]
0000006b  movsd       mmword ptr [rsp+58h],xmm0
        locDouble2 += 1
00000071  movsd       xmm0,mmword ptr [000002F8h]
00000079  addsd       xmm0,mmword ptr [rsp+58h]
0000007f  movsd       mmword ptr [rsp+58h],xmm0
```

Hier werden, entgegen einer möglichen Erwartung, keine speziellen Methoden der `Double`-Struktur aufgerufen; die Addition wird vielmehr durch die Fließkommafunktionalität des Prozessors (`addsd`,[3] im Listing fett markiert) selbst erledigt. Ganz anders ist das weiter unten in der Disassembly, wenn die gleichen Operationen mit dem `Decimal`-Datentyp ausgeführt werden:

```
locDec2 += 1
0000017e  mov         rcx,129F1180h
00000188  mov         rcx,qword ptr [rcx]
0000018b  add         rcx,8
0000018f  mov         rax,qword ptr [rcx]
00000192  mov         qword ptr [rsp+000000A8h],rax
0000019a  mov         rax,qword ptr [rcx+8]
0000019e  mov         qword ptr [rsp+000000B0h],rax
000001a6  lea         rcx,[rsp+000000A8h]
000001ae  mov         rax,qword ptr [rcx]
000001b1  mov         qword ptr [rsp+000000E0h],rax
000001b9  mov         rax,qword ptr [rcx+8]
000001bd  mov         qword ptr [rsp+000000E8h],rax
000001c5  lea         rcx,[rsp+40h]
000001ca  mov         rax,qword ptr [rcx]
000001cd  mov         qword ptr [rsp+000000D0h],rax
000001d5  mov         rax,qword ptr [rcx+8]
000001d9  mov         qword ptr [rsp+000000D8h],rax
000001e1  lea         r8,[rsp+000000E0h]
000001e9  lea         rdx,[rsp+000000D0h]
000001f1  lea         rcx,[rsp+000000B8h]
```

[2] Das *Disassembly*-Fenster kann übrigens nur Assemblercode anzeigen, der nicht sehr gut optimiert ist. Daran ändert auch keine andere Einstellung etwas – Sie sehen immer nur den Debug- und nicht den optimierten Code. Im späteren optimierten Code werden, wann immer es möglich ist, die Prozessorregister als Träger lokaler Variablen genutzt, was die Laufgeschwindigkeit Ihrer Anwendungen natürlich drastisch erhöht.

[3] *Scalar Double-Precision Floating-Point Add*, in etwa: *Skalare Fließkommaaddition mit doppelter Genauigkeit*.

```
000001f9  call        FFFFFFFFEF381460                        // Hier wird die Additionsroutine von …
000001fe  mov         qword ptr [rsp+00000128h],rax
00000206  lea         rcx,[rsp+000000B8h]
0000020e  mov         rax,qword ptr [rcx]
00000211  mov         qword ptr [rsp+40h],rax
00000216  mov         rax,qword ptr [rcx+8]
0000021a  mov         qword ptr [rsp+48h],rax
        locDec1 += locDec2
0000021f  lea         rcx,[rsp+40h]
00000224  mov         rax,qword ptr [rcx]
00000227  mov         qword ptr [rsp+00000110h],rax
0000022f  mov         rax,qword ptr [rcx+8]
00000233  mov         qword ptr [rsp+00000118h],rax
0000023b  lea         rcx,[rsp+30h]
00000240  mov         rax,qword ptr [rcx]
00000243  mov         qword ptr [rsp+00000100h],rax
0000024b  mov         rax,qword ptr [rcx+8]
0000024f  mov         qword ptr [rsp+00000108h],rax
00000257  lea         r8,[rsp+00000110h]
0000025f  lea         rdx,[rsp+00000100h]
00000267  lea         rcx,[rsp+000000F0h]
0000026f  call        FFFFFFFFEF381460                      // … Decimal aufgerufen. Hier auch.
           .
           .
           .
```

Es sind hier ungleich mehr Vorbereitungen zur Addition notwendig, da die notwendigen Operanden zunächst auf den Stack kopiert werden müssen. Und hier wird die eigentliche Addition auch nicht durch den Prozessor selbst erledigt, sondern durch entsprechende Routinen der BCL, die, im Disassembly zu sehen, mit `Call` aufgerufen werden (im Listing fett markiert).

TIPP Das ist übrigens auch der Grund, weshalb die Performance des `Decimal`-Datentyps auch nur etwa einem Zehntel der Performance des `Double`-Datentyps entspricht. `Decimal` sollten Sie daher nur dann einsetzen, wenn Sie absolut genaue Berechnungen durchführen müssen und sich keine Rundungsfehler erlauben können (lesen Sie dazu bitte auch die Ausführungen im Abschnitt »Tabellarische Zusammenfassung der numerischen Datentypen« auf Seite 268).

Hinweis zur CLS-Konformität

Einige der in Visual Basic 2005 eingeführten Datentypen entsprechen nicht der so genannten CLS-Compliance.[4] Dazu gehören sowohl generische Datentypen als auch einige primitive Datentypen, die vorzeichenlose Integerwerte speichern (sowie der primitive Datentyp `SByte`). Methoden, die Typen benötigen oder zurückliefern, die als nicht CLS-konform gelten, sollten nicht offen gelegt werden, also nicht in Komponenten zur Verfügung gestellt werden, die auch andere .NET-Programmiersprachen verwenden können. Sie dürfen nämlich nicht davon ausgehen, dass diese Typen auch in allen .NET-Programmiersprachen »erreichbar« sind. Visual Basic prüft übrigens nicht automatisch auf CLS-Konformität. Falls Sie Komponenten vom Visual Basic-Compiler auf Konformität überprüfen lassen wollen, können Sie das so genannte `CLSCompliant`-Attribut auf Klassenebene einsetzen, also etwa so:

[4] Etwa: »Entsprechung der Common Language Specification«.

```
<CLSCompliant(True)> _
    Public Class EineKlasse

    Private myMember As UShort

    Public Property NichtCLSEntsprechend() As UShort
        Get
            Return myMember
        End Get
        Set(ByVal value As UShort)
            myMember = value
        End Set
    End Property
End Class
```

Falls Sie eine einzelne Methode auf CLS-Compliance überprüfen wollen, funktioniert das äquivalent:

```
    <CLSCompliant(True)> _
    Public Shared Function EineNichtCLSComplianceMethode() As UShort
        Dim locTest As ClassLibrary1.EineKlasse
        locTest = New ClassLibrary1.EineKlasse
    End Function
```

HINWEIS Entgegen vieler landläufiger Meinungen ist es nicht richtig, dass eine Methode, eine Assembly oder gar Ihre gesamte Anwendung nicht CLS-konform ist, wenn Sie nur einen nicht konformen Typen verwenden. Nicht CLS-konform wird Ihre Assembly nur dann, wenn Sie Typen offen legen, die nicht CLS-konform sind, also einen solchen Typ entweder von einer öffentlichen Methode mit Rückgabewert zurückliefern oder ihn als Parameter erwarten.

Die numerischen Datentypen auf einen Blick

Die Verwendung numerischer Datentypen und ihre darstellbaren Wertebereiche finden Sie in den folgenden kurzen Abschnitten beschrieben.

HINWEIS Der Punkt *Arithmetik durch den Prozessor* in der nachstehenden Aufstellung sagt aus, dass bestimmte arithmetische oder boolesche Operationen nicht durch Prozeduren von .NET Framework, sondern durch den Prozessor ausgeführt werden; das äußert sich in einer äußerst schnellen Verarbeitung.[5]

Byte

.NET-Datentyp: `System.Byte`

Stellt dar: Integerwerte (Zahlen ohne Nachkommastellen) im angegebenen Wertebereich

Wertebereich: 0 bis 255

[5] Diese Aussage gilt unter Umständen *nicht* für eine andere als die Intel Pentium-Plattform. Auf Pocket-PCs beispielsweise, auf denen Applikationen unter .NET ebenfalls entwickelt werden können, kann das Verhalten entsprechend unter Umständen ein anderes sein.

Typliteral: nicht vorhanden

Speicherbedarf: 1 Byte

Arithmetik durch den Prozessor: ja

Deklaration und Beispielzuweisung:

```
Dim einByte As Byte
einByte = 123
```

Anmerkung: Dieser Datentyp speichert nur vorzeichenlose, positive Zahlen im angegebenen Zahlenbereich

CLS-compliant: ja

Konvertierung von anderen Zahlentypen: `CByte(objVar)` oder `Convert.ToByte(objVar)`

```
einByte = CByte(123.45D)
einByte = Convert.ToByte(123.45D)
```

SByte

.NET-Datentyp: `System.SByte`

Stellt dar: Integerwerte (Zahlen ohne Nachkommastellen) im angegebenen Wertebereich

Wertebereich: –128 bis 127

Typliteral: nicht vorhanden

Speicherbedarf: 1 Byte

Arithmetik durch den Prozessor: ja

Deklaration und Beispielzuweisung:

```
Dim einByte As SByte
einByte = 123
```

Anmerkung: Dieser Datentyp speichert negative und positive Zahlen im angegebenen Zahlenbereich.

CLS-compliant: nein

Konvertierung von anderen Zahlentypen: `CSByte(objVar)` oder `Convert.ToSByte(objVar)`

```
einByte = CSByte(123.45D)
einByte = Convert.ToSByte(123.45D)
```

Short

.NET-Datentyp: `System.Int16`

Stellt dar: Integerwerte (Zahlen ohne Nachkommastellen) im angegebenen Wertebereich.

Wertebereich: –32.768 bis 32.767

Typliteral: S

Speicherbedarf: 2 Byte

Arithmetik durch den Prozessor: ja

Deklaration und Beispielzuweisung:

```
Dim einShort As Short
einShort = 123S
```

Anmerkung: Dieser Datentyp speichert vorzeichenbehaftete, also negative und positive Zahlen im angegebenen Zahlenbereich. Bei der Konvertierung in den Datentyp `Byte` kann durch den größeren Wertebereich von `Short` eine `OutOfRangeException` erzeugt werden.

CLS-compliant: ja

Konvertierung von anderen Zahlentypen: `CShort(objVar)` oder `Convert.ToInt16(objVar)`

```
'Nachkommastellen werden abgeschnitten
einShort = CShort(123.45D)
einShort = Convert.ToInt16(123.45D)
```

UShort

.NET-Datentyp: `System.UInt16`

Stellt dar: positive Integerwerte (Zahlen ohne Nachkommastellen) im angegebenen Wertebereich.

Wertebereich: 0 bis 65.535

Typliteral: US

Speicherbedarf: 2 Byte

Arithmetik durch den Prozessor : ja

Deklaration und Beispielzuweisung:

```
Dim einUShort As UShort
einUShort = 123US
```

Anmerkung: Dieser Datentyp speichert vorzeichenlose, also nur positive Zahlen im angegebenen Zahlenbereich. Bei der Konvertierung in den Datentyp `Byte` oder `Short` kann durch den (teilweise) größeren Wertebereich von `UShort` eine `OutOfRangeException` erzeugt werden.

CLS-compliant: nein

Konvertierung von anderen Zahlentypen: `CUShort(objVar)` oder `Convert.ToUInt16(objVar)`

```
'Nachkommastellen werden abgeschnitten
einUShort = CUShort(123.45D)
einUShort = Convert.ToUInt16(123.45D)
```

Integer

.NET-Datentyp: `System.Int32`

Stellt dar: Integerwerte (Zahlen ohne Nachkommastellen) im angegebenen Wertebereich.

Wertebereich: –2.147.483.648 bis 2.147.483.647

Typliteral: I

Speicherbedarf: 4 Byte

Arithmetik durch den Prozessor: ja

Deklaration und Beispielzuweisung:

```
Dim einInteger As Integer
Dim einAndererInteger%     ' auch als Integer deklariert
einInteger = 123I
```

Anmerkung: Dieser Datentyp speichert vorzeichenbehaftete, also negative und positive Zahlen im angegebenen Zahlenbereich. Bei der Konvertierung in die Datentypen `Byte`, `Short` und `UShort` kann durch den größeren Wertebereich von `Integer` eine `OutOfRangeException` erzeugt werden. Durch Anhängen des Zeichens »%« an eine Variable kann der `Integer`-Typ für die Variable erzwungen werden (darauf sollten Sie allerdings zugunsten eines besseren Programmierstils lieber verzichten).

CLS-compliant: ja

Konvertierung von anderen Zahlentypen: `CInt(objVar)` oder `Convert.ToInt32(objVar)`

```
einInteger = CInt(123.45D)
einInteger = Convert.ToInt32(123.45D)
```

UInteger

.NET-Datentyp: `System.UInt32`

Stellt dar: positive Integerwerte (Zahlen ohne Nachkommastellen) im angegebenen Wertebereich.

Wertebereich: 0 bis 4.294.967.295

Typliteral: UI

Speicherbedarf: 4 Byte

Arithmetik durch den Prozessor: ja

Deklaration und Beispielzuweisung:

```
Dim einUInteger As UInteger
einUInteger = 123UI
```

Anmerkung: Dieser Datentyp speichert vorzeichenlose, also nur positive Zahlen im angegebenen Zahlenbereich. Bei der Konvertierung in die Datentypen `Byte`, `Short`, `UShort` und `Integer` kann durch den (teilweise) größeren Wertebereich von `UInteger` eine `OutOfRangeException` erzeugt werden.

CLS-compliant: nein

Konvertierung von anderen Zahlentypen: `CUInt(objVar)` oder `Convert.ToUInt32(objVar)`

```
einUInteger = CUInt(123.45D)
einUInteger = Convert.ToUInt32(123.45D)
```

Long

.NET-Datentyp: `System.Int64`

Stellt dar: Integerwerte (Zahlen ohne Nachkommastellen) im angegebenen Wertebereich.

Wertebereich: –9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807

Typliteral: L

Speicherbedarf: 8 Byte

Arithmetik durch den Prozessor: ja

Deklaration und Beispielzuweisung:

```
Dim einLong As Long
Dim einAndererLong& ' auch als Long definiert
einLong = 123L
```

Anmerkung: Dieser Datentyp speichert vorzeichenbehaftete, also negative und positive Zahlen im angegebenen Zahlenbereich. Bei der Konvertierung in alle anderen Integerdatentypen kann durch den größeren Wertebereich von `Long` eine `OutOfRangeException` erzeugt werden. Durch Anhängen des Zeichens »&« an eine Variable kann der `Long`-Typ für die Variable erzwungen werden (darauf sollten Sie allerdings zugunsten eines besseren Programmierstils lieber verzichten).

CLS-compliant: ja

Konvertierung von anderen Zahlentypen: `CLng(objVar)` oder `Convert.ToInt64(objVar)`

```
einLong = CLng(123.45D)
einLong = Convert.ToInt64(123.45D)
```

ULong

.NET-Datentyp: `System.UInt64`

Stellt dar: positive Integerwerte (Zahlen ohne Nachkommastellen) im angegebenen Wertebereich.

Wertebereich: 0 bis 18.446.744.073.709.551.615

Typliteral: UL

Speicherbedarf: 8 Byte

Arithmetik durch den Prozessor: ja

Deklaration und Beispielzuweisung:

```
Dim einULong As ULong
einULong = 123L
```

Anmerkung: Dieser Datentyp speichert vorzeichenlose, also nur positive Zahlen im angegebenen Zahlenbereich. Bei der Konvertierung in alle anderen Integerdatentypen kann durch den größeren Wertebereich von `ULong` eine `OutOfRangeException` erzeugt werden.

CLS-compliant: nein

Konvertierung von anderen Zahlentypen: `CULng(objVar)` oder `Convert.ToUInt64(objVar)`

```
einULong = CULng(123.45D)
einULong = Convert.ToUInt64(123.45D)
```

Single

.NET-Datentyp: `System.Single`

Stellt dar: Fließkommawerte (Zahlen mit Nachkommastellen, deren Skalierung[6] mit Anwachsen des Werts kleiner wird) im angegebenen Wertebereich.

Wertebereich: Die Werte reichen von $-3{,}4028235*10^{38}$ bis $-1{,}401298*10^{-45}$ für negative Werte und von $1{,}401298*10^{-45}$ bis $3{,}4028235*10^{38}$ für positive Werte.

Typliteral: F

Speicherbedarf: 4 Byte

Arithmetik durch den Prozessor: ja

Deklaration und Beispielzuweisung:

```
Dim einSingle As Single
Dim einAndererSingle! ' auch als Single definiert
einSingle = 123.0F
```

Anmerkung: Dieser Datentyp speichert vorzeichenbehaftete, also negative und positive Zahlen im angegebenen Zahlenbereich. Durch Anhängen des Zeichens »!« an eine Variable kann der `Single`-Typ für die Variable erzwungen werden (darauf sollten Sie allerdings zugunsten eines besseren Programmierstils lieber verzichten).

CLS-compliant: ja

Konvertierung von anderen Zahlentypen: `CSng(objVar)` oder `Convert.ToSingle(objVar)`

```
einSingle = CSng(123.45D)
einSingle = Convert.ToSingle(123.45D)
```

[6] Skalierung in diesem Zusammenhang bezeichnet die Anzahl der Nachkommastellen einer Fließkommazahl.

Double

.NET-Datentyp: `System.Double`

Stellt dar: Fließkommawerte (Zahlen mit Nachkommastellen, deren Skalierung mit Anwachsen des Werts kleiner wird) im angegebenen Wertebereich.

Wertebereich: Die Werte reichen von $-1{,}79769313486231570*10^{308}$ bis $-4{,}94065645841246544*10^{-324}$ für negative Werte und von $4{,}94065645841246544*10^{-324}$ bis $1{,}79769313486231570^{308}$ für positive Werte

Typliteral: R

Speicherbedarf: 8 Byte

Arithmetik durch den Prozessor: ja

Deklaration und Beispielzuweisung:

```
Dim einDouble As Double
Dim einAndererDouble# ' auch als Double definiert
einDouble = 123.0R
```

Anmerkung: Dieser Datentyp speichert vorzeichenbehaftete, also negative und positive Zahlen im angegebenen Zahlenbereich. Durch Anhängen des Zeichens »#« an eine Variable kann der `Double`-Typ für die Variable erzwungen werden (darauf sollten Sie allerdings zugunsten eines besseren Programmierstils lieber verzichten).

CLS-compliant: ja

Konvertierung von anderen Zahlentypen: `CDbl(objVar)` oder `Convert.ToDouble(objVar)`

```
einDouble = CDbl(123.45D)
einDouble = Convert.ToDouble(123.45D)
```

Decimal

.NET-Datentyp: `System.Decimal`

Stellt dar: Fließkommawerte (Zahlen mit Nachkommastellen, deren Skalierung mit Anwachsen des Werts kleiner wird) im angegebenen Wertebereich.

Wertebereich: Der Wertebereich hängt von der Anzahl der verwendeten Dezimalstellen ab. Werden keine Dezimalstellen verwendet – man spricht dabei von einer Skalierung von 0 –, liegen die maximalen/minimalen Werte zwischen +–79.228.162.514.264.337.593.543.950.335. Bei der Verwendung einer maximalen Skalierung (28 Stellen hinter dem Komma – es können dann nur noch Werte zwischen > –1 und <+1 dargestellt werden) liegen die maximalen/minimalen Werte zwischen +–0,9999999999999999999999999999.

Typliteral: D

Speicherbedarf: 16 Byte

Arithmetik durch den Prozessor: nein

Deklaration und Beispielzuweisung:

```
Dim einDecimal As Decimal
Dim einAndererDouble@ ' auch als Decimal definiert
einDecimal = 123.23D
```

Anmerkung: Dieser Datentyp speichert vorzeichenbehaftete, also negative und positive Zahlen im angegebenen Zahlenbereich. Durch Anhängen des Zeichens »@« an eine Variable kann der `Decimal`-Typ für die Variable erzwungen werden (darauf sollten Sie allerdings zugunsten eines besseren Programmierstils lieber verzichten). Wichtig: Bei sehr hohen Werten müssen Sie das Typliteral an eine Literalkonstante anhängen, um eine `Overflow`-Fehlermeldung zu vermeiden.

HINWEIS Bitte beachten Sie ebenfalls, dass beim `Decimal`-Datentyp keine Delegation arithmetischer Funktionen an den Prozessor erfolgt, dieser Datentyp also im Vergleich zu den Fließkommadatentypen `Single` und `Double` sehr viel langsamer verarbeitet wird. Gleichzeitig treten aber keine Rundungsfehler durch die interne Darstellung von Werten im Binärsystem auf. Darüber erfahren Sie im übernächsten Abschnitt Näheres.

CLS-compliant: ja

Konvertierung von anderen Zahlentypen: `CDec(objVar)` oder `Convert.ToDecimal(objVar)`

```
einDecimal = CDec(123.45F)
einDecimal = Convert.ToDecimal(123.45F)
```

Tabellarische Zusammenfassung der numerischen Datentypen

Typenname	.NET-Typname	Aufgabe	Wertebereich
`Byte`	`System.Byte`	Speichert vorzeichenlose Integerwerte mit 8 Bit Breite	0 bis 255
`SByte`	`System.SByte`	Speichert vorzeichenbehaftete Integerwerte mit 8 Bit Breite	-127 bis 128
`Short`	`System.Int16`	Speichert vorzeichenbehaftete Integerwerte mit 16 Bit (2 Byte) Breite	-32.768 bis 32.767
`UShort`	`System.UInt16`	Speichert vorzeichenlose Integerwerte mit 16 Bit (2 Byte) Breite	0 bis 65.535
`Integer`	`System.Int32`	Speichert vorzeichenbehaftete Integerwerte mit 32 Bit (4 Byte) Breite HINWEIS: Auf 32-Bit-Systemen ist dies der am schnellsten verarbeitete Integerdatentyp	-2.147.483.648 bis 2.147.483.647
`UInteger`	`System.UInt32`	Speichert vorzeichenlose Integer-Werte mit 32 Bit (4 Byte) Breite	0 bis 4.294.967.295
`Long`	`System.Int64`	Speichert vorzeichenbehaftete Integerwerte mit 64 Bit (8 Byte) Breite	-9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807
`ULong`	`System.UInt64`	Speichert vorzeichenlose Integerwerte mit 64 Bit (8 Byte) Breite	0 bis 18.446.744.073.709.551.615
`Single`	`System.Single`	Speichert Fließkommazahlen mit einfacher Genauigkeit. Benötigt 4 Bytes zur Darstellung	-3,4028235E+38 bis -1,401298E-45 für negative Werte; 1,401298E-45 bis 3,4028235E+38 für positive Werte ▶

Typenname	.NET-Typname	Aufgabe	Wertebereich
`Double`	`System.Double`	Speichert Fließkommazahlen mit doppelter Genauigkeit. Benötigt 8 Bytes zur Darstellung. HINWEIS: Dies ist der schnellste Datentyp zur Fließkommazahlenberechnung, da er direkt an die Mathe-Einheit des Prozessors zur Berechnung delegiert wird.	1,79769313486231570E+308 bis -4,94065645841246544E-324 für negative Werte; 4,94065645841246544E-324 bis 1,79769313486231570E+308 für positive Werte
`Decimal`	`System.Decimal`	Speichert Fließkommazahlen im binärcodierten Dezimalformat. HINWEIS: Dies ist der langsamste Datentyp zur Fließkommazahlenberechnung, seine besondere Speicherform schließt aber typische Computerrundungsfehler aus.	0 bis +/- 79.228.162.514.264.337.593.543.950.335 (+/-7,9...E+28) ohne Dezimalzeichen; 0 bis +/-7,9228162514264337593543950335 mit 28 Stellen rechts vom Dezimalzeichen; kleinste Zahl ungleich 0 (null) ist +/-0,0000000000000000000000000001 (+/-1E-28)

Tabelle 7.2 Die numerischen, primitiven Datentypen in .NET 2.0/3.5/4.0 bzw. Visual Basic 2010

Rundungsfehler bei der Verwendung von Single und Double

Es ist eigentlich eine ganz normale Sache, dass ein bestimmtes Zahlensystem einige Brüche nicht genau darstellen kann. Dennoch gibt es immer wieder Programmierer, die glauben, einen Fehler in einer Programmiersprache gefunden zu haben, oder behaupten, der Computer könne nicht richtig rechnen. Dabei kennen Sie Rundungs- bzw. Konvertierungsfehler von einem Zahlensystem in das andere aus dem täglichen Leben auch beim 10er-System: Wenn Sie die Zahl 1 durch 3 teilen, erhalten Sie eine Zahl mit unendlich vielen Nachkommastellen, nämlich 0,333333333333… Im Dreiersystem ein Drittel darzustellen, benötigt wesentlich weniger Ziffern. Es ist schlicht 0,1.

Nun ist es ganz gleich, wie viele Ziffern Sie für die Darstellung eines für ein Zahlensystem problematischen Bruches verwenden; solange Sie eine endliche Anzahl von Ziffern in einem Zahlensystem verwenden, das einen Bruch nur periodisch darstellen kann, erhalten Sie beim Addieren dieser Zahlen Rundungsfehler.

Ein Beispiel: 3*1/3 im Dreiersystem führt zur Berechnung von:

```
 0.1
+0.1
+0,1
=========================================
+1,0
```

Und das entspricht im Dezimalsystem ebenfalls 1,0. Das Ausrechnen dieser Addition im Dezimalsystem ist ungenau, denn selbst wenn Sie über 60 Nachkommastellen für die Darstellung der Zahlen verwenden, so erreichen Sie in der Addition dennoch niemals den Wert 1:

```
 0,333333333333333333333333333333333333333333333333333333333333333333
+0.333333333333333333333333333333333333333333333333333333333333333333
+0.333333333333333333333333333333333333333333333333333333333333333333
=================================================================
 0.999999999999999999999999999999999999999999999999999999999999999999
```

Dieser Wert ist zwar ziemlich nah dran an 1, aber eben nicht ganz 1. Und wenn Sie mehrere Ergebnisse im Laufe einer Berechnung haben, können sich diese Darstellungsfehler schnell zu größeren Fehlern summieren, die auch irgendwann relevant werden.

Das gleiche Problem hat der Computer bei bestimmten Zahlen, wenn er im Binärsystem rechnet. Während wir beispielsweise die Zahl 69,82 im Dezimalsystem ganz genau mit einer endlichen Anzahl von Ziffern darstellen können, bekommt der Computer mit dem Binärsystem Probleme; die Umwandlung von 69 funktioniert noch einwandfrei, aber bei der 0,82 wird es schwierig:

Wenn Sie wissen, dass Nachkommastellen durch negative Potenzen der Basiszahl dargestellt werden, dann ergibt sich folgende Rechnung:

```
0.5             1*2^-1   Zwischenergebnis: 0.5
0.25            1*2^-2   Zwischenergebnis: 0.75
0.125           1*2^-3   Zwischenergebnis: 0.8125
0.0625          0*2^-4   Zwischenergebnis: 0,8125
0.03125         0*2^-5   Zwischenergebnis: 0,8125
0,015625        0*2^-6   Zwischenergebnis: 0,8125
0,0078125       0*2^-7   Zwischenergebnis: 0,8125
0,00390625      1*2^-8   Zwischenergebnis: 0,81640625
0,001953125     1*2^-9   Zwischenergebnis: 0,818359375
0,0009765625    1*2^-10  Zwischenergebnis: 0,8193359375
0,00048828125   1*2^-11  Zwischenergebnis: 0,81982421875
```

Wir sind inzwischen bei der Zahl 0,11100001111 angelangt und haben das gewünschte Ziel immer noch nicht erreicht. Die Wahrheit ist: Sie können dieses Spielchen bis in alle Ewigkeit weiterspielen. Sie werden die Zahl 0,82 des Dezimalsystems mit einer endlichen Anzahl an Ziffern im Binärsystem niemals darstellen können.

BEGLEITDATEIEN Was hat das für Auswirkungen auf die Programmierung unter Visual Basic? Nun, schauen Sie sich dazu einmal das folgende kleine Beispielprogramm an, das Sie in folgendem Verzeichnis finden:

```
...\VB 2010 Entwicklerbuch\Kapitel 07\Primitives02
```

```
Public Class Primitives

    Public Shared Sub main()

        Dim locDouble1, locDouble2 As Double
        Dim locDec1, locDec2 As Decimal

        locDouble1 = 69.82
        locDouble2 = 69.2
        locDouble2 += 0.62

        Console.WriteLine("Die Aussage locDouble1=locDouble2 ist {0}", locDouble1 = locDouble2)
        Console.WriteLine("aber locDouble1 lautet {0} und locDouble2 lautet {1}", _
                          locDouble1, locDouble2)
```

```
        locDec1 = 69.82D
        locDec2 = 69.2D
        locDec2 += 0.62D
        Console.WriteLine("Die Aussage locDec1=locDec2 ist {0}", locDec1 = locDec2)

        Console.WriteLine()
        Console.WriteLine("Taste drücken zum Beenden!")
        Console.ReadKey()

        …

    End Sub

End Class
```

Auf den ersten Blick sollte man meinen, dass beide `WriteLine`-Methoden den gleichen Text ausgeben. Sie brauchen keinen Taschenrechner zu bemühen, um zu sehen, dass der erste Wert und damit die erste Variable innerhalb des Programms die Addition des zweiten und dritten Werts darstellt und die beiden Variablenwerte aus diesem Grund gleich sein sollten. Leider ist dem nicht so. Während Sie mit dem `Decimal`-Datentyp im zweiten Teil des Programms den richtigen Wert herausbekommen, versagt der `Double`-Typ im ersten Teil des Programms.

Noch verwirrender ist die zweite `WriteLine`-Methode. Beide Variablen enthalten nämlich augenscheinlich den gleichen Wert, bis auf die letzte Nachkommastelle genau:

```
Die Aussage locDouble1=locDouble2 ist False
aber locDouble1 lautet 69,82 und locDouble2 lautet 69,82
Die Aussage locDec1=locDec2 ist True

Taste drücken zum Beenden!
```

Das Geheimnis darum ist aber schnell gelüftet: Bei der Umwandlung in eine Zeichenkette findet eine Rundung statt, die über das wahre Ergebnis hinwegtäuscht. Aus dieser Tatsache leiten sich folgende Grundsätze ab:

- Vermeiden Sie es nach Möglichkeit, innerhalb von Schleifen gebrochene `Double`- oder `Single`-Werte zu verwenden. Sie laufen sonst Gefahr, dass sich Ihr Programm aufgrund der beschriebenen Ungenauigkeiten in Endlosschleifen verrennt.
- Verwenden Sie `Single`- und `Double`-Datentypen nur dort, wo es nicht auf die x-te Stelle hinter dem Komma ankommt. Bei der Berechnung von Grafiken beispielsweise, wo Rundungsfehler durch eine geringere Bildschirmauflösung als die Rechenpräzision ohnehin keine Rolle spielen, sollten Sie immer die schnelleren, prozessorberechneten Datentypen `Single` und `Double` dem manuell berechneten `Decimal`-Datentyp vorziehen.
- Bei finanztechnischen Anwendungen sollten Sie *in jedem Fall* den `Decimal`-Datentyp einsetzen. Nur mit ihm ist gewährleistet, dass Additionen und andere Berechnungen von nicht exakt darstellbaren Zahlen nicht in größere Fehler münden.

- Verwenden Sie andererseits den `Decimal`-Datentyp, wenn es eben geht, nie in Schleifen und setzen Sie ihn schon gar nicht als Zählvariable ein. Er erfährt nämlich keine Unterstützung durch den Prozessor und bremst Ihr Programm extrem aus. Verwenden Sie dort nach Möglichkeit nur einen der zahlreichen Integervariablentypen.
- Wenn Sie – aus Geschwindigkeitsgründen – dennoch `Double`- oder `Single`-Typen auf Gleichheit testen müssen, arbeiten Sie besser mit einer Abfrage des Deltas, etwa:

```
If Math.Abs(locDouble1 – locDouble2) < 0.0001 then
   'Werte sind annähernd dieselben, also quasi gleich.
End If
```

Besondere Funktionen, die für alle numerischen Datentypen gelten

Alle numerischen Datentypen verfügen über Methoden, die bei allen Typen nach der gleichen Vorgehensweise verwendet werden. Sie dienen zur Umwandlung einer Zeichenkette (einer Ziffernfolge) in den entsprechenden Wert sowie zur Umwandlung des Werts in eine Zeichenkette. Mit anderen Funktionen können Sie den größten oder kleinsten Wert ermitteln, den ein Datentyp darstellen kann.

Zeichenketten in Werte wandeln und Vermeiden von kulturabhängigen Fehlern

Zum Umwandeln einer Zeichenkette in einen Wert dienen die statischen Funktionen `Parse` oder `TryParse`, die jedem numerischen Datentyp zur Verfügung stehen. Um beispielsweise die Ziffernfolge »123« in den Integerwert 123 umzuwandeln, genügen die folgenden Anweisungen:

```
Dim locInteger As Integer
locInteger = Integer.Parse("123")
```

WICHTIG Allerdings können Sie seit Visual Basic 2005 nicht mehr

```
locInteger = locInteger.Parse("123")   ' Sollte so nicht mehr gemacht werden!
```

schreiben, denn `Parse` ist eine statische Funktion und sollte nicht mehr über eine Objektvariable, sondern nur über den entsprechenden Klassennamen angesprochen werden. Zwar ließe sich das Programm beim Ansprechen der statischen Funktion über eine Objektvariable noch kompilieren, bereits im Code-Editor würden Sie aber eine Warnung in der Fehlerliste sehen.

Es gibt auch die Möglichkeit des Umwandlungsversuchs einer Zeichenkette in einen numerischen Wert, wie es das folgende Beispiel zeigt:

```
Dim locInteger As Integer
If Integer.TryParse("123", locInteger) Then
    'Umwandlung war erfolgreich
Else
    'Umwandlung war nicht erfolgreich
End If
```

Bei erfolgreicher Umwandlung steht die umgewandelte Zahl anschließend in der von `TryParse` zurückgegebenen Variablen – im Beispiel also in `locInteger`. Auch das .NET Framework-Äquivalent von `Integer` ermöglicht die Umwandlung durch

```
locInteger = System.Int32.Parse("123") ' Und auch das ginge.
```

Und um die Liste komplett zu machen, geht es natürlich auch über die `Convert`-Klasse im .NET Framework-Stil mit

```
locInteger = Convert.ToInt32("123")     ' Und das ginge.
```

und in alter Visual Basic-Manier täte es die folgende Anweisung ebenfalls:

```
locInteger = CInt("123")                ' Letzte Möglichkeit, mehr fallen mir nicht ein.
```

Aber aufgepasst: Wenn Sie das folgende Programm auf einem deutschen System starten, passiert möglicherweise nicht das, was Sie erwarten:

```
Dim locString As String = "123.23"
Dim locdouble As Double = Double.Parse(locString)
Console.WriteLine(locdouble.ToString)
```

Sie rechnen vielleicht damit, dass die Ziffernfolge korrekt in den Wert 123,23 umgewandelt wird. Anstelle dessen gibt das Programm

```
12323
```

aus – definitiv nicht das Ergebnis, das Sie erwartet haben. Starten Sie das Programm auf einem englischsprachigen System, ist das Ergebnis korrekt und wie erwartet:[7]

```
123.23
```

Na ja, vielleicht nicht ganz. Wir Deutschen haben uns angewöhnt, die Nachkommastellen von den Vorkommastellen mit einem Komma zu trennen (vermutlich heißen sie auch deshalb Nach*komma*stellen). Englischsprachige Länder machen das allerdings mit einem Punkt, und die Ausgabe, die Sie über diesem Absatz sehen, ist eine korrekte englische Formatierung. Welche Auswirkungen hat dieses Verhalten auf Ihre Programme? Nun, zunächst einmal sollten Sie es unbedingt vermeiden, im Programmcode selbst numerische Konstanten als `String` zu speichern, wenn Sie sie später in einen numerischen Typ wandeln wollen (wie im letzten Beispiel gezeigt). Wenn Sie numerische Datentypen innerhalb Ihres Programms definieren, dann machen Sie es bitte grundsätzlich nur im Code direkt, nicht mit Zeichenketten (in Anführungszeichen) und deren Umwandlungsfunktionen. Sie haben sicherlich schon festgestellt, dass im Code abgelegte Ziffernfolgen (ohne Anführungszeichen) zur Zuweisung eines Werts grundsätzlich nur im englischen Format abgelegt werden.

Solange Sie keine Dateien mit als Text gespeicherten Informationen, aus denen Ihr Programm Werte generieren muss, über Kulturgrenzen hinweg austauschen müssen, haben Sie nichts zu befürchten: Läuft Ihre Anwendung auf einem englischsprachigen System, werden Zahlen mit Punkt als Trennzeichen in die Datei geschrieben, hier im deutschsprachigen Raum eben als Komma. Da die Kultureinstellungen beim Einlesen äquivalent berücksichtigt werden, kann Ihre Anwendung auch die richtigen Werte aus der Textdatei zurückgenerieren.

---

[7] Wobei dieses Verhalten streng genommen nicht dadurch bedingt wird, dass es sich um ein englischsprachiges System handelt, sondern dass ein englischsprachiges Betriebssystem voreingestellt andere Ländereinstellungen aufweist als ein deutschsprachiges. Natürlich könnten Sie auch ein deutschsprachiges Betriebssystem so konfigurieren, dass es das gleiche Ergebnis liefert.

Problematisch wird es dann, wenn auch die Dateien mit den Texten über Kulturgrenzen hinweg ausgetauscht werden sollen. Dies geschieht zum Beispiel schnell, wenn Sie mit einer .NET Windows Forms-Anwendung von Ihrem deutschen Windows XP auf einen Datenbankserver im Unternehmen zugreifen. Denn viele IT-Abteilungen betreiben ihre Server aus den verschiedensten Gründen ausschließlich in den englischsprachigen Versionen. Dann würde eine USA-Plattform die Datei mit Punkt als Trennzeichen exportieren und hier in Deutschland würde die `Parse`-Funktion den Punkt als Tausenderpunkt ansehen, damit unter den Tisch fallen lassen und so fälschlicherweise den Wert verhundertfachen. In diesem Fall müssen Sie dafür sorgen, dass der Export in eine Textdatei kulturneutral erfolgt, und das können Sie auf folgende Weise erreichen:

Sowohl die `Parse`-Funktion als auch die `ToString`-Funktion aller numerischen Typen kann bei der Umwandlung durch einen so genannten *Formatprovider* (etwa: Formatanbieter) spezifisch gesteuert werden. Es gibt die unterschiedlichsten Formatprovider in .NET für numerische Typen, nämlich solche, mit denen Sie Formate in Abhängigkeit von der Anwendung (finanztechnisch, wissenschaftlich etc.) steuern können oder aber jene, bei denen dies in Abhängigkeit von der Kultur geschieht. Dazu dienen die Klassen `NumberFormatInfo` und `CultureInfo`. Beiden lassen sich, zuvor entsprechend instanziiert und aufbereitet, als Parameter sowohl der `ToString`- als auch der `Parse`-Funktion übergeben.

WICHTIG Diese Vorgehensweise, wie sie im Folgenden zu sehen ist, sollten Sie immer bei Anwendungen anwenden, die mit internationalem Anspruch entwickelt werden, um Fehler bei Typkonvertierungen von vornherein auszuschließen.

```
Dim locString As String = "123.23"
Dim locdouble As Double

locdouble = Double.Parse(locString, CultureInfo.InvariantCulture)
Console.WriteLine(locdouble.ToString(CultureInfo.InvariantCulture))
Console.ReadLine()
```

HINWEIS Damit Sie auf kulturbezogene Klassen und Funktionen zugreifen können, müssen Sie am Anfang des Programms den entsprechenden Namespace `System.Globalization` mit `Imports` eingebunden haben, etwa so:

```
Imports System.Globalization
```

Abbildung 7.1 Falls die Zeichenkette zur Umwandlung aufgrund ihres Formats nicht umgewandelt werden kann, generiert .NET Framework eine Ausnahme

Die statische Eigenschaft `InvariantCulture` liefert direkt eine Instanz einer `CultureInfo`-Klasse zurück, die mit den entsprechenden Eigenschaften bestückt wurde.

HINWEIS Wenn der Umwandlungsversuch fehlschlägt, da die Zeichenkette schlicht und ergreifend kein konvertierbares Format enthält und sich deswegen nicht in einen Wert umwandeln lässt, generiert .NET Framework eine Ausnahme (Abbildung 7.1). Sie können die Ausnahme entweder mit `Try … Catch` abfangen oder alternativ die statische Funktion `TryParse` (siehe Seite 277) verwenden, die grundsätzlich keine Ausnahme beim Konvertierungsversuch erzeugt.

Performance- und Rundungs-Problematik beim Konvertieren von Fließkomma- zu Integerwerten

Wenn Sie in Visual Basic .NET typsicher programmieren – was Sie in jedem Fall machen sollten übrigens, indem Sie `Option Strict On` in den Projekteigenschaften einschalten – dann wissen Sie sicherlich, dass es üblich ist, beispielsweise mit dem Konvertierungsoperator `CInt` eine Fließkommazahl in einen Wert vom Typ `Integer` umzuwandeln.

Was viele nicht wissen: Der Visual Basic-Compiler verhält sich hier völlig anders, als beispielsweise der äquivalente Casting-Operator in C# (der nämlich gar nicht so äquivalent ist). `CInt` in Visual Basic rundet nämlich kaufmännisch, und tatsächlich macht der Compiler aus

```
Dim einInt = CInt(123.54R)
```

die folgende Version:

```
Dim einInt = CInt(Math.Round(123.54R))
```

Ein einfaches `CInt`, so wie es der VB-Compiler schließlich selbst benutzt, und wie es in C# Standard ist, *können* Sie in Visual Basic selbst überhaupt nicht implementieren. In C# wird beim Konvertieren von Fließkomma nach Integer einfach die Nachkommastelle abgeschnitten, also nicht gerundet. Um das zu simulieren, müssten Sie sich des folgenden Konstrukts bedienen:

```
Dim einInt = CInt(Math.Truncate(123.54R))
```

Das Problem ist, dass der Compiler daraus folgenden, völlig redundanten Code erzeugt:

```
Dim einInt = CInt(Math.Round(Math.Truncate(123.54R)))
```

Im Bereich der Grafikverarbeitung beispielsweise ist das natürlich ein riesiger Leistungseinbruch, da zwei Funktionen aus der `Math`-Library aufgerufen werden; C# ist hier deutlich schneller, da es ein `CInt` direkt zur Verfügung stellt.

TIPP Für den Fall, dass Sie wirklich auf Leistungsprobleme durch dieses Verhalten stoßen, erstellen Sie eine C#-Assembly, die Ihnen eine Funktion zur direkten Konvertierung von `Double`, `Single` oder `Decimal` nach `Integer` zur Verfügung stellt. Die Wahrscheinlichkeit, dass diese Funktion ge-inline-t wird (also dass der JITter den wenigen Code der C#-Funktion einfach in Ihre Assembly überträgt, und gar keinen Funktionsaufruf-Sprung generiert) ist vergleichsweise groß, und so bekommen Sie nahezu die gleiche Performance wie unter C#.

Ermitteln von minimal und maximal darstellbarem Wert eines numerischen Typs

Die numerischen Datentypen kennen zwei spezielle statische Eigenschaften, mit denen Sie jeweils den größten und kleinsten darstellbaren Wert ermitteln lassen können. Die Eigenschaften lauten `MinValue` und `MaxValue`, und Sie können sie wie jede statische Funktion entweder durch den Typennamen selbst oder durch eine Objektvariable des Typs aufrufen (letzteres führt allerdings seit Visual Basic 2005 zu einer Warnung; und auch wenn es daher an sich funktioniert, sollten Sie der besseren Lesbarkeit Ihres Code wegen darauf verzichten). Beispiel:

```
Dim locInteger As Integer
Dim locdouble As Double
Dim locDecimal As System.Decimal

Console.WriteLine(locInteger.MaxValue)
Console.WriteLine(Double.MinValue)
Console.WriteLine(locDecimal.MaxValue)
```

Spezielle Funktionen der Fließkommatypen

Die Fließkommatypen verfügen über einige besondere Eigenschaften zur Darstellung von speziellen Werten, auf deren Zustand Sie eine Objektvariable mit entsprechenden Funktionen überprüfen können.

Unendlich (Infinity)

Wenn Sie einen Fließkommawertetyp durch 0 teilen, dann erzeugen Sie damit keine Ausnahme (keinen Fehler). Vielmehr ist das Ergebnis *unendlich* (*infinity*), und sowohl `Single` als auch `Double` können dieses Ergebnis darstellen, wie das folgende Beispiel demonstriert:

```
Dim locdouble As Double
locdouble = 20
locdouble /= 0
Console.WriteLine(locdouble)
Console.WriteLine("Die Aussage locDouble ist +unendlich ist {0}.", locdouble = Double.PositiveInfinity)
```

Wenn Sie dieses Beispiel ausführen, erzeugt es keine Fehlermeldung in Form einer Ausnahme, sondern das Programm schreibt vielmehr ein Ergebnis auf den Bildschirm:

```
+unendlich
Die Aussage locDouble ist +unendlich ist True.
```

Anstatt den Vergleich auf Unendlichkeit durch den Vergleichsoperator durchzuführen, können Sie auch die statische Funktion `IsInfinity` verwenden:

```
Console.WriteLine("Die Aussage locDouble ist +unendlich ist {0}.", locdouble.IsInfinity(locdouble))
```

Das hat den Vorteil, dass Sie im Vorfeld nicht wissen müssen, ob ein Ergebnis positiv oder negativ unendlich ist. Mit den Funktionen `IsPositiveInfinity` und `IsNegativeInfinity` können Sie dennoch die Differenzierung innerhalb einer Abfrage vornehmen.

Um einer Variable gezielt den Wert *unendlich* zuzuweisen, verwenden Sie die statischen Funktionen `PositiveInfinity` und `NegativeInfinity`, die entsprechende Konstanten zurückliefern.

Keine Zahl (NaN, Not a Number)

Und noch einen Sonderfall decken die primitiven Fließkommatypen ab: Die Division von 0 durch 0, die mathematisch nicht definiert ist und *keine gültige Zahl* ergibt:

```
'Sonderfall: 0/0 ist mathematisch nicht definiert und ergibt "Not a Number"
einDouble = 0
einDouble = einDouble / 0
If Double.IsNaN(einDouble) Then
    Debug.Print("einDouble ist keine Zahl!")
End If
```

Lassen Sie diesen Code laufen, wird der Text in der `If`-Abfrage ausgegeben.

WICHTIG Überprüfungen auf diese Sonderwerte lassen sich nur durch die Eigenschaften testen, die statische Funktionen direkt an den Typen »hängen«. Zwar können Sie beispielsweise mit der Fließkommatypenkonstante `NaN` den Wert »keine gültige Zahl« einer Variablen zuweisen; diese Konstante eignet sich allerdings nicht, auf diesen Zustand (»Wert«) zu testen, wie das folgende Beispiel zeigt:

```
Dim einDouble As Double

'Sonderfall: 0/0 ist mathematisch nicht definiert und ergibt "Not a Number"
einDouble = 0
einDouble = einDouble / 0

'Der Text sollte erwartungsgemäß ausgegeben werden,
'wird er aber nicht!
If einDouble = Double.NaN Then
    Debug.Print("Test 1:einDouble ist keine Zahl!")
End If

'Nur so kann der Test erfolgen!
If Double.IsNaN(einDouble) Then
    Debug.Print("Test 2:einDouble ist keine Zahl!")
End If
```

In diesem Beispiel wird nur der zweite Text ausgegeben.

Versuchte Umwandlungen mit TryParse

Alle numerischen Datentypen kennen die statische Funktion `TryParse`, die versucht, eine Zeichenkette in einen Wert umzuwandeln. Im Gegensatz zu `Parse` erzeugt sie allerdings keine Ausnahme, wenn die Umwandlung nicht gelingt. Vielmehr übergeben Sie ihr eine Variable als Referenz zur Speicherung des Rückgabewerts, und das Funktionsergebnis informiert Sie darüber, ob die Konvertierung gelang (`True`) oder nicht (`False`):

```
Dim locdouble As Double
Dim locString As String = "Einhundertdreiundzwanzig"

'locdouble = Double.Parse(locString) ' Ausnahme
'Keine Ausnahme:
Console.WriteLine("Konvertierung erfolgreich? {0}", _
         Double.TryParse(locString, NumberStyles.Any, New CultureInfo("de-DE"), locdouble))
```

Spezielle Funktionen des Wertetyps Decimal

Der Wertetyp `Decimal` verfügt ebenfalls über spezielle Funktionen, von denen viele allerdings in Visual Basic nicht zur Anwendung kommen (nichtsdestotrotz können Sie sie verwenden, aber es ergibt wenig Sinn – tatsächlich wurden sie für andere Sprachen, die eine Operatorenüberladung (noch) nicht kennen, aufgenommen). Nehmen wir als Beispiel die statische `Add`-Funktion, die zwei Zahlen vom Typ `Decimal` addiert. Sie liefert als Ergebnis ein `Decimal` zurück. Stattdessen können Sie aber auch auf den +-Operator von Visual Basic zurückgreifen, der ebenfalls zwei Zahlen vom Typ *Decimal* addieren kann – und das natürlich für die spätere Nacharbeitung in einem viel leichter lesbaren Code. Lediglich die Funktionen der folgenden Tabelle sind sinnvoll im Gebrauch:

Funktionsname	Aufgabe
`Remainder(Dec1, Dec2)`	Ermittelt den Rest der Division der beiden *Decimal*-Werte.
`Round(Dec, Integer)`	Rundet einen *Decimal*-Wert auf die angegebene Anzahl der Nachkommastellen.
`Truncate(Dec)`	Liefert den Vorkommateil des angegebenen *Decimal*-Werts zurück.
`Floor(Dec)`	Rundet den *Decimal*-Wert auf die nächste kleinere ganze Zahl.
`Negate(Decimal)`	Multipliziert den *Decimal*-Wert mit –1.

Tabelle 7.3 Die wichtigsten Funktionen des *Decimal*-Typs

Der Datentyp Char

Der `Char`-Datentyp speichert ein Zeichen im Unicode-Format (mehr zu diesem Thema finden Sie im Abschnitt »Speicherbedarf von Strings« auf Seite 284) und belegt damit 16 Bit bzw. 2 Byte. Anders als `String` ist der `Char`-Datentyp ein Wertetyp. Die folgende Kurzübersicht klärt nähere Details:

.NET-Datentyp: `System.Char`

Stellt dar: ein einzelnes Zeichen

Wertebereich: Die Wertebereich beträgt 0–65.535, damit Unicode-Zeichen dargestellt werden können

Typliteral: c

Speicherbedarf: 2 Byte

Delegation an den Prozessor: ja

CLS-compliant: ja

Anmerkungen: `Char`-Werte werden häufig in Arrays verwendet, da ihre Verarbeitung in vielen Fällen praktischer ist als die Verarbeitung von *Strings*. Sie können `Char`-Arrays wie jeden anderen Datentyp mit Konstanten definieren; ein Beispiel dafür finden Sie unter dem nächsten Punkt. Weitere Beispiele, wie Sie `Char`-Arrays anstelle von *Strings* beispielsweise zum Verarbeiten Buchstabe für Buchstabe verwenden, hält der gleich folgende Abschnitt über Strings bereit.

Auch wenn Char intern als vorzeichenloser 16-Bit-Wert und damit wie ein Short gespeichert wird, lassen sich keine impliziten Konvertierungen in einen numerischen Typ vornehmen. Sie können aber, in Ergänzung zur in der Onlinehilfe beschriebenen Möglichkeit nicht nur die Funktionen AscW und ChrW zur Umwandlung von Char in einen numerischen Datentyp und umgekehrt verwenden, sondern auch die Convert-Klasse. Beispiel:

```
'Normale Deklaration und Definition
Dim locChar As Char
locChar = "1"c
Dim locInteger As Integer = Convert.ToInt32(locChar)
Console.WriteLine("Der Wert von '{0}' lautet {1}", locChar, locInteger)
```

Wenn Sie dieses Beispiel laufen lassen, sehen Sie folgende Ausgabe auf dem Bildschirm:

```
Der Wert von '1' lautet 49
```

Die Funktionen Chr und Asc können Sie ebenfalls verwenden; sie funktionieren, aber nur für Nicht-Unicode-Zeichen (ASCII 0-255). Außerdem haben sie durch etliche interne Bereichsüberprüfungen einen enormen Overhead und sind deswegen lange nicht so schnell wie AscW, ChrW (die am schnellsten sind, weil eine direkte interne Typumwandlung von Char in Integer und umgekehrt stattfindet) oder die Convert-Klasse (die den Vorteil hat, auch von Nicht-Visual Basic-Entwicklern verstanden zu werden).

Deklaration und Beispielzuweisung (auch als Array):

```
'Normale Deklaration und Definition
Dim locChar As Char
locChar = "K"c

'Ein Char-Array mit Konstanten deklarieren und definieren.
Dim locCharArray() As Char = {"A"c, "B"c, "C"c}

'Ein Char-Array in einen String umwandeln.
Dim locStringAusCharArray As String = New String(locCharArray)

'Einen String in ein Char-Array umwandeln.
'Das geht natürlich auch mit einer Stringvariablen.
locCharArray = "Dies ist ein String".ToCharArray
```

Der Datentyp String

Mit Strings speichern Sie Zeichenketten. Anders als bei Visual Basic 6.0 gibt es in den modernen Visual Basic-Versionen für den Umgang mit Strings einen objektorientierten Ansatz, mit dem die Programmierung der Stringverarbeitung viel einfacher wird. Man wird Ihre Programme viel leichter lesen können, wenn Sie dieses OOP-Konzept verwenden.

Im Laufe der vergangenen Kapitel sind Ihnen Strings schon an vielen Stellen begegnet. Dennoch lohnt es sich, einen weiteren Blick hinter die Kulissen zu werfen, und nicht zuletzt durch die Klasse Regex (Abkürzung von »Regular Expressions« – etwa: »reguläre Ausdrücke«) gibt Ihnen .NET Framework eine Unterstützung in Sachen Zeichenketten an die Hand, wie sie besser eigentlich nicht mehr sein kann.

Anders als bei anderen primitiven Typen handelt es sich bei Strings um Referenztypen. Dennoch ist es nicht notwendig, eine neue `String`-Instanz mit dem Schlüsselwort `New` zu definieren.[8] Erreicht wird das durch das Eingreifen des Compilers, der ohnehin anderen Code generieren muss als bei anderen Objekten.

Die folgenden Abschnitte geben Ihnen eine Übersicht über die besonderen Eigenarten der Strings der Base Class Library. Am Ende dieses Abschnitts finden Sie im Referenzstil die Anwendung der wichtigsten `String`-Funktionen beschrieben.

Strings – gestern und heute

Durch die neue Implementierung des Datentyps `String`, der ebenso wie andere Objekte durch Instanziierung seiner Klasse entsteht, gibt es seit Visual Studio 2002 und .NET Framework 1.0 eine völlig neue Herangehensweise bei der Arbeit mit Zeichenketten.

Fast alle Befehle und Funktionen, die es in Visual Basic 6.0 bzw. VBA noch »allein stehend« gab, gibt es auch noch in den .NET Framework-Versionen von Visual Basic. Allerdings sind sie nicht nur überflüssig, da es sich mit den vorhandenen Methoden und Eigenschaften des `String`-Objekts viel eleganter zum Ziel kommen lässt, sondern sie bremsen Programme auch unnötig aus, da sie letzten Endes selbst die `String`-Objektfunktionen aufrufen.

Für (fast) jede der alten String-Funktionen gibt es eine entsprechende Klassenfunktion, die Sie als Ersatz verwenden sollten. Die folgenden Abschnitte demonstrieren den Umgang mit Strings anhand kurzer Beispiele.

Strings deklarieren und definieren

Strings werden, wie alle primitiven Datentypen, ohne das Schlüsselwort `New` deklariert; Zuweisungen können direkt im Programm erfolgen. Eine Zeichenkette kann also beispielsweise mit der Anweisung

```
Dim locString As String
```

deklariert und sofort verwendet werden. Die Instanzbildung des `String`-Objekts geschieht auf IML-Ebene. Strings werden definiert, indem Sie ihnen eine Zeichenkette zuweisen, die Sie in Anführungszeichen setzen, wie im folgenden Beispiel:

```
locString = "Miriam Sonntag"
```

Genau wie andere primitive Datentypen können Deklaration und Zuweisung in einer Anweisung erfolgen. So könnten Sie natürlich die beiden oben stehenden einzelnen Anweisungen durch die folgende ersetzen:

```
Dim locString As String = "Miriam Sonntag"
```

---

[8] Ähnlich wie bei Nullables beim Boxing greift die CLR hier in das Standardverhalten für Referenztypen ein und ändert es – da Strings Referenztypen sind, müssten sie natürlich streng genommen mit `New` instanziiert werden. Der Gleichheitsoperator müsste allerdings ebenfalls lediglich eine Referenz zuweisen – bei Strings wird beim Zuweisen einer Instanz an eine Objektvariable allerdings ein Cloning des Inhalts vorgenommen, ebenfalls abweichend vom Standard. Mehr zum Thema Referenz- und Verweistypen, die diesen Zusammenhang genauer erklären, finden Sie in Kapitel 13.

Prüfen, ob ein String »leer« ist – String.IsNullOrWhiteSpace

Visual Basic-Entwickler sind seit Jahrzehnten – möchte man sagen – ein bestimmtes Muster gewohnt, um zu überprüfen, ob eine Zeichenfolgenvariable entweder nicht definiert (`Nothing`) ist oder auf einen Leerstring zeigt:

```
Dim meineZeichenfolge As String
meineZeichenfolge = WinFormsTextBox.Text

If meineZeichenfolge Is Nothing Or meineZeichenfolge = "" Then
     MessageBox.Show("meineZeichenfolge ist leer.")
End If
```

Der Datentyp `String` verhält sich erst einmal wie ein ganz normaler Referenztyp. Sobald er definiert wurde, ist er `Nothing`, denn sein Inhalt zeigt auf *nichts*. Er hat aber auch noch einen weiteren Zustand, der dem Wert »nichts« entspricht, und das ist, wenn er auf einen Speicherbereich zeigt, der für das Speichern von Zeichen vorgesehen ist, aber nicht genutzt wird. In diesem Fall speichert die `String`-Variable einen so genannten Leerstring. Eine dritte Möglichkeit ist, wenn der String aus Computersicht zwar Daten enthält, die aber keine echten Inhaltsdaten darstellen, da sie nicht druckbar (genauer bei der Bildschirmdarstellung und beim Druck nicht sichtbar) sind. Solche Zeichen nennt man auch Whitespaces. Es handelt sich dabei um, Leerzeichen, Tabulatoren oder sonstige Steuerzeichen.

Alle diese Zustände prüft eine statische Methode des `String`-Typs, und sie vereinfacht das Testen auf solche Zustände:

```
        If String.IsNullOrWhiteSpace(meineZeichenfolge) Then
            MessageBox.Show("meine Zeichenfolge ist leer.")
        End If
```

Die Meldung »meine Zeichenfolge ist leer« erhalten Sie also hier in allen Fällen, wenn die Zeichenfolge `meineZeichenfolge` faktisch leer ist, also einen Leerstring aufweist, `Nothing` ist oder ausschließlich ein oder mehrere Whitespaces enthält (Leerzeichen, Tabulator, anderes Steuerzeichen).

Die `IsNullOrWhiteSpace`-Methode steht ab .NET Framework 4.0 zur Verfügung. Falls Sie aus Gründen der Abwärtskompatibilität gegen eine frühere .NET Framework-Version entwickeln müssen oder Whitespaces als gültige Eingabe zulassen wollen, verwenden Sie stattdessen besser die `IsNullOrEmpty`-Methode, die bei `Nothing und` Leerstrings `True` ergibt:

```
        If String.IsNullOrEmpty(meineZeichenfolge) Then
            MessageBox.Show("meineZeichenfolge ist leer.")
        End If
```

Automatische Zeichenwiederholungen durch den String-Konstruktor (String$-Funktionalität aus VBA)

Normalerweise dient ein Konstruktor einer Klasse zum Erstellen einer Instanz und einer Struktur, um bestimmte Parameter mit Werten vorzubelegen – der OOP-Teil dieses Buchs verrät Ihnen mehr zu diesem Thema.

Und obwohl Sie Strings wie primitive Datentypen ohne Konstruktor erstellen, haben Sie dennoch die Möglichkeit, einen Konstruktor zu verwenden. Jedoch setzen Sie den Konstruktor nicht ausschließlich zur Neuinstanziierung eines leeren `String`-Objekts ein (der parameterlose Konstruktor ist dafür auch gar nicht erlaubt), sondern emulieren (unter anderem) eigentlich die alte `String$`-Funktion aus Visual Basic 6.0.

Mit ihrer Hilfe war es möglich, eine Zeichenfolge programmgesteuert zu wiederholen und in einem String zu speichern. Übrigens: Während viele der alten Visual Basic 6.0-Befehle auch noch in den .NET Framework Versionen 1.0, 1.1 und 2.0 vorhanden sind, gibt es die `String`-Funktion selbst – wohl wegen des gleich lautenden Typbezeichners – in den .NET-Versionen von Visual Basic nicht mehr. Um sich des `String`-Konstruktors als `String$`-Funktionsersatz zu bedienen, verfahren Sie folgendermaßen:

```
Dim locString As String = New String("A"c, 40)
```

Sie sehen am Typliteral »c«, dass Sie im Konstruktor einen Wert vom Typ `Char` übergeben müssen. Damit beschränkt sich die Wiederholungsfunktion dummerweise auf ein Zeichen, was bei `String$` nicht der Fall war. Eine eigene `Repeat`-Funktion zu implementieren, die diese Aufgabe löst, ist aber kein wirkliches Problem:

```
Public Function Repeat(ByVal s As String, ByVal repeatitions As Integer) As String

    Dim locString As String

    For count As Integer = 1 To repeatitions
        locString &= s
    Next

    Return locString
End Function
```

HINWEIS Dieses Konstrukt dient in erster Linie als Beispiel, und Sie sollten Zeichenketten auf diese Weise nur »zusammenbauen«, wenn es sich um wenige Zeichen handelt. Für größere Mengen verwenden Sie aus Performancegründen besser die `StringBuilder`-Klasse, die Sie im Abschnitt »StringBuilder vs. String – wenn es auf Geschwindigkeit ankommt« ab Seite 293 beschrieben finden. Warum das so ist, klärt der Abschnitt »Strings sind unveränderlich« ab Seite 284.

Neben der Fähigkeit des Konstruktors, Strings aus einem sich wiederholenden Zeichen zu generieren, können Sie ihn ebenfalls dazu verwenden, einen String aus einem `Char`-Array oder einem Teil eines `Char`-Arrays zu erstellen, wie das folgende Beispiel zeigt:

```
Dim locCharArray() As Char = {"K"c, "."c, " "c, "L"c, "ö"c, "f"c, "f"c, "e"c, "l"c, "m"c, "a"c, "n"c,
"n"c}
Dim locString As String = New String(locCharArray)
Console.WriteLine(locString)
locString = New String(locCharArray, 3, 6)
Console.WriteLine(locString)
```

Wenn Sie dieses Programm laufen lassen, erscheint im Konsolenfenster folgende Ausgabe:

```
K. Löffelmann
Löffel
```

Einem String Zeichenketten mit Sonderzeichen zuweisen

Wenn Sie Anführungszeichen im String selbst verwenden wollen, dann bedienen Sie sich doppelter Anführungszeichen. Um die Zeichenkette

```
Miriam sagte, "es ist erst 13.00 Uhr, ich schlafe noch ein wenig!".
```

im Programm zu definieren, würde die Zuweisungsanweisung folgendermaßen lauten:

```
locString = "Miriam sagte, ""es ist erst 13.00 Uhr, ich schlafe noch ein wenig!""."
```

Möchten Sie andere Sonderzeichen in Strings einbauen, bedienen Sie sich im Basic-Sprachschatz vorhandener Konstanten. Um beispielsweise einen Absatz in einen String einzubauen, müssen Sie die ASCII-Codes für *Linefeed* (Zeilenvorschub) und *Carriage Return* (Wagenrücklauf) in den String einfügen. Sie erreichen das durch die Verwendung der Konstanten, wie im folgenden Beispiel zu sehen:

```
locAndererString = "Miriam sagte ""ich schlafe noch ein wenig!""" + vbCr + vbLf + _
                    "Sie schlief direkt wieder ein."
```

Weniger Schreibarbeit haben Sie mit der folgenden Version, die exakt das gleiche Resultat liefert:

```
locAndererString = "Miriam sagte ""ich schlafe noch ein wenig!""" + vbNewLine + _
                   "Sie schlief direkt wieder ein."
```

Ihnen stehen zum Einfügen von Sonderzeichen die folgenden Konstanten in Visual Basic zur Verfügung:

Konstante	ASCII	Beschreibung
`vbCrLf` oder `vbNewLine`	13; 10	Wagenrücklaufzeichen/Zeilenvorschubzeichen
`vbCr`	13	Wagenrücklaufzeichen
`vbLf`	10	Zeilenvorschubzeichen
`vbNullChar`	0	Zeichen mit dem Wert 0
`vbNullString`		Zeichenfolge mit dem Wert 0. Entspricht nicht einer Zeichenfolge mit 0-Länge (""); diese Konstante ist für den Aufruf externer Prozeduren gedacht (COM-Interop).
`vbTab`	9	Tabulatorzeichen
`vbBack`	8	Rückschrittzeichen
`vbFormFeed`	12	Wird in Microsoft Windows nicht verwendet
`vbVerticalTab`	11	Steuerzeichen für den vertikalen Tabulator, der in Microsoft Windows aber nicht verwendet wird

Tabelle 7.4 Die einfachsten Möglichkeiten, Sonderzeichen in Strings einzubauen

Speicherbedarf von Strings

Jedes Zeichen, das in einem String gespeichert wird, belegt zwei Bytes an Arbeitsspeicher. Auch wenn Strings in Buchstabenform ausgegeben werden, so hat im Speicher selbst natürlich jedes Zeichen einen bestimmten Wert. Die Codewerte von Strings entsprechen bis 255 dem ***A**merican **S**tandard **C**ode for **I**nformation **I**nterchange* – kurz ASCII –, wobei nur Werte bis 127 bei jedem verwendeten Ausgabezeichensatz einheitlich sind. Sonderzeichen spezieller Länder sind, abhängig vom verwendeten Zeichensatz, in den Bereichen 128–255 definiert, wobei auch hier in der Regel die Codes für die in europäischen Ländern verwendeten Sonderzeichen wie »öäüÖÄÜâéè« in jedem Font dieselben Codes haben (Ausnahmen bestätigen wie immer die Regel). Werte über 255 stellen Sonderzeichen dar, die beispielsweise für kyrillische, arabische oder asiatische Zeichen verwendet werden. Die Kodierungskonvention, nach der ein Zeichen Werte über 255 annehmen darf und welche die Kodierung einer größeren Anzahl von Zeichen erlaubt, nennt man übrigens *Unicode*. Strings in .NET Framework speichern Zeichenketten generell im *Unicode*-Format.

Strings sind unveränderlich

Strings sind generell Referenztypen[9], aber dafür grundsätzlich statisch im Speicher, also unveränderlich. Das bedeutet für Sie in der Praxis keine Einschränkung mit dem gewohnten Umgang von Zeichenketten, denn: Wenn es eigentlich so aussieht, als hätten Sie einen String verändert, dann haben Sie in Wahrheit einen neuen erzeugt, der die Veränderungen widerspiegelt. Wissen müssen Sie das nur bei Anwendungen, die sehr viele Stringoperationen beanspruchen. In diesem Fall sollten Sie die so genannte `StringBuilder`-Klasse verwenden, da diese zwar nicht die Flexibilität von Strings und auch nicht den Vorteil von primitiven Datentypen mit sich bringt, aber für diese Fälle deutlich leistungsfähiger ist (der Abschnitt »StringBuilder vs. String – wenn es auf Geschwindigkeit ankommt« ab Seite 293 verrät mehr darüber).

Viel wichtiger ist die Auswirkung dieser Unveränderlichkeit von Strings beim Einsatz in Ihren Programmen: Obwohl Strings als Referenztypen gelten, haben sie letzten Endes das Verhalten von Wertetypen, eben dadurch, dass sie unveränderlich sind. Wenn zwei Stringvariablen auf denselben Speicherbereich verweisen und Sie den Inhalt eines Strings verändern, dann sieht es nämlich nur so aus, als würden Sie ihn verändern. In Wirklichkeit legen Sie ja ein komplett neues `String`-Objekt im Speicher ab und lassen die vorhandene Variable darauf verweisen. Damit kommen Sie nie in die Situation, die Sie von Referenztypen kennen: Das Verändern eines Objektinhalts durch die eine Objektvariable führt bei Strings nie dazu, dass eine andere Objektvariable, die auf den gleichen Speicherbereich zeigt, den Stringinhalt verändert wiedergibt, denn Strings werden ja nicht verändert. Diese Tatsache erklärt, dass Strings zwar Referenztypen sind, sich aber wie Wertetypen »anfühlen«.

Mehr Informationen dazu erhalten Sie auch im nächsten Abschnitt und den programmtechnischen Beweis dafür im Abschnitt »Trimmen von Strings« auf Seite 290.

[9] Mehr zu Referenz- und zu Wertetypen erfahren Sie im Kapitel 9.

Speicheroptimierung von Strings durch .NET Framework

Für die Speicherung von Strings gibt es einen so genannten *internen Pool*, der dazu verwendet wird, Redundanzen bei der Zeichenkettenspeicherung zu vermeiden. Wenn Sie innerhalb Ihres Programms zwei Strings mit der gleichen Konstante definieren, erkennt der Visual Basic-Compiler das und legt den String im Speicher nur ein einziges Mal ab, lässt beide Objektvariablen aber auf denselben Speicherbereich zeigen, wie das folgende Beispiel beweist:

```
Dim locString As String
Dim locAndererString As String

locString = "Miriam" & " Sonntag"
locAndererString = "Miriam Sonntag"
Console.WriteLine(locString Is locAndererString)
```

Wenn Sie dieses Programm starten, gibt es `True` aus – beide Strings verweisen also auf den gleichen Speicherbereich. Diese Tatsache gilt aber nur so lange, wie der Compiler die Gleichheit der Strings erkennen kann, und dafür müssen die Konstanten in der gleichen Zeile zusammengesetzt werden. Schon bei der folgenden Veränderung kann der Compiler die Gleichheit der Strings nicht mehr feststellen, und das Ergebnis wird `False`:

```
Dim locString As String
Dim locAndererString As String

locString = "Miriam"
locString &= " Sonntag"
locAndererString = "Miriam Sonntag"
Console.WriteLine(locString Is locAndererString)
```

Es liegt auf der Hand, dass ein Redundanzvermeidungsverhalten zur Laufzeit zu viel Zeit kosten würde, um es sinnvoll anzuwenden. Bei sehr hohem Stringaufkommen würde die BCL zu viel Zeit nach der Suche bereits vorhandener Strings verschwenden. Allerdings haben Sie die Möglichkeit, einen String, den Sie zur Laufzeit erstellen, gezielt dem Pool hinzuzufügen. Gleichen sich mehrere Strings, die Sie in den internen Pool aufnehmen, werden die Speicherbereiche wieder nicht redundant zugewiesen – mehrere, gleich lautende Strings teilen sich dann den Speicher. Sinnvoll ist das natürlich nur dann, wenn vorauszusehen ist, dass es viele gleich lautende Strings innerhalb eines Programms geben wird. Ein Beispiel zeigt, wie das Hinzufügen eines Strings mit der statischen Funktion `Intern` zum internen Pool explizit vorgenommen wird:

```
Dim locString As String
Dim locAndererString As String

locString = "Miriam"
locString &= " Sonntag"
locString = String.Intern(locString)
locAndererString = String.Intern("Miriam Sonntag")
Console.WriteLine(locString Is locAndererString)
```

Wenn Sie dieses Programm starten, lautet die Ausgabe wieder `True`.

Ermitteln der String-Länge

Visual Basic 6.0-kompatibler Befehl: `Len`

.NET-Versionen von Visual Basic: `strVar.Length`

Anmerkung: Mit diesem Befehl ermitteln Sie die Länge eines Strings in Zeichen (nicht in Byte!).

Beispiel: Das folgende Beispiel liest eine Zeichenkette von der Tastatur ein und gibt die Zeichen in umgekehrter Reihenfolge aus:

```
Dim locString As String
Console.Write("Geben Sie einen Text ein: ")
locString = Console.ReadLine()
For count As Integer = locString.Length - 1 To 0 Step -1
   Console.Write(locString.Substring(count, 1))
Next
```

Das gleiche Beispiel mit den Visual Basic 6.0-Kompatibilitätsbefehlen finden Sie im nächsten Abschnitt.

Ermitteln von Teilen eines Strings oder eines einzelnen Zeichens

Visual Basic 6.0-kompatible(r) Befehl(e): `Left, Right, Mid`

.NET-Versionen von Visual Basic: `strVar.SubString`

Anmerkung: Mit diesem Befehl können Sie einen bestimmten Teil eines Strings als String zurückgeben lassen.[10]

Beispiel: Das folgende Beispiel liest eine Zeichenkette von der Tastatur ein und gibt die Zeichen in umgekehrter Reihenfolge aus. Das gleiche Beispiel finden Sie im vorherigen Abschnitt mit den Funktionen des `String`-Objekts.

```
Dim locString As String
Console.Write("Geben Sie einen Text ein: ")
locString = Console.ReadLine()
For count As Integer = Len(locString) To 1 Step -1
    Console.Write(Mid(locString, count, 1))
Next
```

HINWEIS Bitte beachten Sie, dass die Visual Basic 6.0-Kompatibilitätsfunktionen die Zeichenzählung eines Strings durch `Left, Right` oder `Mid` bei 1 beginnen lassen, die Funktion `SubString` jedoch die Zeichenzählung bei 0 startet.

[10] Warum das gute alte `Left` bzw. `Right` als jeweilige Methoden der Klasse `String` weggelassen wurden, weiß allein der Programmierer. Vielleicht hat es der Programmierer auch schlicht vergessen oder kennt nur C und kann sich nicht vorstellen, dass die Welt so einfach wie in Basic sein kann.

Angleichen von String-Längen

Visual Basic 6.0-kompatible(r) Befehl(e): `RSet`, `LSet`

.NET-Versionen von Visual Basic: `strVar.PadLeft`; `strVar.PadRight`

Anmerkung: Mit diesen Befehlen können Sie die Länge eines Strings auf eine bestimmte Zeichenanzahl erweitern; der String wird dabei entweder vorne oder am Ende mit Leerzeichen aufgefüllt.

Beispiel: Das folgende Beispiel demonstriert den Umgang mit der `PadLeft`- und der `PadRight`-Methode.

```
Dim locString As String = "Dieser String ist so lang"
Dim locString2 As String = "Dieser nicht"
Dim locString3 As String = "Dieser"
Len(locString)
locString2 = locString2.PadLeft(locString.Length)
locString3 = locString3.PadRight(locString.Length)

Console.WriteLine(locString + ":")
Console.WriteLine(locString2 + ":")
Console.WriteLine(locString3 + ":")
```

Wenn Sie dieses Programm laufen lassen, generiert es die folgende Ausgabe:

```
Dieser String ist so lang:
            Dieser nicht:
Dieser                  :
```

Suchen und Ersetzen

Visual Basic 6.0 kompatible(r) Befehl(e): `InStr`; `InStrRev`; `Replace`

.NET-Versionen von Visual Basic: `strVar.IndexOf`; `strVar.IndexOfAny`; `strVar.Replace`; `strVar.Remove`

Anmerkung: Mit dem Visual Basic 6.0-kompatiblen Befehl `InStr` können Sie nach dem Vorkommen eines Zeichens oder einer Zeichenfolge in einem String suchen. `InStrRev` macht das Gleiche, beginnt die Suche aber von hinten. `Replace` erlaubt Ihnen, eine Zeichenfolge im String durch eine andere zu ersetzen.

Mit der String-Funktion `IndexOf` suchen Sie nach dem Vorkommen eines Zeichens oder einer Zeichenfolge im aktuellen String. `IndexOfAny` erlaubt Ihnen darüber hinaus, das Vorhandensein verschiedener Zeichen, die in einem `Char`-Array übergeben werden, im String zu finden. `Replace` ersetzt einzelne Zeichen oder Zeichenfolgen durch andere im aktuellen String, und mit `Remove` haben Sie die Möglichkeit, eine bestimmte Zeichenfolge ganz aus dem String zu entfernen.

Beispiel: Das folgende Beispiel demonstriert den Umgang mit den Suchen-und-Ersetzen-Methoden des `String`-Objekts:

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 07\Strings - Suchen und Ersetzen
```

Öffnen Sie dort die Projektmappe (.SLN-Datei).

```
Imports System.Globalization

Module Strings
    Sub Main()
        Dim locString As String =
            "Weisheiten:" + vbNewLine +
            "* Wenn man 8 Jahre, 7 Monate und 6 Tage schreien würde," + vbNewLine +
            "  hätte man genug Energie produziert, um eine Tasse Kaffee heiß zu machen." + vbNewLine +
            "* Wenn man seinen Kopf gegen die Wand schlägt, verbraucht man 150 Kalorien." + vbNewLine +
            "* Elefanten sind die einzigen Tiere, die nicht springen können." + vbNewLine +
            "* Eine Kakerlake kann 9 Tage ohne Kopf überleben, bevor sie verhungert." + vbNewLine +
            "* Gold und andere Metalle entstehen ausschließlich in" + vbNewLine +
            "  Supernovae (Sternenexplosionen)." + vbNewLine +
            "* Der Mond besteht aus den Trümmern der Kollision eines Mars großen" + vbNewLine +
            "  Planeten mit der Erde." + vbNewLine +
            "* New York wird ""Big Apple"" genannt, weil ""Big Apple"" in der Sprache" + vbNewLine +
            "  der Jazz-Musiker ""das große Los ziehen"" bedeutete. In New York Karriere" + vbNewLine +
            "  zu machen war ihr großes Los." + vbNewLine +
            "* Der Ausdruck ""08/15"" für etwas Unoriginelles war ursprünglich " + vbNewLine +
            "  die Typenbezeichnung für das Maschinengewehr LMG 08/15;" + vbNewLine +
            "  er wurde Metapher für geistlosen, militärischen Drill." + vbNewLine +
            "* ""Durch die Lappen gehen"" ist ein Begriff aus der Jagd:" + vbNewLine +
            "  Hirsche liefen nicht durch eine aus Lappen bestehende," + vbNewLine +
            "  flatternde Umzäunung - aus Angst. Außer manchmal."

        'Zahlenkombi durch Buchstaben ersetzen
        locString = locString.Replace("08/15", "Null-Acht-Fünfzehn")

        'Satzzeichen zählen
        Dim locPosition, locCount As Integer

        Do
            locPosition = locString.IndexOfAny(New Char() {"."c, ","c, ":"c, "?"c}, locPosition)
            If locPosition = -1 Then
                Exit Do
            Else
                locCount += 1
            End If
            locPosition += 1
        Loop

        Console.WriteLine("Der folgende Text...")
        Console.WriteLine(New String("="c, 79))
        Console.WriteLine(locString)
        Console.WriteLine(New String("="c, 79))
        Console.WriteLine("...verfügt über {0} Satzzeichen.", locCount)
        Console.WriteLine()
        Console.WriteLine("Und sieht nach dem Ersetzen von 'Big Apple' durch 'Großer Apfel' so aus:")
        Console.WriteLine(New String("="c, 79))

        'Noch eine Ersetzung
        locString = locString.Replace("Big Apple", "Großer Apfel")
        Console.WriteLine(locString)
        Console.ReadLine()

    End Sub

End Module
```

Dieses Beispiel gibt Folgendes auf dem Bildschirm aus:

```
Der folgende Text...
==============================================================================
Weisheiten:
* Wenn man 8 Jahre, 7 Monate und 6 Tage schreien würde,
  hätte man genug Energie produziert, um eine Tasse Kaffee heiß zu machen.
* Wenn man seinen Kopf gegen die Wand schlägt, verbraucht man 150 Kalorien.
* Elefanten sind die einzigen Tiere, die nicht springen können.
* Eine Kakerlake kann 9 Tage ohne Kopf überleben, bevor sie verhungert.
* Gold und andere Metalle entstehen ausschließlich in
  Supernovae (Sternenexplosionen).
* Der Mond besteht aus den Trümmern der Kollision eines Mars großen
  Planeten mit der Erde.
* New York wird "Big Apple" genannt, weil "Big Apple" in der Sprache
  der Jazz-Musiker "das große Los ziehen" bedeutete. In New York Karriere
  zu machen war ihr großes Los.
* Der Ausdruck "Null-Acht-Fünfzehn" für etwas Unoriginelles war ursprünglich
  die Typenbezeichnung für das Maschinengewehr LMG Null-Acht-Fünfzehn;
  er wurde Metapher für geistlosen, militärischen Drill.
* "Durch die Lappen gehen" ist ein Begriff aus der Jagd:
  Hirsche liefen nicht durch eine aus Lappen bestehende,
  flatternde Umzäunung - aus Angst. Außer manchmal.
==============================================================================
...verfügt über 23 Satzzeichen.

Und sieht nach dem Ersetzen von 'Big Apple' durch 'Großer Apfel' so aus:
==============================================================================
Weisheiten:
* Wenn man 8 Jahre, 7 Monate und 6 Tage schreien würde,
  hätte man genug Energie produziert, um eine Tasse Kaffee heiß zu machen.
* Wenn man seinen Kopf gegen die Wand schlägt, verbraucht man 150 Kalorien.
* Elefanten sind die einzigen Tiere, die nicht springen können.
* Eine Kakerlake kann 9 Tage ohne Kopf überleben, bevor sie verhungert.
* Gold und andere Metalle entstehen ausschließlich in
  Supernovae (Sternenexplosionen).
* Der Mond besteht aus den Trümmern der Kollision eines Mars großen
  Planeten mit der Erde.
* New York wird "Großer Apfel" genannt, weil "Großer Apfel" in der Sprache
  der Jazz-Musiker "das große Los ziehen" bedeutete. In New York Karriere
  zu machen war ihr großes Los.
* Der Ausdruck "Null-Acht-Fünfzehn" für etwas Unorginelles war ursprünglich
  die Typenbezeichnung für das Maschinengewehr LMG Null-Acht-Fünfzehn;
  er wurde Metapher für geistlosen, militärischen Drill.
* "Durch die Lappen gehen" ist ein Begriff aus der Jagd:
  Hirsche liefen nicht durch eine aus Lappen bestehende,
  flatternde Umzäunung - aus Angst. Außer manchmal.
```

TIPP Das Beispiel zu »Algorithmisches Auflösen eines Strings in Teile« auf Seite 291 enthält eine weitere, selbst geschriebene Funktion, die ich `ReplaceEx` genannt habe und mit der Sie nach mehreren Zeichen suchen und, falls eines von ihnen dem gesuchten Zeichen entsprach, es durch ein angebbares Zeichen ersetzen können.

Trimmen von Strings

Visual Basic 6.0 kompatible(r) Befehl(e): `Trim; RTrim; LTrim`

.NET-Versionen von Visual Basic: `strVar.Trim; strVar.TrimEnd; strVarTrimStart`

Anmerkung: Mit diesen Befehlen können Sie überflüssige Zeichen am Anfang, am Ende oder an beiden Seiten eines Strings entfernen. Die Objektmethoden des Strings sind dabei den Kompatibilitätsfunktionen vorzuziehen, da Sie bei Ersteren auch bestimmen können, welche Zeichen getrimmt werden sollen, wie das unten stehende Beispiel zeigt. Die Visual Basic 6.0-Kompatibilitätsfunktionen beschränken ihre Trimmfähigkeit auf Leerzeichen.

Beispiel: Das folgende Beispiel generiert ein `String`-Array, dessen einzelne Elemente am Anfang und am Ende unerwünschte Zeichen (nicht nur Leerzeichen) haben, die durch die `Trim`-Funktion entfernt werden.

HINWEIS Dieses Beispiel zeigt dabei ebenfalls, dass sich Strings, obwohl sie als Referenztypen gelten, durch die Tatsache, dass sie an sich unveränderlich sind, anders verhalten als herkömmliche Objekte. Wenn Sie einem Objekt zwei Objektvariablen zuweisen und den Inhalt eines Objekts über eine Variable verändern, dann spiegelt die zweite Objektvariable ebenfalls den geänderten Inhalt des Objekts wider. Obwohl Strings Referenzvariablen sind, zeigen sie dennoch dieses Verhalten nicht, da Sie den String-Inhalt nicht verändern können. Strings werden immer nur neu erstellt, nie verändert (lesen Sie Weiteres zu diesem Thema im Abschnitt »Strings sind unveränderlich« auf Seite 284).

```
Dim locStringArray() As String = { _
   "  - Hier geht der eigentliche Text los!", _
   "Dieser Text endet mit komischen Zeichen! .-     ", _
   "  - Hier sind beide Seiten problematisch -  "}

For Each locString As String In locStringArray
   locString = locString.Trim(New Char() {" "c, "."c, "-"c})
   Console.WriteLine("Sauber und ordentlich: " + locString)
Next

'Wichtig: String ist zwar ein Referenztyp, am Array hat sich dennoch nichts verändert.
'Das liegt daran, dass Strings nicht direkt verändert, sondern immer neu – dabei verändert – angelegt
werden.
For Each locString As String In locStringArray
   Console.WriteLine("Immer noch unordentlich: " + locString)
Next
```

Wenn Sie dieses Programm laufen lassen, generiert es die folgende Ausgabe:

```
Sauber und ordentlich: Hier geht der eigentliche Text los!
Sauber und ordentlich: Dieser Text endet mit komischen Zeichen!
Sauber und ordentlich: Hier sind beide Seiten problematisch
Immer noch unordentlich:   - Hier geht der eigentliche Text los!
Immer noch unordentlich: Dieser Text endet mit komischen Zeichen! .-
Immer noch unordentlich:   - Hier sind beide Seiten problematisch –
```

Algorithmisches Auflösen eines Strings in Teile

Visual Basic 6.0-kompatible(r) Befehl(e): `Split`

.NET-Versionen von Visual Basic: `strVar.Split`

Anmerkung: Die .NET-`Split`-Methode des `String`-Objekts ist der Kompatibilitätsfunktion insofern überlegen, als sie erlaubt, mehrere Separatorzeichen in einem `Char`-Array anzugeben. Damit werden Ihre Programme ungleich flexibler bei der Analyse und dem Neuaufbau von Texten.

Beispiel: Das folgende Beispiel zerlegt die einzelnen, durch verschiedene Separatorzeichen getrennten Begriffe oder Abschnitte eines Strings in Teilstrings, die anschließend als Elemente eines `String`-Arrays vorliegen und durch weitere Funktionen noch besser aufbereitet werden.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 07\String - Split
```

Öffnen Sie dort die Projektmappe (.SLN-Datei).

```
Module Strings
    Sub Main()
        Dim locString As String = _
           "Einzelne, Elemente; durch, die , verschiedensten - Zeichen , getrennt."
        Console.WriteLine("Aus der Zeile:")
        Console.WriteLine(locString)
        Console.WriteLine()
        Console.WriteLine("Wird ein String-Array mit folgenden Elementen:")
        Dim locStringArray As String()
        locStringArray = locString.Split(New Char() {","c, ";"c, "-"c, "."c})
        For Each locStr As String In locStringArray
            Console.WriteLine(ReplaceEx(locStr, New Char() {","c, ";"c, "-"c, "."c}, _
                   Convert.ToChar(vbNullChar)).Trim)
        Next
        Console.ReadLine()
    End Sub

    Public Function ReplaceEx(ByVal str As String, ByVal SearchChars As Char(), _
                        ByVal ReplaceChar As Char) As String
        Dim locPos As Integer
        Do
            locPos = str.IndexOfAny(SearchChars)
            If locPos = -1 Then Exit Do
            If AscW(ReplaceChar) = 0 Then
                str = str.Remove(locPos, 1)
            Else
                str = str.Remove(locPos, 1).Insert(locPos, ReplaceChar.ToString)
            End If
        Loop
        Return str
    End Function
End Module
```

Wenn Sie dieses Programm laufen lassen, generiert es die folgende Ausgabe:

```
Aus der Zeile:
Einzelne, Elemente; durch, die , verschiedensten - Zeichen , getrennt.

Wird ein String-Array mit folgenden Elementen:
Einzelne
Elemente
durch
die
verschiedensten
Zeichen
getrennt
```

Iterieren durch einen String

Sie sehen, dass diese Routine für das Iterieren durch die einzelnen Buchstaben eine weitere Variation verwendet: Mit der `Chars`-Eigenschaft eines `String`-Objekts greifen Sie auf ein `Char`-Array zu, das die einzelnen Zeichen des Strings repräsentiert. Da das `String`-Objekt auch die Funktion `GetEnumerator` anbietet, gäbe es auch die folgende Möglichkeit, durch einen String zu iterieren:

```
For Each locChar As Char In "Dies ist ein String"
   'Tu irgendwas.
Next
```

Durch die Mächtigkeit des `String`-Objekts sind die Funktionen Suchen und Ersetzen des Programms erschreckend einfach zu realisieren, wie der folgende Codeausschnitt eindrucksvoll zeigt:

```
Private Sub btnCheckFound_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
                Handles btnCheckFound.Click
    For Each locFEI As FilenameEnumeratorItem In fneFiles.Items
        Dim locFilename As String = locFEI.Filename.Name
        If locFilename.IndexOf(txtSearch.Text) > -1 Then
            locFEI.Checked = True
        Else
            locFEI.Checked = False
        End If
    Next
End Sub

Private Sub btnReplaceChecked_Click(ByVal sender As System.Object,ByVal e As System.EventArgs) _
    Handles btnReplaceChecked.Click
    For Each locFEI As FilenameEnumeratorItem In fneFiles.Items
        Dim locFilename As String = locFEI.SubItems(1).Text
        If locFEI.Checked Then
            If locFilename.IndexOf(txtSearch.Text) > -1 Then
                locFilename = locFilename.Replace(txtSearch.Text, txtReplace.Text)
                locFEI.SubItems(1).Text = locFilename
            End If
        End If
    Next
End Sub
```

StringBuilder vs. String – wenn es auf Geschwindigkeit ankommt

Sie haben im Laufe des String-Abschnitts gesehen, dass Ihnen .NET mit dem Datentyp `String` ein mächtiges Werkzeug für die Bearbeitung von Zeichenketten in die Hände legt. Wenn Sie den Abschnitt über die Speicherverwaltung gelesen haben, dann werden Sie aber auch bemerkt haben, dass es um die Geschwindigkeit bei der Verarbeitung von Strings in bestimmten Szenarien nicht so gut bestellt ist. Der Grund dafür ist einfach: Strings sind unveränderlich. Wenn Sie mit Algorithmen hantieren, die Strings im Laufe ihrer Entstehungsgeschichte zeichenweise zusammensetzen, dann wird für jedes Zeichen, das zum String hinzukommt, ein komplett neuer String erstellt. Und das kostet Zeit.

Eine Alternative dazu bildet die `StringBuilder`-Klasse. Sie hat bei weitem nicht die Funktionsvielfalt eines Strings, aber sie hat einen entscheidenden Vorteil: Sie wird dynamisch verwaltet und ist damit ungleich schneller. Das heißt für Sie: Wann immer es darum geht, Strings zusammenzusetzen (indem Sie Zeichen anhängen, einfügen oder löschen), sollten Sie ein `StringBuilder`-Objekt einsetzen – gerade wenn große Datenmengen im Spiel sind.

Der Umgang mit einem `StringBuilder`-Objekt ist denkbar einfach. Um auf das Objekt zurückgreifen zu können, benötigen Sie Zugriff auf den Namespace `Text`, den Sie mit der Anweisung

```
Imports System.Text
```

in Ihre Klassen- oder Moduldatei einbinden. Sie deklarieren eine Variable einfach vom Typ `StringBuilder` und definieren sie mit einer der folgenden Anweisungen:

```
'Deklaration ohne Parameter:
Dim locSB As New StringBuilder
'Deklaration mit Kapazitätsreservierung
locSB = New StringBuilder(1000)
'Deklaration aus einem vorhandenen String
locSB = New StringBuilder("Aus einem neuen String entstanden")
'Deklaration aus String mit der Angabe einer zu reservierenden Kapazität
locSB = New StringBuilder("Aus String entstanden mit Kapazität für weitere", 1000)
```

Sie können, falls Sie das möchten, eine Ausgangskapazität bei der Definition eines `StringBuilder`-Objekts angeben. Damit wird der Platz, den Ihr `StringBuilder`-Objekt voraussichtlich benötigen wird, direkt reserviert – zusätzlicher Speicher muss zur Laufzeit nicht angefordert werden, und das spart zusätzlich Zeit.

Um den String um Zeichen zu erweitern, verwenden Sie die `Append`-Methode. Mit `Insert` können Sie weitere Zeichen in ein `StringBuilder`-Objekt einfügen. `Replace` erlaubt Ihnen das Ersetzen einer Zeichenkette durch eine andere. Und mit `Remove` haben Sie die Möglichkeit, eine bestimmbare Anzahl von Zeichen ab einer bestimmten Zeichenposition zu entfernen. Beispiel:

```
locSB.Append(" - und das wird an den String angefügt")
locSB.Insert(20, ">>das kommt irgendwo in die Mitte<<")
locSB.Replace("String", "StringBuilder")
locSB.Remove(0, 4)
```

Wenn der String komplett zusammengesetzt wurde, können Sie ihn mit der `ToString`-Funktion in einen »echten« String umwandeln:

```
'StringBuilder hat den String fertig zusammengesetzt,
'in String umwandeln.
Dim locString As String = locSB.ToString
Console.WriteLine(locString)
```

Führen Sie dieses Beispiel aus, zeigt es folgenden Text im Konsolenfenster an:

```
StringBuilder entstande>>das kommt irgendwo in die Mitte<<n mit Kapazität für
weitere - und das wird an den StringBuilder angefügt
```

Performance-Vergleich: String gegen StringBuilder

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 07\StringVsStringBuilder
```

Das Programm kreiert eine bestimmbare Anzahl von `String`-Elementen, die jeweils aus einer ebenfalls bestimmbaren Menge aus zufälligen Zeichen bestehen. Wenn Sie das Programm starten, bestimmen Sie diese Parameter:

```
Geben Sie die String-Länge eines Elements ein: 100
Geben die Anzahl der zu erzeugenden Elemente ein: 100000

Erzeugen von 100000 Stringelementen mit der String-Klasse...
Dauer: 2294 Millisekunden

Erzeugen von 100000 Stringelementen mit der StringBuilder-Klasse...
Dauer: 1111 Millisekunden
```

Sie sehen, dass bei einer Elementlänge von *100* Zeichen die Verwendung der `StringBuilder`-Klasse bereits eine Verdopplung der Geschwindigkeit mit sich bringt. Starten Sie anschließend das Programm erneut. Geben Sie für die Elementlänge *1000* ein und bestimmen Sie für die Anzahl der zu erzeugenden Elemente den Wert *10000*. Die Geschwindigkeitsausbeute ist jetzt noch beeindruckender:

```
Geben Sie die String-Länge eines Elements ein: 1000
Geben die Anzahl der zu erzeugenden Elemente ein: 10000

Erzeugen von 10000 Stringelementen mit der String-Klasse...
Dauer: 6983 Millisekunden

Erzeugen von 10000 Stringelementen mit der StringBuilder-Klasse...
Dauer: 1091 Millisekunden
```

Mit diesen Parametern ist der `StringBuilder` ca. um den Faktor 6 schneller – im Vergleich zum normalen `String`-Objekt. Je mehr Zeichen für einen String zur Laufzeit generiert werden müssen, desto mehr lohnt sich der Einsatz eines `StringBuilder`-Objekts.

Das Programm selbst greift für die Messungen übrigens auf die Klasse `StopWatch` zurück, mit der Sie auf einfache Weise die Möglichkeit haben, eine Zeitdauer in einer sehr präzisen Auflösung zu messen. Das folgende Listing zeigt seine Verwendungsweise:

```
Imports System.Text

Module StringsVsStringBuilder

    Sub Main()
        Dim locAmountElements As Integer
        Dim locAmountCharsPerElement As Integer
        Dim locVBStringElements As VBStringElements
        Dim locVBStringBuilderElements As VBStringBuilderElements

        'StringBuilderBeispiele()
        'Return

        Console.Write("Geben Sie die String-Länge eines Elements ein: ")
        locAmountCharsPerElement = Integer.Parse(Console.ReadLine)
        Console.Write("Geben die Anzahl der zu erzeugenden Elemente ein: ")
        locAmountElements = Integer.Parse(Console.ReadLine)
        Console.WriteLine()
        Console.WriteLine("Erzeugen von " & locAmountElements &
                    " Stringelementen mit der String-Klasse...")
        Dim locTimeGauge = Stopwatch.StartNew
        locVBStringElements = New VBStringElements(locAmountElements, locAmountCharsPerElement)
        locTimeGauge.Stop()
        Console.WriteLine("Dauer: " & locTimeGauge.ElapsedMilliseconds.ToString())
        locTimeGauge.Reset()
        Console.WriteLine("Erzeugen von " & locAmountElements &
                    " Stringelementen mit der StringBuilder-Klasse...")
        locTimeGauge.Start()
        locVBStringBuilderElements =
                    New VBStringBuilderElements(locAmountElements, locAmountCharsPerElement)
        locTimeGauge.Stop()
        Console.WriteLine("Dauer: " & locTimeGauge.ElapsedMilliseconds.ToString())
        locTimeGauge.Reset()
        Console.WriteLine()
        Console.ReadLine()
    End Sub

    Sub StringBuilderBeispiele()

        'Deklaration ohne Parameter:
        Dim locSB As New StringBuilder
        'Deklaration mit Kapazitätsreservierung
        locSB = New StringBuilder(1000)
        'Deklaration aus einem vorhandenen String
        locSB = New StringBuilder("Aus einem neuen String entstanden")
        'Deklaration aus String mit der Angabe einer zu reservierenden Kapazität
        locSB = New StringBuilder("Aus String entstanden mit Kapazität für weitere", 1000)

        locSB.Append(" - und das wird an den String angefügt")
        locSB.Insert(20, ">>das kommt irgendwo in die Mitte<<")
        locSB.Replace("String", "StringBuilder")
        locSB.Remove(0, 4)

        'StrinBuilder hat den String fertig zusammengesetzt,
        'in String umwandeln
        Dim locString As String = locSB.ToString
```

```
        Console.WriteLine(locString)
        Console.ReadLine()
    End Sub

End Module

Public Class VBStringElements

    Private myStrElements() As String

    Sub New(ByVal AmountOfElements As Integer, ByVal AmountChars As Integer)

        ReDim myStrElements(AmountOfElements - 1)
        Dim locRandom As New Random(DateTime.Now.Millisecond)
        Dim locString As String

        For locOutCount As Integer = 0 To AmountOfElements - 1
            locString = ""
            For locInCount As Integer = 0 To AmountChars - 1
                Dim locIntTemp As Integer = Convert.ToInt32(locRandom.NextDouble * 52)
                If locIntTemp > 26 Then
                    locIntTemp += 97 - 26
                Else
                    locIntTemp += 65
                End If
                locString += Convert.ToChar(locIntTemp).ToString
            Next
            myStrElements(locOutCount) = locString
        Next
    End Sub

End Class

Public Class VBStringBuilderElements

    Private myStrElements() As String

    Sub New(ByVal AmountOfElements As Integer, ByVal AmountChars As Integer)

        ReDim myStrElements(AmountOfElements - 1)
        Dim locRandom As New Random(DateTime.Now.Millisecond)
        Dim locStringBuilder As StringBuilder

        For locOutCount As Integer = 0 To AmountOfElements - 1
            locStringBuilder = New StringBuilder(AmountChars)
            For locInCount As Integer = 0 To AmountChars - 1
                Dim locIntTemp As Integer = Convert.ToInt32(locRandom.NextDouble * 52)
                If locIntTemp > 26 Then
                    locIntTemp += 97 - 26
                Else
                    locIntTemp += 65
                End If
                locStringBuilder.Append(Convert.ToChar(locIntTemp))
            Next
            myStrElements(locOutCount) = locStringBuilder.ToString
        Next
    End Sub

End Class
```

Der Datentyp Boolean

Der `Boolean`-Datentyp speichert binäre Zustände, also eigentlich nicht viel: Sein Wert kann entweder *falsch* oder *wahr* sein – etwas anderes kann er nicht speichern. Dieser Datentyp wird am häufigsten bei der Ausführung von bedingtem Programmcode verwendet; zu diesem Thema haben Sie bereits in Kapitel 2 alles Wesentliche erfahren.

.NET-Datentyp: `System.Boolean`

Stellt dar: einen von zwei Zuständen – `True` oder `False`.

Typliteral: keins vorhanden

Speicherbedarf: 2 Byte

Delegation an den Prozessor: ja, als `Integer`

Anmerkungen:

Wenn Sie eine boolesche Variable definieren wollen, verwenden Sie dazu die Schlüsselwörter `True` und `False` direkt und ohne Anführungszeichen im Programmtext, etwa wie im folgenden Beispiel:

```
Dim locBoolean As Boolean
locBoolean = True ' Ausdruck ist 'wahr'.
locBoolean = False ' Ausdruck ist 'falsch'.
```

Konvertieren von und in numerische Datentypen

Sie können einen booleschen Typ in einen numerischen Datentyp umwandeln.

WICHTIG Beachten Sie, dass Visual Basic bei der internen Darstellung des primitiven Datentyps `Boolean` von .NET Framework abweicht. Wenn Sie mit Visual Basic-Befehlen einen `Boolean`- in beispielsweise einen `Integer`-Datentyp umwandeln, wird der Wert `True` in –1 umgewandelt. Wenn Sie mit .NET Framework-Konvertierungen – also beispielsweise der `Convert`-Klasse – arbeiten, wird `True` dagegen zu +1 umgewandelt.

Das folgende Beispiel verdeutlicht, was gemeint ist:

```
Dim locInt As Integer = CInt(locBoolean)    ' locInt ist -1
locInt = Convert.ToInt32(locBoolean)        ' locInt ist jetzt +1!!!
Dim locLong As Long = CLng(locBoolean)      ' locLong ist -1
locLong = Convert.ToInt64(locBoolean)       ' locLong ist +1
```

Bei der umgekehrten Konvertierung ist das Verhalten der `Convert`-Klasse von .NET Framework und den Konvertierungsanweisungen von Visual Basic einerlei. Nur der Zahlenwert 0 ergibt das boolesche Ergebnis `False`, alle anderen Werte ergeben `True`, wie das folgende Beispiel zeigt.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 07\Primitives03
```

Öffnen Sie dort die Projektmappe (.SLN-Datei).

```
locBoolean = CBool(-1)                    ' locBoolean ist True.
locBoolean = CBool(0)                     ' locBoolean ist False.
locBoolean = CBool(1)                     ' locBoolean ist True.
locBoolean = Convert.ToBoolean(-1)        ' locBoolean ist True.
locBoolean = Convert.ToBoolean(+1)        ' locBoolean ist True.
locBoolean = CBool(100)                   ' locBoolean ist True.
locBoolean = Convert.ToBoolean(100)       ' locBoolean ist True.
```

Konvertierung von und in Strings

Wenn Sie einen booleschen Datentyp in eine Zeichenkette konvertieren – beispielsweise um ihren Status in einer Datei zu speichern –, wird der jeweilige Wert in eine Zeichenkette umgewandelt, die durch die statischen Nur-Lese-Eigenschaften `TrueString` und `FalseString` der `Boolean`-Struktur festgehalten sind. Sie ergeben in der aktuellen .NET Framework-Version (4.0) »True« und »False«, egal, welche Kultureinstellungen für das jeweilige .NET Framework gelten. Bei der Konvertierung von `String` in `Boolean` ergibt jede Zeichenkette außer »True« den Wert `False`.

Der Datentyp Date

Mit dem `Date`-Datentyp speichern Sie Datumswerte. Er hilft Ihnen, Zeitdifferenzen zu berechnen, Datumswerte aus Zeichenketten einzulesen (zu parsen) und wieder formatiert in Zeichenketten zurückzuverwandeln.

.NET-Datentyp: `System.DateTime`

Stellt dar: Zeiten und Uhrzeiten im Bereich vom 1.1.0001 (0 Uhr) bis 31.12.9999 (23:59:59 Uhr) mit einer Auflösung von 100 Nanosekunden (diese Einheit wird als *Tick* bezeichnet).

Typliteral: keins vorhanden

Speicherbedarf: 8 Byte

Delegation an den Prozessor: ja, als *Long*

Anmerkungen: Da `Date`-Datentypen ebenfalls zu den primitiven Datentypen zählen, sind sie direkt durch Literale im Programmcode definierbar. Allerdings gilt hier das Gleiche wie bei numerischen Werten: Das englische bzw. amerikanische Darstellungsformat zählt. Falls Sie mit dieser Kultur – was die Zeitmessung und Schreibweise anbelangt – nicht so sehr vertraut sind, lassen Sie mich kurz die Besonderheiten erklären. Datumswerte werden in der Reihenfolge Monat/Tag/Jahr vorgenommen und durch Schrägstrich und nicht durch Punkt voneinander getrennt. Diese Schreibweise kann leicht für Verwirrung sorgen (und zu totalen Fehlbuchungen führen), wenn Sie nicht darüber Bescheid wissen. Beim Datum

```
12/17/2010
```

ahnen Sie noch, dass das kein Datum in der gewohnten Schreibweise sein kann, da es keinen 17. Monat im Jahr gibt. Aber es macht schon einen Unterschied, ob Sie das Datum

```
12/06/2010
```

als 12. Juni oder 6. Dezember interpretieren. Ähnliches gilt für die Uhrzeit. Die 24-Stunden-Anzeige finden Sie in den USA vielleicht auf dem einen oder anderen Busfahrplan oder beim Militär. Ansonsten wird durch die Postfixe »AM« (für »ante meridiem«, etwa »am Vormittag«) und »PM« (»post meridiem«, etwa »am Nachmittag«) definiert, welches 3:00 Uhr beispielsweise gemeint ist. Kritisch wird es bei 12:00 (0:00 gibt es nicht!) – vielleicht haben Sie selbst schon mal die Erfahrung gemacht, ein amerikanisches Videorekordermodell zu programmieren, das nicht die gewünschte Sendung, sondern eine andere, 12 Stunden später ausgestrahlte, aufgenommen hat. 12:00 AM entspricht unserem 0:00 Uhr, also Mitternacht; 12:00 PM entspricht Mittag.

Wertzuweisungen an einen `Date`-Datentyp im Programmcode werden durch Einbetten per Nummernzeichens (»#«) realisiert. Die folgenden Beispiele zeigen, wie es geht:

```
Dim Mitternacht As Date = #12:00:00 AM#
Dim Mittag As Date = #12:00:00 PM#
Dim Silvester As System.DateTime = #12/31/2010#
Dim ZeitFürSekt As System.DateTime = #12/31/2010 11:58:00 PM#
Dim ZeitFürAspirin As System.DateTime = #1/1/2011 11:58:00 AM#
```

Der halbwegs intelligente Editor hilft Ihnen dabei ein wenig, das korrekte Format zu finden. Den Ausdruck

```
#0:00#
```

wandelt er beispielsweise selbstständig in

```
#12:00:00 AM#
```

um. Er ergänzt ebenfalls fehlende Sekundeneingaben oder Nullen, wenn die Eingabe eines Bereichs nur einstellig erfolgte. Sie können Uhrzeiten auch im 24-Stunden-Format eingeben; der Editor wandelt diese Zeit dann automatisch ins 12-Stunden-Format um.

Rechnen mit Zeiten und Datumswerten – TimeSpan

Das Besondere am `Date`-Datentyp ist, dass er das Errechnen von Zeitdifferenzen erlaubt. Um jedoch sinnvolle Ergebnisse, beispielsweise die Differenz zweier Datumswerte oder Zeitwerte darzustellen, ist der `Date`-Datentyp selbst nicht sonderlich geeignet. Aus diesem Grund gibt es einen weiteren Datentyp – `TimeSpan` –, der Ergebnisse von Zeitberechnungen aufnimmt. Er selbst zählt allerdings nicht zu den primitiven Datentypen von .NET.

Der Umgang mit diesem Datentyp ist sehr einfach: Sie können entweder einen Datumswert von einem anderen subtrahieren, um die dazwischenliegende Zeitspanne zu ermitteln. Oder Sie erstellen einen `TimeSpan`-Datentyp und verwenden ihn, um einen bestimmten Zeitabschnitt auf ein Datum zu addieren oder es von ihm abzuziehen. Ein paar Beispielcodezeilen sollen das verdeutlichen.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 07\DateTime
```

Öffnen Sie dort die Projektmappe (.SLN-Datei).

```
Dim locDate1 As Date = #3:15:00 PM#
Dim locDate2 As Date = #4:23:32 PM#
Dim locTimeSpan As TimeSpan = locDate2.Subtract(locDate1)
Console.WriteLine("Der Zeitunterschied zwischen {0} und {1} beträgt", _
                  locDate1.ToString("HH:mm:ss"), _
                  locDate2.ToString("HH:mm:ss"))
Console.WriteLine("{0} Sekunde(n) oder", locTimeSpan.TotalSeconds)
Console.WriteLine("{0} Minute(n) und {1} Sekunde(n) oder", _
                   Math.Floor(locTimeSpan.TotalMinutes), _
                   locTimeSpan.Seconds)
Console.WriteLine("{0} Stunde(n), {1} Minute(n) und {2} Sekunde(n) oder", _
                   Math.Floor(locTimeSpan.TotalHours), _
                   locTimeSpan.Minutes, locTimeSpan.Seconds)
Console.WriteLine("{0} Ticks", _
                   locTimeSpan.Ticks)
```

Bibliothek mit brauchbaren Datumsrechenfunktionen

Sie finden im gleichen Beispiel eine Klassendatei namens *DateCalcHelper.vb*, die eine statische Klasse gleichen Namens enthält. Diese Klasse stellt einige, wie ich meine, ganz brauchbare Funktionen zur Verfügung, die einerseits die Berechnung bestimmter relativer Zeitpunkte vereinfacht und andererseits das Rechnen mit Datumswerten verdeutlicht.

Die Klasse erklärt sich durch die XML-Kommentare selbst – das komplette Listing dieser Klasse finden Sie im Folgenden. Wenn Sie eigene Programme entwickeln, die intensiv von Berechnungen relativer Zeitpunkte Gebrauch machen, fügen Sie diese Codedatei einfach Ihrem Projekt (oder der Assembly Ihres Projekts) hinzu.

Zur besseren Orientierung sind die Funktionserklärungen und Methodennamen der einzelnen Funktionen im folgenden Listing in Fettschrift gesetzt:

```
Public NotInheritable Class DateCalcHelper

    ''' <summary>
    ''' Errechnet das Datum, das dem 1. des Monats entspricht,
    ''' der sich aus dem angegebenen Datum ergibt.
    ''' </summary>
    ''' <param name="CurrentDate">Datum, dessen Monat für die Berechnung zugrunde gelegt wird.</param>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public Shared Function FirstDayOfMonth(ByVal CurrentDate As Date) As Date
        Return New Date(CurrentDate.Year, CurrentDate.Month, 1)
    End Function

    ''' <summary>
    ''' Errechnet das Datum, das dem Letzten des Monats entspricht,
    ''' der sich aus dem angegebenen Datum ergibt.
    ''' </summary>
    ''' <param name="CurrentDate">Datum, dessen Monat für die Berechnung zugrunde gelegt wird.</param>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public Shared Function LastDayOfMonth(ByVal CurrentDate As Date) As Date
        Return New Date(CurrentDate.Year, CurrentDate.Month, 1).AddMonths(1).AddDays(-1)
    End Function
```

```
''' <summary>
''' Errechnet das Datum, das dem 1. des Jahres entspricht,
''' das sich aus dem angegebenen Datum ergibt.
''' </summary>
''' <param name="CurrentDate">Datum, dessen Jahr für die Berechnung zugrunde gelegt wird.</param>
''' <returns></returns>
''' <remarks></remarks>
Public Shared Function FirstOfYear(ByVal CurrentDate As Date) As Date
    Return New Date(CurrentDate.Year, 1, 1)
End Function

''' <summary>
''' Errechnet das Datum, das dem ersten Montag der ersten Woche des Monats entspricht,
''' der sich aus dem angegebenen Datum ergibt.
''' </summary>
''' <param name="CurrentDate">Datum, dessen Woche für die Berechnung zugrunde gelegt wird.</param>
''' <returns></returns>
''' <remarks></remarks>
Public Shared Function MondayOfFirstWeekOfMonth(ByVal CurrentDate As Date) As Date
    Dim locDate As Date = FirstDayOfMonth(CurrentDate)
    If Weekday(locDate) = DayOfWeek.Monday Then
        Return locDate
    End If
    Return locDate.AddDays(6 - Weekday(CurrentDate))
End Function

''' <summary>
''' Errechnet das Datum, das dem Montag der Woche entspricht,
''' die sich aus dem angegebenen Datum ergibt.
''' </summary>
''' <param name="CurrentDate">Datum, dessen Woche für die Berechnung zugrunde gelegt wird.</param>
''' <returns></returns>
''' <remarks></remarks>
Public Shared Function MondayOfWeek(ByVal CurrentDate As Date) As Date
    If Weekday(CurrentDate) = DayOfWeek.Monday Then
        Return CurrentDate
    Else
        Return CurrentDate.AddDays(-Weekday(CurrentDate) + 1)
    End If
End Function

''' <summary>
''' Errechnet das Datum, das dem ersten Montag der zweiten Woche des Monats entspricht,
''' der sich aus dem angegebenen Datum ergibt.
''' </summary>
''' <param name="CurrentDate">Datum, dessen Woche für die Berechnung zugrunde gelegt wird.</param>
''' <returns></returns>
''' <remarks></remarks>
Public Shared Function MondayOfSecondWeekOfMonth(ByVal currentDate As Date) As Date
    Return MondayOfFirstWeekOfMonth(currentDate).AddDays(7)
End Function
```

```
    ''' <summary>
    ''' Errechnet das Datum, das dem ersten Montag der letzten Woche des Monats entspricht,
    ''' der sich aus dem angegebenen Datum ergibt.
    ''' </summary>
    ''' <param name="CurrentDate">Datum, dessen Woche für die Berechnung zugrunde gelegt wird.</param>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public Shared Function MondayOfLastWeekOfMonth(ByVal CurrentDate As Date) As Date
        Dim locDate As Date = FirstDayOfMonth(CurrentDate).AddDays(-1)
        If Weekday(locDate) = DayOfWeek.Monday Then
            Return locDate
        End If
        Return locDate.AddDays(-Weekday(CurrentDate) + 1)
    End Function

    ''' <summary>
    ''' Ergibt das Datum des nächsten Arbeitstages.
    ''' </summary>
    ''' <param name="CurrentDate">Datum der Berechnungsgrundlage</param>
    ''' <param name="WorkOnSaturdays">True, wenn Samstag Arbeitstag ist.</param>
    ''' <param name="WorkOnSundays">True, wenn Sonntag Arbeitstag ist.</param>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public Shared Function NextWorkday(ByVal CurrentDate As Date, ByVal WorkOnSaturdays As Boolean, _
             ByVal WorkOnSundays As Boolean) As Date
        CurrentDate = CurrentDate.AddDays(1)
        If Weekday(CurrentDate) = DayOfWeek.Saturday And Not WorkOnSaturdays Then
            CurrentDate = CurrentDate.AddDays(1)
        End If
        If Weekday(CurrentDate) = DayOfWeek.Sunday And Not WorkOnSundays Then
            CurrentDate = CurrentDate.AddDays(1)
        End If
        Return CurrentDate
    End Function

    ''' <summary>
    ''' Ergibt das Datum des vorherigen Arbeitstages.
    ''' </summary>
    ''' <param name="CurrentDate">Datum der Berechnungsgrundlage</param>
    ''' <param name="WorkOnSaturdays">True, wenn Samstag Arbeitstag ist.</param>
    ''' <param name="WorkOnSundays">True, wenn Sonntag Arbeitstag ist.</param>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public Shared Function PreviousWorkday(ByVal CurrentDate As Date, ByVal WorkOnSaturdays As Boolean,
_
            ByVal WorkOnSundays As Boolean) As Date
        CurrentDate = CurrentDate.AddDays(-1)
        If Weekday(CurrentDate) = DayOfWeek.Sunday And Not WorkOnSundays Then
            CurrentDate = CurrentDate.AddDays(-1)
        End If
        If Weekday(CurrentDate) = DayOfWeek.Saturday And Not WorkOnSaturdays Then
            CurrentDate = CurrentDate.AddDays(-1)
        End If
        Return CurrentDate
    End Function
End Class
```

Zeichenketten in Datumswerte wandeln

Genau wie die numerischen primitiven Datentypen können Sie Zeichenketten auch in einen `Date`-Datentyp umwandeln lassen. Allerdings stellt Ihnen der `Date`-Datentyp zwei Funktionen zur Verfügung, die eine Zeichenkette analysieren und den eigentlichen Datumswert daraus bilden: `Parse` und `ParseExact`.

Umwandlungen mit Parse

Wenn Sie `Parse` verwenden, versucht der Parser jeden Trick, den er kennt, um ein Datum, eine Uhrzeit oder eine Kombination aus beiden in einen Zeitwert umzuwandeln, wie das folgende Beispiel zeigt:

```
Dim locToParse As Date
locToParse = Date.Parse("13.12.10") ' OK, deutsche Grundeinstellung wird verarbeitet.
locToParse = Date.Parse("6/7/10")   ' OK, aber deutsch trotz "/" wird angewendet.
locToParse = Date.Parse("13/12/10") ' OK, wie oben.
locToParse = Date.Parse("06.07")    ' OK, wird um Jahreszahl erweitert.
locToParse = Date.Parse("06,07,10") ' OK, Komma ist akzeptabel.
locToParse = Date.Parse("06,07")    ' OK, Komma ist akzeptabel; Jahreszahl wird ergänzt.
'locToParse = Date.Parse("06072010") ' --> Exception: wurde nicht als gültiges Datum erkannt!
'locToParse = Date.Parse("060705")   ' --> Exception: wurde nicht als gültiges Datum erkannt!
locToParse = Date.Parse("6,7,4")    ' OK, Komma wird akzeptiert; führende Nullen werden ergänzt.

locToParse = Date.Parse("14:00")    ' OK, 24-Stunden-Darstellung wird akzeptiert.
locToParse = Date.Parse("PM 11:00") ' OK, PM darf vorne...
locToParse = Date.Parse("11:00 PM") ' ...und auch hinter der Zeitangabe stehen.
'locToParse = Date.Parse("12,00 PM") ' --> Exception: wurde nicht als gültiges Datum erkannt!

'Beide Datum- Zeitkombinationen funktionieren:
locToParse = Date.Parse("6.7.10 13:12")
locToParse = Date.Parse("6,7,10 11:13 PM")
```

Sie sehen hier aber auch, dass ein in Deutschland sehr übliches Eingabeformat nicht erkannt wird – wenn Sie nämlich die einzelnen Wertegruppen des Datums ohne irgendein Trennzeichen hintereinander schreiben. Doch auch dafür gibt es eine Lösung.

Umwandlungen mit ParseExact

Wenn `Parse` – so flexibel es auch sein mag – versagt, haben Sie die Möglichkeit, ein Erkennungsmuster für die Eingabe vorzugeben, indem Sie die Methode `ParseExact` benutzen, um die Zeichenkettenumwandlung vorzunehmen. `ParseExact` sollten Sie auch dann verwenden, wenn Sie die Eingabe eben nicht so flexibel gestalten möchten, wie `Parse` es vorsieht.

Insbesondere wenn es Zeitwerte und Datumswerte zu unterscheiden gilt, ist `ParseExact` der richtige Kandidat. In einem Feld, in dem vom Benutzer Ihres Programms eine Zeiteingabe verlangt wird, weiß Ihr Programm genau, dass die Eingabe

```
23,12
```

die Uhrzeit 23:12:00 meint und nicht den 23.12.2010. `Parse` kann den Kontext natürlich nicht erkennen und wird in diesem Fall nicht funktionieren.

`ParseExact` benötigt, neben dem zu analysierenden String, mindestens zwei weitere Parameter. Zum einen ist dies eine Zeichenkette, die das spezifische Erkennungsmuster enthält, und zum anderen ein so genannter *Formatprovider,* der weitere Formatierungsvorschriften vorgibt. Es gibt verschiedene Formatprovider, die Sie dafür verwenden können. Wichtig ist, dass Sie auf diese nur zugreifen können, wenn Sie zuvor folgende Zeile zur Einbindung des erforderlichen Namespaces am Anfang Ihres Moduls oder Ihrer Klassendatei eingefügt haben:

```
Imports System.Globalization
```

Die einfachste Version, um eine Uhrzeit als Uhrzeit zu erkennen, wenn sie im oben stehenden Format eingegeben würde, sähe beispielsweise folgendermaßen aus:

```
locToParse = Date.ParseExact("12,00", "HH,mm", CultureInfo.CurrentCulture)
```

Die Zeichenfolge »HH« besagt, dass Uhrzeiten (Hour = Stunde) im 24-Stunden-Format akzeptiert werden. Gäben Sie hier zwei kleine »h« ein, würde der Parser nur das englische 12-Stunden-Format akzeptieren. Mit dem anschließenden Komma geben Sie das Trennzeichen vor und anschließend die Minuten mit der Zeichenfolge »mm«.

Nun passiert es in der Praxis recht selten, dass sich Benutzer an bestimmte Vorgaben halten, und darum sollte Ihr Programm in der Lage sein, mehrere Versionen von Zeiteingabeformaten zu erkennen. Aus diesem Grund können Sie mit der `ParseExact`-Funktion gleich eine ganze Reihe von möglichen Formaten vorgeben, anhand derer der Parser die Umwandlung vornehmen kann. In diesem Fall definieren Sie einfach ein `String`-Array mit den zugelassenen Formaten und übergeben es anschließend der `ParseExact`-Methode zusammen mit der zu analysierenden Zeichenfolge. Wenn Sie sich zu dieser Methode des Parsens entschließen, müssen Sie jedoch noch einen weiteren Parameter bestimmen, der die Flexibilität des Parsens regelt (beispielsweise ob Leerzeichen im zu analysierenden String enthalten sein dürfen, die aber ignoriert werden sollen). Diese Angabe wird durch einen Parameter vom Typ `DateTimeStyles` geregelt, der die in Tabelle 7.5 aufgelisteten Einstellungen zulässt.

Membername	Beschreibung	Wert
`AdjustToUniversal`	Bestimmt, dass das Datum und die Zeit in koordinierte Weltzeit bzw. Greenwich Mean Time[11] (GMT) (Deutschland –1 Stunde) konvertiert werden müssen	16
`AllowInnerWhite`	Bestimmt, dass zusätzliche Leerzeichen innerhalb der Zeichenfolge beim Analysieren ignoriert werden, es sei denn, die `DateTimeFormatInfo`-Formatmuster enthalten Leerzeichen	4
`AllowLeadingWhite`	Bestimmt, dass vorangestellte Leerzeichen während der Analyse ignoriert werden, es sei denn, die `DateTimeFormatInfo`-Formatmuster enthalten Leerzeichen	1
`AllowTrailingWhite`	Bestimmt, dass nachgestellte Leerzeichen während der Analyse ignoriert werden, es sei denn, die `DateTimeFormatInfo`-Formatmuster enthalten Leerzeichen	2 ▶

[11] Zum besseren Mitreden: Die korrekte Aussprache lautet »Grännitsch Miehn Teim«. Und auch wenn es sich so ausgesprochen wie ein irischer Dialekt anhört – auch Engländer aus Oxford sprechen es so aus.

Membername	Beschreibung	Wert
AllowWhiteSpaces	Bestimmt, dass zusätzliche Leerzeichen, die sich an einer beliebigen Stelle in der Zeichenfolge befinden, während der Analyse ignoriert werden müssen, es sei denn, die DateTimeFormatInfo-Formatmuster enthalten Leerzeichen. Dieser Wert stellt eine Kombination aus dem AllowLeadingWhite, AllowTrailingWhite und AllowInnerWhite dar	7
NoCurrentDateDefault	Datum und Zeit sind untrennbar im Date-Datentyp vereint. Auch wenn Sie nur eine Zeit zuweisen, spiegelt ein Date-Wert immer auch ein gültiges Datum wider. Diese Einstellung bestimmt, dass die DateTime.Parse-Methode und die DateTime.ParseExact-Methode das Datum nach dem gregorianische Kalender mit Jahr = 1, Monat = 1 und Tag = 1 zugrunde legen, wenn die analysierte Zeichenfolge nur die Uhrzeit und nicht das Datum enthält. Wenn dieser Wert nicht verwendet wird, wird vom gerade aktuellen Datum ausgegangen.	8
None	Bestimmt, dass die Standardformatierungsoptionen verwendet werden müssen. Dies ist das Standardformat für DateTime.Parse und DateTime.ParseExact.	0

Tabelle 7.5 Diese erweiterten Einstellungen können Sie mit *ParseExact* anwenden

Die folgenden Codezeilen zeigen, wie Sie ParseExact für die Umwandlung von Zeichenketten in Datumswerte mit geregelten Vorgaben für zugelassene Datumsformate einsetzen können:

```
Imports System.Globalization

Module Module1

    Sub Main()

        Dim locToParseExact As Date
        Dim locZeitenMuster As String() = {"H,m", "H.m", "ddMMyy", "MM\/dd\/yy"}

        'Funktioniert, ist im Uhrzeitenmuster.
        locToParseExact = Date.ParseExact("12,00", _
                          locZeitenMuster, _
                          CultureInfo.CurrentCulture, _
                          DateTimeStyles.AllowWhiteSpaces)

        'Funktioniert, ist im Uhrzeitenmuster, und "Whitespaces" sind erlaubt.
        locToParseExact = Date.ParseExact(" 12 , 00 ", _
                          locZeitenMuster, _
                          CultureInfo.CurrentCulture, _
                          DateTimeStyles.AllowWhiteSpaces)

        'Funktioniert nicht, ist zwar im Uhrzeitenmuster, es sind aber keine "Whitespaces" erlaubt.
        'locToParseExact = Date.ParseExact(" 12 , 00 ", _
        '                   locZeitenMuster, _
        '                   CultureInfo.CurrentCulture, _
        '                   DateTimeStyles.None)

        'Funktioniert, ist im Uhrzeitenmuster.
        locToParseExact = Date.ParseExact("1,2", _
                          locZeitenMuster, _
                          CultureInfo.CurrentCulture, _
                          DateTimeStyles.None)
        'Funktioniert, ist im Uhrzeitenmuster.
```

```
        'Das Datum entspricht aber dem 1.1.0001 und wird
        'als Tooltip deswegen nicht mitangezeigt, im Gegensatz
        'zu allen anderen hier gezeigten Beispielen.
        locToParseExact = Date.ParseExact("12.2",
                          locZeitenMuster,
                          CultureInfo.CurrentCulture,
                          DateTimeStyles.NoCurrentDateDefault)

        'Funktioniert, ist nicht im Uhrzeitenmuster, da hier Sekunden mit im Spiel sind
        'locToParseExact = Date.ParseExact("12,2,00",
        '                   locZeitenMuster,
        '                   CultureInfo.CurrentCulture,
        '                   DateTimeStyles.NoCurrentDateDefault)

        'Funktioniert nicht, ist mit Doppelpunkt nicht im Uhrzeitenmuster.
        'locToParseExact = Date.ParseExact("1:20",
        '                   locZeitenMuster,
        '                   CultureInfo.CurrentCulture,
        '                   DateTimeStyles.None)

        'Funktioniert jetzt, da im Zeitenmuster als Datum hinterlegt.
        '(drittes Element im String-Array)
        locToParseExact = Date.ParseExact("241205",
                          locZeitenMuster,
                          CultureInfo.CurrentCulture,
                          DateTimeStyles.AllowWhiteSpaces)

        'Funktioniert mit Übernahme im englisch-amerikanischen Format,
        'da durch die Schrägstriche und die vorgegebene Reihenfolge der Gruppen definiert.
        '(viertes Element im String-Array).
        locToParseExact = Date.ParseExact("12/24/05",
                          locZeitenMuster,
                          CultureInfo.CurrentCulture,
                          DateTimeStyles.AllowWhiteSpaces)
    End Sub

End Module
```

Beachten Sie bei der Definition von Schrägstrichen als Gruppentrennzeichen, dass Sie ihnen jeweils einen Backslash voransetzen, damit der einfache Schrägstrich nicht als Steuerzeichen behandelt wird.

.NET-Äquivalente primitiver Datentypen

Trotz ihrer festen Verankerung in der Sprache gibt es für jeden der primitiven Datentypen ein .NET-Äquivalent. Die folgende Tabelle gibt Ihnen Auskunft darüber:

Primitiver Datentyp in VB	.NET-Datentyp Äquivalent
Byte	System.Byte
SByte	System.SByte
Short	System.Int16
UShort	System.UInt16
Integer	System.Int32 ▶

Primitiver Datentyp in VB	.NET-Datentyp Äquivalent
UInteger	System.UInt32
Long	System.Int64
ULong	System.UInt64
Single	System.Single
Double	System.Double
Decimal	System.Decimal
Boolean	System.Boolean
Date	System.DateTime
Char	System.Char
String	System.String

Tabelle 7.6 Primitive Visual Basic-Datentypen und ihre .NET-Äquivalente

Das bedeutet: Es ist vollkommen egal, ob Sie beispielsweise einen 32-Bit-Integer mit

```
Dim loc32BitInteger as Integer
```

oder mit

```
Dim loc32BitInteger as Systen.Int32
```

deklarieren. Die Objektvariable `loc32BitInteger` ist in beiden Fällen nicht nur vom gleichen, sondern vom selben[12] Typ. Wenn Sie sich den erzeugten IML-Code aus dem kleinen Programm

```
Public Shared Sub main()

    Dim locDate As Date = #12/14/2003#
    Dim locDate2 As DateTime = #12/14/2003 12:13:22 PM#
    If locDate > locDate2 Then
        Console.WriteLine("locDate ist größer als locDate2")
    Else
        Console.WriteLine("locDate2 ist größer als locDate")
    End If
```

anschauen, werden Sie anhand der generierten Codezeilen

```
.method public static void  main() cil managed
{
  // Code size       75 (0x4b)
  .maxstack  2
  .locals init ([0] valuetype [mscorlib]System.DateTime locDate,
```

[12] Und zwar in der ursprünglichen Bedeutung des Unterschiedes von »selber« und »gleicher« – Sie und ich können zu einer bestimmten Zeit nie dasselbe, nur das gleiche Auto fahren.

```
          [1] valuetype [mscorlib]System.DateTime locDate2,
          [2] bool VB$CG$t_bool$S0)
  IL_0000:  nop
  IL_0001:  ldc.i8     0x8c58fec59f98000
  IL_000a:  newobj     instance void [mscorlib]System.DateTime::.ctor(int64)
  IL_000f:  nop
  IL_0010:  stloc.0
  IL_0011:  ldc.i8     0x8c59052cd35dd00
  IL_001a:  newobj     instance void [mscorlib]System.DateTime::.ctor(int64)
  IL_001f:  nop
  IL_0020:  stloc.1
  IL_0021:  ldloc.0
  IL_0022:  ldloc.1
.
.
.
```

feststellen, dass diese Tatsache zutrifft. Beide lokalen Variablen sind, wie in den fett markierten Zeilen zu sehen, als `System.DateTime`-Typ deklariert worden.

GUIDs

GUID ist die Abkürzung von Globally Unique Identifier – in etwa: global-eindeutige Kennung. »Global« im Begriff meint dabei die Eigenschaft einer GUID, dass zwei Generatoren einer GUID voneinander nichts wissen und räumlich voneinander getrennt sind und diese dennoch mit an Sicherheit grenzender Wahrscheinlichkeit keine zwei gleichen Kennungen produzieren.

GUIDs werden durch 16 Bytes (128 Bit) gespeichert, und werden in der Regel im Format {XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX} dargestellt, wobei ein »X« in dieser schematischen Darstellung einer hexadezimalen Ziffer zwischen 0 und F entspricht.

In .NET entspricht der Name des Datentyps seiner Abkürzung.

HINWEIS Um eine neue GUID zu erstellen, bedienen Sie sich nicht des Konstruktors. Da eine GUID ein Wertetyp ist, würde das auch gar keinen Sinn ergeben, denn die Verwendung des Standardkonstruktors (also des parameterlosen) mit einem Wertetyp führt nur dazu, dass dieser Wertetyp nur kurz auf dem Managed Heap erzeugt und dann sofort verworfen wird. Um eine neue GUID zu erstellen, verwenden Sie stattdessen die statische Funktion `NewGuid`, etwa wie im folgenden Beispiel:

```
Dim t As Guid
t = Guid.NewGuid
```

GUIDs finden mit Vorliebe als Primärschlüssel in Datenbanken Anwendung. Da die Wahrscheinlichkeit nahezu gegen null geht, dass zwei Computersysteme identische GUIDs generieren, eignen sich Datenbanktabellen mit Fremdschlüsselbeziehungen auf der Basis von GUIDs hervorragend zur Synchronisation mit verschiedenen Datenbanken, die aus technischen Gründen nicht ständig miteinander verbunden sein können.

Der Datentyp `Guid` verfügt über einen parametrisierten Konstruktor, der die Definition einer bestimmten (nicht neuen!) GUID unter anderem über eine Zeichenkette erlaubt. Diese Verwendung kann immer dann von Vorteil sein, wenn beispielsweise eine Komponente (Klasse, Struktur), die Sie entwickelt haben, immer eine eindeutige Kennung und immer dieselbe Kennung zurückliefern soll.

So könnte die folgende automatisch implementierte Eigenschaft einer Klasse zum Beispiel eine `UniqueID`-Eigenschaft implementieren:[13]

```
Public Class EineKlasse

    'Eigenschaft liefert immer die gleiche Guid zurück:
    Property UniqueID As Guid = New Guid("{46826D55-6FDD-44FA-BADE-515E04770816}")
End Class
```

TIPP Sie brauchen sich nicht selbst eine Anwendung zu programmieren, die Ihnen mithilfe der statischen Funktion `NewGuid` eine neue GUID erstellt, wenn Sie sie wie oben benötigen. Stattdessen können Sie sich einer Funktion von Visual Studio bedienen: Wählen Sie aus dem Menü *Extras* den Befehl *GUID erstellen*. Im Dialog, der jetzt erscheint, wählen Sie *Registrierungsformat*, und anschließend klicken Sie auf *Kopieren* (Abbildung 7.2). Wechseln Sie zurück in den Code-Editor, und fügen Sie den Inhalt der Zwischenablage als Parameter für den `Guid`-Konstruktor ein, wie im Beispiel oben zu sehen.

Abbildung 7.2 Mit diesem Dialog der Visual Studio-IDE können Sie auf einfache Weise eine GUID-Konstante erstellen

[13] Mehr zu Klassen und Eigenschaften erfahren Sie in Teil B in den entsprechenden Kapiteln.

Konstanten und Nur-Lesen-Felder (Read-only-Feldvariablen)

Neben den Variablentypen, die Sie in diesem Kapitel kennen gelernt haben, gibt es in Visual Basic zwei »Wertspeichertypen«, die sich auf den ersten Blick recht ähnlich sehen, *Konstanten* und *Nur-Lesen-Felder*. Doch diese beiden Wertspeichertypen, die ja im Laufe ihres Programmlebens nur ein einziges Mal definiert werden, verhalten sich unter bestimmten Umständen ganz verschieden, und ihre Inhalte werden auch auf völlig andere Art und Weise gespeichert.

Bevor wir auf die Unterschiede kommen, werfen wir einen Blick auf ihren gemeinsamen Nenner: Sie verwenden Nur-Lesen-Felder und Konstanten in ähnlichem Kontext, nämlich immer dann, wenn Sie bestimmte Werte an mehreren Stellen Ihres Programms verwenden müssen, zum Beispiel, wenn Sie den Namen Ihres Programms oder ein »Verfallsdatum« für das Ablaufen einer Demoversion benötigen, die Sie an mehreren Stellen innerhalb Ihres Programms abfragen. Damit Sie dann nicht Ihren Quellcode an unzähligen Stellen anpassen müssen, sollte sich doch irgendwann einmal die Notwendigkeit einer Änderung ergeben, definieren Sie diesen Wert als Konstante oder Nur-Lesen-Feld zentral und verwenden dann den Feld- bzw. Konstantennamen an den Stellen im Programm, an denen er benötigt wird.

Natürlich müssen Sie dafür sorgen, dass dieser Wert nicht überschrieben werden darf, und deswegen gilt für Konstanten und für Nur-Lesen-Felder gleichermaßen: Sie können nicht verändert werden.

Konstanten

Konstanten definieren Sie mit dem Schlüsselwort `Const` auf Modulebene. Modulebene bedeutet, dass Sie eine Konstante in einer Klasse, in einem Modul oder einer Struktur definieren können. Sie definieren eine Konstante syntaxmäßig ansonsten genauso wie eine Variable, nur, dass Sie zusätzlich das Schlüsselwort `Const` verwenden. Zum Beispiel:

```
Public Const ANWENDUNGSNAME As String = "Typdemo"
```

oder

```
Private Const ABLAUFDATUM as Date=#12/31/2010#
```

Auf diese Weise können Sie natürlich auch andere Konstanten definieren – Sie brauchen lediglich den entsprechenden Typ hinter der `As`-Klausel anzugeben.

Doch aufgepasst!

Sie können als Konstante nur wirklich Konstanten definieren. Auch wenn zum Beispiel der Abruf des Rückgabeergebnisses einer Methode wie beispielsweise `Date.MaxValue` (für das höchste Datum, das dieser Wert darstellen kann) eigentlich den Charakter einer Konstante hat, führt die folgende Zeile dennoch zu einem Compiler-Fehler:

```
Private Const ABLAUFDATUM as Date=Date.MaxValue
```

In diesem Fall sollten Sie lieber auf ein Nur-Lesen-Feld zurückgreifen, wie es im folgenden Abschnitt beschrieben wird.

Und noch ein weiterer Punkt ist wichtig, und zwar bei der Verwendung von öffentlichen Konstanten in mehreren Assemblys:

WICHTIG Damit könnte man meinen, dass man Konstanten auch als Feldvariablen »missbrauchen« könnte, die nur lesbar sind. Das kann allerdings fürchterlich nach hinten losgehen, wenn auf öffentliche Konstanten in anderen Assemblys zugegriffen wird.

Wenn Sie eine Konstante mit einem Wert versehen, wird nämlich niemals ein konkreter Wert erzeugt und zur Laufzeit des Programms gespeichert. Konstanten benötigen so gesehen also überhaupt keinen Speicherplatz zur Laufzeit – weder auf dem Managed Heap noch auf dem Stack, noch innerhalb irgendwelcher Prozessorregister. Der eigentliche Wert versteckt sich vielmehr in den so genannten Metadaten der Assembly, in der die Konstante definiert wird. Und das kann tragischerweise zu Problemen bei Versionsunterschieden führen.

Konstanten können nämlich, eben weil sie nur in den Metadaten Ihrer Assembly gespeichert werden, nur zur Kompilierungszeit Ihrer Anwendung ausgewertet werden. Wenn Sie also beispielsweise in Assembly A eine öffentliche Konstante definieren und auf diese von einer Assembly B aus zugreifen, benötigen Sie Assembly A im Grunde genommen nur während der Erstellung von Assembly B. Der Compiler schaut nämlich vereinfacht ausgedrückt lediglich in Assembly A nach, *während* er Assembly B erstellt, und überträgt die Werte der Konstanten in Assembly B an den Stellen, an denen es notwendig ist. Bräuchten Sie die Assembly A ausschließlich zum Abrufen der Konstanten, könnten Sie sie im Anschluss genauso gut entsorgen. Und dabei liegt das Problem. Da nicht wirklich zur Laufzeit auf die Assembly A zugegriffen wird, sondern nur zur Kompilierungszeit, ist es nicht ausreichend, Assembly A auszutauschen, falls sich der dort definierte Wert der Konstante ändert. Da Assembly B nämlich nur zur Kompilierungszeit auf Assembly A zugegriffen hat, bekommt sie davon nichts mit.

Aus diesem Grund müssen Sie mit öffentlichen Konstanten vorsichtig sein und eine Anwendung grundsätzlich inklusive aller beteiligten Assemblys neu erstellen, wenn sich eine wie hier beschriebene Konstellation ergibt. Möchten Sie dieses Verhalten bei diesem Szenario von vornherein vermeiden, sollten Sie alternativ auf so genannte Nur-Lesen-Eigenschaften zurückgreifen. Den Themenkomplex *Eigenschaften*, der erklärt wie das funktioniert, finden Sie in Kapitel 10 beschrieben.

Nur-Lesen-Felder

Nur-Lesen-Felder dienen ebenfalls zur Definition von konstanten Werten. Auch sie werden ausschließlich auf Modulebene (Klasse, Struktur, Modul) definiert, und sie unterscheiden sich von einer üblichen Variablendeklaration dadurch, dass ihnen das Schlüsselwort `ReadOnly` vorangestellt wird, also zum Beispiel:

```
Private ReadOnly THEDATE As Date=#07/24/1969#
Friend ReadOnly MAXDATE As Date=Date.MaxValue
```

Oder auch:

```
Public ReadOnly ERLAUBTE_STÄDTE As New List(Of String) From {"Lippstadt", "Las Vegas",
                                                              "Kempen", "Los Angeles"}
```

Sie sehen, dass Sie bei Nur-Lesen-Feldern anders als bei »richtigen« Konstanten auch Programmcode bei der Zuweisung ausführen können.

WICHTIG Wichtig ist, dass diese Zuweisung ausschließlich im Konstruktor der Klasse, der Struktur oder des Moduls stattfinden darf. Diese Vorschrift ist bei den oben stehenden Beispielen natürlich ebenfalls erfüllt, da der Compiler die Definitionen der Felder automatisch in den Konstruktor verschiebt (dazu mehr im entsprechenden Abschnitt von Kapitel 10).

Durchaus erlaubt ist also auch die folgende Konstruktion:

```
Public Class Test
    Friend ReadOnly MAXDATE As Date
    Public ReadOnly ERLAUBTE_STÄDTE As List(Of String)

    Sub New()
        'Durchaus, aber nur einmal erlaubt!
        MAXDATE = Date.MaxValue
        ERLAUBTE_STÄDTE = New List(Of String) From {"Lippstadt", "Las Vegas", "Kempen", "Los Angeles"}
    End Sub
.
.
.
```

In einer weiteren Methode dürften Sie hingegen die Variablen, die Sie als Nur-Lesen-Felder definiert haben, kein zweites Mal neu definieren. Der folgende Code …

```
    Sub NeueMethode()
        MAXDATE = Date.MaxValue
        ERLAUBTE_STÄDTE = New List(Of String) From {"München", "Paris", "London", "Seattle"}
```

…würde deswegen zum Fehler »Eine Read-only-Variable kann nicht das Ziel einer Zuweisung sein.« führen.

WICHTIG Doch aufgepasst: Bei Objekten, die durchaus, wie wir an der Auflistung gesehen haben, als Nur-Lesen-Felder definiert werden können, verbietet die Definition als `ReadOnly` lediglich die komplette Neuzuweisung. Da es sich bei der Variablen in diesem Fall aber um einen Zeiger auf ein Objekt handelt, spricht natürlich nichts dagegen, den Inhalt des Objekts zu manipulieren. Der folgende Code ist deswegen wiederum durchaus statthaft – ob er sinnvoll ist, steht hingegen auf einem anderen Blatt Papier:

```
        'Aber das geht:
        ERLAUBTE_STÄDTE.Clear()
        ERLAUBTE_STÄDTE.AddRange(New List(Of String) From {"München", "Paris", "London", "Seattle"})
    End Sub

End Class
```

Teil B

Objektorientiertes Programmieren

In diesem Teil:

Einführung in die objektorientierte Programmierung 315
Klassentreffen 321
Klassencode entwickeln 335
Vererben von Klassen und Polymorphie 373
Entwickeln von Wertetypen 441
Typenumwandlung (Type Casting) und Boxing von Wertetypen 455
Beerdigen von Objekten – Dispose, Finalize und der Garbage Collector 471
Eigene Operatoren für benutzerdefinierte Typen 497
Ereignis, Delegaten und Lambda-Ausdrücke 513
Enums 553
Generics (Generika) und generische Auflistungen 561
Typen für Fortgeschrittene 579

Kapitel 8

Einführung in die objektorientierte Programmierung

In diesem Kapitel:

Einführung in Klassen und Objekte 318

Ohne Klassen und Objekte läuft in .NET gar nichts mehr – und das gilt für Visual Basic nicht weniger als für alle die anderen .NET-Programmiersprachen. Dass Visual Basic lange Zeit als nicht objektorientierte Programmiersprache galt, stimmte selbst zu Visual Basic 6.0-Zeiten nur teilweise – denn selbst diese Version von Visual Basic nutzte schon beispielsweise Schnittstellen (Interfaces) zur COM-Entwicklung recht intensiv, auch wenn es regelrechte Vererbung nicht und Polymorphie nur durch späte Bindung emuliert gab.

Objektorientierte Programmierung (OOP) ist aber wichtig für viele, viele Aspekte, und die wichtigsten dabei sind:

- **Sie spart Entwicklungskosten** Nur mit angewandter OOP-Entwicklung lassen sich *sinnvoll* wiederverwendbare Komponenten entwickeln. Softwareentwicklungsabteilungen oder Independent Software Vendors (ISV) investieren damit in die Zukunft. OOP-Komponenten stellen ihr wirkliches Kapital dar, da mit jedem erfolgreich fertig gestellten Projekt Komponenten entstehen, die die Entwicklungskosten des nächsten Projekts deutlich reduzieren.
- **Sie reglementiert Entwicklungsteams bei der Zusammenarbeit**, und zwar nicht auf bevormundende, diktierende, sondern auf helfende und unterstützende Art und Weise. Teams können Code für andere Teams vorbereiten und durch Objektmodelle so gestalten, dass Code von Team A nahtlos mit Code von Team B zusammenlaufen wird, ohne dass beide Teams während der Entwicklung ständig zusammenarbeiten müssen. Teams können sogar Klassen entwickeln, die imstande sind, Code zu verarbeiten, der erst Jahre später entwickelt wird – und garantieren, dass dieser noch nicht existierende Code reibungslos seinen Dienst verrichten wird.
- **Sie sorgt für robusten Code** Objektorientierte Programmierung in .NET macht Ihre Anwendung verlässlich, absturzsicher und robust, solange man sich an bestimmte Regeln hält: Späte Bindung – also das Zulassen des Kompilierens von Eigenschaften oder Methoden für Objektvariablen, die Nachkömmlinge ihrer Klassentypen enthalten – sollte man in Visual Basic nur in Ausnahmefällen erlauben. Auf diese Weise verhindert das Typsystem von .NET, dass – etwa wie in C++ möglich – zur Laufzeit der Versuch fehlschlägt, eine Variable vom Typ »JPEG-Bild« als Zeichenkette zu verwenden, wodurch eine Anwendung instabil werden könnte oder sich durch Pufferüberläufe vielleicht sogar dazu missbrauchen ließe, Schadcode auszuführen.

Diese Punkte werden neben anderen umso wichtiger, als dass LINQ nicht nur beim Abfragen von Datenbanken eine immer wichtigere Rolle spielt, sondern auch in der Parallelisierung von Prozessen. Denken Sie daran, dass Prozessoren sich seit ca. 2002 kaum mehr über die Taktfrequenz leistungsmäßig skalieren lassen. Bis dahin galt: Wird ein schnelleres System benötigt, bedient man sich einfach eines Prozessors mit höherer Taktfrequenz,[1] denn überwiegend darauf waren die jeweils leistungsfähigeren Versionen der nächsten Generationen beschränkt. Das funktioniert nicht mehr ganz so, denn die Taktfrequenz eines Prozessors lässt sich aufgrund des Verlustleistungsproblems ab einer bestimmten Grenze nur noch in Maßen erhöhen. Also skalierte man nicht mehr auf der vertikalen Geschwindigkeitsachse, sondern baute die Prozessoren in die Breite, was bedeutet: Ein Prozessor bekommt mehrere so genannte Kerne (*cores*), und ein moderner Prozessor besteht heutzutage mindestens aus zwei Kernen, bei Desktopsystemen sind inzwischen vier üblich. Doch was nutzt das? Wenn Software so konzepti-

[1] Natürlich immer gemessen an der Architektur des vorherigen Prozessormodells, um nicht Äpfel mit Birnen zu vergleichen. Schon in den 1980er-Jahren war ein 1 MHz getakteter 6502-Prozessor von MOS Technology in etwa so schnell wie ein Z80-Prozessor mit 4 MHz von Zilog. Ein einziger Kern eines heutzutage verbauten Intel Dual Core-Prozessors T9300 mit 2,5 GHz ist sicherlich deutlich schneller als ein 3 GHz schneller Pentium 4 aus dem Jahre 2003.

oniert ist, dass sie nicht mehrere Dinge zur gleichen Zeit ausführt – für die parallele Ausführung müssen Sie als Entwickler nämlich jedenfalls zurzeit noch explizit sorgen –, läuft Ihre Anwendung selbst bei Volllast aus Anwendungssicht auf einem Dual Core-Prozessor nur mit 50%, auf einem Quad Core-Prozessor gar nur mit 25% der möglichen Leistung. Und daran ändert keine Einstellung oder auch kein Betriebssystem dieser Welt etwas. LINQ kann nun nicht nur beim Abfragen von Daten, sondern auch überhaupt beim iterierten Berechnen von Daten der Schlüssel zu einer halbwegs automatischen Parallelisierung von Algorithmen sein. Es untersucht dazu die Abfrage bzw. Iteration und erstellt einen Ausführungsplan, der dann automatisch auf mehrere, zur Verfügung gestellte Prozessorkerne verteilt wird. In der vorliegenden .NET 4.0-Version hat diese Technik mit P/LINQ (*parallel LINQ*) bereits Einzug gehalten, und es gibt auch weitere Klassen in .NET Framework 4.0 (`Task`-Klasse, `Parallel`-Klasse), die Funktionalitäten für die automatische und teilautomatische Parallelisierung von Code zur Verfügung stellen. Um all diese Features zu nutzen, gilt: Sie benötigen in jedem Fall umfassende Kenntnisse von Klassen, Instanzen, Auflistungen, Generics und Delegaten. Die reine prozedurale Programmierung ist hier eine komplette Sackgasse, sodass fundiertes Wissen über die OOP der Schlüssel zum Entwicklerglück wird.

Und vielleicht ein Tipp an dieser Stelle: Lesen oder »browsen« Sie wenigstens doch auch dann durch die folgenden Kapitel, wenn Sie meinen, Klassenprogrammierung grundsätzlich zu beherrschen und in Ihren eigenen Projekten schon längst verwenden. Riskieren Sie es, sich die folgenden Kapitel zu Gemüte zu führen, um Altes aufzufrischen und auch, um den neuen Technologien von Visual Basic 2010 im richtigen Kontext immer mal wieder hier und da zu begegnen. Denken Sie daran: Zum Thema Klassen und OOP gehört weit mehr als nur das Wissen um ihre bloße Instanziierung und das Erstellen von Eigenschaften und Methoden, denn: Wie sieht es mit Klassenvererbung, abstrakten Klassen und Schnittstellen aus? Kennen Sie die Unterschiede zwischen Verweis- und Wertetypen? Können Sie erklären, wieso ein Objekt ein Referenztyp ist, alle primitiven Typen zwar rein .NET-infrastrukturgemäß von `Object` abgeleitet sind, diese aber dennoch zu Wertetypen werden? Wie steht es mit dem *Boxing* von Wertetypen – wissen Sie genau, was dabei wirklich passiert und welche Tücken Ihnen hier und da begegnen? Kennen Sie auch alle Feinheiten der Polymorphie und wissen Sie, wann Sie die Referenzierungen `Me`, `MyClass` und `MyBase` einsetzen müssen? Wie schaut es aus mit Generics, den an C++-Templates angelehnten Objekttypen von .NET, mit denen Sie – die richtige Anwendung vorausgesetzt – viele Dinge des täglichen Entwicklerlebens enorm vereinfachen und extrem schnell wiederverwendbare Klassen schaffen können?

Sie sehen: Es gibt eine ganze Menge, was Sie über objektorientierte Programmierung und den Einsatz von Klassen wissen können (und sollten). Je trittfester Sie beim Einsatz von Klassen werden, desto robuster, pflegeleichter und weniger fehleranfällig werden später Ihre Programme sein.

TIPP Die Kapitel und einzelnen Abschnitte dieses Teils, so umfangreich sie auch sind, bauen schrittweise aufeinander auf. Deswegen sei Ihnen empfohlen, dass Sie sich ein wenig Zeit nehmen und die folgenden Abschnitte am besten hintereinander durcharbeiten. Das Verstehen des ganzen Zusammenhangs von Klassen, Vererbung, Schnittstellen und allem, was sonst noch dazu gehört, wird Ihnen dann sicherlich viel leichter fallen – und Sie werden sich im Handumdrehen zum »Klassenprimus« mausern!

Einführung in Klassen und Objekte

Klassen sind der Hauptstützpfeiler der objektorientierten Programmierung. Spricht man von Klassen und Klassenkonzepten, dann sind Technologien wie Schnittstellen, Klassenmodelle, das Ableiten bzw. Vererben von Klassen sowie Polymorphie mit dem gezielten Überschreiben von Klassenmethoden und –eigenschaften nicht weit. Doch darum soll es vordergründig in diesem Kapitel gar nicht gehen.

Klassen sind in erster Linie mal Schablonen, die ihre Daten reglementieren und von der Außenwelt schützen – und auch nur unter diesem Gesichtspunkt sollen sie in diesem einführenden Kapitel erklärt werden.

»Eine Klasse schafft die Strukturen für das Speichern von Daten und beinhaltet gleichzeitig Programmcode, der diese Daten reglementiert.« Diese Erklärung finden Sie nicht nur oftmals als *die* Erklärung für das Konzept von Klassen, sie ist auch kurz, knackig, absolut zutreffend und so abstrakt, dass jemand, der sich zum ersten Mal mit dieser Materie beschäftigt, damit überhaupt nichts anfangen kann.

Deswegen wollen wir im Folgenden das Pferd bewusst von der anderen Seite aufzäumen und ein Beispiel bemühen, das zeigt, wie ein Programm Daten speichert und organisiert, das *ohne* Klassen zurechtkommen muss – und in welche Fallen man als Entwickler dabei geraten kann.

Miniadresso – die prozedurale Variante

Also lassen Sie uns mal einen Blick auf die »So besser nicht«-Variante werfen – hochtrabend *Miniadresso-Prozedur* genannt.

HINWEIS Bei den Beispielprogrammen der folgenden Abschnitte handelt es sich nicht um Windows-, sondern um so genannte Konsolenanwendungen – um Anwendungen also, die sich ausschließlich unter der Windows-Eingabeaufforderung verwenden lassen und die Sie schon in Kapitel 2) dieses Buchs kennen gelernt haben.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 08\MiniAdressoProzedu
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Wenn Sie das Programm starten, sehen Sie ein Konsolenfenster, das die Programmausgabe beinhaltet, wie Sie sie in etwa auch in Abbildung 8.1 betrachten können.

TIPP Bei den vergleichsweise großen Auflösungen der 22- und 24 Zoll-Monitore, die heutzutage schon für kleines Geld zu haben sind, stellen Sie die Schriftart des Konsolenfensters am besten auf die größte Schriftart (10 × 18). Die Einstellungsmöglichkeit finden Sie dabei über das Systemmenü unter dem Menüeintrag *Standardwerte* bzw. *Eigenschaften*. Erstere gilt für alle zukünftigen Instanzen und Letztere nur für die aktuelle Instanz des Konsolenfensters.

```
file:///C:/Data/FolderShare/SharedCurrent/Books/VB 2008/German/Samples/C - OOP/Kapitel 12/MiniAdressoProzedu/bin/Debug/MiniAdr...
003: Langenbach, Christian, 41729, Lippstadt
004: Löffelmann, Michaela, 69493, Dortmund
005: Meier, Melanie, 47824, Bad Waldliesborn
006: Albrecht, Michaela, 91508, Bad Waldliesborn
007: Jungemann, Hans, 61733, Hildesheim
008: Tiemann, Klaus, 99072, Wuppertal
009: Weichelt, Christian, 95897, Wiesbaden
010: Neumann, Uta, 98014, Lippstadt

Adressen werden sortiert ... fertig!

000: Ademmer, Margarete, 93068, Dortmund
001: Ademmer, Axel, 83968, Berlin
002: Ademmer, Momo, 71591, Stirpe
003: Ademmer, Theo, 71591, Dortmund
004: Ademmer, Uta, 71591, Braunschweig
005: Albrecht, Michaela, 71591, Stirpe
006: Albrecht, Rainer, 83968, Soest
007: Albrecht, Franz, 71591, Soest
008: Albrecht, Gabriele, 71591, Bielefeld
009: Albrecht, Anne, 71591, Stirpe
010: Braun, Theo, 71591, Soest

Taste zum Beenden drücken...
```

Abbildung 8.1 Die erste Version des Programms macht scheinbar Dienst nach Vorschrift – doch durch das prozedurale Konzept hat sich leider ein schwerer Fehler eingeschlichen, der nicht gleich ersichtlich ist

O.k. – hier haben wir also die erste Version unseres kleines Beispielprogramms, das nichts weiter macht, als sich ein paar Kontaktadressen auszudenken, deren Einzeldaten in dafür vorgesehene Arrays abzulegen, die Kontakte nach Nachnamen zu sortieren und sie anschließend auszugeben. Schauen wir uns an, wie das Programm generell mit seinen Daten umgeht:

```
Module Hauptmodul

    Dim Nachname(0 To 100) As String
    Dim Vorname(0 To 100) As String
    Dim PLZ(0 To 100) As String
    Dim Ort(0 To 100) As String

    Sub Main()
```

Es benötigt für jede Kontaktadresse jeweils vier Einzelelemente, die es in insgesamt vier Arrays ablegt, und genau das macht es so schwierig für den Entwickler, die Daten einer einzigen Kontaktadresse auch wirklich zusammenzuhalten. Ohne eine *zusammenhängende* Entität zu schaffen, die die Daten eines Kontakts auch programmtechnisch mehr oder weniger automatisch bzw. hervorgerufen durch ihr Konzept zusammenhält, sind Fehler buchstäblich vorprogrammiert – und genau ein solcher Fehler hat sich dann auch in dieser Programmversion eingeschlichen (O.k., natürlich habe ich den Bug zu Demozwecken bewusst eingebaut, aber dieses prozedurale Programmierbeispiel ist typisch für eine interne Datenverwaltung; genauso typisch eben wie die Fehler, die dabei auftreten können).

```
    Sub AdressenSortieren()

        'Beispiel für lokalen Typrückschluss
        Dim anzahlElemente = 101
        Dim delta = 1

        Dim aeussererZaehler As Integer
        Dim innererZaehler As Integer
```

Bei einem Blick auf die Sortierroutine fällt zunächst etwas Besonderes auf, das es seit Visual Basic 2008 gibt und unter dem Namen *lokaler Typrückschluss* bezeichnet wird. Dabei werden die Typen von Variablen nur durch reine Zuweisung von Konstanten festgelegt. Kapitel 2 erklärte bereits, wie der lokale Typrückschluss genau funktioniert.

Doch zurück zum eigentlichen Ausgangsproblem: Die Sortierroutine des Programms ist in diesem Beispiel der Übeltäter: Gerade sie müsste durch sorgfältige Ausführung bei der Programmierung dafür sorgen, dass ein Adressdatensatz logisch zusammengehörig bleibt, damit die Integrität des gesamten Datenaufkommens nicht verletzt wird. Schauen wir uns diese Sortierroutine jedoch genauer an …

```
        Dim tempVorname, tempNachname, tempPLZ, tempOrt As String

        'Größten Wert der Distanzfolge ermitteln
        Do
            delta = 3 * delta + 1
        Loop Until delta > anzahlElemente
        '
        Do
            'Späteres Abbruchkriterium - entsprechend kleiner werden lassen
            delta \= 3

            'Shellsort's Kernalgorithmus
            For aeussererZaehler = delta To anzahlElemente – 1
                tempVorname = Vorname(aeussererZaehler)
                tempNachname = Nachname(aeussererZaehler)
                tempPLZ = PLZ(aeussererZaehler)
                tempOrt = Ort(aeussererZaehler)

                innererZaehler = aeussererZaehler
                Do
                    If tempNachname >= Nachname(innererZaehler - delta) Then Exit Do
                    Vorname(innererZaehler) = Vorname(innererZaehler - delta)
                    Nachname(innererZaehler) = Nachname(innererZaehler - delta)
                    PLZ(innererZaehler) = PLZ(innererZaehler - delta)

                    innererZaehler = innererZaehler – delta
                    If (innererZaehler <= delta) Then Exit Do
                Loop
                Vorname(innererZaehler) = tempVorname
                Nachname(innererZaehler) = tempNachname
                Ort(innererZaehler) = tempOrt
            Next
        Loop Until delta = 0
    End Sub
```

… so stellen wir bei näherem Hinsehen fest, dass sich, was die Datenintegrität anbelangt, ein dicker Fehler in den Algorithmus eingeschlichen hat: Beim Dreieckstausch eines Kontaktelements nämlich werden nicht wirklich alle Array-Elemente getauscht. Beim »Hintauschen« bleibt der Ort, beim »Zurücktauschen« die Postleitzahl auf der Strecke; für ein kommerzielles Programm wäre das natürlich eine Katastrophe, zumal selbst bei einer Liste von nur 100 Kontaktadressen dieser Fehler nicht sofort ins Auge sticht.

Solche Fehler sind typisch für die prozedurale Programmierung, die auf Techniken verzichten musste, welche es Entwicklern erleichtert, ihre Daten zusammenzuhalten. Es besser zu machen und solche Fehler in Zukunft zu vermeiden, ist jetzt angesagt – und die Organisation von Daten und Programmcode in Klassen ist hier der Schlüssel zum Glück. Die nächsten Kapitel verraten dazu mehr.

Kapitel 9

Klassentreffen

In diesem Kapitel:

Was ist eine Klasse? 322
Klassen mit New instanziieren 324
New oder nicht New – wieso es sich bei Objekten um Verweistypen handelt 325
Nothing 329
Klassen verwenden 331
Wertetypen 332

Was ist eine Klasse?

Versetzen Sie sich in Ihre Kindheit zurück. Und zwar so weit, dass Sie sich vorstellen können, im Sandkasten zu sitzen und mit Förmchen und nassem Sand zu spielen. Oder, falls es in Ihrer Vorstellung keinen Sandkasten gibt, mit Förmchen und Knete. Sie werden es nicht glauben, aber genau zu diesem Zeitpunkt haben Sie bereits das Klassenkonzept das erste Mal angewendet. Ein Förmchen ist im Grunde genommen nämlich nichts anderes als eine Klasse. Das Förmchen gibt vor, wie Objekte ausschauen sollen, die aus ihm entstehen werden, aber selbst ist es noch kein Objekt, sondern nur eine Vorlage. Wenn Sie aus einer Klasse (einem Förmchen) einen Sand- oder Knetekuchen (ein Objekt) machen wollen, dann müssen Sie ein Objekt dieser Klasse instanziieren. Um bei der Analogie zu bleiben: Sie instanziieren einen Kuchen aus einem Förmchen, indem Sie nassen Sand oder Knete in das Förmchen hineingeben, das ganze Ding umdrehen und das Förmchen abziehen.

Abbildung 9.1 Sogar Kinder spielen objektorientiert und bilden Instanzen. Indem sie nämlich Knete in ein Förmchen geben...

Klassen in .NET Framework sind natürlich etwas abstrakter, aber Sie machen sich daran auch nicht die Hände schmutzig. Sie bestehen erst einmal aus einer Reihe von sogenannten Feldvariablen – auch Felder oder Membervariablen genannt – und der Klassenrumpfdefinition, die diese Variablen einschließt und der Klasse ihren Namen gibt.

Damit existiert dann bereits die einfachste Version einer Klasse, die, wie wir gleich sehen werden, ihre Daten logisch kapselt. Immerhin erlaubt sie uns schon, das Problem unseres ersten Beispielprogramms im Ansatz zu ersticken – die zweite Version dieses Demoprogramms stellt das unter Beweis. Hier gibt es nun eine Klasse namens `Kontakt`, die aus vier Feldern besteht – die vier Datenfelder, die eine einzige Adresse ausmachen. In der Folge benötigen wir nun nicht mehr vier eigentlich völlig zusammenhanglose Arrays, sondern nur noch ein einziges Array[1], das aus Elementen besteht, die jeweils eine komplette Adresse speichern.

[1] Mehr zu Arrays und Auflistungen erfahren Sie in Kapitel 20.

Abbildung 9.2 ... und dann die Methode »Zusammenpressen« anwenden, ...

Um das zu erreichen, fügen wir dem bestehenden Programm einfach eine neue Codedatei hinzu – eine Klassendatei – namens *Kontakt.vb*. In diese Klassencodedatei schreiben wir dann folgende Zeilen:

```
Public Class Kontakt
    Public Nachname As String
    Public Vorname As String
    Public PLZ As String
    Public Ort As String
End Class
```

Diese Klasse namens `Kontakt` stellt die einfachste Form einer Klasse dar. Sie enthält nur vier öffentlich zugängliche Felder, aber die Klasse fasst eben diese vier Felder logisch zu einem Datensatz zusammen.

Abbildung 9.3 ... erhalten sie ganz viele kleine Knetaffen, oder in Programmierer-deutsch ausgedrückt: Aus der Klasse »Affe« bilden sie viele Instanzen, die dann alle unterschiedliche Eigenschaften haben können. In diesem Beispiel können sich die Instanzen durch ihre Farbeigenschaft (engl: *color property*) unterscheiden, obwohl sie alle von derselben Klasse (nämlich dem Förmchen) abstammen

Klassen mit New instanziieren

Um eine so genannte Instanz dieser Klasse zu schaffen, bedient man sich des Schlüsselwortes `New` – intern wird damit der entsprechende Speicherplatz geschaffen, den man benötigt, um die Daten der Feldvariablen der Klasse im Arbeitsspeicher[2] abzulegen. Eine Objektvariable, die man zuvor als Typ dieser Klasse definiert, dient dann dazu, auf die Elemente der Klasseninstanz zuzugreifen. Die Anwendung dieser einfachen Beispielklasse würde also wie folgt aussehen:

```
Dim adr As Kontakt
adr = New Kontakt
```

Die erste Codezeile legt eine Objektvariable vom Typ `Kontakt` an und die zweite Codezeile weist ihr eine neue Instanz zu. Sie können diese beiden Zeilen auch in einer Zeile zusammenfassen:

```
Dim adr As New Kontakt
```

Anschließend verwenden Sie die Instanz dieser Klasse mithilfe der definierten Objektvariablen, um auf die einzelnen Felder zuzugreifen:

```
adr.Nachname = "Dröge"
adr.Vorname = "Ute"
adr.PLZ = "59555"
adr.Ort = "Lippstadt"
```

Um nochmals auf unsere Analogie mit der Knete zurückzukommen: Die Objektinstanz entspricht hier einem Knetkuchen, der aus einem Förmchen hervorgegangen ist. Die Klasse hingegen ist das Förmchen. Die Objektvariable benennt wiederum den Knetekuchen, der aus dem Förmchen hervorgegangen ist, und natürlich können Sie, wie aus dem Förmchen, beliebig viele Sandkucheninstanzen erstellen. Beliebig viele? Nicht ganz: Natürlich nur so viele, bis der »verwaltete Sandberg« zuneige geht, aus dem Sie die Sandkuchen machen, der in der Analogie dem Managed Heap entspricht, aus dem Sie den Speicherplatz für Ihre Objektinstanzen beziehen.

Ein gern gemachter Fehler am Anfang ist es übrigens, die Klasse nicht zu instanziieren, sondern direkt verwenden zu wollen:

```
'So geht's natürlich nicht!
Kontakt.Nachname = "Dröge"
Kontakt.Vorname = "Ute"
Kontakt.PLZ = "59555"
Kontakt.Ort = "Lippstadt"
```

Warum? – Denken Sie mal drüber nach. In Analogie zu einer primitiven Variablen würden Sie praktisch statt

```
Dim i as Integer
i = 5
```

[2] Genau genommen in einem bestimmten Teil des Arbeitsspeichers, der durch das .NET Framework verwaltet, das heißt ständig aufgeräumt wird. Sein Name: *Managed Heap*.

einfach

```
Integer = 5
```

schreiben, quasi den Bezeichner für den Datentyp als Variablennamen verwenden. Das geht natürlich nicht! Später werden wir sehen, dass es tatsächlich Eigenschaften und Methoden gibt, die Sie direkt über den Typennamen aufrufen können wie etwa bei `if File.Exists("C:\windows\calc.exe") then` – aber dabei handelt es sich um so genannte statische Methoden, die wir später als Ausnahme von der Regel betrachten werden.

Öffentliche Felder oder Eigenschaften beim Instanziieren initialisieren

Visual Basic erlaubt seit der Version 2008 auch eine kürzere Variante beim Vorbelegen der Felder bzw. Eigenschaften mit Werten: Sie können öffentliche Felder oder Eigenschaften direkt beim Instanziieren mithilfe des `With`-Schlüsselwortes definieren, wie im folgenden Beispiel zu sehen:

```
Dim adr As New Kontakt With {.Nachname = "Dröge", _
                        .Vorname = "Ute", .PLZ = "59555", .Ort = "Lippstadt"}
```

New oder nicht New – wieso es sich bei Objekten um Verweistypen handelt

Zum besseren Verständnis für alle späteren Kapitel ist es überaus sinnvoll, ein paar Worte zur Speicherung von Objekten, also Instanzen von Klassen zu verlieren. Objektvariablen und Objekte sind nämlich nicht *so* miteinander verbunden, wie man es sich zunächst vielleicht vorstellt. Im Gegenteil: Der Verbund einer Objektvariablen mit den eigentlichen Daten des Objekts hält bestenfalls so gut wie eine Prominentenehe: Was auf den ersten Blick so innig und für immer geschaffen zu sein scheint, ist in der nächsten Sekunde auch schon wieder sauber getrennter Schnee von gestern.

Abbildung 9.4 Durch das Instanziieren wird ein Teil vom Hauptspeicher weggenommen und beim Instanziieren den neuen Objekten zur Verfügung gestellt, die dann auf dem Managed Heap (etwa: »verwalteter Haufen«) unter Beobachtung gestellt werden

Die Wahrheit ist nämlich: Eine Objektvariable speichert im Grunde genommen nur einen *Speicheradressenzeiger* (oder einfach nur kurz *Zeiger* genannt) auf die eigentlichen Daten in einem besonders überwachten Teil des Hauptspeichers, dem so genannten Managed Heap. In eben diesen gelangen die Daten der Objekte, die Sie mithilfe von `New` aus Klassen instanziieren.

Abbildung 9.5 Objektvariablen sind im Grunde nichts anderes als Klebehefter, die ein Objekt (also eine Instanz eines Objekts) bezeichnen. Und natürlich können Sie an ein Äffchen zwei Klebehefter zum Bezeichnen hängen, oder im Programmiererdeutsch ausgedrückt: Mehrere Objektvariablen können auf dieselbe Instanz zeigen

Und wie wir (die Älteren unter uns jedenfalls) aus unseren Anfängertagen mit dem Commodore 64, Atari ST, Amiga und Maschinensprache noch alle wissen, untergliedert sich der Arbeitsspeicher eines Computers in bestimmte Speicherbereiche, die alle bestimmte »Hausnummern« (die Speicheradressen) besitzen.

Wenn Sie nun ein Objekt aus einer Klasse erstellen – zum Beispiel indem Sie einen `Kontakt`-Datentyp[3] mit `New` in die Objektvariable `objVarKontakt` instanziieren –, dann legt .NET Framework die Daten für diese Objektinstanz beispielsweise an Speicheradresse 460.386 auf dem Managed Heap ab, und die Objektvariable wird intern sozusagen zu einer `Integer`-Variablen (oder auf 64-Bit-Systemen zu einer `Long`-Variablen), die diese Adresse trägt. Bildlich sieht das wie in Abbildung 9.6 gezeigt aus.

[3] Für diese Erklärung wurde übrigens auch der Datentypname `Kontakt` dem Namen `Adresse` vorgezogen, um nicht zu viel Konfusion mit dem Wort Speicher*adresse* entstehen zu lassen.

Abbildung 9.6 Objektvariablen speichern im Grunde genommen nur die Speicheradressen auf die eigentlichen Daten, die .NET Framework im Managed Heap ablegt

Diese Tatsache hat aber entscheidende Folgen, wie das folgende Beispiel gleich zeigen wird.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 09\KlassenObjekteSpeicher
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
Module Module1

    Sub Main()

        'Instanziieren mit New und dadurch
        'Speicher für das Kontakt-Objekt
        'auf dem Managed Heap anlegen.
        'Instanz gleichzeitig mit Daten initialisieren.
        Dim objVarKontakt As New Kontakt With {.Nachname = "Pfeiffer", _
                .Vorname = "Ute", .PLZ = "59555", .Ort = "Lippstadt"}

        'Nur Objektvariable anlegen,
        'es wird aber kein Speicher reserviert!
        Dim objVarKontakt2 As Kontakt

        'objVarKontakt2 "zeigt" ab jetzt auf
        'dieselbe Instanz wie objVarKontakt
        objVarKontakt2 = objVarKontakt

        'Und das kann man auch beweisen:
        'Das Ändern der Instanz geschieht...
        objVarKontakt2.Nachname = "Dröge"

        'durch beide Objektvariablen, die natürlich
        'auch dasselbe widerspiegeln.
        Console.WriteLine(objVarKontakt.Nachname)
```

```
        Console.WriteLine()
        Console.WriteLine("Zum Beenden Taste drücken")
        Console.ReadKey()
    End Sub

End Module

Public Class Kontakt
    Public Vorname As String
    Public Nachname As String
    Public PLZ As String
    Public Ort As String
End Class
```

Wenn Sie dieses Beispiel laufen lassen, erhalten Sie die Ausgabe

```
Dröge

Zum Beenden Taste drücken
```

Abbildung 9.7 Das Kopieren einer Objektvariablen in eine andere Objektvariable kopiert nur den Zeiger auf die Instanz – die dann durch beide Variablen manipulierbar und abrufbar wird

Beachten Sie, dass es in diesem Beispiel zwar nur eine einzige Dateninstanz der Klasse `Kontakt` gibt, die jedoch durch zwei Objektvariablen angesprochen und damit auch widergespiegelt wird: Beim Instanziieren wird die Adresse des Speichers, an der die Daten der Instanz abgelegt werden, in der Objektvariablen `objVarKontakt` gespeichert. Diese Adresse wird in die Variable `objVarKontakt2` übertragen und, wichtig, hierbei wird nicht etwa eine Kopie der gesamten Instanz im Managed Heap angelegt!

Eine Objektvariable »zeigt« also quasi auf die Daten, die sie verwaltet. In anderen Programmiersprachen wie C++ gibt es zu diesem Zweck besondere Variablen, die wie schon erwähnt, auch als (*Speicheradressen-*)*Zeiger* bezeichnet werden. In Visual Basic wird eine Objektvariable, die die Instanz einer Klasse verwaltet, automatisch zu einem solchen *Zeiger*. Diesen Zusammenhang zeigen die Abbildungen 9.6 und 9.7 schematisch.

Dieses Verhältnis zwischen Objektvariable und eigentlichem Objekt (eigentlicher Instanz) sollten Sie sich gut einprägen, da dieses einerseits die Quelle schwer zu findender Fehler ist – schließlich kann es auch versehentlich passieren, dass wenn Sie nicht aufpassen, eine Objektvariable die Instanz eines Objekts verändert, die eigentlich und ausschließlich durch eine ganz andere Objektvariable angesprochen werden sollte. Andererseits kann Ihnen dieses Verhältnis auch zum Vorteil gereichen, nämlich wenn es darum geht, nicht Objekte zu kopieren, sondern nur die Zeiger auf diese – beispielsweise wenn es beim Sortieren großer Objektmengen auf Geschwindigkeit ankommt und deswegen keine ganzen Speicherblöcke mit den eigentlichen Daten, sondern nur die Zeiger auf die Objektinstanzen kopiert werden sollen.

Nothing

Nothing heißt *nichts*. Und wenn man über nichts schreiben soll, könnte man meinen, dass man vergleichsweise schnell damit fertig ist. Aber offensichtlich ist *nichts* in Visual Basic bzw. .NET doch mehr als nichts: Und wie! Man könnte fast schon sagen, nichts hat in Visual Basic mehr Bedeutungen als Nothing!

Nothing als Null-Referenz-Zeiger

Mit dem Wissen des vorherigen Abschnitts können Sie sich auch erklären, wieso eine Objektvariable den Wert `Nothing` (`null` in C# – nur der Vollständigkeit halber erwähnt) aufweisen kann. Wenn Sie mit `New` eine Instanz einer Klasse erstellen, weisen Sie diese Instanz (genauer: die Adresse dieser Instanz) der Objektvariablen zu. Erst dann können Sie mithilfe der Objektvariablen auf die öffentlichen Felder, Eigenschaften und Methoden der Klasseninstanz (des Objekts) zugreifen.

Abbildung 9.8 Wenn Sie mit einer Objektvariablen, der keine Klasseninstanz zugeordnet ist, auf Klassenelemente (Felder, Eigenschaften, Methoden) zugreifen, wird eine `NullReferenceException` ausgelöst

Allerdings: Sie können natürlich auch eine Objektvariable definieren, der eben noch nichts zugewiesen wurde. Wenn Sie dann versuchen, mit einer solchen nicht initialisierten Objektvariablen auf Klassenmember zuzugreifen, wird zur Laufzeit eine entsprechende Ausnahme ausgelöst – die obige Abbildung zeigt dies eindrucksvoll. In vielen Fällen kann der Hintergrundcompiler von Visual Basic diesen Zustand voraussehen und zeigt eine entsprechende Warnmeldung an. Ihre Objektvariable zeigt dann nämlich auf Nichts (auf jeden Fall auf kein gültiges Objekt), also – jetzt auf Englisch – auf *Nothing*.

Wenn Sie überprüfen wollen, ob eine Variable auf ein gültiges Objekt oder auf `Nothing` verweist, müssen Sie übrigens (anders als in C# beispielsweise) in Visual Basic leider einen besonderen Vergleichsoperator benutzen. Die Abfrage `If locObjEinObjekt = Nothing Then…` funktioniert nicht, die richtige Syntax lautet: `If locObjEinObjekt Is Nothing Then…`

WICHTIG Dass `Nothing` nur mit `Is` und nicht mit = funktioniert, ist der Tatsache geschuldet, dass = in Visual Basic *Wertevergleiche* und `Is` eben *Referenz-* (also Zeiger-)Vergleiche durchführt. Das Gleichheitszeichen sorgt dafür, dass die Klasse eine bestimmte Methode aufruft, die dann den Inhalt und eben nicht die Speicheradresse mit einer anderen Variablen vergleicht (bzw., um Pedanten vorzubeugen, mit dem Speicherinhalt einer anderen Variablen).

Falls Sie, natürlich immer bei eingeschaltetem `Option Strict`, es dennoch versuchen, generiert der Compiler einen Fehler: »`Option Strict On` lässt Operanden des Typs `Object` für den =-Operator nicht zu. Verwenden Sie den `Is`-Operator, wenn Sie die Objektidentität testen möchten.«

Übrigens können Sie `Nothing` natürlich auch in Zuweisungen an Referenzvariablen verwenden, nämlich immer dann, wenn Sie die Instanz einer Klasse von einer Objektvariablen lösen und diese Instanz für das Entsorgen durch den Garbage Collector freigeben möchten:

```
k2 = Nothing
```

War `k2` in diesem Fall ein Zeiger auf eine gültige Instanz eines Kontaktes, so zeigt die Variable nach dieser Zuweisung auf nichts mehr, auf `Nothing` eben! Die Zuweisung von `Nothing` an eine Variable kann aber auch noch in anderen Szenarien Sinn ergeben, wie der folgende Abschnitt zeigt.

Nothing in Zuweisungen für Standardwerte

In Visual Basic kommt dem Schlüsselwort `Nothing` eine weitere Bedeutung zu, wenn Sie es nämlich im Zusammenhang mit Wertetypen, also beispielsweise primitiven Datentypen wie `Integer`, `Double`, `Date` oder `Decimal`, verwenden. In diesem Fall weisen Sie mit `Nothing` diesen Datentypen nämlich deren so genannten Standardwert zu. Die folgenden Zeilen sollen das verdeutlichen:

```
Dim intValue = 5
Debug.Print(intValue.ToString)

intValue = Nothing
Debug.Print(intValue.ToString)

Dim dateValue = #7/24/1969#
Debug.Print(dateValue.ToString)

dateValue = Nothing
Debug.Print(dateValue.ToString)
```

Die Codezeilen erzeugen im Debug-Fenster die folgenden Ausgaben:

```
5
0
24.07.1969 00:00:00
01.01.0001 00:00:00
```

HINWEIS Der Vollständigkeit halber sei erwähnt: Zu dieser Zuweisung von Standardwerten gibt es in C# übrigens nur bei Generics (siehe dort) ein Äquivalent durch das `default`-Schlüsselwort. Die Zuweisung von `null` (was `Nothing` in Visual Basic entspricht) an normale (nicht generische) Wertetypen in C# ist ebenso wenig gestattet wie die Zuweisung von `default`.

HINWEIS Streng genommen ist `Nothing` nur der Wertelieferant für Standardwerte des entsprechenden Datentyps, und der Standardwert bei Referenzwerten ist ein Null-Zeiger. Didaktisch ist das leider eine Katastrophe, denn `Nothing` sollte dann besser *Default* heißen, deswegen lassen wir die vorherigen Aussagen so stehen, wie sie sind. ;-)

Klassen verwenden

Natürlich handelt es sich bei Klassen zwar um ganz neue Datentypen, die Sie aber genauso handhaben können wie alle anderen Datentypen von .NET Framework. Und damit können einzelne Instanzen einer Klasse ebenso in Arrays abgelegt werden, wie das bei ganz normalen primitiven Datentypen der Fall ist – wir sind damit beim ersten richtigen Anwendungsbeispiel für Klassen, nämlich der zweiten Version unseres Adressenbeispiels.

Mit all den Infos aus den vorherigen Abschnitten können wir das anfängliche Beispielprogramm nun so umschreiben, dass es »nur« noch mit einem Array auskommt, in dem Elemente vom gerade neu geschaffenen Typ `Kontakt` vorhanden sind, und dazu sind nicht einmal viele Änderungen notwendig:

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010Entwicklerbuch\Kapitel 09\MiniAdressoClass
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Die erste betrifft das Einrichten der Datenstruktur am Anfang des Programms:

```
Module Hauptmodul

    Dim Kontakte(0 To 100) As Kontakt
```

Im Unterschied zur ursprünglichen Version können wir uns jetzt mit nur noch einem Array begnügen, das Elemente enthält, die die Felder eines kompletten Kontaktes kapseln. Viel aufgeräumter sieht nun auch der Sortieralgorithmus aus (Änderungen an ihm sind fett hervorgehoben):

```
    Sub AdressenSortieren()

        Dim anzahlElemente As Integer = 101

        Dim aeussererZaehler As Integer
        Dim innererZaehler As Integer
        Dim delta As Integer

        Dim tempKontakt As Kontakt

        delta = 1

        'Größten Wert der Distanzfolge ermitteln
```

```
        Do
            delta = 3 * delta + 1
        Loop Until delta > anzahlElemente

        Do
            'Späteres Abbruchkriterium - wieder kleiner werden lassen
            delta = delta \ 3

            'Shellsort's Kernalgorithmus
            For aeussererZaehler = delta To anzahlElemente - 1
                tempKontakt = Kontakte(aeussererZaehler)

                innererZaehler = aeussererZaehler
                Do
                    If tempKontakt.Nachname >= Kontakte(innererZaehler - delta).Nachname Then Exit Do
                    Kontakte(innererZaehler) = Kontakte(innererZaehler - delta)

                    innererZaehler = innererZaehler - delta
                    If (innererZaehler <= delta) Then Exit Do
                Loop
                Kontakte(innererZaehler) = tempKontakt
            Next
        Loop Until delta = 0
    End Sub
```

Da der neue Datentyp `Kontakt` die Daten einer kompletten Kontaktadresse kapselt, kann der Fehler aus der letzten Programmversion gar nicht mehr passieren. Der Dreieckstausch vollzieht sich jetzt zwangsläufig an einem vollständigen Kontakt. Somit ist die Gefahr ausgeschlossen, dass einzelne Felder auf der Strecke bleiben. Die fett gedruckten Zeilen im vorherigen Listing dokumentieren das Aufräumen.

Wertetypen

Im Gegensatz zu den Referenztypen, die aus Klassen entstehen, gibt es in .NET Framework ein weiteres Grundelement zum Speichern von Daten, und zwar den Wertetyp. Und warum? Ganz einfach: Sie können sich vorstellen, dass es ein vergleichsweise aufwändiger Vorgang ist, schon bei der kleinsten Integer-Variablen Speicherplatz reservieren zu müssen, den Wert, der durch die Integer-Variable repräsentiert werden soll, dann dort hineinzuschreiben, die Referenz auf diesen Speicherplatz in einer Objektvariablen zu hinterlegen und, wann immer man auf diese Variable zugreifen will, sie wieder mit ganz, ganz langen Armen aus dem Speicher zu fischen. So geht's nicht.

Also führte man in .NET Framework die so genannten Wertetypen ein, und diese landen nicht auf dem Managed Heap, also auf dem Objektespeicherhaufen, sondern im so genannten Prozessorstack, dem Speicherbereich, auf den der Prozessor am schnellsten zugreifen kann. Bei einigen primitiven Datentypen erzeugt der CIL-Compiler derart optimierten Code, dass bei der Verwendung einer Integer-Variablen sogar direkt ein Prozessorregister für die Speicherung des Wertes verwendet werden kann. Das ist doppelt performant, denn bei dieser Vorgehensweise fällt auch das Dereferenzieren durch eine Adressvariable unter den Tisch. Der Wert wird *direkt* auf dem Stack abgelegt oder sogar in einem Prozessorregister gehalten, daher auch der Ausdruck Wertetyp.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 09\ReferenzVsWertetypen
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Das hat allerdings auch Auswirkungen auf die Handhabung, denn wenn der Speicher für einen Wertetyp implizit zugewiesen wird, brauchen Sie ihn auch nicht explizit mit `New` anzufordern. Und genau das ist der Grund, weswegen Sie das folgende Konstrukt zwar schreiben können, es aber überhaupt keinen Sinn ergibt:

```
Dim intVar As New System.Int32
intVar = 21
```

Dim intVar As Integer

Abbildung 9.9 Sie können jeden primitiven Typ über sein .NET Framework-Äquivalent deklarieren – im Ergebnis macht das aber keinen Unterschied

Sie sehen in der Abbildung anhand des ToolTips, dass diese Konstruktion auf den ersten Blick auch keinen Unterschied bei der Definition des Variablentyps macht: Obwohl Sie `System.Int32` – das ist der Integertyp aus Sicht von .NET Framework – angegeben haben, sagt der ToolTip, dass `intVar` vom Typ `Integer` ist.

HINWEIS Für diejenigen unter Ihnen, die sich für die Details dieses Beispiel interessieren: Was hier passiert, ist, dass Sie einen Typ, der eigentlich ein Wertetyp ist, als Objekt anlegen – denn Sie haben das Schlüsselwort `New` ohne Parameter angegeben. Doch mit dieser so erstellten Instanz des `Int32`-Objektes passiert anschließend gar nichts. Sie wird auf dem Managed Heap angelegt und verfällt. Der Garbage Collector – die Müllabfuhr in .NET, auf die wir im Detail noch in Kapitel 14 zu sprechen kommen – entsorgt diese Instanz bei nächster Gelegenheit.

In Wahrheit findet das eigentliche Geschehen entweder auf dem Prozessorstack oder, wenn später der optimierte Code zur Ausführung kommt, eben in einem Register des Prozessors statt. Denn `System.Int32` ist nun einmal ein Wertetyp, und, wie der ToolTip richtig informiert, als solcher wird er auch verwendet. Die betroffene Codezeile hat also nur eine einzige Daseinsberechtigung, und die liegt darin, die Variable selbst vom Typ `Int32` zu definieren, was normalerweise durch den Compiler auch mit dem Schlüsselwort `Integer` geschieht.[4]

Merken wir uns das Wichtige hierbei: *Wertetypen müssen nicht instanziiert werden, sie können direkt verwendet werden*. Die CLR sorgt dafür, dass Wertetypen anders als Verweistypen gespeichert werden, und zwar rein aus Performancegründen.

Dieses Verhalten ergibt nicht nur für primitive Datentypen wie `Integer`, `Date` oder `Decimal` Sinn, sondern auch für Datentypen, die Sie selbst entwerfen möchten und bei denen Sie wünschen, dass sie in erster Linie sehr schnell verarbeitet werden.

Wertetypen mit Structure erstellen

Prinzipiell erstellen Sie Wertetypen genau wie Referenztypen. Doch während Referenztypen aus Klassen entstehen, schaffen Sie die Vorlagen für Wertetypen mit so genannten Strukturen. Wenn wir also unsere Beispielklasse `Kontakt` als Wertetyp erstellen möchten – wir nennen ihn im Folgenden zur besseren Unterscheidung `KontaktStructure` –, verwenden wir dazu das Schlüsselwort `Structure` im Code, der dann auf das vorhandene Klassenbeispiel übertragen folgendermaßen auszusehen hat:

[4] Das Konstrukt `New Integer()` hätte der Compiler im Übrigen direkt mit einem Fehler quittiert, da er »weiß«, dass das Konstrukt lediglich *eine Instanz* von `Int32` als *Objekt* anlegen würde, auf die es keine Referenz gäbe: Die einzige Variable hält nämlich die Verbindung zum Speicherplatz für den Integer auf dem Stack.

```
Public Structure KontaktStructure
    Public Vorname As String
    Public Nachname As String
    Public PLZ As String
    Public Ort As String
End Structure
```

Für diese Struktur gilt jetzt das Gleiche wie für primitive Variablen – Sie brauchen sie nicht mehr mit `New` zu instanziieren, sodass der folgende Code funktioniert:

```
        'Ein Kontakt wird als Struktur definiert und kann
        'auch ohne New sofort verwendet werden - der Speicher
        'wird implizit auf dem Stack zur Verfügung gestellt.
        Dim einKontakt As KontaktStructure
        einKontakt.Vorname = "Angela"
        einKontakt.Nachname = "Wördehoff"
        einKontakt.PLZ = "59555"
        einKontakt.Ort = "Lippstadt"
```

Wertetypen durch Zuweisung klonen

Und noch ein anderes Verhalten ist erwähnenswert, in Anlehnung an die primitiven Datentypen aber zu erwarten: Genau wie primitive Datentypen werden die Inhalte eines Wertetyps durch Zuweisung an eine andere Variable gleichen Typs kopiert – man nennt dieses Verhalten auch klonen. Schauen Sie sich den folgenden Code an:

```
        'Und hier zeigt sich das andere Verhalten von
        'Wertetypen zu Referenztypen:
        'Beim Zuweisen von Wertetyp zu Wertetyp werden keine Referenzen,
        'sondern die Inhalte kopiert.
        Dim einAndererKontakt As KontaktStructure
        einAndererKontakt = einKontakt

        'Ein Ändern der einen "Instanz" hat damit
        'keine Auswirkung auf das Ändern der zweiten Instanz.
        einAndererKontakt.Nachname = "Löffelmann"
        Console.WriteLine(einKontakt.Nachname & ", " & _
                          einKontakt.Vorname)
        Console.ReadLine()
```

Dieser Code gibt die Zeile

```
Wördehoff, Angela
```

aus. Anders als bei Referenztypen zeigt die zweite Variable des Beispiels nämlich nicht auf die erste Instanz von `Kontakt`; es gibt nämlich keine Instanz im Objektsinne. Vielmehr liegen die relevanten Daten auf dem Stack, und wenn Sie diese Daten durch eine Zuweisung an eine andere Wertevariable übertragen, dann werden diese Daten dupliziert. Wertetypvariablen können also bestenfalls die gleichen, aber niemals dieselben Daten preisgeben.

HINWEIS Fürs Erste soll diese Beschreibung des Unterschieds zwischen Werte- und Referenztypen ausreichen. Es gibt jedoch bei der Programmierung, gerade wenn Sie Daten in Auflistungen und Arrays speichern, diesbezüglich noch viel mehr Wissenswertes, mit dem sich dann Kapitel 12 und 13 auseinander setzen werden.

Kapitel 10

Klassencode entwickeln

In diesem Kapitel:

Eigenschaften 336

Der Konstruktor einer Klasse – bestimmen, was bei New passiert 347

Klassenmethoden mit Sub und Function 356

Überladen von Methoden, Konstruktoren und Eigenschaften 357

Zugriffsmodifizierer für Klassen, Methoden und Eigenschaften – Gültigkeitsbereiche definieren 365

Statische Elemente 368

Über mehrere Codedateien aufgeteilter Klassencode – das Partial-Schlüsselwort 371

Betrachten wir nochmals die anfängliche Definition von Klassen: »Eine Klasse schafft die Strukturen für das Speichern von Daten und beinhaltet gleichzeitig Programmcode, der diese Daten reglementiert.« Bislang haben wir erst die eine Komponente einer Klasse kennen gelernt: so genannte Feldvariablen, die die Möglichkeit schaffen, bestimmte Dinge in einer Klasseninstanz zu speichern.

Der zweite Teil, der den Zugriff auf die Daten reglementieren soll, fehlt bislang. Das Klassenkonzept sieht es vor, dass datenreglementierender Code in Form von zwei Prozedurtypen in einer Klasse vorhanden sein kann: in Methoden (`Sub`, `Function`) und in Eigenschaftenprozeduren (`Property`).

Darüber hinaus haben Sie die Möglichkeit, auch durch den so genannten Konstruktor zu bestimmen, was beim Instanziieren einer Klasse passieren soll – also zusätzlich zu den Maßnahmen, welche die CLR ohnehin durchführt, wie beispielsweise das Reservieren des Speichers für die eigentlichen Daten Ihrer Klasseninstanz.

Eigenschaften

Die bislang einzige Beispielklasse dieses Kapitels bestand aus nur vier Feldvariablen, auf die man, da sie mit dem Schlüsselwort `public` (also als öffentlich zugänglich) definiert wurden, in jeder Instanz zugreifen konnte. Das hat mit der Kapselung von Daten natürlich denkbar wenig zu tun; dieses Beispiel diente bislang lediglich dazu, verschiedene Daten zu einer logischen Dateneinheit zusammenzufassen – von wirklicher Kapselung der Daten, bei der das Rein und Raus der Daten durch irgendeine Klassenpolizei geregelt worden wäre, konnte man dabei allerdings noch nicht sprechen.

Jetzt stellen Sie sich vor, Sie möchten, dass Ortsnamen unserer `Kontakte`-Klasse gekürzt werden, falls diese mehr als 30 Zeichen aufweisen. Damit verdeutlicht wird, dass die Namen abgeschnitten wurden, sollen drei Auslassungspunkte »...« an den Ortsnamen gehängt werden.

In der prozeduralen Welt würden Sie dazu eine Funktion entwickeln, die überprüft, ob eine Zeichenfolge mehr als 30 Zeichen umfasst. Wäre das der Fall, würden Sie alle Zeichen nach dem 27. Zeichen abschneiden und drei Punkte (.) anhängen. Somit erhalten Sie immer einen Ortsnamen mit höchstens 30 Zeichen.

Sie würden dann nicht direkt auf die Feldvariable zugreifen, sondern, wie im folgenden Beispiel, jeden Zugriff auf das Feld *Ort* in einen Funktionsaufruf einbauen. Das könnte dann im Ergebnis folgendermaßen ausschauen:

Wir haben also zunächst unsere »AuslassungszeichenAnText«-Routine …

```
Function EllipseString(ByVal text As String, ByVal MaxLength As Integer) As String
    Dim tmpText As String

    If text.Length > MaxLength Then
        tmpText = text.Substring(0, MaxLength - 3) + "..."
    Else
        tmpText = text
    End If
    Return tmpText
End Function
```

… und diese Routine rufen Sie immer dann auf, wenn Sie auf die kritisch zu behandelnde Feldvariable `Ort` einer Instanz der Klasse `Kontakt` zugreifen:

```
    Sub KlasseVerarbeiten()
    .
    .
    .
        Dim tmpKontakt As New Kontakt
    .
    .
    .
        tmpKontakt.Ort = EllipseString(Console.ReadLine, 30)
    End Sub
```

Es gibt aber noch eine viel elegantere und vor allen Dingen *sichere* Möglichkeit, dafür zu sorgen, dass die Feldvariable `Ort` der `Kontakt`-Klasse niemals eine zu lange Zeichenkette zugewiesen bekommt: und das sind die so genannten *Eigenschaftenprozeduren*.

Wenn Sie schon längere Zeit mit Visual Basic (egal, ob mit einer .NET-Version oder mit Version 6.0) programmieren, haben Sie Eigenschaften vermutlich längst kennen gelernt. Mithilfe von Eigenschaften können Sie in der Regel bestimmte Zustände von Objekten abfragen *und* verändern, und von außen »fühlt« sich das dann so an, als würden Sie direkt ein öffentliches Feld einer Klasse manipulieren oder abfragen. Möchten Sie beispielsweise wissen, ob die Schaltfläche eines Formulars anwählbar ist, verwenden Sie die Eigenschaft in Abfrageform, etwa wie hier:

```
If Schaltfläche.Enabled Then TuWas
```

Oder Sie legen die Eigenschaft eines Objekts fest, etwa wie mit der folgenden Zeile:

```
Schaltfläche.Enabled = False    ' Abschließen, an den Button kommt keiner mehr ran
```

Sie wissen nun aber aus Erfahrung, dass Sie bei einer solchen Festlegung einer Eigenschaft einer Schaltfläche nicht nur den Zustand einer Variablen dieser `Button`-Instanz verändern, sondern dass das Setzen dieser Eigenschaft auch noch etwas Weiteres bewirkt: Die Schaltfläche wird in diesem Beispiel nämlich auch auf dem Formular deaktiviert, was durch eine graue Darstellung signalisiert wird (oder eben wieder – beim Zuweisen von `True` – aktiviert und normal dargestellt). Der Code, der dafür sorgt, muss innerhalb der Klasse natürlich irgendwo platziert werden, und so lautet die Frage: Wie statten Sie Ihre eigenen Klassen mit Eigenschaften aus?

Visual Basic stellt Ihnen zu diesem Zweck, wie schon erwähnt, Eigenschaftenprozeduren zur Verfügung. Eine Eigenschaft wird in Visual Basic innerhalb einer Klasse folgendermaßen definiert:

```
Property EineEigenschaft() As Datentyp

    Get
        Return DatentypObjektvariable
    End Get

    Set(ByVal Value As Datentyp)
        DatentypObjektvariable = Value
    End Set

End Property
```

Wenn Sie diese Eigenschaft in einer Klasse implementieren, können Sie diese bei instanziierten Objekten dieser Klasse wie im nächsten Abschnitt beschrieben verwenden.

HINWEIS Die Editorunterstützung hat sich bei Eigenschaftenprozeduren mit der Einführung der so genannten *automatisch implementierten Eigenschaften* etwas verändert. Bis Visual Basic 2008 verhielt es sich so, dass es ausreichte, nach dem Schreiben des Eigenschaftenrumpfs ↵ zu drücken, um den `Get`- und `Set`-Rumpfteil (mehr dazu später im Abschnitt über Eigenschaften) der Eigenschaft automatisch einfügen zu lassen. Da nunmehr der Eigenschaftenrumpf schon eine gültige Codezeile ist (nämlich die automatisch implementierte Eigenschaft), ist etwas mehr Aufwand erforderlich:

```
'Eigenschaft, um den Wert verändern zu können.
Property EinWert() As Integer
```

Schreiben Sie in die nächste Zeile einfach das Schlüsselwort `Get` und drücken Sie anschließend ↵:

```
'Eigenschaft um den Wert ändern zu können
Property EinWert As Integer
Get
```

Get

Get-Anweisung
Deklariert eine Get-Eigenschaftenprozedur, mit der der aktuelle Wert einer Eigenschaft zurückgegeben wird.

Der Editor fügt den Rumpf von Setter- und Getter-Prozedur nun selbstständig ein:

```
'Eigenschaft, um den Wert verändern zu können.
Property EinWert() As Integer
    Get

    End Get
    Set(ByVal value As Integer)

    End Set
End Property
```

Zuweisen von Eigenschaften

Mit der Anweisung

```
Object.EineEigenschaft = Irgendetwas
```

weisen Sie der Eigenschaft `EineEigenschaft` des Objekts einen Wert zu. Sie können im *Set*-Accessor[1] (`Set(ByVal value as Datentyp)`) der Eigenschaftenprozedur mit `value` auf das Objekt zugreifen, das sich in `Irgendetwas` befindet. Nur der `Set`-Teil der Eigenschaftenprozedur wird in diesem Fall ausgeführt.

[1] Etwa: »Zugreifer«.

Ermitteln von Eigenschaften

Umgekehrt können Sie mit der Anweisung

```
Irgendetwas = Object.EineEigenschaft
```

den Inhalt der Eigenschaft wieder auslesen. In diesem Fall wird nur der *Get*-Accessor der Eigenschaftenprozedur ausgeführt, die das Ergebnis mit `Return` zurückliefert und dieses der Objektvariablen `Irgendetwas` zuweist, die natürlich vom gleichen Typ wie die Eigenschaft sein muss.

Mit diesem Wissen können wir unsere Klasse jetzt folgendermaßen umschreiben:

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 10\MiniAdressoClass V2
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
Public Class Kontakt
    Private myVorname As String
    Private myNachname As String
    Private myPLZ As String
    Private myOrt As String

    Public Property Vorname() As String
        Get
            Return myVorname
        End Get
        Set(ByVal value As String)
            myVorname = value
        End Set
    End Property

    Public Property Nachname() As String
        Get
            Return myNachname
        End Get
        Set(ByVal value As String)
            myNachname = value
        End Set
    End Property

    Public Property PLZ() As String
        Get
            Return myPLZ
        End Get
        Set(ByVal value As String)
            myPLZ = value
        End Set
    End Property
```

```
    Public Property Ort() As String
        Get
            Return myOrt
        End Get
        Set(ByVal value As String)
            myOrt = EllipseString(value, 30)[2]
        End Set
    End Property

    Private Function EllipseString(ByVal text As String, ByVal MaxLength As Integer) As String

        Dim tmpText As String

        If text.Length > MaxLength Then
            tmpText = text.Substring(0, MaxLength - 3) + "..."
        Else
            tmpText = text
        End If
        Return tmpText
    End Function

End Class
```

Sie sehen an diesem einfachen Beispiel, wie der Schutz der eigentlichen Datenstruktur durch Eigenschaftenprozeduren vonstatten geht: Die Daten, die in einer Klasseninstanz gespeichert werden, weisen private Zugriffsmodifizierer auf (mehr zu Zugriffsmodifizierern finden Sie im Abschnitt »Zugriffsmodifizierer für Klassen, Methoden und Eigenschaften – Gültigkeitsbereiche definieren« ab Seite 365). Dadurch können die Daten nicht mehr direkt von außen, sondern nur noch durch den Klassencode selbst verändert werden. Die Feldvariablen heißen auch nicht mehr wie zuvor, sondern tragen den Zusatz bzw. das Präfix *my* als rein textlichen Hinweis darauf, dass es sich dabei um »meine« Variablen aus Klassen- bzw. Klasseninstanzsicht handelt. Dies ist eine durchaus übliche, aber nicht vorgeschriebene Konvention – Sie können die Benennung der Feldvariablen nach eigenem Belieben vornehmen; Sie sollten allerdings darauf achten, dass Sie den Feldvariablen andere Namen als den dazugehörigen, von außen zugreifbaren Eigenschaften geben, denn das würde ansonsten zu einem Kompilierungsfehler führen.

Die Eigenschaftenprozeduren schließlich heißen so wie die ursprünglich öffentlichen Feldvariablen. Aus Entwicklersicht hat sich damit nichts an der Handhabung geändert – im Ergebnis allerdings schon, denn durch die Eigenschaftenroutinen wird nun der Zugriff auf die Feldvariablen exklusiv geregelt. In diesem kleinen Beispiel sehen Sie, dass der Ort durch dieses Verfahren niemals mehr als 30 Zeichen aufweisen wird. Bei mehr als 30 Zeichen wird durch das Einbinden der `EllipseString`-Methode die Anpassung der Zeichenkette mit den Auslassungszeichen vorgenommen, und auf den eigentlichen Datenspeicher des Orts – die Feldvariable `myOrt` – kann von außen niemand mehr zugreifen, da ihr Zugriff durch Verwendung des Zugriffsmodifizierers `private` für diese (wie auch für alle anderen Feldvariablen) geschützt ist. Genau das ist es, was eines der Hauptanliegen einer Klasse ist, nämlich *Daten zu kapseln und den Zugriff auf sie durch Klassencode zu reglementieren*.

[2] Fußnote im Listing? Wieso nicht: Der Fachlektor dieses Buchs regt an dieser Stelle zurecht eine Alternative an: Anstelle `myOrt` hier wirklich »kaputt« zu machen, könnte man die Ursprungszuweisung auch in der vollen Länge speichern und beim *Abrufen* der Eigenschaft die Methode `EllipseString` aufrufen. So könnte man im Bedarfsfall immer noch auf die ursprüngliche Eigenschaftendefinition zugreifen.

Viel Tipparbeit sparen mit automatisch implementierten Eigenschaften

Neu seit Visual Basic 2010 sind die so genannten automatisch implementierten Eigenschaften – sie werden Ihnen sicherlich auch häufig unter ihrem englischen Begriff *Auto Implementing Properties* begegnen. In vielen Fällen ist es so, dass Sie Eigenschaften zum Beispiel zur Datenbindung in einer Windows Presentation Foundation (dazu im entsprechenden Teil dieses Buchs mehr) benötigen, der Code aber nichts weiter macht, als die entsprechenden Feldvariablen Ihrer Klasse zu setzen, also in etwa folgendermaßen:

```
Public Property Nachname() As String
    Get
        Return _Nachname
    End Get
    Set(ByVal value As String)
        _Nachname = value
    End Set
End Property
```

In diesem Fall unterscheidet sich die Feldvariable, die der Eigenschaft zugrunde liegt, nur durch einen Unterstrich (_) vom Eigenschaftennamen. Solche Arten von Eigenschaften, die an sich keinen zusätzlichen Code benötigen, kosten besonders viel Zeit durch viel Tipparbeit, und genau die muss mit Visual Basic ab der Version 2010 nicht mehr sein:

Mit der Zeile

```
Public Property Nachname() As String
```

erreichen Sie nämlich exakt dasselbe. Der Compiler sorgt dafür, dass die entsprechenden Getter- und Setter-Prozeduren erstellt werden, und er legt sogar eine entsprechende Feldvariable an.

HINWEIS Wie schon erwähnt, legt Visual Basic unter der Motorhaube eine Feldvariable an. Doch so weit unter der Motorhaube liegt diese Feldvariable gar nicht, denn sie ist sogar direkt aus ihrer Klasse heraus erreichbar, wenngleich sie nicht so offensichtlich zu finden ist: IntelliSense verschweigt sie nämlich. Das bedeutet jedoch nicht, dass es sie nicht gibt.

Wie im oben stehenden Beispiel nennt der Visual Basic-Compiler die Feldvariable, die sich hinter einer Eigenschaft verbirgt, genau wie die Eigenschaft selbst, nur mit einem Unterstrich vorangestellt. Heißt ihre Eigenschaft also `Nachname`, so lautet die entsprechende Feldvariable `_Nachname`.

In diesem Fall wird die Feldvariable auch als *Hilfsfeld* oder *Backing Field* bezeichnet. Dieses Hilfsfeld ist übrigens immer `Private`, unabhängig davon, mit welchem Zugriffsmodifizierer Sie die Eigenschaft selbst ausgestattet haben. Sollten Sie die Eigenschaft statisch definieren (siehe im Abschnitt »Statische Eigenschaften« ab Seite 371), wird das Hilfsfeld natürlich auch statisch definiert – genau so, wie Sie es beim eigenhändigen Implementieren einer Eigenschaft mit explizit definierter Feldvariable machen würden.

Wieso Sie Eigenschaften bzw. automatisch implementierten Eigenschaften vor öffentlichen Feldern den Vorrang geben sollten, verrät Ihnen übrigens die Abschnitt »Öffentliche Variablen oder Eigenschaften – eine Glaubensfrage?« ab Seite 345.

Vorbelegen von Eigenschaftswerten von automatisch implementierten Eigenschaften

Es ist durchaus üblich, in der Regel im Konstruktor einer Klasse, Eigenschaften mit Standardwerten vorzubelegen. Auch das ist bei automatisch implementierten Eigenschaften möglich, ohne großartige Verrenkungen zu machen und ohne zusätzlichen Code im Konstruktor schreiben zu müssen, und es funktioniert genau wie beim Initialisieren von Variablen. Ein paar Beispiele:

```
Public Property Nachname() As String = "Ramona Leenings"
```

Diese Anweisung definiert eine Eigenschaft vom Typ `String` (Zeichenkette) und belegt sie mit der Zeichenfolge »Ramona Leenings« vor.

```
Property Cabriotemperaturen As Integer() = New Integer() {23, 25, 31}
```

Die Anweisung definiert eine Eigenschaft von Typ `Integer-Array` und belegt sie mit drei Elementen vor.

```
Public Property Nachnamen As New List(Of String) From {"Leenings", "Feigenbaum", "Löffelmann"}
```

Diese Anweisung definiert eine Eigenschaft vom generischen Typ `List(Of String)` und verwendet einen Auflistungsinitialisierer (*Collection Initializer*), um die generische Liste mit drei Zeichenketten vorzubelegen.

Eigenschaften mit Parametern

In Visual Basic können Eigenschaften, wie Funktionen, beliebige Parameter übernehmen. Die Parameter werden in der Eigenschaftenprozedur ebenso wie bei Funktionen eingebunden, und auf die Parameter lässt sich dann ebenfalls Zugriff nehmen. Dazu ein Beispiel. Angenommen, Sie haben eine Eigenschaftenprozedur etwa wie die folgende definiert:

```
Property EigenschaftMitParametern(ByVal Par1 As Integer, ByVal Par2 As String) As Integer

    Get
        If Par1 = 0 Then
            Return 10
        Else
            Return 20
        End If
    End Get

    Set(ByVal Value As Integer)
        If Par2 = "Klaus" Then
            'MachIrgendEtwas
        ElseIf Par1 = 20 Then
            'MachIrgendetwasAnderes
        End If
    End Set

End Property
```

dann können Sie diese beispielsweise mit folgenden Anweisungen zuweisen bzw. abfragen:

```
'Eigenschaft mit Parametern setzen.
locObjektInstanz.EigenschaftMitParametern(10, "Klaus") = 5

'Eigenschaft mit Parametern abfragen.
If locObjektInstanz.EigenschaftMitParametern(20, "Test") = 20 Then
    'Mach Irgendetwas
End If
```

HINWEIS Wenn Sie sehr alten Code anpassen oder aus Gründen der Abwärtskompatibilität immer noch mit (möglicherweise in Visual Basic 6.0 geschriebenem) COM unterwegs sind, denken Sie daran: Im Gegensatz zu Visual Basic 6.0 werden Wertetypen-Parameter als Wert und nicht als Referenz übergeben. Änderten Sie einen Wert innerhalb einer Eigenschaftenprozedur in Visual Basic 6.0, so veränderte sich auch der Wert der Variablen im aufrufenden Code (es sei denn, er war explizit durch Einklammern vor dem Überschreiben geschützt oder Sie haben, wie es in .NET-Versionen von Visual Basic automatisch geschieht, die Übergabe explizit mit `ByVal` vorgenommen). In .NET-Versionen von Visual Basic übergeben Sie standardmäßig den *Wert* an eine Eigenschaft – Sie arbeiten also in jedem Fall mit einer Kopie der übergebenden Variablen.

Eine Differenzierung mit `Set` und `Let` wie in Visual Basic 6.0 gibt es übrigens in .NET-Versionen von Visual Basic nicht mehr. Da alles von `Object` abgeleitet ist und dadurch im Grunde genommen ausschließlich Objekte manipuliert werden, ist auch das `Let` überflüssig geworden, und alles müsste mit `Set` manipuliert werden – und dann kann man `Set` auch direkt weglassen.

WICHTIG Eigenschaften mit Parametern sollten Sie nur in Ausnahmefällen verwenden, da sie eine besondere Eigenart von Visual Basic sind. Im Gegensatz zu Visual Basic kennt beispielsweise C# zwar auch Eigenschaften, kann aber nur auf parameterlose Eigenschaften oder Default-Eigenschaften zugreifen (mehr dazu im Abschnitt »Default-Eigenschaften« im folgenden Abschnitt). Wenn Sie also Klassen im Team oder für die breite Öffentlichkeit entwickeln und erwarten, dass Ihre Assemblys auch von Entwicklern genutzt werden, die in einer anderen .NET-Sprache als Visual Basic entwickeln, sollten Sie auf Eigenschaften mit Parametern nach Möglichkeit ganz verzichten.

Default-Eigenschaften (Standardeigenschaften)

Default Properties Mithilfe einer Default-Eigenschaft können Sie in VBA und bis Visual Basic 6.0 bestimmen, welche Eigenschaft verwendet wird, wenn Sie beim Zugriff auf ein Objekt gar keine Eigenschaft angegeben haben. Default-Eigenschaften (Standardeigenschaften) haben in Visual Basic 6.0 eine erleichternde Funktion für schreibfaule Entwickler gehabt. Das Typsystem unter .NET erlaubt derartige Typunsicherheiten nicht – denn bei der Zuweisung beispielsweise von

```
Dim EineTextBox as TextBox
Dim einObjekt as Object
.
.
.
einObjekt = EineTextBox
```

ist natürlich nicht sichergestellt, ob Sie die TextBox selbst oder das Ergebnis der Default-Eigenschaft der Textbox an `einObjekt` zuweisen wollen. Ausnahmen bilden dabei parametrisierte Eigenschaften, da durch die

Signatur der Eigenschaft deutlich wird, dass Sie nicht das Objekt selbst, sondern das Resultat einer parametrisierten Eigenschaft zurückliefern wollen.[3] In diesem Fall ergibt das auch Sinn, denn:

Stellen Sie sich vor, Sie entwickeln eine `NumericList`-Klasse, die verschiedenste numerische Werte als Liste speichert sowie verwaltet und die es erlaubt, auf diese mit einem Index zuzugreifen, um diese Elemente abzurufen. Gäbe es keine Eigenschaften mit Parametern, müssten Sie ein Element der Liste auf folgende Weise abfragen:

```
'Index setzen.
numListe.Index = 5
'Element abfragen.
MachEtwasMit = numListe.Item
```

Das wäre natürlich äußerst umständlich. Einfacher wird es, so wie es auch tatsächlich funktioniert, nämlich durch die Spezifizierung des Indexes und die Abfrage in einer Zeile. In Visual Basic ist das kein Problem durch eine Eigenschaft mit einem Parameter, etwa wie folgt:

```
MachEtwasMit = numListe.Item(5)
```

Noch einfacher verhält es sich, wenn die Eigenschaft `Item` in diesem Beispiel zur Default-Eigenschaft erklärt wird.[4] Dann brauchen Sie den Eigenschaftennamen nämlich gar nicht mehr anzugeben, und die folgende Zeile wäre ausreichend:

```
MachEtwasMit = numListe(5)
```

Die entsprechende Definition für die `Item`-Eigenschaft sähe in diesem Fall folgendermaßen aus:

```
Default Public Property Item(ByVal Index As Integer) As Integer

    Get
        Return myInterneListe(Index)
    End Get

    Set(ByVal Value As Integer)
        myInterneListe(Index) = Value
    End Set

End Property
```

WICHTIG Im Gegensatz zu VBA oder Visual Basic 6.0 gibt es in den .NET-Versionen von Visual Basic keine parameterlosen Default-Eigenschaften. Ebenfalls gut zu wissen: Default-Eigenschaften können nicht als statisch deklariert werden. Das hindert Sie aber natürlich nicht daran, eine Eigenschaft zu implementieren, die sich statisch verhält.

[3] Ähnlich wie bei Überladungen, bei denen auch nur durch die Signaturen unterschieden wird, welche der mehreren vorhandenen Funktionen gemeint ist.

[4] Wie gesagt: In C# sind Default-Eigenschaften sogar die einzige Möglichkeit, einer Eigenschaft einen Parameter mitzugeben.

Öffentliche Variablen oder Eigenschaften – eine Glaubensfrage?

Jetzt haben Sie schon so viel über Eigenschaften erfahren – vielleicht fragen Sie sich, wieso man sie anstelle von einfachen öffentlichen Feldvariablen einsetzen sollte.

So lange, wie Sie Eigenschaften in einer Klasse nur benötigen, um irgendwelche Werte zu speichern, aber beim Abfragen oder Setzen dieser Werte nichts Weiteres passieren muss, wären öffentliche Variablen eigentlich ausreichend. Die Klasse

```
Class MitEinerEigenschaft
    Public DieEigenschaft As Integer
End Class
```

erfüllt zunächst nämlich den gleichen Zweck wie die folgende Klasse

```
Class AuchMitEinerEigenschaft

    'Selbst-implementierende Eigenschaft
    Public Property DieEigenschaft() As Integer

End Class
```

die seit Visual Basic 2010 noch nicht einmal mehr Schreibarbeit erfordert. Die zweite Methode ist generell der ersten Methode vorzuziehen, und zwar aus folgenden Gründen, die Ihnen teilweise schon bekannt sind:

- **Datenkapselung** Eigenschaften kapseln und reglementieren den Zugriff auf Daten, wie Sie im vorangegangenen Beispiel bereits gesehen haben. Auch wenn Sie eine Eigenschaft nur komplett an eine Feldvariable durchreichen – für mögliche Erweiterungen, was Reglementierungsalgorithmen anbelangt, sind Sie auf jeden Fall schon mal vorbereitet.
- **Datenbindung** Felder lassen sich grundsätzlich nicht für die Datenbindung[5] verwenden. Vielfach werden nicht nur Quellen wie Datenbanken, sondern auch Objekte, wie beispielsweise Auflistungen (Collections), als Datenquellen verwendet. Öffentliche Felder (also öffentliche Feldvariablen) können dabei nicht als Datenquelle dienen.
- **Polymorphie** Ein weiteres wichtiges Argument ist das Ersetzen von Eigenschaften durch die so genannte Polymorphie beim Vererben von Klassen, das wir im nächsten Kapitel noch kennen lernen werden. Vorab schon mal so viel: Eine einmal als öffentlich deklarierte Variable bleibt für alle Zeiten öffentlich. Sie können in vererbten Klassen keine zusätzliche Steuerung hinzufügen, die ihren Zugriff reglementieren würde. Haben Sie hingegen Ihre Daten nur durch Eigenschaftenprozeduren nach außen offen gelegt, können Sie zu einem späteren Zeitpunkt noch zusätzliche Regeln (Bereichsabfragen, Fehler abfangen) hinzufügen oder vorhandene Verhaltensweisen modifizieren. Sie brauchen dazu die ursprüngliche Klasse kein bisschen zu verändern.

[5] Ein Verfahren, bei dem man Eigenschaften eines Objekts an die eines anderen bindet. Damit kann man Daten zwischen Objekten automatisch synchronisieren, und häufig wird dieses Verfahren bei Steuerelementen angewandt, die den Inhalt einer Datenklasse darstellen sollen und deren Inhalt sich automatisch ändert, wenn sich der Datenklasseninhalt ändert.

Eigenschaften sind also öffentlichen Feldern in jedem Fall vorzuziehen. Und für den Fall, für den Sie nicht auf selbstimplementierende Eigenschaften zurückgreifen können, gibt es in Form der Visual Basic-Codeausschnittsbibliothek eine Eingabehilfe, die der folgende Kasten beschreibt.

Zeit sparen beim Erstellen von Eigenschaftenprozeduren mit Codeausschnitten

An der Beispielklasse der vorherigen Abschnitte lässt sich eindrucksvoll aufzeigen, dass das Verwalten, Regeln und auch Schützen der Feldvariablen einer Klasse durch entsprechende Eigenschaftenprozeduren durchaus sinnvoll ist, Ihnen jedoch im Einzelfall aber auch ein Mehr an Tipparbeit abverlangt, gerade wenn Sie Steuercode erstellen müssen und Sie nicht auf selbstimplementierende Eigenschaften zurückgreifen können. Codeausschnitte können Ihnen hier jedoch helfen, ein wenig Zeit gut zu machen:

Mithilfe von Codeausschnitten können Sie gerade beim »Properties kloppen«, wie es umgangssprachlich in Entwicklerkreisen genannt wird, eine Menge an Zeit sparen. Möchten Sie eine neue Eigenschaft in Ihrer Klasse einführen, die auf einer Feldvariablen basiert, verfahren Sie am besten wie folgt:

- Schreiben Sie die Zeichenfolge **Pro** an die Stelle, an der Sie die neue Eigenschaftenprozedur erstellen wollen. Sie sehen anschließend einen Dialog, etwa wie auch in Abbildung 10.1 zu sehen.

Abbildung 10.1 Um eine Eigenschaftenprozedur durch Codeausschnitte einzufügen, schreiben Sie die Zeichenfolge »Pro«. Drücken Sie, wie im Hinweis der IntelliSense-Liste zu sehen, zweimal [⇥], um den Property-Codeausschnitt einzufügen, ...

- Drücken Sie zweimal [⇥], um den `Property`-Codeausschnitt einzufügen. Sie sehen anschließend ein Szenario, etwa wie es Abbildung 10.2 zeigt.
- Ändern Sie anschließend die Feldvariable, die zunächst mit dem Vorgabenamen `newPropertyValue` eingefügt wurde, in den Namen, den Sie für die Feldvariable vorsehen.

```
Public Class Kontakt

    Private newPropertyValue As String
    Public Property NewProperty() As String
        Get
            Return newPropertyValue
        End Get
        Set(ByVal value As String)
            newPropertyValue = value
        End Set
    End Property

End Class
```

Abbildung 10.2 ... um dann anschließend die ausgewählte Feldvariable zu bearbeiten und mit [⇥] den passenden Datentyp hinzuzufügen, ...

- Sobald Sie ⇆ gedrückt haben, ändern sich alle Referenzen der Feldvariablen im Codeblock in den neu gewählten Namen. Gleichzeitig gelangen Sie zum nächsten editierbaren Feld des Codeausschnitts, dem Typen der Eigenschaftenprozedur.

```
Public Class Kontakt

    Private myVorname As String
    Public Property NewProperty() As String
        Get
            Return myVorname
        End Get
        Set(ByVal value As String)
            myVorname = value
        End Set
    End Property

End Class
```

Abbildung 10.3 ... den Sie anschließend im Bedarfsfall ebenfalls ändern, um schließlich mit einem weiteren (i) das Erstellen der Eigenschaftenprozedur mit dem Editieren des eigentlichen Eigenschaftennamens abzuschließen.

- Ändern Sie diesen und drücken Sie abermals ⇆. Damit ändern Sie die Typen sowohl für die Eigenschaftenprozedur (und deren `value`-Variablen, mit dem im `Set`-Accessor der Zuweisungswert an die Eigenschaft übergeben wird) als auch für die korrelierende Feldvariable, die in der Eigenschaftenprozedur gesetzt wird.
- Schließlich ändern Sie noch den Namen der Eigenschaft. Damit sind Sie mit der Definition der Eigenschaftenprozedur fertig.

Im Ergebnis stehen alle Eigenschaften und deren entsprechende Feldvariablen jeweils als Pärchen zusammen. Viele Entwickler mögen das nicht in dieser Form; sie bevorzugen, dass Feldvariablen am Anfang der Klassencodedatei definiert werden – Sie können die Feldvariablen natürlich anordnen, wie Sie möchten; das tut der Funktionalität der Klasse keinen Abbruch.

Der Konstruktor einer Klasse – bestimmen, was bei New passiert

Wenn Sie eine Klasse in ein Objekt instanziieren, dann schaffen Sie gesteuert durch die Common Language Runtime zunächst mal nur den entsprechenden Speicherplatz für die Feldvariablen auf dem Managed Heap. Intern passieren zwar noch ein paar weitere Dinge, die für die Verwaltung des Objekts auf dem Managed Heap zuständig sind, doch das ist es dann erst einmal gewesen.

In vielen Fällen ist es aber erforderlich, dass gewisse Pflegearbeiten sozusagen zum Vorbereiten der Klasse auf ein korrektes Funktionieren ebenfalls durchgeführt werden. Denken Sie zum Beispiel daran, was passiert, wenn Sie eine neue Schaltfläche für Ihre Windows Forms-Anwendung instanziieren: Hier muss man auch die notwendige Windows-Infrastruktur berücksichtigen, damit nicht nur Platz für die Daten, die die Steuerinformationen für die Schaltfläche bilden, auf dem Managed Heap geschaffen wird, sondern notwendigerweise unter anderem auch die `CreateWindow`-Funktion der Datei *Win32.Dll* des Windows-Betriebssystems aufgerufen wird, damit die Schaltfläche auch aus Betriebssystemsicht entstehen kann.

Oder, um es an einem einfachen Beispiel festzumachen: Wie würden Sie in unsere bisherige Beispielklasse eine Funktionalität implementieren, die automatisch das Datum der Klassenerstellung festhält, sobald eine Instanz der `Kontakt`-Klasse erstellt wurde? Prinzipiell gibt es dazu zwei Möglichkeiten:

- Sie implementieren eine weitere Eigenschaft, sagen wir namens `ErstellungsZeitpunkt`. Und wann immer Sie eine Instanz der `Kontakt`-Klasse erstellen, setzen Sie die Eigenschaft auf `Date.Now` – eine Funktion, die die aktuelle Datum/Uhrzeit-Kombination ermittelt. Die Gefahr dabei: Sie könnten diesen Schritt vergessen, und genau das widerspricht eigentlich guter objektorientierter Programmierung, bei der Sie idealerweise Objekte schaffen, die in einem derart hohen Maße selbstverwaltend sind, dass Sie solche Dinge gar nicht erst vergessen oder falsch machen *können* – ganz ähnlich, wie Sie mit Eigenschaftenprozeduren den Zugriff auf Klassenmember so regeln, dass Sie auch diese nicht versehentlich kompromittieren können.
- Die bessere Alternative: Sie implementieren eine Funktionalität, bei der Sie auf die Eigenschaft `ErstellungsZeitpunkt` nur noch lesend zugreifen können – wie das im Detail funktioniert, erfahren Sie ausführlicher im Abschnitt »Unterschiedliche Zugriffsmodifizierer für Eigenschaften-Accessors« ab Seite 367; das ist aber der eher unwichtige Teil. Viel wichtiger in diesem Zusammenhang: Sie implementieren eine spezielle Methode, die die Feldvariable, auf der die Eigenschaft basiert, automatisch mit dem Erstellungsdatum ausstattet, sobald ein Objekt der Klasse durch ihre Instanziierung ins Leben gerufen wird. Damit garantieren Sie, dass keine Instanz der Klasse erstellt werden kann, ohne dass das Erstellungsdatum gesetzt wird.

Genau diesen letzten Punkt wollen wir im Folgenden umsetzen, und dazu bedienen wir uns der Möglichkeit, einen Konstruktor in unserer Klasse zu implementieren. Der Programmcode des Konstruktors wird nämlich genau dann ausgeführt, nachdem die CLR das Schaffen der standardmäßigen Infrastruktur hinter sich gebracht hat. Das Erstellen eines Konstruktors funktioniert in Visual Basic mit der `Sub New`:

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 10\MiniAdressoClass V3
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
Public Class Kontakt

    Private myVorname As String
    Private myNachname As String
    Private myPLZ As String
    Private myOrt As String
    Private myErstellungsdatum As Date

    Sub New()
        myErstellungsdatum = Date.Now
    End Sub

    Public ReadOnly Property ErstellungsDatum() As Date
        Get
            Return myErstellungsdatum
        End Get
    End Property
    .
    .
    .
End Class
```

Die relevanten Änderungen sind hier im Listing fett hervorgehoben und erleichtern so das Verstehen der Funktionsweise: Sobald Sie eine neue Instanz der Klasse ins Leben rufen, legt die CLR das Objekt intern an und ruft im Anschluss daran sofort den Konstruktor der Klasse auf. Der Konstruktor – definiert durch `Sub New` – setzt nun die Feldvariable `myErstellungsdatum` auf die aktuelle Uhrzeit/das aktuelle Datum. Möchten Sie nun im Folgenden herausfinden, wann die Klasse erstellt worden ist, greifen Sie einfach auf das Erstellungsdatum über die entsprechende Nur-Lesen-Eigenschaft zu. Durch diese Vorgehensweise sind die drei Wunschpunkte unseres Pflichtenheftes erfüllt:

- Die Klasse kann nicht instanziiert werden, ohne dass das Erstellungsdatum festgehalten wird – das erledigt nämlich der Konstruktor mit `Sub New`.
- Das Erstellungsdatum ist durch die Eigenschaftenprozedur abrufbar.
- Das Erstellungsdatum wird aber durch die Verwendung einer *Nur-Lesen-Eigenschaft* eben *nur abrufbar* gemacht, kann also nachträglich nicht verändert werden.

Und damit ist dieser Funktionsblock unserer Klasse bombensicher!

Parametrisierte Konstruktoren

Sicherlich haben Sie bemerkt, dass die Klasse aus dem ersten Beispiel ein wenig unhandlich war. Sie musste erst instanziiert werden, daran führt sowieso nie ein Weg vorbei, und anschließend konnten Sie dem aus ihr hervorgehenden Objekt durch die Benutzung von Eigenschaften entsprechende Inhalte zuweisen. Von Klassen, die Sie aus .NET Framework möglicherweise bereits verwendet haben, wissen Sie aber sicherlich schon, dass die Instanziierung einer Klasse mit `New` und der *Angabe eines Parameters* die Initialisierung der Klasseninstanz einfach und unkompliziert macht. Die bisher verwendete Beispielklasse ist hinsichtlich dessen ein wenig armselig, doch das soll sich jetzt ändern.

Wenn Sie möchten, dass Code im Konstruktor einer Ihrer Klassen ausgeführt wird, in dem Sie dann beispielsweise Feldvariablen initialisieren können, dann implementieren Sie ebenfalls `Sub New`. Der einzige Unterschied zu normalen Methoden besteht darin, dass der Prozedurenname aus einem Schlüsselwort besteht (nämlich `New`), welches Visual Studio-Editor und -Compiler als ein solches erkennen und blau markieren. Konstruktoren können übrigens nur aus `Sub`- und nicht aus `Function`-Methoden bestehen – Letzteres würde ja auch keinen Sinn ergeben, da das Instanziieren einer Klasse mit `New` ja eigentlich schon das instanziierte Objekt sozusagen als »Funktionsergebnis« von `New` zurückliefert.[6]

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 10\MiniAdressoClass V4
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Zurück zum Beispielprogramm: Um die Klasseninstanz zu deklarieren *und* ihr Initialwerte zuzuweisen, benötigen Sie mit dieser Version des Beispielprogramms nur noch eine kleine Modifikation. Der Klassencode des Beispielprogramms dieser Version sieht dann folgendermaßen aus:

[6] Wobei diese Erklärung rein technisch natürlich falsch ist und lediglich als Eselsbrücke dienen soll.

```
Public Class Kontakt

    Private myVorname As String
    Private myNachname As String
    Private myPLZ As String
    Private myOrt As String
    Private myErstellungsdatum As Date

    'Parameterloser Konstruktor
    Sub New()
        myErstellungsdatum = Date.Now
    End Sub

    'Konstruktor übernimmt Initialisierungsparameter
    Sub New(ByVal Vorname As String, ByVal Nachname As String,
            ByVal PLZ As String, ByVal Ort As String)

        'Und ruft - und das geht NUR als erstes oder gar nicht -
        'die parameterlose Konstruktorprozedur auf.
        Me.New()

        'Feldvariablen mit den übergebenen Werten initialisieren.
        myVorname = Vorname
        myNachname = Nachname
        myPLZ = PLZ
        myOrt = Ort
    End Sub

    Public ReadOnly Property ErstellungsDatum() As Date
        Get
            Return myErstellungsdatum
        End Get
    End Property
    .
    .
    .
End Class
```

Und mit dieser Version unserer Klasse können wir einen Kontakt nun auf die folgende Weise instanziieren und auch Standardparameter direkt zuweisen:

```
Module Hauptmodul

    Sub Main()

        'Hier 'mal farbtechnisch was Anderes im Konsolenfenster
        Console.BackgroundColor = ConsoleColor.White
        Console.ForegroundColor = ConsoleColor.Black
        Console.Clear()

        'Eine Adresse mit Parametrisiertem Konstruktor angeben:
        Dim AngisKontakt As New Kontakt("Angela", "Wördehoff", _
                                        "99999", "Bahamas")

        'So geht's natürlich ab 2008 auch - das *ersetzt* aber nicht
        'den parametrisierten Konstruktor. Außerdem ist's mehr Tipparbeit.
        Dim LöffelsKontakt As New Kontakt With {.Vorname = "Klaus", _
                                                .Nachname = "Löffelmann", _
```

```
                                                .PLZ = "59555", _
                                                .Ort = "Lippstadt"}
    End Sub
End Module
```

HINWEIS Vielleicht könnte man argumentieren, dass parametrisierte Konstruktoren seit Visual Basic 2008 ein wenig obsolet geworden sind, weil Visual Basic ja nunmehr auch über Eigenschafteninitialisierer beim Konstruktor mit dem Schlüsselwort `With` verfügt – wie hier im zweiten Beispiel zu sehen. Aber das ist nur begrenzt richtig. Denn zum einen können Sie diese Eigenschaftenzuweisungen nur durch die Eigenschaftenprozeduren selbst steuern – ein parametrisierter Konstruktor will hier vielleicht noch andere Wertüberprüfungen übernehmen. Zum anderen ist die Standardwerteübergabe für bestimmte Member mit einem parametrisierten Konstruktor immer noch weniger Tipparbeit – und vielleicht möchten Sie ohnehin Parameter an die Klasse übergeben, die vielleicht durch gar keine Eigenschaft abdeckt oder offengelegt sind. Denkbar wäre es beispielsweise, dem Konstruktor auch eine `Kontakt`-Instanz zu übergeben, damit er einen vorhandenen Kontakt klont.

Darüber hinaus kennen andere .NET-Sprachen zum Teil keine Eigenschafteninitialisierer – diese werden nämlich ausschließlich durch den Compiler umgesetzt, der entsprechenden Code dafür generiert. Falls Sie also Assemblys entwickeln, auf die von anderen .NET-Sprachen ebenfalls zugegriffen werden soll, sind parametrisierte Konstruktoren sicherlich auch ein erweiterter Komfort für den Entwickler, der Ihre Klassen verwendet.

Konstruktoren lassen sich übrigens genau wie Methoden überladen – nur beim gegenseitigen Aufrufen müssen bestimmte Dinge beachtet werden. Mehr dazu hält der Abschnitt »Überladen von Methoden, Konstruktoren und Eigenschaften« ab Seite 357 dieses Kapitels für Sie bereit.

Hat jede Klasse einen Konstruktor?

Oh ja. Zwar gab es ganz am Anfang dieses Kapitels eine Version des Beispielprogramms, die weder über Eigenschaften noch über einen Standardkonstruktor im Quellcode verfügte, aber für einen solchen Fall verlangt es das CTS,[7] dass der Compiler entsprechende Maßnahmen ergreift, damit dieser Zustand eben nicht eintritt. Und das heißt was im Klartext?

Dazu ein Beispiel: Lassen Sie uns noch einmal zu der vorherigen Version des Beispielprogramms zurückkehren, in der es erst einen Konstruktor gab – die V3-Version unseres Beispiels. Diesen Konstruktor werden wir dann anschließend auskommentieren, und eine der Feldvariablen mit einem Standardwert vorinitialisieren, also in etwa wie folgt:

```
Public Class Kontakt

    'Sub New()
    '    myErstellungsdatum = Date.Now
    'End Sub

    Private myVorname As String
    Private myNachname As String
    Private myPLZ As String
    Private myOrt As String
```

[7] Wir erinnern uns an Kapitel 3: Das **C**ommon **T**ype **S**ystem ist ein System, das strikte Typbindung und Codekonsistenz erfordert und damit die Common Language Runtime (CLR) zur Coderobustheit zwingt.

```
Private myErstellungsdatum As Date = Date.Now

Public ReadOnly Property ErstellungsDatum() As Date
    Get
        Return myErstellungsdatum
    End Get
End Property
```

Fassen wir zusammen: In dieser Version ist nun der Standardkonstruktor auskommentiert und damit nicht mehr vorhanden; gleichzeitig gibt es eine Zuweisung der aktuellen Zeit/Datum-Kombination an die entscheidende Feldvariable `myErstellungsdatum` – was im Übrigen auch statthaft ist, denn der Compiler meckert ja nicht. Die Frage: Wann und durch was wird diese Initialisierung eigentlich ausgeführt?

Wie in Kapitel 3 schon kurz angerissen, übersetzt der Compiler Ihr Programm nicht direkt in nativen Maschinencode, sondern in eine »Zwischensprache« namens *Intermediate Language* oder kurz *IL*. Die Rahmendaten Ihres Programms, beispielsweise Konstanten, definierte Attribute, aber auch Informationen über die Version, werden in einem speziellen Datenbereich in der gleichen Datei abgelegt. Diese Daten nennt man in .NET *Metadaten*. Wenn Sie nun ein Programm starten, dann gibt es zu diesem Zeitpunkt natürlich noch nichts, was der Prozessor ausführen kann, denn er versteht ja – was Intel-Plattformen anbelangt – nur Pentium- bzw. x86-Code. Es muss also einen Mechanismus geben, der zwischen den Zeitpunkten von Programmstart und Programmausführung Ihr Programm in Maschinencode übersetzt – und dieses Werkzeug ist Bestandteil der CLR und nennt sich *JITter*.[8] Den »vorläufigen« IL-Code können Sie sich mit einem speziellen Werkzeug aus der .NET-Werkzeugsammlung anschauen – er offenbart alle Wahrheiten, auch solche, die der Compiler ohne Ihr Zutun hinzugefügt hat.

Mit diesem Wissen bewaffnet, können Sie sich jedes kompilierte Visual Basic-Programm in seinem »IL-Zustand« anschauen und herausfinden, was der VB-Compiler daraus gemacht hat. Dabei sind gar nicht so sehr die einzelnen Befehle entscheidend, sondern die Metadaten, die auch Auskunft darüber geben, aus welchen Komponenten, Typen, Signaturen etc. sich Ihr Programm zusammensetzt.

Zusätzliche Werkzeuge für .NET

Es gibt einige zusätzliche und nützliche Tools zu Visual Studio 2010, bei denen es Microsoft nach ein paar Anläufen endlich mal geschafft hat, diese dort zu platzieren, wo man sie erwartet: Im Startmenü in der Visual Studio-Programmgruppe, und zwar unterhalb des Eintrags *Microsoft Windows SDK Tools*.

SDK ist die Abkürzung von Software Development Kit, das alle Tools beinhaltet, mit denen Sie auch ohne Visual Studio Programme kompilieren (aber nicht editieren und die Projektdateien verwalten!), austesten, Ressourcen anpassen und vieles mehr können. Anders als bei den Vorgängerversionen scheint das SDK in seiner vollständigen Version bei Visual Studio 2010 mit installiert zu werden.

Im Folgenden werden wir uns für ein bestimmtes Tool namens IL-Disassembler aus dieser erwähnten Programmgruppe interessieren, mit dem wir uns den CIL- bzw. MSIL-Code sowie die Metadaten anschauen können, aus denen Ihre Assemblys bestehen, bevor der JITter ausführbaren Code daraus macht.

[8] Zur Auffrischung: JIT ist die Abkürzung von »Just in time«, auf Deutsch etwa »genau rechtzeitig«.

Nach diesem kurzen Exkurs lassen Sie uns zurück zu unserem Vorhaben kommen:

- Wählen Sie *Projektmappe neu erstellen* aus dem Menü *Erstellen*, damit der Visual Basic-Compiler den IL-Code innerhalb der Exe-Datei (also unserer Ausgabe-Assembly) neu erstellt.
- Starten Sie anschließend den Intermediate-Language-Disassembler (siehe vorheriger grauer Kasten) aus der Programmgruppe *Microsoft Windows SDK Tools*.
- Wählen Sie *Öffnen* aus dem Menü *Datei* und im Dialog, der jetzt erscheint, die .EXE-Datei des Beispielprogramms. Sie finden die Programmdatei *MiniAdressoClass.Exe* im Projektverzeichnis und dort im Unterverzeichnis *\bin\Debug*.

HINWEIS Sie werden im Debug-Verzeichnis übrigens zwei .EXE-Dateien finden – eine davon endet mit *vshost.exe*. Diese verwenden Sie bitte *nicht*. Bei ihr handelt es sich um ein kleines Hilfsprogramm, das von der Visual Studio-IDE benötigt wird, um einerseits die Erstellzeiten Ihrer Projekte zu beschleunigen und um andererseits das vielfach gewünschte *Edit and Continue* (*Bearbeiten und Fortfahren*) zu ermöglichen, das es Ihnen gestattet, Änderungen an Ihrem Programm vorzunehmen, *während* Sie es debuggen. *Edit and Continue* steht Ihnen nach wie vor in Visual Studio übrigens nicht zur Verfügung, wenn dieses unter einem 64-Bit-Windows-Betriebssystem läuft und Ihr Projekt auch gegen 64 Bit kompiliert wird (bzw. unter einem 64-Bit-Betriebssystem die Voreinstellung »Any« gewählt ist – Sie können x86-Code erzwingen, indem Sie mit dem Konfigurationsmanager die entsprechende Zielplattform forcieren, und dann steht Ihnen auch Edit and Continue zur Verfügung.

Abbildung 10.4 Die Metadaten der EXE-Datei erlauben die Ansicht der Programmstruktur im IL-Disassembler

- Im Fenster, das anschließend erscheint, öffnen Sie den Zweig *MiniAdressoClass* und den sich darunter befindlichen Zweig *MiniAdressoClass.Kontakt* ebenfalls – denn darin sind alle Elemente enthalten, aus denen unsere `Kontakt`-Klasse besteht.
- Aus dem Menü *Ansicht* wählen Sie die Option *Quellzeilen anzeigen*. Wenn wir dann gleich den CIL-Code betrachten, werden die entsprechenden Visual Basic-Quellcodezeilen mit dort eingeblendet. Sie sollten anschließend ein Bild vor Augen haben, das etwa dem in Abbildung 10.4 entspricht.

In dieser Abbildung sehen Sie die Struktur des Programms. Sie finden Elemente wieder, die Sie selbst bestimmt haben – beispielsweise die Funktionsnamen. Und Sie erkennen ebenfalls eine Methode, für die Sie nicht der unmittelbare Urheber waren – die Methode `.ctor`.

Und das liegt daran, dass diese Methode keine Methode im herkömmlichen Sinne ist, sondern vielmehr handelt es sich bei ihr um einen Konstruktor, um den Part der Klasse also, der entsteht, wenn Sie in Visual Basic eine `Sub New` implementieren würden. Genau die haben Sie aber im Quellcode auskommentiert, bevor Sie die Anwendung neu erstellt und den IL-Disassembler gestartet haben.

Der Compiler hat also selbstständig dafür gesorgt, dass ein Konstruktor in der Klasse vorhanden ist, und genau das schreibt das CTS auch vor: Jede instanziierbare Klasse muss einen Konstruktor haben. Ob der dann aufrufbar ist, steht noch mal auf einem anderen Papier geschrieben; Sie können beispielsweise den Konstruktor der Klasse mit einem `Private`-Zugriffsmodifizierer ausstatten, sodass die Klasse nur noch durch sich selbst instanziiert werden könnte, aber eben nicht mehr von außen.[9]

Abbildung 10.5 Der disassemblierte IML-Code des Standardkonstruktors

[9] Ergibt keinen Sinn, meinen Sie? Im Gegenteil! Es gibt ein so genanntes Entwurfsmuster (Pattern), eine bestimmte Art, eine Klasse zu konzipieren, die sich Singleton nennt. Dieses Entwurfsmuster hat lediglich einen privaten Konstruktor, aber dafür eine öffentliche statische Methode, über die Sie eine Instanz anfordern können. Ein Beispiel für eine Singleton-Klasse finden Sie am Ende des nächsten Kapitels.

Auch ohne genau zu wissen, welche IL-Anweisung für welche Aufgabe zuständig ist, wird eines doch sofort deutlich: Im ersten Teil der Methode wird offensichtlich ein weiterer Konstruktor aufgerufen, und das ist richtig: Da jede Klasse, die Sie neu erstellen, implizit vom Grundtypen der obersten Hierarchieebene `Object` abgeleitet wird, wenn Sie es nicht anders bestimmen, wird hier der Konstruktor dieser Basisklasse aufgerufen.[10] Und anhand der eingeblendeten Quellzeile können wir hier auch sicher sein, dass die ursprünglich nur unterhalb des Klassenrumpfs stehende Variablendeklaration/-definition des Members `myErstellungsdatum` ihren Weg in den Konstruktor gefunden hat – denn dort wird sie nämlich mit der aktuellen Uhrzeit/dem aktuellen Datum ausgestattet.

Dieses Beispiel zeigt also, dass Ihnen der Visual Basic-Compiler eine ganze Menge an Arbeit abgenommen hat. Er hat dafür gesorgt,

- dass es einen Standardkonstruktor mit der Methode *.ctor* überhaupt gibt, auch wenn Sie nicht explizit mit `Sub New()` einen implementiert haben.
- dass innerhalb des Standardkonstruktors der Standardkonstruktor der Basisklasse `Object` (`call instance void [mscorlib]System.Object::.ctor()`) aufgerufen wird und
- dass alle Feldvariablen, falls erforderlich, innerhalb dieses Standardkonstruktors definiert – also mit den Werten »gefüllt« werden, sofern Sie dies bei deren Deklarierung unterhalb des Klassenrumpfs angegeben haben.

WICHTIG Und jetzt wissen Sie auch, wieso Sie die Feldvariablen einer Klasse, die vorinitialisiert werden müssen, grundsätzlich am Platz ihrer Deklaration definieren sollten. Wenn Sie nämlich mehrere überladene Konstruktoren (siehe Abschnitt »Überladen von Methoden, Konstruktoren und Eigenschaften« ab Seite 357) in einer Klasse definiert haben, laufen Sie auf keinen Fall Gefahr, diese Vorinitialisierung der Variablen in allen Konstruktoren zu vergessen, da Ihnen der Visual Basic Compiler diese Arbeit abnimmt. Das folgende Beispiel zeigt, was gemeint ist. Die auskommentierten Zeilen fügt der Compiler durch die Definition der deklarierten Feldvariablen mit Standardwerten auf Modulebene selbst dem Code hinzu:

```
Public Class EineKlasse

    'Hier sollten Initialisierungen von Klassenfeldern passieren,
    'damit der entsprechende Initialisierungscode automatisch in alle Konstruktoren wandert!
    Private myFeldvariable As Date = Date.Now
    Private myAndereFeldvariable As Integer = 5
    Private myName As String

    'Sub New()
    '    myFeldvariable = Date.Now
    '    myAndereFeldvariable = 5
    'End Sub

    Sub New(ByVal name As String)
        'myFeldvariable = Date.Now
        'myAndereFeldvariable = 5
        myName = name
    End Sub
End Class
```

[10] Was *Ableiten von Klassen* genau bedeutet und wieso genau hier der Konstruktor der Basisklasse aufgerufen wird, erfahren Sie im nächsten Kapitel.

Klassenmethoden mit Sub und Function

Klassencode wird nicht nur in Eigenschaftenprozeduren ausgeführt. Auch Methoden, die in Visual Basic durch `Sub` oder `Function` definiert werden, gehören dazu. Methoden nicht statischer Natur (solche also, die nicht, wie in »Statische Methoden« ab Seite 369 beschrieben über die Klasseninstanz definiert sind) sind dabei Methoden, die mit den Feldvariablen einer Klasse arbeiten sollten.

Ein Beispiel dafür ist das folgende: Eine Funktion wie beispielsweise `KontaktText` könnte für das Zurückliefern der textrepräsentativen Ausgabe der Klasse sorgen – und die entsprechenden Routinen dafür würden dann folgendermaßen aussehen:

```
Public Function KontaktText() As String

    Return Nachname & ", " & Vorname & ", " & _
            PLZ & ", " & Ort

End Function
```

Diese Membermethode der Klasse bedient sich direkt der Instanzvariablen und gibt sie verkettet als einen zusammenhängenden String aus. Das bedeutet, dass wir entsprechende Änderungen auch bei der Ausgaberoutine im Modul machen können, die die einzelnen Instanzen verwendet:

```
Sub AdressenAusgeben(ByVal von As Integer, ByVal bis As Integer)

    For c As Integer = von To bis
        Console.WriteLine(c.ToString("000") & ": " & Kontakte(c).KontaktText())
    Next

End Sub
```

Die Methode `KontaktText` wird hier für jede auszugebende `Kontakt`-Instanz aufgerufen und liefert natürlich auch für jede Instanz ein unterschiedliches Ergebnis – hieran sieht man deutlich, dass es sich um eine Membermethode handelt.

BEGLEITDATEIEN Die Begleitdateien zum Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 10\MiniAdressoClass V5
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Übrigens: Eine Anwendung startet immer mit einer `Sub Main` – das gilt auch für Windows-Anwendungen, die Sie als Windows Forms-Projekte unter Visual Basic erstellen. Warum das so ist, lesen Sie in Kapitel 2.

Überladen von Methoden, Konstruktoren und Eigenschaften

Falls Sie zu den alten Visual Basic 6.0-Hasen gehören, dann kennen Sie sicherlich den Vorteil von optionalen Parametern. Das Überladen von Funktionen in .NET ist ein nur auf den ersten Blick ähnliches, aber letzten Endes dennoch völlig anderes Konzept. Gemeinsam haben beide Konzepte, dass sie eine Liberalisierung von Parameterübergaben an Funktionen ermöglichen. Das war es aber dann auch schon mit den Gemeinsamkeiten.

Mit dem Überladen von Funktionen geben Sie Ihren Klassen eine enorme Flexibilität und Anpassungsgabe. Überladen von Funktionen bedeutet: Sie erstellen verschiedene Funktionen mit gleichen Namen, die sich nur durch den Typ, die Typreihenfolge oder die Anzahl der übergebenen Parameter unterscheiden. Als Beispiel schauen Sie sich bitte den folgenden Codeausschnitt an:

```
Sub EineProzedur()
    'Tu was.
End Sub

Sub EineProzedur(ByVal ein_Parameter As Integer)
    'Tu was anderes.
End Sub

Sub EineProzedur(ByVal ein_anderer_Parameter As String)
    'Tu was anderes.
End Sub

Sub EineProzedur(ByVal ein_Parameter As Integer, ByVal ein_anderer_Parameter As String)
    'Tu was anderes.
End Sub

Sub EineProzedur(ByVal ein_ganz_anderer_Parameter As Integer)
    'Fehler: ein Integer als Parameter gab's schon mal.
    'Die Methode 'EineProzedur' wurde mehrfach mit identischen Signaturen definiert.
End Sub
```

Es ist, als würde die Signatur – so nennt man die Mischung aus Parametertypen und Parameterreihenfolge beim Aufrufen einer Funktion – Bestandteil des Namens werden, und daran wird dann erkennbar, welche der vorhandenen `EineProzedur`-Varianten aufgerufen werden soll. Der Variablenname hat damit übrigens überhaupt nichts zu tun – nur der übergebene Typ ist für die Identifizierung der Signatur entscheidend.

Aus diesem Grund bereitet die letzte `Sub` des Beispiels auch Probleme. Ihr wird genau wie der ersten eine Variable vom Typ `Integer` übergeben. Zwar ist der Variablenname ein anderer, aber darauf kommt es überhaupt nicht an – Namen sind hier tatsächlich nicht mehr als Schall und Rauch.

Wozu eignet sich die Funktionsüberladung in der Praxis? Nun, die Anwendung von Klassen wird dadurch ungleich flexibler. Schon bei unserem ständigen Klassenbeispielbegleiter kommen Sie spätestens beim Anwenden der Klasse `Kontakt` in den Genuss des Komforts von überladenen Methoden bzw. Konstruktoren, für die das Gesagte gleichermaßen gilt. Sie müssen nicht wissen, welche Methode beispielsweise für welche

Teilaufgabe zuständig ist; Sie können einfach die (eine) Funktion verwenden, und IntelliSense[11] unterstützt Sie bei der Auswahl der richtigen Signatur sogar. Wenn Sie die Zeile in Ihr Programm eingeben, die die Klasse in ein Objekt instanziiert, dann zeigt Ihnen IntelliSense, nachdem Sie die geöffnete Klammer hinter `New Kontakt` eingegeben haben, die einzelnen Signaturen des Konstruktors an, etwa wie in Abbildung 10.6 zu sehen.

```
'Eine Adresse mit Parametrisierten Konstruktor angeben:
Dim AngisKontakt As New Kontakt(
                          ▲ 2 of 2 ▼ New(Vorname As String, Nachname As String, PLZ As String, Ort As String)
'So geht's natürlich ab
'den parametrisierten Konstruktor. Außerdem ist's mehr Tipparbeit.
Dim LöffelsKontakt As New Kontakt With {.Vorname = "Klaus", _
                                        .Nachname = "Löffelmann", _
                                        .PLZ = "59555", _
                                        .Ort = "Lippstadt"}
```

Abbildung 10.6 IntelliSense hilft Ihnen bei der Auswahl der richtigen Signatur bei überladenen Methoden wie Konstruktoren

Im Gegensatz zu optionalen Parametern können Sie Methoden bzw. Konstruktoren implementieren, deren Versionen völlig unterschiedliche Dinge tun. Natürlich haben beide Konstruktorversionen des Beispiels thematisch miteinander zu tun (sollten sie auch, anderenfalls sollten Sie ihnen komplett andere Namen geben). Wichtig ist: Die Funktionsweise der beiden überladenen Prozeduren kann im Gegensatz zu *einer* Prozedur mit optionalen Parametern nicht nur komplett anders implementiert werden (die erste initialisiert etwa die Klasse mit einem Integer, die andere durch die Angabe eines römischen Numerals), die beiden Methoden sind auch visuell sauber voneinander getrennt.

HINWEIS Das Prinzip der Funktionsüberladung funktioniert für `Sub`- genauso wie für `Function`-Methoden. Allerdings ist Folgendes bei der Verwendung von Funktionen wichtig zu wissen: Der Rückgabetyp kann nicht als Differenzierungskriterium von Signaturen verwendet werden. Das heißt im Klartext:

```
Function EineFunktion() As Integer
    'Tu was.
End Function

Function EineFunktion(ByVal AndereSignatur As Integer) As Integer
    'Das funktioniert.
End Function

Function EineFunktion() As String
    '"Function EineFunktion() As Integer" und "Function EineFunktion() As String"
    ' können sich nicht gegenseitig überladen, da sie sich nur durch Rückgabetypen unterscheiden.
End Function
```

[11] Zur Wiederholung: So nennt sich die Eingabehilfe des Codeeditors, mit deren Hilfe sich Elementnamen vervollständigen, alle Elemente eines Objekts in Listen anzeigen oder Funktions- und Eigenschaftenüberladungen sich schon bei der Codeeingabe sichtbar machen lassen. Abbildung 10.6 zeigt IntelliSense in Aktion.

Die ersten beiden Funktionen sind o.k., da sich die Signaturen voneinander unterscheiden. Die letzte Funktion unterscheidet sich von der ersten allerdings nur durch den Rückgabetyp, und aus diesem Grund meldet der Visual Basic-Compiler schon zur Entwurfszeit einen Fehler.

TIPP Sie können – zur besseren Lesbarkeit – das `Overloads`-Schlüsselwort verwenden, um die Überladung einer Methode deutlich zu machen:

```
Overloads Function EineFunktion() As Integer
    'Wenn Overloads verwenden,...
End Function

Overloads Function EineFunktion(ByVal AndereSignatur As Integer) As Integer
    '...dann bei bei den Funktionen
End Function
```

Dabei sollten Sie berücksichtigen: Wenn Sie sich für das `Overloads`-Schlüsselwort entscheiden, müssen Sie es bei *allen* Methodenvariationen mit Überladungen verwenden.

Methodenüberladung und optionale Parameter

Als Alternative zum Überladen von Methoden bietet Ihnen Visual Basic noch das Hilfsmittel der optionalen Parameter an. Optionale Parameter haben gegenüber überladenen Methoden Vorteile, aber auch Nachteile:

- Optionale Parameter werden von anderen .NET-Programmiersprachen unter Umständen nicht unterstützt – das gilt für C# bis zur Version 2008 bzw. bis zu .NET Framework 3.5. Erst danach ist C# in der Lage, mit optionalen Parametern umzugehen. Wenn Sie in Ihren Klassen optionale Parameter verwenden, rechnen Sie damit, dass andere .NET-Sprachen unter Umständen keine Unterstützung für optionale Parameter bieten, gerade wenn Sie noch nicht gegen das neueste .NET Framework entwickeln können.
- Der Einsatz von optionalen Parametern macht den Code, der innerhalb einer Methode mit diesen optionalen Parametern umgehen muss, schwerer lesbar und weniger gut wiederverwendbar. Wenn Sie optionale Parameter verwenden, müssen Sie Fallunterscheidungen durchführen, indem Sie durch die Abfrage von Standardwerten, die nur aus Konstanten definiert werden können, herausfinden, welche Parameter vom Aufrufer übergeben wurden und welche nicht, falls ihr Programm beim Weglassen eines Parameters anders reagieren soll, als wenn der Parameter übergeben wird. Eine Methode, die nur aufgrund ihrer Parameter vielleicht zwei völlig verschiedene Dinge macht, trägt dann quasi den gequetschten »Doppelcode« in einem Funktionsrumpf. Mit überladenen Funktionen hingegen haben Sie zwei Problemlösungen auch optisch sauber voneinander getrennt.
- Sollten Sie ohne strikte Typbindung arbeiten (`Option Strict Off`) und gleichzeitig überladene Funktionen und optionale Parameter für die gleichen Methoden verwenden, bedeutet das einen unglaublichen Leistungsverzicht, da die Laufzeitbibliothek von Visual Basic unter Umständen erst herausfinden muss, welche der Routinen am ehesten zum angegebenen Parameter passt. Bei sehr flexiblen Parameterübergaben können das obendrein potenzielle Fehlerquellen werden, die ich – ganz ehrlich gesagt – bei Fehlverhalten niemals debuggen möchte.
- **Aber:** Gerade die Entwicklung von Office-Anwendungen mit VSTO ist in Visual Basic oder mit C# ab der Version 2010 sicherlich angenehmer, eben *weil* die Office-Librarys optionale Parameter intensiv verwenden, die in Visual Basic dann anwendbar sind. In C# müssen Sie beim Methodenaufruf für jeden nicht angegebenen Parameter ein entsprechendes Auslassungsschlüsselwort übergeben (`System.Reflection.Missing.Value`).

HINWEIS Bis hoch zur Version 4.0 von .NET Framework findet sich keine Methode in .NET Framework selbst, die optionale Parameter verwendet.

Anwenden von optionalen Parametern

Das Definieren von optionalen Parametern für eine Methode ist vergleichsweise simpel und auch das Anwenden von optionalen Parametern vergleichsweise durchsichtig. Das folgende Beispiel demonstriert einerseits, wie optionale Parameter für eine Methode definiert werden und andererseits, wie Sie dann anschließend eine solche Methode aufrufen können – in verschiedenen Versionen:

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 10\OptionalParametersDemo
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
Module OptionalParametersDemo

    Private Const MAXDATEVALUE As Date = #12/31/9999 11:59:59 PM#
    Private Const DEFAULTDATEFORMAT = "dd.MM.yyyy"

    Sub Main()

        'Aufruf mit gestern:
        PrintTimeDeltaOpt(Date.Now.AddDays(-1))

        'Aufruf vorgestern, anderes Ausgabeformat:
        PrintTimeDeltaOpt(Date.Now.AddDays(-2), DateFormatString:="dddd, dd. MMMM yyyy")

        'So wäre das auch gegangen:
        PrintTimeDeltaOpt(Date.Now.AddDays(-2), , , "dddd, dd. MMMM yyyy")

        'Alles selbst ausgewählt:
        PrintTimeDeltaOpt(Date.Now.AddDays(-2), Date.Now, "TimeSpan: ", _
                          "dddd, dd. MMMM yyyy", "Start time: ", "End time: ", True)

        Console.WriteLine("Taste drücken zum Beenden:")
        Console.ReadKey()

    End Sub

    Public Sub PrintTimeDeltaOpt(ByVal Starttime As Date, _
                                 Optional ByVal Endtime As Date = MAXDATEVALUE, _
                                 Optional ByVal TimeSpanCaption As String = "Zeitdifferenz: ", _
                                 Optional ByVal DateFormatString As String = DEFAULTDATEFORMAT, _
                                 Optional ByVal StartTimeCaption As String = "Startzeit: ", _
                                 Optional ByVal EndTimeCaption As String = "Endzeit: ", _
                                 Optional ByVal PrintDelimitter As Boolean = True)

        'Typisch für optionale Parameter:
        'Hier müssen wir "erkennen", ob ein optionaler Parameter
        'angegeben wurde, und wenn nicht, diesen durch den Wert ersetzen,
        'den wir eigentlich "meinen".
        If Endtime = MAXDATEVALUE Then
            Endtime = DateTime.Now
        End If
```

```
        Dim zeitDifferenz = Endtime - Starttime

        Console.WriteLine(StartTimeCaption & "{0:" & DateFormatString & "}" & vbNewLine &
                          EndTimeCaption & "{1:" & DateFormatString & "}" & vbNewLine &
                          TimeSpanCaption & "{2:dd\.hh\:mm}", Starttime, Endtime,
                          zeitDifferenz)

        If PrintDelimitter Then
            Console.WriteLine(New String("-"c, 75))
            Console.WriteLine()
        End If
    End Sub

End Module
```

Drei Dinge sind an diesem kleinen Demoprogramm erwähnenswert:

In der ersten in Fettschrift gesetzten Zeile finden Sie die Anwendung von so genannten *benannten Parametern*. Diese erlauben es, Werte für Parameter zu bestimmen, ohne diese Parameter an die Stelle in die Parameterliste zu setzen, an der sich durch die definierte Methodensignatur eigentlich befinden müssten.

Diese Art der Parameterauswahl ist auch ein Vorteil von optionalen Parametern gegenüber Methodenüberladungen, bei denen benannte Parameter nicht möglich sind. Im Übrigen unterstützt auch IntelliSense benannte Parameter, wie die Abbildungen 10.7 und 10.8 zeigen.

Abbildung 10.7 Auch IntelliSense bietet Unterstützung für benannte Parameter, die in der Liste an der Zeichenfolge »:= «zu erkennen sind, ...

Abbildung 10.8 ...und auch bei den Signaturen zeigt Ihnen IntelliSense nicht nur die Variablennamen, sondern auch die Standardwerte, mit denen die Parameter beim Weglassen initialisiert werden

Zusätzlich sehen Sie hier, auf welche Weise Sie durch das Hintereinanderschreiben von Kommas Parameter gezielt auslassen können. Und schließlich demonstriert dieses Beispiel auch die negativen Seiten von optionalen Parametern, nämlich ein gezieltes Ersetzen von Werten für den Fall, dass ein Standardwert, der ja nur Konstantencharakter haben kann (`DateTime.Now` wäre zum Beispiel also nicht als Standardwert möglich, da dies keine Konstante ist), durch einen dynamischen Wert ersetzt werden muss: In unserem Beispiel ersetzen wir das Maximaldatum, das als Standardwert beim Weglassen des Parameters `Endtime` angenommen wird, einfach durch die aktuelle Uhrzeit (bzw. das aktuelle Datum), was in diesem Szenario natürlich viel mehr Sinn macht.

Gegenseitiges Aufrufen von überladenen Methoden

Das Überladen von Methoden begegnet Ihnen in .NET Framework an jeder Ecke. In der Regel wird die Funktionsüberladung von .NET Framework verwendet, um dem Benutzer die Handhabung so angenehm wie möglich zu machen. So werden ihm Signaturen für bestimmte Funktionen mit nur sehr wenigen Parametern angeboten, um Tipparbeit sparen, und gleichzeitig andere Versionen derselben Funktion mit sehr viel mehr Parametern für die größtmögliche Flexibilität.

Die `WriteLine`-Methode, die Sie ja schon mehrfach im Einsatz bewundern durften, ist hier ein sehr anschauliches Beispiel, da sie mit nicht weniger als 18 Überladungen daherkommt. Wenn Sie im Codeeditor von Visual Basic die Anweisung **Console.WriteLine** eingeben und anschließend die geöffnete Klammer eintippen, zeigt Ihnen IntelliSense die Signaturen der 18 verschiedenen Überladungen an.

Abbildung 10.9 Die WriteLine-Methode hat nicht weniger als 18 Überladungen vorzuweisen!

Natürlich wäre das Überladen von Methoden keine wirkliche Arbeitserleichterung, wenn Entwickler die eigentliche Funktionalität für alle überladenen Methoden immer wieder implementieren müssten. Überladene Methoden können sich deswegen gegenseitig – vom Sonderfall `Sub New` einmal abgesehen – ohne Einschränkungen aufrufen.

Der übliche Weg, den Benutzern Ihrer Klassen (und damit meistens sich selbst) große Flexibilität in die Hand zu geben, besteht darin, eine universale Methode zu entwickeln, die alles kann, und sie anschließend durch Überladungen »nach unten abzuspecken«.

Ein Beispiel: Angenommen, Sie haben eine Klasse entwickelt, die eine Methode zur Verfügung stellt, mit der man Kreise auf den Bildschirm malen kann. Sie nennen diese Methode *Circle*, und diese stellt in ihrer Universalversion folgende Fähigkeiten zur Verfügung:

```
Public Sub Circle(ByVal Xpos As Integer, ByVal YPos As Integer, ByVal XRadius As Integer, _
                  ByVal YRadius As Integer, ByVal StartAngle As Integer, ByVal EndAngle As Integer)
        'Hier steht der Code für Circle.
End Sub
```

Nun benötigen Sie die komplette Flexibilität dieser Methode aber nur in den seltensten Fällen. Sie müssen nicht jedes Mal den x- *und* y-Radius der Figur und noch seltener den Start- und Endwinkel des Kreises mit angeben. Der Aufruf einer solchen vereinfachten Version ist einfach praktischer, daher nennt man solche Methoden auch Convenience-Methoden. Eine abgespeckte Version könnte daher wie folgt aussehen:

```
Public Sub Circle(ByVal Xpos As Integer, ByVal YPos As Integer, ByVal Radius As Integer)
    Circle(Xpos, YPos, Radius, Radius, 0, 359)
End Sub
```

Auf den ersten Blick sieht es so aus, als würde sich die Funktion selbst aufrufen. Macht sie aber nicht. Da die Signatur der verwendeten `Circle`-Methode (2. Zeile) nicht der eigenen Signatur (1. Zeile) entspricht, schaut der Visual Basic-Compiler, welche `Circle`-Methode infrage kommt, und greift in diesem Beispiel auf die erste Variante der Methode zurück.

Gegenseitiges Aufrufen von überladenen Konstruktoren

Ein wenig problematischer wird es, wenn sich überladene Konstruktoren gegenseitig aufrufen sollen – betriebsbedingt gibt es dabei nämlich Einschränkungen. Mit dem Wissen des vorherigen Abschnitts starten Sie vielleicht rein aus dem Gefühl heraus den Versuch, das Beispiel auch bei den Konstruktoren wie in Abbildung 10.10 gezeigt umzugestalten.

Abbildung 10.10 Wenn Sie versuchen, eine andere Konstruktormethode nicht an erster Stelle eines Konstruktors aufzurufen, quittiert Ihnen Visual Basic das mit einem Fehler

Visual Basic quittiert diesen Aufruf mit einem simplen Syntaxfehler[12] und lässt den Aufruf ganz einfach nicht zu.

Übrigens ist es an dieser Stelle schon mal wichtig zu wissen, dass Sie genau spezifizieren, welches `New` Sie aufrufen. Prinzipiell gibt es bei jeder Klasse nämlich zwei Versionen von `New`, die aber nicht durch Überladen entstanden sind: das `New`, also der Konstruktor in Ihrer Klasse, und das `New` der Klasse, von der Sie Ihre Klasse abgeleitet haben. Da Sie hier im Beispiel nicht explizit bestimmt haben, aus welcher Klasse Sie ableiten, hat Ihre Klasse automatisch von `Object` geerbt. Natürlich hat auch `Object` einen Konstruktor, und Sie können sowohl Methoden der Basisklasse als auch Methoden Ihrer Klasse aufrufen. Wenn Sie `Me` angeben, spezifizieren Sie Ihre Klasse; wenn Sie `MyBase` angäben, würden Sie damit die Basisklasse spezifizieren – in diesem Fall die Klasse `Object`. Mehr über das Vererben und über Aufrufe von Funktionen von Basisklassen erfahren Sie im nächsten Kapitel.

Überladen von Eigenschaftenprozeduren mit Parametern

Eigenschaften, die Parameter entgegennehmen, lassen sich wie Funktionen überladen. Zum Einsatz kommt dieses Prinzip fast ausschließlich bei Default-Eigenschaften (siehe »Default-Eigenschaften (Standardeigenschaften)« auf Seite 343). Wie bei »normalen« Funktionen kann nur die Eigenschaftensignatur (die Reihenfolge, die Typen und die Anzahl der übergebenden Parameter), nicht aber der Rückgabetyp zur Unterscheidung herhalten. Daher ist die Überladung von Eigenschaften auch nur dann möglich, wenn mindestens eine Eigenschaftenvariation Parameter entgegennimmt. Ein Beispiel für das Überladen von Eigenschaften:

```
Property Überladung() As Integer
    Get
       'Hier der Code für die Ermittlung der Eigenschaft.
    End Get
    Set(ByVal Value As Integer)
        'Hier der Code für die Zuweisung.
    End Set
End Property

Property Überladung(ByVal Par1 As Integer) As Integer
    Get
        'Hier der Code für die Ermittlung der Eigenschaft.
    End Get
    Set(ByVal Value As Integer)
        'Hier der Code für die Zuweisung.
    End Set
End Property

Property Überladung() As String
    Get
        'Geht nicht, da sich diese Eigenschaft...
    End Get
    Set(ByVal Value As String)
        'nur durch den Rückgabetyp von der ersten unterscheidet.
    End Set
End Property
```

[12] Kleine nostalgische Randbemerkung: Das ist eine Fehlermeldung, die es schon beim Commodore 64 gab – der verfügte auch über ein Microsoft-Basic – und die sich bis heute gehalten hat!

HINWEIS Für das Überladen von Eigenschaften benötigen Sie mindestens eine Eigenschaftenprozedur, die Parameter entgegennimmt. Denken Sie an das bereits Gesagte, dass das – von Standardeigenschaften mal abgesehen – eine Visual Basic-spezifische Vorgehensweise ist und andere Sprachen wie C# möglicherweise daran hindern würde, von Eigenschaften Gebrauch zu machen, die Parameter entgegennehmen. C# beispielsweise kennt nämlich außer bei Standardeigenschaften keine Eigenschaftenprozeduren, die Parameter entgegennehmen.

Zugriffsmodifizierer für Klassen, Methoden und Eigenschaften – Gültigkeitsbereiche definieren

Zugriffsmodifizierer sind Schlüsselwörter, mit denen Sie in Visual Basic bestimmen können, von wo aus der Zugriff auf ein Element einer Klasse gestattet ist. Die Zugriffsmodifizierer `Private` und `Public` haben Sie schon kennen gelernt. Diese bestimmen, ob auf ein Element nur innerhalb eines bestimmten Gültigkeitsbereiches zugegriffen werden kann (`Private`) oder von überall aus (`Public`). Welche weiteren Zugriffsmodifizierer es für Objekte, Klassen und Funktionen/Eigenschaften gibt, zeigen die folgenden Tabellen:

Zugriffsmodifizierer bei Klassen

HINWEIS Wenn nichts anderes angegeben ist, werden Klassen standardmäßig als `Friend` deklariert.

Zugriffsmodifizierer	CTS-Bezeichnung	Beschreibung

Private	Private	Als Private können Klassen nur dann definiert werden, wenn sie geschachtelt in einer anderen Klasse definiert sind. Beispiel:
		`Public Class A`
		`  Private Class B`
		`  End Class`
		`End Class`
		
		`Public Class C`
		`  'Zugriff verweigert, Class B ist Private!`
		`  Dim b as A.B`
		`End Class`

Public	Public	Sie können auf die Klasse uneingeschränkt von außen zugreifen, auch aus anderen Assemblys heraus
Friend	Assembly	Sie können innerhalb der Assembly auf die Klasse zugreifen, aber nicht aus einer anderen Assembly heraus
Protected	Family	Es gilt zunächst das für `Private` Gesagte. Zusätzlich gilt: Auch von der Klasse abgeleitete Klassen können auf die mit `Protected` gekennzeichneten und geschachtelten »inneren« Klassen zugreifen.
Protected Friend	FamilyOrAssembly	Der Zugriff auf die geschachtelte Klasse ist in abgeleiteten Kassen und von Klassen der gleichen Assembly aus möglich

Tabelle 10.1 Mögliche Zugriffsmodifizierer für Klassen in Visual Basic

Zugriffsmodifizierer bei Prozeduren (Subs, Functions, Properties)

HINWEIS Wenn nichts anderes angegeben wird, werden `Subs`, `Functions` und `Properties` standardmäßig als `Public` deklariert. Sie sollten gerade bei diesen Elementen aber auf jeden Fall einen Zugriffsmodifizierer verwenden, damit beim Blättern durch den Quellcode schnell deutlich wird, welchen Zugriffsmodus ein Element innehat.

Zugriffsmodifizierer	CTS-Bezeichnung	Beschreibung
Private	Private	Nur innerhalb einer Klasse kann auf die Prozedur zugegriffen werden
Public	Public	Sie können auf die Prozedur uneingeschränkt von außen zugreifen, auch aus anderen Assemblys heraus
Friend	Assembly	Sie können innerhalb der Assembly auf die Prozedur zugreifen, aber nicht aus einer anderen Assembly heraus
Protected	Family	Nur innerhalb der Klasse oder einer abgeleiteten Klasse kann auf die Prozedur zugegriffen werden
Protected Friend	FamilyOrAssembly	Nur innerhalb der Klasse, einer abgeleiteten Klasse oder innerhalb der Assembly kann auf die Prozedur zugegriffen werden

Tabelle 10.2 Mögliche Zugriffsmodifizierer für Prozeduren in Visual Basic

Zugriffsmodifizierer bei Variablen

Variablen, die auf Klassenebene nur mit `Dim` deklariert werden, gelten als `Private`, also nur von der Klasse aus zugreifbar. Variablen, die innerhalb eines Codeblocks oder auf Prozedurebene deklariert werden, gelten nur für den entsprechenden Codeblock. Innerhalb eines Codeblocks können Sie nur die `Dim`-Anweisung und keine anderen Zugriffsmodifizierer verwenden. Auf Prozedurebene können Sie eine Variable zusätzlich als `Static` deklarieren. Mehr über den `Static`-Zugriffsmodifizierer erfahren Sie im Abschnitt »Statische Elemente« (ab Seite 368).

Zugriffsmodifizierer	CTS-Bezeichnung	Beschreibung
Private	Private	Nur innerhalb einer Klasse kann auf die Variable zugegriffen werden. Variablen innerhalb von Prozeduren oder noch kleineren Gültigkeitsbereichen können nicht explizit als Private definiert werden, sind es aber standardmäßig.
Public	Public	Von außen kann auf die Klassenvariable uneingeschränkt zugegriffen werden. Sie sollten Variablen aber bestenfalls als Protected deklarieren und sie nur durch Eigenschaften nach außen offen legen. Mehr zu diesem Thema erfahren Sie im Abschnitt »Öffentliche Variablen oder Eigenschaften – eine Glaubensfrage?« auf Seite 341.
Friend	Assembly	Sie können innerhalb der Assembly auf die Klassenvariable zugreifen, aber nicht aus einer anderen Assembly heraus. Es gilt das für `Public` Gesagte.
Protected	Family	Nur innerhalb derselben oder einer abgeleiteten Klasse kann auf die Variable zugegriffen werden. Variablen sollten in Klassen, bei denen Sie davon ausgehen, dass sie später des Öfteren vererbt werden, als Protected definiert werden, damit abgeleitete Klassen ebenfalls darauf zugreifen können.
Protected Friend	FamilyOrAssembly	Nur innerhalb der Klasse, einer abgeleiteten Klasse oder innerhalb der Assembly kann auf die Klassenvariable zugegriffen werden. Von dieser Kombination sollten Sie absehen.
Static	–	Sonderfall in Visual Basic. Lesen Sie dazu bitte die Ausführungen im Abschnitt »Statische Elemente« (ab Seite 368).

Tabelle 10.3: Mögliche Zugriffsmodifizierer für Variablen in Visual Basic

Diese Tabellen sollen Ihnen kompakt und auf einen Blick die Zugriffsmodifizierer verdeutlichen. Die CTS-Bezeichnungen der Zugriffsmodifizierer benötigen Sie, wenn Sie sich den IML-Code einer Klasse anschauen, um zu erkennen, welchen Zugriffsmodus beispielsweise eine Methode hat.

Unterschiedliche Zugriffsmodifizierer für Eigenschaften-Accessors

Seit Visual Studio 2005 ist es möglich, dass die `Get`- und `Set`-Accessors unterschiedliche Zugriffsmodifizierer aufweisen. So haben Sie beispielsweise die Möglichkeit, zu bestimmen, dass zwar eine bestimmte Eigenschaft von jedem Ort aus gelesen werden kann (`Public`), aber Eigenschaften nur von derselben Klasse aus geschrieben werden dürfen (`Private`). Im Code würde eine solche Eigenschaft folgendermaßen ausschauen:

```
Module Module1

    Sub Main()
        Dim locEigenschaftenTest As New EigenschaftenTest("Text für Eigenschaft")

        'Das Auslesen der Eigenschaft ist problemlos möglich
        Console.WriteLine("Eigenschaft enthält: " & locEigenschaftenTest.EineEigenschaft)

        'FEHLER: Der Set-Zugriffsmodifizierer verbietet aber das Schreiben,
        'weil er 'private' ist.
        locEigenschaftenTest.EineEigenschaft = "Neuer Text"
    End Sub

End Module

Public Class EigenschaftenTest

    Private myEineEigenschaft As String

    Sub New(ByVal textFürEigenschaft As String)
        'Ist erlaubt - die Klasse darf die
        'Eigenschaft beschreiben!
        EineEigenschaft = textFürEigenschaft
    End Sub

    Public Property EineEigenschaft() As String
        Get
            Return myEineEigenschaft
        End Get
        Private Set(ByVal value As String)
            myEineEigenschaft = value
        End Set
    End Property

End Class
```

Dieses kleine Beispiel besteht aus zwei Einheiten – einem Modul und einer Klasse. Die Klasse `EigenschaftenTest` enthält einen parametrisierten Konstruktor sowie eine Eigenschaft, die aber Accessors mit unterschiedlichen Zugriffsmodifizierern hat.

Innerhalb des Konstruktors (`Sub New`) ist der Schreibzugriff auf die Eigenschaft problemlos möglich, da aus der Klasse selbst heraus auch auf Elemente zugegriffen werden kann, deren Zugriff mit `Private` eingeschränkt wurde. Innerhalb des Moduls funktioniert der schreibende Zugriff allerdings nicht mehr, da sich das Programm zum Zeitpunkt des Zugriffs außerhalb der Klasse befindet, und wegen `Private` ist von diesem Punkt aus kein Herankommen an die Eigenschaft zum Schreiben möglich.

Doch wozu brauchen Sie unterschiedliche Zugriffsmodifizierer in Eigenschaften während der Entwicklung Ihrer Software? Denken Sie beispielsweise an die Zurverfügungstellung von Assemblys (Klassenbibliotheken) für andere Entwickler: Sie möchten beispielsweise in der Lage sein, bestimmte Eigenschaften Ihrer Klassen von jedem Punkt Ihrer Assembly aus zu manipulieren; Sie möchten aber gleichzeitig verhindern, dass ein Entwickler, der Ihre Assembly verwendet, dazu auch in der Lage ist. In diesem Fall definieren Sie den `Get`-Accessor als `Public` – das Lesen der Eigenschaft stellt schließlich kein Risiko dar und kann von überall aus erfolgen –, aber den `Set`-Accessor als `Friend`. Vom gesamten Programmcode, der sich in Ihrer Klassenbibliothek befindet, können Sie dann die Eigenschaft der betreffenden Klasse manipulieren; Entwickler, die die Assembly einbinden, also von außerhalb Ihrer Assembly zugreifen, können diese aber nicht mehr direkt manipulieren. Solcherlei Eigenschaften sind dann wichtig, wenn es sich bei ihnen um Quasikonstanten handeln soll. Ihre Assembly definiert die Eigenschaft wann und wie sie will auf Grund bestimmter Zustände. Die Assemblys, die sie konsumieren, müssen aber mit dieser Einstellung leben. Klassen, die beispielsweise Einstellungen aus der Windows-Registry widerspiegeln, können davon Gebrauch machen. Denkbar wäre auch, eine Verbindungszeichenfolge zu einem SQL Server auf diese Weise freizulegen – Sie hätten zwar aus Ihrer Assembly heraus Manipulationsspielraum für die Verbindungszeichenfolge; eine Assembly könnte aber immer nur die Verbindungszeichenfolge mit der Eigenschaft auslesen, die Sie innerhalb Ihrer Assembly vorgeben.

Statische Elemente

Bislang haben Sie im Rahmen von Klassen nur Feldvariablen (Felder), Membereigenschaften und Membermethoden kennen gelernt, also solche Methoden, die zum sauberen Funktionieren auf die Instanzvariablen zugreifen mussten, wofür sie in jedem Fall eine Instanz der Klasse benötigten – anderenfalls hätten Sie eine `NullReferenceException` ausgelöst.

Sie haben aber auch die Möglichkeit, statische Komponenten zu erstellen, die eben nicht auf Member einer Klasse zurückgreifen und damit auch ohne Instanziierung aufrufbar sind. Die folgenden Abschnitte zeigen, wie es geht.

Nicht durch die Schlüsselwörter Static und Shared in Visual Basic verwirren lassen!

`Static` im Gegensatz zu `Shared` bewirkt, dass eine Variable, die nur innerhalb einer Prozedur (`Sub` oder `Function`) verwendet wird, ihren Inhalt auch nach Verlassen der Unterroutine nicht verliert. Dennoch können Sie auf diese Variable nur innerhalb der Unterroutine zugreifen, in der sie definiert wird. Im Prinzip ist eine mit `Static` definierte Variable eine, die als `Shared`-Member für die Klasse definiert und mit einem internen Attribut versehen wurde, das die Verwendung auf den Gültigkeitsbereich reglementiert, in dem sie deklariert wurde.

Vorhanden sind `Static`-Variablen übrigens nur aus Gründen der Kompatibilität zum alten Visual Basic 6.0 (und vorherigen Versionen). Und um die Verwirrung komplett zu machen: In C# ist `static` das, was in .NET-Versionen von Visual Basic `Shared` ist – alles klar?

Statische Methoden

Statische Methoden sind Methoden (`Sub`, `Function`) die mit dem Schlüsselwort `Shared` (nicht `Static`! – siehe vorheriger Kasten!) gekennzeichnet und damit direkt über die Klasse und nicht über die Klasseninstanz definiert sind. Das bedeutet: Anders als bei Instanzmethoden können Sie eine statische Methode direkt über den Klassennamen aufrufen und brauchen eben keine Instanz dafür.

Ein gutes Beispiel für eine statische Methode ist die `Parse`-Methode aller numerischen Datentypen. Möchten Sie beispielsweise eine Zeichenfolge in einen `Integer`-Wert konvertieren, bedienen Sie sich genau dieser Funktion:

```
Dim intVar as Integer = Integer.Parse("1234")
```

Hier sehen Sie, dass Sie für den Funktionsaufruf nicht auf eine Instanzmethode zurückgreifen, da keine Feldvariablen der Instanz für den Konvertierungsprozess benötigt werden. Im umgekehrten Fall ist das anders. Möchten Sie einen `Integer`-Wert in eine Zeichenkette umwandeln, verwenden Sie die Instanzmethode `ToString`, die natürlich auf die interne Feldvariable des `Integer`-Datentyps zurückgreift:

```
Dim intVar as Integer = 5
Debug.Print(intVar.ToString())
```

HINWEIS Im Grunde genommen müsste die `EllipseString`-Methode unseres Klassenbeispiels ebenfalls statisch und damit mit `Shared` definiert werden, da sie ebenfalls nur einen Algorithmus kapselt, der nicht auf eine Feldvariable der Klasse direkt zugreift. Konsequenterweise sollte diese Methode dann auch als öffentlich definiert werden, sodass ihre korrekte Signatur folgendermaßen aussieht:

```
    Public Shared Function EllipseString(ByVal text As String, ByVal MaxLength As Integer) As String
    .
    .
    .
    End Function
```

Die richtige Funktionsweise dieser Routine wäre dann:

```
Dim s as String = Kontakt.EllipseString("Dies ist die abzuschneidende Zeichenkette", 20)
```

Sollten Sie eine Objektvariable auf eine instanziierte `Kontakt`-Instanz haben, könnten Sie rein theoretisch auch über diese Objektvariable auf die statische Methode zugreifen. Doch Sie sollten das nicht machen, da es für Verwirrung sorgt. Die Verwendung von Objektvariablen impliziert immer, dass Sie etwas mit der Klasseninstanz (also dem Dateninhalt der Klasse) anstellen wollen, was Sie in diesem Fall gar nicht machen, da, wie gerade gesagt, statische Methoden keine Feldvariablen nutzen (und auch gar nicht nutzen können!). Dieser Meinung ist auch der Hintergrundcompiler von Visual Basic, denn einen Versuch, eine statische Methode über eine Objektvariable aufzurufen, quittiert er, wie in Abbildung 10.11 zu sehen, mit einer Warnung.

Wie in Abbildung 10.11 zu sehen, gibt es – vielleicht kennen Sie das schon von Microsoft Word – im Visual Studio-Editor nach einer Bearbeitung eines Objekts bzw. Objektnamens so genannte Smarttags; so Sie sich Kapitel 5 zu Gemüte geführt haben, wissen Sie darüber längst Bescheid. Der folgende graue Kasten fasst den Umgang mit Smarttags nochmals kurz zusammen.

Abbildung 10.11 Einen Versuch, eine statische Methode über eine Objektvariable aufzurufen, quittiert der Hintergrundcompiler von Visual Basic mit einer Warnung. Die Fehlerkorrekturoptionen helfen Ihnen übrigens beim Berichtigen.

Smarttags im Editor von Visual Basic

Einen Smarttag erkennen Sie zunächst als kleinen, hellen Unterstrich in einer Codezeile im Visual Basic-Editor, an der es seiner Meinung nach irgendetwas zu verbessern oder anzumerken gibt.

Bewegen Sie den Mauszeiger auf diesen Unterstrich, verwandelt er sich in ein kleines Symbol mit der Form eines Ausrufezeichens, das Ihnen verschiedene Arten von Unterstützung anbietet (welche, kommt ganz auf den Zusammenhang an). In vielen Fällen verbirgt sich hinter dem Smarttag ein Dialog, der Ihnen einen Korrekturvorschlag für den erkannten »Fehler« unterbreitet. Abbildung 10.12 verdeutlicht den Zusammenhang.

Abbildung 10.12 So verwenden Sie Smarttags im Codeeditor von Visual Basic 2010

Statische Eigenschaften

Visual Basic sieht auch statische Eigenschaften vor. Genau wie bei statischen Funktionen sind statische Eigenschaften direkt über die Klasse und nicht über die Klasseninstanz definiert. Das bedeutet, dass statische Eigenschaften Funktionalitäten bereitstellen sollten, die für alle Objekte dieser Klasse gelten und nicht für eine bestimmte Objektinstanz.

Das beste Beispiel für eine statische Eigenschaft ist die `Now`-Eigenschaft von `DateTime`. Sie liefert die aktuelle Uhrzeit zurück und ist natürlich von den Eigenschaften einzelner `DateTime`-Instanzen völlig unabhängig. Statische Eigenschaften werden, ebenfalls wie statische Funktionen, mit dem `Shared`-Schlüsselwort deklariert.

HINWEIS Wie bei normalen Eigenschaften können Sie auch bei statischen Eigenschaften automatisch implementierte Eigenschaften verwenden. In diesem Fall werden die entsprechenden Hilfsfelder (*Backing Fields*), also die Feldvariablen, die daraus entstehen, ebenfalls statisch angelegt. Auch diese heißen wieder wie die Eigenschaft selbst und haben nur einen Unterstrich davor stehen. IntelliSense listet auch diese Variablen nicht auf, was aber nicht bedeutet, dass Sie diese nicht verwenden können.

Über mehrere Codedateien aufgeteilter Klassencode – das Partial-Schlüsselwort

Sie können die Definition einer Klasse oder Struktur mithilfe des `Partial`-Schlüsselworts auf mehrere Definitionsabschnitte aufteilen. Das bedeutet, dass sich beliebig viele Teildefinitionen einer Klasse in beliebig vielen unterschiedlichen Quelldateien unterbringen lassen. Alle Definitionen müssen sich jedoch in der gleichen Assembly und dem gleichen Namespace befinden.

HINWEIS Visual Basic verwendet partielle Klassendefinitionen, um in jeweils eigenen Quelldateien generierten Code von dem Code zu trennen, der vom Benutzer erstellt wurde. Zum Beispiel definiert der Windows Form-Designer partielle Klassen für Steuerelemente, zum Beispiel `Form`. Sie sollten den generierten Code in diesen Codedateien nicht ändern.

Normalerweise wird die Entwicklung einer einzelnen Klasse oder Struktur nicht auf zwei oder mehr Codeklassen verteilt. In der Regel benötigen Sie das `Partial`-Schlüsselwort daher nicht. Sinn ergibt die Anwendung von `Partial` aber dann, wenn der Code einer Klasse dermaßen anwächst, dass die Übersicht sehr darunter leiden würde. Eine gute Vorgehensweise besteht dann darin, innerhalb der Projektmappe für die mit `Partial` aufgeteilte Klasse einen zusätzlichen Ordner anzulegen, und die einzelnen Codedateien der Klasse dort abzulegen.

Beim Erstellen einer partiellen Klasse oder Struktur gelten übrigens alle Regeln für die Erstellung von Klassen und Strukturen, beispielsweise diejenigen für die Verwendung von Zugriffsmodifizierern und die Vererbung. Übrigens: Nur die zusätzlichen Codedateien benötigen das `Partial`-Schlüsselwort, um den Klassencode zu erweitern – beim Ausgangsklassencode *kann* man es angeben, muss dies aber nicht tun.

Partieller Klassencode bei Methoden und Eigenschaften

Was für die Klasse selbst gilt, hat auch für Methoden- und Eigenschaftendefinitionen Gültigkeit, nur in leicht abgeänderter Weise. Partieller Klassencode dient insbesondere dazu, automatisch generierten Code in Verbindung mit Code, der diesen modifiziert, besser pflegen zu können.

Angenommen, Sie haben aus einem Entitätsmodell des Entity Framework für den Zugriff auf eine Datenbank über ein Objektmodell Code generieren lassen und müssen diesen aus irgendwelchen Gründen erweitern. Ändern Sie anschließend das Entitätsmodell und lassen diesen Code dann neu erstellen, wären Ihre Änderungen verloren.

```
'Auszug aus einer per Codegenerator erstellten Datei für ein Objektmodell eines Entity Data Modells:

    '''<summary>
    '''Initialisiert ein neues AWEntities-Objekt.
    '''</summary>
    Public Sub New(ByVal connection As Global.System.Data.EntityClient.EntityConnection)
        MyBase.New(connection, "AWEntities")

        'Hier wird eine Methode der generierten Klasse aufgerufen ...
        Me.OnContextCreated
    End Sub

    'Die aber hier nur durch Partial ihren Methodenrumpf definiert.
    'Der eigentliche Code kann in einer separaten Codedatei liegen,
    'der beim Neuerstellen dieses Codes nicht mehr überschrieben werden kann.
    Partial Private Sub OnContextCreated()
    End Sub
```

Aus diesem Grund gibt es auch auf Methodenebene die Möglichkeit, mit `Partial` »leere« Methodenrümpfe zu definieren, den eigentlichen Code, der diese Methoden »ausfüllt«, aber dann in einer anderen Codedatei unterzubringen. Das Ergebnis: Sie können den generierten Code nun immer wieder neu erstellen, Ihr manuell editierter Code bleibt stets in einer separaten Codedatei erhalten.

Kapitel 11

Vererben von Klassen und Polymorphie

In diesem Kapitel:

Wiederverwendbarkeit von Klassen durch Vererbung (Inheritance) 374
Überschreiben von Methoden und Eigenschaften 386
Polymorphie 389
Abstrakte Klassen und virtuelle Prozeduren 409
Schnittstellen (Interfaces) 412
Die Methoden und Eigenschaften von Object 427
Shadowing (Überschatten) von Klassenprozeduren 432
Sonderform »Modul« in Visual Basic 437
Singleton-Klassen und Klassen, die sich selbst instanziieren 438

Wiederverwendbarkeit von Klassen durch Vererbung (Inheritance)

Vererbung und Wiederverwendbarkeit im Sinne als Erweiterung von Klassen sind zentrale Programmiertechniken, von denen .NET Framework sehr regen Gebrauch macht. Man könnte sogar so weit gehen zu sagen, dass ohne die Möglichkeit, Klassen zu vererben, .NET keinen Sinn ergeben würde.

Klassenvererbung wirklich verstanden zu haben und damit später Objektmodelle für eigene Problemlösungen bauen zu können, ist eine unschätzbar wichtige Fähigkeit – wenn nicht sogar die wichtigste, die Sie sich überhaupt aneignen können, um als Entwickler konkurrenzfähig zu sein und um Anwendungen entwickeln zu können, die wirklich zuverlässig, gut wartbar und von langer Lebensdauer – da erweiterbar – sind.

Das wirkliche Verstehen und Anwendung der Klassenvererbung gehört aber gerade auch für Programmiereinsteiger zu den schwierigsten Themen, bei denen man darauf wartet, dass es »endlich Klick macht«, und wenn es das erst einmal gemacht hat, dann tun sich auf den Schlag neue Türen auf: Sie werden keinen OOP-Entwickler finden, der diese Techniken wirklich beherrscht und sich nicht gleichzeitig mit einem Grinsen wie ein Honigkuchenpferd an diesen Moment erinnern könnte.

Dabei kennen wir das Konzept, Objekte zu vererben, doch zu gut aus dem täglichen Leben: So wissen Sie, wenn Sie einen Hund sehen, mit sehr großer Wahrscheinlichkeit, dass es sich um einen Hund handelt, selbst wenn Sie diese Hunderasse noch nie in Ihrem Leben zuvor gesehen haben. Aber Sie haben ein Bild von einem »Basishund« vor Augen und können seine Ableitung (Pudel, Dackel, Schäferhund) erkennen, weil Sie sie auf eben diese Basis zurückführen können. Ähnlich ist es beispielsweise mit Behältern. Wetten, Sie können sich keinen Behälter vorstellen? Sie können sich lediglich eine *Ableitung* (also die Vererbung) eines *bestimmten* Behälter*typs* vorstellen, also zum Beispiel einen Eimer oder eine Flasche. Sich nur den allgemein gültigen Behälter schlechthin vorzustellen, ist aber nicht möglich. Zwar ist es richtig, dass eine Flasche ein Behälter ist. Aber ein Eimer und eine Badewanne sind genauso gut Behälter, und eine Flasche ist sicherlich keine Badewanne. Es sind eben nur eine Reihe von Eigenschaften, die all diesen Objekten gemein sind und die diese Objekte aber eben als Behälter ausmachen.

Offensichtlich funktionieren wir Menschen also bereits ohnehin objektorientiert, und wenn das so ist, können wir uns dieses Verfahren in der Programmierung zunutze machen. Wenn ohnedies einige Objekte ähnliche Verhaltensweisen aufweisen, sich aber – eben wie bei einem Behälter – auf das gleiche Ausgangsschema zurückführen lassen, wieso muss man dann durch immer wieder den gleichen Code das Rad ständig neu erfinden? Angenommen, Sie könnten eine Flasche programmieren – wie auch immer das aussehen würde. Und Sie müssten anschließend ein Tetra Pak programmieren. Dann wette ich abermals mit Ihnen, dass Sie in beiden Objekten zum Beispiel Code für die Definition der Eigenschaft *Behältervolumen* unterbringen müssen. Sinnvoller und einfacher wäre es dann sicher, erst eine total abstrakte (nicht vorstellbare und damit natürlich auch nicht anwendbare) Klasse *Behälter* zu programmieren und dann diese Klasse *Behälter* mit allen Grundeigenschaften auszustatten. Zum Beispiel mit einer Eigenschaft *Volumen*. Und diese Klasse *Behälter* vererben Sie anschließend sowohl an *Flasche* als auch an *Tetra Pak*, die dann beide schon einmal die Eigenschaft *Volumen* aufweisen. [1]

---

[1] Ihr Geist ist in der Lage, Objekte wie Eimer, Fass, Tetra Pak, Glasflasche, Becher, Badewanne, Tank und was es sonst noch an Behältern gibt, zu kategorisieren – übrigens etwas, was schon Kant im Rahmen von *transzendentalen Schemata* beschreibt.

Soviel zum Prinzip der Vererbung als kleine Einführung. Eigentlich wäre es sehr schön, wenn man sagen könnte, dass Visual Basic auf eine sehr lange Historie in Sachen Vererbung zurückblicken könnte – Klassen kennt Visual Basic schließlich schon seit Visual Basic 4.0. Doch eine richtige Vererbung war bis einschließlich zur Version 6.0 Fehlanzeige, und Visual Basic beherrschte ebenso bis dahin die Polymorphie[2] – die Möglichkeit also, beispielsweise über die gleichen Methodennamen unterschiedliche Klassen anzusprechen – nur sehr unzulänglich. Bis Visual Basic 6.0 (und jetzt noch in VBA) konnten Sie Klassen nur durch die so genannte Delegation vererben: Dabei wird eine Klasse in eine andere Klasse als Feldvariable eingebunden und deren Eigenschaften durch neue Funktionen und Eigenschaftenprozeduren nach außen offen gelegt. In Visual Basic 6.0 war die richtige Polymorphie auf Schnittstellen beschränkt – mehr zu diesem Thema gibt es übrigens im Abschnitt »Schnittstellen (Interfaces)« ab Seite 412.

Bevor wir uns die Technik der Vererbung für unser Standardbeispiel zunutze machen, möchte ich Ihnen anhand einfacher Codebeispiele zunächst die Syntax für den Vorgang des Vererbens erklären. Unser bisher verwendetes Langzeitbeispiel möchte ich dazu zunächst nicht verwenden, da es bereits einer Sonderbehandlung bei der Vererbung bedarf, die am Anfang verwirren würde und dem Verständnis für einen Moment im Wege stünde.

BEGLEITDATEIEN Sie finden das folgende Beispielprojekt im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 11\Vererbung01
```

Sie sollten aber vorschlagsweise die folgende Übung Schritt für Schritt nachvollziehen, die dieses Projekt aufbaut, um ein besseres Verständnis für das Drumherum zu bekommen.

- Legen Sie ein neues Projekt an, indem Sie *Neu* und *Projekt* aus dem Menü *Datei* wählen.
- Klicken Sie auf *Visual Basic-Projekte* in der Liste *Projekttypen*, wählen Sie *Konsolenanwendung* unter *Vorlagen* und geben Sie **Vererbung** als Projektnamen ein.
- Bestimmen Sie ein Speicherverzeichnis Ihrer Wahl.
- Klicken Sie auf *OK*, um das neue Projekt zu erstellen.
- Klicken Sie doppelt auf *Module1.vb* im Projektmappen-Explorer, um den Code für das Projekt anzuzeigen. Lassen Sie sich zunächst nicht durch das Schlüsselwort `Module` irritieren. Auf das Thema Module (das aus .NET Framework-Sicht betrachtet in Visual Basic wieder eine Sonderrolle spielt) werde ich im Laufe dieses Kapitels noch eingehen.
- Ändern Sie den Namen von *Module1.vb* in **mdlMain.vb**, indem Sie aus dem Kontextmenü den Befehl *Umbenennen* wählen und einen neuen Dateinamen eingeben. Denken Sie daran, die Dateinamenerweiterung *.vb* mit anzugeben!
- Klicken Sie im Projektexplorer doppelt auf *mdlMain.vb*, um das Codefenster zu öffnen.

Im Codeeditor unterhalb der Moduldefinition (also hinter *End Module*) geben Sie nun

```
Class ErsteKlasse
```

ein. Visual Basic erstellt automatisch die Zeile

[2] Etwa: »Vielgestaltigkeit«.

```
End Class
```

darunter, um den Klassencodeblock abzuschließen. Diese erste Klasse soll eine Eigenschaft für die Wertezuweisung sowie eine Methode erhalten, mit der der Inhalt der Klasse quasi ausgedruckt werden kann. Unsere Klasse kann also nichts Besonderes, sie kann in ihrer einzigen Eigenschaft gerade mal einen `Integer`-Wert speichern. Fügen Sie folgenden Code in die Klasse ein:

```
Class ErsteKlasse

    Private myEinWert As Integer

    'Eigenschaft, um den Wert verändern zu können.
    Property EinWert() As Integer

        Get
            Return myEinWert
        End Get

        Set(ByVal Value As Integer)
            myEinWert = Value
        End Set

    End Property

    'Funktion, um den Inhalt als Zeichenkette (String) zurückzuliefern.
    Function AlsZeichenkette() As String

        Return CStr(myEinWert)

    End Function

End Class
```

HINWEIS Auch wenn Sie das Konzept von automatisch implementierten Eigenschaften bereits im vorangegangenen Kapitel kennen gelernt haben und die folgenden Beispiele diese theoretisch verwenden könnten, ohne andere Verhaltensweisen zu zeigen, werden wir in den folgenden Kapiteln mitunter darauf verzichten, um ein Gefühl dafür zu bekommen, was tatsächlich passiert, und diese zusätzlichen Codezeilen für den Getter/Setter und die Verwaltung der Feldvariablen nicht den Compiler schreiben lassen.

Zur Wiederholung: Das Beispiel würde auch durchaus folgendermaßen funktionieren; die Eigenschaftenvariable hieße dann `_EinWert`, würde durch den Compiler definiert, und das Beispiel wäre lediglich ein paar Zeilen kürzer:

```
Class ErsteKlasse

    'Eigenschaft, um den Wert verändern zu können.
    Property EinWert() As Integer

    'Funktion, um den Inhalt als Zeichenkette (String) zurückzuliefern.
    Function AlsZeichenkette() As String

        Return CStr(_EinWert)
```

```
    End Function

End Class
```

Im Modul *mdlMain* erstellen Sie nun ein kleines Rahmenprogramm, das diese neue Klasse zu Demonstrationszwecken verwendet:

```
Module mdlMain

    Sub Main()
        Dim klasse1 As New ErsteKlasse
        klasse1.EinWert = 5
        Console.WriteLine(klasse1.AlsZeichenkette())

        Console.WriteLine()
        Console.WriteLine("Zum Beenden Taste drücken")
        Console.ReadKey()
    End Sub

End Module
```

Sie können das Programm nun starten, und das Ergebnis wird dem entsprechen, was Sie sicherlich erwartet haben. Für die nächsten Schritte stellen Sie sich einfach vor, dass Ihnen der Quelltext von `ErsteKlasse` nicht zur Verfügung steht. Unterhalb der Klassendefinition (die Sie natürlich im Geiste gar nicht sehen …) fügen Sie nun eine weitere Klassendefinition ein:

```
Class ZweiteKlasse
    Inherits ErsteKlasse

End Class
```

Sie haben damit eine neue Klasse namens `ZweiteKlasse` geschaffen. Diese Klasse hat keine Eigenschaften und keine Methoden, und sie hat sie doch. Wenn Sie die Einfügemarke in die erste Prozedur der Codedatei platzieren und unterhalb der Zeile

```
Console.WriteLine(klasse1.AlsZeichenkette())
```

die Zeile

```
Dim klasse2 As New ZweiteKlasse
```

einfügen, darunter den Instanznamen einfügen und den Punkt eintippen, sehen Sie, wie IntelliSense Ihnen die Elemente der neuen Klasse anbietet:

```
Module mdlMain

    Sub Main()
        Dim klasse1 As New ErsteKlasse
        klasse1.EinWert = 5
        Console.WriteLine(klasse1.AlsZeichenkette())

        Dim klasse2 As New ZweiteKlasse
        klasse2.
                  AlsZeichenkette      Public Function AlsZeichenkette() As String
                  EinWert
        Console.  Common  All
        Console.WriteLine("Zum Beenden Taste drücken")
        Console.ReadKey()
    End Sub

End Module
```

Abbildung 11.1 Obwohl es bislang keine Elementdefinitionen für `ZweiteKlasse` gibt, zeigt Ihnen IntelliSense dennoch eine Eigenschaft und eine Methode an

Was ist passiert? Durch die Anweisung

```
Inherit ErsteKlasse
```

haben Sie bestimmt, dass `ZweiteKlasse` alle Elemente von `ErsteKlasse` erbt. Alles, was `ErsteKlasse` kann, können Sie auch mit `ZweiteKlasse` machen.

WICHTIG Genau darum geht es bei der objektorientierten Programmierung: Durch die Vererbung schreiben Sie wiederverwendbaren Code. Wenn Sie vorhandene Klassen verändern wollen, um sie an ein bestimmtes Problem anzupassen, dann können Sie die Originalklasse in dem Zustand belassen, den sie hat. Sie vererben sie einfach an eine neue Klasse (man sagt auch: Sie »leiten sie ab«) und verändern dann die vererbte Klasse, um sie für Ihre neue Problemlösung anzupassen und zu erweitern.

Einer der Vorteile dieser Methodik: Wenn Sie feststellen, dass in allen Klassen, die auf der Ausgangsklasse basieren, noch derselbe Fehler existiert, dann zieht die Beseitigung des Fehlers in der Basisklasse auch die Beseitigung des Fehlers in allen abgeleiteten Klassen nach sich. Bei der Kopieren/Einfügen-Vererbung, wie sie beispielsweise in Visual Basic 6.0 oder in VBA üblich ist – »wir brauchen Code an einer bestimmten Stelle leicht abgewandelt und kopieren ihn einfach« –, müssen Sie dagegen genau wissen, wie oft Sie den Code »wiederverwendet« haben und alle Stellen finden, an denen sich derselbe Fehler befindet. Daher sind Anwendungen, die prozedural aufgebaut wurden, oft sehr fehlerhaft und teilweise kaum mehr wartbar.

Angenommen, Sie brauchen für ein bestimmtes Problem eine Klasse, die genau das kann, was `ErsteKlasse` kann, nur benötigen Sie zusätzlich eine weitere Eigenschaft, die den aktuellen Instanzwert um den Wert 10 erhöht ermittelt. In diesem Fall vererben Sie die Klasse `ErsteKlasse`, so wie Sie es im Beispiel schon kennen gelernt haben, an `ZweiteKlasse` und fügen dann dort die neue Eigenschaftenprozedur hinzu:

```
Class ZweiteKlasse
    Inherits ErsteKlasse

    ReadOnly Property Um10Mehr() as Integer

        Get
            Return EinWert + 10
        End Get
    End Property
End Class
```

Und das war es schon. Die gesamte Funktionalität, die `ErsteKlasse` hatte, hat `ZweiteKlasse` auch, und sie hat obendrein noch eine Eigenschaft mehr. Gegenprobe: Unterhalb der Zeile

```
Dim klasse2 As New ZweiteKlasse
```

fügen Sie den Text **Console.WriteLine(klasse2.** ein; sobald Sie den Punkt tippen, zeigt Ihnen IntelliSense wieder die Elemente der Klasse an, und siehe da: Alle alten Elemente und die neue Eigenschaft sind in der Liste vorhanden!

Abbildung 11.2 Die neue Eigenschaft ist jetzt zusammen mit den alten Elementen in *ZweiteKlasse* zu finden

Komplettieren Sie die Zeile,

```
Console.WriteLine(klasse2.Um10Mehr)
```

starten Sie anschließend das Programm, und schauen Sie, was passiert:

```
5
10

Zum Beenden Taste drücken
```

Haben Sie erwartet, dass 15 als zweites Ergebnis ausgegeben wird? Natürlich nicht, denn die aus `ErsteKlasse` entstandene Instanz `klasse1` ist völlig unabhängig von der Instanz `klasse2`, die aus `ZweiteKlasse` entstanden ist. Sie haben sich von `ErsteKlasse` nur den Code als Vorlage »geborgt« – die Objekte, die aus beiden Klassen entstehen können, haben natürlich beide einen eigenen und damit unterschiedlichen Datenbereich.

HINWEIS Wichtig für das Verständnis von Klassen ist, wie sie intern verwaltet werden. Der Code einer Klasse ist immer nur ein einziges Mal vorhanden – auch wenn Sie mehrere Instanzen einer Klasse anlegen. Deswegen ist es natürlich auch Unsinn zu glauben, dass eine Klasse, die zehnmal so viel Programmcode hat wie eine Klasse »X«, bei zehnfacher Instanziierung in zehn verschiedene Objekte 100 Mal so viel Speicher benötigt wie Klasse X. Sie braucht für den Programmcode immer gleich viel Speicher, unabhängig davon, wie viele Instanzen aus ihr entstehen. Der Programmcode ist natürlich auch nicht doppelt vorhanden, wenn Sie eine Klasse aus einer anderen ableiten. Nur der zusätzliche Programmcode durch das Hinzufügen oder Verändern vorhandener Elemente belegt zusätzlichen Speicher. Ausschließlich die Feldvariablen einer Klasse sind dafür ausschlaggebend, wie viel zusätzlichen Speicher eine Klasse beim Instanziieren in Objekte benötigt.

Damit das erwartete Ergebnis eintritt, müssen Sie das Programm wie folgt abändern:

```
Module mdlMain

    Sub Main()
        Dim klasse1 As New ErsteKlasse
        klasse1.EinWert = 5
        Console.WriteLine(klasse1.AlsZeichenkette())

        Dim klasse2 As New ZweiteKlasse
        klasse2.EinWert = 5
        Console.WriteLine(klasse2.Um10Mehr)
        Console.WriteLine()
        Console.WriteLine("Zum Beenden Taste drücken")
        Console.ReadKey()
    End Sub
End Module
```

Jetzt betrachten wir die Klassen mit dem IL-Disassembler und schauen, ob uns der Visual Basic-Compiler wieder irgendetwas an Arbeit abgenommen hat.

- Starten Sie den IL-Disassembler aus dem Startmenü (der Programmgruppe *Microsoft Visual Studio 2010/Microsoft Windows SDK Tools*), und öffnen Sie die Datei *Vererbung.exe*, die Sie im Verzeichnis *\bin\Debug* des Verzeichnisses finden, in dem Sie das Projekt angelegt hatten.
- Öffnen Sie per Klick auf das davor stehende Pluszeichen zunächst den Zweig `Vererbung` und dann die Unterzweige `Vererbung.ErsteKlasse` und `Vererbung.ZweiteKlasse`. Anschließend sollten Sie ein Bild vor sich sehen, etwa wie es Abbildung 11.3 zeigt.

Abbildung 11.3 Die beiden Testklassen im IL-Disassembler

Genau wie in einem der vorherigen Beispiele hat Ihnen Visual Basic wieder Arbeit abgenommen und sowohl in der Basisklasse `ErsteKlasse` als auch in der abgeleiteten Klasse `ZweiteKlasse` entsprechende Konstruktorprozeduren (`.ctor`) eingefügt.

HINWEIS In der Abbildung ebenfalls auf den ersten Blick erkennbar: Eigenschaften werden intern in Funktionen bzw. Methoden umgewandelt. Aus der `EinWert`-Eigenschaft der Basisklasse, die jeweils einen *Get*- und einen *Set*-Accessor hat, macht der Visual Basic-Compiler die beiden Funktionen `get_EinWert` und `set_EinWert`. Die Eigenschaft selbst wird darunter definiert, und der IML-Code in ihr bestimmt lediglich, welche der in der Klasse vorhandenen Funktionen die Eigenschaft auflösen (per Doppelklick auf den Eigenschaftennamen können Sie sich den folgenden Code anzeigen lassen).

```
.property instance int32 EinWert()
{
  .set instance void Vererbung.ErsteKlasse::set_EinWert(int32)
  .get instance int32 Vererbung.ErsteKlasse::get_EinWert()
} // end of property ErsteKlasse::EinWert
```

Was noch auffällt: Genau wie vermutet, sind in `ZweiteKlasse` nur die zusätzlich vorhandenen Elemente als Code vorhanden. Dass die Klasse von der Basisklasse erbt und damit auch deren Elemente mitnutzen kann, wird – wie in Abbildung 11.3 gezeigt – durch `extends Vererbung.ErsteKlasse` geregelt. Da die einzige Eigenschaft nur einen *Get*-Accessor hat, gibt es übrigens auch nur eine Eigenschaftsfunktion in Form von `get_Um10Mehr`.

TIPP Was die Deklaration von Feldvariablen anbelangt: Nur Feldvariablen, die Sie später auch in vererbten Klassen direkt manipulieren wollen, deklarieren Sie als `Protected`. Sie machen das aber nur in Ausnahmefällen. Auch bei Basisklassen gilt das Prinzip: Möglichst viel kapseln und schützen: Feldvariablen sollten also nach Möglichkeit als `private` definiert sein, NIEMALS aber als `public`.[3]

Nachdem nun bekannt ist, dass abgeleitete Klassen automatisch den Standardkonstruktor der Basisklasse aufrufen, wäre es interessant zu erfahren, was passiert, wenn die Basisklasse keinen Standardkonstruktor hat. Einen solchen Fall haben Sie mit der letzten Version der Klasse aus dem `Kontakt`-Beispiel nämlich kennen gelernt. Eine Klasse hat dann keinen automatisch generierten Standardkonstruktor, wenn sie einen parametrisierten Konstruktor aufweist.

Was passiert also, wenn wir der Klasse `ErsteKlasse` einen Konstruktor hinzufügen, der einen Parameter (zum Beispiel den Initialisierungswert für `myEinWert`) übernimmt? Probieren Sie es aus: Fügen Sie die Zeile

```
    Sub New(ByVal Initialwert As Integer)
```

in den Klassencode von `ErsteKlasse` ein. Sobald Sie die Änderungen eingefügt haben, sehen Sie diverse Fehlermarkierungen im Quelltext und die entsprechenden Beschreibungen dazu in der Fehlerliste, etwa wie in Abbildung 11.4 zu sehen.

[3] Ich habe keinen blassen Schimmer, wieso dieser Tipp in vorherigen Auflagen des Buchs nicht schon so wie an dieser Stelle formuliert war. Vielleicht habe ich im Laufe der Jahre auch einfach meine Meinung geändert und dazugelernt. Wie sagte unser erster Bundenkanzler Adenauer doch so schön: »Was kümmert mich…« ;-)

Abbildung 11.4 Nach dem Einfügen eines parametrisierten Konstruktors sehen Sie gleich zwei Fehler in der Fehlerliste

Der Grund: Im Modul innerhalb von `Sub Main` kann `klasse1` nicht mehr instanziiert werden, denn hinter

```
Dim klasse1 As New ErsteKlasse
```

steht kein Parameter. Da wir einen parametrisierten Konstruktor in die Klassendefinition eingefügt haben, gibt es keinen Standardkonstruktor mehr. Sobald Sie selbst irgendeinen Konstruktor (parametrisiert oder nicht) in eine Klasse einfügen, hört Visual Basic auf, Ihnen mit dem »Hineinkompilieren« eines Standardkonstruktors unter die Arme zu greifen. Kein Standardkonstruktor heißt: nehmen, was da ist.

Daher können wir noch einmal festhalten: Wenn Sie keinen Konstruktor (`Sub New…`) explizit implementieren, schreibt der Compiler einen solchen für Sie implizit (automatisch). Wenn Sie jedoch einmal anfangen, Konstruktoren explizit zu implementieren, müssen Sie alle Konstruktoren, die Sie wünschen (auch den Standardkonstruktor ohne Parameter) ausdrücklich implementieren.

Und da existiert nur einer, dem Parameter übergeben werden – der erste Fehler ist also erst erledigt, wenn Sie den Initialisierungswert für die Klasse mit angeben, etwa so:

```
Dim klasse1 As New ErsteKlasse(5)
Console.WriteLine(klasse1.AlsZeichenkette())
.
.
.
```

Kein Standardkonstruktor bedeutet aber auch: Die abgeleitete Klasse weiß nicht, wie sie die Basisklasse instanziieren soll (was auf jeden Fall passieren muss). In diesem Fall müssen Sie also selbst dafür Sorge tragen, dass die Basisklasse aufgerufen wird. Sie erreichen das, indem Sie den Konstruktor der Basisklasse mit dem `MyBase`-Schlüsselwort aufrufen.

Das folgende Listing stellt die funktionierende Version dar (Änderungen sind fett hervorgehoben; einige unwichtige Teile sind durch »…« abgekürzt):

```
Module mdlMain

    Sub Main()
        Dim klasse1 As New ErsteKlasse(5)
        Console.WriteLine(klasse1.AlsZeichenkette())

        Dim klasse2 As New ZweiteKlasse(5)
        Console.WriteLine(klasse2.Um10Mehr)

        Console.WriteLine()
        Console.WriteLine("Zum Beenden Taste drücken")
        Console.ReadLine()
    End Sub

End Module

Class ErsteKlasse

    Protected myEinWert As Integer

    Sub New(ByVal NeuerWert As Integer)
        myEinWert = NeuerWert
    End Sub

    'Eigenschaft, um den Wert verändern zu können.
    Property EinWert() As Integer
    ...
    End Property
    'Funktion, um den Inhalt als Zeichenkette (String) zurückzuliefern.
    Function AlsZeichenkette() As String

        Return CStr(myEinWert)

    End Function
End Class

Class ZweiteKlasse
    Inherits ErsteKlasse

    Sub New(ByVal NeuerWert As Integer)
        MyBase.New(NeuerWert)
    End Sub

    ReadOnly Property Um10Mehr() As Integer
    ...
    End Property

End Class
```

Initialisierung von Feldvariablen bei Klassen ohne Standardkonstruktoren

Interessant ist zu sehen, was passiert, wenn Sie Feldvariablen schon bei ihrer Deklarierung definieren, etwa wie im folgenden Beispiel:

```
Class ErsteKlasse

    Protected myEinWert As Integer = 9

    Sub New()
        'Hat keinen Sinn, füllt aber den Konstruktor mit Code.
        Console.WriteLine("Testausgabe: Parameterloser Konstruktor")
    End Sub

    Sub New(ByVal AuszugebenderText As String)
        'Hat keinen Sinn, füllt aber den Konstruktor mit Code.
        Console.WriteLine("Testausgabe: " & AuszugebenderText)
    End Sub
.
.
.
```

In diesem Beispiel gibt es einen Standardkonstruktor und zusätzlich zwei parametrisierte. Da kein Code »außerhalb« einer Klasse ausgeführt werden kann, stellt sich die Frage: Wo findet die Zuweisung

```
    Protected myEinWert As Integer = 9
```

denn eigentlich statt? Die Antwort offenbart wieder ein Blick in den IML-Code der Klasse. Der Compiler treibt hier den größten Aufwand, denn er muss den Code für das Initialisieren der Feldvariablen in jeden Konstruktor einbauen (Abbildung 11.5). Nur so ist gewährleistet, dass alle erforderlichen Variableninitialisierungen *in jedem Fall* durchgeführt werden.

Abbildung 11.5 Initialisierungen von Feldvariablen werden im Bedarfsfall in jede Konstruktorprozedur eingebaut

TIPP Deswegen ist das auch die richtige Vorgehensweise bzw. der richtige Platz, Feldvariablen einer Klasse mit Standardwerten zu initialisieren: Auf diese Weise vergessen Sie nämlich niemals, die Definitionen in *jedem* der vorhandenen Konstruktoren durchzuführen, da der Compiler dafür sorgt, dass die Zuweisungen der Standardwerte an die Feldvariablen in wirklich *jedem* der vorhandenen Konstruktoren vorhanden sind.

Überschreiben von Methoden und Eigenschaften

Vielleicht finden Sie inzwischen ja auch, dass das Ableiten von Klassen eine ziemlich geniale Sache ist. Es strukturiert Ihre Programme, macht sie leichter lesbar und hilft Ihnen insbesondere beim sauberen Aufbau größerer und komplexer Projekte. Doch Klassen wären nicht evolutionär, und zwar im wahrsten Sinne des Wortes, wenn es nicht die Möglichkeit gäbe, bestimmte Funktionen einer Basisklasse durch andere in der abgeleiteten Klasse auszutauschen. Klassen, die andere Klassen einbinden, werden dadurch unglaublich flexibel.

Das Ersetzen von Methoden oder Eigenschaften durch andere in einer abgeleiteten Basisklasse erfordert allerdings, dass die Basisklasse der Klasse, die sie ableitet, auch gestattet, bestimmte Funktionen zu überschreiben. Funktionen bzw. Methoden oder Eigenschaften müssen von Ihnen also explizit gekennzeichnet werden, damit sie später überschrieben werden dürfen. In Visual Basic geschieht die Kennzeichnung einer Prozedur einer Basisklasse zum Überschreiben mit dem Schlüsselwort `Overridable` (überschreibbar). Alle Funktionen und Eigenschaften, die mindestens als `Protected` gekennzeichnet sind, können den `Overridable`-Modifizierer tragen. Die Eigenschaften bzw. Methoden, die dann anschließend eine Funktion in der Basisklasse überschreiben, müssen wiederum mit dem Schlüsselwort `Overrides` (überschreibt) gekennzeichnet sein.

HINWEIS Es ist übrigens wichtig, dass Sie Überschreiben und Überladen nicht in einen Topf werfen, kräftig darin herumrühren und schauen, was anschließend dabei herauskommt: Visual Basic erlaubt es nämlich, dass eine abgeleitete Klasse die Prozedur einer Basisklasse überlädt. In diesem Fall ersetzen Sie die Basisfunktion nicht, Sie ergänzen diese nur um eine weitere Überladung. Das heißt im Klartext: Wenn Sie Prozeduren einer Basisklasse wirklich überschreiben (sie also ersetzen) wollen, dann grundsätzlich nur mit der gleichen Signatur wie die der Basisklasse.

WICHTIG In diesem Zusammenhang eine vielleicht nicht unwichtige Ergänzung: Während Sie beim Überladen von Funktionen innerhalb einer Klasse, die Sie komplett neu implementieren, das `Overloads`-Schlüsselwort auch weglassen und nur mit doppelten Methodennamen (aber unterschiedlichen Signaturen!) auskommen können, *müssen* Sie `Overloads` unbedingt verwenden, wenn Sie eine Methode einer Basisklasse um eine weitere »Überladungsversion« in einer abgeleiteten Klasse ergänzen!

Ein kleines Beispiel zu überschriebenen Methoden: Angenommen, in der abgeleiteten Klasse `ZweiteKlasse` passt Ihnen die Art und Weise nicht, wie der String durch die Funktion `AlsZeichenkette` den Wert der Variablen als Zeichenkette zurückliefert. Sie möchten beispielsweise, dass die `AlsZeichenkette`-Funktion den Wert im Stil »der Wert lautet: xxx« ausgibt. In diesem Fall würden Sie die Klassen wie folgt verändern:

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 11\Vererbung02
```

```
Module mdlMain

    Sub Main()
        Dim klasse1 As New ErsteKlasse(5)
        Console.WriteLine("Klasse1 ergibt durch AlsZeichenkette: " & klasse1.AlsZeichenkette())

        Dim klasse2 As New ZweiteKlasse(5)
        Console.WriteLine("Klasse2 ergibt durch AlsZeichenkette: " & klasse2.AlsZeichenkette())
```

```
        Console.WriteLine()
        Console.WriteLine("Return drücken, zum Beenden")
        Console.ReadLine()
    End Sub

End Module

Class ErsteKlasse

    Private myEinWert As Integer

    Sub New(ByVal NeuerWert As Integer)
        myEinWert = NeuerWert
    End Sub
    'Eigenschaft, um den Wert verändern zu können.
    Property EinWert() As Integer
    .
    .
    .
    End Property

    'Funktion, um den Inhalt als Zeichenkette (String) zurückzuliefern.
    Public Overridable Function AlsZeichenkette() As String

        Return CStr(myEinWert)

    End Function

End Class

Class ZweiteKlasse
    Inherits ErsteKlasse

    Sub New(ByVal NeuerWert As Integer)
        MyBase.New(NeuerWert)
    End Sub

    ReadOnly Property Um10Mehr() As Integer
    .
    .
    .
    End Property

    Public Overrides Function AlsZeichenkette() As String
        Return "der Wert lautet: " & MyBase.AlsZeichenkette()
    End Function

End Class
```

Wenn Sie diese veränderte Version des Programms laufen lassen, sehen Sie das folgende Ergebnis auf dem Bildschirm:

```
Klasse1 ergibt durch AlsZeichenkette: 5
Klasse2 ergibt durch AlsZeichenkette: der Wert lautet: 5

Zum Beenden Taste drücken
```

Überschreiben vorhandener Methoden und Eigenschaften von .NET Framework-Klassen

Nun ist der Name unserer Beispielfunktion nicht sonderlich glücklich gewählt – `AlsZeichenkette` ist bestenfalls ein deutsches Ausdrucksfragment. Um die Umwandlung in eine Zeichenkette beispielsweise in korrektem Englisch auszudrücken, um sich der Sprache von .NET Framework selbst anzunähern, müsste es `ToString` heißen.

Wenn Sie allerdings die vorhandene Funktion `AlsZeichenkette` in der ersten Klasse in `ToString` ändern, passiert etwas, was nicht unbedingt vorhersagbar ist (Abbildung 11.6).

```
'Funktion, um den Inhalt als Zeichenkette (String) zurückzuliefern.
Public Overridable Function ToString() As String

    Return CStr(myEinWert)

End Function
```

Fehlerliste

3 Fehler | 1 Warnung | 0 Meldungen

	Beschreibung	Datei	Zeile	Spalte	Projekt
1	"AlsZeichenkette" ist kein Member von "Vererbung.ErsteKlasse".	mdlMain.vb	6	70	Vererbung
2	function "ToString" führt Shadowing für eine überschreibbare Methode in Basis-class "Object" durch. Wenn Sie die Basismethode überschreiben möchten, muss die Methode als "Overrides" deklariert werden.	mdlMain.vb	40	33	Vererbung
3	function "AlsZeichenkette" kann nicht als "Overrides" deklariert werden, da es function in Basis class nicht überschreibt.	mdlMain.vb	63	31	Vererbung
4	"AlsZeichenkette" ist kein Member von "Vererbung.ErsteKlasse".	mdlMain.vb	64	38	Vererbung

Fehlerliste | Ausgabe | Anstehende Eincheckvorgänge

Abbildung 11.6 Versuchen Sie, *ToString* zu implementieren, sehen Sie diese Fehlermeldung

Nur, welche Methode ist hier gemeint? IntelliSense listet die vorhandenen Member eines Objekts doch auf, wie in Abbildung 11.2 (Seite 379) zu sehen, war eine Methode namens `ToString` nicht dabei. Doch wir haben uns in der verwendeten Einstellung bisher auch nicht alle Member anzeigen lassen. IntelliSense erlaubt es nämlich, die weniger wichtigen Methoden in der Vervollständigungsliste auszublenden. Erst wenn Sie in der Liste auf *Alle* umschalten, werden auch wirklich alle Member angezeigt, so auch `ToString`, wie in Abbildung 11.7 zu sehen.

```
Module mdlMain

    Sub Main()

        Dim klasse1 As New ErsteKlasse(5)
        Console.WriteLine("Klasse1 ergibt durch ToString: " & klasse1.ToString())
        klasse1.
                AlsZeichenkette
                EinWert
        Dim klas Equals                 sse
        klasse2  GetHashCode            sse(5)
                 GetType
        Console. ReferenceEquals        se2 ergibt durch ToString: " & klasse2.ToString())
                 ToString               Public Overrides Function ToString() As String
        Console. Common  All
        Console.                        Beenden Taste drücken")
        Console.ReadKey()
    End Sub

End Module
```

Abbildung 11.7 Jetzt sehen Sie alle Member der *ErsteKlasse*-Instanz

IntelliSense ist Ihnen an dieser Stelle von besonderem Nutzen, weil Sie die Modifizierer der `ToString`-Funktion in der ToolTip-Beschreibung auch gleich sehen können. Diese verraten Ihnen, dass die Funktion als `Overridable` deklariert wurde – Sie haben also selbst die Möglichkeit, die Ableitung dieser Klasse in `ErsteKlasse` mit `Overrides` zu überschreiben.

BEGLEITDATEIEN Die auf diese Weise modifizierte Version des Beispiels finden Sie im Verzeichnis:

`...\VB 2010 Entwicklerbuch\Kapitel 11\Vererbung03`

Polymorphie

Sie haben nun viel über Vererbung und über das Überschreiben von Klassenmembern erfahren – jetzt lautet die große Frage, wie Ihnen diese Techniken von Nutzen sein können. Damit Vererbung und Funktionsüberschreibung richtig Sinn ergeben, braucht es eine Technik, die sich *Polymorphie*[4] nennt. Und würden Sie mich bitten, Polymorphie in einem Satz zu erklären, dann würde ich Sie zunächst bitten, sich festzuhalten und Ihnen anschließend Folgendes zur Antwort geben: »Polymorphie in der OOP beschreibt die Möglichkeit, Methoden oder Eigenschaften in den Klassen einer Klassenerbfolge auszutauschen und die gleichnamigen, aber dennoch funktionell unterschiedlichen Methoden und Eigenschaften der veränderten Klassen der Erbfolge über *dieselbe* Objektvariable vom Typ der Basisklasse in Abhängigkeit von der tatsächlich instanziierten abgeleiteten Klasse erreichen zu können.« Das klingt sehr abstrakt, und das ist es auch.

Ich habe lange darüber nachgedacht, ein halbwegs plausibles Beispiel für Polymorphie im täglichen Leben zu finden, um zunächst einmal nur das abstrakte Konzept ein wenig zu vereinfachen. Ich gebe zu, dass ich zunächst auch nach stundenlangem Grübeln entnervt aufgab. Doch dann kam mir mein damaliger Arbeitskollege Jürgen mit einem Zufall zu Hilfe, den ich in meinem Leben so bislang noch nicht erlebt hatte.

[4] Etwa »vielgestaltig«, »in verschiedenen Formen auftretend«.

Jürgen stand an der Kasse unseres örtlichen Lebensmittelhändlers und war gerade dabei, seine Einkäufe zu bezahlen, als sein Handy schellte, was die freundliche Kassiererin allerdings nicht mitbekam. Da er sein Handy zwischen Schulter und Ohr einklemmte, *sah* die Kassiererin auch nicht, dass er telefonierte. Und wie es der Zufall wollte, wählte Jürgen, der im Grunde genommen nun mit seiner Freundin sprach und – eben weil er an der Kasse stand – vergleichsweise kurz angebunden war, seine Worte so, dass sie zufällig, aber perfekt auf zwei Gespräche passten, die er dann (eines freiwillig mit seiner Freundin, eines unfreiwillig mit der Kassiererin) auch führte. Die Worte passten sogar *so* perfekt zu dem, was er zur Kassiererin hätte sagen können, dass sich zwischen ihm und der Kassiererin ein regelrechtes Kurzgespräch entwickelte. Diese Schlagabtausche wiederholten sich einige Male, bis beiden das Missverständnis auffiel, da er die Aussage der Kassiererin »das macht dann also 21,45 Euro« mit der schönen Floskel »ich dich auch« parierte.

Jürgen führte also zwei komplett unabhängige Gespräche mit einem jeweils absolut anderen Inhalt; Jürgen verwendete aber die exakt gleiche Methode zur »Erledigung seines Anliegens«. Seine Methode, also die Wahl seiner Worte, war an dieser Stelle also absolut vielgestaltig – eben polymorph –, obwohl die Worte in beiden Gesprächen dieselben waren.

Doch zurück zur Programmierung: Was würden Sie erwarten, wenn Sie im Beispiel von *Vererbung03* die Zeile des Moduls *mdlMain* von

```
Dim klasse2 As New ZweiteKlasse(5)
```

in die Zeilen

```
Dim klasse2 As ErsteKlasse
klasse2 = New ZweiteKlasse(5)
```

abändern? Glauben Sie, dass Visual Basic einen Fehler auslöst, da Sie *klasse2* als *ErsteKlasse* deklariert, dieser Objektvariablen anschließend aber eine Instanz von zweiter Klasse zugewiesen haben? Mitnichten! Diese Vorgehensweise ist nicht nur kein Fehler, Sie haben gerade sogar das wohl mächtigste Werkzeug der objektorientierten Programmierung kennen gelernt.

Vererbung und Polymorphie – »Wie, ok!« aber: »Warum?«

In den letzten Abschnitten haben Sie schon einiges über Vererbungstechniken und Polymorphie kennen gelernt. Diese Abschnitte haben bereits einige Techniken vorgestellt, wie Sie eine Klasse dazu bekommen, sich polymorph zu verhalten. Und diese Technik anhand von Beispielen zu vermitteln, ist ja so schwierig nicht; auch nicht, die Sachverhalte zu verstehen. Aber was nützt Ihnen das? Was nützt es Ihnen, die Beispiele zu verstehen und nachvollziehen zu können, wenn aus den Erklärungen nicht wirklich hervorgeht, wie Sie Probleme erkennen können, die danach verlangen, Ihre eigenen Klassenmodelle zu entwickeln. Die größte didaktische Schwierigkeit ist also nicht, zu vermitteln, *wie* Klassenvererbung und Polymorphie funktionieren, sondern vielmehr, *warum* Klassenvererbung und Polymorphie in vielen Szenarien so wichtig sind und Sinn ergeben, da diese Techniken großes Potenzial haben, Ihre Programme stabiler zu machen und viel, viel einfacher strukturieren zu können.

Im Prinzip nähern wir uns dem wirklichen Leben beim Beschreiben von Objekten mithilfe der objektorientierten Programmierung viel mehr, als es die prozedurale Programmierung jemals könnte. Dummerweise sind aber gerade langjährige Visual Basic-Programmierer geschichtsbedingt viel prozeduraler in ihrer Denke, sodass das in vielen Fällen ihr »Normalzustand« ist; dass es »Klick« macht (und das muss es!), dauert bei Entwicklern mit langem Visual Basic 6-Hintergrund länger und ist schwieriger als bei Entwicklern, die direkt mit einer .NET-Version von Visual Basic begonnen haben. ▶

Bei der objektorientierten Programmierung gilt es nun, solche Objektzusammenhänge auch bei der Programmierung zu erkennen, übrigens auch oder sogar zum großen Teil mit dem Zweck, viel leichter im Team entwickeln zu können.

Nehmen wir zum Beispiel eine Anwendung wie Microsoft Publisher, Adobe Illustrator oder CorelDraw. Sie alle arbeiten – vereinfacht ausgedrückt – mit sehr ähnlichen Objekten, die sich alle nur in ihrer Ausgestaltung unterscheiden. Sowohl eine Linie, ein Kreis, ein Rechteck, eine Ellipse, ein Pentagon oder eine Grafik lassen sich auf einen Nenner bringen:

- Sie müssen an einer bestimmten Stelle auf dem Papier bzw. in einem Dokumentenfenster platziert werden können, haben also Ausmaße, die durch zwei Koordinaten (x1, y1, x2, y2) definiert werden
- Sie müssen sich selbst rendern können, also die Möglichkeit haben, sich in einem bestimmten grafischen Kontext zeichnen zu können (Drucker, Bildschirm, Plotter, etc.)
- Sie müssen bestimmte Eigenschaften haben, mit denen näher beschrieben wird, wie sie sich zeichnen sollen, also beispielsweise mit welcher Strichstärke, mit welcher Farbe etc

Und Eine solche Anwendung objektorientiert zu verfassen, drängt sich geradezu auf. Man implementiert eine Basisklasse, die all diese Grundfähigkeiten zur Verfügung stellt. Diese Basisklasse könnte man nun beispielsweise `GraphicsObject` nennen. All die Methoden, wie beispielsweise das Zeichnen selbst, können natürlich nicht in dieser Basisklasse implementiert werden – denn `GraphicsObject` selbst ist ja nur der größte gemeinsame Nenner, zu dem das konkrete Zeichnen des Objekts aber nicht gehört.

Was soll `GraphicsObject` zum Zeitpunkt seiner Definition auch zeichnen? Es kann aber schon zu diesem Zeitpunkt beispielsweise Eigenschaften wie Farbe oder Ausmaße konkret implementieren – denn diese Eigenschaften sollen ja auch alle Objekte aufweisen, die sich später aus diesem `GraphicsObject` ableiten.

Erst wenn es dann darum geht, die konkrete Implementierung der wirklichen Objekte zu programmieren, kann auch das eigentliche Rendern in den abgeleiteten Klassen implementiert werden. Ein `Circle`-Objekt malt dann beispielsweise einen Kreis, ein `Line`-Objekt eine Linie und ein `Rectangle`-Objekt eben ein Rechteck.

Und wieso ist die objektorientierte Implementierung dieses Problems nun so teamfreundlich? Ganz einfach: Zwei Probleme können gleichzeitig von Teams gelöst werden, ohne dass sie zeitlich voneinander abhängig sind. Nachdem die Basisklassenimplementierung von `GraphicsObject` abgeschlossen wurde, kann sich Team A daran machen, die Objektverwaltung zu implementieren. Team A ist es dabei völlig egal, was eine Objektvariable vom Typ `GraphicsObject` tatsächlich malt. Team A verwaltet nur eine Liste dieses Typs und sorgt dafür, dass an den schon durch `GraphicsObject` definierten Positionen und in den entsprechenden Farben und in der entsprechenden Linienstärke die Objekte gemalt werden. Zum Testen benötigt Team A bestenfalls noch ein einziges konkret implementiertes Objekt. Doch was Team A beim Durchlaufen der Liste dann letzten Endes zeichnet, kann Team B völlig egal sein. Zwar ruft Team A die `Render`-Methode beim Zeichnen des Objekts von `GraphicsObject` auf, aber die Liste von `GraphicsObject` umfasst ja in Wirklichkeit keine `GraphicsObject`-Objekte, sondern andere, die Team B nacheinander konkretisiert. Nur eine Objektvariable vom Typ der Basisklasse wird also verwendet, um das Objekt zu zeichnen, das sich in der Liste wirklich hinter der Basisklassenobjektvariable verbirgt. Team B konkretisiert also von Team A unabhängig, was alles gezeichnet werden kann. Während Team A zwar die `Render`-Methode aufruft, hat Team B ja in den abgeleiteten konkreten Implementierungen der Klassen genau diese überschrieben, und dann passiert angewandte Polymorphie: Trotz gleichen Aufrufs geschieht etwas anderes – im Beispiel wird halt je nach wirklichem Objekttyp in der Liste etwas anderes gezeichnet. So programmiert Team A mithilfe des entworfenen Objektmodells eine Funktionalität, deren wirkliches Ergebnis Team A gar nicht kennt und – das ist das Entscheidende – auch gar nicht kennen muss!

Im Vergleich zur prozeduralen Programmierung gibt es hier darüber hinaus noch einen weiteren Vorteil, die ich schon angesprochen, aber an dieser Stelle noch einmal genauer erläutern möchte: Bei der prozeduralen Programmierung ist das »Vererben« von »Objekten« in Form von Code-Kopieren/Einfügen an der Tagesordnung gewesen: Wenn die Routine zum Zeichnen des Kreises mit allen Attributen stand, dann hat man (leider) vielfach den Code kopiert und anschließend daraus die Prozedur für das Linienzeichnen gebastelt. Wurde dann ein Fehler in der ersten `Circle`-Routine festgestellt, musste dieser Fehler natürlich auch in allen »abgeleiteten« (da manuell kopierten) Codezeilen behoben werden. Natürlich führte das oft zu Fehlern, die ihren Weg bis in die Endversion der Anwendung nahmen, denn bei vielleicht 20 oder 30 Prozeduren, die so zu korrigieren waren, wurde vielleicht schon mal die eine oder andere vergessen.

Bei der objektorientierten Vorgehensweise kann das nicht passieren. Ein Fehler in der Basisklasse `GraphicsObject` wird dort behoben; der entsprechende, ehemals fehlerhafte Code war aber auch nur ein einziges Mal implementiert. Das bedeutet: Alle abgeleiteten Klasse erben nicht nur die Grundfunktionsweise der Basisklasse, sondern natürlich auch alle Fehlerkorrekturen an ihr. Stabilere Programme lassen sich deswegen mithilfe der objektorientierten Entwicklungsweise viel eher garantieren, als es mit der prozeduralen Entwicklung möglich wäre.

Eine Gefahr bei der Entwicklung von Objektmodellen besteht allerdings umso mehr, wenn vormalige und langjährige Entwickler, die bislang prozedural entwickelt hatten, sich der OOP erstmalig bedienen. Diese Entwickler sind oftmals so fasziniert von den neuen Möglichkeiten der Vererbung und Polymorphie, dass sie beginnen, die unglaublichsten Objektmodelle zu entwerfen und dabei häufig den Fokus auf das verlieren, was eigentlich benötigt wird: Sie bauen das Fundament für einen 100-stöckigen Wolkenkratzer, obwohl sie dann nur ein Einfamilienhaus darauf setzen sollen. Eine Kontrollinstanz in Form eines Projektleiters, die dafür sorgt, dass ein Objektmodell nicht in Selbstverliebtheit seine OOP-Features verliert, ist deswegen oft von großer Bedeutung.

Ein erstes einfaches Beispiel zur Polymorphie

Ein erstes einfaches Beispiel für Polymorphie entführt uns für einen Moment in die Windows Forms-Entwicklung – genauer gesagt zu einer simplen Anwendung, die eine `ListBox` verwendet.

BEGLEITDATEIEN Das Grundgerüst für die folgenden Experimente finden Sie unter:

```
...\VB 2010 Entwicklerbuch\Kapitel 11\ToStringDemo
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Dieses Projekt besteht aus zwei »Elementen«, dem Formular mit der Steuerung der einzigen Schaltfläche sowie einer Klasse, die die Aufgabe hat, Kontaktdaten in Form von Nachnamen und Vornamen zu speichern:

```
'Der Formularcode
Public Class frmMain

    Private Sub btnKontaktHinzufügen_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
            Handles btnKontaktHinzufügen.Click
```

```
        'Neues Kontaktobjekt instanziieren und es mit den Inhalten
        'der Textbox füllen
        Dim locKontakt As New Kontakt(txtVorname.Text, txtNachname.Text)

        'Der Listbox hinzufügen
        lstKontakte.Items.Add(locKontakt)
    End Sub
End Class

'Die Kontakt-Klasse
Public Class Kontakt

    Private myVorname As String
    Private myNachname As String

    Sub New(ByVal Vorname As String, ByVal Nachname As String)
        myVorname = Vorname
        myNachname = Nachname
    End Sub

    Public Property Vorname() As String
        Get
            Return myVorname
        End Get
        Set(ByVal value As String)
            myVorname = value
        End Set
    End Property

    Public Property Nachname() As String
        Get
            Return myNachname
        End Get
        Set(ByVal value As String)
            myNachname = value
        End Set
    End Property
End Class
```

Interessant an dieser Stelle ist die fett geschriebene Codezeile im Listing. Sobald der Benutzer Vor- und Nachnamen in die `TextBox`-Steuerelemente eingegeben und die Schaltfläche betätigt hat, legt das Programm eine neue Instanz der `Kontakt`-Klasse an und fügt diesen Kontakt dann mithilfe der `Add`-Methode der `Items`-Auflistung der `ListBox` den `ListBox`-Elementen hinzu. Diese Vorgehensweise ist möglich, da anders als noch in Visual Basic 6.0, die `Add`-Methode der `ListBox` nicht nur `String`-Elemente, sondern beliebige Objekte aufnehmen kann.

Das Problem: Wenn die `ListBox` in ihrer `Items`-Auflistung Objekte vom Typ `Object` aufnimmt, kann man ihr sämtliche Typen übergeben (so auch unser `Kontakt`-Objekt). Das ist möglich, da, wie wir ja schon wissen, eine Objektvariable nicht nur auf den »eigenen Typ«, sondern auch auf jeden abgeleiteten Typ verweisen kann (Abschnitt »Polymorphie« und folgende ab Seite 389). Und da alle Klassen implizit von `Object` abgeleitet werden, basiert letzten Endes jede neu erfundene Klasse auch auf `Object`. Wann immer Ihnen also ein Parameter vom Typ `Object` begegnet, können Sie jeden beliebigen Typ übergeben. So weit, so gut.

Doch wer oder was sorgt nun dafür, dass ein Text in der `ListBox` angezeigt wird, der einen Sinn ergibt? In der jetzigen Ausbaustufe des Programms zeigt die `ListBox` jedenfalls noch nicht die gewünschten Ergebnisse an, was Sie schnell herausfinden können, wenn Sie das Programm laufen lassen:

Abbildung 11.8 Das Hinzufügen eines Objekts zur *ListBox* führt zunächst nur zur Anzeige des voll qualifizierten Objektnamens

Das Ergebnis entspricht sicherlich nicht dem erwarteten. Aber das kann es auch gar nicht, und Sie werden sehen, warum das so ist, wenn Sie erfahren, wie die `ListBox` arbeitet, um den Text eines Objekts zu ermitteln.

Vielleicht ahnen Sie es schon: Wenn die `ListBox` ihren Anzeigebereich mit Texten füllt, ruft sie die `ToString`-Funktion eines jeden Elements auf,[5] das ihrer `Items`-Auflistung hinzugefügt wurde. Das kann sie machen, ohne befürchten zu müssen, dass ein Objekt, das sie beinhaltet, kein `ToString` hat, denn `ToString` ist Bestandteil von `Object`. Und `Object` ist wiederum die Basisklasse aller Basisklassen, sodass jedes andere Objekt auch über eine `ToString`-Funktion verfügt. Die Standardimplementierung von `ToString` in `Object` sorgt allerdings lediglich dafür, dass der Typname des Objekts ausgegeben wird, und sieht folgendermaßen aus:

```
Public Overridable Function ToString() As String
      Return Me.GetType.ToString
End Function
```

`GetType` liefert den Typ eines Objekts zurück, der wiederum durch `ToString` in eine lesbare Zeichenfolge übersetzt wird. Und auf diese Weise zeigt `ToString` für jedes Objekt dessen Typnamen an – der Polymorphie sei Dank.

[5] Um genau zu sein: Sie ruft `GetItemText` ihrer Basisklasse `ListControl` auf, und *die* ermittelt den Text durch `ToString`, falls es sich beim Objekt nicht um ein Steuerelement handelt, das an eine andere Eigenschaft gebunden werden kann. In diesem Fall würde `GetItemText` den Text des Objekts per `ToString` ermitteln, der an das Steuerelement gebunden ist (oder einen Leerstring zurückliefern, falls es keine Bindung gibt). Aus diesem Grund würden Sie auch einen Leerstring in der Liste sehen, falls Sie der `Items`-Auflistung die Instanz irgendeines Steuerelements übergäben (wie beispielsweise eine Schaltfläche oder das Formular selbst).

Damit `ToString` einer Klasse beispielsweise bei der Darstellung in einer `ListBox` einen sinnvollen Text ausgibt, müssen Sie die `ToString`-Funktion in der entsprechenden Klasse überschreibend implementieren. Ergänzen Sie also die Klasse `Kontakt` um folgenden Code:

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

ist das Ergebnis wie erwartet, was auch Abbildung 11.9 zeigt.

Abbildung 11.9 Durch das Überschreiben von *ToString()* in der Klasse, deren Objektinstanz der Liste hinzugefügt wird, lässt sich der dargestellte Listentext steuern

Polymorphie in der Praxis

Ein neues CorelDraw im Rahmen dieses Buchs nur als Paradebeispiel für Polymorphie zu entwerfen, wäre vielleicht ein wenig sehr ambitioniert. Dennoch soll ein etwas größeres Praxisbeispiel zur anschaulichen Demonstration der Polymorphie dienen. Stellen Sie sich vor, Sie arbeiten in der EDV-Abteilung eines größeren Versandhauses. Ihre Aufgabe besteht darin, eine Software zu entwickeln, die Textlisten mit Artikeln verarbeitet, sie formatiert und anschließend den Summenwert ausgibt. Dabei können die verarbeiteten Artikel im Programm Datentypen mit ganz unterschiedlichen Eigenschaften sein: Video- und DVD-Artikeldatentypen etwa speichern neben dem Titel auch die Lauflänge und den Hauptdarsteller; Bücherartikeldatentypen speichern stattdessen den Autor und haben obendrein einen anderen Mehrwertsteuersatz (7% statt 19%).

Ohne Polymorphie wäre die Erstellung ein vergleichsweise schwieriges Unterfangen. Sie müssten eine Art »Superset« schaffen, das alle Eigenschaften beherrscht und die jeweils geforderten Sonderfälle programmieren. Regelrechter Spaghetticode wäre dabei buchstäblich vorprogrammiert.

Die Listen, die in simplen Textdateien vorliegen, sollen in diesem Beispiel ein bestimmtes Format ausweisen – Ihr Hauptprogramm muss dieses Format berücksichtigen und intern die Daten entsprechend speichern. Eine Liste, wie Sie sie von anderen Mitarbeitern Ihrer Abteilung als Textdatei bekommen, sieht typischerweise wie folgt aus:

```
;Inventarliste, die durch das Programm Inventory ausgewertet werden kann
;Format:
;Typ (1=Buch, 2=Cd oder Video), Bestellnummer, Titel, Zusatz, Brutto in Cent, Zusatz2
1;0001;Die Nachtwächter;Terry Pratchett und Andreas Brandhorst;1990;
1;0002;Kristall der Träume;Barbara Wood;2490;
1;0003;Volle Deckung - Mr. Bush: Dude where is my country;Michael Moore;1290;
1;0004;Du bist nie allein;Nicholas Sparks und Ulrike Thiesmeyer;1900;
2;0005;X-Men 2 - Special Edition;128;2299;Patrick Steward
2;0006;Sex and the City: Season 5;220;2999;Sarah Jessica Parker
2;0007;Indiana Jones (Box Set 4 DVDs);359;4499;Harrison Ford
2;0008;Die Akte;135;1499;Julia Roberts
```

Sie sehen: Die Liste enthält verschiedene Artikeltypen, die berücksichtigt werden müssen. Ihr Programm muss die einzelnen Artikel der Liste verarbeiten und sollte anschließend eine neue Liste mit folgendem Format ausspucken (die Preiszeile zeigt dabei nebeneinander den Nettopreis, den Mehrwertsteueranteil und den Bruttopreis):

```
0001    Die Nachtwächter
18,60 Euro      1,30 Euro       19,90 Euro

0002    Kristall der Träume
23,27 Euro      1,63 Euro       24,90 Euro

0003    Volle Deckung - Mr. Bush: Dude where is my country
12,06 Euro      0,84 Euro       12,90 Euro

0004    Du bist nie allein
17,76 Euro      1,24 Euro       19,00 Euro

0005    X-Men 2 - Special Edition
19,32 Euro      3,67 Euro       22,99 Euro

0006    Sex and the City: Season 5
25,20 Euro      4,79 Euro       29,99 Euro

0007    Indiana Jones (Box Set 4 DVDs)
37,81 Euro      7,18 Euro       44,99 Euro

0008    Die Akte
12,60 Euro      2,39 Euro       14,99 Euro

-----------------------------
Gesamtsumme: 189,66 Euro
-----------------------------
-----------------------------
```

Die Artikeltypen unterscheiden sich dabei in zweierlei Hinsicht: Zum einen gibt es bei Büchern einen anderen Mehrwertsteuersatz. Zum anderen sind es bei Büchern die Autoren, die in die Artikelinfo einfließen, bei Filmen ist es die Lauflänge. Das Programm muss obendrein die Möglichkeit bieten, kurze und ausführliche Listen zu erstellen. In der ausführlichen Liste sollen dann die erweiterten Eigenschaften der einzelnen Artikeltypen zu sehen sein. Eine ausführliche Liste sieht folgendermaßen aus:

```
0001    Die Nachwächter
Autor: Terry Pratchett und Andreas Brandhorst
18,60 Euro      1,30 Euro       19,90y Euro

0002    Kristall der Träume
```

```
Autor: Barbara Wood
23,27 Euro      1,63 Euro       24,90 Euro

0003    Volle Deckung - Mr. Bush: Dude where is my country
Autor: Michael Moore
12,06 Euro      0,84 Euro       12,90 Euro

0004    Du bist nie allein
Autor: Nicholas Sparks und Ulrike Thiesmeyer
17,76 Euro      1,24 Euro       19,00 Euro

0005    X-Men 2 - Special Edition
Laufzeit: 128 Min.
Hauptdarsteller: Patrick Steward
19,32 Euro      3,67 Euro       22,99 Euro

0006    Sex and the City: Season 5
Laufzeit: 220 Min.
Hauptdarsteller: Sarah Jessica Parker
25,20 Euro      4,79 Euro       29,99 Euro

0007    Indiana Jones (Box Set 4 DVDs)
Laufzeit: 359 Min.
Hauptdarsteller: Harrison Ford
37,81 Euro      7,18 Euro       44,99 Euro

0008    Die Akte
Laufzeit: 135 Min.
Hauptdarsteller: Julia Roberts
12,60 Euro      2,39 Euro       14,99 Euro
-----------------------------
Gesamtsumme: 189,66 Euro
-----------------------------
```

Zur Programmplanung: Da Sie vorher (also während der Entwurfszeit Ihres Programms) nicht wissen, wie viele Artikel Ihnen eine Datei zur Verfügung stellt, müssen Sie den Artikelspeicher dynamisch gestalten. Dazu gibt es zwei Möglichkeiten:

- Sie lesen die Datei komplett von vorne bis hinten durch und finden, noch bevor Sie einen Artikel verarbeitet haben, heraus, wie viele Artikel Sie verarbeiten müssen. Dann dimensionieren Sie ein Array in der Größe, die der Anzahl der Artikel entspricht, die Sie ja jetzt kennen. Anschließend lesen Sie in einem zweiten Durchgang die Artikel in das Array ein.
- Oder: Sie schaffen eine Klasse zum Speichern der Artikel, die sich dynamisch vergrößert. Die Klasse selbst könnte beispielsweise ein Array vordefinieren, mit einer Initialgröße von beispielsweise 4 Elementen. Wenn diese 4 Elemente nicht mehr ausreichen, legt sie ein neues temporäres Array an, dann beispielsweise mit 8 Elementen, kopiert die vorhandenen 4 Elemente des »alten« Arrays in das neue und tauscht anschließend die beiden Arrays aus, sodass das alte Member-Array der Klasse nun Platz für 8 Elemente hat. Das Kopieren der Array-Elemente von einem zum anderen Array scheint nur auf den ersten Blick zeitintensiv – dieser Vorgang kopiert, wie wir im vorherigen Abschnitt gelernt haben, nur die Zeiger auf die eigentlichen Artikeldaten, und das absolvieren moderne 32- und 64-Bit-Prozessoren in Lichtgeschwindigkeit.

  Würden Sie sich für die erste Option entscheiden, müsste die zu importierende Datei ein zweites Mal verarbeitet werden, was gerade bei größeren Dateien natürlich viel zeitintensiver ist.

Die Artikel selbst werden ebenfalls in Klassen gespeichert. Da beide Artikel viele Gemeinsamkeiten aufweisen und sich nur marginal voneinander unterscheiden, liegt es nahe, eine so genannte Basisklasse zu entwerfen, die die Gemeinsamkeiten abdeckt, und die Sonderfälle der jeweiligen Artikeltypen in zwei davon abgeleiteten Klassen zu implementieren. Wichtig für die Typsicherheit: Die »Listenklasse«, die die einzelnen Artikeldatentypen speichert, sollte nur Artikelklassen und von ihr abgeleitete Klassen aufnehmen.

Polymorphie ist in unserem Beispiel von unschätzbarem Wert. Die Artikelklassen, die von der Artikelbasisklasse abgeleitet sind, lassen sich nämlich über eine Objektvariable der Artikelbasisklasse steuern. Für die Entwicklung bedeutet das: Sie können ein Array verwalten, das nur den Typ »Artikelbasisklasse« aufnimmt, obwohl ganz andere (na ja, vielleicht nicht *ganz* andere, sondern nur erweiterte) Klasseninstanzen darin gespeichert werden. Der entscheidende Clou dabei ist, dass Sie im Code zwar beispielsweise eine Methode der Artikelbasisklasse für das Ausführen einer bestimmten Funktion angeben, dann aber doch die (andere) Methode einer abgeleiteten Klasse aufgerufen wird, wenn diese in der Basisklasse überschrieben wurde.

BEGLEITDATEIEN Das mag zunächst ein wenig verwirrend klingen, wird aber klar, wenn Sie das Beispielprogramm zur Hilfe nehmen, das Sie unter dem Projektmappen-Namen *Inventory01* im folgenden Verzeichnis finden:

```
...\VB 2010 Entwicklerbuch\Kapitel 11\Inventory01
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Doch bevor wir uns mit dem Polymorphieaspekt dieses Beispiels beschäftigen, lassen Sie uns zunächst die Techniken klären, mit denen die Voraussetzungen zum Speichern der Daten geschaffen werden. Betrachten Sie dazu als Erstes die Klasse `DynamicList` zur Speicherung der Artikel:

```
Class DynamicList

    private myStep As Integer = 4         ' Schrittweite, um die das Array erhöht wird.
    private myCurrentArraySize As Integer ' Aktuelle Array-Größe
    private myCurrentCounter As Integer   ' Zeiger auf aktuelles Element
    private myArray() As ShopItem         ' Array mit den Elementen.

    Sub New()
        myCurrentArraySize = myStep
        ReDim myArray(myCurrentArraySize - 1)
    End Sub

    Sub Add(ByVal Item As ShopItem)

        'Element im Array speichern
        myArray(myCurrentCounter) = Item

        'Zeiger auf nächstes Element erhöhen
        myCurrentCounter += 1

        'Prüfen, ob aktuelle Arraygrenze erreicht wurde
        If myCurrentCounter = myCurrentArraySize - 1 Then
            'Neues Array mit mehr Speicher anlegen
            'und Elemente hinüberkopieren. Dazu:

            'Neues Array wird größer:
            myCurrentArraySize += myStep
```

```
            'temporäres Array erstellen
            Dim locTempArray(myCurrentArraySize - 1) As ShopItem

            'Elemente kopieren; das geht mit dieser
            'statischen Methode extrem schnell, da zum einen nur die
            'Zeiger kopiert werden, zum anderen diese Routine
            'intern nicht in Managed Code sondern nativem Assembler ausgeführt wird.
            Array.Copy(myArray, locTempArray, myArray.Length)

            'Auch hier werden nur die Zeiger auf die Elemente "verbogen".
            'Die vorherige Liste der Zeiger in myArray, die nun verwaist ist,
            'fällt dem Garbage Collector zum Opfer.
            myArray = locTempArray
        End If
    End Sub

    'Liefert die Anzahl der vorhandenen Elemente zurück
    Public Overridable ReadOnly Property Count() As Integer
        Get
            Return myCurrentCounter
        End Get
    End Property

    'Erlaubt das Zuweisen
    Default Public Overridable Property Item(ByVal Index As Integer) As ShopItem
        Get
            Return myArray(Index)
        End Get

        Set(ByVal Value As ShopItem)
            myArray(Index) = Value
        End Set
    End Property
End Class
```

Die `Add`-Methode in dieser Klasse ist das Entscheidende. Sie überprüft, ob das Array noch ausreichend groß ist. Falls das nicht der Fall ist, führt sie den Tauschvorgang mit einer lokalen Array-Variablen durch, die entsprechend größer definiert wurde und in die die Array-Elemente zuvor kopiert wurden. Damit zeigt das Klassenmember `myArray` jetzt auf die neuen Array-Elemente – die Zeiger auf die alten Array-Elemente werden buchstäblich »vergessen«. Übrigens: Den Arbeitsspeicher, der auf diese Weise unnötigerweise belegt wird, holt sich .NET Framework eigenständig mithilfe der so genannten Garbage Collection (»Müllabfuhr«) wieder zurück. Der Garbage Collector (kurz »GC«) schaut sich – vereinfacht ausgedrückt – an, ob Objekte, die Speicher belegen, noch irgendeinen Bezug zu einer verwendeten Objektvariablen haben. Falls nicht, werden sie entsorgt. In diesem Beispiel haben die Zeiger auf die ursprünglichen Elemente, die im Member-Array `myArray` gespeichert waren, nach dem Kopieren keinen Bezug mehr zu irgendeinem Objekt: Der Objektname wurde mit der Zeile

```
'temporäres Array dem Memberarray zuweisen
myArray = locTempArray
```

auf die neuen Elemente »umgebogen«. Die ursprünglichen Elemente stehen anschließend bezugslos im Speicher und werden beim nächsten GC-Durchlauf entsorgt.

WICHTIG Behalten Sie im Hinterkopf, dass die Daten der eigentlichen Objektinstanzen bei diesem Vorgang überhaupt nicht angetastet werden. Und das ist auch der Grund, weshalb das Kopieren eines Arrays mit `Array.Copy` so unglaublich schnell vonstatten geht (was ebenfalls der Fall wäre, würden Sie den Vorgang selbst in die Hand nehmen, das ganze Array in einer `For`-Schleife durchlaufen lassen und die einzelnen Elemente dem neuen Array zuweisen).

Die `Add`-Methode nimmt ausschließlich Objekte eines bestimmten Typs entgegen. In diesem Beispiel habe ich sie `ShopItem` (»Ladenartikel«) genannt. Diese Klasse stellt die Basis für die Artikelspeicherung dar und sieht folgendermaßen aus:

```
Class ShopItem

    private myTitle As String                 ' Titel
    private myNetPrice As Double              ' Nettopreis
    private myOrderNumber As String           ' Artikelnummer
    private myPrintTypeSetting As PrintType   ' Ausgabeform

    Public Sub New()
        myPrintTypeSetting = PrintType.Detailed
    End Sub

    Public Sub New(ByVal StringArray() As String)
        Title = StringArray(FieldOrder.Titel)
        'FieldOrder ist übrigens eine Enum und wird weiter unten definiert:
        GrossPrice = Double.Parse(StringArray(FieldOrder.GrossPrice)) / 100
        OrderNumber = StringArray(FieldOrder.OrderNumber)
        PrintTypeSetting = PrintType.Detailed
    End Sub

    Public Property Title() As String
        Get
            Return myTitle
        End Get
        Set(ByVal Value As String)
            myTitle = Value
        End Set
    End Property

    Public Property OrderNumber() As String
        Get
            Return myOrderNumber
        End Get
        Set(ByVal Value As String)
            myOrderNumber = Value
        End Set
    End Property

    Public Property NetPrice() As Double
        Get
            Return myNetPrice
        End Get

        Set(ByVal Value As Double)
            myNetPrice = Value
        End Set

    End Property

    Public ReadOnly Property NetPriceFormatted() As String
```

```
        Get
            Return NetPrice.ToString("#,##0.00") + " Euro"
        End Get
    End Property

    Public Overridable Property GrossPrice() As Double
        Get
            Return myNetPrice * 1.19
        End Get

        Set(ByVal Value As Double)
            myNetPrice = Value / 1.19
        End Set
    End Property

    Public ReadOnly Property GrossPriceFormatted() As String
        Get
            Return GrossPrice.ToString("#,##0.00") + " Euro"
        End Get
    End Property

    Public ReadOnly Property VATAmountFormatted() As String
        Get
            Return (GrossPrice - myNetPrice).ToString("#,##0.00") + " Euro"
        End Get
    End Property

    Public Overridable ReadOnly Property Description() As String
        Get
            Return OrderNumber & vbTab & Title
        End Get
    End Property

    Public Property PrintTypeSetting() As PrintType
        Get
            Return myPrintTypeSetting
        End Get

        Set(ByVal Value As PrintType)
            myPrintTypeSetting = Value
        End Set
    End Property

    Public Overrides Function ToString() As String

        If PrintTypeSetting = PrintType.Brief Then
            'Kurzform: Es wird die Description-Eigenschaft des Objekts gemäß
            'ursprünglicher Implementierung in der Basisklasse verwendet
            Return MyClass.Description & vbCr & vbLf &
                Me.NetPriceFormatted & vbTab & Me.VATAmountFormatted & vbTab &
                Me.GrossPriceFormatted & vbCr & vbLf
        Else
            'Langform: Die überschriebene, erweiterte Description-Eigenschaft
            'des Objekts selbst wird verwendet
            Return Me.Description & vbCr & vbLf &
                Me.NetPriceFormatted & vbTab & Me.VATAmountFormatted & vbTab &
                Me.GrossPriceFormatted & vbCr & vbLf
        End If

    End Function

End Class
```

Es ist nicht sonderlich schwer, diese Klasse zu begreifen, denn sie dient nur zwei Aufgaben: dem Speichern von Daten, die durch Eigenschaften zugänglich gemacht werden und der formatierten Ausgabe einiger dieser Daten. Einige der Eigenschaften sind obendrein nur als `ReadOnly` definiert, da es keinen Sinn ergäbe, auf sie schreibend zuzugreifen.

Zahlen mit ToString formatiert in Zeichenketten umwandeln

Erwähnenswert an dieser Stelle ist das Merkmal der `ToString`-Funktion primitiver Datentypen (`Integer`, `Double` etc.), Zahlen formatiert auszugeben. In diesem Fall verwenden Sie eine Überladung von `ToString`, die einen String als Parameter akzeptiert. In diesem Beispiel lautet der String »#,##0.00«, der bewirkt, dass Nachkommastellen unabhängig vom Wert grundsätzlich zweistellig und der Wert vor dem Komma im Bedarfsfall mit Tausendertrennzeichen formatiert werden. Bitte beachten Sie, dass Sie hierbei die amerikanische/englische Schreibweise verwenden, bei der das Tausendertrennzeichen ein Komma und das Nachstellentrennzeichen ein Punkt ist.

Interessant ist es, als Nächstes die aus der Basisklasse abgeleiteten Klassen zu erkunden, die Sie im Folgenden abgedruckt finden:

```
Class BookItem
    Inherits ShopItem

    private myAuthor As String

    Public Sub New(ByVal StringArray() As String)
        MyBase.New(StringArray)
        Author = StringArray(FieldOrder.AdditionalRemarks1)
    End Sub

    Public Overridable Property Author() As String
        Get
            Return myAuthor
        End Get
        Set(ByVal Value As String)
            myAuthor = Value
        End Set
    End Property

    Public Overrides Property GrossPrice() As Double
        Get
            Return NetPrice * 1.07
        End Get

        Set(ByVal Value As Double)
            NetPrice = Value / 1.07
        End Set
    End Property

    Public Overrides ReadOnly Property Description() As String
        Get
            Return OrderNumber & vbTab & Title & vbCr & vbLf & "Autor: " & Author
        End Get
    End Property

End Class
```

Durch das `Inherits`-Schlüsselwort direkt nach dem `Class`-Schlüsselwort bestimmen Sie, dass die neue Klasse, die die Bücherartikel speichert, von der Basisklasse `ShopItem` abgeleitet wird. Das befähigt Objektvariablen, die als `ShopItem` definiert wurden, auch Instanzen von `BookItem` zu referenzieren, denn es gilt: Jede abgeleitete Klasse kann durch eine Objektvariable der Basisklasse »angesprochen« werden. Dazu ein Beispiel: Wenn Sie eine Objektvariable namens `EinArtikel` auf folgende Weise deklariert

```
Dim EinArtikel as ShopItem
```

und entsprechend, etwa durch

```
EinArtikel=New ShopItem()
```

definiert haben, kann ihr Inhalt später im Programm durch die Anweisung

```
Console.WriteLine(EinArtikel.GrossPriceFormatted)
```

ausgegeben werden, und es wird, wie zu erwarten, `GrossPriceFormatted` (etwa: »Bruttopreis formatiert«) der Basisklasse `ShopItem` verwendet. Wird hingegen – und jetzt wird es interessant – eine Instanz der abgeleiteten Klasse etwa durch die folgende Zeile

```
EinArtikel=New BookItem(…)
```

durch **dieselbe** Objektvariable `EinArtikel` angesprochen und wird später die gleiche Funktion aufgerufen, etwa erneut durch

```
Console.WriteLine(EinArtikel.GrossPriceFormatted)
```

so wird dieses Mal nicht die `GrossPriceFormatted`-Funktion der Basisklasse, sondern die der abgeleiteten Klasse `BookItem` verwendet. Und genau das sind die unschlagbaren Vorteile der Polymorphie: Sie haben ein Steuerprogramm, das in einer Basisklassenobjektvariablen die augenscheinlich gleiche Funktion aufruft, und doch können Sie durch das Überschreiben genau dieser Funktion in einer abgeleiteten Klasse ein anderes Ergebnis erzielen.

Die Basisklasse unseres Beispiels stellt einen parametrisierten Konstruktor bereit, der ein `String`-Array übernimmt. Aus diesem `String`-Array werden die Daten für den Inhalt einer Klasseninstanz entnommen:

```
    Public Sub New(ByVal StringArray() As String)
        Title = StringArray(FieldOrder.Titel)
        GrossPrice = Double.Parse(StringArray(FieldOrder.GrossPrice)) / 100
        OrderNumber = StringArray(FieldOrder.OrderNumber)
        PrintTypeSetting = PrintType.Detailed
    End Sub
```

Das übergebende Array enthält die einzelnen Daten immer in Elementen mit dem gleichen Index, die zum einfacheren Verständnis des Quellcodes übrigens in einer `Enum`-Aufzählung festgehalten sind (mehr zum Thema *Enums* erfahren Sie in Kapitel 17.

```
Enum FieldOrder ' Zuständig für die Reihenfolge der Felder
    Type
    OrderNumber
    Titel
    AdditionalRemarks1
    GrossPrice
    AdditionalRemarks2
End Enum
```

Die abgeleitete Klasse `BookItem` hat nun viel weniger Arbeit, eine Instanz zu definieren. Sie ruft einfach den Konstruktor der Basisklasse auf und ergänzt ihren eigenen Konstruktor nur noch um die Zuweisung eines bestimmten Array-Elements des ihr übergebenen Parameters:

```
Public Sub New(ByVal StringArray() As String)
    MyBase.New(StringArray)
    Author = StringArray(FieldOrder.AdditionalRemarks1)
End Sub
```

Da Bücher im Gegensatz zu vielen anderen Artikeln mit einem anderen Mehrwertsteuersatz belegt sind (jedenfalls noch), muss die Funktion, die den Bruttopreis berechnet, überschrieben werden:

```
Public Overrides Property GrossPrice() As Double
    Get
        Return NetPrice * 1.07
    End Get

    Set(ByVal Value As Double)
        NetPrice = Value / 1.07
    End Set
End Property
```

Und schon wieder sehen Sie Polymorphie in Aktion: Die abgeleitete Klasse muss jetzt die Methode, die den formatierten Bruttopreis als `String` zurückgibt, nicht noch zusätzlich überschreiben, denn wenn Sie die Funktion der Basisklasse betrachten:

```
Public ReadOnly Property GrossPriceFormatted() As String
    Get
        Return GrossPrice.ToString("#,##0.00") + " Euro"
    End Get
End Property
```

stellen Sie fest, dass sie nicht direkt auf eine Feldvariable zurückgreift, sondern ihrerseits eine in der Klasse implementierte Eigenschaft bemüht. Aus Sicht der abgeleiteten Klasse `BookItem` wird hier nicht `GrossPrice` der Basisklasse, sondern der (überschriebenen) abgeleiteten Klasse aufgerufen und damit der formatierte Bruttopreis eines Buchs berücksichtigt, also korrekt mit 7% Mehrwertsteuer und nicht mit 19% der Basisklasse. `GrossPriceFormatted` ruft also die überschriebene Eigenschaftenprozedur der abgeleiteten Klasse auf, obwohl `GrossPriceFormatted` ausschließlich in der Basisklasse zu finden ist!

Eine ähnliche Vorgehensweise legen die Funktionen `Description` (für die Zusammensetzung des Beschreibungstextes) und `VATAmountFormatted` (für den Betrag der Mehrwertsteuer) an den Tag.

Und auch die zweite Klasse zur Speicherung von Video- und DVD-Artikeln – sie nennt sich `DVDItem` – macht sich die Polymorphie zunutze:

```
Class DVDItem
    Inherits ShopItem

    Protected myRunningTime As Integer
    Protected myActor As String

    Public Sub New(ByVal StringArray() As String)
        MyBase.New(StringArray)
        RunningTime = Integer.Parse(StringArray(FieldOrder.AdditionalRemarks1))
        Actor = StringArray(FieldOrder.AdditionalRemarks2)
    End Sub

    Public Overridable Property RunningTime() As Integer
        Get
            Return myRunningTime
        End Get
        Set(ByVal Value As Integer)
            myRunningTime = Value
        End Set
    End Property

    Public Overridable Property Actor() As String
        Get
            Return myActor
        End Get
        Set(ByVal Value As String)
            myActor = Value
        End Set
    End Property

    Public Overrides ReadOnly Property Description() As String
        Get
            Return OrderNumber & vbTab & Title & vbCr & vbLf & "Laufzeit: " & myRunningTime & " Min." &
vbCr & vbLf & "Hauptdarsteller: " & Actor
        End Get
    End Property

End Class
```

Sie sehen an diesem Beispiel, wie sehr Sie die Polymorphie bei der Wiederverwendbarkeit von Code unterstützt und zu einer enormen Arbeitsersparnis beiträgt. Und das gleich in doppeltem Sinne: Wenn die Basisklasse funktioniert, funktionieren die abgeleiteten Klassen bis auf ihre zusätzliche Funktionalität genauso reibungslos. Neben dem reduzierten Tippaufwand sparen Sie obendrein viel Zeit beim Suchen von Fehlern.

Die Klassen für unser Beispiel sind damit komplett fertig gestellt. Was jetzt noch zu tun bleibt, ist die Implementierung des Hauptprogramms, das die Ursprungstextdatei einliest, die Elemente aus jeder Textzeile erstellt, sie der Liste hinzufügt und die Ergebnisse schließlich ausgibt:

```
Module mdlMain

    'Die Inventardatei muss im Programmverzeichnis stehen.
    Private Filename As String = My.Application.Info.DirectoryPath & "\Inventar.txt"

    Sub Main()

        'Streamreader zum Einlesen der Textdateien
        Dim locSr As StreamReader
        Dim locList As New DynamicList  ' Die dynamische Liste
        Dim locElements() As String     ' Die einzelnen ShopItem-Elemente
        Dim locShopItem As ShopItem     ' Ein einzelnes Shop-Element
        Dim locDisplayMode As PrintType ' Der Darstellungsmodus

        'Schauen, ob die Textdatei vorhanden ist:
        'Der Einfachheit halber einfach hier mögliche Fehler abfangen … sorry, Herr Fachlektor! :-)
        Try
            locSr = New StreamReader(Filename, System.Text.Encoding.Default)
        Catch ex As Exception
            Console.WriteLine("Fehler beim Lesen der Inventardatei!" & _
                              vbNewLine & ex.Message)
            Console.WriteLine()
            Console.WriteLine("Taste drücken zum Beenden")
            Console.ReadKey()
            Exit Sub
        End Try

        Console.WriteLine("Wählen Sie (1) für kurze und (2) für ausführliche Darstellung")
        Dim locKey As Char = Console.ReadKey.KeyChar
        If locKey = "1"c Then
            locDisplayMode = PrintType.Brief
        Else
            locDisplayMode = PrintType.Detailed
        End If

        Do
            Try
                'Zeile einlesen
                Dim locLine As String = locSr.ReadLine()

                'Nichts eingegeben, dann war's das!
                If String.IsNullOrEmpty(locLine) Then
                    locSr.Close()
                    Exit Do
                End If

                'Semikolon überlesen
                If Not locLine.StartsWith(";") Then
                    'So braucht man kein explizites Char-Array zu deklarieren
                    'um die Zeile in die durch Komma getrennten Elemente zu zerlegen
                    locElements = locLine.Split(New Char() {";"c})

                    If locElements(FieldOrder.Type) = "1" Then
                        locShopItem = New BookItem(locElements)
                    Else
```

```
                    locShopItem = New DVDItem(locElements)
                End If
                locList.Add(locShopItem)
            End If

        Catch ex As Exception
            Console.WriteLine("Fehler beim Auswerten der Inventardatei!" & _
                              vbNewLine & ex.Message)
            Console.WriteLine()
            Console.WriteLine("Taste drücken zum Beenden")
            Console.ReadKey()
            locSr.Close()
            Exit Sub
        End Try

    Loop

    Dim locGrossAmount As Double = 0

    'Alle Elemente ausgeben
    For count As Integer = 0 To locList.Count - 1
        locList(count).PrintTypeSetting = locDisplayMode
        Console.WriteLine(locList(count).ToString())
        locGrossAmount += locList(count).GrossPrice
    Next

    Console.WriteLine()
    Console.WriteLine("-----------------------------")
    Console.WriteLine("Gesamtsumme: " & locGrossAmount.ToString("#,##0.00") & " Euro")
    Console.WriteLine("-----------------------------")
    Console.WriteLine("-----------------------------")
    Console.WriteLine()
    Console.WriteLine("Return drücken, zum Beenden")
    Console.ReadLine()

End Sub

End Module
```

Sie sehen: Den meisten Platz beanspruchen die `WriteLine`-Zeilen, die sich um die Ausgaben kümmern und die Artikel auf dem Bildschirm darstellen. Da die überwiegenden Lösungsschritte des Problems bereits von den Artikelklassen bewältigt werden, beschränkt sich die eigentliche Aufgabe des Hauptprogramms darauf, die Datei zu öffnen, ein Element aus der Klasse zu lesen und zu speichern.

Auch das Hauptprogramm bedient sich der Polymorphie, nämlich dann, wenn es die Liste auf dem Bildschirm ausgibt:

```
'Alle Elemente ausgeben
For count As Integer = 0 To locList.Count – 1
    locList(count).PrintTypeSetting = locDisplayMode
    Console.WriteLine(locList(count).ToString())
    locGrossAmount += locList(count).GrossPrice
Next
```

Es iteriert in einer Zählschleife durch die einzelnen Elemente der Klasse. Da die `Item`-Eigenschaft der `DynamicList`-Klasse die Default-Eigenschaft ist, muss der Eigenschaftenname `Item` an dieser Stelle noch nicht einmal angegeben werden – es reicht aus, das gewünschte Element durch eine direkt an den Objektnamen angefügte Klammer zu indizieren.

Mit `ToString` des indizierten Objekts erfolgt anschließend der Ausdruck auf dem Bildschirm. Welches `ToString` das Programm dabei verwendet, ist wieder abhängig von der Klasse, deren Instanz im Array-Element gespeichert wird. Ist es ein `BookItem`-Objekt, wird `ToString` von `BookItem` aufgerufen; bei einem `DVDItem`-Objekt wird die `ToString`-Funktion eben dieser Klasse aufgerufen. Gleiches gilt anschließend für die Berechnung der Mehrwertsteuer und die Funktion `GrossPrice`.

Polymorphie und der Gebrauch von Me, MyClass und MyBase

Die Option, dem Benutzer entweder eine ausführliche Liste oder eine sehr kurz gehaltene Liste auszudrucken, ist eine der Anforderungen an das Programm. Nun gäbe es zwei theoretische Ansätze, die Lösung dieses Problems zu realisieren. Die erste Möglichkeit: Der Ausdruck wird komplett durch das steuernde Hauptprogramm durchgeführt. In diesem Fall müsste das Hauptprogramm dafür sorgen, dass die Sonderfälle unterschieden würden. Im Prinzip gäbe es dabei zwei verschiedene Druckroutinen, die jeweils einmal für den ausführlichen und einmal für den kompakten Ausdruck zuständig wären. Möglichkeit Nr. 2: Die Klassen selbst sorgen für die Aufbereitung der entsprechenden Texte.

Zufälligerweise gibt es eine Eigenschaft in der Basisklasse, die einen recht kompakten, beschreibenden Text für den Inhalt einer Objektinstanz zurückliefert. Diese Eigenschaft hat den Namen `Description`. Diese Eigenschaft wird von den abgeleiteten Klassen überschrieben; sie erweitern die Eigenschaft dahingehend, dass auch die zusätzlichen gespeicherten Informationen bei der Ausgabe berücksichtigt werden.

Viel leichter wäre es jetzt also, wenn das Programm in Abhängigkeit von der Eingabe des Benutzers entweder die Eigenschaft der Basisklasse oder die der jeweiligen abgeleiteten Klasse für die Ausgabe auf den Bildschirm verwenden würde. Die Basisklasse müsste zu diesem Zweck eine weitere Eigenschaft bereitstellen, mit der sich steuern ließe, ob die kompakte oder die ausführliche Form bei der Ausgabe der Artikelbeschreibung zu berücksichtigen ist.

Aus diesem Grund gibt es bereits in der Basisklasse eine Eigenschaft namens `PrintType`, die nur zwei Zustände aufweisen kann, welche in einer `Enum` definiert sind:

```
Enum PrintType  ' Reportform
    Brief       ' kurz
    Detailed    ' ausführlich
End Enum
```

Bevor nun die eigentliche Bildschirmausgabe des Inhalts eines Artikelobjekts erfolgt, muss das die Ausgabe steuernde Programm dafür sorgen, dass die `PrintType`-Eigenschaft auf den korrespondierenden Wert gesetzt wird, den der Benutzer beim Start des Programms definiert hat. Wenn diese Voraussetzung erfüllt ist, kann ein Artikelobjekt selbst entscheiden, ob die Beschreibung der Basisklasse oder der eigenen Klasse zurückgeliefert werden soll. Da dieser »Entscheidungsalgorithmus« sowohl in der Basisklasse als auch in allen abgeleiteten Klassen derselbe ist, reicht es aus, ihn ausschließlich in der Basisklasse zu implementieren. Durch die Polymorphie ist es anschließend möglich, die »richtige« `Description`-Eigenschaft der jeweiligen Klasse abzufragen und daraus den Rückgabetext zusammenzubasteln.

Das letzte Problem ist die Realisierung der kompakten Form des Artikeldrucks. Hier müsste der `ToString`-Funktion die Möglichkeit gegeben sein, die Polymorphie außer Kraft zu setzen und gezielt die `Description`-Eigenschaft der Basisklasse aufzurufen, egal ob es sich bei dem gespeicherten Objekt um ein abgeleitetes oder um ein Original handelt. Genau das erreichen Sie mit dem Schlüsselwort `MyClass`. Dieses erlaubt, gezielt auf ein Element der Klasse zuzugreifen, in der `MyClass` »steht«. Sie heben damit quasi die Überschreibung einer Methode für die Dauer des Aufrufs auf und verwenden das Original der Klasse, in der `MyClass` eingesetzt wird. Die `ToString`-Funktion der Basisklasse macht sich genau diese Eigenschaft zunutze:

```
Public Overrides Function ToString() As String

    If PrintTypeSetting = PrintType.Brief Then
        'Kurzform: Es wird die Description-Eigenschaft des Objekts gemäß
        'ursprünglicher Implementierung in der Basisklasse verwendet
        Return MyClass.Description & vbCr & vbLf & _
            Me.NetPriceFormatted & vbTab & Me.VATAmountFormatted & vbTab & _
            Me.GrossPriceFormatted & vbCr & vbLf
    Else
        'Langform: Die überschriebene, erweiterte Description-Eigenschaft
        'des Objekts selbst wird verwendet
        Return Me.Description & vbCr & vbLf & _
            Me.NetPriceFormatted & vbTab & Me.VATAmountFormatted & vbTab & _
            Me.GrossPriceFormatted & vbCr & vbLf
    End If
End Function
```

Die nachstehende Tabelle fasst die verschiedenen Schlüsselwörter zusammen, die den Klassenebenenzugriff spezifizieren:

Schlüsselwort	Beschreibung
Me	Das Element der eigenen Klasse wird verwendet
MyClass	Das Element der Klassenableitung, die das *MyClass*-Schlüsselwort enthält, wird verwendet
MyBase	Das Element der Basisklasse wird verwendet.

Tabelle 11.1 Schlüsselwörter zur Bestimmung von überschriebenen Elementen unterschiedlichen Rangs in der Vererbungshierarchie einer Klasse

Abstrakte Klassen und virtuelle Prozeduren

Schreiben Sie Ihre Texte auch mit Microsoft Word? Kennen Sie dann auch Dokumentenvorlagen? Ja? Gut. Dann wissen Sie quasi auch, was abstrakte Klassen sind. Dokumentenvorlagen in Word dienen dazu, eine Formatierungsrichtlinie für Dokumente festzuschreiben.

Wenn ich, wie in diesem Buch, einen »Standardabsatz« schreibe, dann bestimmt die Dokumentenvorlage, dass die Schrift dafür »Minion« und die Schriftgröße auf 10 Punkt gesetzt wird. Wenn ein Dokument auf Basis einer Dokumentenvorlage erstellt ist, übernimmt es alle ihre Eigenschaften. Nur so ist gewährleistet, dass die Kapitel eines Buchs später auch alle im gleichen Stil formatiert sind.

Abstrakte Klassen funktionieren nach ähnlichem Konzept. Sie stellen Prototypen von Prozeduren bereit, aber sie dienen ausschließlich als Vorlage. Um Entwickler dabei – rabiat ausgedrückt – dazu zu zwingen, bestimmte Elemente neu zu implementieren, stellen sie zusätzlich virtuelle Prozeduren bereit. Eine virtuelle Prozedur hat zwei Aufgaben:

- Sie gibt dem Entwickler die Signatur einer Prozedur (oder die Signaturen für überladene Prozeduren) vor
- Sie »zwingt« den Entwickler, in der Ableitung der Klasse die vorhandene Prozedur zu überschreiben und sie vor allen Dingen damit zu implementieren

Ein Praxisbeispiel (und dazu muss ich gar nicht weit ausholen): Stellen Sie sich vor, das Beispielprogramm aus dem letzten Abschnitt verrichtet seinen Dienst in einem großen Versandhaus wie Otto oder Amazon. Hier sind natürlich wesentlich mehr Artikelgruppen zu berücksichtigen – bei Versendern wie Amazon, die weltweit operieren, müssen auch noch weitere Mehrwertsteuersätze mit einbezogen werden.

Es würde ziemlich viel Sinn ergeben, die vorhandene Artikelklasse `ShopItem` als abstrakte Klasse zu formulieren. Damit Teammitglieder gezwungen sind, die mehrwertsteuerbezogenen Funktionen neu zu implementieren (sodass sie nicht vergessen werden), läge es nahe, die betroffenen Funktionen als virtuelle Funktionen auszulegen. Sie stellen damit sicher, dass jeder Entwickler *bewusst* programmiert, denn implementiert er die mehrwertsteuerbezogenen Funktionen nicht, wird seine Klasse nicht kompilierbar sein.

HINWEIS Das gilt übrigens auch für unser hypothetisches Beispiel, das Sie im grauen Kasten ab Seite 390 beschrieben finden. Es macht hier viel Sinn, die Basisklasse `GraphicsObject` abstrakt zu implementieren, da sie selbst eine `Render`-Methode noch gar nicht implementieren könnte! Was sollte sie auch zeichnen?

An dem Beispielprogramm müssen Sie dabei kaum Änderungen vornehmen. Sie können nach dem Umbau `ShopItem` auch weiterhin als Objektvariable verwenden. Sie können die Klasse nur nicht mehr direkt instanziieren – aber das haben wir im Beispielprogramm ohnehin nicht gemacht (als hätte ich es kommen sehen …).

Eine Klasse mit MustInherit als abstrakt deklarieren

Um eine abstrakte Klasse zu erstellen, verwenden Sie das Schlüsselwort `MustInherit`. Wenn Sie dieses Schlüsselwort vor das Schlüsselwort `Class` der Klasse `ShopItem` in unserem Beispiel stellen, passiert erst einmal gar nichts. Sie haben damit zunächst einmal erreicht, dass die Klasse nur noch in einer Ableitung instanziiert werden kann. Da wir `ShopItem` selbst nicht instanziieren, bleibt alles beim Alten, und es gibt auch keine Fehlermeldungen. Was Sie allerdings nun nicht mehr können (aber vorher hätten tun können) ist, die Klasse irgendwo im Programm zu instanziieren. Versuchen Sie es: Fügen Sie in der `Sub Main` folgende Zeile ein,

```
Dim locShopItemVersuchsInstanz As New ShopItem
```

so erhalten Sie eine Fehlermeldung etwa wie in Abbildung 11.10 zu sehen:

```
Dim locShopItemVersuchsInstanz As New ShopItem

Dim locKey As Char = Console.ReadKey.KeyChar
If locKey = "1"c Then
    locDisplayMode = PrintType.Brief
Else
    locDisplayMode = PrintType.Detailed
End If
```

"New" darf nicht für eine Klasse verwendet werden, die als "MustInherit" deklariert ist.
Löschen Sie "New".
Dim locShopItemVersuchsInstanz As ~~New~~ ShopIte
Alle Seitenansichten erweitern

Abbildung 11.10 Eine mit *MustInherit* als abstrakt definierte Klasse kann nicht instanziiert werden

Eine Methode oder Eigenschaft einer abstrakten Klasse mit MustOverride als virtuell deklarieren

Fehler wünschen Sie sich als Entwickler normalerweise eher weniger. Ist die Fehlerquelle jedoch eine virtuell definierte Methode oder Eigenschaft, sieht das allerdings anders aus: In diesem Fall tritt eine Fehlermeldung in der Regel deswegen auf, weil eine Methode oder Eigenschaft in einer abgeleiteten Klasse nicht überschrieben – ergo: neu implementiert – wurde. Aber genau das wollen Sie mit virtuellen Methoden bzw. Eigenschaften erreichen.

In unserem Beispiel ist die Eigenschaft `GrossPrice`, die den Bruttopreis eines Artikels aus dem Nettopreis mit dem gültigen Mehrwertsteuersatz errechnet, eine willkommene Demonstration für eine virtuelle Eigenschaft, denn: Es ist wichtig, dass sich jeder Entwickler darüber im Klaren ist, dass er in einer Ableitung der Klasse `ShopItem` selbst für das korrekte Errechnen des Bruttopreises verantwortlich ist und eine Variante dieser Eigenschaft auf jeden Fall in seiner abgeleiteten Klasse implementiert.

```
64    Public MustOverride Property GrossPrice() As Double
65        Get
66            Return myNetPrice * 1.19
67        End Get
68
69        Set(ByVal Value As Double)
70            myNetPrice = Value / 1.19
71        End Set
72    End Property
```

Fehlerliste
9 Fehler | 0 Warnungen | 0 Meldungen

	Beschreibung	Datei	Z	Spalte	Projekt
2	Syntaxfehler.	mdlMain.vb	16	9	Inventory
3	Die Anweisung kann nicht außerhalb einer Methode/eines mehrzeiligen lambda-Ausdrucks stehen.	mdlMain.vb	16	13	Inventory
4	"End Get" muss ein entsprechendes "Get" voranstehen.	mdlMain.vb	16	9	Inventory
5	Syntaxfehler.	mdlMain.vb	16	9	Inventory
6	Deklaration erwartet.	mdlMain.vb	17	13	Inventory
7	"End Set" muss ein entsprechendes "Set" voranstehen.	mdlMain.vb	17	9	Inventory
8	"End Property" muss ein entsprechendes "Property" voranstehen.	mdlMain.vb	17	5	Inventory
9	Die DVDItem-Klasse muss als MustInherit deklariert werden oder folgende geerbte MustOverride-Member überschreiben: Inventory.ShopItem : Public MustOverride Property GrossPrice As Double.	mdlMain.vb	26	7	Inventory

Fehlerliste | Ausgabe | Ausstehende Änderungen | Ergebnisse der Symbolsuche

Abbildung 11.11 Mit *MustOverride* als virtuell gekennzeichnete Prozeduren dürfen keinen Code enthalten und müssen in abgeleiteten Klassen überschrieben werden

Sie deklarieren eine Prozedur mit dem Schlüsselwort `MustOverride` als virtuell. Wenn Sie die vorhandene Eigenschaft `GrossPrice` der Klasse `ShopItem` folgendermaßen ändern:

```
Public MustOverride Property GrossPrice() as Double
```

bekommen Sie ein paar – in diesem Fall – »gewünschte« Fehlermeldungen, etwa wie in Abbildung 11.11 zu sehen. Ein Fehler liegt direkt in der Eigenschaftenprozedur, denn: Virtuelle Prozeduren selbst dürfen keinen Code enthalten. Sie dienen nur dazu, zu sagen: »Vergiss nicht, lieber Entwickler, du musst diese Prozedur in deiner abgeleiteten Klasse implementieren.« Und dass das gut funktioniert, zeigt die folgende Fehlermeldung an:

```
Die DVDItem-Klasse muss als MustInherit deklariert werden oder folgende geerbte MustOverride-Member
überschreiben:
    Inventory.ShopItem : Public MustOverride Property GrossPrice() As Double.
```

Denn es ist in der Tat richtig, dass die Klasse `DVDItem` diese Eigenschaft nicht implementiert. Stellen Sie sich vor, Videos und DVDs hätten einen speziellen Mehrwertsteuersatz, der vom normalen und reduzierten Steuersatz abweicht, sagen wir einmal von 21% – wie leicht hätte dem Entwickler »durchrutschen« können, die Eigenschaft neu zu implementieren! Die betroffenen Artikel wären in diesem Fall fälschlicherweise mit 19% MwSt. besteuert worden. Die Änderungen, die Sie am Programm durchführen müssen, halten sich in Grenzen:

- Sie entfernen den Code der Eigenschaftenprozedur `GrossPrice` in der Klasse `ShopItem`, sodass nur der Prototyp stehen bleibt (der Prozedurenkopf ohne `End Property`)
- Sie fügen eine Eigenschaftenprozedur `GrossPrice` in die Klasse `DVDItem` ein – im Wesentlichen genau den Code, der sich zuvor in der Klasse `ShopItem` befand (aber entsprechend angepasst, das heißt Sie ersetzen `MyNetPrice` durch `NetPrice`).

Danach sind alle Fehler eliminiert, und Ihr Programm ist bereit für die kommende Mehrwertsteuererhöhung, für Sondermehrwertsteuersätze und auch »Subventionsstreichungen«, etwa einen möglichen Wegfall des reduzierten Mehrwertsteuersatzes für Bücher. Natürlich hoffen wir, dass dies nicht eintritt ;-).

BEGLEITDATEIEN Die Begleitdateien zum Beispiel mit abstrakten Klassen finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 11\Inventory02
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Schnittstellen (Interfaces)

Das Konzept von Schnittstellen bietet Ihnen eine weitere Möglichkeit, Klassen zu standardisieren. Schnittstellen dienen dazu – ähnlich wie abstrakte Klassen –, Vorschriften zu erstellen, dass innerhalb der Klassen, die von einer Schnittstelle ableiten, bestimmte Elemente zwingend erforderlich sind. Im Gegensatz zu abstrakten Klassen enthalten Schnittstellen jedoch nur diese Vorschriften; sie enthalten überhaupt keinen Code. Und, das ist besonders interessant: Klassen können bei .NET (anders als in C++, wo auch noch eine oftmals verwirrende Mehrfachvererbung möglich ist) immer nur von einer Klasse erben, aber beliebig viele Schnittstellen imple-

mentieren. In der Tat kann man sich Schnittstellen unter .NET genau so vorstellen, wie wir den Begriff im Alltag verwenden: Eine Klasse – und damit ein Objekt –, zum Beispiel ein `Notebook` (das von der Basisklasse `Rechner` abgeleitet wurde), kann durchaus mehrere Schnittstellen implementieren. In unserem Beispiel kann die `Notebook`-Klasse die `Bluetooth`-, die `VGA`-, die `USB`- und weitere Schnittstellen implementieren.

Da Schnittstellen keinerlei Code enthalten, lassen sie sich, genau wie abstrakte Klassen, nicht direkt instanzieren. Allerdings können Objektvariablen vom Typ einer Schnittstelle dazu verwendet werden, ableitende Klassen zu referenzieren. Dieses Verhalten erlaubt es, Klassenbibliotheken zu erstellen, die eine enorme Flexibilität aufweisen.

Durch die Eigenarten, die Schnittstellen aufweisen, werden sie gerade innerhalb von .NET Framework zu zweierlei Zwecken verwendet:

- Sie dienen auf der einen Seite lediglich dazu, den Programmierer, der sie einbindet, an bestimmte Konventionen zu binden, ohne dass aber die Instanzen der Klassen, in die er die Schnittstellen einbindet, jemals von Schnittstellenvariablen direkt gesteuert würden. Man spricht in diesem Fall von einem so genannten »Interface Pattern« (*Schnittstellenmuster*).
- Sie dienen auf der anderen Seite dazu, allgemein gültige Verwalterklassen zu entwickeln, in der später alle die Objekte verwendet werden können, die die vorgegebenen Schnittstellen dafür implementieren. Um noch mal auf unser hypothetisches Beispiel zu sprechen zu kommen, das Sie im grauen Kasten ab Seite 390 beschrieben finden: Auch hier böte es sich an, die Objekte, die gezeichnet werden sollen, in Form von Schnittstellen zu beschreiben, sodass das Team, das die konkreten Objekte zum Zeichnen implementieren muss, eine *größtmögliche* Flexibilität bei der Implementierung hätte. Sie könnten dann sowohl codeenthaltene Basisklassen für die Objekte mit dem größten gemeinsamen Nenner entwerfen, könnten aber auch Sonderobjekte entwerfen, die lediglich die entsprechenden Schnittstellen einbinden, aber komplett von Grund auf neu entwickelt wären. Damit ließen sich auch sehr individuelle Implementierungen erreichen, die sich dennoch in das durch die Schnittstellen vorgegebene Objektmodell einfügen.

Für den letzten Punkt gibt es dazu häufig eine Art Drei-Stufen-Konzept für Komponenten, die dem Entwickler zur Verfügung gestellt werden: Auf oberster Ebene gibt es eine Klasse, die die eigentliche Aufgabe übernimmt. Sie kann Objekte einbinden, die eine bestimmte Schnittstelle implementieren. Auf unterster Ebene stellt die Komponente dem Entwickler eben diese Schnittstelle(n) zur Verfügung. Damit der Entwickler, der die Komponente verwenden will, nicht die komplette Implementierung selbst durchführen muss, sollte ihm die Komponente basierend auf den angebotenen Schnittstellen abstrakte Klassen zur Verfügung stellen, die eine gewisse Grundfunktionalität enthalten (die mittlere Ebene).

Ein weiteres Beispiel soll diese Zusammenhänge einmal mehr klären: Angenommen, Sie werden mit der Aufgabe betraut, eine Komponente zu entwickeln, die Daten tabellarisch auf dem Bildschirm darstellt. Diese Komponente – nennen wir sie *Tabellensteuerelement* – ist prinzipiell aus zwei Unterkomponenten aufgebaut: Zum einen ist das die Komponente, die die Tabelle zeichnet, sie mit Gitternetzlinien versieht, dafür sorgt, dass der spätere Benutzer eine Einfügemarke in der Tabelle bewegen kann usw. Diese Komponente wird aber idealerweise nicht selbst dafür zuständig sein, den Inhalt einer Tabellenzelle zu malen, sondern diese Aufgabe einer weiteren Klasse überlassen und bindet sie lediglich ein. Die Aufgabe der weiteren Komponente auf der anderen Seite ist es, den Inhalt einer einzigen Tabellenzelle zu zeichnen. Diese zweite Komponente speichert also nicht nur die Daten für die einzelnen Tabellenzellen, sondern sie sorgt auch dafür, dass diese Daten zu gegebener Zeit in Form einer Tabellenzelle auf den Bildschirm gemalt werden. Damit der spätere Entwickler, der das Tabellensteuerelement verwendet, die größtmögliche Flexibilität in seiner Verwendung hat, sollte die zweite Klasse – die Tabellenzelle – nach Möglichkeit nicht als festgeschriebene Klasse, sondern

auf unterster Ebene auch als Schnittstelle zur Verfügung stellen. Auf der zweiten Ebene sollte das Tabellensteuerelement dem Entwickler auch mindestens eine auf der Schnittstelle basierende abstrakte Klasse zur Verfügung stellen, die die wichtigste Grundfunktionalität bereits enthält. Und schließlich stellt die Komponente auf oberster Ebene fix und fertige Tabellenzellenkomponenten zur Verfügung, mit denen der Entwickler direkt loslegen und Tabellen mit Daten füllen kann.

Stellt der Entwickler dann nach geraumer Zeit fest, dass die vorhandenen Zellenkomponenten nicht die Möglichkeiten bieten, die er benötigt, kann er sich entscheiden:

- Er kann entweder die abstrakte Klasse ableiten und daraus eine Tabellenzellenkomponente entwickeln, die seinen Anforderungen genügt. Der Aufwand dafür hält sich in Grenzen, weil die abstrakte Klasse einen Großteil des notwendigen Verwaltungscodes bereits zur Verfügung stellt, den er lediglich ergänzen muss.
- Falls ihn diese Vorgehensweise immer noch zu sehr einschränkt, kann er die *vollständige* Implementierung der Zellenklasse übernehmen. Er muss in diesem Fall die von der Tabellenkomponente zur Verfügung gestellte Schnittstelle einbinden, um sicherzustellen, dass alles an Funktionalität innerhalb seiner Zellenklasse vorhanden ist, was die Tabellenkomponente vorschreibt. Der Nachteil: Der Aufwand, dieses Vorhaben zu realisieren, ist logischerweise erheblich größer.

Das bislang verwendete Beispiel eignet sich ebenfalls, die Verwendung von Schnittstellen zu demonstrieren. Dazu wird eine Schnittstelle implementiert, die die Grundfunktionen vorschreibt, auf die das Hauptprogramm bislang über die Klasse `ShopItem` Zugriff hatte. Die Schnittstelle für das Beispielprogramm soll den Namen `IShopItem` bekommen (das Voranstellen des Buchstabens »I« für *Interface* ist eine gängige Konvention für die Benennung einer Schnittstellenklasse), und die Implementierung geschieht im Beispielprojekt noch vor dem Hauptmodul:

```
Interface IShopItem
    Property PrintTypeSetting() As PrintType
    ReadOnly Property ItemDescription() As String
    Function ToString() As String
    Property GrossPrice() As Double
End Interface
```

Als Nächstes kümmern wir uns um das Hauptprogramm, das zur Steuerung jetzt nicht mehr die abstrakte Artikelbasisklasse, sondern die neue Schnittstelle verwenden soll. Dazu ist eine einzige Änderung notwendig, die Deklaration der abstrakten Artikelbasisklasse:

```
Sub Main()
    'Streamreader zum Einlesen der Textdateien
    Dim locSr As StreamReader
    Dim locList As New DynamicList  ' Die dynamische Liste
    Dim locElements() As String     ' Die einzelnen ShopItem-Elemente
    Dim locShopItem As IShopItem    ' Ein einzelnes Shop-Element
    Dim locDisplayMode As PrintType ' Der Darstellungsmodus
```

Da die Schnittstelle `IShopItem` alle Eigenschaften der ursprünglichen abstrakten Basisklasse `ShopItem` enthält, sind keine weiteren Änderungen erforderlich. Dennoch hat das Verändern der einen Zeile enorme Auswirkungen, und es hagelt geradezu Fehler (Abbildung 11.12):

```
If locElements(FieldOrder.Type) = "1" Then
    locShopItem = New BookItem(locElements)
Else
    locShopItem = New DVDItem(locElements)
End If
```

Fehlerliste

16 Fehler | 0 Warnungen | 0 Meldungen

	Beschreibung	Datei	Z	Spalte	Projekt
1	"Option Strict On" lässt keine impliziten Konvertierungen von Inventory.BookItem in Inventory.IShopItem zu.	mdlMain.vb	64	39	Inventory
2	"Option Strict On" lässt keine impliziten Konvertierungen von Inventory.DVDItem in Inventory.IShopItem zu.	mdlMain.vb	66	39	Inventory

Abbildung 11.12 Da zum jetzigen Zeitpunkt noch keine der abgeleiteten Klassen die Schnittstelle implementiert, ist ein Referenzieren der verwendeten Klassen noch nicht möglich

Das Hauptprogramm kann zwar nun alle Objekte verarbeiten, die die neue Schnittstelle implementiert haben. Da aber bislang keine der Klassen die Schnittstellen implementiert, ist das Programm zu diesem Zeitpunkt nicht lauffähig. Visual Basic beschwert sich zu diesem Zeitpunkt mit einem nicht so klar verständlichen Fehler: Da es keinen Zusammenhang zwischen `locShopItem` (das als `IShopItem` deklariert ist) und den Klassen `BookItem` und `DVDItem` herstellen kann, macht es das Nächstliegende:

Es versucht, die Typen implizit zu konvertieren. Da ein implizites Konvertieren von einem Typ zum anderen durch `Option Strict On` (als global gültig in den Projekteigenschaften eingestellt) unterbunden ist, liefert es eine entsprechende Fehlermeldung, die zunächst ein wenig in die Irre führen kann.

Nun leiten sich aber alle Klassen, die wir im Programm tatsächlich verwenden, von der bisherigen Klasse `ShopItem` ab. Die Implementierung der Schnittstelle `IShopItem` in der Klasse `ShopItem` bewirkt, dass anschließend auch alle von `ShopItem` abgeleiteten Klassen automatisch `IShopItem` implementieren. Es reicht also aus, wenn Sie alle geforderten Prozeduren in der abstrakten Basisklasse einbauen, damit das Programm anschließend wieder lauffähig ist.

Bei der Implementierung einer Schnittstelle in eine Klasse müssen Sie in Visual Basic auf zwei Sachen achten:

- Sie geben bei der Definition der Klasse mit `Implements` an, welche Schnittstelle implementiert werden soll
- Sie bestimmen für jede betroffene Prozedur der Klasse individuell, welche Schnittstellenprozedur sie implementieren soll. Auch das geschieht mit dem Schlüsselwort `Implements`.

Um die `Implements`-Anweisung hinzuzufügen, geben Sie unterhalb der Zeile `MustInherit Class ShopItem` **Implements IShopItem** ein, drücken aber zunächst NICHT [↵], sondern verlassen die Zeile mit einer der Pfeiltasten.

Sobald Sie das `Implements`-Schlüsselwort der Klassendefinition auf diese Weise hinzugefügt haben, verschwinden zwar die bisherigen Fehlermeldungen. Allerdings werden diese in der nächsten Sekunde durch andere ersetzt, wie in Abbildung 11.13 zu sehen.

```
MustInherit Class ShopItem
    Implements IShopItem

    Private myTitle As String                  ' Titel
    Private myNetPrice As Double               ' Nettopreis
    Private myOrderNumber As String            ' Artikelnummer
    Private myPrintTypeSetting As PrintType    ' Ausgabeform

    Public Sub New()
        myPrintTypeSetting = PrintType.Detailed
    End Sub
```

Fehlerliste

8 Fehler | 0 Warnungen | 0 Meldungen

	Beschreibung	Datei	Z	Spalte	Projekt
1	Class "ShopItem" muss "Function ToString() As String" für die IShopItem-Schnittstelle implementieren.	mdlMain.vb	12	16	Inventory
2	Class "ShopItem" muss "Property GrossPrice As Double" für die IShopItem-Schnittstelle implementieren. Die implementierende Eigenschaft muss übereinstimmende ReadOnly- oder WriteOnly-Spezifizierer aufweisen.	mdlMain.vb	12	16	Inventory
3	Class "ShopItem" muss "Property PrintTypeSetting As PrintType" für die IShopItem-Schnittstelle implementieren. Die implementierende Eigenschaft muss übereinstimmende ReadOnly- oder WriteOnly-Spezifizierer aufweisen.	mdlMain.vb	12	16	Inventory
4	Class "ShopItem" muss "ReadOnly Property ItemDescription As String" für die IShopItem-Schnittstelle implementieren. Die implementierende Eigenschaft muss übereinstimmende ReadOnly- oder WriteOnly-Spezifizierer aufweisen.	mdlMain.vb	12	16	Inventory
5	property "GrossPrice" kann nicht als "Overrides" deklariert werden, da es property in Basis class nicht überschreibt.	mdlMain.vb	24	31	Inventory
6	property "Description" kann nicht als "Overrides" deklariert werden, da es property in Basis class nicht überschreibt.	mdlMain.vb	25	40	Inventory
7	property "Description" kann nicht als "Overrides" deklariert werden, da es property in Basis class nicht überschreibt.	mdlMain.vb	29	40	Inventory
8	property "GrossPrice" kann nicht als "Overrides" deklariert werden, da es property in Basis class nicht überschreibt.	mdlMain.vb	29	31	Inventory

Abbildung 11.13 Wenn Sie eine Schnittstelle implementieren, müssen Sie auch für die Implementierung der einzelnen Prozeduren der Schnittstelle sorgen

Jetzt fragen Sie sich möglicherweise, wieso es nicht ausreicht, der Prozedur in der Klasse denselben Namen wie den der Schnittstellenprozeduren zu geben – beim Ableiten von Klassen genügt es schließlich auch.

In C# beispielsweise funktioniert das übrigens ohne Probleme, die Frage ist also berechtigt. Sie haben in C# aber auch nicht wie in Visual Basic die Möglichkeit, einen ganz anderen Namen in Ihrer Klassenprozedur zu definieren, der eben nicht dem Namen der Schnittstellenprozedur entspricht. Insofern bringt Ihnen dieses Feature von Visual Basic schon eine größere Flexibilität. Dazu kommt, dass die Microsoft Intermediate Language, in die jede .NET-Anwendung zunächst kompiliert wird, ebenfalls so konzipiert ist, dass sie die explizite Angabe der zu implementierenden Schnittstellenprozedur erfordert – in C# erledigt diese Umsetzung der Compiler zwar implizit, aber auch eben nur ausschließlich implizit.

Außerdem genießen Sie in Visual Basic dank IntelliSense einen Vorteil in Form einer Eingabehilfe: Sobald Sie am Ende einer Prozedur, der Sie ein Schnittstellenelement zuweisen wollen, das `Implements`-Schlüsselwort eingegeben haben, bietet Ihnen IntelliSense die möglichen Implementierungen zur Auswahl in einer Liste an (Abbildung 11.14).

Abbildung 11.14 IntelliSense unterstützt Sie bei der Auswahl der richtigen Schnittstellenimplementierung

Um die Beispielanwendung wieder zum Laufen zu bewegen, müssen Sie die folgenden Prozeduren bearbeiten und ihnen die richtigen Schnittstellenprozeduren zuweisen:

```
Public MustOverride Property GrossPrice() As Double Implements IShopItem.GrossPrice
.
.
.
Public Overridable ReadOnly Property Description() As String Implements IShopItem.ItemDescription
    Get
        Return OrderNumber & vbTab & Title
    End Get
End Property

Public Property PrintTypeSetting() As PrintType Implements IShopItem.PrintTypeSetting
    Get
        Return myPrintTypeSetting
    End Get

    Set(ByVal Value As PrintType)
        myPrintTypeSetting = Value
    End Set
End Property

Public Overrides Function ToString() As String Implements IShopItem.ToString
    If PrintTypeSetting = PrintType.Brief Then
        'Kurzform: Es wird die Description-Eigenschaft des Objekts gemäß
        'ursprünglicher Implementierung in der Basisklasse verwendet
        Return MyClass.Description & vbCr & vbLf & _
            Me.NetPriceFormatted & vbTab & Me.VATAmountFormatted & vbTab & _
            Me.GrossPriceFormatted & vbCr & vbLf
    Else
        'Langform: Die überschriebene, erweiterte Description-Eigenschaft
        'des Objekts selbst wird verwendet
        Return Me.Description & vbCr & vbLf & _
            Me.NetPriceFormatted & vbTab & Me.VATAmountFormatted & vbTab & _
            Me.GrossPriceFormatted & vbCr & vbLf
    End If
End Function
```

HINWEIS Die zweite Eigenschaft verdient hier besonderes Interesse, denn sie bindet eine Schnittstellenprozedur mit einem abweichenden Namen ein. Sie sehen also, dass Namen an dieser Stelle nur Schall und Rauch sind. Erst mit `Implements` wird festgelegt, welches Schnittstellenelement welchem Klassenelement zugeordnet werden soll. Dabei haben Sie natürlich nicht komplett freie Hand: Sie können nicht – wie anzunehmen ist – eine in einer Schnittstelle definierte Eigenschaft als Klassenmethode, sondern eben auch nur als Eigenschaft implementieren. Auch ist es nicht möglich, eine Methode einer Schnittstellendefinition, die zum Beispiel als Parameter einen `String` entgegennimmt, einer Methode einer Klasse zuordnen, die einen `Integer` als Parameter erwartet. Und auch die Rückgabetypen müssen entsprechend zwischen Schnittstelle und einbindender Klasse identisch sein. Fehlzuweisungen in dieser Form würden das Common Type System von .NET grob verletzen, und deswegen spielt auch schon der Hintergrundcompiler nicht mit, wenn Sie einen solchen Versuch unternehmen, wie Abbildung 11.15 zeigt.

Abbildung 11.15 Signaturen, Elementtyp (Eigenschaft, Methode, Ereignis) und Rückgabetypen in einer Schnittstellendefinition müssen denen in der einbindenden Klasse entsprechen

Nachdem Sie diese Änderungen durchgeführt haben, werden Sie nur noch einige »Fehler« in der Klasse `DynamicList` finden. Diese Klasse akzeptierte bisher lediglich `ShopItem`-Objekte in den Methoden `Add` und `Item`. Wenn Sie die verwendeten `ShopItem`-Objekte durch solche vom Typ `IShopItem` ersetzen, ist das Programm wieder lauffähig (die geänderten Stellen sind wieder fett hervorgehoben):

```
Class DynamicList

    private myStepIncreaser As Integer = 4
    private myCurrentArraySize As Integer
    private myCurrentCounter As Integer
    private myArray() As IShopItem

    Sub New()
        myCurrentArraySize = myStepIncreaser
        ReDim myArray(myCurrentArraySize)
    End Sub

    Sub Add(ByVal Item As IShopItem)

        'Prüfen, ob aktuelle Arraygrenze erreicht wurde.
        If myCurrentCounter = myCurrentArraySize - 1 Then
            'Neues Array mit mehr Speicher anlegen,
            'und Elemente hinüberkopieren. Dazu:

            'Neues Array wird größer:
            myCurrentArraySize += myStepIncreaser

            'Temporäres Array erstellen.
```

```
            Dim locTempArray(myCurrentArraySize - 1) As IShopItem

            'Elemente kopieren.
            'Wichtig: Um das Kopieren müssen Sie sich,
            'anders als bei VB6, selber kümmern!
            For locCount As Integer = 0 To myCurrentCounter
                locTempArray(locCount) = myArray(locCount)
            Next

            'Temporäres Array dem Memberarray zuweisen.
            myArray = locTempArray
        End If

        'Element im Array speichern
        myArray(myCurrentCounter) = Item

        'Zeiger auf nächstes Element erhöhen.
        myCurrentCounter += 1

    End Sub

    'Liefert die Anzahl der vorhandenen Elemente zurück.
    Public Overridable ReadOnly Property Count() As Integer
        Get
            Return myCurrentCounter
        End Get
    End Property

    'Erlaubt das Zuweisen.
    Default Public Overridable Property Item(ByVal Index As Integer) As IShopItem
        Get
            Return myArray(Index)
        End Get
        Set(ByVal Value As IShopItem)
            myArray(Index) = Value
        End Set
    End Property
End Class
```

BEGLEITDATEIEN Das Programmbeispiel mit den Änderungen für Schnittstellen finden Sie übrigens im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 11\Inventory03
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Unterstützung bei abstrakten Klassen und Schnittstellen durch den Editor

Ihnen ist sicherlich aufgefallen, dass Sie vorhin angewiesen wurden, die Änderung für die Schnittstellenimplementierung durch `Implements` in der Klasse `ShopItem` nicht mit ↵ zu bestätigen, sondern die Zeile mit einer der Pfeiltasten zu verlassen.

Hätten Sie nämlich die Zeile mit ↵ verlassen, dann wäre die Editorunterstützung für das Implementieren der Funktionsrümpfe angesprungen. In unserem Fall, in dem die Routinen aber schon komplett vorhanden waren, wäre das allerdings eher hinderlich gewesen, wie die beiden folgenden Beispiele zeigen.

Zu diesem Zweck erstellen Sie ein neues Visual Basic-Konsolenprojekt unter einem beliebigen Namen. Implementieren Sie anschließend eine Schnittstelle und eine Klasse nach folgender Vorlage, die Sie oberhalb der Zeile `Module Module1` in der Codedatei *Module1.vb* erfassen:

```
Interface ITest
    Property EineEigenschaft() As String
    Function EineMethode() As Integer
End Interface

Public Class BindetITestEin

End Class
```

Anschließend bewegen Sie die Einfügemarke unter die Zeile `Public Class BindetITestEin`, geben **Implements ITest** ein und drücken anschließend ↵.

Abbildung 11.16 Nachdem Sie mit `Implements` eine Schnittstelle eingebunden haben, fügt der Editor die kompletten Coderümpfe der benötigten Elemente ein

Diese Vorgehensweise funktioniert auch ergänzend, und das heißt: Sie können eine Methode, eine Eigenschaft oder ein Ereignis (zu Ereignissen später mehr) in der Schnittstellendefinition hinzufügen, bekommen aber dann natürlich zunächst eine Fehlermeldung, weil dieses neue Element nicht in der implementierenden Klasse vorhanden ist. Bewegen Sie die Einfügemarke allerdings anschließend hinter den Schnittstellennamen der `Implements`-Anweisung unterhalb der Klassendefinition und drücken ↵, wird der Elementcoderumpf wieder an das Ende der Klasse angehängt. Probieren Sie es aus:

- Fügen Sie in der Schnittstelle `ITest` eine weitere Methode vom Typ `String` namens `ZweiteMethode` ein, sodass sich folgender Code ergibt:

```
Interface ITest
    Property EineEigenschaft() As String
    Function EineMethode() As Integer
    Function ZweiteMethode() As String
End Interface
```

  Anschließend sehen Sie in der Fehlerliste eine Fehlermeldung mit dem Hinweis darauf, dass diese neue Methode nicht in der Klasse implementiert wurde, die die Schnittstelle einbindet.

- Um diesen Fehler zu beheben, reicht es aus, die Einfügemarke unter die Zeile `Public Class BindetITestEin` hinter `Implements ITest` zu bewegen und ↵ zu betätigen.

In diesem Moment verschwindet der Fehler, da der Editor den Coderumpf für die Methode `ZweiteMethode` einfügt, sie mit der Schnittstelle per `Implements` verbindet und damit den Implementierungsvorschriften der Schnittstelle Genüge tut.

Die Tücken der automatischen Codeergänzung bei Schnittstellen oder abstrakten Klassen

Allerdings funktioniert die Unterstützung durch den Editor bisweilen nicht immer so reibungslos und kann zu – na ja, nennen wir es – »Orientierungsproblemen« führen, wenn es bereits Methoden oder Eigenschaften der Basisklasse gibt, die genau so heißen wie die zu implementierenden Methoden oder Eigenschaften, aber noch nicht mit diesen verknüpft sind. Um das nachzustellen, entfernen Sie hinter der Funktion

```
    Public Function ZweiteMethode() As String Implements ITest.ZweiteMethode

    End Function
```

die Schnittstellenimplementierung `Implements ITest.ZweiteMethode`. Daraufhin unterschlängelt der Editor die Zeile

```
    Implements ITest
```

blau, da, wie er richtig erkennt, die Methode `ZweiteMethode` der Schnittstelle `ITest` nicht mehr korrekt implementiert ist. Doch die Editorunterstützung versagt anschließend ein wenig. Denn wenn Sie nun abermals die Einfügemarke unter die Zeile `Public Class BindetITestEin` hinter `Implements ITest` bewegen und ↵ betätigen, erscheint am Ende der Klasse ein Funktionsrumpf, wie Sie ihn auch in Abbildung 11.17 erkennen können.

```
Public Class BindetITestEin
    Implements ITest

    Public Property EineEigenschaft() As String Implements ITest.EineEigenschaft
        Get

        End Get
        Set(ByVal value As String)

        End Set
    End Property

    Public Function EineMethode() As Integer Implements ITest.EineMethode

    End Function

    Public Function ZweiteMethode() As String

    End Function

    Public Function ZweiteMethode1() As String Implements ITest.ZweiteMethode

    End Function

End Class
```

Abbildung 11.17 Beim automatischen Komplettieren von Coderümpfen von Elementen, die mit gleichem Namen schon vor-handen sind, wird ein »ähnlicher« Elementname erfunden

Die Methode `ZweiteMethode` gibt es nun (mehr oder weniger) zweimal – einmal mit nachgestellter »1«. Anstelle also hinter `ZweiteMethode` korrekterweise die `Implements`-Anweisung mit dem entsprechenden Schnittstellenelementnamen zu ergänzen, kreiert der Editor eine ganz neue Methode und implementiert das Schnittstellenelement in dieser.

Falls Ihnen solche Dinge beim Entwickeln Ihrer eigenen Schnittstellenkreationen passieren, müssen Sie die `Implements`-Anweisung zur eigentlich gemeinten (und schon vorhandenen) Methode übertragen und die vom Editor »erfundene« Methode löschen.

Noch diffuser wird es, wenn es um die Implementierung einer Methode oder Eigenschaft geht, die nur aus der Basisklasse hervorgeht. `ToString` ist standardmäßig solch ein Kandidat. Dabei ist `ToString` in jeder neu erstellten Klasse enthalten, da sie eine Methode von `Object` darstellt, und jede neue Klasse wird, wie wir ja schon wissen, implizit von `Object` abgeleitet, wenn nichts anderes gesagt wird. Damit erbt natürlich auch jede neue Klasse die `ToString`-Methode (warum die `ToString`-Methode an `Object` »hängt«, klärt übrigens der Abschnitt »Die Methoden und Eigenschaften von Object« ab Seite 427).

Nun achten Sie darauf, was passiert, wenn wir der Schnittstelle exakt diese Methode hinzufügen. Erwartungsgemäß sehen wir zunächst einen Fehler im Programm, da die neue Schnittstellendefinition verlangt, dass wir eine `ToString`-Funktion implementieren. Das machen wir durch das standardmäßige Vorhandensein von `ToString` ja auch ohne weiteres Zutun; was allerdings fehlt, ist, unsere (geerbte) `ToString`-Funktion mit der Schnittstellenmethodendefinition `ToString` per `Implements` (à la Abbildung 11.14) zu verheiraten.

Mit `Implements` würde das auch nicht so ohne Weiteres funktionieren, denn es gibt ja schließlich überhaupt keinen Methodencode, hinter den wir `Implements` für die Zuweisung des Schnittstellen-`ToString` hängen könnten.

Ergründen wir also zunächst, wie uns der Editor »hilft«, und wie wir anschließend wirklich zum Ziel kommen:

- Fügen Sie in der Schnittstelle `ITest` eine weitere Methode vom Typ `String` namens `ToString` ein, sodass sich folgender Code ergibt:

```
Interface ITest
    Property EineEigenschaft() As String
    Function EineMethode() As Integer
    Function ZweiteMethode() As String
    Function ToString() As String
End Interface
```

  Anschließend sehen Sie wieder in der Fehlerliste eine Fehlermeldung mit dem Hinweis darauf, dass diese neue Methode nicht in der Klasse implementiert wurde, die die Schnittstelle einbindet.

- Versuchen wir den Fehler auf die gleiche Weise mit der Editorunterstützung zu beheben: Platzieren Sie die Einfügemarke wieder unterhalb der Zeile `Public Class BindetITestEin` hinter `Implements ITest`, und betätigen Sie ↵.

Das Ergebnis ist wieder dasselbe wie im vorherigen Beispiel, nur leider nicht so einfach zu durchschauen, da es in der Klasse ja keinen Code für `ToString` gibt (denn dieser wurde ja aus der Basisklasse `Object` übernommen).

```
    Public Property EineEigenschaft() As String Implements ITest.EineEigenschaft
        Get

        End Get
        Set(ByVal value As String)

        End Set
    End Property

    Public Function EineMethode() As Integer Implements ITest.EineMethode

    End Function

    Public Function ZweiteMethode() As String Implements ITest.ZweiteMethode

    End Function

    Public Function ToString1() As String Implements ITest.ToString

    End Function
End Class
```

Abbildung 11.18 Verwirrender ist das Vorgehen des Editors bei der Codeergänzung, wenn es eine geerbte Methode oder Eigenschaft mit gleichem Namen wie in der Schnittstellendefinition gibt

Prinzipiell hat der Editor das gleiche Problem. Es gibt bereits eine Methode (`ToString`), nur ist sie dieses Mal nicht sichtbar, denn sie wurde aus der Basisklasse (`Object` in diesem Beispiel) übernommen. Also kreiert der Editor wieder eine Abwandlung des Funktionsnamens und nennt die Methode wie die eigentliche, hängt aber wieder eine »1« am Ende an.

Diesen Fehler können Sie nur durch folgende Schritte beheben:

- Sie überschreiben die Methode bzw. Eigenschaft, die es zu implementieren gilt
- Sie rufen innerhalb der Methode oder Eigenschaft nichts weiter auf als die Basismethode (oder Basiseigenschaft)
- Sie implementieren die Schnittstelle an der überschriebenen Methode bzw. Eigenschaft

Der Code dafür würde in unserem Fall folgendermaßen aussehen:

```
Public Overrides Function ToString() As String Implements ITest.ToString
    Return MyBase.ToString
End Function
```

Editorunterstützung bei abstrakten Klassen

Übrigens: Die Unterstützung, die Sie bei der Implementierung von Schnittstellen erfahren, gibt es auch bei abstrakten Klassen. Ergänzen Sie zur Demonstration das Beispiel um folgende abstrakte Klasse:

```
MustInherit Class AbstractTest
    Public MustOverride Property EineEigenschaft() As String
    Public MustOverride Function EineMethode(ByVal EinParameter As String) As String
End Class
```

Erstellen Sie nun eine Klasse, die auf `AbstractTest` basiert:

```
Public Class BasiertAufAbstractTest
    Inherits AbstractTest

End Class
```

In dem Moment, in dem Sie hinter `Inherits AbstractTest` ↵ betätigen, komplettiert der Editor für Sie automatisch alle Methoden und Eigenschaften in Form von Coderümpfen, die mit `MustOverride` als virtuelle Methoden bzw. Eigenschaften gekennzeichnet wurden:

```
Public Class BasiertAufAbstractClass
    Inherits AbstractTest


    Public Overrides Property EineEigenschaft As String
        Get

        End Get
        Set(ByVal value As String)

        End Set
    End Property
```

```
    Public Overrides Function EineMethode(ByVal EinParameter As String) As String

    End Function
End Class
```

Schnittstellen, die Schnittstellen implementieren

Auch Schnittstellen können vom Prinzip her Schnittstellen implementieren. Allerdings tun sie das nicht auf die gleiche Weise, wie Klassen Schnittstellen implementieren. Stattdessen sind Schnittstellen in der Lage, von anderen Schnittstellen zu erben – genau wie Klassen von anderen Klassen erben können und dabei deren komplette Funktionalität übernehmen.

Da es aber bei Schnittstellen keine Funktionalitäten, sondern nur virtuelle Methoden, Eigenschaften oder Ereignisse gibt, erben Schnittstellen von anderen Schnittstellen nur deren Implementierungsvorschriften. Oder anders und einfacher ausgedrückt: Eine Schnittstelle, die von einer anderen Schnittstelle erbt, enthält sämtliche der eigenen Vorschriften und die der Schnittstelle, von der sie erbt. Und darüber hinaus können Schnittstellen auch nur von Schnittstellen erben, nicht von Klassen (was aber wahrscheinlich auch ohne explizite Erwähnung klar ist, denn Schnittstellen dürfen ja anders als Klassen keinen Code aufweisen).

Sie können sich auch diese Zusammenhänge an einem Beispiel verdeutlichen. Ergänzen Sie das Modul aus dem vorherigen Beispiel um folgenden Code:

```
Interface IBaseInterface
    Property EineEigenschaft() As String
End Interface

Interface IMoreComplexInterface
    Inherits IBaseInterface

    Property ZweiteEigenschaft() As String
End Interface
```

Die Schnittstelle `IMoreComplexInterface` beinhaltet in diesem Fall die Vorschriften beider Schnittstellendefinitionen (der eigenen und der, von der sie erbt).

Wenn Sie diese »vereinigte« Schnittstelle in einer Klasse implementieren, ergibt sich – dank Codeergänzung durch den Editor ist das ja schnell getan – nach dem Betätigen von ↵ hinter `Implements IMoreComplexInterface` folgender Code:

```
Public Class ComplexClass
    Implements IMoreComplexInterface

    Public Property EineEigenschaft() As String Implements IBaseInterface.EineEigenschaft
        Get

        End Get
        Set(ByVal value As String)

        End Set
    End Property
```

```
    Public Property ZweiteEigenschaft() As String Implements IMoreComplexInterface.ZweiteEigenschaft
        Get

        End Get
        Set(ByVal value As String)

        End Set
    End Property
End Class
```

Einbinden mehrerer Schnittstellen in eine Klasse

Mehrfachvererbung ist in der .NET-Infrastruktur nicht vorgesehen. Bei der Mehrfachvererbung entsteht eine neue Klasse aus mehr als einer Basisklasse und dabei übernimmt die erbende Klasse die Funktionalität von beiden Basisklassen.

Allerdings können Sie auch in Visual Basic mehr als eine Schnittstelle in einer Klasse implementieren, und auch wenn das nicht so »toll«[6] wie Mehrfachvererbung ist, so können Sie sich mit der Einbindung mehrerer Schnittstellen zumindest ein wenig behelfen. Man könnte das Einbinden mehrerer Schnittstellen in einer Klasse sozusagen als »Mehrfachvererbung Light« bezeichnen.

Das Einbinden mehrerer Schnittstellen ist ein Kinderspiel. Das vorherige Beispiel der Schnittstellendefinitionen ändern wir dazu folgendermaßen ab:

```
Interface IBaseInterface
    Property EineEigenschaft() As String
End Interface

Interface IMoreComplexInterface
    'Inherits wurde entfernt, beide Schnittstellen liegen somit auf "gleicher Ebene".
    Property ZweiteEigenschaft() As String
End Interface

Public Class ComplexClass
    Implements IBaseInterface, IMoreComplexInterface

    Public Property EineEigenschaft() As String Implements IBaseInterface.EineEigenschaft
        Get

        End Get
        Set(ByVal value As String)

        End Set
```

[6] Mehrfachvererbung ist nur auf den ersten Blick etwas Feines, und es wird sie in Visual Basic sehr wahrscheinlich nie geben, da selbst einer der Erfinder von C++, der Herr Stroustrup, das inzwischen sehr skeptisch sieht. (In Java ist es ja auch *absichtlich* weggelassen worden.) Mehrfachvererbung sorgt sogar für unentscheidbare Konflikte, die der Compiler willkürlich auflösen muss (zwei Methoden gleichen Namens in beiden Klassen mit Implementierung usw.). Stroustrup dazu: »[…] Mehrfachvererbung bleibt verständlich, wenn nur eine Basisklasse wirklich Memberfunktionen implementiert und die anderen nur pure virtuelle Funktionen deklarieren […]«.

```
    End Property

    Public Property ZweiteEigenschaft() As String Implements IMoreComplexInterface.ZweiteEigenschaft
        Get

        End Get
        Set(ByVal value As String)

        End Set
    End Property
End Class
```

Am Code der einbindenden Klasse muss in diesem Fall nur die in Fettschrift gesetzte Zeile geändert werden. An dem restlichen Code, der die Schnittstellenelemente den Klassenelementen zuordnet, müssen keinerlei Änderungen vorgenommen werden.

Die Methoden und Eigenschaften von Object

`Object` selbst stellt einige Grundmethoden und -eigenschaften zur Verfügung, die damit jede neue Klasse automatisch erbt. Der Hintergrund ist, dass die Entwickler von .NET Framework damit sichergestellt haben, dass jedes Objekt über eine gewisse Grundfunktionalität verfügt, auf deren Vorhandensein sich andere Klassen hundertprozentig verlassen können.

Ausgeben der String-Repräsentation eines Objekts mit ToString

Die `ToString`-Methode haben Sie im Laufe dieses Kapitels schon kennen gelernt. `ToString` ergibt die Textrepräsentation der Instanz einer Klasse. Primitive Variablen nutzen beispielsweise die `ToString`-Methode, um ihre Werte in Zeichenketten umzuwandeln – teilweise sogar mit entsprechenden Formatierungsoptionen. Sie wissen auch, dass die `ToString`-Methode der erste Anlaufpunkt für die Implementierung von einfachem polymorphen Verhalten ist, bzw. um solches zu erreichen. So brauchen Sie die `ToString`-Methode einer Klasse lediglich zu überschreiben, um es der Klasse selbst zu überlassen, wie ihre Textrepräsentation beispielsweise in einer ListBox aussehen soll. Der Abschnitt »Ein erstes einfaches Beispiel zur Polymorphie« ab Seite 392 ist darauf bereits im Beispiel eingegangen. In der Standardimplementierung liefert die `ToString`-Methode übrigens nur den Namen der Klasse, also den Typnamen zurück.

Prüfen auf Gleichheit von Objekten mit Object.Equals oder dem Is/IsNot-Operator

Die deutsche Sprache wird immer mehr vereinfacht. Man sieht das an Wörtern wie beispielsweise »das Gleiche« und »dasselbe«. Inzwischen sind diese laut Duden dasselbe (oder das Gleiche?) – aber das war nicht immer so. *Dasselbe* bezeichnete einst buchstäblich ein und dasselbe. Zwei verschiedene Leute konnten nicht gleichzeitig dasselbe Auto steuern – es sei denn, sie hätten es in der Mitte durchgeschnitten (und irgendwie fahrbereit gemacht) oder es wäre ein Fahrlehrer mit seinem Schüler, was einfacher wäre. Wohl aber das Gleiche: Dann wären es unterschiedliche Autos gewesen, die einfach nur baugleich waren. Diesen Unterschied sollten Sie für das bessere Verständnis der folgenden Erklärung kennen.

Es gibt für jedes Objekt eine Methode namens `Equals`, die überprüft, ob es sich bei einer Objektinstanz, die durch eine Objektvariable referenziert wird, um dieselbe handelt, die durch eine andere Objektvariable referenziert wird.

WICHTIG Es wird dabei NICHT überprüft, ob der Inhalt zweier Objekte der gleiche ist, falls die Objektvariablen auf zwei *verschiedene* Instanzen verweisen würden.

Ein Beispiel soll das verdeutlichen, und zum besseren Verständnis sollten Sie zuvor nochmals einen Blick auf Abbildung 9.7 in Kapitel 9 werfen.

HINWEIS Pedantisch betrachtet müsste die `Equals`-Methode eigentlich `IsSame` heißen, denn wollte man die Ausdrücke »das Gleiche« und »dasselbe« ins Englische übersetzen, würde wohl am ehesten »*the equal*« für Ersteres und »*the same*« für Letzteres dabei herauskommen. Statt die `Equals`-Methode zu verwenden, der Sie ein anderes Objekt zum Vergleichen übergeben können, haben Sie aber auch die Möglichkeit, den `Is`-Operator (bzw. den `IsNot`-Operator) einzusetzen – mit identischem Ergebnis. Das wiederum kommt der natürlichen Sprache schon näher. Das folgende Beispiel demonstriert den Einsatz beider Möglichkeiten.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 11\EqualsDemo
```

```
Module mdlMain

    Sub Main()

        'Instanziieren mit New und dadurch
        'Speicher für das Kontakt-Objekt
        'auf dem Managed Heap anlegen
        Dim objVarKontakt As New Kontakt With {
            .Nachname = "Halek",
            .Vorname = "Sarah",
            .Plz = "99999",
            .Ort = "Musterhausen"}

        'objVarKontakt2 zeigt auf dasselbe Objekt;
        'die referenzierte Instanz ist dieselbe!
        Dim objVarKontakt2 As Kontakt
        objVarKontakt2 = objVarKontakt

        'objVarKontakt3 zeigt auf ein gleiches Objekt;
        'die referenzierte Instanz ist nicht dieselbe, nur die gleiche!
        Dim objVarKontakt3 As New Kontakt With {
                        .Nachname = "Halek",
                        .Vorname = "Sarah",
                        .Plz = "99999",
                        .Ort = "Musterhausen"}

        'Der Beweis dafür:
        Console.WriteLine("Die Aussage ""objVarKontakt entspricht objVarKontakt2 ist "" " & _
            objVarKontakt.Equals(objVarKontakt2))
```

```
        Console.WriteLine("Die Aussage ""objVarKontakt entspricht objVarKontakt3 ist "" " & _
            objVarKontakt.Equals(objVarKontakt3))

        Console.WriteLine()

        'Alternativ durch das IS-Schlüsselwort:
        Console.WriteLine("Die Aussage ""objVarKontakt entspricht objVarKontakt2 ist "" " & _
            (objVarKontakt Is objVarKontakt2))

        Console.WriteLine("Die Aussage ""objVarKontakt entspricht objVarKontakt3 ist "" " & _
            (objVarKontakt Is objVarKontakt3))

        Console.WriteLine()
        Console.WriteLine("Zum Beenden Taste drücken")
        Console.ReadKey()
    End Sub

End Module

Class Kontakt

    Public Property Nachname As String
    Public Property Vorname As String
    Public Property Plz As String
    Public Property Ort As String

End Class
```

Diese kleine Demo ist weitestgehend selbsterklärend. Sie definiert drei Objektvariablen auf Basis der Klasse `Kontakt`. Sie instanziiert daraus zwei Objekte (aus `objVarKontakt1` und `objVarKontakt3`) und füttert sie mit den gleichen Daten. Die zweite Objektvariable (`objVarKontakt2`) verweist allerdings auf dieselbe Instanz wie die erste Objektvariable. Deswegen ist ein Vergleich mit `Equals` oder `Is` wahr.

TIPP Die Regel lautet: Verweisen zwei Objektvariablen auf dieselbe Instanz, und speichern sie deswegen dieselben Zeiger auf die eigentlichen Daten im Managed Heap, ergeben `Is` oder `Equals` als Ergebnis `True`, ansonsten `False`. Die Daten selbst (die Instanzen der Objekte) werden bei diesem Vergleich nicht verwendet.

Das Ausgabeergebnis dieses Programms sieht demzufolge folgendermaßen aus:

```
Die Aussage "objVarKontakt entspricht objVarKontakt2 ist " True
Die Aussage "objVarKontakt entspricht objVarKontakt3 ist " False

Die Aussage "objVarKontakt entspricht objVarKontakt2 ist " True
Die Aussage "objVarKontakt entspricht objVarKontakt3 ist " False

Zum Beenden Taste drücken
```

Equals, Is und IsNot im praktischen Einsatz

Die Frage die sich stellt, lautet wie bei allem, was man neu lernt: Wozu brauche ich das? Was das Testen auf identische Objekte anbelangt, ist die Antwort einfach: an fast jeder Stelle.

Das geht spätestens dann los, wenn Sie ein bestimmtes Objekt in einer Auflistung wie beispielsweise der `Items`-Auflistung der `ListBox` suchen. Sie möchten beispielsweise einen bestimmten Eintrag aus der Liste einer `ListBox` löschen. Zu diesem Zweck würden Sie die `Remove`-Methode verwenden, die an der `Items`-Eigenschaft der `ListBox` »hängt«. Die `Remove`-Methode nimmt jedes beliebige Objekt entgegen und ist natürlich nicht in der Lage, die Inhalte der Objektinstanzen zu vergleichen, die durch ein `ListBoxItem` repräsentiert werden und dem Objekt entsprechen, das Sie entfernen möchten. Sie ist aber in der Lage herauszufinden, ob zwei Objektvariablen auf dieselbe Instanz verweisen, und daran erkennt `Remove`, welches Objekt es zu entfernen hat.

Die folgende Demo zeigt dies im praktischen Einsatz. Mit dem schon bekannten Beispiel können Sie eine `ListBox` mit Kontaktdaten füllen. Doch ferner verfügen Sie über eine weitere Schaltfläche, mit der Sie einen selektierten Eintrag wieder aus der Liste entfernen können.

Den Verweis auf ein selektiertes Objekt einer `ListBox` können Sie mit `SelectedItem` der `Items`-Auflistung in Erfahrung bringen. Liefert `SelectedItem` den Wert `Nothing` zurück, handelt es sich um einen so genannten Null-Verweis; eine Objektvariable zeigt in diesem Fall auf keine Instanz mit Daten.

Gibt `SelectedItem` jedoch einen »Wert« zurück, der eben nicht `Nothing` ist, dann war ein Objekt der Liste ausgewählt. Dieses Objekt können Sie dann der `Remove`-Methode der `ListBox.Items`-Auflistung übergeben:

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 11\EqualsPraxis
```

Der folgende Auszug zeigt die Ereignisbehandlungsroutine der Schaltfläche, die ein Element aus der Liste löscht, sofern es selektiert war:

```
Private Sub btnKontaktLöschen_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
            Handles btnKontaktLöschen.Click
    Dim locKontakt As Object
    locKontakt = lstKontakte.SelectedItem
    If locKontakt IsNot Nothing Then
        lstKontakte.Items.Remove(locKontakt)
    End If
End Sub
```

Intern macht die `Remove`-Methode genau das gerade Gesagte: Sie iteriert (durchläuft) durch die Elemente und stellt fest, ob das zu löschende Element dem gerade bearbeiteten Element der Liste entspricht. Dabei führt sie die Prüfung auf Entsprechung ebenfalls mit `Is` bzw. der `Equals`-Methode durch.

Übersicht über die Eigenschaften und Methoden von Object

Die folgende Tabelle fasst die Methoden der Klasse `Object` zusammen:

Member	Beschreibung
Equals	Stellt fest, ob das aktuelle Objekt, dessen `Equals`-Member verwendet wird, mit dem angegebenen *Object* identisch ist. Zwei Objekte sind dann identisch, wenn ihre Instanzen (das heißt der Datenbereich, auf den sie zeigen) übereinstimmen. `ValueType` überschreibt diese Methode und vergleicht die einzelnen Member. Da Strukturen automatisch von `ValueType` ableiten, gilt dieses Verhalten für alle Strukturen.
GetHashCode	Produziert einen Hashcode (eine Art Identifizierungsschlüssel) auf Grundlage des Objektinhalts. Dieser Hashcode wird beispielsweise dann verwendet, wenn ein Objekt in einer Tabelle (einem Array) gesucht und dazu möglichst schnell möglichst eindeutig werden muss. Daher sollte `GetHashCode` nach Möglichkeit eindeutige Werte liefern, aber gleichzeitig auch vom Inhalt abhängige Werte als Grundlage der Hashcodeberechnung miteinbeziehen. Der in `Object` implementierte Algorithmus, den ja alle neuen Klassen, da sie implizit von `Object` erben, automatisch übernehmen, garantiert weder Eindeutigkeit noch Konsistenz – Sie sind also angehalten, nach Möglichkeit eigene Hashcodealgorithmen für abgeleitete Objekte zu entwickeln und diese Methode zu überschreiben, wenn Sie Ihr Objekt des Öfteren in Hashtabellen speichern wollen.
GetType	Ruft den aktuellen Typ der Instanz als Type-Objekt ab. Sie brauchen den Typ des Objekts als Ausgangspunkt, um über ein Verfahren namens Reflection (sprich: »refläkschen«, auf Deutsch: *Reflektion*, aber die Übersetzung ist unter Entwicklern nicht üblich) programmtechnisch untersuchen zu können, über welche Features es verfügt (Eigenschaften, Methoden, Felder, etc.). Mehr dazu erfahren Sie im Kapitel 22
(Shared) ReferenceEquals	Diese statische Methode entspricht der *Equals*-Methode, übernimmt die beiden zu vergleichenden Objekte aber als Parameter. Da diese Methode statischer Natur ist, können Sie sie nur über den Typnamen aufrufen (`Object.ReferenceEquals`).
ToString()	Liefert eine Zeichenkette zurück, die das aktuelle Objekt beschreibt. In der ursprünglichen Version ist das wörtlich zu nehmen; `ToString` gibt nämlich den Klassennamen zurück, wenn Sie diese Funktion nicht überschreiben. `ToString` sollte nach Möglichkeit den Inhalt des Objekts – wenigstens teilweise – als Zeichenkette zurückliefern.
Finalize	Wenn die `Finalize`-Methode eines Objekts aufgerufen wird, sollte die Quelle des Aufrufs nur der Garbage Collector sein. Er signalisiert dem Objekt durch diesen Aufruf, dass es im Rahmen der »Müllabfuhr« im Begriff ist, entsorgt zu werden, und das Objekt hat hierbei die Möglichkeit, Ressourcen freizugeben, die es nicht mehr benötigt. Klassen, die `Finalize` durch eigene Implementierungen ersetzen, sollten dazu das `IDisposable`-Pattern verwenden. Mehr dazu finden Sie in Kapitel 14.
MemberwiseClone	Erstellt eine so genannte »flache Kopie« von dem Objekt, das die Methode beherbergt. Wenn Sie `MemberwiseClone` aufrufen, dann legt diese Methode eine Kopie aller Wertetypmember an und stellt diese in einer weiteren, neuen Objektinstanz zur Verfügung. Für Referenztypen werden nur Adresskopien erstellt – sie zeigen anschließend also auf dieselben Objekte im Managed Heap, auf die auch die Referenztypen des Originals zeigen. `MemberwiseClone` ist `Protected` und kann deshalb nur aus einer Klasse selbst heraus aufgerufen werden. Klassen können eine `Clone`-Methode implementieren und dabei ihre eigene `MemberwiseClone`-Methode zu Hilfe nehmen. Klassen, die diese Methode implementieren, sollten auch die `ICloneable`-Schnittstelle implementieren.

Tabelle 11.2 Die Beschreibung der *Object*-Member

Shadowing (Überschatten) von Klassenprozeduren

Visual Basic kennt eine weitere Version des Ersetzens von Prozeduren in einer Basisklasse durch andere gleichen Namens einer abgeleiteten Klasse. Diesen Vorgang nennt man »Shadowing« oder »Überschatten« von Elementen.

Wenn Sie in einer Basisklasse eine Funktion definiert haben, dann kann eine Funktion gleichen Namens in einer abgeleiteten Klasse das Original schlichtweg ausblenden, wie im folgenden Beispiel zu sehen:

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 11\Shadowing01
```

```
Module mdlMain

    Sub Main()
        Dim locBasisinstanz As New Basisklasse
        Console.WriteLine(locBasisinstanz.EineFunktion().ToString())

        locBasisinstanz = New AbgeleiteteKlasse
        Console.WriteLine(locBasisinstanz.EineFunktion().ToString())

        Console.WriteLine()
        Console.WriteLine("Taste drücken zum Beenden!")
        Console.ReadKey()
    End Sub

End Module

Public Class Basisklasse

    Protected test As Integer

    Sub New()
        test = 10
    End Sub

    Public Function EineFunktion() As Integer
        Return test * 2
    End Function

End Class

Public Class AbgeleiteteKlasse
    Inherits Basisklasse

    Public Function EineFunktion() As Integer
        Return test * 3
    End Function

End Class
```

Ohne zunächst ganz genau auf die beiden Klassenimplementierungen zu achten, würden Sie wahrscheinlich annehmen, dass das Modul zunächst den Wert 20 und in der folgenden Zeile den Wert 30 ausgibt. Doch bei diesem Beispiel handelt es sich nur augenscheinlich um die Anwendung von Polymorphie, denn die einzige Funktion der Basisklasse ist weder durch das Schlüsselwort `Overridable` als überschreibbar definiert, noch versucht die gleiche Funktion der abgeleiteten Klasse, diese Funktion mit `Overrides` zu überschreiben. Bei genauerer Betrachtung zeigt Visual Basic für `EineFunktion` der abgeleiteten Klasse eine Warnmeldung an, etwa wie in Abbildung 11.19 zu sehen.

Abbildung 11.19 In diesem Beispiel blendet eine Funktion der abgeleiteten Klasse die der Basisklasse aus

Visual Basic gibt hier zwar eine Meldung in der Fehlerliste aus, kompiliert das Programm aber dennoch. Die vermeintliche Fehlermeldung ist in diesem Fall nämlich nur eine Warnmeldung und macht Sie darauf aufmerksam, dass die Funktion von einer anderen überschattet wird.

Das hat Auswirkungen auf das Verhalten der Klasse. Denn obwohl die Objektvariable eine Instanz der abgeleiteten Instanz enthält, wird dennoch die Funktion der Basisklasse aufgerufen. Der von Visual Basic durchgeführte Überschattungsvorgang unterbricht die Erbfolge der Klasse an dieser Stelle und stellt sicher, dass eine Funktion, die nicht zum Überschreiben markiert ist, auch tatsächlich nicht überschrieben werden kann.

Sie werden diese Warnmeldung übrigens los, indem Sie das Schlüsselwort `Shadows` vor die Funktion in der abgeleiteten Klasse setzen. Das sieht dann so aus:

```
Public Shadows Function EineFunktion() As Integer
```

`Shadows` hat also grundsätzlich eine reine kosmetische Auswirkung (das gilt auch für die folgenden Abschnitte) und überschattet selbst nichts. Es beseitigt nur die Warnung. Wie Sie die hier neu definierte Methode, die ja jetzt brachliegt, dennoch nutzen können, erfahren Sie in den folgenden Abschnitten.

Shadows als Unterbrecher der Klassenhierarchie

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 11\Shadowing02
```

```
Option Explicit On
Option Strict On

Module mdlMain

    Sub Main()
        Dim locVierteKlasse As New VierteKlasse
        Dim locErsteKlasse As ErsteKlasse = locVierteKlasse
        Dim locZweiteKlasse As ZweiteKlasse = locVierteKlasse
        Dim locDritteKlasse As DritteKlasse = locVierteKlasse

        Console.WriteLine(locErsteKlasse.EineFunktion)
        Console.WriteLine(locZweiteKlasse.EineFunktion)
        Console.WriteLine(locDritteKlasse.EineFunktion)
        Console.WriteLine(locVierteKlasse.EineFunktion)

        Console.WriteLine()
        Console.WriteLine("Taste drücken zum Beenden!")
        Console.ReadKey()

    End Sub
End Module

Public Class ErsteKlasse

    Public Overridable Function EineFunktion() As String
        Return "Erste Klasse"
    End Function

End Class

Public Class ZweiteKlasse
    Inherits ErsteKlasse

    Public Overrides Function EineFunktion() As String
        Return "Zweite Klasse"
    End Function
End Class

Public Class DritteKlasse
    Inherits ZweiteKlasse

    Public Overrides Function EineFunktion() As String
        Return "Dritte Klasse"
    End Function

End Class

Public Class VierteKlasse
    Inherits DritteKlasse
    Public Overrides Function EineFunktion() As String
        Return "Vierte Klasse"
    End Function

End Class
```

Was glauben Sie, kommt heraus, wenn Sie das Beispiel laufen lassen? Die verschiedenen Objektnamen und Klassennamen mögen anfangs ein wenig verwirren, aber das Ergebnis ist klar: Das Programm gibt viermal den Text »vierte Klasse« aus. Klar, denn die vierte Klasse wird ein einziges Mal instanziiert und durch jede andere Variable referenziert.

Jede der anderen Objektvariablen ist über eine Klasse der Klassenerbfolge definiert und kann damit auch – wie Sie es an vielen Beispielen auf den vergangenen Seiten dieses Kapitels schon kennen gelernt haben – auf jede Instanz einer abgeleiteten Klasse verweisen. Jetzt nehmen Sie eine kleine Veränderung an der dritten Klasse vor:

```
Public Class DritteKlasse
    Inherits ZweiteKlasse

    '"Overrides" wurde gegen "Overridable Shadows" ausgetauscht:
    Public Overridable Shadows Function EineFunktion() As String
        Return "Dritte Klasse"
    End Function

End Class
```

und raten Sie, was beim erneuten Start des Programms passiert! Die Ausgabe lautet:

```
Zweite Klasse
Zweite Klasse
Vierte Klasse
Vierte Klasse

Taste drücken zum Beenden!
```

Hätten Sie es gewusst? Dabei ist das Ergebnis gar nicht so schwer zu verstehen:

Im Prinzip hat `EineFunktion` der dritten und vierten Klasse überhaupt nichts mehr mit den ersten beiden Klassen zu tun. Durch den Überschattungsvorgang in der dritten Klasse wird die Funktion komplett neu implementiert. Aus diesem Grund ist wie bei einer komplett anderen Funktion, die in `DritteKlasse` »dazukommt«, auch ein erneutes `Overridable` notwendig, denn anderenfalls könnte `VierteKlasse` die Funktion gar nicht mit `Overrides` überschreiben.

HINWEIS Der eigentliche Überschattungsvorgang, also das Shadowing, wird übrigens nicht durch das Schlüsselwort Shadows ausgelöst; das passiert schon durch die Definition der Methode unter gleichem Namen. Shadows sorgt dann nur dafür, dass der Warnhinweis verschwindet.

Ein wenig Verwirrend mag sein, dass die erste Klasse den Text »Zweite Klasse« ausgibt. Doch welchen anderen Text soll sie sonst ausgeben? »Erste Klasse«, mag man auf den ersten Blick denken, doch das ist falsch, denn das würde gegen das Prinzip der Polymorphie verstoßen. »Vierte Klasse« kann nicht ausgegeben werden, denn diese Funktion ist aus Sicht von `ErsteKlasse` gesehen nicht erreichbar. In `DritteKlasse` wird diese Funktion schlicht und ergreifend durch eine komplett neue Version ersetzt. .NET Framework rettet also, was zu retten ist, und versucht in der Erbfolge soweit wie möglich nach vorne zu gehen – und damit ist `ZweiteKlasse` die letzte Funktion, die durch Polymorphie erreichbar ist, bevor die Erbfolge durch den Überschattungsvorgang in `DritteKlasse` unterbrochen wird.

Dieses Verhalten ist durchaus erwünscht, denn wenn Sie nicht wollen, dass ein Element einer Klasse überschrieben wird, dann lässt es sich auch nicht überschreiben. Die CLR garantiert immer, dass eine nicht überschreibbare Funktion ihre ursprünglichen Fähigkeiten behält, selbst wenn sie andere Funktionen in abgeleiteten Klassen mit gleichem Namen überschatten.

Intern gibt es zwei verschiedene Versionen von `EineFunktion`, und wenn Sie das Beispielprogramm wie folgt abändern, wird klar, was eigentlich passiert.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 11\Shadowing03
```

```
Option Explicit On
Option Strict On

Module mdlMain
    Sub Main()
        Dim locVierteKlasse As New VierteKlasse
        Dim locErsteKlasse As ErsteKlasse = locVierteKlasse
        Dim locZweiteKlasse As ZweiteKlasse = locVierteKlasse
        Dim locDritteKlasse As DritteKlasse = locVierteKlasse

        Console.WriteLine(locErsteKlasse.EineFunktion_a)
        Console.WriteLine(locZweiteKlasse.EineFunktion_a)
        Console.WriteLine(locDritteKlasse.EineFunktion_b)
        Console.WriteLine(locVierteKlasse.EineFunktion_b)

        Console.WriteLine()
        Console.WriteLine("Taste drücken zum Beenden!")
        Console.ReadKey()

    End Sub

End Module

Public Class ErsteKlasse

    Public Overridable Function EineFunktion_a() As String
        Return "Erste Klasse"
    End Function

End Class

Public Class ZweiteKlasse
    Inherits ErsteKlasse

    Public Overrides Function EineFunktion_a() As String
        Return "Zweite Klasse"
    End Function

End Class

Public Class DritteKlasse
```

```
    Inherits ZweiteKlasse

    Public Overridable Function EineFunktion_b() As String
        Return "Dritte Klasse"
    End Function
End Class

Public Class VierteKlasse
    Inherits DritteKlasse

    Public Overrides Function EineFunktion_b() As String
        Return "Vierte Klasse"
    End Function

End Class
```

Sonderform »Modul« in Visual Basic

Module gibt es bei allen bislang existierenden .NET-Programmiersprachen nur in Visual Basic. Und auch hier ist ein Modul nichts weiter als eine Mogelpackung, denn das, was als Modul bezeichnet wird, ist im Grunde genommen nichts anderes als eine Klasse mit statischen Methoden, Eigenschaften und Membern, die nicht instanziiert werden kann (weil sie beispielsweise einen Konstruktor aufweist, der mit dem Zugriffsmodifizierer `Private` gekennzeichnet ist, oder weil es sich um eine abstrakte Klasse handelt). Halten wir fest:

- Ein Modul ist nicht instanziierbar. Eine abstrakte Klasse oder eine mit privatem Konstruktor auch nicht.
- Ein Modul kann keine überschreibbaren Prozeduren zur Verfügung stellen. Die statischen Prozeduren einer Klasse können das auch nicht.
- Ein Modul kann nur Prozeduren zur Verfügung stellen, auf die aber nur ohne Instanzobjekt direkt zugegriffen werden kann. Das gleiche gilt für die statischen Prozeduren einer abstrakten Klasse oder einer mit privatem Konstruktor.

Es gibt aber auch feine Unterschiede: So können Module beispielsweise keine Schnittstellen implementieren; das können zwar Klassen, aber diese können keine *statischen* Schnittstellenelemente definieren. Insofern ist dieser scheinbare Unterschied in Wirklichkeit gar keiner. Ein Modul kann auch nur auf oberster Ebene definiert und nicht in einem anderen geschachtelt werden.

Module setzen Sie bei der OOP vorschlagsweise so wenig wie möglich ein, denn sie widersprechen dem Anspruch von .NET, möglichst wiederverwendbaren Code zu schaffen.

Hier im Buch finden Sie aus diesem Grund Module nur, wenn es um »Quick and dirty«-Projekte geht, bei denen beispielsweise eine Konsolenanwendung Tests durchzuführen hat oder »mal eben« etwas demonstrieren soll, genau so, wie Sie es in vorangegangenen Kapiteln bereits erlebt haben.

Singleton-Klassen und Klassen, die sich selbst instanziieren

Stellen Sie sich vor, Sie möchten eine Klasse entwerfen, die beispielsweise die Funktionen eines Bildschirms oder eines Druckers steuert. Das Besondere daran ist, dass Sie eine Kontrolle über die Instanziierung dieser Klasse benötigen. Es reicht nicht aus, der Klasse selbst zu überlassen, wie oft sie sich instanziiert – einen bestimmten Drucker gibt es nur ein einziges Mal, und nach einer Instanz sollte Schluss sein.

Eine abstrakte Klasse mit statischen Prozeduren (meinetwegen auch ein Modul) könnte vielleicht eine Alternative dazu sein – doch das Problem dabei ist: Weder die Funktionen eines Moduls noch die statischen Funktionen einer abstrakten Klasse können Sie in anderen Klassen überschreiben.

Die Lösung dazu sind so genannte Singleton-Klassen. Singleton-Klassen sind, anders als die Klassenvarianten, die Sie bislang kennen gelernt haben, keine »eingebauten« Klassentypen der CLR bzw. von .NET Framework. Sie müssen sie selbst entwickeln – doch das ist einfacher, als Sie denken.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 11\Singleton
```

```
Imports System.Threading

Module mdlMain

    Dim varModule As Integer = 5

    Sub New()

    End Sub

    Sub Main()
        Dim locSingleton As Singleton = Singleton.GetInstance()
        Dim locSingleton2 As Singleton = Singleton.GetInstance()

        Console.WriteLine(locSingleton Is locSingleton2)
        Console.WriteLine()
        Console.WriteLine("Taste drücken zum Beenden!")
        Console.ReadKey()
    End Sub

    Property test() As Integer
        Get

        End Get
        Set(ByVal Value As Integer)

        End Set
    End Property

End Module
```

```
Class Singleton

    Private Shared locSingleton As Singleton
    Private Shared locMutex As New Mutex

    'Konstruktor ist privat,
    'damit kann die Klasse nur von "sich selbst" instanziiert werden
    Private Sub New()
    End Sub

    'GetInstance gibt eine Singleton-Instanz zurück
    'Nur durch diese Funktion kann die Klasse instanziiert werden
    Public Shared Function GetInstance() As Singleton
        'Vorgang Thread-sicher machen
        'Wartet, bis ein anderer Thread, der diese Klasse
        'instanziiert, damit fertig ist
        locMutex.WaitOne()
        Try
            If locSingleton Is Nothing Then
                locSingleton = New Singleton
            End If
        Finally
            'Instanziieren abgeschlossen,
            '...kann weiter gehen...
            locMutex.ReleaseMutex()
        End Try

        'Instanz zurückliefern
        Return locSingleton

    End Function

End Class
```

Ihnen wird als Erstes auffallen, dass diese Klasse nicht direkt instanziiert werden kann, denn ihr Konstruktor ist privat. Wie also kommen Sie dann überhaupt an eine Instanz der Klasse?

Die Antwort auf diese Frage liegt in der statischen Funktion `GetInstance`. Sie sorgt dafür, dass die Klasse sich selbst instanziiert, wenn es erforderlich ist. Gleichzeitig achtet sie darauf, dass der Instanziierungsvorgang der Klasse threadsicher ist: .NET Framework erlaubt es nämlich, wahrhaftiges Multitasking in eigenen Programmen zu implementieren. Dieser Vorgang bezeichnet Zustände, bei denen mehrere Aufgaben innerhalb eines Programms (oder mehrerer Programme) gleichzeitig ausgeführt werden können. Damit ein weiterer Instanziierungsversuch einer Singleton-Klasse nicht ausgerechnet dann passiert, wenn ein anderer Thread fast (aber eben noch nicht ganz) mit der Instanziierung fertig ist – beide Threads sich sozusagen »mittendrin« treffen würden und es damit doch die zu verhindernden zwei Instanzen gäbe –, nutzt die Klasse eine Instanz von `Mutex`[7], um dieses Aufeinandertreffen zu vermeiden.

[7] Mutex, abgeleitet von »**mut**ual **ex**clusion«, etwa »gegenseitiger Ausschluss«.

Solange Sie keine Multithreading-Programmierung anwenden, brauchen Sie jedoch keinen Gedanken daran zu verschwenden. Mehr zu diesem Thema erfahren Sie im Threading-Kapitel später in diesem Buch (Kapitel 29)

Das Hauptprogramm prüft die Funktionalität der Singleton-Klasse. Es holt sich eine Instanz mit `GetInstance` und speichert sie in einer Objektvariablen. Den gleichen Vorgang wiederholt es anschließend mit einer weiteren Objektvariablen und vergleicht die beiden »Instanzen« anschließend mit `Is`.

Und in der Tat: Das Programm gibt `True` auf dem Bildschirm aus, denn die Singleton-Klasse hat nur *eine* Instanz ihrer selbst kreiert – beide Objektvariablen zeigen also auf die gleiche Instanz.

Kapitel 12

Entwickeln von Wertetypen

In diesem Kapitel:

Erstellen von Wertetypen mit Structure am praktischen Beispiel 442

Kapitel 9 ist ja bereits grundsätzlich auf die Möglichkeit eingegangen, neben den aus Klassen entstehenden Referenztypen auch Wertetypen mit `Structure` zu entwickeln. Dieses Kapitel möchte sich diesem Thema noch mal etwas ausführlicher widmen, und vor allen Dingen ein wenig tiefer in die Praxis eintauchen: Es zeigt nicht nur die Entwicklung einer wieder verwendbaren Komponente, die Sie auch in Ihren eigenen Anwendungen einsetzen können, sondern geht in weiteren Abschnitten auch auf spezielle Details von Wertetypen ein – wie Sie beispielsweise die Speicherabfolge von Feldvariablen innerhalb einer Struktur genau steuern können oder wie es sich mit Konstruktoren bei Wertetypen verhält.

Erstellen von Wertetypen mit Structure am praktischen Beispiel

Erinnern wir uns. Grundsätzlich gilt: Eine Klasse produziert einen Referenztyp und eine Struktur einen Wertetyp. Prinzipiell erstellen Sie eine Struktur genau wie eine Klasse, und beide haben auch ähnliche Fähigkeiten, wie es durch das folgende Beispiel deutlich werden soll.

Zum Beispielszenario: Gerade beim Programmieren kommt es immer wieder vor, dass Sie Zahlen von einem Zahlensystem ins andere konvertieren müssen. Einige Konvertierungen werden von .NET Framework unterstützt – so können Sie beispielsweise hexadezimale Werte mit der statischen `Parse`-Funktion etwa so

```
Dim EinInteger As Integer = Integer.Parse("FFFF", Globalization.NumberStyles.HexNumber)
```

in eine Dezimalzahl umwandeln; eine andere Möglichkeit besteht durch die Benutzung der `ToInt`-Funktion bzw. der `ToString`-Funktion der `Convert`-Klasse. Mit

```
Dim EinInteger as Integer = Convert.ToInt32(AnderesZahlensystemString, ZahlenSystemInteger)
```

lässt sich ebenfalls ein String, der eine Zahl eines anderen Zahlensystems beinhaltet, in einen Integer zurückverwandeln. Allerdings können Sie mit `ZahlenSystemInteger` nur eine Zahl des Dual- (Basis 2), des Oktal- (Basis 8), natürlich des Dezimal- (Basis 10) und des Hexadezimalsystems (Basis 16) umwandeln. Die gleichen Beschränkungen gelten für die Konvertierungen eines Integerwerts in ein anderes Zahlensystem, der dann in Form eines Strings abgebildet wird. Diese Aufgabe können Sie mit dem Gegenstück durchführen, etwa durch

```
Dim AnderesZahlensystemString as String = Convert.ToString(IntegerUmzuwandeln, ZahlenSystemInteger)
```

um die Integer-Variable `IntegerUmzuwandeln` in ein anderes Zahlensystem umzuwandeln, das durch `ZahlenSystemInteger` (nur 2, 8, 10 oder 16 sind auch hier wieder gültige Werte) definiert wird. Zum Glück – denn einen Wertetyp zu schaffen, der beliebige Konvertierungen (nicht nur vom und ins Hexadezimalsystem) beherrscht, ist ein willkommenes, da brauchbares Beispiel.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 12\Structure01
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

So könnte Ihnen beispielsweise sogar das 32er-Zahlensystem[1] von Nutzen sein, also ein Zahlensystem zur Basis 32: Sie meinen, Sie hätten das nie benutzt? Vielleicht ja doch: Denn wenn Sie Visual Studio selbst installiert haben, dann mussten Sie zur Installation einen Key eingeben, der natürlich mithilfe eines Algorithmus berechnet wurde. Der Algorithmus basiert wahrscheinlich weniger auf dem Hantieren mit Buchstaben, sondern wird – viel wahrscheinlicher – numerisch berechnet. Ein `ULong`-Wert (64 Bit, ohne Vorzeichen, gibt es übrigens seit Visual Basic 2005), der dieses Ergebnis des Seriennummernalgorithmus speichert, könnte dann beispielsweise auf Basis des Duotrigesimalsystems (32er-Zahlensystem) in eine Zeichenkette umgewandelt werden. Die Zahl

```
9.223.372.036.854.775.807
```

würde in diesem Zahlensystem wie folgt ausschauen

```
7VVVVVVVVVVVV
```

und die Zahl

```
9.153.672.076.852.735.401
```

um ein anderes Beispiel zu zeigen, lautete wie folgt:

```
7U23085PJGBD9
```

Und hier das erklärte Listing der Struktur `NumberSystems`:

```
Public Structure NumberSystems

    Private myUnderlyingValue As ULong
    Private myNumberSystem As Integer
    Private Shared myDigits As Char()

    Shared Sub New()
        myDigits = New Char() {"0"c, "1"c, "2"c, "3"c, "4"c, "5"c, "6"c, "7"c, "8"c, "9"c, "A"c, _
                               "B"c, "C"c, "D"c, "E"c, "F"c, "G"c, "H"c, "I"c, "J"c, "K"c, "L"c, _
                               "M"c, "N"c, "O"c, "P"c, "Q"c, "R"c, "S"c, "T"c, "U"c, "V"c, "W"c, _
                               "X"c, "Y"c, "Z"c}
    End Sub
```

Die Tabelle für die Umwandlung wird in einem statischen Array gespeichert, das im statischen Konstruktor der Struktur initialisiert wird. Der Konstruktor ist überladen: Ihm wird der Wert übergeben, den die Struktur speichert und im Bedarfsfall als `String` mit entsprechender Numerale des gewünschten Zahlensystems

[1] Bester Name dafür wäre das »Duotrigesimalsystem«, für den es jedoch keine historische Absicherung gibt; ohne Gewähr, dass dieser Link zum Zeitpunkt der Drucklegung noch funktioniert, können Sie sich unter *http://tinyurl.com/35c4e6* über die Benennung von Zahlensystemen informieren. Kleines Kuriosum am Rande: Das Hexadezimalsystem ist eigentlich recht inkonsistent benannt, denn es wurde aus einem griechischem und einem lateinischem Wortstamm gebildet (»Hexa« griechisch; »decem« lateinisch). Streng genommen müsste es »sedezimal« oder »sexadezimal« heißen. Da die Kurzform für Hexadezimal kurz »Hex« lautet, können wir für den griechisch-lateinischen Mischmasch aber eher dankbar sein.

zurückliefern kann. Sie können dem Konstruktor entweder nur einen `Integer`- oder einen `ULong`-Wert übergeben oder einen `ULong`-Wert und einen weiteren Parameter, der bestimmt, in welchem Zahlensystem Sie arbeiten möchten (bis maximal zum 33er-System).

```
Sub New(ByVal Value As Integer)
    Me.New(CULng(Value), 16)
End Sub

Sub New(ByVal Value As ULong)
    Me.New(Value, 16)
End Sub

Sub New(ByVal Value As ULong, ByVal NumberSystem As Integer)

    myUnderlyingValue = Value
    If NumberSystem < 2 OrElse NumberSystem > 33 Then
        Dim Up As Exception = New OverflowException _
            ("Kennziffer des Zahlensystems außerhalb des gültigen Bereichs!")
        'Kleiner Scherz für die Englisch Sprechenden:
        Throw Up
    End If
    myNumberSystem = NumberSystem

End Sub

Public Property Value() As ULong

    Get
        Return myUnderlyingValue
    End Get
    Set(ByVal Value As ULong)
        myUnderlyingValue = Value
    End Set

End Property
```

Die `Value`-Eigenschaft dient lediglich dazu, den für die Konvertierung in das jeweilige Zahlensystem gespeicherten Wert neu zu bestimmen oder abzufragen.

```
Public Property NumberSystem() As Integer
    Get
        Return myNumberSystem
    End Get
    Set(ByVal Value As Integer)
        If Value < 2 OrElse Value > 33 Then
            Dim Up As Exception = New OverflowException _
                ("Kennziffer des Zahlensystems außerhalb des gültigen Bereichs!")
            Throw Up
        End If
        myNumberSystem = Value
    End Set
End Property
```

Die NumberSystem-Eigenschaft verwenden Sie, um das Zahlensystem, in das Sie später die Umwandlungen vornehmen wollen, neu zu bestimmen oder abzufragen.

```
Public Overrides Function ToString() As String

    Dim locResult As String = ""
    Dim locValue As ULong = myUnderlyingValue

    Do
        Dim digit As Integer = CInt(locValue Mod NumberSystem)
        locResult = CStr(myDigits(digit)) & locResult
        locValue \= CULng(NumberSystem)
    Loop Until locValue = 0

    Return locResult

End Function
```

Die ToString-Methode ist der erste »Dienstleister« für die eigentliche Aufgabe. Sie verfährt nach folgendem Algorithmus, um die Zeichen (die Numeralia) für das eingestellte Zahlensystem zu ermitteln:

Zunächst kopiert sie den Ursprungswert in eine temporäre Variable, um die Value-Eigenschaft nicht zu zerstören – diese Variable wird im Folgenden nämlich verändert. Nun führt ToString eine Restwertdivision mit der Mod-Funktion durch. Diese liefert nicht das Ergebnis der Division, sondern den Restwert. Ein Beispiel im 10er-System soll verdeutlichen, was gemeint ist:

Wenn Sie den Wert 129 durch 10 teilen, kommt 12 dabei heraus, und es bleibt ein Rest von 9. Genau diese 9 ist aber in diesem Fall wichtig, denn sie entspricht der ersten gesuchten Ziffer der Ergebniszeichenkette (der äußerst rechts stehenden, um genau zu sein). Anschließend wird der Wert durch die Basiszahl des Zahlensystems geteilt – um bei diesem Beispiel zu bleiben also durch 10 –, und als Ergebnis kommt 12 dabei raus. Da das Divisionsergebnis (dieses Mal das Ergebnis, nicht der Restwert) größer ist als 0, wiederholt sich der Vorgang. Wieder wird der Divisionsrestwert ermittelt, und der beträgt dieses Mal 2. Die 2 entspricht der zweiten ermittelten Ziffer. Die Schleife wird nun so lange wiederholt, bis alle Ziffern (bzw. Numeralia) bekannt sind. Die Numeralia selbst werden übrigens aus dem myDigits-Array gelesen, dem statischen Array, das durch den statischen Konstruktor der Struktur bei ihrer ersten Verwendung angelegt wird.

HINWEIS Sie finden in diesem Beispiel einige Konvertierungen von Wertetypen in andere mithilfe der Cxxx-Operatoren, zu denen das nächste Kapitel mehr Infos gibt. Nur so viel fürs Erste: Eine Konvertierung vom Typ Integer in den ULong-Datentyp mithilfe von CULng ist in der fett geschriebenen Zeile des oben stehenden Codeausschnittes notwendig, damit die Division über die volle 64-Bit-Breite stattfindet. Ohne diese Konvertierung würde Visual Basic eine (standardmäßige) 32-Bit-Integerdivision initiieren, die mit der 64 Bit breiten ULong-Variable locValue nicht funktionieren kann.

```
Public Shared Function Parse(ByVal Value As String, ByVal NumberSystem As Integer) As NumberSystems

    'Hier wird der Value zusammengebaut
    Dim locValue As ULong

    For count As Integer = 0 To Value.Length - 1
        Try
```

```
            'Aktuelles Zeichen im String, das verarbeitet wird
            Dim locTmpChar As String = Value.Substring(count, 1)

            'Binärsuche anwenden, um das Zeichen im Array zu finden und damit die Ziffernnummer
            Dim locDigitValue As Integer = CInt(Array.BinarySearch(myDigits, CChar(locTmpChar)))

            'Prüfen, ob sich das Zeichen im Gültigkeitsbereich befindet
            If locDigitValue >= NumberSystem OrElse locDigitValue < 0 Then
                Dim Up As Exception = New FormatException _
                    ("Ziffer '" & locTmpChar & "' ist nicht Bestandteil des Zahlensystems!")
                Throw Up
            End If

            'Aus der gefundenen Ziffernnummer die Potenz bilden, und zum Gesamtwert addieren
            locValue += CULng(Math.Pow(NumberSystem, Value.Length - count - 1) * locDigitValue)

        Catch ex As Exception
            'Für den Fall, dass zwischendurch 'was schiefgeht
            Dim Up As Exception = New InvalidCastException _
                ("Ziffer des Zahlensystems außerhalb des gültigen Bereichs!")
            Throw Up
        End Try
        'nächstes Zeichen verarbeiten
    Next

    Return New NumberSystems(locValue, NumberSystem)

    End Function
End Structure
```

Die `Parse`-Funktion ist eine statische Funktion (genau wie die `Parse`-Funktionen vieler Datentypen in der CLR) und dient dazu, eine Zeichenkette, die dem Wert eines bestimmten1 Zahlensystems entspricht, in einen `NumberSystem`-Wert umzuwandeln. Prinzipiell arbeitet sie nach der Formel

$$\text{NumeralerWert} = \text{Ziffernwert} \times \text{Zahlensystembasiswert}^{\text{Ziffernposition}-1}$$

Die Werte der Ziffern sind dabei durchnummeriert von 0–33 (Ziffern von 0–9, gefolgt vom Alphabet (Großbuchstaben) von A–Z). Mit einer Schleife iteriert die Funktion dabei von vorne nach hinten durch die Zeichen des umzuwandelnden Strings. Mit der statischen `BinarySearch`-Funktion der `Array`-Klasse ermittelt sie dabei den Ziffernwert. Dieser stellt anschließend den Wert eines Multiplikators des Produkts dar. Der andere Multiplikator ergibt sich durch das Potenzieren des Basiswerts des Zahlensystems mit der Ziffernposition. Auch hier soll ein Beispiel helfen, den Algorithmus besser zu verstehen, dieses Mal jedoch mit einer Konvertierung aus dem Hexadezimalsystem:

Gegeben sei die Zahl »F3E«. Um diese umzurechnen, ermittelt die Funktion den Ziffernwert für »F«, der dem Wert 15 entspricht. Da es das dritte Zeichen im String ist (von hinten gesehen), wird 16 mit 2 (es ist Ziffernposition 1) potenziert (ergibt 256) und mit 15 multipliziert – das Ergebnis lautet: 3840. Nun ist das mittlere Zeichen an der Reihe. Der Ziffernwert beträgt 3, der Exponent 1, denn es ist das 2. Zeichen der Zeichenkette. Zwischenergebnis: 48, addiert zum vorherigen Wert ergibt 3888. Was fehlt, ist die letzte Ziffer.

Der Exponent[2] ist 0, damit entspricht das Produkt dem Ziffernwert (Multiplikation mit eins verändert nichts), und dieser entspricht 14 für das »E«. Die letzte »14« wird zum Zwischenergebnis addiert, und wir haben das Ergebnis 3902.

Zum Projekt: Hauptprogramm und Struktur sind in dem Programmbeispiel in zwei verschiedenen Dateien untergebracht. Ein Doppelklick auf *NumberSystems.vb* im Projekt-Explorer öffnet den Quellcode der Struktur; *mdlMain.vb* enthält das Modul für das Hauptprogramm.

```
Module mdlMain

    Sub Main()

        Dim locLong As ULong
        locLong = &HFFFFFFFFFFFFFFFFUL
        Dim locNS As New NumberSystems(locLong)
        Console.WriteLine("{0:#,##0} entspricht:", locLong)

        locNS.NumberSystem = 2 : Console.WriteLine("Binär: " & locNS.ToString)
        locNS.NumberSystem = 8 : Console.WriteLine("Oktal: " & locNS.ToString)
        locNS.NumberSystem = 10 : Console.WriteLine("Dezimal: " & locNS.ToString)
        locNS.NumberSystem = 16 : Console.WriteLine("Hexadezimal: " & locNS.ToString)
        locNS.NumberSystem = 32 : Console.WriteLine("Duotrigesimal: " & locNS.ToString)

        'Der umgekehrte Weg:
        Console.WriteLine()
        Console.WriteLine("Gegenbeispiel:")
        Console.Write("'7U23085PJGBD9' duotrigesimal entspricht dezimal: ")
        locNS = NumberSystems.Parse("7U23085PJGBD9", 32)
        Console.WriteLine(locNS.Value)

        Console.WriteLine()
        Console.WriteLine("Return drücken zum Beenden")
        Console.ReadLine()
    End Sub
End Module
```

Wenn Sie das Programm starten, erhalten Sie folgende Ausgabe:

```
18.446.744.073.709.551.615 entspricht:
Binär: 1111111111111111111111111111111111111111111111111111111111111111
Oktal: 1777777777777777777777
Dezimal: 18446744073709551615
Hexadezimal: FFFFFFFFFFFFFFFF
Duotrigesimal: FVVVVVVVVVVVV

Gegenbeispiel:
'7U23085PJGBD9' duotrigesimal entspricht dezimal: 9153672076852735401

Return drücken zum Beenden
```

[2] Sie erinnern sich an die Schulzeit? – Jeder Wert potenziert mit 0 ergibt 1. Die Zahl wird dabei durch sich selbst geteilt.

Verhalten der Parameterübergabe mit ByVal und ByRef steuern

Es kann wünschenswert sein, einen Wertetypen durch das Schlüsselwort `ByRef` als Referenz einer Prozedur zu übergeben. In diesem Fall werden beim Aufruf nicht wie sonst die Daten selbst auf den Stack kopiert, sondern ein Zeiger auf die entsprechende Speicherstelle im Stack. Änderungen, die an den Daten innerhalb der Prozedur erfolgen, spiegeln sich damit direkt in der ursprünglichen Variable wider. Eine Änderung der Variablen innerhalb der *aufgerufenen* Prozedur wirkt sich also auch in einer Änderung des Variableninhaltes des *aufrufenden* Programmteils aus. Die Abänderung des Codes macht diesen Sachverhalt deutlich:

```
Dim Wertetyp1 as New NumberSystems(10)
NimmtWertetyp(Wertetyp1)
Console.WriteLine(Wertetyp1.Value)
Dim Wertetyp2 as NumberSystems = Wertetyp1
Wertetyp2.Value = 50
Console.WriteLine(Wertetyp1.Value)
.
.
.
Sub NimmtWertetyp(ByRef Wertetyp As NumberSystems)

    Wertetyp.Value = 99

End Sub
```

Eine solche Übergabe `ByRef` kann zum Beispiel sinnvoll sein, wenn Sie mehrere Informationen an das übergeordnete Programm zurückgeben wollen. Eine Funktion hat nun einmal nur einen Rückgabewert. Wie aber sollten Sie in einer Funktion mit dem Hauptprogramm kommunizieren, wenn Sie beispielsweise eine Zeichenkette an eine Funktion übergeben wollen, die diese Zeichenkette in eine Zahl konvertiert, und einerseits die Information benötigen, ob dies überhaupt möglich war (Rückgabe der Funktion dann entweder `True` oder `False`), Sie aber auf der anderen Seite in einer Zahlenvariablen den konvertierten Wert wissen müssen?

So löst .NET Framework das Problem: Der Typ `Int32` – unser Visual Basic-`Integer`-Datentyp also – stellt beispielsweise die statische Methode `TryParse` bereit, die als ersten Parameter die zu konvertierende Zeichenkette erwartet und als zweiten Parameter eine `Integer`-Variable `ByRef`, die innerhalb der Funktion auf den konvertierten Wert gesetzt wird und die `True` oder `False` (ob die Umwandlung überhaupt möglich war) als Funktionsergebnis zurückliefert. Das folgende Beispiel zeigt, wie das dann in der Praxis funktioniert:

```
Dim retWert As Integer
If Integer.TryParse("456123", retWert) Then
   MessageBox.Show("String konnte umgewandelt werden:" & retWert)
End If
```

HINWEIS `ByVal` und `ByRef` haben für die Übergabe von Parametern, bei denen es sich um Referenztypen handelt, übrigens auch Auswirkungen, allerdings nicht beim Verändern des Objektinhalts, sondern beim Neuzuweisen von Objektvariablen an andere Objektinstanzen. Wenn Sie eine Objektvariable, also eine Referenz, mit `ByVal` übergeben und innerhalb dieser Prozedur der Objektvariablen eine neue Instanz zuweisen, dann ändert sich auch die Instanz, auf die die Objektvariable in der aufrufenden Prozedur zeigt. Mit `ByRef` ist das nicht der Fall.

Konstruktoren und Standardinstanziierungen von Wertetypen

Da es keine Strukturinstanzen ohne Daten gibt, sorgt die CLR übrigens automatisch dafür, dass bei der Definition einer Struktur immer auch eine entsprechende Dateninstanz erstellt wird – ganz gleich, ob Sie *New* zur Instanziierung verwendet haben oder nicht. Auch hier zeigt ein Beispiel, was gemeint ist:

```
Dim EinWert As NumberSystems
Dim EineReferenz As ReferenzTyp

EinWert.Value = 10
EineReferenz.Value = 10
```

Diese Zeilen werden bis auf die letzte anstandslos verarbeitet. Bei der letzten tritt jedoch eine Ausnahme auf, weil Sie versuchen, die Eigenschaft eines Objekts zu verändern, das auf dem Managed Heap gar nicht existiert (Abbildung 12.1).

Abbildung 12.1 Versuchen Sie die Eigenschaft eines Objekts zu ändern, das nicht instanziiert wurde, erhalten Sie eine Ausnahme

Keine Code für Standardkonstruktoren bei Wertetypen

Strukturen in Visual Basic dürfen also nicht über Standardkonstruktoren verfügen. Ein Standardkonstruktor ist im Falle von Visual Basic eine `Sub New`, die keine Parameter entgegennimmt. Das folgende Konstrukt ist also beispielsweise fehlerhaft:

```
Public Structure TestValueType

    Private myTestEigenschaft As String

    'Fehler: Ein parameterloser Konstruktor, der nicht als "Shared" deklariert ist,
    'kann nicht in einer Struktur deklariert werden.
    Sub New()
        myTestEigenschaft = "Vorinitialisiert"
    End Sub
```

```
    'Das ist OK:
    Sub New(ByVal Vorgabe As String)
        myTestEigenschaft = Vorgabe
    End Sub

    Property TestEigenschaft() As String
        Get
            Return myTestEigenschaft
        End Get
        Set(ByVal Value As String)
            myTestEigenschaft = Value
        End Set
    End Property

End Structure
```

Strukturen bilden Wertetypen; und Wertetypen arbeiten, was Konstruktoren und Initialisierung der Instanz anbelangt, sehr abweichend von dem, was wir bislang über Referenztypen, die aus Klassen entstehen, kennen gelernt haben. Die CLR erlaubt es immer, eine Instanz eines Wertetyps zu erstellen; es gibt keine Möglichkeit zu verhindern, dass das geschieht (im Falle von Klassen reichte ja abweichend davon schon ein privater Konstruktor). Und, wir erinnern uns: Wertetypen sind deswegen in .NET Framework implementiert, weil wir als Entwickler Datenstrukturen benötigen, die extrem schnell arbeiten. Wir fassen also zusammen:

- Wertetypen können immer instanziiert werden
- Wertetypen sollten sehr schnell instanziiert werden

Und aus diesen beiden Punkten folgt: Strukturen dürfen in Visual Basic (auch nicht in C#) keinen Standardkonstruktor haben, weil das den Entwickler denken lassen könnte, er hätte Einfluss darauf, *wann* dieser Standardkonstruktor aufgerufen würde. Weil .NET Framework aus Performancegründen genau das aber nicht garantieren kann, nämlich dass die Konstruktoren auch zum Zeitpunkt der Instanziierung angewendet werden, darf es eben keine selbst erstellbaren Standardkonstruktoren geben. Die Instanziierung von Wertetypen, wie sie durch Strukturen definiert werden, geschieht anders als bei »normalen« Klassen, die mit `Class` definiert wurden. Klassen können ausschließlich durch das Schlüsselwort `New` instanziiert und anschließend verwendet werden. Strukturen müssen nicht mit `New` vor ihrer ersten Verwendung instanziiert werden – vielmehr kann ihre Instanziierung, wie wir schon kennen gelernt haben, implizit erfolgen, also ist beispielsweise das folgende Konstrukt durchaus erlaubt:

```
'Keine Instanziierung mit New...
Dim tvt As TestValueType
'...aber dennoch vorhanden:
tvt.TestEigenschaft = "Test"
```

Sobald ein Wertetyp, wie oben gezeigt, implizit definiert wird, sorgt .NET Framework dafür, dass alle seine Feldvariablen mit ihren Standardwerten vorbelegt werden (also beispielsweise `0` bei numerischen Typen, `False` für `Boolean`, etc.). Das geschieht aber nicht notwendigerweise dann, wenn der Datentyp instanziiert wird, sondern vielleicht erst bei seiner ersten Verwendung.

Das bedeutet aber wiederum, dass der Standardkonstruktor bei einer impliziten Verwendung unter Umständen nicht oder nicht zeitnah verwendet wird, und um Fehler von vornherein auszuschließen (nämlich, dass der Entwickler von den Standardwerten abweichende Initialisierungen im Standardkonstruktor vornimmt, die den Feldvariablen aber später gar nicht zugewiesen wurden), verbietet Visual Basic das Vorhandensein von Standardkonstruktoren bei Wertetypen von vornherein.

Wann Werte-, wann Referenztypen verwenden?

Sie haben bereits gelernt, dass Wertetypen in erster Linie aus Performancegründen geschaffen worden sind. Bei der Umsetzung von Algorithmen, die äußerst schnell agieren müssen, macht es also Sinn, Datentypen zu verwenden oder zu entwickeln, die sich quasi stromlinienförmig im Speicher bewegen können. Beispiele:

- Eine Grafik ist, wenn Sie von der Festplatte in den Arbeitsspeicher geladen wird, ein riesiger Speicherbereich, der natürlich als Referenztyp angelegt werden sollte. Koordinaten hingegen, die Positionen auf der Grafik beschreiben und häufig in Algorithmen zur Berechnung dieser Grafik einfließen, ergeben als Wertetypen natürlich mehr Sinn, denn mit ihnen muss schnell hantiert werden.
- Datensätze, die Sie zum Beispiel im Rahmen einer Kontaktmanagementanwendung im Speicher halten müssen, benötigen viel Platz; sie sollten daher als Referenztypen konzeptioniert werden. Eine Datenstruktur, die lediglich eine Beziehung zwischen einer Datensatz-ID und einem Suchschlüssel herstellt, könnte als Wertetyp besser konzeptioniert sein.
- Sollten Sie den Ehrgeiz entwickeln, einen eigenen Webbrowser zu programmieren, würden die Daten, die eine HTML-Seite ausmachen, natürlich als Referenztyp angelegt werden. Ein Element einer DNS-Liste, die Aufschluss zwischen IP-Adresse und Hostheader gibt, machte als Wertetyp vielleicht wieder mehr Sinn.
- Wertetypen sind auch dann sinnvoll, wenn Sie die vorhandenen primitiven Datentypen um weitere ergänzen wollen. Microsoft SQL Server kennt ab der 2008-Version beispielsweise geographische Daten; diese sind nicht nur komplett in Managed .NET-Code realisiert, sondern bei ihnen handelt es sich auch um Wertetypen auf Basis von Strukturen.
- Denkbar als Wertetypen sind auch neue numerische Typen, die andere Zahlensysteme.

Sie sehen: Wann immer kleine Datenmengen schnell verarbeitet werden sollen, bietet es sich an, Wertetypen auf Basis einer Struktur zu entwerfen. Wenn Datentypen komplexer werden oder selbst auf Referenztypen basieren, sollten Sie sich für Ihren Entwurf als Referenztyp auf Basis einer Klasse entscheiden.

Gezieltes Zuweisen von Speicherbereichen für Strukturmember mit den StructLayout- und FieldOffset-Attributen

Strukturen erlauben das gezielte Platzieren von Speicherbereichen für Feldvariablen. Wichtig dafür ist, dass Sie die Struktur mit einem speziellen Attribut markieren und dem Compiler damit »Bescheid geben«, wie er die Speicherbereiche der verschiedenen Feldvariablen überlappen soll.

Mit dieser Möglichkeit wird es zum Kinderspiel, mit den verschiedenen Wertigkeiten der Bytes von Integerwerten zu hantieren. Integerzahlen werden, wie Sie sicherlich wissen, aus Bytes »zusammengebaut«. Eine Zahl vom Datentyp `Short` besteht aus 2 Bytes, ein `Integer` aus 4 Bytes und ein `Long` aus 8 Bytes. Bei den

vorzeichenbehafteten Integerwerten wird das höchste Bit für das Vorzeichen verwendet (ist es gesetzt, ist der Wert negativ, ansonsten positiv). Relativ aufwändig ist es, Zahlentypen aus anderen größeren Zahlentypen zu »extrahieren«. Ein `Long` setzt sich beispielsweise aus zwei 32-Bit-Integerwerten zusammen – dem so genannten höherwertigen DWord (ein Word entspricht zwei Bytes, ein DWord – Double Word – zwei Words und damit vier Bytes) und dem niederwertigen DWord. Wenn Sie nun wissen möchten, wie der Zahlenwert des höherwertigen DWords lautet, können Sie ihn entweder berechnen oder nur geschickt aus dem Speicher lesen, was natürlich sehr viel schneller geht.

Bleibt die Frage: Wozu brauchen Sie das? 32-Bit-Grafiken beispielsweise speichern pro Pixel drei Farben und einen Alpha-Wert in insgesamt vier Bytes oder als vorzeichenloser 32-Bit-Integerwert (`UInteger` in Visual Basic). Auf die im Folgenden gezeigte Weise könnten Sie beispielsweise eine Struktur schaffen, die auf einem `Integer`-Wert basiert – und entsprechende Eigenschaften innerhalb der Struktur könnten ohne eine einzige Zeile Rechenaufwand die einzelnen Farbwerte eines Pixels als Byte zurückliefern.

Einige Betriebssystemfunktionen von Windows, die Sie auch durchaus aus Ihren .NET-Programmen heraus aufrufen können, verpacken verschiedene Parameter ebenfalls als Kombination in einem einzigen »großbittigen« Datentyp. Der eine Parameter liegt in den höherwertigen Bytes, der andere in den niederwertigen.

Ein Beispiel: Der `Long`-Wert (vorzeichenlos) $FFFFAABB00FF besteht aus zwei DWords: $0000FFFF sowie $AABB00FF. Um herauszufinden, wie der Wert des oberen DWords lautet, müssten Sie es bei einer Berechnung zunächst mit $FFFFFFFF00000000 ausmaskieren (eine logische AND-Operation durchführen) und dann den kompletten verbleibenden Wert um 32 Bits nach rechts verschieben. Oder aber Sie lesen die oberen 32 Bits direkt aus dem Speicher, weil Sie diesen Teil der `Long`-Variable auch noch gezielt einer `Integer`-Variable zuordnen konnten.

Dazu sind zwei Voraussetzungen notwendig: Sie müssen dafür sorgen, dass die Feldvariablen einer Struktur ihren benötigten Speicher exakt in der Reihenfolge definieren, in der sie im Quellcode angegeben werden. Dazu dient das `StructLayout`-Attribut. Und: Sie bestimmen mit dem `FieldOffset`-Attribut, an welcher Stelle der Speicher einer Feldvariable auf dem Prozessorstack beginnen soll (immer von 0 an gerechnet). Auf diese Weise können Sie dafür sorgen, dass sich auch Feldvariablen verschiedener Typen überlappen, und dann gezielt auf den Speicher der Feldvariablen eines bestimmten Typs, sozusagen unter »vorgegaukeltem Typdeckmantel«, also als anderer Typ, zugreifen.

Die prinzipielle Vorgehensweise beim Erstellen einer solchen Struktur zeigt das folgende Beispielprogramm. Mit ihr als Vorlage können Sie sie problemlos für eigene Bedürfnisse erweitern.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 12\StructLayout
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Die Struktur, die dort definiert wird, nennt sich `LongEx`, und sie erlaubt es, einen 64-Bit-Wert zu definieren und diesen wahlweise als vorzeichenbehafteten, vorzeichenlosen oder als Teilwert in Form von höher- und niederwertigen 32-Bit-Integerwerten (wahlweise vorzeichenbehaftet oder vorzeichenlos) abzurufen:

```
'Mitteilen, dass die Reihenfolge der
'Bytedefinitionen strikt einzuhalten ist!
<StructLayout(LayoutKind.Explicit)> _
Public Structure LongEx
    <FieldOffset(0)> Private myUnsignedLong As ULong
    <FieldOffset(0)> Private mySignedLong As Long
    <FieldOffset(0)> Private myLowUnsignedInt As UInteger
    <FieldOffset(4)> Private myHighUnsignedInt As UInteger
    <FieldOffset(0)> Private myLowSignedInt As Integer
    <FieldOffset(4)> Private myHighSignedInt As Integer
```

Mit dem `StructLayout`-Attribut am Anfang der Struktur bestimmen Sie, dass die Reihenfolge der Memberdefinition (oder besser die Reihenfolge der Festlegung derer Speicherplätze) strikt eingehalten werden soll. Mit dem `FieldOffset`-Attribut erreichen Sie anschließend das »Überlappen« der Speicherbereiche für die entsprechenden Feldvariablen. `FieldOffset` bestimmt dabei durch den angebbaren Parameter, an welcher Speicherstelle im Stack relativ zur Startposition der Struktur selbst die Daten eines Datentyps abgelegt werden.

WICHTIG Strukturen können auch Referenztypen aufnehmen. Da in diesem Fall allerdings Zeiger auf die eigentlichen Daten gespeichert werden, wäre die Gefahr groß, dass Sie durch `FieldOffset` einen Zeiger auf einen Referenztyp »verbiegen« – und das verbietet .NET Framework rigoros, da ein solches Vorgehen nicht nur typunsicheren Code erzeugen, sondern das gesamte verwaltete Speicherkonzept über den Haufen werfen würde. Wenn Sie also mit `FieldOffset` arbeiten, dürfen Sie als Feldvariablen auch nur ausschließlich Wertetypen einsetzen – was im Übrigen auch den Einsatz von Arrays ausschließt, da eine Array-Variable auch nur den Zeiger (die Referenz) auf die eigentlichen Array-Daten darstellt.

Die restlichen Eigenschaftenprozeduren dienen dann nur noch dem Auslesen der Feldvariablen auf die gewohnte Weise:

```
    Sub New(ByVal Value As ULong)
        myUnsignedLong = Value
    End Sub

    Sub New(ByVal Value As Long)
        mySignedLong = Value
    End Sub

    Sub New(ByVal value As UInteger)
        myLowUnsignedInt = value
    End Sub

    Sub New(ByVal value As Integer)
        myLowSignedInt = value
    End Sub

    Public Property Value() As ULong
        Get
            Return myUnsignedLong
        End Get
        Set(ByVal value As ULong)
            myUnsignedLong = value
        End Set
    End Property
```

```
    Public ReadOnly Property SignedLong() As Long
        Get
            Return mySignedLong
        End Get
    End Property

    Public ReadOnly Property HighUnsignedInt() As UInteger
        Get
            Return myHighUnsignedInt
        End Get
    End Property

    Public ReadOnly Property LowUnsignedInt() As UInteger
        Get
            Return myLowUnsignedInt
        End Get
    End Property

    Public ReadOnly Property HighSignedInt() As Integer
        Get
            Return myHighSignedInt
        End Get
    End Property

    Public ReadOnly Property LowSignedInt() As Integer
        Get
            Return myLowSignedInt
        End Get
    End Property
End Structure
```

Kapitel 13

Typenumwandlung (Type Casting) und Boxing von Wertetypen

In diesem Kapitel:

Konvertieren von primitiven Typen 456
Konvertieren von und in Zeichenketten (Strings) 458
Casten von Referenztypen mit DirectCast 461
Boxing von Wertetypen und primitiven Typen 463
Wertänderung von durch Schnittstellen geboxten Wertetypen 467

Typen können in vielen Fällen in andere Typen umgewandelt werden; diesen Vorgang, einen Typen in einen anderen umzuwandeln, nennt man neudeutsch auch »Type Casting«,[1] und dazu gehören:

- Das physische Umwandeln eines konkreten Werts in einen anderen Typ – dabei werden Daten verarbeitet, analysiert und an anderer Stelle neu gespeichert
- Das Zuweisen des Zeigers auf eine Klasseninstanz an eine andere Objektvariable anderen Typs, die aber ebenfalls Teil der Klassenerbfolge ist. Eine Objektvariable der Basisklasse `ErsteKlasse` referenziert dabei beispielsweise eine Instanz von `ZweiterKlasse`; eine Objektvariable der abgeleiteten Klasse `ZweiteKlasse` soll die Instanz nach der Umwandlung referenzieren.
- Der Vorgang des »Boxen« oder »Boxing« (etwa: *Schachteln, in eine Schachtel packen*) oder »Unboxing« (»Auspacken«). Beim Boxing wird ein Wertetyp in einen Referenztyp umgewandelt, sodass dieser anschließend auch durch eine Objektvariable einer Klasse der Klassenfolge referenziert werden kann. Boxing kann aber auch dann notwendig sein, wenn aus bestimmten Gründen ein Wertetyp wie ein Referenztyp behandelt werden soll, beispielsweise bei Arrays vom Typ `Object`.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 13\Casting
```

Konvertieren von primitiven Typen

In Visual Basic gibt es mehrere Möglichkeiten, einen primitiven Datentyp in einen anderen umzuwandeln. Die einfachste Vorgehensweise ist die einer direkten Zuweisung, die, `Option Strict On` vorausgesetzt, nicht mit allen Datentypen funktionieren kann. Ein Beispiel:

```
Dim EinInteger as Integer=1000
Dim EinLong as Long
'Int kann verlustfrei konvertiert werden; implizite Konvertierung ist möglich!
EinLong=EinInteger
```

Bei diesem Vorgang wird eine implizite Konvertierung des Werts einer `Integer`-Variablen in eine `Long`-Variable vorgenommen. Das ist implizit (also ohne weiteres Zutun) möglich, da bei diesem Vorgang niemals ein Verlust auftreten kann. `Long` speichert nämlich einen weit größeren Zahlenbereich als `Integer`; alle denkbaren `Integer`-Werte sind locker von `Long` speicherbar. Der Datentyp kann hier erweitert werden (*widening* auf Englisch). Andersherum sieht es hingegen schon schlechter aus:

```
Dim EinInteger As Integer
Dim EinLong As Long = Integer.MaxValue + 1L
'Long kann nicht verlustfrei konvertiert werden; implizite Konvertierung ist nicht möglich!
EinInteger = EinLong
```

[1] Von Engl. »to cast«, zu Deutsch »auswerfen«, »abgießen« (aus einer Form). Aber auch »eine Rolle besetzen«.

Wenn Sie diesen Code eingeben, meckert Visual Basic nach der Eingabe der letzten Zeile. Der Grund: Visual Basic kann eine verlustfreie Konvertierung nicht gewährleisten und nimmt Sie in die Verantwortung. Der Datentyp wird verkleinert oder geschmälert (englisch: *narrowing*). Sie müssen jetzt selbst Hand anlegen, und Ihnen stehen dazu mehrere Optionen zur Verfügung, um dennoch zum gewünschten Ziel zu gelangen:

- Sie verwenden `CInt`, um den `Long`-Wert in einen `Integer`-Typ umzuwandeln. Das sähe dann folgendermaßen aus:

```
'Int kann nicht verlustfrei konvertiert werden, explizite Konvertierung ist nötig!
EinInteger = CInt(EinLong)
```

- Sie verwenden `CType` für den gleichen Vorgang. `CType` arbeitet prinzipiell wie *CInt*, ist allerdings nicht auf `Integer` als Zieltyp beschränkt. Deswegen bestimmen Sie als zweiten Parameter, in welchen Typ Sie das angegebene Objekt umwandeln möchten:

```
'Int kann nicht verlustfrei konvertiert werden; explizite Konvertierung ist nötig!
EinInteger = CType(EinLong, Integer)
```

- Als letzte Option können Sie die `Convert`-Klasse von .NET Framework verwenden – allerdings ist dies auch die langsamste Methode. Seit Visual Basic 2005 sorgt der Visual Basic-Compiler nämlich dafür, dass mit `Cxxx` oder `CType` immer die schnellste Konvertierungsmöglichkeit zur Anwendung kommt. Visual Basic 2002 und 2003 waren an dieser Stelle längst nicht so hoch optimiert – in einigen Fällen war hier der Einsatz der `Convert`-Klasse den eingebauten Möglichkeiten mit `Cxxx` bzw. `CType` sogar vorzuziehen. Der Vollständigkeit halber – der Aufruf mit der bestimmt eineinhalb Mal langsameren `Convert`-Klasse gestaltet sich folgendermaßen:

```
'Int kann nicht verlustfrei konvertiert werden; explizite Konvertierung ist nötig!
EinInteger = Convert.ToInt32(EinLong)
```

Rein theoretisch gäbe es noch folgende Möglichkeit, die Konvertierung durchzuführen: Sie entscheiden sich für `Option Strict Off`. In diesem Fall kümmert sich Visual Basic zur Laufzeit um die Konvertierung in den richtigen Datentyp. Da Ihnen der Codeeditor in diesem Fall aber nicht einmal eine Warnung im Falle einer verlustmöglichen Konvertierung meldet, rate ich (schon wieder) dringend davon ab, von dieser Möglichkeit Gebrauch zu machen!

HINWEIS Wenn Sie die Definition des Ausgangswerts wie unten zu sehen abändern, erhalten Sie eine Ausnahme (Abbildung 13.1). Das liegt daran, dass Sie versuchen, einen Wert zu konvertieren, der sich schlicht und ergreifend nicht konvertieren lässt. Der zu konvertierende Wert wird mit

```
Dim EinLong As Long = Integer.MaxValue + 1L
```

auf den größtmöglichen mit dem `Integer`-Datentyp speicherbaren Wert plus eins festgelegt. Damit überschreitet er die zulässige »Wertekapazität« von `Integer`, und .NET Framework präsentiert Ihnen die Ausnahme.

```
Sub Main()

    Dim EinInteger As Integer
    'Dim EinLong As Long = 1000
    Dim EinLong As Long = Integer.MaxValue + 1L
    'EinInteger = EinLong
    EinInteger = CInt(EinLong)
    EinInteger = CType(EinLong,
    EinInteger = Convert.ToInt3

    Dim EinStringMitDatum As St
    Dim EinAndererStringMitDatu
    Dim EinStringMitDouble As S
    Dim EinDatum As DateTime
    Dim EinDouble As Double

    'Ein Array mit möglichen Da
    Dim DatumsformatierungsForm                                  ddMM"}
```

OverflowException wurde nicht behandelt.
Die arithmetische Operation hat einen Überlauf verursacht.
Hinweise zur Fehlerbehebung:
Dividieren Sie nicht durch 0.
Für diese Ausnahme die allgemeine Hilfe aufrufen.
Weitere Hilfeinformationen online suchen...
Aktionen:
Details anzeigen...
Ausnahmedetail in die Zwischenablage kopieren

Abbildung 13.1 Beim Type Casting müssen Sie natürlich darauf achten, dass sich ein Typ, was seine Größe oder Eigenschaften anbelangt, auch in einen anderen Typ casten lässt!

Konvertieren von und in Zeichenketten (Strings)

Eine Konvertierung von primitiven Datentypen ist natürlich nicht nur auf numerische Typen beschränkt. Viel interessanter ist die Konvertierung von einer Zeichenkette in einen numerischen Wert oder in einen Datumswert oder umgekehrt.

Grundsätzlich besteht auch hierbei die Möglichkeit, mit den bislang vorgestellten Verfahren eine Konvertierung durchzuführen. So haben Sie die Möglichkeit, beispielsweise einen String, der ein Datum als Zeichenkette speichert, in einen echten Datumswert umzuwandeln. Die Möglichkeiten dafür sind:

```
Dim EinStringMitDatum As String = "24.12.2003"
Dim EinDatum As DateTime

EinDatum = CDate(EinStringMitDatum)
EinDatum = CType(EinStringMitDatum, DateTime)
EinDatum = Convert.ToDateTime(EinStringMitDatum)
```

Konvertieren von Strings mit den Parse- und ParseExact-Methoden

Allerdings gibt es eine weitere Möglichkeit, die Ihnen eine größere Flexibilität zur Verfügung stellt. Die `DateTime`-Struktur[2] verfügt über die statische Methode `Parse`[3], die ebenfalls eine Zeichenkette, die ein Datum enthält, in einen `DateTime`-Wert umwandeln kann; sie bietet Ihnen aber eine wesentlich größere Flexibilität:

[2] Diese entspricht dem `Date`-Datentyp von Visual Basic übrigens exakt – es ist also völlig egal, ob Sie `Date.Parse` oder `DateTime.Parse` schreiben.

[3] Vom Englischen »to parse«, etwa: »analysieren«.

Die `Parse`-Funktion enthält mehrere Überladungen. Von der »einfachen« Version angefangen, können Sie einen so genannten »Format-Provider« angeben, mit dem Sie bestimmen, wie die Datumsvorgabe auszusehen hat, damit die Umwandlung gelingen kann. Zusätzlich lässt sich mit einem optionalen dritten Parameter ein gewisses Toleranzverhalten bei der Zeichenkettenanalyse festlegen.

Noch mehr Kontrolle erhalten Sie, wenn Sie die `ParseExact`-Methode verwenden. Diese erlaubt Ihnen auch zusätzlich noch, ein String-Array mit Eingabemustern als Richtlinie für die Stringanalyse zu geben. Möchten Sie beispielsweise ein Eingabefeld in einem Programm schaffen, in dem das Datum nicht starr im Format »dd.MM.yy« eingegeben werden muss, sondern – ergonomisch für den Benutzer – auch Eingabeformate wie »ddMMyy« oder »ddMM« möglich sind, verwenden Sie die `ParseExact`-Methode, um den Datumswert umzuwandeln. Dabei nimmt Ihnen .NET Framework alles an Analysearbeit ab und wandelt den String nach Ihren Vorgaben in einen Datumswert um oder generiert eine abfangbare Ausnahme, wenn der Benutzer eben nicht die Daten im entsprechenden Format eingegeben hat.

Das über Zeichenketten Gesagte gilt in gleichem Maße auch für die numerischen Datentypen. Auch sie verfügen über eine `Parse`-Methode zur Zahlenumwandlung, die wesentlich flexibler als die bisher vorgestellten Alternativen sind.

HINWEIS Eine `ParseExact`-Methode steht für numerische Datentypen nicht zur Verfügung.

Ein Beispiel für die Anwendung dieser Methoden finden Sie im nächsten Abschnitt.

Konvertieren in Strings mit der ToString-Methode

Das Gegenstück zu `Parse` bildet die `ToString`-Methode. Grundsätzlich können Sie – was primitive Datentypen wie `Date`, `Integer`, `Double`, `Decimal` etc. anbelangt – jeden Variableninhalt mit der `ToString`-Methode in eine Zeichenkette umwandeln. Gerade die Formatierung von Zahlen und Datumswerten wird aber von .NET Framework besonders unterstützt – ein Blick in die Hilfe eines jeweiligen Objekts für die aktuelle Funktionsweise ist auf jeden Fall hilfreich und angebracht.

Die folgenden kleinen Programmauszüge zeigen Ihnen im Schnellüberblick, wie der Einsatz mit den Methoden `Parse`, `ParseExact` und `ToString` aussehen kann. Sie werden sie aber nicht nur hier, sondern auch an vielen anderen Stellen in diesem Buch noch zu sehen bekommen.

```
Dim EinStringMitDatum As String = "24122003"
Dim EinAndererStringMitDatum As String = "2412"
Dim EinStringMitDouble As String = "1.123,23 €"
Dim EinDatum As DateTime
Dim EinDouble As Double

'Ein Array mit möglichen Datumsformaten für Date.ParseExact
Dim DatumsformatierungsFormate() As String = New String() {"ddMMyyyy", "ddMM"}

'Beispiele für Datumskonvertierung
EinDatum = DateTime.ParseExact(EinStringMitDatum,
                               DatumsformatierungsFormate,
                               Nothing,
                               Globalization.DateTimeStyles.AllowWhiteSpaces)
Console.WriteLine("Datum " & EinDatum.ToString("ddd, dd.MMM.yyyy"))

EinDatum = DateTime.ParseExact(EinAndererStringMitDatum, _
                               DatumsformatierungsFormate, _
```

```
                                Nothing,
                                Globalization.DateTimeStyles.AllowWhiteSpaces)
Console.WriteLine("Datum " & EinDatum.ToString("ddd, dd.MMM.yyyy"))

'Zahlenbeispiele
EinDouble = Double.Parse(EinStringMitDouble, Globalization.NumberStyles.Currency)
Console.WriteLine("Wert " & EinDouble.ToString("#,###.## Euro"))
```

Abfangen von fehlschlagenden Typkonvertierungen mit TryParse oder Ausnahmehandlern

Wenn Ihre Anwendungen später beim Kunden laufen, müssen Sie natürlich mit dem »Fehlverhalten« der Benutzer rechnen oder anders gesagt: dass Benutzer in Ihren Anwendungen alles Mögliche einzugeben versuchen, nur nicht das, was Sie sich beim Programmieren der Anwendungen vorgestellt haben. Bei Eingaben von Datumswerten oder Zahlen ist es also angebracht, Fehler abzufangen und den Benutzer im Bedarfsfall darauf aufmerksam zu machen.

Dazu stehen Ihnen zwei Möglichkeiten zur Verfügung:

- Sie arbeiten mit `Parse` oder `ParseExact` (bei Datumswerten) und schließen diese Methode in einen `Try … Catch`-Block ein. Lösen `Parse` oder `ParseExact` eine Ausnahme aus, was bei falschem Format der Fall ist, dann fangen Sie diese Ausnahme im `Catch`-Block ab, geben eine entsprechende Fehlermeldung aus und erlauben dem Benutzer, einen weiteren Eingabeversuch durchzuführen.
- Sie arbeiten mit der ebenfalls statischen `TryParse`-Methode, die es seit Visual Basic 2005 auch für alle numerischen Datentypen gibt. `TryParse` liefert nicht den eigentlich geparsten Wert als Funktionsergebnis zurück, sondern `True` bzw. `False`. Lautete das Ergebnis `True`, war das Parsen der Zeichenfolge erfolgreich. `TryParse` übergeben Sie gleichzeitig eine entsprechende Variable, die bei erfolgreichem Parsen das Ergebnis aufnimmt (die Variable wird also explizit `ByRef` übergeben – um ein wenig an das letzte Kapitel zu erinnern).

Das folgende Beispiel demonstriert diese beiden Vorgehensweisen:

```
'So können fehlerhafte Eingabeformate abgefangen werden:

'Version 1: Mit TryParse (gibt's für Datum- und numerische Typen)
If Double.TryParse("1.234,56", EinDouble) Then
    Console.WriteLine("Zeichenkette konnte geparst werden. Ergebnis: " & EinDouble.ToString)
Else
    Console.WriteLine("Zeichenkette konnte nicht geparst werden!")
End If

'Version 2: Mit einer Ausnahmebehandlung
Try
    'Das kann nicht klappen!
    EinDatum = DateTime.ParseExact("2005 12 24",
                        DatumsformatierungsFormate,
                        Nothing,
                        Globalization.DateTimeStyles.AllowWhiteSpaces)

Catch ex As Exception
    Console.WriteLine("Das Parsen generierte eine Ausnahme:" & vbNewLine &
        ex.Message)
End Try
```

Casten von Referenztypen mit DirectCast

Wenn Sie mit Polymorphie arbeiten, dann müssen Sie vergleichsweise häufig Referenztypen konvertieren, die in einer Erbfolge stehen. Ein Beispiel: Sie haben eine Klasse `AbgeleiteteKlasse`, die von `EineKlasse` erbt. Sie definieren eine Objektvariable vom Typ `EineKlasse`, die Sie aber mit der Instanz von `AbgeleiteteKlasse` belegen. Die Gründe, das zu tun, haben mit der Nutzung von Polymorphie zu tun – Beispiele dafür haben Sie schon kennen gelernt.

Nun brauchen Sie im Laufe des Programms aber eine Funktion, die nur von `AbgeleiteteKlasse` zur Verfügung gestellt wird. Da es sich um eine Instanz dieser Klasse handelt, steht diese Funktion, die Sie brauchen, zwar prinzipiell zur Verfügung, doch Sie kommen über die verwendete Objektvariable nicht an die Funktion heran. `DirectCast` bietet Ihnen hier die Möglichkeit, den Verweis auf die Objektinstanz auf eine Objektvariable vom »richtigen« Typ einzurichten; die Funktion lässt sich anschließend aufrufen.

Ein weiteres Beispiel soll diesen Sachverhalt verdeutlichen:

```
'Klassen-Casting
Dim locEineKlasse As EineKlasse
Dim locAbgeleiteteKlasse As AbgeleiteteKlasse = New AbgeleiteteKlasse

'Implizites Casting möglich, denn es geht in der Erbhierarchie Richtung Basisklasse
locEineKlasse = locAbgeleiteteKlasse

'Geht nicht, Funktion nicht vorhanden.
'locEineKlasse.AddValues()

'Geht auch nicht; es geht in der Erbhierarchie nach unten, und dann
'kann nicht implizit konvertiert werden:
'locAbgeleiteteKlasse = locEineKlasse

'So gehts:
locAbgeleiteteKlasse = DirectCast(locEineKlasse, AbgeleiteteKlasse)
locAbgeleiteteKlasse.AddValues()

Console.WriteLine("KAbgeleitet: " & locAbgeleiteteKlasse.ToString())
Console.WriteLine("KWertepaar: " & locEineKlasse.ToString())
```

Sie könnten übrigens in diesem Beispiel ebenfalls wieder `CType` einsetzen, indem Sie die Zeile

```
locAbgeleiteteKlasse = DirectCast(locEineKlasse, AbgeleiteteKlasse)
```

durch diese

```
locAbgeleiteteKlasse = CType(locEineKlasse, AbgeleiteteKlasse)
```

ersetzen. Allerdings empfiehlt es sich der besseren Übersichtlichkeit wegen, bei Referenztypen grundsätzlich `DirectCast` zu verwenden; der Compiler wandelt intern `CType` in `DirectCast` um; da können Sie direkt `DirectCast` verwenden, und Sie sehen so auf den ersten Blick, dass es sich nicht um eine Datenkonvertierung im Sinne von primitiven Datentypen, sondern nur um eine Art Hinweis an den Compiler handelt, den Typen, um den es sich dreht, als den Typen anzusehen, den Sie mit `DirectCast` angeben.

TryCast – herausfinden, ob gecastet werden kann

Da eine Objektvariable einer Basisklasse immer auch auf eine Instanz einer ihrer Ableitungen verweisen darf, kann es in bestimmten Kontexten sinnvoll sein herauszufinden, ob eine bestimmte Objektvariable auf eine Ableitung in der Erbfolge zeigt.

Ein Beispiel: Mit der `Handles`-Anweisung lässt sich bestimmen, dass eine Methode die Ereignisse gleich mehrerer Quellen behandelt. In einer Windows Forms-Anwendung können Sie so beispielsweise in einer Methode das `Click`-Ereignis von mehreren, unterschiedlichen Steuerelementen in einem Rutsch bearbeiten:

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 13\TryCast
```

Abbildung 13.2 Mit *TryCast* oder *IsAssignableFrom* finden Sie heraus, ob eine Basisklassenobjektvariable auf eine Instanz eines abgeleiteten Typs verweist – bei Steuerelementen brauchen Sie das recht oft

Wenn Sie dieses Beispiel starten, können Sie sowohl auf eine der TextBoxen als auch auf einen der Buttons im Formular klicken und werden jeweils über eine entsprechende MessageBox darüber informiert, welchen Typ von Steuerelement Sie angeklickt haben. So weit ist das noch nichts Besonderes. Erst ein Blick in den Programmcode offenbart, was diese Demo anders macht:

```
Public Class Form1

    Private Sub Control Click(ByVal sender As Object,
                              ByVal e As System.EventArgs)
                          Handles TextBox1.Click, TextBox2.Click,
                          Button1.Click, Button2.Click

        'Wir behandeln hier alle 4 Click-Ereignisse, in
        'sender kann also auf eine TextBox oder einen Button verweisen:

        'TryCast-Demo:
        Dim button = TryCast(sender, Button)
        If button IsNot Nothing Then
            MessageBox.Show("Eine Schaltfläche wurde angeklickt!")
        End If
```

Die `Click`-Ereignisse aller Steuerelemente werden hier in einer Methode behandelt. Um herauszufinden, welches Steuerelement und damit welcher von `Object` (`sender`) abgeleitete Typ tatsächlich das Ereignis ausgelöst hat, *versucht* die Anwendung, mit `TryCast` die Ereignisquelle in eine Objektvariable vom Typ `Button` zu casten. Falls das gelingt, ist `button` nicht `Nothing`, und das Programm gibt eine entsprechende Meldung aus.

IsAssignableFrom – Test auf erlaubtes Casting ohne zusätzliche Objektvariable

Falls es die Umstände erfordern, dass Sie lediglich auf die Casting-Möglichkeit *testen*, aber keine zusätzliche Objektvariable dafür verwenden wollen, können Sie auf folgendes Verfahren zurückgreifen. Die Methode `IsAssignableFrom` eines Typs untersucht einen anderen Typ auf die *Möglichkeit* seines Castings in den Ausgangstyp. Dazu ermitteln Sie zunächst mit `GetType` das Typobjekt des Typs, der das Ziel des Casting-Vorgangs sein soll, und übergeben dessen `IsAssignableFrom`-Methode den zu überprüfenden Typ, etwa wie im folgenden Beispiel zu sehen (das Beispiel ist die Fortführung des Beispiels aus dem vorherigen Abschnitt):

```
    Private Sub Control_Click(ByVal sender As Object,
                              ByVal e As System.EventArgs) _
                          Handles TextBox1.Click, TextBox2.Click,
                          Button1.Click, Button2.Click
.
.
.
        'IsAssignableForm-Demo
        If GetType(TextBox).IsAssignableFrom(sender.GetType) Then
            MessageBox.Show("Eine TextBox oder eine Ableitung wurde angeklickt!")
        End If
    End Sub
End Class
```

Dieses Beispiel überprüft, ob die Objektvariable `sender` (ist vom Typ `Object`) sich in eine `TextBox` casten ließe, *ohne* dass Sie dazu, wie bei `TryCast`, eine zusätzliche Objektvariable eines bestimmten Typs verwenden müssten.

Boxing von Wertetypen und primitiven Typen

Wenn Sie mit Wertetypen arbeiten, ganz gleich ob mit primitiven Datentypen wie beispielsweise `Integer`, `Long` oder `Double` oder mit selbst gestrickten Strukturen, werden Sie niemals in Verlegenheit kommen, Probleme wie im vorherigen Beispiel lösen zu müssen, denn Wertetypen können Sie nicht vererben.

Allerdings gibt es eine Ausnahme. Dass alle Wertetypen von `Object` abgeleitet sind, gilt für Wertetypen gleichermaßen. Das hat aber zur Folge, dass Sie zwar keine eigene Erbfolge auf einem Wertetyp basierend erstellen können, aber `Object` und `ValueType` an sich bereits in der Erbfolge vorhanden sind, heißt: Eine `Object`-Objektvariable müsste in der Lage sein, auf einen Wertetyp zu verweisen – doch das klingt schon wie ein Gegensatz in sich.

.NET Framework löst dieses Problem durch eine Sonderregel des Common Type Systems, die mit »Boxing«[4] oder – eingedeutscht – »Boxen« bezeichnet wird.

Dazu ein kleines Beispiel: Nehmen wir an, Sie haben eine Struktur entwickelt und ihr den Namen `Matrjoschka`[5] gegeben. Dieser Wertetyp dient als Träger einer bestimmten Datenstruktur, von der Sie mehrere Elemente erstellen wollen und diese in einem Array speichern. Nun soll dieses Array nicht nur `Matrjoschka`-Werte aufnehmen, sondern auch für zukünftige Erweiterungen vorbereitet sein und deswegen auch andere Typen aufnehmen können. Also definieren Sie das Array nicht vom Typ `Matrjoschka`, sondern vom Typ `Object` und können die verschiedensten Elementtypen darin speichern. Das folgende Beispiel demonstriert diesen Vorgang und verwendet dazu den Wertetyp `Matrjoschka`, der – um das Beispiel simpel zu halten – nichts weiter macht, als eine Generationsnummer zu speichern:

```
Structure Matrjoschka
    Private myGeneration As Integer

    Sub New(ByVal Generation As Integer)
        myGeneration = Generation
    End Sub

    Property Generation() As Integer
        Get
            Return myGeneration
        End Get
        Set(ByVal Value As Integer)
            myGeneration = Value
        End Set
    End Property

End Structure
```

Zufallszahlen mit der Random-Klasse

Da wir es gerade brauchen, hier ein kleiner Ausritt in Sachen Zufallszahlen: In Visual Basic 6.0 war es üblich, Zufallszahlen mit der `RND`-Funktion zu erzeugen. Die .NET Framework Class Library bietet zu diesem Zweck eine spezielle Klasse an, die die alte Funktion ersetzt. Wenn Sie diese Klasse instanziieren, geben Sie in ihrem Konstruktor einen Ausgangswert an, der die Basis für eine Zufallszahlenfolge darstellt, die im Folgenden generiert werden soll. Damit schon dieser Wert zufällig ist, ergibt es Sinn, hier wirklich »zufällige« Werte, wie beispielsweise die aktuelle Millisekunde (Sie werden mit großer Wahrscheinlichkeit immer eine andere »treffen«) oder eine Mauszeigerposition zu verwenden. ▶

[4] Von engl. »to box«, etwa »einpacken«, »verpacken«; »the box«: »der Behälter«, »die Box«. Kann aber auch (hat nichts mit dem Thema zu tun, ist aber dennoch interessant) »Anhieb« bedeuten. Das ist die Einkerbung in einem zu fällenden Baum, um dessen Fallrichtung zu bestimmen und den Stamm vor dem Splittern zu bewahren…

[5] Sie kennen die kleinen, russischen Figuren, von denen die äußere in sich geschachtelt mehrere kleine aufnehmen kann? Gut. Das ist der Name dieser Figur, und das, was sie macht, kann gut mit Boxing verglichen werden.

Wenn Sie die Klasse instanziiert haben, können Sie ihre Instanz verwenden. Sie haben mehrere Methoden, mit denen Sie Zufallszahlen ermitteln können, nämlich die `Next`-, die `NextBytes`- und die `NextDouble`-Methode.

Die `Next`-Methode liefert dabei den jeweils nächsten zufälligen Integerwert zurück. Im Bedarfsfall können Sie bei dieser Methode auch noch die Grenzen angeben, innerhalb derer sich die Zufallszahlen bewegen dürfen. Mit der `NextDouble`-Methode erhalten Sie eine Zufallszahl zwischen 0 und 1, also einen Bruch. Diese Methode kommt der ursprünglichen `RND`-Funktion am nächsten. Die `NextBytes`-Methode liefert Ihnen ein definierbar großes `Byte`-Array mit Zufallszahlen zurück.

Die wirklich einfache Verwendung dieser Klasse demonstriert ebenfalls das folgende Beispiel.

Im Hauptprogramm des Beispiels findet anschließend erst das eigentlich interessante Geschehen statt. Das Programm erstellt 10 Elemente vom Typ `Matrjoschka` und weist ihnen als Generationsnummer mithilfe der `Random`-Klasse zufällige Werte im Integerwertebereich zu. Anschließend findet es heraus, welches `Matrjoschka`-Element des Arrays die größte Generationsnummer hatte.

```
Module Module1
    Sub Main()
        'Boxen von Wertetypen
        Dim locObjectArray(9) As Object
        Dim locRandom As New Random(Now.Millisecond)
        Dim locMaxValue As Integer

        For locCount As Integer = 0 To 9
            'Implizites Casting ist möglich, es geht in der Erbhierarchie nach oben
            'aber Deckung! - Hier wird geboxt!
            locObjectArray(locCount) = New Matrjoschka(locRandom.Next)
        Next

        'Rausfinden, welches das Objekt mit der höchsten Generationsnummer war
        For locCount As Integer = 0 To 9
            'Zwar sind nur Matrjoschka-Werte im Array drin, doch das Array
            '"kann" nur Objects. Die Generation-Eigenschaft steht nicht zur
            'Verfügung, deswegen funktioniert diese Zeile nicht:
            'locMaxValue = locObjectArray(locCount).Generation
            Dim locMatrjoschka As Matrjoschka

            'Entboxen - aus dem referenzierten Wertetyp wird wieder ein "echter" Wertetyp
            locMatrjoschka = DirectCast(locObjectArray(locCount), Matrjoschka)

            'Jetzt kommen wir an die Generation-Eigenschaft heran:
            If locMaxValue < locMatrjoschka.Generation Then
                locMaxValue = locMatrjoschka.Generation
            End If
        Next

        Console.WriteLine("Die höchste Generationsnummer war: " & locMaxValue.ToString())

        Console.WriteLine()
        Console.WriteLine("Return drücken zum Beenden!")
        Console.ReadLine()
    End Sub
End Module
```

Was passiert hier genau? Mit einer einfachen »Umreferenzierung« von Verweisen in einen Speicherbereich wie bei Klassen ist es bei Wertetypen nicht getan, denn: Es gibt nichts, das auf etwas zeigen könnte. Sie erinnern sich: Die Daten von Wertetypen befinden sich für die direkte Verwendung auf dem Stack. Beim Boxing von Wertetypen werden diese deshalb kopiert und dabei wie bei einem Referenztyp auf den Managed Heap geschrieben. Die Objektvariable kann jetzt auf diesen Speicherbereich zeigen und obwohl es sich um die Daten eines Wertetyps handelt, diese doch referenzieren.

Beim »Entboxen« passiert genau das Gegenteil: Die Wertetypvariable nimmt jetzt die Daten entgegen, die sich auf dem Managed Heap befinden und auf die die Objektvariable zeigt, die zum Boxing verwendet wurde. Die Daten werden also aus dem Managed Heap wieder zurück auf den Stack kopiert.

HINWEIS Typisierte Wertetypen-Arrays bilden eine Art Zwitterstellung: Wenn Sie beispielsweise ein Array aus `Integer`-Werten erstellen, dann landen diese Werte natürlich auf dem Managed Heap, aber nicht geboxt! Das heißt: Jedes Array-Element besteht tatsächlich aus der Anzahl der Bytes, die erforderlich ist, ein Element auf dem Managed Heap zu speichern. Anders ist das, würden Sie eine Reihe von `Integer`-Werten in einem `Object`-Array speichern. Jetzt würden die einzelnen Werte wie ein Referenzwert auf dem Managed Heap erzeugt, also quasi instanziiert werden, und jedes Element des `Object`-Arrays würde nur den Zeiger jedes einzelnen Werts, nicht aber den Wert direkt speichern. Sie können sich vorstellen, dass das einerseits viel mehr Speicher benötigt und andererseits viel länger dauert, sowohl das Abspeichern der Werte im Array als auch das Zurückholen. Hier muss nämlich beim »Entboxen« mehr Aufwand betrieben werden: Erst wird die Speicheradresse der eigentlichen Daten aus dem Array-Element gelesen, und diese können mit der so ermittelten Adresse anschließend gelesen werden.

TIPP Viele Auflistungen in .NET Framework basieren auf `Object`-Arrays, und deswegen werden bei der Verwendung vieler Auflistungen Werte bei ihrer Zuweisung geboxt. Generischen Auflistungen, wie beispielsweise die `List(Of )` oder die `Collection(Of )`, über die Sie genau wie über Auflistungen selbst in den folgenden Kapiteln noch viel mehr erfahren werden, basieren direkt auf typisierten Wertetypen-Arrays und brauchen ihre einzelnen Elemente deswegen nicht zu boxen. Sie sind dadurch wesentlich schneller und benötigen lange nicht so viel Speicher; bei numerischen Werten oder Datumswerten oftmals nur die Hälfte; auf 64-Bitsystemen, die doppelt so breite Zeiger (8 statt 4 Bytes pro Element!) aufweisen, noch weniger.

Übrigens: Was DirectCast nicht kann

`DirectCast` kann Referenztypen, die einen Wertetyp boxen, *zurück* in den Wertetyp casten (»Entboxen«), es kann allerdings keine Wertetypen casten. Das kann auch nicht funktionieren, denn schließlich können Sie in der Vererbungshierarchie nur in Richtung »weitere Ableitung« casten. Da Wertetypen an sich aber nicht vererbt werden können, bleibt dieser Weg verschlossen.

`DirectCast` kann natürlich auch keine primitiven Datentypen in andere primitive Datentypen konvertieren; hier verwenden Sie, wie schon gesagt, die Methoden `Cxxx`, `CType` (je nach Datentyp), `Parse`, `ParseExact` und `TryParse` oder (für alle Datentypen) die `Convert`-Klasse.

Wann wird und wann wird nicht geboxt?

Wir haben gesehen, dass Wertetypen dann geboxt werden, wenn sie – beispielsweise für den Fall, dass Sie eine ganze Menge von ihnen in `Object`-Arrays und damit auch in bestimmten Auflistungen speichern müssen – in ein Objekt umgewandelt werden müssen. Es gibt aber auch noch einige andere Fälle, wo das Boxing nicht direkt offensichtlich ist. Beispielsweise:

- Geboxt wird, wenn Sie einen Wertetyp als Parameter an eine Methode übergeben, die ein Objekt erwartet. In diesem Fall wird natürlich ihr Wertetyp zuvor in ein Objekt geboxt, und anschließend erst wird die Referenz des geboxten Wertetyps als Parameter weitergereicht.
- Geboxt wird auch, wenn Sie eine Schnittstelle in einem Wertetyp implementiert haben und jetzt irgendein öffentliches Member über diese Schnittstelle aufrufen. In diesem Fall muss die CLR ebenfalls den Wertetyp boxen, da es sich bei Schnittstellen per Definition um Referenztypen handelt.
- Geboxt wird ebenfalls, wenn Sie `GetType` für einen Wertetypen aufrufen, denn `GetType` ist eine von `Object` geerbte Methode, die nur mit Referenztypen funktioniert.
- Geboxt wird jedoch nicht, wenn Sie eine normale Methode eines Wertetyps aufrufen. Die CLR ist in der Lage, Methoden, die an einem Wertetyp »hängen«, direkt anzuspringen.
- Geboxt wird auch nicht, wenn Sie die `ToString`-Funktion aufrufen. Man könnte das meinen, denn die `ToString`-Funktion ist ja prinzipiell überschreibbar, und wenn diese Methode überschrieben *würde*, dann müsste natürlich eine unter Umständen überschreibende Methode und nicht die Ausgangsroutine aufgerufen werden. Dazu wäre es allerdings notwendig, in der Methodentabelle des überschreibenden Typen nachzuschlagen, wozu wiederum dessen Typ ermittelt werden müsste, was im Umkehrschluss nur dann ginge, läge dieser als Referenztyp vor, und das würde bedeuten: *Boxing*. Dann wiederum: Wertetypen lassen sich nicht vererben, und nur aus diesem Grund ist `ToString` eines Wertetyps direkt aufrufbar, ohne dass die CLR den Wertetyp zuvor in einen Referenztyp umwandeln müsste.

Wertänderung von durch Schnittstellen geboxten Wertetypen

Beim Boxen und dem anschließenden Entboxen von Wertetypen kann es mitunter zu Verhaltensweisen kommen, die auf den ersten Blick nicht wirklich nachzuvollziehen sind. Sie haben eben in der oben stehenden Auflistung gelesen, dass ein Zugreifen auf Member eines Wertetyps automatisch zur Folge hat, dass der betroffene Wertetyp sofort in ein Objekt geboxt wird. Das folgende Beispiel demonstriert, was dieses Verhalten für Auswirkungen haben kann, und dass sich bestimmte Verhaltensweisen, obwohl es sich um dieselbe Wertetypinstanz handelt, stark unterscheiden.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 13\Boxing
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
Interface IMussValueHaben
    Property Value() As Integer
End Interface

Module mdlMain

    Sub Main()
        Dim EinWertetyp As New Wertetyp(10)
        Dim EinVerweistyp As New Verweistyp(10)
```

```
        EinWertetyp.Value = 20
        Console.WriteLine(EinWertetyp.Value) ' 20 -> direkt geändert!
        WertetypÄndern(EinWertetyp)
        Console.WriteLine(EinWertetyp.Value) ' 20 -> in der Stackkopie geändert
        VerweistypÄndern(EinVerweistyp)
        Console.WriteLine(EinVerweistyp.Value) ' 30 -> auf dem Managed Heap geändert

        Dim EinInterface As IMussValueHaben = EinWertetyp
        ÜberInterfaceÄndern(EinInterface)
        Console.WriteLine(EinInterface.Value) ' 40, wird auf dem Managed Heap geändert
        Console.WriteLine(EinWertetyp.Value) ' 20, haben nichts miteinander zu tun

        Console.WriteLine()
        Console.WriteLine("Taste drücken zum Beenden")
        Console.ReadLine()

    End Sub

    'Ändert die Eigenschaft des Wertetyps.
    Sub WertetypÄndern(ByVal EinWertetyp As Wertetyp)
        EinWertetyp.Value = 30
    End Sub

    'Ändert die Eigenschaft des Verweistyps.
    Sub VerweistypÄndern(ByVal EinVerweistyp As Verweistyp)
        EinVerweistyp.Value = 30
    End Sub

    'Ändert die Eigenschaft über das Interface.
    Sub ÜberInterfaceÄndern(ByVal EinInterface As IMussValueHaben)
        EinInterface.Value = 40
    End Sub
End Module

'Testklasse des Wertetyps
Structure Wertetyp
    Implements IMussValueHaben

    Dim myValue As Integer

    Sub New(ByVal Value As Integer)
        myValue = Value
    End Sub

    Property Value() As Integer Implements IMussValueHaben.Value
        Get
            Return myValue
        End Get
        Set(ByVal Value As Integer)
            myValue = Value
        End Set
    End Property
End Structure

'Testklasse des Verweistyps
Class Verweistyp
    Implements IMussValueHaben

    Dim myValue As Integer
```

```
    Sub New(ByVal Value As Integer)
        myValue = Value
    End Sub

    Property Value() As Integer Implements IMussValueHaben.Value
        Get
            Return myValue
        End Get
        Set(ByVal Value As Integer)
            myValue = Value
        End Set
    End Property
End Class
```

Dieses Beispiel definiert eine Schnittstelle, einen Referenztyp und einen Wertetyp. Beide Typen binden die ganz am Anfang definierte Schnittstelle ein. Insgesamt drei Prozeduren dienen dazu, die Inhalte der übergebenen Objekte auf sehr einfache Weise zu ändern.

Die erste Wertänderung ist klar: Die Eigenschaft des Wertetyps wird hier direkt geändert, folglich spiegelt sich der geänderte Wert der Eigenschaft auch beim Ausgeben wider.

Die zweite Wertänderung ist hingegen schon nicht mehr so offensichtlich – aber ein Fall, den wir bereits besprochen haben. Hier wird eine Kopie des Werts auf dem Stack abgelegt; Änderungen auf dem Stack sind nur temporär. Es gibt keine Verbindung zum Objekt, also wird der ursprünglich zugewiesene Wert beibehalten.

Anders ist das bei der Werteänderung des Referenztyps durch die Methode `VerweistypÄndern`. Es gibt keine Kopie des Objekts, sondern nur verschiedene Zeiger auf die Daten im Managed Heap. Ergo: Eine Änderung in der Prozedur spiegelt die Änderung der Objektvariablen auch in dem Programmteil wider, der die Prozedur aufruft.

Interessant wird es, wenn die Verwendung einer Schnittstellenvariablen ins Spiel kommt, die einen Wertetyp referenziert. Hier wird der Wertetyp nämlich von vornherein in einen Referenztyp umgewandelt; seine Daten landen auf dem Managed Heap. Die Änderungen erfolgen durch die Unterroutine genau dort und spiegeln sich deshalb auch durch den Ursprungsverweis auf das Objekt des Managed Heap wider – durch die Objekt-(Schnittstellen-)Variable `EinInterface`. Aber: Nur durch diese Variable greifen Sie auf die »Version« auf dem Managed Heap zu. Die Ursprungsvariable, aus der die Kopie auf dem Managed Heap entstanden ist, steht in keiner Verbindung zur Objektvariablen. Die Ursprungsvariable `EinWertetyp` behält deswegen auch ihren ursprünglichen Wert.

HINWEIS Sie sehen: Boxing ist deswegen eine oft mit Vorsicht zu genießende Geschichte, weil die Boxing-Vorgänge oftmals auf den ersten Blick nicht transparent sind und eine Menge Fehlerpotenzial bilden können. Gerade beim Zugreifen auf Wertetypen mit Schnittstellenvariablen müssen Sie besonders vorsichtig sein, wie das letzte Beispiel, wie ich finde, eindrucksvoll demonstriert hat.

Oberste Prämisse bei der Verwendung von Wertetypen ist: Nutzen Sie sie immer dann, wenn es wirklich auf Geschwindigkeit ankommt, und nur dann, wenn die Typen die Sie entwickeln, nicht viel Speicherplatz benötigen. Und darüber hinaus: Verwenden Sie sie am besten direkt; wenn Sie viele Wertetypen in Auflistungen speichern müssen, bearbeiten Sie einen Wertetyp der Liste nicht durch irgendwelche Tricks, sondern casten Sie ihn in eine Wertetypvariable, bearbeiten Sie die Instanz dort, und boxen Sie sie anschließend wieder zurück in das entsprechende Array-Element.

Kapitel 14

Beerdigen von Objekten – Dispose, Finalize und der Garbage Collector

In diesem Kapitel:

Der Garbage Collector – die Müllabfuhr in .NET 474
Die Geschwindigkeit der Objektbereitstellung 478
Finalize 479
Dispose 484
Gezieltes Freigeben von Objekten mit Using 494

.NET Framework stellt Entwicklern eine außerordentlich solide Plattform für das Erstellen und das Verwenden der unterschiedlichsten Typen zur Verfügung. Wer in seinem Entwicklerleben aber schon mit Speicherlöchern und Pufferüberläufen zu kämpfen hatte, der weiß eine Funktionalität von .NET besonders zu schätzen: dessen »Müllabfuhr« nämlich. Müllabfuhr heißt auf Englisch *Garbage Collection*, und genau das ist der technisch korrekte Ausdruck für den Teil der Common Language Runtime, die Objekte, die nicht mehr benötigt werden, identifiziert und bei Gelegenheit (wenn Zeit) oder Bedarf (wenn Platzmangel) entsorgt.

Um diese Vorgehensweise zu erörtern und deutlich zu machen, möchte ich noch einmal auf ein Beispiel zu sprechen kommen, das ursprünglich der Demonstration eines ganz anderen Themas diente. Sie erinnern sich noch an die Klasse `DynamicList` im Abschnitt zur Polymorphie, die die Beispielanwendung nutzte, um die Artikeldatensätze (`ShopItem`) zu speichern? Neben der eigentlichen Fähigkeit, eine Methode zu implementieren, die eine dynamische Vergrößerung des benötigten Speichers demonstriert, zeigt dieses Beispiel noch etwas anderes – was allerdings mehr eine Fähigkeit von .NET Framework selbst ist: nämlich den nicht benötigten Speicher wieder freizugeben.

Rufen Sie sich die `Add`-Methode dieser Klasse noch mal in Erinnerung:

```
Sub Add(ByVal Item As ShopItem)

    'Prüfen, ob aktuelle Arraygrenze erreicht wurde
    If myCurrentCounter = myCurrentArraySize - 1 Then
        'Neues Array mit mehr Speicher anlegen,
        'und Elemente hinüberkopieren. Dazu:

        'Neues Array wird größer:
        myCurrentArraySize += myStepIncreaser

        'Temporäres Array erstellen.
        Dim locTempArray(myCurrentArraySize - 1) As ShopItem

        'Elemente kopieren
        'Wichtig: Um das Kopieren müssen Sie sich,
        'anders als bei VB6, selber kümmern!
        For locCount As Integer = 0 To myCurrentCounter
            locTempArray(locCount) = myArray(locCount)
        Next

        'Temporäres Array dem Memberarray zuweisen.
        myArray = locTempArray
    End If

    'Element im Array speichern.
    myArray(myCurrentCounter) = Item

    'Zeiger auf nächstes Element erhöhen.
    myCurrentCounter += 1

End Sub
```

Schauen Sie sich diesen Codeblock noch einmal an, aber dieses Mal unter einem anderen Aspekt. Dieses Mal steht nicht der Speicherplatz im Vordergrund, der benötigt wird, und auch nicht die Art und Weise, wie Arrays wachsen können, sondern der Speicher, der durch denselben Vorgang überflüssig wird.

Abbildung 14.1 Was passiert mit den nach der Zuweisung im leeren Raum stehenden Array-Elementen?

Arrays in .NET sind keine Werte, sondern Verweistypen. Benötigter Speicher für alles, was von `System.Array` abgeleitet ist – und dazu zählen auch Arrays, die Sie durch `Dim` deklarieren –, wird also auf dem Managed Heap reserviert. Im Codeauszug des Beispielprogramms gibt es zwei entscheidende Zeilen, die eigentlich ein »Speicherleck«, besser bekannt unter dem neudeutschen Begriff »Memory Leak«, verursachen würden, würden wir nicht in Verbindung mit .NET Framework programmieren. Abbildung 14.1 macht das Problem deutlich. Zunächst gibt es das Array und den auf dem Managed Heap dafür reservierten Speicherbereich, aber das Array ist nunmehr zu klein geworden. Also definiert die Prozedur ein neues Array und nimmt dafür die Variable `locTempArray` zu Hilfe. Nun passiert das Entscheidende: `locTempArray` wird `myArray` zugewiesen, der Zeiger auf den Speicherbereich der entsprechenden Elemente wird dabei quasi »verbogen«. `myArray` zeigt anschließend auf die Array-Elemente, auf die kurz zuvor noch `locTempArray` zeigte. Die Adressen auf die Elemente[1] auf die von `myArray` verwiesen wurde, liegen nun unbrauchbar, da nicht mehr referenziert, irgendwo im Speicher – was passiert jetzt mit ihnen?

[1] Wichtig: Es sind natürlich nur die Adressen, die hier brachliegen, nicht die Elemente selbst. Beim Kopieren der Elemente haben wir ja nicht wirklich die Inhalte der Elemente kopiert, sondern nur die Adressen auf die Elemente. Beim Umbiegen der Arrayzeiger bleiben natürlich nur die 4 Zeiger auf die Elemente übrig. Die Elemente verwenden wir aber jetzt weiter – nämlich als Bestandteil des Arrays mit nunmehr 8 Elementen.

Erinnern wir uns, wie das bei COM (dem »Vorläufer« von .NET, eine Technologie, die Sie immer noch bewusst oder unbewusst benutzen, wenn Sie – auch in .NET – beispielsweise die Möglichkeit nutzen, Word oder Excel »fernzusteuern«) geregelt war. Bei COM gab es für jedes Objekt einen Referenzzähler. Bei der ersten Zuweisung an eine Objektvariable wurde der Zähler auf Eins gesetzt. Mit jeder weiteren Zuweisung an eine Variable – also mit jeder weiteren Referenzierung – wurde dieser Zähler um eins erhöht. Trat nun der umgekehrte Fall ein, das heißt, einer Variablen die zuvor das Objekt referenzierte, wurde ein anderes Objekt oder `Nothing` zugewiesen, oder das Programm verließ den Gültigkeitsbereich der Variablen, sodass sie aus diesem Grund das Objekt nicht mehr referenzieren konnte, wurde der Referenzzähler um eins verringert. Wurde er 0, dann konnte das Objekt entsorgt werden. Es wurde zu diesem Zeitpunkt nicht länger benötigt, da es von keiner Stelle des Programms aus mehr referenziert war.

Dieses Verfahren hatte allerdings zwei Nachteile: Zum einen kostete das Prinzip des Referenzzählers Rechenzeit. Der andere Nachteil war das Problem der so genannten Zirkelverweise: Ein Objekt, das auf ein Objekt zeigte, das seinerseits wieder auf das Ausgangsobjekt zeigte, führte dazu, dass der Referenzzähler niemals null werden konnte. Selbst wenn es in diesem Fall keine Referenzierung mehr durch das eigentliche Programm gab, so referenzierten sich die Objekte dennoch selbst. Ein Speicherleck war oft genug die Folge.

.NET Framework oder um genau zu sein, die Common Language Runtime, löst dieses Problem, indem sie ein komplett anderes Verfahren anwendet, Objekte zu entsorgen.

Der Garbage Collector – die Müllabfuhr in .NET

Den vorhandenen Speicher des Managed Heap teilen sich alle Assemblys, die in einer so genannten *Application Domain* (»Anwendungsdomäne«, kurz »AppDomain«) laufen. Normale Windows-Anwendungen, die man parallel startet, werden durch verschiedene, streng voneinander abgeschottete Prozesse isoliert. Ein Prozess kann unter normalen Umständen nicht auf einen anderen Prozess zugreifen, auch die Daten verschiedener Prozesse sind integer und können bestenfalls durch so genannte Proxys (Stellvertreter) untereinander ausgetauscht werden.

AppDomains in .NET, die durch die Common Language Runtime verwaltet werden, erlauben das gleichzeitige Ausführen mehrerer Anwendungen in *einem* Prozess. Die CLR garantiert dabei, dass die Anwendungen ebenso isoliert und ungestört laufen können, wie das bei einem Windows-Prozess der Fall wäre. Jede AppDomain verwaltet einen Speicherbereich, in dem die notwendigen Daten der Assemblys und Programme,[2] die innerhalb der AppDomain laufen, abgelegt werden. Dieser Speicherbereich wird Managed Heap genannt, und mit ihm haben wir uns bislang schon einige Male beschäftigt.

Wenn der Speicher im Managed Heap knapp wird (und auch sonst schon einmal aus anderen Gründen), dann startet ein Prozess, der im wahrsten Sinne des Wortes der Müllabfuhr im echten Leben entspricht: Während dieses Prozesses – also der Garbage Collection – werden Objekte im Speicher identifiziert, die nicht mehr in Gebrauch sind. Dies trifft für ein Objekt dann zu, wenn es durch keine Objektvariable der Anwendung mehr referenziert wird:

[2] Streng genommen ist auch ein Programm eine Assembly.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 14\GCEvents
```

```
Public Class Form1

    Private mykontakt As Kontakt


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

        'Bleibt erhalten.
        Dim kontakt1 As New Kontakt With {.Nachname = "Wördehoff",
                                          .Vorname = "Angela",
                                          .PLZ = "99999",
                                          .Ort = "Weißichabersagsnicht"}

        'Wird gleich entsorgt.
        Dim kontakt2 As New Kontakt With {.Nachname = "Löffelmann",
                                          .Vorname = "Klaus",
                                          .PLZ = "59555",
                                          .Ort = "Lippstadt"}

        mykontakt = kontakt1

        'Nur zum Testen, sonst nicht machen!
        GC.Collect()

    End Sub

    Private Sub Button2_Click(ByVal sender As System.Object,
                              ByVal e As System.EventArgs) Handles Button2.Click
        'Nur zum Testen, sonst nicht machen!
        GC.Collect()
    End Sub
End Class

Public Class Kontakt
    Public Property Nachname As String
    Public Property Vorname As String
    Public Property PLZ As String
    Public Property Ort As String

    Protected Overrides Sub Finalize()
        Debug.Print("Finalizing:" & Nachname)
    End Sub

End Class
```

Wenn Sie das Programm starten, sehen Sie ein Formular mit zwei Schaltflächen. Klicken Sie auf *Objekte erstellen*, dann werden die beiden Testobjekte `kontakt1` (*Angela Wördehoff*) und `kontakt2` (*Klaus Löffelmann*) erstellt. `Kontakt1` wird anschließend der Klassenvariablen `mykontakt` zugewiesen, und auch wenn der Programm-Scope die Methode verlässt, bleibt damit eine Referenz auf das Objekt erhalten. Ein anschließender Klick auf *Garbage Collector auslösen* führt dann auch dazu, dass im Ausgabefenster von Visual Studio (im Bedarfsfall mit Strg+Alt+O aktivieren) der Text *Finalizing:Löffelmann* ausgegeben wird.[3] Zum Objekt, das zuvor von `kontakt2` referenziert wurde, gibt es nämlich keine Referenz mehr, also entsorgt der Garbage Collector das Objekt und ruft dazu dessen `Finalizer` auf.

Der Garbage Collector arbeitet auch bei Anwendungen, die in kurzer Zeit sehr viele entsorgbare Objekte hervorbringen, mit sehr großer Geschwindigkeit.

Wie der Garbage Collector arbeitet

Die hohe Geschwindigkeit, mit der der Garbage Collector (GC) arbeitet, ist insbesondere auch darauf zurückzuführen, dass der GC die zu testenden Objekte in Generationen klassifiziert. Bei der Entwicklung des GC-Algorithmus nahm man an, dass Objekte, die beim Start einer Applikation erzeugt werden, länger im Speicher verbleiben als solche, die irgendwann zwischendurch oder lokal in Prozeduren generiert werden. Diese Annahme führt zu dem Schluss, dass es Sinn ergibt, bei der Objektentsorgung eine Klassifizierung der Objekte in eben diese Generationen zur Optimierung des GC-Algorithmus vorzunehmen.

Der Garbage Collector markiert Objekte nicht nur für die weitere Instandhaltung, sondern er stattet sie auch mit einem Zähler aus, der aussagt, wie oft ein Objekt für die Entsorgung durch den Garbage Collector getestet wurde. Je öfter der Garbage Collector das Objekt bereits »besucht« und nicht entsorgt hat, desto älter ist logischerweise das Objekt (und umso höher ist demzufolge auch seine Generationsnummer), aber desto unwahrscheinlicher wird es auch, dass das Objekt in einem erneuten GC-Lauf entsorgt werden wird.

Das Ganze läuft also – vereinfacht dargestellt – folgendermaßen ab:

- Es wird Speicher für ein neues Objekt benötigt. Der Speicherbedarf auf dem Managed Heap erreicht dabei eine bestimmte Größe, die sich in etwa im Bereich moderner Prozessorcache-Größen befindet. Dann wird ein Garbage Collection-Durchgang ausgelöst.
- Alle Objekte werden als »untersucht« markiert, und es wird geschaut, welche entsorgt werden können. Generation 0-Objekte, die noch gültig sind, werden damit zu Generation 1-Objekten. Schon seit .NET 1.0 funktioniert das standardmäßig schon auf einem Thread, der parallel zum Hauptanwendungsthread läuft. Sollte es zu entsorgende Objekte geben, hält der Garbage Collector – vereinfacht gesprochen – alle laufenden Threads an und entsorgt die Objekte, auf die nicht mehr referenziert wird und die deshalb nicht mehr benötigt werden. Dieser Prozess ist so hoch optimiert, dass eine Generation 0-Garbage Collection in der Regel im Millisekundenbereich geschieht. Auf diese Weise wird Platz geschaffen. Die zur Generation 1 promovierten, verbleibenden Objekte werden aus Performancegründen bei späteren regulären (kleinen) Läufen außen vor gelassen – der Garbage Collector nimmt nämlich zurecht an, dass Objekte, die nicht beim ersten Mal entsorgt wurden, auch beim nächsten Durchlauf noch mit großer Wahrscheinlichkeit in Verwendung sind.

[3] Dazu überschreibt die Klasse, die wir als Objektlieferant verwenden, ihre `Finalize`-Methode. Wir missbrauchen hier die `Finalize`-Methode quasi, um herauszufinden, wann das Objekt entsorgt wird. Machen Sie das nicht in Ihren eigenen Programmen! Mehr zu `Finalize` erfahren Sie weiter unten in diesem Kapitel in einem eigenen Abschnitt.

HINWEIS Seit .NET 4.0 läuft ein Garbage Collector nicht nur gleichzeitig zum eigentlichen Programm, sondern als so genannter *Hintergrundthread*, bei dem trotz laufender Garbage Collection sogar Speicher für neue Objekte generiert werden kann, ohne dass der Garbage Collector das eigentliche Programm (samt seiner Threads) anhalten muss, wie es bei älteren .NET Framework-Versionen noch der Fall war. Nur bei einem *vollen* Garbage Collector-Durchlauf muss das Anhalten noch passieren, sodass mit .NET 4.0 typische Clientanwendungen (beispielsweise Windows Forms oder WPF-Anwendungen) noch zügiger und störungsfreier laufen können. Für Serveranwendungen (ASP.NET-Anwendungen) gibt es diese Optimierung leider nicht.

- Irgendwann ist eine weitere Schwelle mit Speicherbedarf für neue Objekte erreicht, und erst jetzt werden sowohl Generation 0- als auch Generation 1-Objekte unter die Lupe genommen, die bislang aus Performancegründen bei Generation-0-Durchläufen keine Beachtung mehr gefunden haben. Prinzipiell passiert hier nun das Gleiche wie in der Generation-0-Garbage Collection, nur dass dieses Mal auch Objekte entsorgt werden können, die sich bereits in der Generation 1 befunden haben.
- Das Gleiche passiert schließlich auch bei einer weiteren Schwelle, etwa in der 16 MB-Größenordnung. Immer noch verbleibende Objekte von Generation 1 werden nun Generation 2, Generation 0 wird wieder zu Generation 1. Mit Generation 2 ist dann Schluss; der .NET Garbage Collector kennt nur diese 3 Generationen 0, 1 und 2.

Die Vorteile dieser mehrstufigen Vorgehensweise liegen auf der Hand: Die richtige Annahme, dass Objekte, die erst vor kurzem angelegt worden sind, auch nur kurz benötigt werden (Generation 0-Objekte), bringt einen enormen Geschwindigkeitsvorteil. Wirklich große Speicherblöcke müssen nur verschoben werden, wenn bestimmte Schwellen im MB-Bereich überschritten werden. Zudem wird bei den Schwellen Rücksicht auf die Prozessorcaches genommen, denn die Schwellen sind in etwa auf deren Größen abgestimmt.

Leider wirft diese Vorgehensweise wieder ein Problem ganz anderer Art auf. Objekte können nicht wissen, *wann* sie entsorgt werden – denn nur die CLR entscheidet, wann ein GC-Durchlauf stattfindet (mit einer Ausnahme):

- Die Common Language Runtime fährt herunter. Das passiert in der Regel dann, wenn eine .NET-Applikation beendet wird. Da mehrere AppDomains ihrerseits in einem Prozess laufen können und das Beenden einer AppDomain nicht unweigerlich zum Entladen der CLR führen muss, wird der Garbage Collector ebenfalls ausgeführt, wenn eine AppDomain entladen wird.
- Der Speicher wird knapp, entweder weil es zu viele Objekte gibt oder weil der Windows-Speicher aus anderen Gründen knapper wird. Der Garbage Collector startet dann, um zu sehen, ob Generation 0-Objekte entsorgt werden können und macht im Bedarfsfall davon Gebrauch. Das kann auch dann passieren, wenn Windows-Speicher insgesamt zu knapp wird, denn die Common Language Runtime lässt sich über diesen Zustand von Windows informieren, sie muss diesen nicht notwendigerweise selbst feststellen.
- Der Garbage Collector ist in der AppDomain gezwungenermaßen durch die Anweisung *GC.Collect()* gestartet worden – aber ganz wichtig: Nicht zu Hause nachmachen, liebe Kinder, denn Microsoft rät davon dringend ab. Nur in ganz wenigen Ausnahmefällen kann es Sinn ergeben, den Garbage Collector selbst anzustoßen.[4]

[4] Ein Szenario ist beispielsweise: Sie müssen auf eine Assembly eines Drittherstellers zurückgreifen, die bei bestimmten Objekten deren Dispose-Methode hätte aufrufen sollen, es aber nicht getan hat.

COM hatte beim Entsorgen eines Objekts, das nicht mehr benötigt wurde, einen eindeutigen Vorteil. Wurde die letzte Referenz aufgelöst und der Referenzzähler stand auf 0, dann trat das `Terminate`-Ereignis ein, und das Objekt konnte zur »richtigen« Zeit die notwendigen Schritte einleiten, um sich zu entsorgen.

Normalerweise ist es gar kein Problem, dass ein Objekt nicht weiß, dass es entsorgt wird. Wenn es weg ist, dann ist es eben weg. Wichtig, den Zeitpunkt seiner Entsorgung zu kennen, wird es für ein Objekt erst dann, wenn es Aufräumarbeiten erledigen muss, und zwar nicht hinsichtlich der eigenen Speicherverwaltung (denn wenn es andere Objekte referenziert, sorgt der Garbage Collector ja ebenfalls für deren Entsorgung), sondern hinsichtlich der Freigabe von Ressourcen, auf die der Garbage Collector keinen Zugriff hat.

Dies wurde übrigens früher oder wird heute noch bei anderen OOP-Sprachen mit einem Destruktor erledigt. Genau wie wir den Konstruktor als »Ereignisprozedur« kennen gelernt haben, die ausgeführt wird, wenn ein Objekt erstellt wird, so gelangt der Destruktor zur Ausführung, wenn das Objekt zerstört wird.

Das sind beispielsweise Fälle, in denen das Objekt ein Handle[5] auf eine bestimmte Geräte- oder Betriebssystemressource erhalten hat. Damit dieses Handle wieder freigegeben werden kann – eine geöffnete Datei beispielsweise sollte geschlossen werden –, muss das Objekt die dafür erforderlichen Aktionen spätestens kurz vor dem Zeitpunkt, an dem es vom Garbage Collector zerstört wird, durchführen.

Genau das geht nicht mehr in .NET. Denn Objekte können dort nicht voraussehen oder den genauen Zeitpunkt erfahren, wann sie entsorgt werden. Es gibt allerdings die Möglichkeit, dass Objekte Kenntnis darüber erhalten, *dass* sie entsorgt werden, und dann die notwendigen Schritte einleiten, um Ressourcen freizugeben, die sie belegen.

Daher spricht man in .NET übrigens auch von »nicht deterministischen Destruktoren«, es ist also nicht voraussagbar (determiniert), wann der Destruktor läuft. Er wird nämlich dann ausgeführt, wenn es dem GC passt, und wir wissen nicht, wann es ihm passt.

Die Lösung zu diesem Problem liegt in der Implementierung einer Schnittstelle, die sich `IDisposable` nennt. Aber wie gesagt: Sie benötigen das im Regelfall nur dann, wenn Sie andere als die .NET-Ressourcen freigeben müssen. Ein Beispiel dafür und wie eine Klasse sich dann hilft, indem sie diese `IDisposable`-Schnittstelle implementiert, finden Sie im Abschnitt »Dispose« ab Seite 484.

Die Geschwindigkeit der Objektbereitstellung

.NET bzw. die .NET-Infrastruktur hatte seit ihrer ersten Veröffentlichung mit dem Vorurteil der langsamen Geschwindigkeit zu kämpfen. Nun liegt es in der Natur von Vorurteilen, dass an einigen von ihnen immer auch ein Quäntchen Wahres dran ist. So bildet das On-Demand-Kompilieren durch den JITter der CLR zwar sicherlich eine Technik mit vielen Vorteilen, aber sicherlich nicht in Sachen Geschwindigkeit. Doch man sollte gerade in Sachen Geschwindigkeit auch einmal eine Lanze für die .NET-Infrastruktur brechen.

Denn es gibt es einige Vorteile der .NET-Infrastruktur in puncto Geschwindigkeit, und einer liegt beim Schaffen von Speicherplatz für ein neues Objekt. Und seien wir ehrlich: Es ist in fast jedem Szenario besser, Objekte so schnell wie möglich zu erstellen, als sie so schnell wie möglich zu entsorgen. Und warum ist das Erstellen der Objekte so viel schneller als sagen wir beim herkömmlichen Runtime Heap eines typischen C-basierten Systems?

[5] Eine vom Betriebssystem erteilte Kennung zur Nutzung einer bestimmten Ressource (Datei, Bildschirm, Schnittstellen, spezielle Windows-Betriebssystemobjekte etc.).

Bei einem C-Runtime-Heap liegen alle speicherreservierten Objekte (man sagt auch *allokierte* Objekte) als verkettete Liste vor. Wenn nun Speicherplatz für ein neues Objekt benötigt wird, muss die C-Runtime durch diese Liste iterieren, und falls Sie zwischen zwei Objekten ausreichenden Speicherplatz findet, bricht sie die Kette an dieser Stelle auf und platziert das neue Objekt an dieser Stelle.

In .NET Framework funktioniert das Reservieren des Speichers anders. Es gibt einen Adresszeiger, der auf den Speicherplatz zeigt, an dem das jeweils nächste Objekt abgelegt werden soll; neue Objekte werden also nicht irgendwo mittendrin platziert, sondern immer am Ende der Liste der Objekte im Managed Heap. Dieser Adresszeiger stellt also den jeweils nächsten Speicherstart für ein Objekt jederzeit zur Verfügung. Das Reservieren des Speichers kostet also, anders als bei den gängigen C-Runtime-Systemen, in .NET Framework keine erwähnenswerte Zeit, und das ist ein klarer Vorteil, den die CLR geschwindigkeitstechnisch für sich verbuchen kann.

In C-Runtime-Systemen kann es aus diesem Grund auch passieren, dass Objekte, die nacheinander erstellt werden, speichertechnisch weit voneinander entfernt liegen. Auch das passiert bei .NET Framework nicht. Objekte, die nacheinander erstellt werden, liegen auch hintereinander im Speicher.

Das ist natürlich hinsichtlich der Wahrscheinlichkeit sinnvoll, wie und wann für Objekte Speicher erstellt wird. Wenn Sie Speicher für eine Bitmap reservieren, und lassen wir dabei jetzt mal die Tatsache außen vor, dass dafür systembedingt auch Speicher anderen Typs reserviert werden muss, dann ist es auch wahrscheinlich, dass im gleichen Kontext, auch zeitlich, Objekte für Fonts und Pinsel erstellt werden müssen. Und vor allen Dingen, dass der Speicher dieser Objekte physisch in einem der schnellen Caches des Prozessors liegt. Bislang scheint also die .NET Framework-Speicherverwaltung nur Positives zu bringen, wenn es da nicht ein kleines Problem geben würde: Uns steht zwar inzwischen sehr, sehr viel Speicher zur Verfügung, aber dennoch ist dieser Speicher nicht unerschöpflich. Irgendwann muss eben ein Aufräumprozess stattfinden – die Garbage Collection –, und der kostet Zeit.

Aber immerhin: Der Algorithmus arbeitet eben mit den schon angesprochenen Generationen, und der Generation 0-Algorithmus ist so ausgelegt, dass er in der Regel noch nicht »anspringt«, solange sich der benötigte Speicher noch in einer Größenordnung befindet, die eine Unterbringung im schnellen Cache des Prozessors ermöglicht.

Der Kompromiss, den wir Entwickler eingehen müssen, um in den Genuss der extrem schnellen Speicheranforderung zu kommen, gepaart mit dem Vorteil, dass die Objekte, die wir gleichzeitig benutzen, möglichst eng beieinander und damit mit großer Wahrscheinlichkeit wieder im Prozessorcache liegen, ist damit durchaus vertretbar, wie ich finde.

Finalize

Wenn ein Objekt vom Garbage Collector zur Entsorgung markiert wurde, dann ruft der Garbage Collector in der Regel die `Finalize`-Methode des Objekts auf, bevor er den Speicher des Objekts endgültig freigibt. Schon die `Object`-Klasse hat `Finalize` implementiert, und da alle Klassen von `Object` abgeleitet sind, hat jedes Objekt in .NET eine `Finalize`-Methode.[6]

[6] Wobei die in `Object` enthaltene `Finalize`-Methode streng genommen gar nicht im Rahmen des Garbage Collectors aufgerufen wird, da diese ohnedies nichts durchführt; der GC-Algorithmus findet vielmehr heraus, ob ein »neues« `Finalize` implementiert wurde, und `Finalize` wird nur dann aufgerufen, wenn die `Finalize`-Methode überschrieben wurde.

Die `Finalize`-Methode in ihrer Grundimplementierung von `Object` macht überhaupt nichts. Sie ist in erster Linie einfach nur vorhanden, und das bedeutet, dass ein Objekt die `Finalize`-Methode überschreiben muss, wenn es eine eigene Funktionalität für seine »Entsorgungsvorbereitung« implementieren will.

Der Finalizer ist also im Prinzip der (leider eben nicht deterministische) Destruktor von .NET Klassen.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 14\Finalize01
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
Module mdlMain

    Sub Main()
        Dim locTest As New Testklasse("Erste Testklasse")
        Dim locTest2 As New Testklasse("Zweite Testklasse")
        locTest = Nothing
        locTest2 = Nothing

        'GC.Collect()

        'einen Moment (eine Sekunde, genaugenommen) warten
        Threading.Thread.Sleep(1000)

        Console.WriteLine("Beide Objekte sind nun nicht mehr in Verwendung!")
    End Sub

End Module

Class Testklasse

    Private myName As String

    Sub New(ByVal Name As String)
        myName = Name
    End Sub

    Protected Overrides Sub Finalize()
        MyBase.Finalize()
        Console.WriteLine(Me.myName & " wurde entsorgt")
    End Sub
End Class
```

TIPP Starten Sie dieses Programm mit Strg + F5, also ohne Debuggen, damit das Konsolenfenster nach dem Beenden des Programms nicht einfach wieder verschwindet. Sie würden das Ergebnis sonst nicht sehen können.

Wenn Sie dieses Programm starten, dann stellen Sie fest, dass die Meldung »Beide Objekte sind nun nicht mehr in Verwendung!« zuerst ausgegeben wird. Erst anschließend erscheinen die Texte, die anzeigen, dass die beiden verwendeten Objekte finalisiert worden sind:

```
Beide Objekte sind nun nicht mehr in Verwendung!
Zweite Testklasse wurde entsorgt
Erste Testklasse wurde entsorgt
```

Die Ursache dafür liegt schon fast auf der Hand: Der Garbage Collector arbeitet in diesem Programm nicht, während es läuft, und bei einem Speicheraufkommen von nur ein paar Byte hat er dafür auch gar keinen Grund. Das Finalisieren der Objekte findet dennoch statt, und zwar beim Beenden der Anwendung. Zu diesem Zeitpunkt ist die letzte Zeile des eigentlichen Programms aber längst verarbeitet worden; in diesem Fall war es die Codezeile, die den Meldungstext auf den Bildschirm ausgegeben hat. Eine andere Ausgabe erscheint, wenn Sie das Kommentarzeichen vor der Zeile

```
'GC.Collect()
```

weglassen. Starten Sie das Programm anschließend, ändert sich die Ausgabe in:

```
Zweite Testklasse wurde entsorgt
Erste Testklasse wurde entsorgt
Beide Objekte sind nun nicht mehr in Verwendung!
```

WICHTIG Im Beispielprogramm habe ich die `Console`-Klasse in der `Finalize`-Methode wie selbstverständlich verwendet. Machen Sie das nicht. Mal ganz davon abgesehen, dass Sie in der `Finalize`-Methode keine wie auch immer gearteten Bildschirmausgaben mehr machen sollten, um die Finalisierung so schnell wie möglich hinter sich zu bringen, können Sie sich auch nicht sicher sein, ob Objekte, die Sie verwenden, zu diesem Zeitpunkt noch bestehen.

Wann Finalize nicht stattfindet

Unter Umständen verursachen Sie beim Verwenden bestimmter Objekte in `Finalize`, dass neue Objekte während des Vorgangs entstehen, die ihrerseits wiederum neue Objekte anlegen, usw. Diese Objekte müssen natürlich anschließend ebenfalls finalisiert werden. Im schlimmsten Fall lösen Sie damit eine solch enorme Kaskade von neuen Objekten aus, dass diese alle niemals finalisiert werden könnten. Doch ihre Anwendung hängt sich deshalb nicht auf, denn .NET Framework hat zu diesem Zweck ein paar Sicherheitsmaßnahmen vorgesehen.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 14\FinalizeNoNo01
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei). Denken Sie daran, dass dieses Beispiel zeigt, was Sie in Ihren Anwendungen niemals machen sollten!

```
Module mdlMain

    Sub Main()
        Dim locTest As New Testklasse("Testklasse")
    End Sub
```

```
End Module

Class Testklasse

    Private myName As String

    Sub New(ByVal Name As String)
        myName = Name
    End Sub

    Sub WriteText()
        Console.Write(myName)
    End Sub

    Protected Overrides Sub Finalize()
        MyBase.Finalize()
        Dim locTemp As New Testklasse("locTemp")
        WriteText()
        Console.WriteLine(" wurde entsorgt")
    End Sub
End Class
```

Wenn Sie dieses Programm starten, wird eine Reihe von Meldungen ausgegeben. `Finalize` selbst legt dabei dummerweise eine neue Instanz von `Testklasse` an, um eine Meldung auszugeben. Diese Instanz muss natürlich ebenfalls finalisiert werden, und sie legt ihrerseits wieder eine neue `Testklasse`-Instanz an usw. Der GC erkennt nach einer Weile, dass der Finalisierungsprozess genau das Gegenteil von dem bewirkt, was er eigentlich bewirken sollte, es entstehen nämlich immer mehr Objekte, und der Speicherbedarf wächst und wächst. Er bricht das Finalisieren nach ein paar Sekunden schlicht und ergreifend ab.

Ein weiteres schlechtes Beispiel ist das Folgende (wenn Sie es unbedingt selbst probieren wollen: Sie finden es im Begleitdateienverzeichnis unter *\FinalizeNoNo02*). Es legt zwar keine Unmenge von neuem Speicher an, verbraucht für den Finalisierungsprozess aber einfach zu viel Zeit. Der GC wird nach vergleichsweise kurzer Zeit ungeduldig und bricht den gesamten Finalisierungsprozess ebenfalls wieder ab. Das im Beispielprogramm zuerst deklarierte Objekt bekommt keine Chance mehr, finalisiert zu werden.

```
Module mdlMain

    Sub Main()
        '"Normales" Objekt, könnte problemlos finalisiert werden.
        Dim locTest1 As New Testklasse(False, "Erstes Testobjekt")
        'Der Störenfried, da Warteschleifenflag gesetzt.
        Dim locTest2 As New Testklasse(True, "Zweites Testobjekt")
    End Sub

End Module

Class Testklasse

    'Dieses Flag steuert den Einstieg in die Warteschleife.
    Private myWaitInFinalize As Boolean
    'Eine Eigenschaft zum Unterscheiden von Klasseninstanzen
    Private myName As String
```

```
    'Flag fürs Warten und den Namen definieren.
    Sub New(ByVal WaitInFinalize As Boolean, ByVal Name As String)
        myWaitInFinalize = WaitInFinalize
        myName = Name
    End Sub

    Protected Overrides Sub Finalize()
        MyBase.Finalize()

        'Nur wenn das Flag bei New gesetzt
        'wurde, in die Warteschleife springen.
        If myWaitInFinalize Then

            Dim locSecs As Integer
            Dim lastSec As Integer
            lastSec = Now.Second
            Do
                'Jede Sekunde eine Meldung ausgeben.
                If lastSec <> Now.Second Then
                    lastSec = Now.Second
                    locSecs += 1
                    Console.WriteLine("Warte bereits {0} Sekunden", locSecs)
                    'Nach 60 Sekunden wäre Schluss.
                    If locSecs = 60 Then Exit Do
                End If
            Loop

        End If
        'Erfolgreich finalisiert --> Meldung ausgeben
        Console.WriteLine("Objekt {0} wurde finalisiert!", myName)
    End Sub
End Class
```

Für den Fall, dass Sie eine eigene Finalisierungslogik in Ihren Klassen implementieren müssen – und bitte: immer nur mithilfe der Implementierung von `IDisposable` wie im nächsten Abschnitt beschrieben –, sind also folgende Punkte wichtig:

- Achten Sie darauf, dass der Finalisierungsprozess nicht nur so schnell wie möglich erledigt ist, sondern wirklich kaum Zeit beansprucht
- Stellen Sie sicher, dass Sie *auf keinen Fall* neue Instanzen von irgendwelchen Objekten innerhalb des Finalisierungsprozesses erstellen

Es gibt übrigens Objekte in .NET Framework, deren Vorhandensein die CLR auch noch zum Finalisierungszeitpunkt garantiert.

Da Sie Ausgaben bei der Finalisierung wahrscheinlich nur zu Testzwecken machen werden, verwenden Sie dafür besser die `Debug`-Klasse. Diese Klasse stellt ebenfalls eine `Write`- bzw. `WriteLine`-Methode zur Verfügung, hat aber gegenüber `Console` entscheidende Vorteile: Das Vorhandensein der Klasse zum Finalisierungszeitpunkt ist garantiert, und Ausgaben erfolgen darüber hinaus nur in einen so genannten Trace-Listener.[7]

[7] Ein speicherresidentes Programm, das spezielle Debugausgaben »abhört« und in eigenen Fenstern ausgibt oder sonst wie protokolliert. Falls Sie Programme in der Entwicklungsumgebung von Visual Studio laufen lassen, dann ist das Ausgabefenster der vorinstallierte Trace-Listener. Alle Ausgaben, die Sie mit `Debug.Write(Line)` durchführen, gelangen dann ins Ausgabefenster.

Soweit zur Finalisierung von Objekten durch .NET Framework. `Finalize` ist allerdings eine Methode, die ausschließlich durch .NET Framework aufgerufen werden darf. Was ist aber, wenn Sie Objekte erstellen wollen, die der Benutzer selbst – quasi – schließen oder freigeben will? .NET Framework bietet dazu ein Schnittstellenmuster über die so genannte `IDisposable`- Schnittstelle an. Der nächste Abschnitt verrät mehr über dieses Thema.

Dispose

Mit der `IDisposable`-Schnittstelle stellt .NET Framework eine Implementierungsvorschrift bereit, mit deren Hilfe Sie eine Methode implementieren können, die anders als `Finalize` auch aus Ihrem Code heraus aufgerufen werden darf, um das Objekt zu entsorgen. Wenn Sie die Schnittstelle per `Implements` in Ihrer Klasse implementieren, müssen Sie die `Dispose`-Methode einfügen, die dann für das notwendige Aufräumen die Verantwortung trägt. Damit karikiert man natürlich die Funktion eines echten Destruktors in anderen OOP- Sprachen, der eben automatisch (man braucht sich halt nicht darum zu kümmern) aufgerufen wird. Der Aufruf von `Dispose` ist daher ein Kompromiss, den man für die vielen Vorteile des Garbage Collectors eingeht.

HINWEIS Verwenden Sie `Using` als Strukturblock, um ein implizites `Dispose` für das verwendete Objekt zu erreichen, wenn das Programm den Gültigkeitsbereich des Strukturblocks verlässt.

Diese Aufräumarbeiten sind prinzipiell die gleichen Arbeiten, die auch eine `Finalize`-Methode durchführt. Doch `Dispose` hat ein wenig mehr Arbeit. Denn wenn Sie Aufräumarbeiten mit `Dispose` vorgenommen haben, dann müssen Sie dafür sorgen, dass der GC die Aufräumarbeiten innerhalb Ihres Objekts nicht noch einmal durch den Aufruf dieser Methode erledigt. Zu diesem Zweck gibt es die `SuppressFinalize`-Methode des Garbage Collectors. Rufen Sie diese Methode auf und übergeben Sie ihr Ihre Klasse als Argument, schließt der Garbage Collector sie für alle folgenden GC-Durchläufe von Aufräumarbeiten dieser Art aus.

Nun besteht die eigentliche Aufgabe der `Dispose`-Methode darin, zu unterscheiden, ob ein Aufruf durch das eigene Programm eben über `Dispose` (üblich ist bei bestimmten Klassen auch `Close`, das nichts anderes macht als `Dispose`, nur dass es einen anderen Namen trägt[8]) oder durch den Garbage Collector über `Finalize` erfolgt ist. Ihr `Dispose` muss ebenfalls dafür sorgen, dass erkannt wird, ob ein Objekt aus Ihrer Sicht schon entsorgt wurde, und im Bedarfsfall eine Ausnahme auslösen.

Ein Beispiel für die Implementierung einer vollständigen `Finalize`/`Dispose`-Lösung finden Sie in der folgenden Beispielanwendung:

[8] Aber wie heißt es so schön: Ausnahmen bestätigen die Regel. Die `System.Windows.Forms`-Klasse gehört hierzu: Wenn Sie ein Formular mit `Close` schließen, haben Sie die Möglichkeit, das Schließen des Formulars im `FormClosing`-Ereignis zu verhindern (indem Sie `e.Cancel` auf `True` setzen). »Schließen« Sie das Formular jedoch mit `Dispose`, was auch geht, dann »schießen« Sie es sozusagen ab. Es ist sofort zu, weg, verschwunden, entsorgt, und es löst auch keine Ereignisse mehr aus. Bei anderen Objekten wie denen, die Dateien lesen oder schreiben, wird ebenfalls `Close` benutzt. Entweder um aus traditionellen Gründen eine Datei eben zu »schließen« – oder weil die Programmierer nicht zu den Meetings erschienen sind, in denen man sich auf `Dispose` geeinigt hatte.

Implementierung eines hochauflösenden Timers als IDisposable

Computer, mit denen Sie in der heutigen Zeit programmieren, sind alles andere als langsam. Der Computer zum Beispiel, auf dem ich gerade diese Zeilen schreibe, ausgestattet mit einem Intel I5 mit 2,67 MHz Taktfrequenz, vermag in einer Sekunde in etwa 50.000 so genannter MIPS durchführen (*Million Instructions per Second*), und das ergibt 50 Milliarden Anweisungen (Prozessoranweisungen natürlich, keine Basic-Befehle) pro Sekunde. Ein kleines Programm, das obendrein nur einen der vier Prozessorkerne eines Intel I5-Prozessors nutzt, um von 0 bis 4 Milliarden zu zählen, benötigt deswegen dazu auch nur rund 11 Sekunden. Das sind rund 360 Millionen Zählungen pro Sekunde oder 360.000 Zählungen pro Millisekunde:

```
Dim sw = Stopwatch.StartNew
For c As UInteger = 0 To UInteger.MaxValue - 1
Next
sw.Stop()
MessageBox.Show("Dauer in ms: " & sw.ElapsedMilliseconds.ToString("#,##0"))
```

Und trotz dieser enormen Rechenleistung erlaubt es uns .NET mit Boardmitteln nicht, auch nur jede Hundertstelsekunde irgendeinen Zustand abzufragen oder eine bestimmte Aufgabe auszuführen.

Überlegen wir mal: Eine Hundertstelsekunde, das ist eine Ewigkeit für unseren Prozessor: In 10 ms kann der nämlich rund 500.000.000 Anweisungen verarbeiten – unsere Zählschleife hätte in der Zeit mal eben von 0 bis 3,6 Millionen gezählt. Leistung wäre also genug da, und trotzdem gibt es bei der gestellten Aufgabe ein Problem: Wenn Sie die Zustandsaktualisierungen oder Zustandsabfragen in einer Endlosschleife ausführen, dann blockieren Sie natürlich die Leistung eines kompletten Prozessorkerns. Ein Programm mit der Aufgabe, auf einem Formular (WinForms) oder einem Fenster (WPF) eine Stoppuhr mit Hundertstelsekundenanzeige zu realisieren, sähe wie in Abbildung 14.2 gezeigt aus.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 14\StopWatch
```

Abbildung 14.2 Die einfachste Version unserer Stoppuhr schießt mit Kanonen auf Spatzen: Durch die Endlosschleife beansprucht das Programm die Leistung eines kompletten Prozessorkerns – effizient ist was anderes

Wenn Sie das Programm starten, sehen Sie das Problem mit einem Blick auf den Task-Manager von Windows oder – wie in der Abbildung zu sehen – eine der netten verfügbaren Prozessorauslastungs-Minianwendungen, die Sie sich herunterladen laden können: Das Programm verbraucht nicht weniger als 100% der Leistung eines kompletten Prozessorkerns, da es die Anzeige in einer Endlosschleife aktualisiert und nur zwischendurch `DoEvents` aufruft, um der Anwendung Zeit zum Luftholen zu geben und ihr es damit ermöglicht, Steuerelemente zu aktualisieren und Abfragen zu verarbeiten. Wenn man auf diese Weise Software entwickelt, kann man damit natürlich keinen Blumentopf gewinnen:

```
Public Class Form1

    Private myStopSignaled As Boolean

    Private Sub btnStartStopButton_Click(ByVal sender As System.Object,
                     ByVal e As System.EventArgs) Handles btnStartStopButton.Click

        If btnStartStopButton.Text = "Start" Then
            btnStartStopButton.Text = "Stop"
            Dim sw = Stopwatch.StartNew

            Dim lastMs As Long

            Do
                'Warten auf die nächste Millisekunde
                Do
                    If lastMs <> sw.ElapsedMilliseconds Then
                        lastMs = sw.ElapsedMilliseconds
                        elapsedMillisecondsLabel.Text =
                            TimeSpan.FromMilliseconds(lastMs).ToString("hh\:mm\:ss\:fff")
                        My.Application.DoEvents()
                        If myStopSignaled Then
                            Return
                        End If
                    End If
                Loop
            Loop
        Else
            myStopSignaled = True
            btnStartStopButton.Text = "Stop"
        End If
    End Sub
End Class
```

»Kein Problem«, wird man sich wohl zunächst denken, »dann realisieren wir das eben mit einem Timer, den wir auf einen 1-ms-Zyklus einstellen!«. Das sollte dann so gut wie keine Prozessorleistung mehr kosten. O.k., dann schauen Sie sich mal an, was passiert, wenn Sie dem Beispiel eine weitere Schaltfläche mit folgendem Code hinzufügen:

```
    Private myTimer As Timer

    Private Sub TimerTestButton_Click(ByVal sender As System.Object,
                                      ByVal e As System.EventArgs) Handles TimerTestButton.Click
        myTimer = New Timer
        myTimer.Interval = 1
        myTimer.Start()

        Dim sw = Stopwatch.StartNew
        Dim lastms = sw.ElapsedMilliseconds
```

```
        AddHandler myTimer.Tick, Sub(timerSender As Object, timer_e As EventArgs)
                                     Dim cms = sw.ElapsedMilliseconds
                                     Debug.Print("Vergangene Zeit: " & cms - lastms & " ms.")
                                     lastms = cms
                                 End Sub
    End Sub
```

Wenn Sie anschließend das Programm starten und auf die neue Schaltfläche klicken, dann sehen Sie in etwa die folgende Auflistung im Ausgabefenster von Visual Studio (das Sie im Bedarfsfall mit Strg+Alt+O anzeigen lassen können):

```
Vergangene Zeit: 16 ms.
Vergangene Zeit: 20 ms.
Vergangene Zeit: 13 ms.
Vergangene Zeit: 16 ms.
Vergangene Zeit: 20 ms.
Vergangene Zeit: 12 ms.
Vergangene Zeit: 21 ms.
Vergangene Zeit: 12 ms.
Vergangene Zeit: 14 ms.
Vergangene Zeit: 20 ms.
Vergangene Zeit: 12 ms.
Vergangene Zeit: 14 ms.
Vergangene Zeit: 20 ms.
Vergangene Zeit: 20 ms.
```

Zwar wird jetzt kaum noch Prozessorleistung benötigt (davon können Sie sich über den Task-Manager überzeugen), aber das Ziel ist verfehlt. Denn der Aufruf zur Aktualisierung der Stoppuhr erfolgt nun erheblich seltener als jede Hundertstelsekunde (10 ms), das Intervall liegt deutlich darüber. Sie beobachten damit gerade ein fundamentales Problem eines Nicht-Echtzeit-Betriebssystems, wie Windows eines ist: Sie können den Timer im Beispiel einstellen, wie Sie wollen: Unter 15-20 Millisekunden ist unter Windows 7 nichts zu machen, auf älteren Systemen können schon 25-30 Millisekunden zum Problem werden. Und Ausreißer bis zu 40 ms kommen auf allen Systemen vor. Das liegt nicht zuletzt daran, dass der Scheduler von Windows, der dafür sorgt, dass jeder gleichzeitig laufende Thread im System auch mal zum Zug kommt, ebenfalls keine kleinere Auflösung hat.

Immerhin ist ein Kontextwechsel für einen Thread eine aufwändige Sache – ein Thread unter Windows beansprucht nicht weniger als ein MB, und schon im Normalzustand von Windows, also wenn es gerade gestartet wurde, können mehrere Hundert Threads gestartet werden; wenn sie auch nicht alle aktiv sind (Abbildung 14.3).

Abbildung 14.3 Mit Outlook 2010, Word 2010 und Visual Studio 2010 hat selbst ein modernes Dual Core-Hyperthreading-Netbook schon ausreichend zu tun – nicht weniger als rund 800 Threads werkeln dabei mehr oder weniger aktiv vor sich hin!

Deswegen schaltet der Scheduler von Windows zwischen den Threads auch nicht so oft um – so alle 15-20 ms teilen sich die jeweils nächsten von Hunderten gleichzeitig laufender Threads die verfügbaren Prozessorkerne, und jeder, der will, kommt so mal dran. Aber eben auch nicht öfter. Damit kommen weder die normalen, in .NET zur Verfügung stehenden Timer noch sonst ein anderer Trick in Frage, ohne eine Auflösung von unter 10 ms hinzubekommen, bei der nicht die Rechenleistung eines kompletten Kerns verplempert wird.

Eine Lösung bleibt jedoch noch: Windows bietet schon seit Windows XP einen so genannten Multimediatimer an, der in der Regel zum Synchronisieren von Wiedergabemedien (Audio, Video) verwendet wird. Auch hier wird eine höhere Auflösung benötigt, als sie die normalen Timer zur Verfügung stellen könnten; der Multimediatimer hat deshalb auch eine Auflösung von 1 ms. Da es .NET erlaubt, auch Betriebssystemfunktionen durch so genannte Plattform Invoke-Aufrufe zu nutzen (*P/Invoke*, sprich »Pieh Inwouk«), lässt sich mit relativ wenig Aufwand eine Klasse entwickeln, die einen solchen Timer auch in .NET zur Verfügung stellt:

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 14\HighSpeedTimer01
```

Die Klasse, die den Multimediatimer zur Verfügung stellt, sieht folgendermaßen aus:

```
Imports System.ComponentModel
Imports System.Runtime.InteropServices

Public Class HighSpeedTimer
```

```
Private myTimerID As Integer                ' ID des Multimedia-Timers
Private myUser As Integer                   ' Benuzerdefinierter Parameter, nicht verwendet.
Private myTimerCapabilities As TimerCaps    ' Hält die TimeCaps, die die Timerauflösung beschreiben
Private myMode As Integer = 1               ' 0=Einmalig, 1=Periodisch. 1 wird vordefiniert.

Private myPeriodInMs As Integer             ' Wie oft soll getriggert werden?
Private myresolutionInMs As Integer         ' Welche Auflösung kann der Timer?
Private myHasStarted As Boolean             ' Läuft der Timer?

'Delegate, der für den Rückruf aus dem Betriebssystem benötigt wird,
'wenn der Timer abgelaufen ist...
Private Delegate Sub TimeProc(ByVal id As Integer, ByVal msg As Integer,
                              ByVal user As Integer, ByVal reserved1 As Integer,
                              ByVal reserved2 As Integer)

'...und in dieser Delegatenvariable definiert wird.
Private myCallBackTimeProc As TimeProc

'Das Ereignis, was durch die Rückrufroutine ausgelöst wird,
'wenn der Timer abgelaufen ist.
Public Event Elapsed(ByVal sender As Object, ByVal e As EventArgs)
```

Die folgenden Zeilen zeigen, wie Sie von Visual Basic aus Methoden des Betriebssystems direkt aufrufen können. Die Methoden für die Steuerung des Multimediatimers befinden sich in der Library *winmm.dll* (Abkürzung für *Windows Multimedia*). Das `DllImport`-Attribut, das sich oberhalb der Methode befindet, gibt an, dass sich der Code der Methode nicht im Methodenrumpf befindet, sondern dass es sich um einen externen Aufruf handelt.

TIPP Die Website *http://www.pinvoke.net* ist eine sehr nützliche Quelle, um herauszufinden, welche Methoden durch das Betriebssystem zur Verfügung gestellt werden, und wie Sie am besten vorgehen, um eine entsprechende Methode aufzurufen. Bevor Sie aber anfangen, nun eigene große Aufrufbibliotheken mit Betriebssystemfunktionen zu erstellen, versuchen Sie bitte immer erst, ein .NET-Äquivalent zu finden. Nur in wenigen Fällen, wie im Beispiel, ist es wirklich notwendig, Routinen des Betriebssystems per P/Invoke-Technik aufzurufen.

```
'Ermittelt die Fähigkeiten des Timers auf diesem Rechner.
<DllImport("winmm.dll")>
Private Shared Function timeGetDevCaps(ByRef caps As TimerCaps,
                                       ByVal sizeOfTimerCaps As Integer) As Integer
End Function

'Erstellt und startet den Multimedia-Timer.
<DllImport("winmm.dll")>
Private Shared Function timeSetEvent(ByVal delay As Integer, ByVal resolution As Integer,
                                     ByVal CallBackProc As TimeProc, ByVal user As Integer,
                                     ByVal mode As Integer) As Integer
End Function

'Löscht einen Timer, der gerade in Betrieb ist.
<DllImport("winmm.dll")>
Private Shared Function timeKillEvent(ByVal id As Integer) As Integer
End Function

''' <summary>
''' Erstellt eine Instanz dieser Klasse und setzt Wiederholungsfrequenz
```

```
    ''' des Timer-Triggers und Auflösung in Millisekunden.
    ''' </summary>
    ''' <param name="periodInMs">Wiederholungsfrequenz - wie oft soll der Timer auslösen.</param>
    ''' <param name="resolutionInMs">Auflösung des Timers (der Wert 1ms liefert
    ''' die genauesten Ergebnisse).</param>
    ''' <remarks></remarks>
    Sub New(ByVal periodInMs As Integer, ByVal resolutionInMs As Integer)
        'Ermittelt die Fähigkeiten des Multimedia Timers, die dann über
        'TimerCapabilities aubrufbar sind.
        Dim ret = timeGetDevCaps(myTimerCapabilities, Marshal.SizeOf(myTimerCapabilities))

        myPeriodInMs = periodInMs
        myresolutionInMs = resolutionInMs
        myCallBackTimeProc = AddressOf CallBackTimeProc
    End Sub

    ''' <summary>
    ''' Startet den Timer.
    ''' </summary>
    ''' <remarks></remarks>
    Public Sub Start()

        If myHasStarted Then
            Return
        End If

        Dim ret = timeSetEvent(myPeriodInMs, myresolutionInMs,
                               myCallBackTimeProc, myUser, myMode)
        Dim hr = Marshal.GetHRForLastWin32Error
        If ret = 0 Then
            Throw New Win32Exception("Der Timer konnte nicht gestartet werden!")
        Else
            myTimerID = ret
            myHasStarted = True
        End If

    End Sub
```

Die folgende Methode ist nun die Methode, die uns Kopfzerbrechen machen sollte. Solange ein Programm, das die Multimediatimer-Klasse verwendet, immer dafür sorgt, dass der Timer gestoppt wird, kann nichts passieren. Aber was geschieht, wenn das Programm auf unvorhersehbare Ereignisse stößt, die dazu führen, dass der Timer nicht freigegeben werden kann? Unter Umständen läuft der Timer dann noch zu einem Zeitpunkt weiter, an dem Sie als Entwickler längst nicht mehr damit rechnen, oder schlimmer: Gegebenenfalls kann das Programm sogar beendet werden, ohne dass es die Möglichkeit hatte, den angeforderten Timer anzuhalten und wieder freizugeben! Um das zu verhindern, dient die Implementierung von `IDisposable`, und der nächste Abschnitt zeigt, wie das funktioniert.

```
    ''' <summary>
    ''' Stoppt den Timer.
    ''' </summary>
    ''' <remarks></remarks>
    Public Sub [Stop]()
        If Not myHasStarted Then
            Return
        End If

        timeKillEvent(myTimerID)
    End Sub
```

```
    'Die Rückrufroutine, die vom Betriebssystem aufgerufen wird,
    'wenn der Timer abgelaufen ist.
    Private Sub CallBackTimeProc(ByVal id As Integer, ByVal msg As Integer,
                                 ByVal user As Integer, ByVal param1 As Integer,
                                 ByVal param2 As Integer)
        RaiseEvent Elapsed(Me, EventArgs.Empty)
    End Sub

    ''' <summary>
    ''' Ermittelt die Möglichkeiten des Systems bezüglich des Timers.
    ''' </summary>
    ''' <value></value>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public ReadOnly Property TimerCapabilities As TimerCaps
        Get
            Return myTimerCapabilities
        End Get
    End Property

    ''' <summary>
    ''' Ermittelt, ob der Timer gestartet wurde.
    ''' </summary>
    ''' <value></value>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public ReadOnly Property HasStarted As Boolean
        Get
            Return myHasStarted
        End Get
    End Property

End Class

'Wird benötigt für die Parameterübergabe an die Betriebssystemroutinen
<StructLayout(LayoutKind.Sequential)>
Public Structure TimerCaps
    Public PeriodMin As Integer
    Public periodMax As Integer
End Structure
```

Unterstützung durch den Visual Basic-Editor beim Einfügen eines Disposable-Patterns

Das Muster, auf das Sie beim Implementieren einer »disposebaren« Klasse achten müssen, ist nicht nur etwas komplexer als bei der Implementierung anderer Funktionen, die Sie durch die Einbindung einer oder mehrerer Schnittstellen implementieren müssen.

Sie müssen auch darauf achten, dass ein internes Muster strikt einzuhalten ist – und das geht natürlich weit über das bloße Vorhandensein einer entsprechenden `Dispose`-Methode hinaus.

Aus diesem Grund gibt es eine besondere Editorunterstützung für die `IDisposable`-Schnittstelle. Sobald Sie diese am Klassenkopf einfügen und nach `Implements IDisposable` ↵ betätigen, fügt der Editor nicht nur die Funktionsrümpfe, die das Interface vorschreibt, sondern ein etwas spezifischeres Funktionsgerüst als Code in Ihre Klasse ein.

```
Public Class HighSpeedTimer
    Implements IDisp
                IDisposable                    Interface IDisposable
                                               Definiert eine Methode zur Freigabe von reservierten Ressourcen.
    Private myT  Allgemein  Alle
    Private myU                                                                  et.
    Private myTimerCapabilities As TimerCaps     ' Hält die TimeCaps, die die Timerauflösung beschreiben
    Private myMode As Integer = 1                ' 0=Einmalig, 1=Periodisch. 1 wird vordefiniert.
```

Abbildung 14.4 Das Implementieren von `IDisposable` ist einfach wie bei jeder Schnittstelle: An dieser Stelle drücken Sie [⇥] und [↵], um die Methodenrümpfe der Schnittstelle einzufügen, ...

Sie können das direkt am besprochenen Highspeed-Timer-Beispiel ausprobieren. Nachdem Sie die oben beschriebenen Schritte durchgeführt haben, befindet sich fast die komplett fertige `IDisposable`-Implementierung bereits am Ende der Klasse `HighSpeedTimer`, wie in Abbildung 14.5 zu sehen.

```
HighSpeedTimer.vb*   Quellcodeverwaltu   Hebt die Auskommentierung der ausgewählten Textzeilen auf. (Strg+K, Strg+U)
HighSpeedTimer                                                    (Deklarationen)

#Region "IDisposable Support"
    Private disposedValue As Boolean ' So ermitteln Sie überflüssige Aufrufe

    ' IDisposable
    Protected Overridable Sub Dispose(ByVal disposing As Boolean)
        If Not Me.disposedValue Then
            If disposing Then
                ' TODO: Verwalteten Zustand löschen (verwaltete Objekte).
            End If

            ' TODO: Nicht verwaltete Ressourcen (nicht verwaltete Objekte) freigeben und Finalize() unten überschreiben.
            ' TODO: Große Felder auf NULL festlegen.
        End If
        Me.disposedValue = True
    End Sub

    ' TODO: Finalize() nur überschreiben, wenn Dispose(ByVal disposing As Boolean)
    ' oben über Code zum Freigeben von nicht verwalteten Ressourcen verfügt.
    'Protected Overrides Sub Finalize()
    '    ' Ändern Sie diesen Code nicht. Fügen Sie oben in Dispose(ByVal disposing As Boolean) Bereinigungscode ein.
    '    Dispose(False)
    '    MyBase.Finalize()
    'End Sub

    ' Dieser Code wird von Visual Basic hinzugefügt, um das Dispose-Muster richtig zu implementieren.
    Public Sub Dispose() Implements IDisposable.Dispose
        ' Ändern Sie diesen Code nicht. Fügen Sie oben in Dispose(ByVal disposing As Boolean) Bereinigungscode ein.
        Dispose(True)
        GC.SuppressFinalize(Me)
    End Sub
#End Region
100 %
```

Abbildung 14.5 ... nur mit dem Unterschied zu normalen Schnittstellen, dass hier der komplette `Dispose`- und `Finalize`-Code mit eingefügt wird, den Sie nur noch mit Ihren eigenen Modifizierungen ausstatten müssen!

Die Ergänzungen, die wir an dieser Stelle noch vornehmen müssen, sind schnell gemacht. Zunächst folgen wir den Vorschlägen in den Kommentaren oberhalb der Methode `Finalize`. Da es sich bei unserem Timer tatsächlich um eine nicht verwaltete (*unmanaged*) Ressource handelt, nehmen wir die Auskommentierung der betroffenen Zeilen, wie in der Grafik zu sehen, zurück. Damit stellen wir sicher, dass die `Dispose`-Methode, die wir gerade implementieren, auch über die `Finalize`-Methode über den Garbage Collector im Bedarfsfall aufgerufen wird.

Und schließlich brauchen wir lediglich dafür zu sorgen, an entscheidender Stelle den Timer, falls er noch aktiv ist, wieder freizugeben, und dieser Code ist schnell implementiert (die Änderungen finden Sie in fetter Schrift im folgenden Listing):

```
#Region "IDisposable Support"
    Private disposedValue As Boolean ' So ermitteln Sie überflüssige Aufrufe

    ' IDisposable
    Protected Overridable Sub Dispose(ByVal disposing As Boolean)
        If Not Me.disposedValue Then
            'Wir haben keine verwalteten Objekte, die wir berücksichtigen müssten,
            'deswegen brauchen wir diesen If-Zweig nicht:
            'If disposing Then
            '    ' TODO: Verwalteten Zustand löschen (verwaltete Objekte).
            'End If

            'Hier muss der Timer freigegeben werden, falls er noch aktiv ist.
            If myHasStarted Then
                Me.Stop()
            End If

        End If
        Me.disposedValue = True
    End Sub

    ' TODO: Finalize() nur überschreiben, wenn Dispose(ByVal disposing As Boolean)
    ' oben über Code zum Freigeben von nicht verwalteten Ressourcen verfügt.
    Protected Overrides Sub Finalize()
        ' Ändern Sie diesen Code nicht. Fügen Sie oben in
        ' Dispose(ByVal disposing As Boolean) Bereinigungscode ein.
        Dispose(False)
        MyBase.Finalize()
    End Sub

    ' Dieser Code wird von Visual Basic hinzugefügt, um das Dispose-Muster richtig zu implementieren.
    Public Sub Dispose() Implements IDisposable.Dispose
        ' Ändern Sie diesen Code nicht. Fügen Sie oben in
        ' Dispose(ByVal disposing As Boolean) Bereinigungscode ein.
        Dispose(True)
        GC.SuppressFinalize(Me)
    End Sub
```

BEGLEITDATEIEN Die Begleitdateien des auf diese Weise erweiterten Beispiels finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 14\HighSpeedTimer02- IDisposable
```

Was an dieser Stelle noch wichtig zu erwähnen ist, betrifft die letzte Zeile in der `Dispose`-Methode. Der Entwickler, der unsere `HighSpeedTimer`-Klasse verwendet, ist nun in der Lage, das Objekt manuell durch den Aufruf von `Dispose` zur Entsorgung freizugeben. Wenn das allerdings passiert, müssen die Aufräumarbeiten nicht mehr vom Garbage Collector ausgeführt werden, ja das Gegenteil ist sogar der Fall: Der Garbage Collector darf sogar in einem solchen Fall den Aufräumcode nicht mehr durchlaufen und muss deswegen darüber informiert werden, dass diese Arbeiten bereits durchgeführt wurden. Das geschieht mit der letzten Zeile des Listings, mit der Methode `GC.SuppressFinalize`.

Gezieltes Freigeben von Objekten mit Using

Objekte, die es gestatten, sich auf die Entsorgung hinweisen zu lassen, implementieren eine Methode namens `Dispose` (etwa: *entsorgen*) – das haben Sie in diesem Kapitel bereits kennen gelernt. Rufen Sie `Dispose` eines Objekts auf, gibt es alle belegten Systemressourcen frei und signalisiert dem Garbage Collector gleichzeitig, dass er es »mitnehmen« kann. Wir wissen, wie wir ein solches Muster in eigenen Klassen implementieren, aber wie wenden wir das idealerweise an?

Eine typische Vorgehensweise zur Anwendung sieht so aus, wie es das folgende Beispiellisting demonstriert, das einen einfachen Text in eine Datei schreibt:

```
Dim locSw As StreamWriter
Try
    locSw = New StreamWriter("C:\Textdatei1.txt")
    Try
        locSw.WriteLine("Erste Textzeile")
        locSw.WriteLine("Zweite Textzeile")
        'Schreibpuffer leeren
        locSw.Flush()
    Catch ex As Exception
        Debug.Print("Fehler beim Schreiben der Datei!")
    Finally
        'Alle Systemressourcen wieder freigeben
        locSw.Dispose()
    End Try
Catch ex As Exception
    Debug.Print("Fehler beim Öffnen der Datei!")
End Try
```

Weil das Freigeben der belegten Systemressourcen in diesem Fall so wichtig ist, befindet sich der Aufruf der `Dispose`-Methode in diesem Fall im `Finally`-Teil des `Try/Catch`-Blocks. Dadurch wird sichergestellt, dass die Systemressourcen des Objekts in jedem Fall freigegeben werden, egal ob es innerhalb des `Try`-Teils zu einer Ausnahme kam oder nicht.

HINWEIS Zwei geschachtelte `Try`-Blöcke sind hier übrigens sinnvoll, weil beim Öffnen der Datei andere Ausnahmen auftreten können als beim Schreiben. Die Systemressourcen selbst werden aber nur nach *erfolgreichem* Öffnen des `StreamWriter`-Objekts von diesem belegt und müssen deswegen auch nur in diesem Fall (innerer `Try/Catch`-Block) mit `Dispose` wieder freigegeben werden.

`Using` erlaubt nun, die Lebensdauer eines Objekts, das die `Dispose`-Methodik implementiert, gezielt zu steuern. Das folgende Beispiellisting entspricht dem obigen Beispiel im Detail – `Using` vereinfacht aber den Umgang mit dem Objekt, und der Code wird leichter lesbar:

```
'Schreiben einer Datei - mit Using wird die Lebensdauer von locSw2 gesteuert
Try
    'Alternativ ginge auch die Weiterverwendung von locSw:
    'locSw = New StreamWriter("C:\Textdatei2.txt")
    'Using locSw
    Using locSw2 As New StreamWriter("C:\Textdatei2.txt")
        Try
            locSw2.WriteLine("Erste Textzeile")
```

```
            locSw2.WriteLine("Zweite Textzeile")
            'Schreibpuffer leeren
            locSw2.Flush()
        Catch ex As Exception
            Debug.Print("Fehler beim Schreiben der Datei!")
        End Try
    'Hier wird automatisch locSw.Dispose durchgeführt
    End Using
Catch ex As Exception
    Debug.Print("Fehler beim Öffnen der Datei!")
End Try
```

Sie sehen hier, dass der `Finally`-Teil des inneren `Try/Catch`-Blocks überflüssig geworden ist – denn am `End Using` kommt das Programm nicht vorbei. Selbst, wenn es versuchen würde, beispielsweise mit `Return` aus dem `Using`-Block herauszuspringen, würde die `Dispose`-Methode des Objekts dennoch vor dem eigentlichen Verlassen der Prozedur noch ausgeführt.

TIPP `Using` wird besonders häufig bei Verbindungen zu Microsoft SQL Server eingesetzt. Sobald Sie eine Verbindung zu Microsoft SQL Server benötigen, verwenden Sie das dazu erforderliche `Connection`-Objekt innerhalb einer `Using`-Anweisung. Führen Sie alle Operationen durch, für die Sie eine offene Verbindung benötigen. Am Ende aller Operationen schließen Sie die SQL Server-Verbindung mit `End Using`. Auf diese Weise können Sie auch gefahrlos einen solchen Codeblock verlassen, ohne explizit dafür Sorge getragen haben zu müssen, die Verbindung zu SQL Server wieder zu schließen. Zusätzlich zum Schließen der Verbindung, was auch die Methode `Close()` leisten könnte, wird in `Dispose()` auch der *Shared Lock*[9] in der benutzten Datenbank entfernt. Ihre Anwendung wird dadurch robuster. Denn Sie vermeiden doppelt oder mehrfach offene Verbindungen, etwa weil Sie schlicht vergessen haben, eine offene Verbindung wieder zu schließen, nachdem Sie einen Codeblock, in dem SQL-Operationen verarbeitet wurden, vorzeitig verlassen haben.

[9] Damit die Maxime ACID beim Microsoft SQL Server (*Atomic*, *Consistent*, *Isolated*, *Durable* – in etwa: *ganzheitlich*, *konsequent*, *isoliert* und *verlässlich*) nicht verletzt wird, dürfen sich natürlich keine Daten ändern, während dieser Daten in einer Verbindung, zum Beispiel über eine `SELECT`-Abfrage, zur Verfügung stellt. Ein *Shared Lock* erlaubt also, dass mehrere Verbindungen gleichzeitig lesend zugreifen, und stellt dabei sicher, dass keine Daten geändert werden können.

Kapitel 15

Eigene Operatoren für benutzerdefinierte Typen

In diesem Kapitel:

Einführung in Operatorenprozeduren 498
Vorbereitung einer Struktur oder Klasse für Operatorenprozeduren 499
Implementierung von Operatoren 503
Implementierung von Vergleichsoperatoren 506
Implementierung von Typkonvertierungsoperatoren mit dem Operator CType 506
Implementieren von Wahr- und Falsch-Auswertungsoperatoren 508
Problembehandlungen bei Operatorenprozeduren 510
Übersicht der implementierbaren Operatoren 511

Es gibt ein vergleichsweise neues Wort, das es nicht einmal für nötig befunden hat, sich wenigstens als Anglizismus zu geben, stattdessen hat es sich ganz unkaschiert als ursprünglicher englischer Begriff in die deutsche Sprache eingeschlichen. Die Rede ist von »Convenience«, zu Deutsch etwa: »Bequemlichkeit«. Einige wirkliche Kenner der Hotelbranche haben wegen des berühmt-berüchtigten Convenience Food inzwischen sogar eine psychosomatische Eierallergie entwickelt, und das in so heftigem Ausmaß, dass sie bei Auswärtsübernachtungen längst auf den morgendlichen gelben Glibber am Buffet verzichten müssen. Und das aus vielleicht gutem Grund, kommt doch die sich in den metallenen Warmhalteschalen befindliche Eierspeise nicht mehr aus verschiedenen Schalen und einer Pfanne, sondern einer luftdicht verschlossenen Tüte mit Trockenrühreipulver (wer hätte gedacht, dass es so was überhaupt gibt?) – jedenfalls in vielen Fällen. Beim Convenience Food geht es also in erster Linie darum, dem *Zubereiter* das Leben so angenehm wie möglich zu machen, und weniger dem Gast, der das Wasser-/Rühreipulvergemansche anschließend verdrückt.

Operatorenprozeduren, die mit Visual Basic 2005 eingeführt wurden, könnte man unter dem Begriff Convenience Tools (»Bequemlichkeitswerkzeuge«)[1] laufen lassen, obschon sie dem Entwickler das Leben zwar erleichtern, sich aber nicht negativ auf den »Consumer« auswirken. Sie stellen nichts bereit, was die Lösung eines bestimmten Entwicklungsproblems an sich in irgendeiner Form vereinfachen würde, sie tragen lediglich dazu bei, dass sich Klassen oder Strukturen später einfacher anwenden lassen und Code besser lesbar wird.

Einführung in Operatorenprozeduren

Um was geht es genau? Operatorenprozeduren dienen dazu, es eigenen Klassen zu gestatten, zusammen mit Operatoren verwendet werden zu können. Wenn Sie einen eigenen Typ geschaffen haben, ganz egal ob auf Basis einer Klasse oder einer Struktur, dann stellt dieser bestimmte Funktionalitäten über Methoden bereit. Und mithilfe von Operatorenprozeduren können Sie diese Methoden an Operatoren binden.

Ein Beispiel dafür könnte eine »Superstring«-Klasse sein. Eine Klasse, die zwar wie der primitive Datentyp `String` funktioniert, aber den Umgang mit verschiedenen Funktionen stark vereinfacht. Einen »Rechen«-Operator können Sie bei Strings heute schon verwenden – das Pluszeichen, um zwei Strings aneinanderzuhängen – damit ist aber natürlich kein Additions-, sondern der in fast allen Programmiersprachen vorhandene Stringverkettungsoperator gemeint. Wenn Sie also folgende Codezeilen haben:

```
'Deklaration und Definition eines normalen Strings
Dim locNormaloString As String
locNormaloString = "Wenn man seinen Kopf gegen eine "
locNormaloString = locNormaloString + "Wand schlägt, verbraucht man 150 Kalorien."
Console.WriteLine("Ausgangszeichenfolge:")
Console.WriteLine(locNormaloString)
```

dann hat die Ausführung dieses Codes das nachstehende Ergebnis zur Folge:

```
Ausgangszeichenfolge:
Wenn man seinen Kopf gegen eine Wand schlägt, verbraucht man 150 Kalorien.
```

[1] Und es gibt tatsächlich den Begriff der Convenience-Patterns in der IT, gerade beim Überladen von Methoden.

Diese Idee kann man weiterspinnen. So wäre doch beispielsweise auch eine Multiplikation von Strings möglich. Aus

```
"Klaus" * 5
```

würde die Zeichenfolge

```
KlausKlausKlausKlausKlaus
```

entstehen. Und aus

```
"Klaus Löffelmanns Internetauftritt finden Sie unter http://loeffelmann.de" - "Löffelmanns"
```

würde

```
"Klaus Internetauftritt finden Sie unter http://loeffelmann.de"
```

Die »Division« eines Strings mit einem Trennzeichen könnte ein `String`-Array mit den verschiedenen Teilzeichenketten entstehen lassen. So würde der Ausdruck

```
"Verschiede|Worte|sind|so|getrennt" : "|"c
```

ein `String`-Array mit den Elementen

```
Verschiedene
Worte
sind
so
getrennt
```

ergeben. Und zu guter Letzt müssten auch implizite (direkte Zuweisungen) bzw. explizite (mit `CType`) Typumwandlungen in andere Typen möglich sein, sodass auch folgender Code möglich würde:

```
Dim locSuperString as SuperString = "Das hier ist eine String- und keine SuperString-Konstante"
```

bzw.

```
Dim locSuperString as SuperString = CType("das Ganze mit expliziter Zuweisung", SuperString)
```

Vorbereitung einer Struktur oder Klasse für Operatorenprozeduren

Das Wichtigste, das Sie über Operatorenprozeduren wissen müssen: Sie werden ausschließlich als statische Funktionen implementiert. Das liegt einfach daran, dass bei vielen Operatoren zwei Argumente an eine Operatorenprozedur übergeben werden müssen. Die Codezeile

```
TypInstanz3 = TypInstanz1 + TypInstanz2
```

könnte man auch ohne Operatoren ermöglichen. Dann würde die Zeile etwa

```
TypInstanz3 = TypInstanz.Add(TypInstanz1, TypInstanz2)
```

lauten. Daher müssen Sie sich bei den folgenden Ausführungen immer vor Augen halten, dass es bei der Implementierung von Operatorenprozeduren nur um *eine* (aber eben vielleicht eine besonders praktische) Art und Weise geht, eine bestimmte Aufgabe zu erfüllen. So kann man:

```
Dim testString = "Klaus"
testString = testString + " zeigt, was man mit Operatoren machen kann."
```

auch genauso schreiben

```
Dim testString="Klaus"
testString = String.Concat(testString, " zeigt, was man mit Operatoren machen kann.")
```

Und müsste ergänzen:

```
testString=String.Concat(testString, " Und was auch ohne geht.")
```

Gehen wir davon aus, dass die Klasse oder Struktur, aus der sich `TypInstanz1`, `TypInstanz2` und `TypInstanz3` ableiten `TypInstanz` lautet, wird klar, dass nur eine statische Implementierung Sinn ergibt. Denn Sie brauchen keine Instanz dieser Klasse oder Struktur, um zwei unabhängige Instanzen der Klasse bzw. Struktur zu addieren – lediglich den Code, der die Addition vornimmt und ein Funktionsergebnis vom Typ `TypInstanz` zurückliefert.

Das verhält sich ähnlich wie beispielsweise beim Parsen einer Zeichenfolge zur Umwandlung in eine numerische Variable. Auch hier verwenden Sie eine statische Funktion, nämlich `Parse`, für die Sie keine Strukturinstanz benötigen. Sie können `Parse` direkt mit der Zeile

```
Dim EinDouble as Double = Double.Parse("123,45")
```

verwenden. Sie müssen aber keine Variable vom Typ `Double` definieren, um auf die `Parse`-Funktion zuzugreifen. Da ein zusätzliches Set an statischen Funktionen notwendig wird, ergibt es Sinn, sich zunächst nur um die reine Funktionalitätsimplementierung in der entsprechenden Klasse oder Struktur zu kümmern – und die brauchen fürs Erste natürlich nicht statisch zu sein. Eine Klasse `SuperString` könnte also mit komplett implementierter Funktionalität folgendermaßen ausschauen:

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 15\Operatorenüberladung
```

```
Public Structure SuperString

    Private myValue As String

    Public Sub New(ByVal Value As String)
        myValue = Value
    End Sub
```

```
    Public Overrides Function ToString() As String
        Return myValue
    End Function

    Public Function Addieren(ByVal andererString As SuperString) As SuperString
        Return New SuperString(myValue & andererString.ToString)
    End Function

    Public Function Subtrahieren(ByVal andererString As SuperString) As SuperString
        Return New SuperString(myValue.Replace(andererString.ToString, ""))
    End Function

    Public Function Subtrahieren(ByVal letztenZeichen As Integer) As SuperString
        Try
            Return New SuperString(myValue.Substring(0, myValue.Length - (letztenZeichen + 1)))
        Catch ex As Exception
            Return New SuperString(myValue)
        End Try
    End Function

    Public Function Vervielfachen(ByVal anzahl As Integer) As SuperString

        'Wir sind ordentlich und vermeiden Fehler! ;-)
        If myValue Is Nothing Then
            Return Nothing
        End If

        If myValue = "" Or anzahl < 1 Then
            Return New SuperString("")
        End If

        'Mit dem StringBuilder geht das am schnellsten!
        Dim locSB As New System.Text.StringBuilder

        'Einfach den Ausgangsstring sooft anhängen,
        'wie es 'anzahl' vorgibt.
        For c As Integer = 1 To anzahl
            locSB.Append(myValue)
        Next

        'und zurück damit!
        Return New SuperString(locSB.ToString)
    End Function

    Public Function Teilen(ByVal trennzeichen As Char) As SuperString()
        Dim locStringArray As String()
        locStringArray = myValue.Split(New Char() {trennzeichen})

        Dim locSuperStringArray(locStringArray.Length - 1) As SuperString
        For z As Integer = 0 To locStringArray.Length - 1
            locSuperStringArray(z) = New SuperString(locStringArray(z))
        Next
        Return locSuperStringArray

    End Function
End Structure
```

Sie sehen, dass sich die Funktionen, die wir zur späteren Realisierung der Operatoren zunächst als normale Methoden implementieren, vergleichsweise einfach erstellen lassen – dieser Code bedarf nicht wirklich zusätzlicher Erklärungen. Mit herkömmlicher Programmierung, also ohne Operatoren, sieht die Verwendung dieser Klasse dann so aus, wie es die `Sub Main` des Moduls des Projekts demonstriert:

```
Module Main

    Sub Main()
        'Deklaration und Definition eines normalen Strings
        Dim locNormaloString As String
        locNormaloString = "Wenn man seinen Kopf gegen eine "
        locNormaloString = locNormaloString + "Wand schlägt, verbraucht man 150 Kalorien."
        Console.WriteLine("Ausgangszeichenfolge:")
        Console.WriteLine(locNormaloString)

        'Deklaration und Definition eines Super-Strings
        Dim locSuperString As New SuperString(locNormaloString)


        'SuperString-Funktionsdemo: Addieren (Anhängen) anderer Strings
        Console.WriteLine()
        Console.WriteLine("'Addieren' von Strings - 'Toll, was?' anhängen:")
        locSuperString = locSuperString.Addieren( _
                New SuperString(vbNewLine + " - Toll, was?"))
        Console.WriteLine(locSuperString.ToString)

        'Subtrahieren (rauslöschen) anderer Strings
        Console.WriteLine()
        Console.WriteLine("'Subtrahieren' von Strings - ', was?' abziehen:")
        locSuperString = locSuperString.Subtrahieren( _
                New SuperString(", was"))
        Console.WriteLine(locSuperString.ToString)
        Console.WriteLine()
        'Subtrahieren ist überladen - geht auch mit der Anzahl
        'der letzten Zeichen, die entfernt werden sollen.
        Console.WriteLine("'Subtrahieren' von Strings - die letzten 9 Zeichen abziehen:")
        locSuperString = locSuperString.Subtrahieren(9)
        Console.WriteLine(locSuperString.ToString)
        Console.WriteLine()

        'Vervielfachen von Strings
        Console.WriteLine("'Vervielfachen' von Strings:")
        locSuperString = locSuperString.Vervielfachen(4)
        Console.WriteLine(locSuperString.ToString)
        Console.WriteLine()

        '(Auf)teilen von Strings - schon etwas anspruchsvoller
        locSuperString = New SuperString("Der Glückskeks wurde 1916 von George Jung, " & _
                        "einem amerikanischen Nudelmacher erfunden.")
        Console.WriteLine("'(Auf)teilen' von Strings:")
        Console.WriteLine(locSuperString.ToString)
        Dim locSuperStrings() As SuperString
        locSuperStrings = locSuperString.Teilen(" "c)
        For Each locSString As SuperString In locSuperStrings
            Console.WriteLine(locSString.ToString)
        Next
        Console.WriteLine()
        Console.WriteLine("Taste drücken zum Beenden!")
        Console.ReadKey()
    End Sub
```

Wenn Sie dieses Beispiel laufen lassen, produziert es folgende Ausgabe im Konsolenfenster:

```
Ausgangszeichenfolge:
Wenn man seinen Kopf gegen eine Wand schlägt, verbraucht man 150 Kalorien.

'Addieren' von Strings - 'Toll, was?' anhängen:
Wenn man seinen Kopf gegen eine Wand schlägt, verbraucht man 150 Kalorien.
 - Toll, was?

'Subtrahieren' von Strings - ', was?' abziehen:
Wenn man seinen Kopf gegen eine Wand schlägt, verbraucht man 150 Kalorien.
 - Toll?

'Subtrahieren' von Strings - die letzten 9 Zeichen abziehen:
Wenn man seinen Kopf gegen eine Wand schlägt, verbraucht man 150 Kalorien.

'Vervielfachen' von Strings:
Wenn man seinen Kopf gegen eine Wand schlägt, verbraucht man 150 Kalorien.Wenn m
an seinen Kopf gegen eine Wand schlägt, verbraucht man 150 Kalorien.Wenn man sei
nen Kopf gegen eine Wand schlägt, verbraucht man 150 Kalorien.Wenn man seinen Ko
pf gegen eine Wand schlägt, verbraucht man 150 Kalorien.

'(Auf)teilen' von Strings:
Der Glückskeks wurde 1916 von George Jung, einem amerikanischen Nudelmacher erfu
nden.
Der
Glückskeks
wurde
1916
von
George
Jung,
einem
amerikanischen
Nudelmacher
erfunden.

Taste drücken zum Beenden!
```

Implementierung von Operatoren

Nachdem diese Vorbereitungen abgeschlossen sind, schreiten wir zur nächsten Tat: dem eigentlichen Implementieren der statischen Operatorenprozeduren. Dazu können wir dann die aus der Mathematik bekannten Rechenoperatoren verwenden.

Hierbei unterscheiden wir zwischen zwei Typen: den eigentlichen Operatoren, wie +, –, *, / etc. und den Operatoren, die zur Konvertierung von Typen dienen. Wir beginnen mit der ersten Gruppe – der Implementierung der von uns benutzten Rechenoperatoren. Die generelle Ausführung der Operatorenimplementierung lautet folgendermaßen:

```
Public [Class|Structure] OpTyp
    Public Shared Operator OpChar(ByVal objVar1 As [OpTyp|Typ1], ByVal objVar2 As [OpTyp|Typ2]) As Typ3
        ' Hier steht der Code, der die eigentliche Operation durchführt
    End Operator
End [Class|Structure]
```

Dieser Rumpf soll verdeutlichen, auf was es ankommt:

- Operatoren lassen sich auf Klassen *und* Strukturen anwenden.
- Welcher Operator (+, –, *, / etc.) zur Anwendung kommt, wird durch `OpChar` bestimmt. Tabelle 15.1 listet auf, welche Rechenoperatoren sich implementieren lassen (und wofür sie eigentlich gedacht sind).
- Mindestens einer der Parameter, den Sie einer Operatorenprozedur übergeben, muss vom Typ sein wie die Klasse bzw. Struktur, die die Operatorenprozedur definiert.
- Die Operatorenprozedur muss, wie schon erwähnt, statischer Natur und deswegen mit dem Modifizierer `Shared` definiert sein.
- Der Rückgabetyp kann beliebig sein.

Auf unser Beispiel angewendet, würde die Additionsroutine sich dann folgendermaßen gestalten:

```
Public Shared Operator +(ByVal sstring1 As SuperString, ByVal sstring2 As SuperString) As SuperString
    Return sstring1.Addieren(sstring2)
End Operator
```

Durch den Plus-Operator sollen die beiden Parameter, die jeweils links und rechts vom Plus-Operator stehen, »addiert«, also in unserem Fall miteinander verkettet werden. Der links vom Operator stehende Parameter entspricht dabei dem ersten der Operatorenprozedur übergebenen Parameter, der rechts vom Operator stehende dem zweiten Parameter.

HINWEIS Diese Art der Implementierung funktioniert problemlos, da wir unseren `SuperString`-Typ auf Basis einer Struktur, also eines Wertetyps, konzipiert haben. Hätten wir ihn als Referenztyp konzipiert, wären bei der Implementierung der Operatorenprozeduren auf diese Weise Probleme buchstäblich vorprogrammiert. Lesen Sie vor der Implementierung von Operatorenprozeduren in eigene Klassen deswegen auch unbedingt den Abschnitt »Implementieren von Wahr- und Falsch-Auswertungsoperatoren« ab Seite 508.

Sobald diese Operatorenprozedur Bestandteil der Klasse geworden ist, können wir den Code im Modul für die Addition der beiden Strings umstellen. Aus

```
locSuperString = locSuperString.Addieren( _
        New SuperString(vbNewLine + " - Toll, was?"))
```

wird dann

```
locSuperString = locSuperString + New SuperString(vbNewLine + " - Toll, was?")
```

Und das ist schon wesentlich bequemer zu handhaben und besser lesbar, finden Sie nicht?

In diesem Stil haben wir dann die Möglichkeit, auch die anderen Operatoren zu implementieren:

```
    Public Shared Operator -(ByVal sstring1 As SuperString, ByVal sstring2 As SuperString) As
SuperString
        Return sstring1.Subtrahieren(sstring2)
    End Operator
```

```
    Public Shared Operator *(ByVal sstring1 As SuperString, ByVal anzahl As Integer) As SuperString
        Return sstring1.Vervielfachen(anzahl)
    End Operator

    Public Shared Operator /(ByVal sstring1 As SuperString, ByVal trennzeichen As Char) As SuperString()
        Return sstring1.Teilen(trennzeichen)
    End Operator
```

Überladen von Operatorenprozeduren

Nun gibt es noch einen Punkt, der bei Operatorenprozeduren noch angepasst werden sollte: Unsere ursprüngliche Subtraktionsroutine gibt es in zwei Überladungsversionen. Die erste übernimmt einen `SuperString` als Parameter; dieser wird dann in der Ausgangszeichenkette gesucht und aus ihr entfernt, wenn es eine Suchübereinstimmung gab. Die zweite Möglichkeit: Sie können einen `Integer`-Wert als Parameter angeben, der die Anzahl der Zeichen bestimmt, die vom hinteren Teil der Zeichenkette entfernt werden sollen. Diese Funktion ist im Moment noch nicht durch einen Operator aufrufbar.

Doch wie »normale« Methoden können Sie auch für Operatorenprozeduren Überladungen anwenden. Es gilt dabei das in Kapitel 10 im Abschnitt »Überladen von Methoden, Konstruktoren und Eigenschaften« Gesagte. Wenn wir die Sammlung der Operatorenprozeduren um die folgende überladene Methode ergänzen

```
Public Shared Operator -(ByVal sstring1 As SuperString, ByVal anzahl As Integer) As SuperString
        Return sstring1.Subtrahieren(anzahl)
End Operator
```

wird es möglich, beide Versionen der Subtraktion im Modul auf Operatoren umzustellen:

```
.
.
.
        'Subtrahieren (rauslöschen) anderer Strings
        Console.WriteLine()
        Console.WriteLine("'Subtrahieren' von Strings - ', was?' abziehen:")
        'locSuperString = locSuperString.Subtrahieren( _
        '        New SuperString(", was"))
        locSuperString = locSuperString - New SuperString(", was")
        Console.WriteLine(locSuperString.ToString)
        Console.WriteLine()

        'Subtrahieren ist überladen - geht auch mit der Anzahl
        'der letzten Zeichen, die entfernt werden sollen.
        Console.WriteLine("'Subtrahieren' von Strings - die letzten 9 Zeichen abziehen:")
        locSuperString = locSuperString - 9
        Console.WriteLine(locSuperString.ToString)
        Console.WriteLine()
.
.
.
```

Implementierung von Vergleichsoperatoren

Prinzipiell lassen sich Vergleichsoperatoren für benutzerdefinierte Typen ähnlich implementieren wie Rechenoperatoren. Es gibt nur zwei zusätzliche Bedingungen:

- Sie müssen als Funktionsergebnis grundsätzlich einen booleschen Datentyp zurückliefern, der bestimmt, ob der Vergleich erfolgreich war oder nicht.
- Sie müssen Vergleichsoperatoren paarweise implementieren. Wenn Sie den Operator implementieren, der auf Gleichheit prüft, müssen Sie auch den implementieren, der auf Ungleichheit prüft. Implementieren Sie den Vergleich auf größer, müssen Sie den auf kleiner ebenfalls einbauen. Das Gleiche gilt für größer gleich und kleiner gleich.

Die Implementierung von Vergleichsoperatoren für unsere `SuperString`-Klasse sieht folgendermaßen aus:

```
Public Shared Operator <>(ByVal sString1 As SuperString, ByVal sString2 As SuperString) As Boolean
    Return (sString1.ToString <> sString2.ToString)
End Operator

Public Shared Operator =(ByVal sString1 As SuperString, ByVal sString2 As SuperString) As Boolean
    Return (sString1.ToString = sString2.ToString)
End Operator

Public Shared Operator <(ByVal sString1 As SuperString, ByVal sString2 As SuperString) As Boolean
    Return (sString1.ToString < sString2.ToString)
End Operator

Public Shared Operator >(ByVal sString1 As SuperString, ByVal sString2 As SuperString) As Boolean
    Return (sString1.ToString > sString2.ToString)
End Operator
```

Implementierung von Typkonvertierungsoperatoren mit dem Operator CType

Typkonvertierungsoperatoren sind ein zweischneidiges Schwert. Sie erhöhen den »Degree of Convenience« auf der einen Seite um ein weiteres Maß, können aber unter Umständen auch zu erheblichen Problemen führen – aber dazu später mehr.

Erinnern wir uns: Eine implizite Konvertierung, also eine, bei der nichts Zusätzliches gemacht werden muss, können Sie anwenden, wenn Sie einen kleineren Datentyp in einen größeren Datentyp überführen – beispielsweise einen `Integer`-Wert in einen `Long`-Wert. Dazu ein Beispiel:

```
'Hier geht's mit impliziter (da typerweiternder) Konvertierung
Dim einLong As Long, einInteger As Integer
einInteger = 10
einLong = einInteger
```

Vor den umgekehrten Weg einer typverkleinernden Typkonvertierung schiebt der Basic-Compiler jedoch erst einmal einen Riegel – jedenfalls so lange, bis Sie sich mit `CType` (oder einem `Cxxx`-Operator) da »rauskaufen«. Sie sollten sich bewusst sein, dass Daten bei einer typverkleinernden (oder den Typ völlig verändernden) Konvertierung verloren gehen können, und deswegen macht Sie Visual Basic mit dem Einsatzzwang von `CType` darauf aufmerksam. Derart *explizite* Konvertierungen sehen dann so

```
'Das hier erfordert eine explizite, da typverkleinernde Konvertierung
einLong = 1000
einInteger = CType(einLong, Integer)
```

oder so aus:

```
'Aber auch diese Konvertierung muss explizit durchgeführt werden
Dim einDouble As Double, einString As String
einString = "1.828.488.382,45"
einDouble = CType(einString, Long)
```

Nun werden Typen natürlich nicht »einfach so« konvertiert – gerade bei einer komplexeren Typkonvertierung wie von einer Zeichenkette in einen numerischen Wert läuft ein gar nicht so anspruchloses Programm ab, das diese Konvertierung vornimmt.

Und ein solches Programm – oder besser: eine solche Unterroutine – können Sie mit den so genannten `Operator CType`-Prozeduren für Ihre eigenen Datentypen implementieren. Dabei haben Sie es in der Hand, ob eine implizite oder eine explizite Konvertierung erfolgen soll. Für unser Beispiel wäre es doch schön, wenn die folgende Zeile funktionieren würde:

```
Dim einSuperString As SuperString = "Dies ist eine Zeichenkette"
```

Das können Sie haben. Bei der Konstanten, die dem `SuperString` zugewiesen wird, handelt es sich um eine vom Typ `String`. Da wir an dieser Stelle ohne `CType` arbeiten wollen (und können, denn es droht bei der Konvertierung kein Verlust), müssen wir eine Konvertierungsoperatorenprozedur implementieren, die den Datentyp erweitert. Und das geht so (und jetzt halten Sie sich fest, was die Modifizierer der folgenden Prozedur betrifft):

```
Public Shared Widening Operator CType(ByVal normaloString As String) As SuperString
    Return New SuperString(normaloString)
End Operator
```

`Operator CType` zeigt hier an, dass die Routine für eine Typkonvertierung zuständig ist. Der Modifizierer `Widening` bestimmt, dass es sich um eine *implizite* Konvertierung handelt, bei der die Ausformulierung von `CType` nicht notwendig ist. Und `Shared` schließlich, aber das wissen Sie ja, bestimmt, dass die Routine statischer Natur ist. Der eigentliche Konvertierungscode ist simpel: Die Prozedur legt auf Basis des übergebenden `String` eine neue `SuperString`-Instanz an und liefert diese als Funktionsergebnis zurück.

Würden Sie wollen, dass der Entwickler, der Ihre Klasse verwendet, eine explizite Typkonvertierung mit `CType` einleiten muss, dann würden Sie den Modifizierer `Widening` durch `Narrowing` (verkleinern) ersetzen.

Natürlich ist diese »Erweitern-/Verkleinern-Geschichte« bei Datentypen nur eine Richtlinie. Ihnen steht es selbstverständlich frei, jeden Datentyp implizit oder explizit konvertierbar zu machen, ganz gleich, ob dabei

Daten auf der Strecke bleiben können (wie bei `Long` zu `Integer`) oder nicht. Mir persönlich stoßen die Modifizierernamen ein wenig sauer auf, da sie mehr verwirren als nützen. *Implicit* und *Explicit* als Modifizierernamen wären mir lieber gewesen – aber wahrscheinlich hätte das wieder zu Problemen geführt, weil *Explicit* im Visual Basic-Dialekt schon seit Jahren im Rahmen von `Option Explicit` eingesetzt wird. Doch das ist eine bloße Vermutung.

Übrigens: Die Konvertierung, die Sie nun implementiert haben, funktioniert derzeit nur in eine Richtung. Würden Sie versuchen, den umgekehrten Weg mit

```
Dim einString as String = einSuperString
```

zu gehen, sähen Sie in der Fehlerliste die Meldung:

```
Der Wert vom Typ "SuperStringVorstellung.SuperString" kann nicht in "String" konvertiert werden.
```

Damit beide Richtungen funktionierten, müssten Sie eine weitere `CType Operator`-Prozedur zum Projekt hinzufügen, nämlich:

```
Public Shared Widening Operator CType(ByVal SuperString As SuperString) As String
    Return SuperString.ToString
End Operator
```

Implementieren von Wahr- und Falsch-Auswertungsoperatoren

Eine Möglichkeit, Wahr- und Falsch-Auswertungsmechanismen zu implementieren, bestünde darin, implizite oder explizite Konvertierungen in den Datentyp `Boolean` für Ihre Klasse anzubieten. Doch Visual Basic sieht für diesen Zweck eine weitere Möglichkeit vor – die Operatoren `IsTrue` und `IsFalse`. Diese Operatoren stellen keine Anwendungsmöglichkeit im herkömmlichen Sinne bereit – Sie können die Operatoren `IsTrue` und `IsFalse` also nicht als Namen wie `CType` in Ihren eigenen Programmen einsetzen.

Sie dienen vielmehr nur als Hilfen bei der Definition von Operatorenprozeduren, um eine bestimmte Prozedur für eine Operation festzulegen, die ähnlich der impliziten Datentypkonvertierung, eigentlich gar keine Operatoren benötigt.

Unsere `SuperString`-Klasse könnte beispielsweise einen Mechanismus bereitstellen, der definiert, dass bestimmte Zeichenketteninhalte bei Auswertungen `True` ergeben, und alle anderen dagegen `False` zum Ergebnis haben. In diesem Fall ließe sich folgende Vorgehensweise implementieren:

```
    Sub ExperimenteFürBoolscheAusdrücke()
        'Hier geht's mit impliziter (da typerweiternder) Konvertierung
        Console.Write("Möchten Sie weitere Daten eingeben (Ja, Nein):")
        Dim locSupStr As SuperString = Console.ReadLine
        If locSupStr Then
            Console.WriteLine("OK, dann geben Sie mal ein!")
        Else
            Console.WriteLine("dann halt nicht...")
        End If
```

```
        Console.WriteLine()
        Console.WriteLine("Taste drücken zum Beenden!")
    End Sub
```

Die entsprechenden Operatorenprozeduren, die diese Vorgehensweise ermöglichen, könnte man beispielsweise folgendermaßen aufbauen:

```
    Public Shared Operator IsTrue(ByVal sString As SuperString) As Boolean
        Dim locString As String = sString
        locString = locString.ToUpper
        Select Case locString
            Case "JA"
                Return True
            Case "J"
                Return True
            Case "RICHTIG"
                Return True
            Case "WAHR"
                Return True
            Case "AUSGEWÄHLT"
                Return True
            Case "GEDRÜCKT"
                Return True
            Case "BESTÄTIGT"
                Return True
            Case "Y"
                Return True
            Case "YES"
                Return True
            Case "TRUE"
                Return True
            Case "CORRECT"
                Return True
            Case "SELECTED"
                Return True
            Case "PRESSED"
                Return True
            Case "ACCEPTED"
                Return True
            Case "CONFIRMED"
                Return True
        End Select
        Return False
    End Operator

    Public Shared Operator IsFalse(ByVal sString As SuperString) As Boolean
        If sString Then
            Return False
        Else
            Return True
        End If
    End Operator
```

HINWEIS Wie bei Vergleichsoperatoren müssen Sie die Operatoren `IsTrue` und `IsFalse` paarweise einbinden. Auch wichtig: Sie sollten es bei der Einbindung von `IsTrue` und `IsFalse` in Erwägung ziehen, auch den `Not`-Operator zu verdrahten. Anderenfalls kann der Entwickler, der Ihre Klasse anwendet, einen Ausdruck wie den folgenden nicht anwenden:

```
If Not locSupStr Then
    .
    .
    .
End If
```

Problembehandlungen bei Operatorenprozeduren

Operatorenprozeduren können, richtig eingesetzt, eine enorme Erleichterung für den Entwickler darstellen. Gleichzeitig sollten Sie aber auch einige Dinge beherzigen, bei denen Operatorenprozeduren dafür verantwortlich sein können, dass sich durch ihre Implementierung Fehler einschleichen.

Aufgepasst bei der Verwendung von Referenztypen

In unserem Beispiel haben wir die Operatorenprozeduren in eine Struktur eingebaut. Damit handelt es sich automatisch um einen Wertetyp, den auch die Operatorenprozeduren verarbeiten. Nun schauen Sie sich eine der Rechenoperatorenprozeduren noch einmal genauer an:

```
Public Shared Operator +(ByVal sstring1 As SuperString, ByVal sstring2 As SuperString) As SuperString
    Return sstring1.Addieren(sstring2)
End Operator
```

Die `Addieren`-Funktion wird hier auf `sstring1` angewendet und `Addieren` liefert – wenn Sie sich den entsprechenden Code anschauen – ohnehin eine neue Instanz von `SuperString` zurück. Aber das muss nicht so sein. Manche Implementierungen »neigen dazu«, das Objekt selbst zu verändern. Würde `Addieren` das machen – wäre `Addieren` also eine Methode, die kein Funktionsergebnis liefern würden –, würde der entsprechende Code der Operatorenprozedur vielleicht so aussehen:

```
Public Shared Operator +(ByVal sstring1 As SuperString, ByVal sstring2 As SuperString) As SuperString
    sstring1.Addieren(sstring2)
    Return sstring1
End Operator
```

Solange es sich bei den Werten, die `sstring1.Addieren` manipuliert, um Wertetypen handelt, und solange `sstring1` selbst ein Wertetyp ist, hat eine solche Prozedur immer noch nichts Gefährliches an sich.

Schlimm kann es nur dann werden, wenn `Addieren` einen Referenztypen manipuliert und es sich bei `sstring1` selbst ebenfalls um einen Referenztypen handeln würde.

In diesem Fall würden Sie nämlich als Parameter in `sstring1` im Grunde genommen einen Zeiger auf die eigentlichen Objektdaten entgegennehmen. Das anschließende Addieren würde also keine Kopie von `sstring1` verändern, sondern die einzig existierende Instanz – die ursprünglich im aufrufenden Code vor dem Operator gestanden hätte. Sie würden damit in der Objektvariablen, die Sie als Operanden übergeben, und in der Objektvariablen, der Sie den Ausdruck zuweisen, das gleiche Ergebnis vorfinden. Zur Verdeutlichung (und bei dem folgenden Code nehmen wir an, dass es sich beim Typ, für den Operatoren definiert sind, um einen Referenztyp handelt):

```
Dim objVar1 As New RefType(10)
Dim objVar2 As New RefType(20)
Dim objVar3 As RefType
objVar3 = ObjVar1 + objVar2
```

Vorausgesetzt, `RefType` würde in der Lage sein, Zahlen zu addieren. Dann würde das Ergebnis 30 – das eben vorgestellte Szenario angenommen – nach Abschluss nicht nur in `objVar3`, sondern auch in `objVar1` »stehen«. Und im Grunde genommen ist es sogar noch schlimmer: Da `objVar3` und `objVar1` auf die gleichen Daten zeigen, würde eine Veränderung von `objVar1` immer auch eine Veränderung von `objVar3` nach sich ziehen.

Gerade bei Operatorenprozeduren, bei denen Sie ein solches Verhalten am wenigsten erwarten, sollten Sie daher darauf achten, dass diese als Rückgabewert immer eine neue Instanz ihres Typs zurückgeben, wenn sie Referenztypen verarbeiten.

Aufgepasst bei Mehrdeutigkeiten bei der Auflösung von Signaturen

In unserer Beispielklasse ist, nachdem wir die Typkonvertierungsfunktionen vollständig implementiert haben, Folgendes erlaubt:

```
einSuperString = einSuperString + "Klaus"
```

Auf den ersten Blick mag das merkwürdig erscheinen, denn für die Addition haben wir keine Überladungsversion implementiert, die einen »einfachen« String akzeptieren würde. Dennoch meldet der Visual Basic-Compiler keinen Fehler. Und das zurecht, denn:

Die einzige Operatorenprozedur, die das Addieren mit dem +-Operator erlaubt, nimmt als zweiten Parameter eine Variable vom Typ `SuperString` entgegen. Der wiederum verfügt aber über einen impliziten Typkonvertierungsmechanismus, der einen normalen `String` in einen `SuperString` umwandeln kann. Der Compiler macht also das einzig Richtige: Er sorgt dafür, dass »Klaus« implizit zunächst in einen `SuperString` konvertiert wird, und dieser `SuperString` wird anschließend der `Additions`-Operatorenprozedur übergeben.

Was passiert aber, wenn es sowohl eine implizite Konvertierung von `String` in `SuperString` als auch eine überladene Version des Additionsoperators gibt, die Strings akzeptiert? In diesem Fall wird diese Additionsoperationsprozedur verwendet.

Fehler können sich aber – Sie ahnen es schon – dann einschleichen, wenn beide Routinen auf unterschiedliche Weise vorgehen, um den String in einen `SuperString` zu konvertieren. Die anschließende Fehlersuche bei Fehlern in komplexen Typen kann sich dann unter Umständen als sehr mühselig entpuppen.

TIPP Es empfiehlt sich also in jedem Fall, die möglichen Parameterkombinationen durchzuprobieren, herauszufinden, wo welche Konvertierung wirklich stattfindet, und alle Operatorenprozeduren dazu, am besten Schritt für Schritt, mit dem Visual Studio-Debugger zu durchlaufen.

Übersicht der implementierbaren Operatoren

Sie werden staunen, welche Operatoren sich mit Operatorenprozeduren implementieren lassen. Die folgende Tabelle gibt Ihnen die Übersicht.

WICHTIG Beachten Sie, dass Sie einige Funktionen nur paarweise implementieren können.

Die Spalte *Vorgesehene math. Funktion* soll Ihnen übrigens nur eine grobe Themenrichtung für eine Implementierung vorgeben. Aber letzten Endes könnte Sie natürlich keiner daran hindern, dass Ihre Datentypen mit – addiert und mit + subtrahiert werden, wenn Sie es so wollen.

Operator	Vorgesehene math. Funktion	Bemerkung
+	Addition	
–	Subtraktion	
\	Division ohne Rest	
/	Division	
^	Potenzieren	
&	Verknüpfung	
<<	Nach links verschieben	Normalerweise bitweise bei Integerzahlen
>>	Nach rechts verschieben	Normalerweise bitweise bei Integerzahlen
=	Test auf Gleichheit	Hiermit steuern Sie nicht die Zuweisung an eine Objektvariable – dies ist nur mit dem `CType`-Operator möglich. Wenn Sie diesen Operator implementieren, müssen Sie < > (ungleich) ebenfalls implementieren.
< >	Test auf Ungleichheit	Wenn Sie diesen Operator implementieren, müssen Sie = (gleich) ebenfalls implementieren
<	Test auf kleiner	Wenn Sie diesen Operator implementieren, müssen Sie > (größer) ebenfalls implementieren
>	Test auf größer	Wenn Sie diesen Operator implementieren, müssen Sie < (kleiner) ebenfalls implementieren
<=	Test auf kleiner oder gleich	Wenn Sie diesen Operator implementieren, müssen Sie > = (größer gleich) ebenfalls implementieren
>=	Test auf größer oder gleich	Wenn Sie diesen Operator implementieren, müssen Sie < = (kleiner gleich) ebenfalls implementieren
Like	Test auf Ähnlichkeit	
Mod	Restwertermittlung	
And	Logische Und-Verknüpfung	
Or	Logische Oder-Verknüpfung	
Xor	Logische Exklusiv-Oder-Verknüpfung	
Not	Logische Negation	
CType	Steuerung der impliziten oder expliziten Typkonvertierung	Verwenden Sie `Narrowing` für die explizite und `Widening` für die implizite Typkonvertierung
IsTrue	Auswerten von booleschen Ausdrücken	Sie müssen `IsTrue` und `IsFalse` paarweise implementieren. Sie sollten in diesem Fall auch einen `Not`-Operator zur Verfügung stellen. WICHTIG: Diese Operatoren können nur in Auswertungskonstruktionen wie `If … Then … Else`, `Do While` etc. verwendet werden – sie stellen *keine* implizite Konvertierung in den Datentyp `Boolean` bereit!
IsFalse	Auswerten von booleschen Ausdrücken	Es gilt das zu `IsTrue` gesagte. Mehr Informationen zu diesem Thema finden Sie im Abschnitt »Implementieren von Wahr- und Falsch-Auswertungsoperatoren« ab Seite 508

Tabelle 15.1 Operatoren in Visual Basic 2010, die für die Implementierung in eigenen Typen zur Verfügung stehen

Kapitel 16

Ereignis, Delegaten und Lambda-Ausdrücke

In diesem Kapitel:

Konsumieren von Ereignissen mit WithEvents und Handles 515
Auslösen von Ereignissen 518
Ereignisparameter zur Verfügung stellen 521
Delegaten 525
Lambda-Ausdrücke 534
Dynamisches Anbinden von Ereignissen mit AddHandler 538
Implementieren eigener Event Handler 547

Ereignisse kennen Sie wahrscheinlich in erster Linie aus der Windows-Programmierung, und zwar wenn es darum geht, auf bestimmte Benutzerereignisse wie das Klicken von Schaltflächen oder das Auswählen von Elementen in einer Liste zu reagieren. Doch das reine Konsumieren dieser vordefinierten Ereignisse stellt dabei nur die Spitze des Eisberges dar.

Klassen können auch eigene Ereignisse auslösen, die dann wiederum von anderen Klassen benutzt werden. Dabei sollten sich ereignisauslösende Klassen an bestimmte Regeln halten, von denen später noch die Rede sein wird. Und: Ereignisse können auch nachträglich, also zur Laufzeit, mit entsprechenden Behandlungsroutinen verknüpft und damit nur im Bedarfsfall konsumiert werden. Damit werden auch Szenarien möglich, die noch in Visual Basic 6.0 ohne die Hilfe von externen DLLs nicht möglich gewesen wären.

Hinter den Kulissen basieren Ereignisse auf Listen mit so genannten Delegaten. Variablen vom Typ Delegat (und von ihnen abgeleitete) funktionieren im Prinzip wie normale Variablen, nur dass sie keine Daten, sondern Zeiger auf Methoden speichern. Deswegen kann man eine Delegatenvariable auch »aufrufen«; das Programm überträgt die Ausführung dann nämlich an die Methode, auf die die Delegatenvariable zeigt.

Dieses Kapitel beschäftigt sich mit Ereignissen und stellt diese Technologien nicht nur vor, sondern es zeigt auch, wie Sie für Ihre eigenen Komponenten Ereignisse erstellen, die diese zur Laufzeit auslösen und die dann von anderen Komponenten konsumiert werden können.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 16\Alarm01
```

Dieses Beispielprogramm, das in der ersten Version einen einfachen Wecker imitiert, werden wir im Folgenden für die Ereigniserkundigungen verwenden. Wenn Sie dieses Beispielprojekt starten, sehen Sie ein Formular, wie Sie es auch in Abbildung 16.1 erkennen können.

Abbildung 16.1 Die Alarmwecker-Anwendung soll Ihnen anschauliche Beispiele für den Umgang mit Ereignissen geben

Ein paar Worte zur groben Funktionsweise des Programms: Sie haben die Möglichkeit, im Feld *Alarmzeit* eine Uhrzeit im Format *HH:mm:ss*[1] anzugeben. Aktivieren Sie anschließend das Kontrollkästchen *Wecker aktivieren*, zeichnet das Programm eine kleine rote Markierung in das Ziffernblatt der Uhr ein, die die Alarmzeit markiert. Ist diese Zeit erreicht, geht der Wecker los – in diesem Beispiel durch ein rotes Blinken des gesamten Zeichenbereichs der Uhr.

Wie die Uhr tatsächlich in das `PictureBox`-Steuerelement gezeichnet wird, ist in diesem Kapitel von untergeordnetem Interesse.

Wichtig in diesem Zusammenhang ist eine Klasse, die das »Wecken« übernimmt. Diese Klasse funktioniert im Prinzip wie ein handelsüblicher Alarmwecker im wirklichen Leben. Mit einer bestimmten Eigenschaft stellen Sie die Weckzeit ein, und wenn diese Uhrzeit erreicht ist, löst die Klasse ein Ereignis aus.

Im Grunde genommen gibt es zwei Möglichkeiten, Ereignisse zu konsumieren:

- Durch das Zuweisen einer beliebigen Prozedur mit einer Feldvariable einer Klasse, die mit `WithEvents` deklariert wurde.
- Durch das manuelle Hinzufügen einer Ereignisbehandlungsprozedur zur Laufzeit mit `AddHandler`.

Eine schnellere Möglichkeit als das Konsumieren von Ereignissen gibt es, wenn in der Ableitung einer Klasse Ereignisauslöser mit `Onxxx`-Methoden implementiert werden. In diesem Fall ergibt es mehr Sinn, die ereignisauslösenden Routinen zu überschreiben und die Ereignisbehandlung auf diese Weise zu implementieren. Bei Windows Forms-Anwendungen sollte das der bevorzugte Weg sein, um Formularereignisse zu verdrahten.

HINWEIS Wie Sie mithilfe von Editor bzw. Designer Rümpfe von Ereignisbehandlungsroutinen in Ihren Code einfügen, hat Kapitel 5 (Abschnitt »Das Eigenschaftenfenster«) bereits beschrieben. Dieses Kapitel geht den nächsten Schritt und zeigt, wie die Ereignisbehandlung im Detail funktioniert.

Konsumieren von Ereignissen mit WithEvents und Handles

Wenn Sie in ein leeres Formular eine Schaltfläche einfügen und doppelt auf diese Schaltfläche klicken, sorgt die Visual Studio-IDE dafür, dass der Editor geöffnet, der Formularcode angezeigt und ein Funktionsrumpf in diesem eingefügt wird, der später dann aufgerufen wird, wenn der Benutzer zur Laufzeit die Schaltfläche betätigt.

Damit die Behandlung solcher Ereignisse möglich wird, bedarf es dreier Komponenten: Einerseits muss die Objektvariable, die das Ereignis anbietet, mit `WithEvents` deklariert worden sein. Andererseits muss die

[1] Kleiner Tipp am Rande: Die großgeschriebenen »H«s signalisieren, dass es sich um eine Darstellung im 24-Stunden-Format handelt – kleine »h«s würden das 12-Stunden-Format angeben. »H« ist dabei die Abkürzung des englischen »Hours« (Stunden), »m« steht für »Minutes« (Minuten) und »s« für »Seconds« (Sekunden). Für ein Datum wählt man die Formatabkürzungen »y« für »Years« (Jahre), »M« für »Months« (Monate) – großgeschrieben übrigens, um sie von den Minuten zu unterscheiden – sowie »d« für »Days« (Tage). Ein deutsches Datumsformat entspräche demnach *dd.MM.yyyy*, ein typisch amerikanisches *MM/dd/yyyy* (der Monat wird hier zuerst genannt!).

Signatur[2] der Prozedur, die das Ereignis verarbeiten soll, mit der Signatur des Ereignisses übereinstimmen, die das Objekt anbietet. Und zu guter Letzt muss die Prozedur mit dem Schlüsselwort `Handles` mit dem Objektereignis (oder auch mit anderen Ereignissen dieses Objektes bzw. mit den Ereignissen anderer Objekte, falls deren Signaturen dieselben sind) verdrahtet werden.

Die Routinen, die im Code des Beispiels diese Art von Ereignisbehandlung durchführen, finden Sie im Folgenden:

```
    'Ereignisbehandlungsroutine, die ausgelöst wird,
    'wenn der Alarmgeber Alarm signalisiert, weil eine
    'bestimmte Uhrzeit erreicht wurde.
    Private Sub myAlarmgeber_Alarm(ByVal Sender As Object, ByVal e As AlarmEventArgs) _
                            Handles myAlarmgeber.Alarm
        'Wecker schellt!
        myAlarmStatus = True
        'Und zwar so lange.
        myAlarmDownCounter = myAlarmDauer
        'Abschalten sollten wir das Schellen können.
        btnAusschalten.Enabled = True
        'Und morgen um die gleiche Zeit soll er wieder schellen.
        e.Neustellen = True
    End Sub

    'Ereignisbehandlungsroutine, die ausgelöst wird,
    'wenn der Inhalt der Picturebox neugezeichnet werden soll.
    Private Sub picWecker_Paint(ByVal sender As Object, ByVal e As System.Windows.Forms.PaintEventArgs) _
                            Handles picWecker.Paint
        If myAlarmgeber IsNot Nothing AndAlso myAlarmgeber.AlarmAktiviert Then
            DrawClock(e.Graphics, Date.Now, myAlarmgeber.Alarmzeit, myAktuelleFarbe)
        Else
            DrawClock(e.Graphics, Date.Now, myAktuelleFarbe)
        End If
    End Sub
.
.
.
    'Ereignisbehandlungsroutine, die ausgelöst wird,
    'wenn der Timer abgelaufen ist. Dies passiert
    'alle 500 Millisekunden, und wir zeichnen dann
    'die komplette Uhr neu - berücksichtigen dabei
    'auch das Hintergrundblinken, falls der "Wecker
    'gerade schellt".
    Private Sub myTimer_Tick(ByVal sender As Object, ByVal e As System.EventArgs) Handles myTimer.Tick
        'Läuft der Alarm gerade?
        If myAlarmStatus Then
            'Ja, Farben alle 500 ms wechseln
            If myAktuelleFarbe = Color.White Then
                myAktuelleFarbe = Color.Red
            Else
```

[2] Zur Erinnerung: Die Signatur einer Prozedur ergibt sich aus der Abfolge der Typen, die eine Prozedur als Parameter entgegennimmt.

```
                myAktuelleFarbe = Color.White
            End If
            'Alarmdauerzähler vermindern
            myAlarmDownCounter -= 1
            If myAlarmDownCounter = 0 Then
                'Alarm ausschalten, wenn dieser abgelaufen ist.
                AlarmAusschalten(True)
            End If
        End If

        'Ganze Uhr in jedem Fall neuzeichnen.
        picWecker.Invalidate()
    End Sub
```

In diesem Fall sind es drei Objekte, die im Gültigkeitsbereich der Klasse deklariert wurden und damit Feldvariablen sind, deren Ereignisse von diesen Prozeduren behandelt werden:

- `myTimer`,
- `picWecker` sowie
- `myAlarmgeber`

Und mit der Ausnahme der Objektvariablen `picWecker`, bei der es sich um eine Objektvariable handelt, die ein Steuerelement repräsentiert und die mithilfe des Designers ihren Weg ins Formular gefunden hat, sind dann auch alle ereignisanbietenden Variablen mit dem `WithEvents`-Schlüsselwort deklariert worden:

```
Public Class frmMain

    Private WithEvents myTimer As Timer
    Private WithEvents myAlarmgeber As EinfacherAlarmgeber
```

Erst dann bietet IntelliSense entsprechende Ereignisse zum Objekt an, wenn die Verdrahtung einer Prozedur mit `Handles` erfolgen soll, wie Abbildung 16.2 zeigt.

Abbildung 16.2 IntelliSense hilft Ihnen auch beim Verdrahten von Ereignissen mit einer Prozedur – sowohl bei der Auswahl möglicher ereignisanbietender Objekte als auch bei der Auswahl der Ereignisse

Übrigens: Auch Objektvariablen, die Steuerelemente referenzieren, werden auf die gleiche Weise in den Formularcode eingebunden – Sie können sie jedoch nur nicht auf Anhieb sehen. Der Designer, der ja auch den Formularcode generiert, nutzt nämlich die Möglichkeit von Visual Basic, den Quellcode einer Klasse auf mehrere physische Dateien zu verteilen. Dadurch wird die eigentliche Codedatei eines Formulars aufgeräumter – einige Dinge, die hinter den Kulissen passieren, fehlen aber natürlich in dieser.

Um den Rest des Formularcodes sichtbar zu machen, klicken Sie im Projektmappen-Explorer auf das Symbol *Alle Dateien anzeigen* am oberen Rand dieses Toolfensters (der ToolTip hilft Ihnen beim Finden des richtigen Symbols). Sie werden sehen, dass sich anschließend vor jeder Formulardatei ein kleines Pluszeichen befindet, dessen dahinter stehenden Zweig Sie per Klick öffnen können. Sie werden des Weiteren feststellen, dass jedes Formular über eine Datei namens *form.Designer.vb* verfügt, die diesen »Hinter-den-Kulissen-Code« beinhaltet. Und hier finden wir auch die Deklaration der verwendeten Steuerelemente – mit dem für die Ereignisverdrahtung notwendigen `WithEvents`:

```
.
.  ' Am Ende der Datei frmMain.Designer.vb:
.
    Friend WithEvents picWecker As System.Windows.Forms.PictureBox
    Friend WithEvents Label1 As System.Windows.Forms.Label
    Friend WithEvents mtbAlarmzeit As System.Windows.Forms.MaskedTextBox
    Friend WithEvents chkAlarmAktivieren As System.Windows.Forms.CheckBox
    Friend WithEvents btnOK As System.Windows.Forms.Button
    Friend WithEvents btnAusschalten As System.Windows.Forms.Button

End Class
```

Auslösen von Ereignissen

Sie sehen im Codeauszug der Ereignisbehandlungsroutinen, der ab Seite 516 beginnt, wie Ereignisse generell mit bestimmten Prozeduren verknüpft werden. Der fett hervorgehobene Teil dieses Listingausschnittes konsumiert ein Ereignis der Instanz einer Klasse, die nicht Bestandteil von .NET Framework ist – diese Klasse ist in Heimarbeit entstanden, und sie soll demonstrieren, wie Klassen Ereignisse auslösen können.

Schauen wir uns dazu den relevanten Klassencode der Datei *Alarmgeber.vb* an:

```
Public Class EinfacherAlarmgeber

    Private WithEvents myTrigger As Timer
    Private myAlarmzeit As Date
    Private myAlarmAktiviert As Boolean
    Private mySchwellwert As Integer = 2

    ''' <summary>
    ''' Wird ausgelöst, wenn eine bestimmte Zeit erreicht wurde.
    ''' </summary>
    ''' <param name="Sender">Das Objekt, das dieses Ereignis ausgelöst hat.</param>
    ''' <param name="e">AlarmEventArgs, die Näheres zum Objekt aussagen.</param>
    ''' <remarks></remarks>
    Public Event Alarm(ByVal Sender As Object, ByVal e As AlarmEventArgs)

    Sub New(ByVal Alarmzeit As Date)
        Me.Alarmzeit = Alarmzeit
    End Sub
```

```
    Sub New(ByVal Alarmzeit As Date, ByVal Aktiviert As Boolean)
        Me.Alarmzeit = Alarmzeit
        Me.AlarmAktiviert = Aktiviert
    End Sub
.
.
.
    Private Sub myTrigger Tick(ByVal sender As Object,
                               ByVal e As System.EventArgs) Handles myTrigger.Tick
        If myAlarmzeit < DateTime.Now Then
            Dim locWeckzeitEventArgs As New AlarmEventArgs(Alarmzeit)
            OnWecken(locWeckzeitEventArgs)
            If locWeckzeitEventArgs.Neustellen Then
                Alarmzeit = locWeckzeitEventArgs.Alarmzeit
            Else
                AlarmAktiviert = False
            End If
        End If
    End Sub

    ''' <summary>
    ''' Löst das Wecken aus.
    ''' </summary>
    ''' <param name="e"></param>
    ''' <remarks></remarks>
    Protected Overridable Sub OnWecken(ByVal e As AlarmEventArgs)
        RaiseEvent Alarm(Me, e)
    End Sub
End Class
```

Zunächst einmal benötigt die Klasse selbst eine Hilfe, die für das Auslösen des Ereignisses zur rechten Zeit sorgt, denn sie muss in regelmäßigen Abständen getriggert werden, um herauszufinden, ob die gestellte Alarmzeit bereits erreicht wurde. Dazu verwendet sie ein `Timer`-Objekt namens `myTrigger`, bindet es mit `WithEvents` als Feldvariable ein und sorgt durch die Verdrahtung dessen `Tick`-Ereignisses mit der Prozedur `myTrigger_Tick` dafür, dass diese beim Ablaufen des Timers aufgerufen wird.

Hier findet dann der Vergleich der aktuellen Uhrzeit mit der Weckzeit statt. Und wenn die Weckzeit erreicht wurde, dann wird es interessant: `OnWecken` wird aufgerufen und diese Routine sorgt nun dafür, dass das eigentliche Ereignis mit `RaiseEvent` ausgelöst wird. `RaiseEvent` kann alle Ereignisse auslösen, die zuvor auf Klassenebene durch das `Event`-Schlüsselwort definiert wurden – in unserem Beispiel ist das lediglich das Ereignis `Alarm`.

Events können nicht vererbt werden – der Umweg über Onxxx

Falls Sie sich nun fragen, wieso `myTrigger_Tick` den Umweg über `OnWecken` nimmt: Denken Sie an OOP und an Kapitel 11!

Vielleicht möchten Sie ja Ereignisse in einer Basisklasse implementieren und diese dann in abgeleiteten Klassen nutzen, also dort sowohl in Erfahrung bringen, dass sie ausgelöst werden, aber sie auch auslösen. Letzteres ist allerdings nur über Umwege möglich, denn abgeleitete Klassen können die Ereignisse von ihren Basisklassen nicht verwenden, wie das folgende Beispiel zeigt:

```
'Basisklasse
Public Class BaseClass

    Public Event SomethingHappened(ByVal sender As Object, ByVal e As EventArgs)

    Protected Overridable Sub OnSomethingHappened(ByVal e As EventArgs)
        RaiseEvent SomethingHappened(Me, e)
    End Sub

End Class

'Abgeleitete Klasse
Public Class DerivedClass
    Inherits BaseClass

    Private Sub MethodForRaisingEvent()
        'Fehler:
        'Abgeleitete Klassen können keine Basisklassenereignisse auslösen
        RaiseEvent SomethingHappened(Me, EventArgs.Empty)

        'So aber schon!
        OnSomethingHappened(EventArgs.Empty)

    End Sub

End Class
```

Dieses Beispiel zeigt dann auch, wie es doch möglich ist – nämlich indem die abgeleitete Klasse einfach die `Onxxx`-Methoden aufruft, die die Basisklasse implementiert, und diese wiederum lösen dann das gewünschte Ereignis aus.

Umgekehrt, also für das Mitbekommen eines Ereignisses, gibt es ebenfalls einen Vorteil: Eine abgeleitete Klasse muss bei der Implementierung, wie wir Sie im Wecker-Beispiel kennen gelernt haben, nicht das Ereignis als solches verdrahten. Es reicht vielmehr, dass sie die entsprechende Methode der Basisklasse überschreibt, die das Ereignis auslöst – denn das ist der direkte Weg.

Genau das ist das Prinzip bei allen ereignisauslösenden Komponenten, die Sie in .NET Framework durch Vererbung verwenden – und Formulare gehören übrigens auch dazu. Aus diesem Grund erklärt sich auch die Funktionsweise folgender Routine aus dem Ereignisbeispiel ganz wie von selbst, die die Formularereignisse nicht als Ereignis und mit `Handles` an bestimmte Prozeduren hängt, sondern einfach den Basiscode des Formulars überschreibt: `OnResize` löst nämlich ihrerseits das `Resize`-Ereignis aus, das eintritt, wenn das Formular vergrößert oder verkleinert wird:

```
    'Wird vom Basisklassenteil von System.Windows.Forms.Form angesprungen,
    'wenn sich das Formular vergrößert oder verkleinert hat.
    Protected Overrides Sub OnResize(ByVal e As System.EventArgs)
        'Wichtig: Basisfunktion aufrufen, sonst wird das
        'Resize-Ereignis nicht mehr ausgelöst!
        MyBase.OnResize(e)
        'Inhalt der PictureBox neu zeichnen,
        'wenn sich die Größe des Formulars geändert hat.
        picWecker.Invalidate()
    End Sub
```

HINWEIS Dass überschreibbare Routinen, die Ereignisse auslösen, mit »On« beginnen, liegt einfach an der englischen Sprache: »*Beim* Feststellen der Notwendigkeit zum Auslösen des *Größenändern*-Ereignisses mache Folgendes« würde auf Englisch in etwa »*On* arrival of the *resize* event do the following« heißen, oder kurz: »*BeimGrößenändern*« bzw. `OnResize`.

WICHTIG Denken Sie beim Überschreiben von ereignisauslösenden `Onxxx`-Methoden in abgeleiteten Klassen unbedingt daran, die Basismethode mit `myBase.Onxxx` aufzurufen. Anderenfalls verhindern Sie, dass für Konsumenten von Instanzen Ihrer Klasse Ereignisse ausgelöst werden, da `RaiseEvent` in diesem Fall nicht mehr stattfindet.

Ereignisparameter zur Verfügung stellen

Wenn Sie sich die verschiedenen Ereignisbehandlungsroutinen anschauen, werden Sie ein immer wiederkehrendes Schema in den Ereignissignaturen erkennen:

```
Private Sub myTrigger_Tick(ByVal sender As Object, ByVal e As System.EventArgs) Handles myTrigger.Tick
    .
    .
    .
End Sub
```

- Es gibt einen Parameter – `sender` vom Typ `Object` –, der über den Auslöser des Ereignisses Auskunft gibt. Die Variable `sender` ist deswegen als `Object` deklariert, da es ganz unterschiedliche Objekte (Textboxen, Listenfelder etc.) sein können, die das Ereignis auslösen.
- Es gibt einen Parameter – `e` vom Typ `EventArgs` (oder einer Ableitung von `EventArgs`) –, der entweder nähere Daten zum Ereignis liefert oder mit dessen Hilfe sich die weitere Ereigniskette steuern lässt. Bei der Basisklasse `System.EventArgs` finden sich solche zusätzlichen Informationen nicht. Wenn Sie aber beispielsweise das `KeyDown`-Ereignis eines Formulars implementieren, finden Sie `e` definiert vom Typ `System.Windows.Forms.KeyPressEventArgs`, einem Typ, der unter anderem `e.KeyChar` (also das Zeichen der gedrückten Taste) bereitstellt.

TIPP Wenn Sie ein Ereignis auslösen, das keine individuellen Ereignisparameter benötigt und deswegen mit der Basisimplementierung der Klasse `EventArgs` auskommt, verwenden Sie einfach die statische Methode `Empty`, um an eine `EventArgs`-Instanz zu gelangen:

```
RaiseEvent EinEreignis(Me, EventArgs.Empty)
```

Die Quelle des Ereignisses: Sender

`Sender` beinhaltet grundsätzlich die Quelle des Ereignisses als Objekt. Dies ist insbesondere dann wichtig, wenn eine Ereignisbehandlungsroutine gleich mehrere Ereignisse verschiedener Objekte behandeln soll, was in Visual Basic 6.0 oder VBA noch völlig undenkbar war – denn Sie müssen ja wissen, wer für das Ereignis verantwortlich war und dementsprechend reagieren.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 16\Ereignistest
```

Wenn Sie dieses Projekt starten, sehen Sie wie in Abbildung 16.3 ein Formular mit mehreren Schaltflächen.

Abbildung 16.3 In diesem Beispiel gibt es eine Ereignisbehandlungsroutine für alle drei Schaltflächen

Soweit scheint es an diesem Beispiel auf den ersten Blick nichts Außergewöhnliches zu geben. Das Besondere ist aber: Alle drei Schaltflächen werden für den Fall des Anklickens vom Code berücksichtigt; es gibt aber lediglich eine einzige Ereignisbehandlungsroutine, die diese Behandlung übernimmt:

```
Public Class frmEreignisse

    Private Sub Button1Und2Ereignisse(ByVal sender As System.Object, ByVal e As System.EventArgs) _
            Handles Button1.Click, Button2.Click, Button3.Click

        'Eine MessageBox wird dargestellt, wenn der Anwender Button1, Button2 oder
        'Button3 auslöst.
        MessageBox.Show(sender.ToString & "wurde gedrückt!")

    End Sub
End Class
```

Hier wird deutlich, wozu `sender` gut ist: Um überhaupt zu wissen, welche der Schaltflächen das Ereignis ausgelöst hat, müssen Sie `sender` unter die Lupe nehmen und entsprechende Fallunterscheidungen berücksichtigen, die dann das jeweils Richtige machen.

Übrigens: Da es sich bei `sender` nicht um eine `String`-Variable mit beschreibendem Text, sondern um eine wirkliche Referenz auf das ereignisauslösende Objekt handelt, können Sie `sender` auch wieder zurück zum Beispiel mit `Dim btn As Button=CType(Sender, Button)` in seinen Ausgangstyp casten.

So ist es im Beispiel ganz sicher, dass ein Typ-Casting nicht schiefgehen kann, da ja ausschließlich Schaltflächen zum Einsatz kommen. Die folgenden Zeilen sind also durchaus denkbar, um von `sender` »zurück« zum Schaltflächenobjekt (`Button`-Objekt) zu gelangen und mit diesem Objekt anschließend Manipulationen durchzuführen, die über `sender` selbst nicht möglich gewesen wären:

```
Public Class frmEreignisse

    Private Sub Button1Und2Ereignisse(ByVal sender As System.Object, ByVal e As System.EventArgs)
            Handles Button1.Click, Button2.Click, Button3.Click

        'Eine MessageBox wird dargestellt, wenn der Anwender Button1, Button2 oder
        'Button3 auslöst.
        MessageBox.Show(sender.ToString & " wurde gedrückt!", "Ereignisbehandlung ergab:")

        'Schaltfläche, die gedrückt wurde, rot einfärben.

        'FEHLER! So geht es nicht, da Sender vom Typ Object ist:
        sender.Backcolor = Color.Red

        'So geht es, denn es gibt nur Schaltflächen, also
        'ist es sicher in Button zu casten:
        Dim locGedrückterButton As Button
        locGedrückterButton = DirectCast(sender, Button)

        'Jetzt klappt es mit dem Roteinfärben
        locGedrückterButton.BackColor = Color.Red
    End Sub
End Class
```

Um zu erfahren, ob es sich bei der Variablen `sender` um einen bestimmten Typ handelt, muss man in Visual Basic die wenig schöne Syntax `If TypeOf sender Is Button Then…` benutzen; statt `Button` können Sie dann natürlich auch beliebig andere Typen überprüfen.

Nähere Informationen zum Ereignis: EventArgs

`EventArgs` ist eine Klasse, die ausschließlich dafür gedacht ist, Parameter an ereignisempfangende Instanzen zu senden. Auch wenn ein Ereignis keine Parameter erforderlich macht, werden Ereignisparameter grundsätzlich mit einer `EventArgs`-Instanz übergeben – zu diesem Zweck gibt es die bereits vorhin erwähnte statische Funktion namens `EventArgs.Empty`, die ein inhaltsloses, aber dennoch instanziiertes `EventArgs`-Objekt generiert.

Wenn ein Ereignis bestimmte Parameter erforderlich macht, dann vererbt man `EventArgs` einfach in eine neue Klasse (deren Namen im Übrigen ebenfalls mit `EventArgs` enden sollte) und erweitert diese um die Eigenschaften und Konstruktoren, die erforderlich sind, um die Parameter für das Ereignis zu transportieren.

Eine Instanz von `EventArgs` sorgt in vielen Fällen aber nicht nur dafür, dass Parameter an die ereigniseinbindenden Instanzen übergeben werden. Eine ereigniseinbindende Instanz kann Parameter einer `EventArgs`-Instanz auch verändern, um dann ihrerseits zu signalisieren, dass die weitere Ereigniskette in irgendeiner Form gesteuert werden soll. Wenn Sie auf dem beschriebenen Weg eine Ereignisprozedur für das `Form_Closing`-Ereignis schreiben, können Sie beispielsweise mit `e.Cancel=true` das Schließen des Formulars auch verhindern – zum Beispiel dann, wenn sich in Ihrer Anwendung noch ungesicherte Daten befinden, das Schließen des Formulars aber im konkreten Fall durch das Herunterfahren von Windows ausgelöst wurde.

HINWEIS Wenn dies der Fall ist:

```
If e.ClosingReason=CloseReason.WindowsShutDown And UnsavedData=True Then e.Cancel=True
```

wird sogar das Herunterfahren selbst abgebrochen – eine Besonderheit von Windows, da die Sicherung der Daten vorgeht, wie Sie es auch in Verbindung mit anderen Windows-Programmen ausprobieren können, wenn Sie zum Beispiel in Word einen noch nicht gespeicherten Text geöffnet haben und dann versuchen, den Computer herunterzufahren.

Zurück zu unserem Alarm-Beispiel, das auch diese Komponente der Ereignisprogrammierung demonstriert. Wenn Sie sich die ereignisauslösende Prozedur nochmals vor Augen führen:

```
    Private Sub myTrigger_Tick(ByVal sender As Object, ByVal e As System.EventArgs) Handles _
                                                                    myTrigger.Tick
        If myAlarmzeit < DateTime.Now Then
            Dim locWeckzeitEventArgs As New AlarmEventArgs(Alarmzeit)
            OnWecken(locWeckzeitEventArgs)
            If locWeckzeitEventArgs.Neustellen Then
                Alarmzeit = locWeckzeitEventArgs.Alarmzeit
            Else
                AlarmAktiviert = False
            End If
        End If
    End Sub
```

stellen Sie fest, dass auch sie von einer benutzerdefinierten `EventArgs`-Klasse abgeleitet wurde, um für das Ereignis die Parameter `Alarmzeit` und `NeuStellen` zur Verfügung zu stellen. Diese `AlarmEventArgs`-Klasse hat im Übrigen keine andere Funktion, außer diese beiden Parameter bereitzustellen, und Sie finden Sie im Folgenden:

```
Public Class AlarmEventArgs
    Inherits EventArgs

    Private myAlarmzeit As Date
    Private myNeuStellen As Boolean

    Sub New(ByVal Alarmzeit As Date)
        myAlarmzeit = Alarmzeit
        myNeuStellen = True
    End Sub

    Sub New(ByVal Alarmzeit As Date, ByVal Neustellen As Boolean)
        myAlarmzeit = Alarmzeit
        myNeuStellen = Neustellen
    End Sub

    Public Property Alarmzeit() As Date
        Get
            Return myAlarmzeit
        End Get
        Set(ByVal value As Date)
            myAlarmzeit = value
        End Set
    End Property
```

```
    Public Property Neustellen() As Boolean
        Get
            Return myNeuStellen
        End Get
        Set(ByVal value As Boolean)
            myNeuStellen = value
        End Set
    End Property
End Class
```

Und beide Codeausschnitte im Kontakt betrachtet zeigen nun, dass Ereignisparameter quasi in beide Richtungen fließen. Eine Prozedur, die das Ereignis verarbeitet, kann durch Setzen der `Neustellen`-Eigenschaft bestimmen, ob der Wecker am nächsten Tag zur gleichen Zeit wieder schellen soll. Wenn das Ereignis ausgelöst und eine Instanz der `AlarmEventArgs`-Klasse nach »oben hochgereicht« wird, dient ein Parameter (`Alarmzeit`) also der ereigniseinbindenden Klasse und der andere (`Neustellen`) der ereignisauslösenden Klasse, den sie von der ereigniseinbindenden Klasse zurückerhält. Das Setzen dieses Parameters sehen Sie am Beispiel im Listingausschnitt, der auf Seite 518 beginnt (`Private Sub myTrigger_Tick`).

Delegaten

Wenn Sie Programme schreiben, dann werden die Anweisungen, die Sie in einer Hochsprache schreiben, irgendwann in so genannte Opcodes, in Bytebefehlsfolgen, übersetzt, die die verschiedenen Kerne Ihres Prozessors interpretieren und dann später ausführen können. Sie können sich einen Eindruck davon verschaffen, indem Sie eine Programmzeile im Editor per F9 mit einem Haltepunkt versehen, Ihr Programm starten, auf diesen Haltepunkt treffen lassen und anschließend über das Menü *Debuggen* und *Fenster* den Befehl *Disassembly* aufrufen.

```
        Dim t As Integer = 15
000000a9 C7 45 B8 0F 00 00 00 mov         dword ptr [ebp-48h],0Fh
        Dim produkt = 10 * t
000000b0 8B 45 B8             mov         eax,dword ptr [ebp-48h]
000000b3 BA 0A 00 00 00       mov         edx,0Ah
000000b8 6B C0 0A             imul        eax,eax,0Ah
```

Abbildung 16.4 Das Disassembly-Fenster zeigt, wie die einzelnen Programmzeilen für den Prozessor ausschauen: Es sind echte Bytefolgen, die die Assembleropcodes darstellen; Daten also, die vom Prozessor als Befehle verstanden werden

Abbildung 16.4 zeigt das am Beispiel, an einem Auszug aus einem Visual Basic-Beispiel, und wie sich das Ganze in Assembler gestaltet: Hier sehen Sie, wie der Wert 15 (hexadezimal 0F) als Integer angesehen (`dword`) und in die Speicherstelle geschrieben wird, die sich aus der angegebenen Adresse [*ebp-48h*] ergibt. Das entspricht unserer Wertzuweisung 15 an die Variable `t`. Die nächsten Zeilen sind ähnlich einfach zu interpretieren. Doch es geht gar nicht darum, die Sprache Assembler zu erlernen und zu verstehen.

Sie erkennen an dieser Stelle auch, was das Entscheidende ist, nämlich dass Programme und Programmteile ähnlich wie Daten an bestimmten Stellen im Hauptspeicher Ihres Computers liegen. Und wenn Programme selbst Daten sind, dann müsste es doch auch so was wie Variablen auf Programmteile geben, auf Subs oder Functions, also auf Methoden.

Das Problem dabei ist jedoch, dass Programmteile – also Methoden – auch einer Typsicherheit unterliegen müssen. Wieso, zeigt das folgende Beispiel:

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 16\Delegates\DelegatesDemo1
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
Module Module1

    Sub Main()
        Console.ReadKey()
        If Debugger.IsAttached Then
            Debugger.Break()
        End If
        Dim s = Square(12)
        Dim p = Power(2, 12)
        Console.WriteLine(s.ToString)
        Console.WriteLine(s.ToString)
    End Sub

    Function Square(ByVal var1 As Integer) As Integer
        Dim ret = var1 * var1
        Return ret
    End Function

    Function Power(ByVal var1 As Integer, ByVal var2 As Integer) As Integer
        For c As Integer = 1 To var2
            var1 *= var1
        Next
        Return var1
    End Function

End Module
```

Sie fragen sich wohl jetzt, warum das Programm mit einem Tastendruck beginnt, noch bevor sich sonst irgendwas tut. Seien Sie gespannt; wir schauen uns nämlich das Programm im Folgenden so an, wie es der Prozessor auch sieht und wirklich ausführt:

1. Starten Sie das Programm – aber unbedingt ohne Debugging. Dazu wählen Sie aus dem Menü *Debuggen* den Befehl *Starten ohne Debugging*, oder Sie drücken einfach Strg+F5.
2. Starten Sie eine zweite Instanz von Visual Studio 2010.
3. Wählen Sie aus dem Menü *Debuggen* der neuen Visual Studio-Instanz den Befehl *An den Prozess anhängen*. Daraufhin zeigt Visual Studio den Dialog aus Abbildung 16.5.

Abbildung 16.5 Mit einer zweiten Instanz von Visual Studio können Sie auch die Anwendungen debuggen, die Sie nicht direkt unterbrechen können. Selbst programmierte Steuerelemente im Designer einer anderen Visual Studio-Instanz – oder eben Anwendungen in freier Wildbahn!

4. Wählen Sie die Beispielanwendung aus, die Sie gerade gestartet haben, und klicken Sie abschließend auf die Schaltfläche *Anfügen*.
5. Wechseln Sie zurück zur Konsolenanwendung, und drücken Sie eine beliebige Taste.

In dem Moment, in dem Sie den Tastendruck auslösen, trifft das Programm auf die Anweisung `Debugger.Break` und stoppt in der zweiten Instanz von Visual Studio, zeigt dabei auch den entsprechenden Quellcode an. An dieser Stelle können Sie über das Menü *Debuggen/Fenster* und den Befehl *Disassembly* in die Assemblydarstellung wechseln und sehen nun die Befehle, die der Prozessor als Nächstes ausführen wird (Abbildung 16.6).

Abbildung 16.6 Parameter werden an Methoden übergeben, indem die entsprechenden Register des Prozessors mit den Argumenten geladen werden, die die Methode erhalten sollen

So, und jetzt wissen Sie nicht nur, wie Sie den Debugger an beliebige Anwendungen anhängen können, um diese aus Sicht des Prozessors zu untersuchen, sondern auch, wie in Assembler, also der Maschinensprache des Prozessors, das Übergeben von Parametern an Methoden funktioniert.

Und dabei fällt eines auf: Wenn wir einen Datentyp verwenden wollen, der Adressen von Unterroutinen in einer Objektvariablen speichert, dann reicht das nicht aus, wenn diese Objektvariable nur die Adresse speichert. Sie muss ebenfalls speichern, wie viele und welche Parameter an diese Routine übergeben werden; denn wenn diese Reihenfolge nicht ganz genau stimmt, dann … Sie können sich mit Blick auf Abbildung 16.6 sicherlich vorstellen, was dann passieren kann. Es gibt keine Instanz, die dafür sorgt, dass die entsprechenden Speicherstellen oder Prozessorregister mit den korrekten Werten vorbelegt werden, und die Unterroutine bedient sich dann der Register, in denen aber eben dann irgendwelche Werte drinstehen. Und das könnte schlussendlich verheerende Folgen haben.

Also brauchen wir nicht nur einen neuen Variablentyp, der eine Programmadresse speichern kann, sondern dieser Variablentyp muss auch mit definieren, welche Signatur die Routine hat, deren Adresse im Hauptspeicher später abgelegt werden muss. Und das wiederum wird durch den Variablentyp `Delegate` erreicht.

Nochmals zur Wiederholung: Die Signatur bestimmt, wie viele Parameter welchen Typs an eine Methode bei ihrem Aufruf übergeben werden. Und mit diesen Informationen lassen Sie uns im Folgenden das erste Beispiel anschauen, das von Delegaten Gebrauch macht, einer kleinen WPF-Anwendung:

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 16\Delegates\DelegatesDemo2
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Wenn Sie diese Anwendung starten, zeigt sie sich wie in Abbildung 16.7 zu sehen.

Abbildung 16.7 Durch das Setzen der CheckBoxen ändert sich der Stil, mit dem die Figur gemalt wird – intern wird das über die Verwendung einer Delegate-Variable geregelt

Sie können auf die Schaltfläche *Figuren malen* klicken, um eine aus Rechtecken zusammengesetzte Figur auf den Bildschirm zu malen. Aktivieren Sie das Kontrollkästchen *Abgerundete Ecken*, zeichnet die Beispielanwendung die Figur erneut, verwendet jedoch beim Neuzeichnen abgerundete Ecken. Intern wird das durch eine `Delegate`-Variable geregelt, wie ein Blick in das Beispielprogramm verrät:

```
'Definiert einen neuen Variablentyp für das Speichern von Methoden,
'die diese Signatur erfüllen:
Delegate Function RechteckMalDelegate(ByVal xLinks As Double, ByVal yOben As Double,
                                      ByVal xRechts As Double, ByVal yUnten As Double) As Rectangle

Class MainWindow

    'Delegate-Variable deklarieren und mit der Funktion zum Ermitteln
    'eines "kantigen" Rechtecks definieren
    Private RechteckGetter As RechteckMalDelegate = AddressOf KantigeEcken
```

Da Delegaten genau wie Enums, Strukturen und Klassen neue Typen definieren, gibt es natürlich keinen Zwang, diese codemäßig einer Klasse zuzuordnen, und genau das sehen Sie in diesem Beispiel auch: Die Definition des neuen Delegaten finden Sie in den ersten Zeilen des Beispiels. Objektvariablen, die später vom Typ `RechteckMalDelegate` definiert werden, können damit nur auf Methoden verweisen, die genau der Signatur des definierten Delegatentyps entsprechen – die Reihenfolge der Typen der Parameter, die den kompatiblen Methoden später übergeben werden, muss also genau diejenige sein, die der Delegat definiert. Bei Methoden, die Rückgabewerte zurückliefern (also durch `Function` definiert), müssen die Rückgabetypen ebenfalls übereinstimmen.

Im Beispiel haben wir zwei Methoden, die zum Delegaten signaturkompatibel sind: `AbgerundeteEcken` und `KantigeEcken`. Und in den folgenden Zeilen sehen Sie, wie diese beiden Methoden (oder besser: die *Adressen* dieser Methoden) mit dem `AddressOf`-Operator der Variablen `RechteckGetter`, die vom Typ `RechteckMalDelegate` definiert wurde, in Abhängigkeit des CheckBox-Zustands zugewiesen wird:

```
    'Klick zeichnet die Figur aus Rechtecken.
    Private Sub DrawButton_Click(ByVal sender As System.Object,
                                 ByVal e As System.Windows.RoutedEventArgs) Handles DrawButton.Click
        Neumalen()
    End Sub

    'Per CheckBox umschalten zwischen abgerundeten und normalen Ecken.
    Private Sub AbgerundeteEckenCheckBox_ToggleCheck(ByVal sender As System.Object,
                            ByVal e As System.Windows.RoutedEventArgs)
            Handles AbgerundeteEckenCheckBox.Checked, AbgerundeteEckenCheckBox.Unchecked

        If AbgerundeteEckenCheckBox.IsChecked Then
            RechteckGetter = AddressOf AbgerundeteEcken
        Else
            RechteckGetter = AddressOf KantigeEcken
        End If
        Neumalen()
    End Sub

    'Malt die Figur neu.
    Private Sub Neumalen()
        Dim s As New Rectangle()
        MalbereichCanvas.Children.Clear()

        'Wir fächern ein paar Rechtecke auf, drehen sie also beim Zeichnen...
        For c = 0 To 120 Step 3

            '...und verschieben jedes einzelne zusätzlich ein bisschen
            'nach rechts und nach unten.
            Dim x1, y1 As Double
            x1 = 10 + c * 3
            y1 = c \ 2

            'Hier ermitteln wir das Rechteck-Objekt über den Delegaten,
            'der jeweils eine andere "Adresse" enthält, je nachdem,
            'ob die CheckBox angeklickt ist oder nicht.
            Dim recToDraw = RechteckGetter(x1, y1, x1 + 300, y1 + 200)
```

Beim eigentlichen Malen der Figur ruft das Beispiel nun nicht die Methoden direkt auf, sondern jeweils die Methode, die in der `Delegate`-Variable `RechteckGetter` hinterlegt wurde – so, wie in der oben stehenden Zeile zu sehen.

```
            'Positionsbestimmung in WPF auf dem Canvas funktioniert nur
            'auf diese Weise über Attached Properties.
            Canvas.SetLeft(recToDraw, DirectCast(recToDraw.Tag, Point).X)
            Canvas.SetTop(recToDraw, DirectCast(recToDraw.Tag, Point).Y)

            'Wir drehen das Rechteck um c Grad; der Drehpunkt ist ein
            'wenig nach links und nach oben geschoben.
            recToDraw.LayoutTransform = New RotateTransform(c, -100, -50)

            'Das Hinzufügen zur Children-Auflistung des Canvas sorgt für
            'das Erscheinen auf dem Bildschirm.
            MalbereichCanvas.Children.Add(recToDraw)
        Next
    End Sub

    'Liefert ein WPF-Rechtecke-Objekt in den angegebenen Koordinaten mit normalen Ecken...
    Private Function KantigeEcken(ByVal x1 As Double, ByVal y1 As Double,
                                  ByVal x2 As Double, ByVal y2 As Double) As Rectangle
```

```
        Return New Rectangle() With {.Tag = New Point(x1, y1),
                                     .Width = x2 - x1,
                                     .Height = y2 - y1,
                                     .Stroke = New SolidColorBrush(Colors.Black)}
    End Function

    '...und diese Methode ebenfalls eines, aber mit abgerundeten Ecken.
    Private Function AbgerundeteEcken(ByVal x1 As Double, ByVal y1 As Double,
                                      ByVal x2 As Double, ByVal y2 As Double) As Rectangle

        Return New Rectangle() With {.Tag = New Point(x1, y1),
                                     .Width = x2 - x1,
                                     .Height = y2 - y1,
                                     .RadiusX = 30,
                                     .RadiusY = 30,
                                     .Stroke = New SolidColorBrush(Colors.Black)}
    End Function

End Class
```

Der Rest des Beispiels ist einfach zu verstehen – hier gibt es nur noch die beiden Methoden, die zum neuen Delegatentyp signaturkompatibel sind und deren Adressen je nach CheckBox-Zustand in der Delegatenvariablen gespeichert werden.

Delegaten an Methoden übergeben

Da Delegaten im Prinzip nichts anderes sind als Variablentypen, können sie natürlich auch an Methoden als Parameter übergeben werden. Unter welchen Umständen das zur Anwendung kommen kann, zeigt das folgende Beispiel. Schauen wir uns dazu eine weitere Version unseres bisherigen MiniAdresso-Beispiels aus Kapitel 10 an:

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 16\MiniAdressoClass V6
```

Wir bauen die Sortierroutine des ursprünglichen Beispiels dort so um, dass der Sortierkriteriumsausdruck, der bislang fest im Programm mit dem Nachnamen des Kontakts verdrahtet war, durch einen Delegaten ersetzt wird. Der Delegat wiederum erwartet zwei Parameter, nämlich die Kontakte, die miteinander verglichen werden sollen. Die im Delegat gespeicherte Methode liefert einen Rückgabewert vom Typ `Integer` zurück, mit folgendem Hintergrund: Ergibt das Funktionsergebnis 0, sind beide Kontakte gleich, bei 1 ist der erste größer und bei -1 ist der erste kleiner. Was letzten Endes verglichen wird, regelt dann eine Prozedur, auf die die Delegatenvariable zeigen soll. Die Änderungen dazu sehen zunächst einmal folgendermaßen aus:

```
    'Definiert den Typ ComparerDelegate, der zwei Parameter vom Typ Kontakt erwartet und einen
    'Integer zurückliefert. -1: erster Kontakt ist kleiner, 0: Kontakt ist gleich, 1: 1. Kontakt
    'ist größer. Wieso ein Kontakt tatsächlich größer oder kleiner ist, regelt dann die Methode,
    'deren Adresse im Delegaten gespeichert ist. Dadurch wird der Sortieralgorithmus flexibel anwendbar.
    Public Delegate Function ComparerDelegate(ByVal k1 As Kontakt, ByVal k2 As Kontakt) As Integer
```

```
'cDel ist eine Delegatenvariable vom Typ ComparerDelegate.
Sub AdressenSortieren(ByVal cDel As ComparerDelegate)

    Dim anzahlElemente As Integer = 101

    Dim aeussererZaehler As Integer
    Dim innererZaehler As Integer
    Dim delta As Integer

    Dim tempKontakt As Kontakt

    delta = 1

    'Größten Wert der Distanzfolge ermitteln
    Do
        delta = 3 * delta + 1
    Loop Until delta > anzahlElemente

    Do
        'Ist Abbruchkriterium - deswegen herunterzählen
        delta \= 3

        'Shellsort's Kernalgorithmus
        For aeussererZaehler = delta To anzahlElemente - 1
            tempKontakt = Kontakte(aeussererZaehler)

            innererZaehler = aeussererZaehler
            Do
                'Delegatenvariablen können, wie hier im Beispiel zu sehen, entweder direkt per
                'Namen wie Methoden oder mit der Invoke-Methode aufgerufen werden.
                If cDel.Invoke(tempKontakt, Kontakte(innererZaehler - delta)) = 1 OrElse _
                    cDel(tempKontakt, Kontakte(innererZaehler - delta)) = 0 Then Exit Do
                Kontakte(innererZaehler) = Kontakte(innererZaehler - delta)

                innererZaehler = innererZaehler – delta
                If (innererZaehler <= delta) Then Exit Do
            Loop
            Kontakte(innererZaehler) = tempKontakt
        Next
    Loop Until delta = 0
End Sub
```

An den zuletzt in Fettschrift formatierten Zeilen dieses Listings können Sie erkennen, dass der eigentliche Aufruf einer Methode über zwei mögliche Wege erfolgen kann, nämlich so, wie wir es bislang gelernt haben, also direkt über die Verwendung der Delegatenvariable wie eine Methode, oder indem wir die `Invoke`-Methode verwenden, der genau die Parameter übergeben werden, die normalerweise auch der Methode übergeben werden.

TIPP Meine Empfehlung ist es, die `Invoke`-Methode zu verwenden. Denn dann sehen Sie beim ersten Hinschauen bereits, ob es sich beim Aufruf um eine wirkliche Methode oder um eine Methode handelt, deren Adresse in einer Delegatenvariable gespeichert ist.

Die beiden Methoden, deren Adressen abwechselnd und je nach Bedarf in der Delegatenvariablen gespeichert werden, sorgen dafür, dass die gesamte Liste unterschiedlich sortiert wird, da sie die beiden ihnen übergebenen Kontakteinstanzen auf unterschiedliche Weise vergleichen:

```
    Function KontaktNachnamenVergleich(ByVal k1 As Kontakt, ByVal k2 As Kontakt) As Integer
        Return String.Compare(k1.Nachname, k2.Nachname)
    End Function

    Function KontaktOrtVergleich(ByVal k1 As Kontakt, ByVal k2 As Kontakt) As Integer
        Return String.Compare(k1.Ort, k2.Ort)
    End Function
```

Und jetzt schauen wir uns an, wie wir uns diesen Umbau für eine flexiblere Nutzung der Sortierroutine zu Nutze machen können und wo eigentlich der Delegatenvariable eine Methode zugeordnet wird:

```
    Sub Main()
.
.
.
        'Zufallsadressen generieren
        Console.Write("Zufallsadressen werden generiert ... ")
        ZufallsAdressenGenerieren()
        Console.WriteLine("fertig!")

        'Die ersten 10 Zufallsadressen ausgeben
        AdressenAusgeben(0, 10)

        'Die Adressen nach Nachnamen sortieren
        Console.WriteLine()
        Console.Write("Adressen werden nach Nachnamen sortiert ... ")
        Dim compDelegate As ComparerDelegate = AddressOf KontaktNachnamenVergleich
        AdressenSortieren(compDelegate)
        Console.WriteLine("fertig!")
        Console.WriteLine()

        'Die ersten 10 Zufallsadressen ausgeben
        AdressenAusgeben(0, 10)

        'Die Adressen nach Ortsnamen sortieren
        Console.WriteLine()
        Console.Write("Adressen werden nach Ortsnamen sortiert ... ")
        compDelegate = AddressOf KontaktOrtVergleich
        AdressenSortieren(compDelegate)
        Console.WriteLine("fertig!")
        Console.WriteLine()
.
.
.
    End Sub
```

Hier werden jetzt an zwei Stellen der Sortierroutine über den Delegatenparameter die Adressen der Vergleichsmethoden übergeben, mit denen dann die eigentliche Sortierung durchgeführt wird. Was passiert jetzt genau, wenn das Programm gestartet wird? Die folgende Schritt-für-Schritt-Erklärung macht es deutlich:

1. Nachdem die Zufallsadressen erstellt wurden, definiert die Anwendung die Delegatenvariable `compDelegate` und weist dieser gleichzeitig die Adresse der `KontaktNachnamenVergleich`-Methode zu.
2. Sie ruft die Methode `AdressenSortieren` auf und übergibt ihr gleichzeitig die Delegatenvariable.
3. `AdressenSortieren` wiederum führt `Invoke` auf die Delegatenvariable aus und ruft dabei für den Elementvergleich die Methode `KontaktNachnamenVergleich` auf. Die Kontaktliste wird damit nach Nachnamen sortiert.
4. Wieder zurück im Hauptmodul wird der Delegatenvariable `compDelegate` nach der Sortierung die Adresse der `KontaktOrtVergleich`-Methode zugewiesen.
5. Die Anwendung ruft abermals die Methode `AdressenSortieren` auf und übergibt ihr gleichzeitig die Delegatenvariable, die jetzt allerdings auf eine andere Methode als zuvor zeigt.
6. `AdressenSortieren` führt `Invoke` auf die Delegatenvariable aus und ruft aber dieses Mal dabei für den Elementvergleich die Methode `KontaktOrtVergleich` auf. Die Kontaktliste wird damit beim zweiten Durchlauf nach dem Ortsnamen sortiert.

Lambda-Ausdrücke

Lambda-Ausdrücke vereinen quasi das Beste aus Delegaten und Methoden. Lambdas stammen ursprünglich aus der funktionellen Programmierung und verknüpfen quasi eine Funktion mit einer Objektvariablen, und das sieht im einfachsten Fall wie im folgenden Listing aus.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 16\Lambdas\SimpleLambdaDemo
```

```
Sub Main()

    Dim expressionLambda = Function(square As Double) square * square
    Dim expressionSub = Sub(ValueToPrint As Double) Console.WriteLine(ValueToPrint.ToString)

    '"Druckt" 144 aus. Erst wird 12*12 ausgeführt, dann das Ergebnis ausgegeben.
    expressionSub(expressionLambda(12))

    'Auf Taste warten, damit wir überhaupt was zu sehen bekommen.
    Console.ReadKey()

End Sub
```

In diesem Beispiel erleben Sie die so genannten Ausdrucks-Lambdas (Expression Lambdas) in Aktion. Sie sehen hier bekannte Schlüsselwörter – `Sub` und `Function` – in einer ungewohnten Umgebung, denn das, was als Erstes auffällt ist, dass zwar Übergaben von Parametern definiert werden, aber der Name der Methode fehlt. Und das ist der Grund, weswegen man bei Lambdas auch von anonymen Methoden sprechen kann.[3]

Man muss wissen und sich daran gewöhnen, wie man eine solche Zeile zu lesen hat. Denn sowohl `expressionsLambda` als auch `expressionSub` sind Delegatenvariablen, so wie wir sie im vorherigen Abschnitt kennen gelernt haben. Beide Typen speichern also die Adressen auf die einzeiligen Methoden, die hinter ihnen definiert sind. Der Unterschied ist lediglich: Der `AddressOf`-Operator kommt nicht zur Anwendung, weil die Methoden, deren Adressen in den Delegatenvariablen gespeichert werden, keinen Namen haben. `Function` und `Sub` ohne Namen, könnte man sagen, geben also direkt die interne Speicheradresse der Methode an die Variable zurück, ohne dass Sie diese jemals zu Gesicht bekommen würden oder könnten.

Verwenden Sie also im Anschluss daran, wie im Beispiel zu sehen, `expressionLambda` wie eine Methode, wird der Code ausgeführt, der hinter `Function(…)` im obigen Beispiel zu sehen ist. Bringen Sie `expressionSub` zur Anwendung, wird die `Console.WriteLine`-Methode ausgeführt. Und dass diese beiden Ausdrucks-Lambdas auch geschachtelt eingesetzt werden können, zeigt das Beispiel obendrein.

Einzeilige Ausdrucks- und mehrzeilige Anweisungs-Lambdas

...NET unterscheidet übrigens zwischen so genannten Ausdrucks-Lambdas und Anweisungs-Lambdas (*Statement Lambdas*). Ausdrucks-Lambdas zeichnen sich dadurch aus, dass sie ausschließlich einzeilig sein können – wie oben im Beispiel zu sehen. Das hat den Grund, dass aus mehreren Lambda-Ausdrücken so genannte Ausdrucksbäume dynamisch erstellt werden können – besser bekannt unter dem angelsächsischen Namen *Expression Trees* – die zur Laufzeit kompiliert werden können. Bei mehrzeiligen Lambda-Ausdrücken wäre diese Anforderung zu kompliziert.

Mehrzeilige Anweisungs-Lambdas kommen den anonymen Methoden, die quasi nur über ihre Adressen zu erreichen sind, die in Delegatenvariablen gespeichert werden, tatsächlich am nächsten.

Visual Basic 2008 führte einzeilige Lambda-Funktionen ein, wie sie im obigen Abschnitt beschrieben wurden (allerdings nur auf `Function`-Basis – Ausdrucks-Lambdas ohne Rückgabewerte, die durch `Sub` definiert werden, sind in Visual Basic 2008 noch nicht möglich!), und mit ihnen eine Erleichterung im Umgang mit Delegaten.

Wie flexibel die Programmierung mit mehrzeiligen Lambdas sein kann, zeigt ein Beispiel, das eine Methode aus einer Auflistungsklasse verwendet und dabei etwas im Stoff vorgreift: Die `ForEach`-Methode der generischen Liste (`List(Of )`) zeigt einfach und plausibel den Umgang mit Lambdas. Geht es darum, möglichst schnell die Elemente einer generischen Liste zu verarbeiten, eignet sie sich im Zusammenhang mit einer Lambda-Funktion besonders, da sie – im Gegensatz zum herkömmlichen `For Each`-Konstrukt – intern nicht den Umweg über `GetEnumerator` zum Aufzählen aller Elemente der Liste nehmen muss, sondern direkt auf die Listenelemente zugreifen kann.

---

[3] Tatsächlich unterscheidet C# anonyme Methoden und Lambdas per Definition; der C#-Compiler macht aber intern dasselbe daraus.

Um alle Elemente einer Liste beispielsweise im Debug-Fenster mithilfe einer Lambda-Funktion auszugeben, kann man sich folgenden Codes bedienen:

```
Sub LambdaDemo()
    Dim namenListe As New List(Of String) From {"Peter", "Klaus", "Tanja", "Sonja"}
    'Anstelle eines Delegaten:
    Dim writeToDebugDelegate As New Action(Of String)(AddressOf WriteToDebug)
    namenListe.ForEach(writeToDebugDelegate)

    'Schreiben wir das hier direkt:
    namenListe.ForEach(Function(item) WriteToDebug(item))
End Sub

Shared Function WriteToDebug(ByVal text As String) As Boolean
    Debug.Print(text)
    Return True
End Function
```

`Action(Of String)` ist in diesem Fall der Delegattyp, den die `ForEach`-Methode als Delegatenvariable erwartet. Und hier sind Lambdas eine Erleichterung, weil eine Lambda-Funktion der `ForEach`-Methode auch direkt als Parameter übergeben werden kann.

Bis Visual Basic 2008 war man allerdings, wie schon gesagt, bei Lambdas auf Methoden mit Rückgabeparameter beschränkt. Aus diesem Grund gab es bis zu dieser Visual Basic-Version auch nicht die Möglichkeit, zum Beispiel ein Konstrukt wie `Debug.Print` direkt in der Lambda-Funktion anzugeben, da `Debug.Print` eben eine `Sub`, also eine Methode ohne Rückgabewert darstellt. Ab Visual Basic 2010 ist das anders. Hier ist dieses Konstrukt

```
namenListe.ForEach(Sub(item) Debug.Print(item))
```

ebenso erlaubt wie mehrzeilige Lambdas:

```
    namenListe.ForEach(Sub(item)
                           'Hier können jetzt mehrere Codezeilen platziert werden
                           Debug.Print(item)
                       End Sub)
```

Der Einsatz von Lambdas wird Visual Basic-Programme in Zukunft deutlich anders ausschauen lassen. Lambdas haben hier das Potenzial, einerseits dem Entwickler Arbeit abzunehmen, weil er für viele Zwecke nicht mehr den Umweg über Delegaten nehmen muss, und das spart Codezeilen und Programmierzeit. Andererseits wirkt der Code auch aufgeräumter, da Code an der Stelle steht, wo er passiert.

Das hat ab .NET Framework 4.0 umso mehr Bedeutung, da Delegaten hier verstärkt zum Einsatz kommen, wenn die Codeausführung parallelisiert, also automatisch auf mehrere Prozessoren oder Prozessorkerne verteilt werden muss, wie das folgende Beispiel zeigt:

```
    Parallel.ForEach(namenListe, Sub(item)
                                     'Elemente parallel verarbeiten.
                                     'ACHTUNG: Das passiert durch die Gleichzeitigkeit
                                     'nicht mehr sequenziell, sondern durcheinander!
                                     Debug.Print(item)
                                 End Sub)
```

Mehr zum Thema Parallelisierung finden Sie übrigens in Kapitel 29 im Teil F. In diesem Beispiel muss übrigens der Namespace `System.Threading.Tasks` importiert werden.

In der Praxis kann der Umgang mit Lambdas dann folgendermaßen ausschauen. In dem folgenden Beispiel geht es um die Validierung einer Eingabe in einem Steuerelement, die, wenn sie fehlschlägt, eine Ausnahme zurückliefert, aufgrund derer dann ein ToolTip mit dem entsprechenden Fehlermeldungstext direkt neben dem Steuerelement ausgegeben werden soll, das die Fehlermeldung erzeugt hat:

```
Protected Overrides Sub OnValidating(ByVal e As System.ComponentModel.CancelEventArgs)

    MyBase.OnValidating(e)
    If e.Cancel Then
        Return
    End If

    'ValidateInput überprüft, ob die Eingabe i.O. geht, und liefert
    'bei einem Fehler eine Exception zurück, die letzten Endes
    'auch den Fehlertext ergibt.
    Dim valRes = ValidateInput(myValueControl.Value.ToString)
    If valRes IsNot Nothing Then
        e.Cancel = True

        'Bei Bedarf kann hier ein Warnton ausgegeben werden.
        If myBeepOnFailedValidation Then
            Beep()
        End If

        'Die Fehlermeldung soll in einem Tooltip neben
        'dem Steuerelement für eine gewisse Zeit zu sehen sein.
        Dim tt As New ToolTip()

        'Steuerelement malen wir selbst.
        tt.OwnerDraw = True

        'Soll sich Ausblenden.
        tt.UseFading = True
        tt.UseAnimation = True

        'Dank mehrzeiligen Lambdas können wir den Drawing-Handler-Code
        'direkt dem Draw-Ereignis übergeben.
        AddHandler tt.Draw, Sub(sender As Object, de As DrawToolTipEventArgs)
                                Dim pen As New Pen(Brushes.Black, 1)
                                'Point-Array clonen und Koordinaten anpassen

                                Dim path As New GraphicsPath()
                                Dim p1, p2 As PointF
                                p1 = PointF.Empty

                                'Die Koordinaten des Rahmens des
                                'Tooltipps stehen in einer List(Of PointF),
                                'die wir mit der ForEach-Methode und wieder
                                'mit einem mehrzeiligen Lambda durchiterieren.
                                myToolTipShapeCoordinates.ForEach(
                                    Sub(item)
```

```
                                        'Koordinaten des Tooltips an
                                        'die des Steuerelements anpassen:
                                        Dim x = item.X
                                        Dim y = item.Y
                                        If x = -1 Then
                                            x = de.Bounds.X + de.Bounds.Width - 1
                                        Else
                                            x += de.Bounds.X
                                        End If

                                        If y = -1 Then
                                            y = de.Bounds.Y + de.Bounds.Height - 1
                                        Else
                                            y += de.Bounds.Y
                                        End If
                                        p1 = p2
                                        p2 = New PointF(x, y)
                                        If p1 <> Point.Empty Then
                                            path.AddLine(p1, p2)
                                        End If
                                    End Sub)

                                'Zum Malen auf AntiAlias stellen: sieht "weicher" aus.
                                de.Graphics.SmoothingMode = SmoothingMode.AntiAlias
                                path.CloseFigure()

                                'Hintergrund, Umrandung und Text des Tooltips malen:
                                de.Graphics.FillPath(New SolidBrush(Color.AliceBlue), path)
                                de.Graphics.DrawPath(Pens.Black, path)
                        de.Graphics.DrawString(de.ToolTipText, New Font(FontFamily.GenericSansSerif,
                                                                        8, FontStyle.Regular),
                                                                           Brushes.Black,
                                                        New RectangleF(de.Bounds.X + 15, de.Bounds.Y + 5,
                                                 de.Bounds.Width - 15, de.Bounds.Height - 5))
                                End Sub

            'Tooltip ausgeben. Das erst triggert den obenstehenden Code,
            'da jetzt erst das Draw-Ereignis des Tooltips ausgelöst wird,
            'dem dieser Code mit AddHandler zugeordnet wurde.
            tt.Show("Die Eingabe hatte einen Fehler:" & vbNewLine & vbNewLine &
                    valRes.Message, Me, Me.Width - 10, Me.Height \ 2, Me.ExceptionBalloonDuration)
        End If
    End Sub
```

Dynamisches Anbinden von Ereignissen mit AddHandler

Das Einbinden von Ereignissen versagt bei `WithEvents`, wenn Objekte, die Ereignisse zur Verfügung stellen, nur auf Prozedurebene (also lokal) deklariert werden oder Bestandteil eines Arrays oder einer Auflistung sind. Bei solchen Konstellationen besteht die Möglichkeit, Ereignisse dynamisch und zur Laufzeit mithilfe von `AddHandler` zu verdrahten.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 16\Alarm02
```

Das Beispiel erweitert die einfache Alarmwecker-Anwendung, indem es nicht nur eine einzige Weckzeit, sondern gleich das Hinterlegen einer ganzen Reihe von Terminen erlaubt. Wenn Sie dieses Beispiel starten, sehen Sie ein Dialog, wie ihn auch Abbildung 16.8 darstellt.

Abbildung 16.8 Mit der modifizierten »Wecker«-Applikation können Sie gleich mehrere Weckzeiten erfassen ...

Mit dieser Anwendung erfassen Sie nicht nur eine Weckzeit – Sie können gleich mehrere angeben. Außerdem erlaubt diese Version des Beispiels auch die Eingabe eines Alarmtexts, der im Falle des Eintretens des Ereignisses im Ziffernblatt der Uhr eingeblendet wird – wie Abbildung 16.9 zeigt.

Intern führt diese Version dabei eine Klasse zur Verwaltung einer Liste mit Objektinstanzen ähnlich der `DynamicList`, die Sie schon aus anderen Beispielen vorheriger Kapitel kennen. Doch dieses Mal verwenden wir eine, die bereits fest in .NET Framework eingebaut ist – eine generische Auflistung namens `Collection`.

Die Schwierigkeit hierbei ist, dass wir nun nicht mehr eine globale Variable innerhalb der abgeleiteten `Collection`-Klasse verwenden können, die dann mit `WithEvents` zu deklarieren wäre, denn es geht ja nunmehr um eine *Liste* mit `Alarmgeber`-Instanzen.

Abbildung 16.9 ... und ein zusätzlicher Meldungstext wird im Alarmfall im Ziffernblatt eingeblendet

Wir müssen also einen anderen Weg gehen, und der sieht folgendermaßen aus: Wann immer der Benutzer einen neuen Wecktermin eingibt und somit eine neue `Alarmgeber`-Instanz der Liste mit `Add` hinzugefügt wird, müssen wir das in der `Collection`-Ableitung abfangen und einen Ereignisauslöser, den wir in dieser Klasse neu definieren, zur Laufzeit mit dem `Alarm`-Ereignis der `Alarmgeber`-Instanz verdrahten.

Umgekehrt müssen wir dafür sorgen, dass beim Löschen eines Alarmgebers aus der Liste zuvor diese Ereignisverknüpfung wieder aufgehoben wird. Und alle Ereignisverknüpfungen der vorhandenen `Alarmgeber`-Instanzen in der Liste müssen wir dann löschen, wenn sie mit `Clear` komplett gelöscht wird. Und das sieht dann codemäßig wie folgt aus:

```
Imports System.Collections.ObjectModel

''' <summary>
''' Verwaltet eine Liste mit Alarmgeber-Objekten, und löst ein Ereignis aus,
''' wenn eines der Alarmgeber-Objekte das für erforderlich hält.
''' </summary>
''' <remarks></remarks>
Public Class Terminliste
    Inherits Collection(Of Alarmgeber)

    ''' <summary>
    ''' Wird ausgelöst, wenn eines der dieser Instanz hinzugefügten
    ''' Alarmgeber-Objekte seinerseits ein Alarm-Ereignis auslöst.
    ''' </summary>
    ''' <param name="sender">Referenz auf das Alarmgeber-Objekt, das das Ereignis ausgelöst hat.</param>
    ''' <param name="e">AlarmEventArgs-Instanz, die Parameter zum Ereignis enthalten.</param>
    ''' <remarks></remarks>
    Public Event Alarm(ByVal sender As Object, ByVal e As AlarmEventArgs)

    'Enthält Nothing oder das Datum des als nächstes anstehenden Termins.
    Private myNächsterTermin As Nullable(Of Date)
```

```
'Wird beispielsweise durch Add oder Insert der Collection-Klasse ausgelöst.
'Überschrieben, da das Alarm-Ereignis des Objekts mit der
'AlarmHandler-Prozedur verknüpft werden muss.
Protected Overrides Sub InsertItem(ByVal index As Integer, ByVal item As Alarmgeber)

    'Das Alarm-Ereignis des Objekts dynamisch mit der Prozedur AlarmHandler verbinden.
    AddHandler item.Alarm, AddressOf AlarmHandler

    'Die Basisprozedur machen lassen, was sie machen muss
    '(nämlich das Element an die richtige Stelle setzen).
    MyBase.InsertItem(index, item)

    'Liste hat sich geändert - der nächste Termin könnte ein anderer werden!
    AktualisiereNächsteTerminEigenschaft()
End Sub

'Wird durch Zuweisung eines Elements über die Item-Eigenschaft aufgerufen.
'Überschrieben, da das Alarm-Ereignis des Objekts mit der
'AlarmHandler-Prozedur verknüpft werden muss.
Protected Overrides Sub SetItem(ByVal index As Integer, ByVal item As Alarmgeber)

    'Das alte Element an dieser Stelle lösen:
    RemoveHandler Me(index).Alarm, AddressOf AlarmHandler

    'Das neue Element verknüpfen
    AddHandler item.Alarm, AddressOf AlarmHandler

    'Die Basisprozedur machen lassen, was sie machen muss
    '(nämlich das Element an die richtige Stelle setzen).
    MyBase.SetItem(index, item)
End Sub

'Wird beispielsweise durch Remove oder RemoveAt der Collection-Klasse aufgerufen.
'Überschrieben, um das verknüpfte Ereignis wieder mit AlarmHandler zu lösen.
Protected Overrides Sub RemoveItem(ByVal index As Integer)

    'Das Alarm-Ereignis des Objekts dynamisch von der Prozedur AlarmHandler lösen.
    RemoveHandler Me(index).Alarm, AddressOf AlarmHandler

    'Die Basisprozedur machen lassen, was sie machen muss
    '(nämlich das Element aus der Liste löschen).
    MyBase.RemoveItem(index)

    'Liste hat sich geändert - der nächste Termin könnte ein anderer werden!
    AktualisiereNächsteTerminEigenschaft()
End Sub

'Beim Löschen aller Elemente werden die Ereignisse aller Objekte gelöst.
Protected Overrides Sub ClearItems()

    'Alle Ereignisse lösen.
    For Each locItem As Alarmgeber In Me
        RemoveHandler locItem.Alarm, AddressOf AlarmHandler
    Next

    'Basisroutine aufrufen.
    MyBase.ClearItems()
```

```
        'Gibt keinen "nächsten Termin" mehr.
        myNächsterTermin = Nothing
    End Sub

    ''' <summary>
    ''' Löst ein Ereignis aus, sobald diese Routine ihrerseits als
    ''' Ereignisbehandlungsroutine eines der Elemente in dieser Collection in Aktion tritt.
    ''' </summary>
    ''' <param name="sender"></param>
    ''' <param name="e"></param>
    ''' <remarks></remarks>
    Private Sub AlarmHandler(ByVal sender As Object, ByVal e As AlarmEventArgs)
        RaiseEvent Alarm(sender, e)
    End Sub

    ''' <summary>
    ''' Sucht den als nächstes anliegenden Termin in der Elementeliste.
    ''' </summary>
    ''' <remarks></remarks>
    Private Sub AktualisiereNächsteTerminEigenschaft()

        'Keine Elemente vorhanden...
        If Me.Count = 0 Then

            '...dann gibt es keinen nächsten Termin
            myNächsterTermin = Nothing
        Else

            'Alle Elemente durchsuchen, welches Element "jünger"
            'als alle anderen ist.
            myNächsterTermin = Me(0).Alarmzeit
            For Each locItem As Alarmgeber In Me
                If locItem.Alarmzeit < myNächsterTermin.Value Then
                    myNächsterTermin = locItem.Alarmzeit
                End If
            Next
        End If
    End Sub

    ''' <summary>
    ''' Liefert den nächsten anstehenden Termin oder Nothing zurück.
    ''' </summary>
    ''' <value></value>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public ReadOnly Property NächsterTermin() As Nullable(Of Date)
        Get

            'Nur Wert zurückliefern. Die Suche nach dem jüngsten
            'Datum wurde schon bei jeder Listenänderung durchgeführt.
            Return myNächsterTermin.Value
        End Get
    End Property
End Class
```

Lassen Sie mich zum besseren Verständnis zunächst ein paar Worte zur generischen `Collection` verlieren, die übrigens über den Namespace `System.Collections.ObjectModel` und nicht – wie man vielleicht annehmen könnte – über `System.Collections.Generics` erreichbar ist.[4] Sie tritt an die Stelle unserer bislang verwendeten `DynamicList`. Durch das Vererben dieser Klasse in eine neue Klasse `Terminliste` und das anschließende Überschreiben der Methoden `InsertItem`, `RemoveItem` und `ClearItems` können wir auf das Hinzufügen, das Löschen eines Elements oder das Löschen aller Elemente dennoch genau so Einfluss nehmen, als hätte wir sie selbst entwickelt. Dabei spielt es keine Rolle, ob beispielsweise das Einfügen neuer Elemente etwa durch `Add` oder `Insert` erfolgt – `Collection` sorgt dafür, dass in diesen Fällen `InsertItem` aufgerufen wird und wir so über die Vorgänge der Listenveränderung informiert werden. Mehr zum Thema Auflistungen (*Collections*) erfahren Sie übrigens in Kapitel 20.

Wenn nun ein neues `Alarmgeber`-Objekt der Liste hinzugefügt wurde, ist es wichtig, dass wir dessen Ereignis mitbekommen, damit wir, wenn der Ereignisfall dieses `Alarmgeber`-Objekts eintritt, wiederum ein Ereignis auslösen können. `WithEvents` scheidet aber, wie schon anfangs erwähnt, in diesem Fall aus, da wir es mit einer lokalen Objektvariable zu tun haben, die wir erst dann »kennen lernen«, wenn sie uns durch `InsertItem` zur Verfügung gestellt wird.

Aus diesem Grund verknüpfen wir das Ereignis dynamisch mit `AddHandler` zur Laufzeit – der erste in Fettschrift gesetzte Block des vorherigen Listingauszugs zeigt, wie das funktioniert. `AddHandler` nimmt zwei Parameter: zum einen das Objekt und dessen Ereignis (in diesem Fall `item.Alarm`), zum anderen einen Delegaten (wie im letzten Abschnitt beschrieben). Wird später das `Alarm`-Ereignis für das entsprechende `Alarmgeber`-Objekt ausgelöst, das wir gerade im Begriff sind zur Liste hinzuzufügen, dann behandelt die Prozedur dieses Ereignis, die als zweiter Parameter von `AddHandler` als Delegat übergeben wurde.

Im umgekehrten Fall müssen wir dafür sorgen, das Ereignis wieder von dieser Prozedur zu lösen, wenn das `Alarmgeber`-Element aus der Liste entfernt wird. Dazu verwenden wir das Schlüsselwort `RemoveHandler`, das äquivalent zu `AddHandler` funktioniert.

TIPP Seit Visual Basic 2010 können Sie natürlich beispielsweise auch direkt in `AddHandler` Anweisungs-Lambdas nutzen. Um beispielsweise dynamisch das `Click`-Ereignis eines Windows Forms-Buttons auszuwerten, wäre der folgende Code durchaus denkbar:

```
Sub EineMethode()
    Dim aButton As New Button

    AddHandler aButton.Click, Sub(sender, e)
                                  MessageBox.Show(sender.ToString & " wurde angeklickt!")
                              End Sub


End Sub
```

Achten Sie aber darauf, dass Sie diesen Handler *nicht* mehr mit `RemoveHandler` (siehe unten) entfernen können, falls das notwendig wäre. In diesem Fall müssten Sie dennoch auf eine Delegatenvariable zurückgreifen, die Sie zuvor mit dem Code verknüpfen:

[4] Wieso das so ist, können Sie übrigens im Blog von Krzysztof Cwalina nachlesen, den Sie unter *http://tinyurl.com/3a7992z* finden.

```
        Dim aButton As New Button

        Dim handler As EventHandler = Sub(sender, e)
                                          MessageBox.Show(sender.ToString & " wurde angeklickt!")
                                      End Sub

        AddHandler aButton.Click, handler

        'Jetzt kann man ihn auch wieder entfernen:
        RemoveHandler aButton.Click, handler
```

Und die gleiche »Umpolerei« passiert, wenn ein Element der Terminliste durch Zuweisung etwa wie

```
Terminlisteninstanz.Item(x)=NeuesAlarmgeberElement
```

geändert wird. In diesem Fall trennen wir das Ereignis des bisherigen Elements mit `RemoveHandler` und verknüpfen das neu zugewiesene mit `AddHandler`. Im Ergebnis erreichen wir damit, dass unabhängig davon, welches `Alarmgeber`-Objekt der Terminliste ein `Alarm`-Ereignis auslöst, immer die Prozedur `AlarmHandler` aufgerufen wird, die ihrerseits dann ein Ereignis auslöst, das anschließend durch das Formular `frmMain` verarbeitet werden kann.

Damit das Hauptprogramm, das ja eine Instanz vom Typ `Terminliste` zur Speicherung der Termine einbindet, es mit dem Einzeichnen des jeweils nächsten Termins möglichst einfach hat, existiert eine Prozedur, die den jeweils nächsten Termin findet (`AktualisiereNächsteTerminEigenschaft`), in einer Feldvariablen (`myNächsterTermin`) ablegt und über eine Eigenschaft (`NächsterTermin`) dem Hauptprogramm zur Verfügung stellt. Das Aktualisieren dieser Variablen findet immer dann statt, wenn sich irgendetwas an der Terminliste geändert hat.

Für diese Feldvariable kommt dabei übrigens der generische Wertetyp `Nullable(Of )` zum Einsatz. Wie schon zu erfahren war, erlaubt es dieser, aus jedem Wertetyp einen *null-baren* Typ zu machen, der nicht nur seinen eigentlichen Wert, sondern auch `Nothing` speichern kann (was Wertetypen, wie Sie wissen, standardmäßig nicht können). Man kann dann mit der Eigenschaft `HasValue` prüfen, ob die `Nullable`-Instanz einen Wert hat (oder eben `Nothing` ist) und diesen mit dessen `Value`-Eigenschaft ermitteln. Da unsere Terminliste natürlich auch leer sein kann, kommen uns die Eigenschaften der generischen `Nullable`-Klasse sehr gelegen, da wir für den jeweils nächsten Termin lediglich ein Datum oder eben `Nothing` speichern müssen, falls die Liste leer ist. Und genau das wäre mit einer Objektvariablen nur vom Typ `Date` eben nicht möglich.

Die Verarbeitung der Terminliste innerhalb des Hauptprogramms, also in `frmMain`, sieht dann folgendermaßen aus:

```
Public Class frmMain

    Private WithEvents myTimer As Timer
    Private WithEvents myTerminliste As Terminliste

    'Die Hintergrundfarbe der Uhr, die bei
    'anhaltendem Alarm wechselt.
    Private myAktuelleFarbe As Color
```

```
'Alarmdauer in 500ms-Schritten (=25 Sekunden).
Private myAlarmDauer As Integer = 50

'Zähler für die Dauer des Restalarms.
Private myAlarmDownCounter As Integer

'True: Alarm ist gerade aktiv --> die Uhr blinkt.
Private myAlarmStatus As Boolean

' Ist dieser String nicht Nothing, wird eine Textmeldung in der Uhr
'angezeigt, ansonsten nicht.
Private myLetzteAlarmmeldung As String

Sub New()

    ' Dieser Aufruf ist für den Windows Form-Designer erforderlich.
    InitializeComponent()

    'Diesen Timer benötigen wir zum Darstellen der Uhr
    'und zum Blinken der Uhr bei anhaltendem Alarm
    myTimer = New Timer()
    myTimer.Interval = 500
    myTimer.Start()

    'Standardhintergrundfarbe der Uhr ist weiß.
    myAktuelleFarbe = Color.White

    'Terminliste ist zunächst noch leer.
    myTerminliste = New Terminliste
End Sub
```

Bis zu diesem Abschnitt findet das Vorgeplänkel statt. Benötigte Feldvariablen werden deklariert, und `Sub New` sorgt für die korrekte Initialisierung aller Feldvariablen (sowie mit dem Aufruf von `InitializeComponent` auch für die Ausstattung des Formulars mit allen notwendigen Steuerelementen). Besonders wichtig ist hier natürlich das `Timer`-Objekt `myTimer`, das sich um die regelmäßige Aktualisierung der Uhrdarstellung kümmert. Dieser Timer läuft alle 500 ms ab und leitet dann in der `myTimer_Tick`-Ereignisbehandlung mit `Invalidate` ein Neuzeichnen des `PictureBox`-Inhaltes ein.

Im Unterschied zur vorherigen Version des Programms verwenden wir an dieser Stelle nicht nur ein einfaches `Alarmgeber`-Objekt, sondern die neu implementierte Liste `myTerminliste`, die mit `WithEvents` deklariert wurde, damit uns das `Alarm`-Ereignis erreicht, sobald eines der in ihr enthaltenen `Alarm`-Objekte seinerseits ein `Alarm`-Ereignis auslöst. Dieses Ereignis wird im folgenden Codeteil behandelt:

```
'Ereignisbehandlungsroutine, die ausgelöst wird,
'wenn der Alarmgeber Alarm signalisiert, weil eine
'bestimmte Uhrzeit erreicht wurde.
Private Sub myAlarmgeber Alarm(ByVal Sender As Object, ByVal e As AlarmEventArgs)
     Handles myTerminliste.Alarm
    'Wecker schellt!
    myAlarmStatus = True
    'Und zwar so lange.
    myAlarmDownCounter = myAlarmDauer
    'Abschalten sollten wir das Schellen können.
    btnAlarmAusschalten.Enabled = True
```

```
        'Hochgereichten Meldungstext setzen
        myLetzteAlarmmeldung = e.AlarmText

        'Aus Liste löschen
        lstTermine.Items.Remove(Sender)

        'Aus Terminliste löschen
        myTerminliste.Remove(DirectCast(Sender, Alarmgeber))
    End Sub
```

Die `AlarmEventArgs` – Sie sehen es an dieser Stelle – haben ebenfalls eine kleine Überarbeitung erfahren: Sie liefern den Meldungstext direkt mit, sodass dieser dann anschließend in der Mitte des Ziffernblattes angezeigt werden kann.

Wenn nun eines der `Alarmgeber`-Elemente ein `Alarm`-Ereignis ausgelöst hat, erfahren wir es über den Umweg `Terminliste` an dieser Stelle. Wir sorgen dann dafür, dass die »Weckphase« in der Darstellung eingeleitet wird und im Übrigen auch dafür, dass man den zum Termin gehörigen Meldungstext im Ziffernblatt der Uhr sieht (durch Setzen von `myLetzteAlarmmeldung = e.AlarmText` – dieser gesetzte Meldungstext wird dann beim nächsten Neuzeichnen der Uhr berücksichtigt).

```
.
.
.
    End Sub

    'Wird aufgerufen, wenn der Anwender die Hinzufügen-Schaltfläche betätigt hat
    Private Sub btnHinzufügen_Click(ByVal sender As System.Object, _
                    ByVal e As System.EventArgs) Handles btnHinzufügen.Click

        Dim locAlarmzeit As Date

        'Zeit und Termingrund aus den TextBox-Steuerelementen holen.
        If Date.TryParse(mtbAlarmzeit.Text, locAlarmzeit) And _
            Not String.IsNullOrEmpty(txtGrund.Text) Then

            'Neues Alarmgeber-Objekt instanziieren, das wir anschließend
            'direkt zur Listbox...
            Dim locAlarm As New Alarmgeber(locAlarmzeit, txtGrund.Text, True)
            lstTermine.Items.Add(locAlarm)

            '...sowie zur Terminliste hinzufügen.
            myTerminliste.Add(locAlarm)

            'Uhr neuzeichnen, damit die Weckzeit des nächsten Termins als
            'roter Strich ins Ziffernblatt kommt!
            picWecker.Invalidate()
        Else

            'Ups! Solch eine Uhrzeit gibt es nicht - TryParse
            'ist fehlgeschlagen.
            MessageBox.Show("Bitte überprüfen Sie die Eingabe auf Fehler!")
        End If
    End Sub
```

Neu und damit erwähnenswert ist die oben stehende Routine, die dafür sorgt, dass ein neues `Alarmgeber`-Objekt erstellt und der Liste hinzugefügt wird, wenn der Benutzer die Daten für einen neuen Termin erfasst und anschließend auf *Hinzufügen* klickt.

Hier wird übrigens deutlich, dass das Programm mit einem kleinen Manko zu kämpfen hat, denn die Referenzen auf die eigentlichen Termine – die `Alarmgeber`-Objekte – werden im Grunde genommen in zwei Listen gespeichert: in der `Collection(Of Alarmgeber)`-Auflistung und in der internen Liste des `ListBox`-Steuerelements `lstTermine`. Die `ListBox`-Liste benötigen wir aber einerseits für die Listendarstellung der Termine und die `Collection(Of Alarmgeber)`-Auflistung andererseits dafür, dass wir mitbekommen, wenn eines der in ihr enthaltenen `Alarmgeber`-Objekte ein Ereignis ausgelöst hat. Dementsprechend müssen wir beide Listen pflegen, wenn der Benutzer einen neuen Termin erfasst oder einen bereits erfassten Termin wieder aus der Liste löscht.

Ein Ansatz, dieses Manko zu umgehen, wäre es, die komplette `AddHandler`-Logik in das Hauptprogramm zu verlegen. Das würde allerdings dem OOP-Anspruch, wiederverwendbare Komponenten zu schaffen, nicht gerade entsprechen.

Implementieren eigener Event Handler

Es gibt Fälle, bei denen es wichtig sein kann, dass Sie die volle Kontrolle über das Auslösen von und das Registrieren an Ereignissen haben – gerade für die Fehlersuche kann sich das von großer Bedeutung erweisen.

Angenommen, Sie haben eine Komponente entwickelt, bei der Sie aber, weil sie recht komplex ist, genau überprüfen müssen, welche der Instanzen dieser Komponenten zu welcher Zeit ein bestimmtes Ereignis konsumiert und wie lange die Ausführung auf die Reaktion dieses Ereignisses gedauert hat. Schwierig ist das mit herkömmlichen Bordmitteln, weil Sie in das eigentliche Auslösen des Ereignisses nicht eingreifen können. Zwar »weiß« die Komponente, dass sie ein Ereignis auslöst, aber Sie können den Code nicht so formulieren, dass Sie Kenntnis darüber erlangen, mit welchen *Zielen* Ihre Komponente ereignistechnisch verdrahtet ist.

Mit *Custom Event Handlers* (etwa: *benutzerdefinierte Ereignisbehandler*) ist das möglich, und erwähnenswert ist in diesem Zusammenhang auch, dass die Möglichkeit von Custom Event Handlers bereits in Visual Studio 2008 zur Verfügung stand, leider nur nirgendswo dokumentiert war.

Wichtig zu wissen ist: Ein Custom Event Handler definiert das Ereignis und den Ereignishandler in einem Rutsch, Sie brauchen das eigentliche Ereignis in der Klasse, in der Sie einen Custom Event Handler definieren, nicht zusätzlich zu definieren.

Eine entsprechende Editorunterstützung ist ebenfalls vorhanden. Da es insgesamt drei Methoden gibt, in denen die jeweiligen Aufgaben erledigt werden (Registrieren des Ereignisses, De-Registrieren des Ereignisses und Auslösen des Ereignisses), werden diese auch in den Editor eingefügt, sobald Sie die Custom Event Handler-Definition in der Klasse eingefügt und [↵] betätigt haben (Abbildung 16.10).

```
Public Class WithCustomEventHandler

    Custom Event DebugClick As EventHandler
        AddHandler(ByVal value As EventHandler)

        End AddHandler

        RemoveHandler(ByVal value As EventHandler)

        End RemoveHandler

        RaiseEvent(ByVal sender As Object, ByVal e As System.EventArgs)

        End RaiseEvent
    End Event

End Class
```

Abbildung 16.10 Nachdem Sie die *Custom Event*-Zeile eingefügt haben, drücken Sie ↵. Der Visual Basic-Editor erledigt den Rest.

Die Anwendung ist jetzt ganz einfach:

- In `AddHandler` hinterlegen Sie den Code, der ausgeführt werden soll, wenn eine andere Komponente ein Ereignis verdrahtet, also später konsumieren will. Die Komponente übergibt Ihnen dazu einen Ereignishandler – einen besonderen Delegaten –, den Sie sich merken müssen, denn wenn Sie das Ereignis auslösen, müssen Sie dazu einfach nacheinander alle gemerkten Delegaten der verschiedenen Komponenten anspringen.
- In `RemoveHandler` hinterlegen Sie den Code, der ausgeführt werden soll, wenn eine Komponente ein Ereignis nicht mehr konsumieren will.
- `RaiseEvent` ist schließlich die Methode, in der Sie den Code zum Auslösen der Ereignisse hinterlegen. Die Quelle dieses Aufrufs ist anders als bei den beiden oben besprochenen Methoden nicht einer der Konsumenten, sondern die ereignisauslösende Klasse selbst.

Zum Speichern der Eventhandler der Komponenten stellt Ihnen .NET Framework eine besondere Auflistung zur Verfügung, die Sie dazu verwenden können (mehr zu Auflistungen erfahren Sie in Kapitel 22). Ihr Name: `EventHandlerList`. Sie verfügt über die Methoden `AddHandler`, `RemoveHandler` und `Item` (zum Abrufen eines Elements), mit denen Sie über einen Objektschlüssel sogar verschiedene Ereignisse in einer Liste hinterlegen können – für den Fall, dass Ihre Komponente nicht nur einen, sondern mehrere Custom Event Handler definieren soll. Diese sollten Sie also am Anfang Ihrer Klasse folgendermaßen deklarieren:

```
Public Class DebugButton
    Inherits Button

    Private myEventHandlers As EventHandlerList = New EventHandlerList
```

In den entsprechenden Eventmethoden greifen Sie später beispielsweise über einen Stringschlüssel auf die Elemente der Liste zu. Das Hinzufügen eines Eventhandlers in `AddHandler` und das Entfernen eines Eventhandlers in `RemoveHandler` könnte daher folgendermaßen ausschauen:

```
        AddHandler(ByVal value As EventHandler)
            myEventHandlers.AddHandler("DebugClick", value)
        End AddHandler

        RemoveHandler(ByVal value As EventHandler)
            myEventHandlers.RemoveHandler("DebugClick", value)
        End RemoveHandler
```

Das folgende Beispiel demonstriert Custom Event Handler im Zusammenhang. Es simuliert das, was wir am Anfang dieses Abschnitts als Beispiel konstruiert haben: dass sich nachvollziehen lässt, welche Komponente wie lange für die Abarbeitung eines Ereignisses benötigt hat und wann dieses Ereignis stattgefunden hat.

Dazu stellt das Beispiel ein neues Steuerelement zur Verfügung, das von `Button` abgeleitet ist. Dieses Steuerelement namens `DebugButton` definiert ein neues Ereignis mit dem Namen `DebugClick`, das genau so funktioniert wie das `Click`-Ereignis, darüber hinaus aber in einer statischen Liste definiert, welche Komponente das Ereignis konsumiert hat. Sie finden den Beispielcode im Folgenden:

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 16\CustomEvents
```

```
Imports System.ComponentModel   ' EventHandlerList-Auslistung
Imports System.Threading        ' Thread.Sleep um Arbeitslast zu simulieren.

Public Class Form1

    'Ermittelt Zufallszahlen, die wir brauchen, um verschiedene
    'Zeitspannen lange zu warten (zur Arbeitslastsimulation).
    Private myRandomGenerator As New Random

    'Die Ereignis-Behandlungsmethode für die Click-Ereignisse aller Debug-Buttons.
    Private Sub DebugButton DebugClick(ByVal sender As Object,
                                       ByVal e As System.EventArgs)
        Handles DebugButton1.DebugClick, DebugButton2.DebugClick,
                DebugButton3.DebugClick

        'Simulieren, dass die Arbeitslast zwischen 250 und 2000 ms dauert.
        'Die Schaltflächen werden dabei ge-disablet, damit man sieht,
        'wann es weiter geht.
        DirectCast(sender, Control).Enabled = False

        'Der Windows-Nachrichten-Warteschlange Zeit zum "Luftholen" geben,
        'damit das Disabeln der Schaltfläche stattfindet.
        My.Application.DoEvents()

        'Arbeitslast simulieren: Thread.Sleep setzt den aktuellen Thread aus.
        System.Threading.Thread.Sleep(myRandomGenerator.Next(250, 2000))

        'Danach die Schaltfläche wieder aktivieren.
        DirectCast(sender, Control).Enabled = True

    End Sub

    'Wird ausgelöst, wenn der Result-Button angeklickt wird,
    'und dann zeigt die Anwendung eine Liste der Ereignisse an,
    'die durch das Klicken der verschiedenen Debug-Schaltflächen
    'zustande gekommen ist.
    Private Sub ResultDebugButton Click(ByVal sender As System.Object,
                                        ByVal e As System.EventArgs)
                                    Handles ResultDebugButton.Click

        'Das Ergebnisformular erzeugen wir jetzt 'on the fly'
        Dim resultForm As New Form With {.Text = "Ergebnisliste"}
```

```
        'Listbox dem Formular hinzufügen
        Dim resultList As New ListBox() With {.Dock = DockStyle.Fill}

        'Startposition des Fensters ist Mitte des Bildschirms.
        resultForm.StartPosition = FormStartPosition.CenterScreen

        'Ergebnisliste zuordnen:
        resultList.DataSource = DebugButton.DebugClickInfos

        'Die ListBox auf das Formular packen.
        resultForm.Controls.Add(resultList)

        'Dialog anzeigen.
        resultForm.ShowDialog()
    End Sub
End Class

'Diese Klasse speichert einen Ereignisdatensatz.
Public Class DebugClickInfo
    Public Property Sender As Object                ' Wer hat ausgelöst??
    Public Property EventTime As Date               ' Wann wurde ausgelöst?
    Public Property ExecutionDuration As TimeSpan   ' Wie lange hat Behandlung gedauert?

    'ToString ist überschrieben, damit wir eine Liste mit Instanzen
    'dieser Klasse direkt einer ListBox zuweisen können. Damit übernimmt
    'die DebugClickInfo-Klasse selber die Darstellung seines Inhalts als Text.
    Public Overrides Function ToString() As String

        'Rückgabestring zusammenbasteln. Erst der Ereigniszeitpunkt,
        'dann die Dauer und schließlich der Sender, der das Ereignis ausgelöst hat.
        Dim retString = EventTime.ToLongTimeString & ":  "
        retString &= ExecutionDuration.ToString("ss\:fff") & "    (Sender: "

        'Wenn es sich um ein Steuerelement als Sender handelt,
        'geben wir den Namen aus, anderenfalls das, was ToString liefert.
        If GetType(Control).IsAssignableFrom(Sender.GetType) Then
            retString += DirectCast(Sender, Control).Name
        Else
            retString += Sender.ToString
        End If

        retString &= ")"
        Return retString
    End Function
End Class

'Der DebugButton kann das Gleiche wie der "normale" Button
'bekommt aber ein zusätzliches Ereignis, dass durch einen
'Custom Event Handler ein Protokoll aller "Klicks" führt.
Public Class DebugButton
    Inherits Button

    'Die Liste der Delegaten, wenn die Ereignisse in anderen Komponenten
    '(zum Beispiel vom Form in diesem Beispiel) registriert werden.
    Private myEventHandlers As EventHandlerList = New EventHandlerList

    'Die Ergebnisliste, in der verzeichnet wird, wie lange was gedauert hat.
    Private Shared s debugClickInfos As New List(Of DebugClickInfo)

    'So bekommt die Komponente mit, wenn geklickt wurde.
    Protected Overrides Sub OnClick(ByVal e As System.EventArgs)
        MyBase.OnClick(e)
```

```
        RaiseEvent DebugClick(Me, e)
    End Sub

    'Dieser Code-Block definiert das CustomEreignis.
    Custom Event DebugClick As EventHandler

        'Wird aufgerufen, wenn eine Komponente, die das Ereignis
        'konsumieren möchte, es "verdrahtet".
        AddHandler(ByVal value As EventHandler)
            'Den übergebenden EventHandler in der Ereignis-Delegaten-Liste
            'dieser Instanz speichern. "DebugClick" ist hier ein beliebiger
            'aber eindeutiger Name, den wir zum Wiederfinden aller Registraten
            'an diesem Ereignis benötigen.
            myEventHandlers.AddHandler("DebugClick", value)
        End AddHandler

        'Wird aufgerufen, wenn eine Komponente ein verdrahtetes Ereignis wieder löst.
        RemoveHandler(ByVal value As EventHandler)
            'EventHandler aus der Ereignis-Delegaten-Liste wieder entfernen.
            myEventHandlers.RemoveHandler("DebugClick", value)
        End RemoveHandler

        'Wird aufgerufen, wenn eine Instanz dieser Klasse
        'das DebugClick-Ereignis auslöst.
        RaiseEvent(ByVal sender As Object, ByVal e As System.EventArgs)

            'Gibt es überhaupt Registraten an dieser Ereignis?
            If myEventHandlers.Item("DebugClick") IsNot Nothing Then
                'Alle Registraten des Ereignisses dieser Instanz durchlaufen
                For Each delItem In myEventHandlers.Item("DebugClick").GetInvocationList

                    'Alles entsprechend messen und ...
                    Dim eventTime = Now
                    Dim sw = Stopwatch.StartNew
                    'Ruft den eigentlichen Ereignisbehandlungscode in der Zielkomponente auf.
                    CType(delItem, EventHandler).Invoke(sender, e)
                    sw.Stop()

                    '... die Ergebnisse in die Liste schreiben.
                    s debugClickInfos.Add(New DebugClickInfo With {
                                          .EventTime = eventTime,
                                          .ExecutionDuration =
                                              TimeSpan.FromMilliseconds(sw.ElapsedMilliseconds),
                                          .Sender = sender
                                          })
                Next
            End If
        End RaiseEvent
    End Event

    'Statische Eigenschaft, die die Ergebnisliste der "Ereignisaufzeichnungen"
    'aller Instanzen dieser Klasse hat.
    Public Shared ReadOnly Property DebugClickInfos As List(Of DebugClickInfo)
        Get
            Return s debugClickInfos
        End Get
    End Property

End Class
```

Kapitel 17

Enums

In diesem Kapitel:

Bestimmung der Werte der Aufzählungselemente 555
Bestimmung der Typen der Aufzählungselemente 556
Konvertieren von Enums 556
Flags-Enums (Flags-Aufzählungen) 558

Enums (etwa: *Aufzählungen*, nicht zu verwechseln mit *Auflistungen*, den *Collections*) dienen in erster Linie dazu, Programmierern das Leben zu erleichtern. Programmierer müssen sich gerade in den Zeiten von .NET schon eine ganze Menge merken und sind für jede Unterstützung in dieser Richtung dankbar.

Mit Enums (in der Microsoft-Terminologie auch *Enumerationen* genannt) rufen Sie konstante Werte, die thematisch zu einer Gruppe gehören, per Namen ab. Gleichzeitig haben Sie die Möglichkeit, die Werte auf diese Namen zu beschränken. Angenommen, Sie haben eine kleine Kontaktverwaltung auf die Beine gestellt, und diese Datenbankanwendung erlaubt es Ihnen, Ihre Kontakte zu kategorisieren. Sie können also einstellen, ob ein Kontakt ein Kunde, ein Bekannter, ein Geschäftskollege ist oder sonst einer Gruppierung angehört. Zur Vereinfachung des Beispiels nehmen wir ferner an, dass ein Kontakt jeweils nur zu einer dieser Gruppen gehören kann (es soll sozial Hochbegabte geben, die mit ihren Geschäftskollegen auch noch bekannt sind). Für jede dieser Kategorien haben Sie eine bestimmte Nummer vergeben. Und in Abhängigkeit bestimmter Kategorien (Nummern) können Sie nun spezielle Funktionen in Ihrer Datenbankanwendung aufrufen oder auch nicht (Ansprechpartner-Kontakte lassen sich beispielsweise Firmenkontakten zuordnen, Lieferanten aber nicht, da diese Kontakte je selbst eine Firma darstellen – nur als Beispiel). Ohne Enums könnte man den Code so schreiben, dass man ihre Funktionen auch mit der Nummer (zum Beispiel mit einem Integer) anspricht, aber dies ist zum einen schwer zu merken und zum anderen könnten Sie kaum verhindern, dass man die Funktionen mit 42 aufruft; was als Integer vollkommen in Ordnung wäre – nur haben Sie eben keine 42. Kategorie, sondern gerade mal 10 oder ähnlich. So leisten Enums sowohl die bessere Merkbarkeit als auch die Einschränkung möglicher Werte.

Wie gesagt: Sie können sich eine eigene Liste erstellen und die Zuordnung Kontaktkategorie/Nummer sozusagen im Kopf durchführen. Oder Sie legen praktischerweise eine solche Enum an, die Ihnen die Arbeit erleichtert.

BEGLEITDATEIEN Die in einem Projekt zusammengefassten Beispiele für dieses Kapitel finden Sie im Verzeichnis

```
...\VB 2010 Entwicklerbuch\Kapitel 17\Enums
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
Public Enum KontaktKategorie
    Familie
    Freund
    Bekannter
    Kollege
    Geschäftspartner
    Kunde
    Lieferant
    ZuMeiden
    Firma
    AnsprechpartnerBeiFirmenKontakt
End Enum
```

Wie setzen Sie diese Enum-Elemente nun ein? Ganz einfache Geschichte: Wenn nichts anderes gesagt wird, vergibt .NET Werte für die Enums, und zwar analog der Reihenfolge im `Enum`-Block von oben nach unten bei 0 angefangen in aufsteigender Reihenfolge. Sie können Variablen vom Typ der Aufzählung definieren und anstatt eine Nummer zu verwenden, geben Sie einfach den Enum-Typ an:

```
Sub main()

    Dim locKontakte As KontaktKategorie

    locKontakte = KontaktKategorie.Geschäftspartner
    Console.WriteLine(locKontakte)
    Console.ReadLine()

End Sub
```

Wenn Sie diesen Code ausführen, gibt er im Konsolenfenster den Wert 4 aus.

Bestimmung der Werte der Aufzählungselemente

Falls Sie mit der vorgegebenen Durchnummerierung nicht einverstanden sind, legen Sie bei der Aufzählungsdefinition eben selbst Hand an, wie im folgenden Beispiel gezeigt wird:

```
Public Enum KontaktKategorie
    Familie = 10
    Freund
    Bekannter
    Kollege
    Geschäftspartner
    Kunde  = 20
    Lieferant
    ZuMeiden = 30
    Firma
    AnsprechpartnerBeiFirmenKontakt
End Enum
```

Das gleiche Programm, abermals ausgeführt, liefert anschließend den Wert 14.

Dubletten sind erlaubt!

Dubletten sind bei den Aufzählungselementen übrigens erlaubt. So haben in der Aufzählungsdefinition

```
Public Enum Kontaktkategorie
    Familie = 10
    Freund
    Bekannter
    Kollege
    Geschäftspartner = 20
    Kunde
    Lieferant = 19
    ZuMeiden
    Firma
    AnsprechpartnerBeiFirmenKontakt
End Enum
```

sowohl `Geschäftspartner` als auch `ZuMeiden` den Wert 20.

Bestimmung der Typen der Aufzählungselemente

Solange nichts anderes gesagt wird, werden die Elemente einer Aufzählung intern als `Integer` angelegt. Sie können allerdings bestimmen, ob die Aufzählungselemente darüber hinaus als `Byte`, `Short` oder `Long` deklariert werden sollen. Wir sollten uns bewusst machen, dass `Integer` (in .NET Double Words) aufgrund der Ausrichtung an geraden Speicheradressen bei Intel-Prozessoren aus Performancegründen meist vorzuziehen sind. Eine entsprechende Typenangabe bei der Enum-Definition reicht aus:

```
Public Enum KontaktKategorie As Short
    Familie
    Freund
    Bekannter
    ...
End Enum
```

Ermitteln des Elementtyps zur Laufzeit

Wenn Sie zur Laufzeit ermitteln müssen, welcher primitive Datentyp sich hinter einem Aufzählungselement verbirgt, machen Sie das einfach mit der `GetUnderlyingType`-Eigenschaft:

```
'Ermittelt den Namen des zugrunde liegenden primitiven Datentyps einer Aufzählung.
Console.WriteLine([Enum].GetUnderlyingType(GetType(KontaktKategorie)).Name)

'Ermittelt den Typnamen anhand einer Aufzählungsvariablen.
Console.WriteLine([Enum].GetUnderlyingType(locKontakte.GetType).Name)
```

Auf unser bisheriges Beispiel angewendet, würden diese beiden Zeilen folgende Ausgabe im Konsolenfenster erscheinen lassen:

```
Int16
Int16
```

WICHTIG Wenn Sie auf den `Enum`-Typbezeichner im Quelltext zugreifen, müssen Sie, wie hier im Beispiel, das Schlüsselwort in eckige Klammern setzen, damit es vom Visual Basic-Editor korrekt als Ausdruck verarbeitet werden kann.

Konvertieren von Enums

In vielen Fällen kann es sinnvoll sein, ein Aufzählungselement in seinen zugrunde liegenden Typ umzuwandeln – beispielsweise um den Wert einer Enum-Variablen in eine Datenbank zu schreiben. Einige Fälle machen es auch notwendig, einen Aufzählungswert auf Basis des als Zeichenkette vorliegenden Namens eines Aufzählungselements entstehen zu lassen oder ein Aufzählungselement in seinen Elementnamen (also in einen String) umzuwandeln.

In Zahlenwerte umwandeln und aus Werten definieren

Um ein Aufzählungselement in seinen Wert umzuwandeln (und im Bedarfsfall auch zurück), verfahren Sie folgendermaßen (das folgende Beispiel geht immer noch von einer Aufzählungstypdefinition als `Short` aus):

```
Dim locKontakte As KontaktKategorie
Dim locShort As Short

        locKontakte = KontaktKategorie.Geschäftspartner

        'Typumwandlung von Aufzählung zugrunde liegenden Datentyp...
        locShort = locKontakte
        locShort = KontaktKategorie.Firma

        '...und umgekehrt, auch nicht schwieriger:
        locKontakte = CType(locShort, KontaktKategorie)
```

In Strings umwandeln und aus Strings definieren

Falls Sie wissen wollen, welcher Elementname sich hinter dem Wert einer Aufzählungsvariablen verbirgt, verfahren Sie folgendermaßen:

```
Dim locKontakte As KontaktKategorie = KontaktKategorie.Firma
Console.WriteLine(locKontakte.ToString())
```

Die Ausgabe lautet:

```
Firma
```

Beim umgekehrten Verfahren wird es wieder ein wenig aufwändiger:

```
'Umwandlung zurück in ein Enum-Element aus einem String.
Dim locString As String = "Geschäftspartner"
locKontakte = DirectCast([Enum].Parse(GetType(KontaktKategorie), locString), KontaktKategorie)
```

Hier gilt: Die statische Funktion `Parse` der `Enum`-Klasse erlaubt das Generieren eines Aufzählungselements zur Laufzeit. `Parse` erwartet dabei den Typ der Aufzählung, den Sie zunächst mit `GetType` ermitteln müssen. Da `Parse` ein Objekt erzeugt, das ein geboxtes Aufzählungselement enthält, müssen Sie dieses zu guter Letzt mit `DirectCast` aus dem `Object` wieder »entboxen«.

TIPP Ab .NET Framework 4.0 stellen Enums auch eine `TryParse`-Funktion auf Basis von Generics (siehe Kapitel 18) zur Verfügung, mit der das Zurückwandeln von Zeichenketten in den eigentlichen Enum-Wert sehr viel einfacher wird. Das folgende Beispiel zeigt, wie das oben stehende Beispiel mit diesem Äquivalent ausschaut:

```
        'Mit .NET Framework 4.0 geht auch das ein wenig einfacher:
        If [Enum].TryParse(locString, locKontakte) Then
            Console.WriteLine("Enum-Wert Geschäftspartner wurde erkannt und umgewandelt!")
        End If
```

Flags-Enums (Flags-Aufzählungen)

Flags-Aufzählungen sind eine ideale Einrichtung, wenn Sie Aufzählungen mit Elementen verwenden müssen, die untereinander kombinierbar sind. So kann es durchaus sein, dass – um bei unserem Beispiel zu bleiben – ein Kontakt in Ihrer Datenbank sowohl ein *Freund* als auch ein *Geschäftskollege* ist. .NET Framework unterstützt solche Szenarien auf ideale Weise.

Bei der Definition einer *Flags*-Aufzählung müssen Sie drei Dinge beachten: Sie sollten eine Aufzählungsbenennung für den Fall »keine der Kombinationen» definieren (beispielsweise in Form von `Keine` oder `None`). Dieser Eintrag hat den Wert 0. Zweitens: Sie müssen Werte vergeben, die sich bitweise kombinieren lassen – und dazu zählen Sie die einzelnen Werte als Zweierpotenzen hoch. Und drittens: Sie statten die Aufzählung mit dem `Flags`-Attribut aus.

Das klassische Beispiel sind die Schaltflächenanordnungen und Hinweis-Symbole der in Windows »eingebauten« MessageBox. Diese schon sehr alte Windows-API aus *User32.dll* konnte man früher in C/C++ direkt aufrufen. Die Definition lautete:

```
int MessageBox(HWND hWnd,LPCTSTR lpText,LPCTSTR lpCaption,UINT uType);
```

Die Syntax aus C soll uns hier nicht interessieren – wie man hier mit welchem Parameter welches Ziel erreichte, ist schon eher von Interesse: Wollen Sie beispielsweise ein Fragezeichen anzeigen, müssen Sie als `uType` den Wert 32 angeben, für die *Ja*- und *Nein*-Schaltflächen 4 als `uType`; 4+32=36 also für beide zusammen. .NET stellt Ihnen diese API in einem Objekt gekapselt ebenfalls zur Verfügung:

```
MessageBox.Show("Hallo world", "VB for ever", MessageBoxButtons.YesNo, MessageBoxIcon.Question)
```

Um es allerdings für die Benutzer etwas einfacher zu machen, hat man jedoch die Parameter für Schaltflächen und Hinweis-Symbole dort getrennt. Und jetzt raten Sie mal, welche Werte `MessageBoxButtons.YesNo` und `MessageBoxIcon.Question` zugewiesen wurden – doch nicht etwa 4 und 32?

Das folgende Beispiel zeigt, wie es geht:

```
<Flags()> _
Public Enum KontaktKategorien
    Keine = 0
    Familie = 1
    Freund = 2
    Bekannter = 4
    Kollege = 8
    Geschäftspartner = 16
    Kunde = 32
    Lieferant = 64
    ZuMeiden = 128
    Firma = 256
    AnsprechpartnerBeiFirmenKontakt = 512
End Enum
```

Wenn Sie anschließend kombinierte Zuweisungen an eine Variable vom Typ `KontaktKategorien` vornehmen wollen, nehmen Sie den logischen `Or`-Operator zu Hilfe, wie das folgende Beispiel zeigt:

```
Sub EnumFlags()

    Dim locKontakte As KontaktKategorien
    locKontakte = KontaktKategorien.Freund Or KontaktKategorien.Geschäftspartner
    Console.WriteLine(locKontakte)
    Console.WriteLine(locKontakte.ToString())

    Console.ReadLine()

End Sub
```

Dieses Beispiel erzeugt die folgende Ausgabe:

```
18
Freund, Geschäftspartner
```

Abfrage von Flags-Aufzählungen

Bei der Abfrage von *Flags*-Aufzählungen müssen Sie ein wenig aufpassen, da Kombinationen Werte ergeben, die keinem bestimmten Wert eines Elements entsprechen. Erst ein so genanntes Ausmaskieren (Ermitteln eines einzelnen Bitwerts) mit dem `And`-Operator ergibt einen richtigen Wert. Auch hier demonstriert ein Beispiel, wie es geht:

```
Dim locKontakte As KontaktKategorien
locKontakte = KontaktKategorien.Freund Or KontaktKategorien.Geschäftspartner

'Achtung bei Flags! Bits müssen ausmaskiert werden!
'Diese Abfrage liefert das falsche Ergebnis!
If locKontakte = KontaktKategorien.Geschäftspartner Then
    Console.WriteLine("Du bist ein Geschäftspartner")
Else
    Console.WriteLine("Du bist kein Geschäftspartner")
End If

'So ist's richtig; diese Abfrage liefert das richtige Ergebnis:
If (locKontakte And KontaktKategorien.Freund) = KontaktKategorien.Freund Then
    Console.WriteLine("Du bist ein Freund")
Else
    Console.WriteLine("Du bist kein Freund")
End If

'Und so funktionieren Kombiabfragen:
If locKontakte = (KontaktKategorien.Freund Or KontaktKategorien.Geschäftspartner) Then
    Console.WriteLine("Du bist ein Freund und ein Geschäftspartner")
End If
```

Wenn Sie dieses Beispiel laufen lassen, erhalten Sie das folgende Ergebnis:

```
Du bist kein Geschäftspartner
Du bist ein Freund
Du bist ein Freund und ein Geschäftspartner
```

Die erste Zeile wird falsch ausgegeben, da der Wert der `Enum`-Variablen *locKontakte* nicht ausmaskiert und dann verglichen wird. Stattdessen wird fälschlicherweise direkt verglichen.

TIPP Seit .NET Framework 4.0 ist die Abfrage von Flag-Enums einfacher geworden, denn in dieser .NET Framework-Version stellen Enums die `HasFlag`-Methode zur Verfügung, die folgendermaßen funktioniert:

```
'Mit .NET Framework 4.0 geht das auch:
If locKontakte.HasFlag(KontaktKategorien.Freund) Then
    Console.WriteLine("Du bist ein Freund")
Else
    Console.WriteLine("Du bist kein Freund")
End If
```

Kapitel 18

Generics (Generika) und generische Auflistungen

In diesem Kapitel:

Einführung 562
Lösungsansätze 563
Standardisierung der Codebasis eines Typs durch den Einsatz von Generics 565
Beschränkungen (Constraints) 568

Einführung

Generics – oder »Generika«, wie Microsoft die so schöne deutsche Übersetzung dafür fand – sind nicht neu in Visual Basic 2010, denn sie wurden bereits mit Visual Basic 2005 eingeführt. Für das Entwickeln robuster Anwendungen sind Generics aber von entscheidender Bedeutung, und viele Typen und Klassen, mit denen Sie täglich umgehen, basieren auf Generics – seien es Nullables, generische Auflistungen wie `List(Of T)` oder `Collection(Of T)`, auch generische Delegaten wie `Action(Of T)` und `Function(Of T)` oder `Tupel(Of T)`. Dazu sind sie entscheidend für das Verständnis einer immer entscheidender werdenden Programmiertechnik: *LINQ*, über die Teil E noch zu berichten wissen wird.

Und obwohl Generics eine so grundlegende Bedeutung für die Entwicklung von Software mit Visual Basic haben, nicht jeder Visual Basic-Entwickler traut sich an sie heran – meines Erachtens unnötigerweise, denn hat man erst einmal das Konzept verstanden, vereinfachen Generics das tägliche Entwicklerleben ungemein und lassen Ihre Programme sehr viel stabiler werden.

Generics: Verwenden einer Codebasis für verschiedene Typen

Wenn Sie Methoden und Eigenschaften entwickeln, haben diese unter Umständen einen Nachteil: Sie verarbeiten, wenn sie typsicher sein sollen, nur einen bestimmten Datentyp.

Die Alternative dazu ist, dass Sie einen Typ schaffen, der die Aufnahme beliebiger Datentypen durch den auf `Object` basierenden Einsatz ermöglicht, doch ein solcher Typ ist dann nicht typsicher und kann zur Laufzeit Ausnahmen auslösen, mit denen Sie nicht rechnen.

BEGLEITDATEIEN Im Folgenden sehen Sie zunächst den Klassencode, der einen Adresseneintrag verwaltet – dieses Projekt finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 18\Generics01
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Dieses Beispiel verwendet die aus den zurückliegenden Kapiteln bekannte Klasse `DynamicList` in leicht modifizierter Form. Das Modul verwendet eine Instanz von `DynamicList` und fügt ihr ein paar Elemente hinzu. Diese Elemente gibt es anschließend in einer `For Each`-Schleife wieder im Konsolenfenster aus:

```
Module mdlMain
    Sub Main()
        Dim locListe As New DynamicList
        locListe.Add(123.32)
        locListe.Add(126.32)
        locListe.Add(124.52)
        locListe.Add(29.99)
        locListe.Add(13.54)
        'Der wird Probleme machen!
        locListe.Add(#12/31/2010 4:00:00 PM#)
        locListe.Add(43.32)
        For Each locItem As Double In locListe
            Console.WriteLine(locItem)
        Next
```

```
        Console.WriteLine()
        Console.WriteLine("Taste drücken zum Beenden!")
        Console.ReadKey()
    End Sub

End Module
```

Wenn Sie dieses Programm starten, sehen Sie anschließend eine Ausgabe, wie sie auch Abbildung 18.1 zeigt.

Abbildung 18.1 Die Liste wird nur bis zum `Date`-Element ausgegeben, denn dieses ist nämlich nicht in `Double` konvertierbar

Es ist klar, wieso das passiert. Wir wollten eine Liste mit reinen `Double`-Elementen aufbauen, aber uns ist ein `Date`-Element dazwischen gerutscht. Solche Fehler sind in einem so einfachen Programm, wie wir es hier als Beispiel verwenden, noch auf den ersten Blick zu erkennen – doch in größeren Projekten sollte man solche Fehler von vornherein zu vermeiden versuchen.

Lösungsansätze

Und wie kann man das machen? Indem man eine Klasse wie die `DynamicList` typsicher macht. Indem man sie von vornherein erst gar keinen allgemein gültigen Datentyp wie `Object` akzeptieren lässt, sondern ausschließlich Werte vom Typ `Double`.

Und um das zu erreichen, nehmen wir uns den Quellcode der `DynamicList` vor und führen die entsprechenden Änderungen durch. Konsequenterweise nennen wir diese Klasse dann auch `DynamicListDouble` (wie sie im angesprochenen Beispielprojekt übrigens bereits in einer eigenen Klassendatei vorhanden ist). Im folgenden Listing finden Sie all die Stellen in Fettschrift markiert, an denen der Datentyp `Object` in `Double` geändert wurde:

```
Class DynamicListDouble
    Implements IEnumerable

    Protected myStep As Integer = 4         ' Schrittweite, um die das Array erhöht wird.
    Protected myCurrentArraySize As Integer ' Aktuelle Array-Größe
    Protected myCurrentCounter As Integer   ' Zeiger auf aktuelles Element
    Protected myArray() As Double           ' Array mit den Elementen.

    Sub New()
        myCurrentArraySize = myStep
        ReDim myArray(myCurrentArraySize - 1)
    End Sub

    Sub Add(ByVal Item As Double)

        'Element im Array speichern
        myArray(myCurrentCounter) = Item

        'Zeiger auf nächstes Element erhöhen
        myCurrentCounter += 1

        'Prüfen, ob aktuelle Arraygrenze erreicht wurde
        If myCurrentCounter = myCurrentArraySize - 1 Then
            'Neues Array mit mehr Speicher anlegen,
            'und Elemente hinüberkopieren. Dazu:

            'Neues Array wird größer:
            myCurrentArraySize += myStep

            'Temporäres Array erstellen
            Dim locTempArray(myCurrentArraySize - 1) As Double
            Array.Copy(myArray, locTempArray, myArray.Length)
            myArray = locTempArray
        End If
    End Sub
.
.
.
End Class
```

Wenn Sie die Klasse auf diese Weise abgeändert und das eigentliche Programm wie folgt modifiziert haben:

```
Module mdlMain

    Sub Main()
        Dim locListe As New DynamicListDouble
        locListe.Add(123.32)
        locListe.Add(126.32)
.
.
.
```

zeigt Ihnen Visual Basic schon zur Entwurfszeit eine Fehlermeldung, wie Sie sie auch in Abbildung 18.2 sehen können.

```
    Dim locDoubleListe As New DynamicList(Of Double)
    locDoubleListe.Add(123.32)
    locDoubleListe.Add(126.32)
    locDoubleListe.Add(124.52)
    'Date geht nicht:
    locDoubleListe.Add(#10/18/2005 3:20:00 PM#)

    Dim locDateListe As New DynamicList(Of Da
    locDateListe.Add(#12/31/2005 4:00:00 PM#)
    locDateListe.Add(#11/24/2005 6:20:00 PM#)
    locDateListe.Add(#10/18/2005 3:20:00 PM#)
    'Double geht nicht:
    'locDateListe.Add(124.52)

    Console.WriteLine()
    Console.WriteLine("Taste drücken zum Been
    Console.ReadKey()
End Sub
```

Das Umwandeln von "Date" in "Double" erfordert das Aufrufen der Methode "Date.ToOADate".

Fügen Sie ".ToOADate()" nach "#10/18/2005 3:20:00 PM#" ein.

```
locDoubleListe.Add(124.52)
'Date geht nicht:
locDoubleListe.Add(#10/18/2005 3:20:00

Dim locDateListe As New DynamicList(Of
```

Alle Seitenansichten erweitern

Abbildung 18.2 Bei typsicheren Klassen sehen Sie Fehlermeldungen bei »falschen« Typen bereits im Editor zur Entwurfszeit – wenngleich die Fehlermeldungen ab und zu über das Ziel hinausschießen

Gut – diese Fehlermeldung an dieser Stelle schießt ein wenig über das Ziel hinaus; hätte es der Editor dabei belassen, uns über den falschen Typ an dieser Stelle zu informieren, wäre das ausreichend gewesen. Aber immerhin: Sie sehen auf jeden Fall durch die Verwendung der typsicheren Klasse schon zur Entwurfszeit, dass Sie einen Typen verwendet haben, den Sie mit dieser Klasse nicht verwenden dürfen.

Standardisierung der Codebasis eines Typs durch den Einsatz von Generics

Nun ist das Entwickeln einer Klasse wie `DynamicList` schon ein wenig aufwändiger. Und es werden in Ihrer späteren Entwicklerzeit Klassen und Strukturen eine Rolle spielen, die noch viel, viel komplexer sind. Doch gleichzeitig wird es gerade bei Klassen oder Strukturen, die große Datenmengen verwalten, immer wieder vorkommen, dass Sie diese für den Einsatz unterschiedlichster Typen verwenden wollen – für unser `DynamicList`-Beispiel trifft diese Aussage mehr als zu, denn: Auch Zeichenketten ließen sich mit der Klasse wunderbar verwalten. Und auch `Integer`-Werte. Und auch `Decimal`-Werte. Und auch … – eigentlich ist dieser Typ für alle Datentypen und Objektinstanzen geeignet, die Sie in größeren Mengen in einer Liste speichern wollen. Doch als Programmierer sollten Sie aus den genannten Gründen immer auch auf Typsicherheit bestehen, und so bliebe Ihnen bislang eigentlich nur die Möglichkeit, …

- … für jeden benötigten Datentyp eine neue Version der verarbeitenden Klasse zu erstellen. Dieser Aufwand ist allerdings enorm groß und zieht ein mindestens ebenso großes Problem nach sich: Finden Sie einen Fehler in einer Klasse, müssen Sie diesen in allen Klassen ändern, die sich nur durch ihren verarbeitenden Typ unterscheiden (`DynamicListDouble`, `DynamicListString`, `DynamicListDate` und welche Klasse Sie auch immer sonst noch eingerichtet hätten).
- … eine Klasse auf Basis einer Schnittstelle zu erstellen, mit der die Typen durch Vererbung anpassbar sind. Damit hält sich der Pflegeaufwand in Grenzen, da eine Fehlerbehebung in der Basisklasse sich natürlich auch in den Klassenableitungen widerspiegelt. Doch solche Klassen zu implementieren erfordert extrem abstraktes Denken und sorgfältige Planung, und dafür steht nicht unbedingt immer die Zeit zur Verfügung.

Mit generischen Datentypen wird das anders. Bei generischen Datentypen legen Sie sich während der Entwicklung überhaupt noch nicht fest, welchen Typ Ihre Klasse oder Struktur später einmal verarbeiten soll. Sie arbeiten stattdessen mit so genannten Typparametern, durch die der Typ – dem JITter sei Dank – erst bei der ersten Verwendung zur Laufzeit ersetzt wird.

Mit anderen Worten: Das, was Sie mit dem Kopieren und Typanpassen Ihres Codes zur Entwicklungszeit manuell machen, dafür sorgen JITter und die Technik der Generics zur Laufzeit automatisch. Vereinfacht gesagt: So wie Sie bei Word mit Suchen und Ersetzen arbeiten können, wird der IML-Code Ihrer generischen Klasse kopiert, und alle Typparameter werden durch den angegebenen Typ ersetzt.[1]

In der Praxis und für unser `DynamicList`-Beispiel sieht das wie folgt aus: Anstelle sich von vornherein für einen Datentyp wie `Double`, `Integer` oder `String` zu entscheiden, platzieren Sie an den entscheidenden Stellen eine Art Platzhalter – einen Typparameter –, den Sie im Kopf der Klasse mit dem Zusatz `Of` benennen, etwa so:

```
Class DynamicList(Of flexiblerDatentyp)
```

Und anstatt anschließend innerhalb der Klasse einen fixen Datentyp zu verwenden, setzen Sie diese Typparameter als Stellvertreter ein. Für unsere `DynamicList`-Klasse bedeutet das:

```
Class DynamicList(Of flexiblerDatentyp)
    Implements IEnumerable

    Protected myStep As Integer = 4          ' Schrittweite, durch die das Array erhöht wird.
    Protected myCurrentArraySize As Integer  ' Aktuelle Array-Größe
    Protected myCurrentCounter As Integer    ' Zeiger auf aktuelles Element
    Protected myArray() As flexiblerDatentyp ' Array mit den Elementen.

    Sub New()
        myCurrentArraySize = myStep
        ReDim myArray(myCurrentArraySize - 1)
    End Sub

    Sub Add(ByVal Item As flexiblerDatentyp)

        myArray(myCurrentCounter) = Item
        myCurrentCounter += 1
        If myCurrentCounter = myCurrentArraySize - 1 Then
            myCurrentArraySize += myStep

            'temporäres Array erstellen
            Dim locTempArray(myCurrentArraySize - 1) As flexiblerDatentyp

            'Elemente kopieren;
            Array.Copy(myArray, locTempArray, myArray.Length)
            myArray = locTempArray
        End If
    End Sub
    'Liefert die Anzahl der vorhandenen Elemente zurück
    Public Overridable ReadOnly Property Count() As Integer
        Get
```

[1] Ganz so einfach geht es natürlich nicht. Es gibt für den JITter bzw. Compiler durchaus die Möglichkeit, Codegemeinsamkeiten in generischen Klassen bestehen zu lassen, und nur dort neuen Code zu generieren, wo es nicht anders möglich ist.

```
            Return myCurrentCounter
        End Get
    End Property

    'Erlaubt das Zuweisen
    Default Public Overridable Property Item(ByVal Index As Integer) As flexiblerDatentyp
        Get
            Return myArray(Index)
        End Get
        Set(ByVal Value As flexiblerDatentyp)
            myArray(Index) = Value
        End Set
    End Property

    Public Function GetEnumerator() As System.Collections.IEnumerator
                        Implements System.Collections.IEnumerable.GetEnumerator
        Dim locTempArray(myCurrentArraySize) As flexiblerDatentyp
        Array.Copy(myArray, locTempArray, myArray.Length)
        Return myArray.GetEnumerator
    End Function
End Class
```

Und nun können Sie `DynamicList` für jeden Datentyp verwenden, den Sie möchten, und Sie müssen dabei nicht auf Typsicherheit verzichten, wie Abbildung 18.3 zeigt. In der Abbildung sehen Sie zweierlei. Zunächst wird gezeigt, wie Sie einen generischen Datentyp anwenden. In der `Sub Main` des Moduls wird die generische Klasse einmal auf Basis des Datentyps `Double` definiert

```
Dim locDoubleListe As New DynamicList(Of Double)
```

und einmal auf Basis des Datentyps `Date`:

```
Dim locDateListe As New DynamicList(Of Date)
```

Abbildung 18.3 Mit `Of` bestimmen Sie, für welchen Datentyp Ihre generische Klasse zur Anwendung kommen soll

Und anhand der Fehlerliste, die Sie ebenfalls in der Abbildung sehen können, erkennen Sie auch, dass beide »Typversionen« der Klasse auf ihre Weise typsicher sind. Sie können der einen nur Werte vom Typ `Double` und der anderen nur Werte vom Typ `Date` hinzufügen. Jeder Versuch, einer Liste einen jeweils anderen Typ unterzujubeln, wird schon zur Entwurfszeit mit einer entsprechenden Fehlermeldung bestraft.

Beschränkungen (Constraints)

Im gezeigten Beispiel können Sie eine `DynamicList` so definieren, dass sie sich aus jedem beliebigen Typ bilden kann. Unter bestimmten Umständen kann das aber nicht erwünscht sein, und zwar genau dann, wenn Sie innerhalb einer generischen Klasse andere generische Typen verwenden, deren Typ Sie aber zur Entwicklungszeit noch nicht kennen.

BEGLEITDATEIEN Das folgende Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 18\Generics02
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Beschränkungen für generische Typen auf eine bestimmte Basisklasse

Dazu ein Beispiel: Angenommen, Sie haben eine Anwendung geschaffen, die die verschiedensten Körper (Quader, Kugel, Pyramiden) berechnen, verwalten und darstellen muss. Sie möchten jetzt eine generische Klasse schaffen, die nicht nur die verschiedenen Typen von Körpern in einer Liste wie die `DynamicList` speichert, sondern Sie möchten, dass diese Klasse auch deren Gesamtvolumen berechnen soll.

Nehmen wir weiter an, dass es in unserem Beispiel eine Basisklasse gibt, auf der alle Körperklassen basieren. Diese Basisklasse speichert dann die Position und die Farbe eines Körpers. Die einzelnen Körperklassen leiten anschließend von dieser Körperbasisklasse ab, damit sie die für alle gleich bleibenden Eigenschaften nicht ständig wieder implementieren müssen. Eine solche Klassenerbfolge sieht dann in etwa folgendermaßen aus:

```
Imports System.Drawing

'Stellt die Grundeigenschaften eines Körpers bereit
Public MustInherit Class KörperBasis

    Private myFarbe As Color
    Private myPosition As Point

    MustOverride ReadOnly Property Volumen() As Double

    Public Property Farbe() As Color
        Get
            Return myFarbe
        End Get
        Set(ByVal value As Color)
            myFarbe = value
        End Set
    End Property
```

```
    Public Property Position() As Point
        Get
            Return myPosition
        End Get
        Set(ByVal value As Point)
            myPosition = value
        End Set
    End Property
End Class

Public Class Quader
    Inherits KörperBasis

    Private mySeitenLänge_a As Double
    Private mySeitenLänge_b As Double
    Private mySeitenLänge_c As Double

    Sub New(ByVal a As Double, ByVal b As Double, ByVal c As Double)
        mySeitenLänge_a = a
        mySeitenLänge_b = b
        mySeitenLänge_c = c
    End Sub

    Public Overrides ReadOnly Property Volumen() As Double
        Get
            Return mySeitenLänge_a * mySeitenLänge_b * mySeitenLänge_c
        End Get
    End Property
End Class

Public Class Pyramide
    Inherits KörperBasis

    Private myGrundfläche As Double
    Private myHöhe As Double

    Sub New(ByVal Grundfläche As Double, ByVal Höhe As Double)
        myGrundfläche = Grundfläche
        myHöhe = Höhe
    End Sub

    Public Overrides ReadOnly Property Volumen() As Double
        Get
            Return (myGrundfläche * myHöhe) / 3
        End Get
    End Property
End Class
```

Rein theoretisch könnten wir natürlich nun die bereits vorhandene `DynamicList`-Klasse für die Speicherung der `Körper`-Objekte verwenden, wie Abbildung 18.4 zeigt.

```
            .Add(New Pyramide(300, 30))
            .Add(New Quader(20, 10, 30))
        End With
        Console.WriteLine("Das Gesamtvolumen aller Körper beträgt:" & _
            locKörperliste.Gesamtvolumen)

        Co
        Co
        Co
    End Su
End Module
```

Fehlerliste

1 Fehler | 0 Warnungen | 0 Meldungen

	Beschreibung	Datei	Zeile	Spalte	Projekt
1	"Gesamtvolumen" ist kein Member von "Generics.DynamicListKörper(Of Generics.KörperBasis)".	mdlMain.vb	13	13	Generics

Fehlerliste | Ausgabe

Abbildung 18.4 Die *DynamicList* soll auch eine *Gesamtvolumen*-Eigenschaft zur Verfügung stellen, doch die ist zurzeit noch nicht implementiert

Doch dabei ergibt sich eine Kette von Problemen. Wie in Abbildung 18.4 zu sehen, möchten wir eine Eigenschaft der Liste verwenden, die es zu diesem Zeitpunkt noch nicht gibt. Und mit herkömmlichen Mitteln haben wir auch leider keine Chance, diese Eigenschaft zu implementieren. Denn innerhalb der generischen `DynamicList`-Klasse müssten wir eine Eigenschaft `Gesamtvolumen` erschaffen, die durch alle Elemente iteriert, die sie hält und deren `Volumen`-Eigenschaft abfragt. Doch genau eine solche Prozedur können wir nicht implementieren, wie Abbildung 18.5 zeigt.

```
Public ReadOnly Property Gesamtvolumen() As Double
    Get
        Dim locVolumen As Double
        For Each locItem As flexiblerDatentyp In Me
            locVolumen += locItem.
        Next
        Return locVolumen
    End Get
End Property
```

Equals
GetHashCode
GetType
ReferenceEquals
ToString

Public Overridable Functi
Bestimmt, ob das angege

Abbildung 18.5 Die Eigenschaft, die zur Berechnung des Gesamtvolumens benötigt wird, ist nicht erreichbar!

Wenn wir die Schleife zum Iterieren durch die Elemente der `DynamicList` erstellen, dann müssen wir natürlich darauf achten, dass ein einzelnes Element dieser Iteration vom gleichen Typ ist wie für die gesamte generische Klasse – und damit muss es vom Typ `flexiblerDatentyp` sein (anderenfalls wäre die Klasse nicht mehr generisch, also allgemeingültig anwendbar).

`flexiblerDatentyp` kann aber jeder beliebige Datentyp sein, und deswegen sind auch nur die Eigenschaften und Methoden erreichbar, die von jedem Datentyp *gleichermaßen* zur Verfügung gestellt werden. Das sind logischerweise genau die Eigenschaften und Methoden, die `Object` zur Verfügung stellt und die jede neue Klasse implizit erbt. Das wiederum sind die Elemente, die Ihnen IntelliSense, wie in Abbildung 18.5 zu sehen ist, anzeigt.

Das ist nun der richtige Zeitpunkt, Beschränkungen ins Spiel zu bringen. Wenn wir dem generischen Datentyp sagen, »pass auf, du darfst nur solche Datentypen als Typparameter akzeptieren, die auf `Körperbasis` basieren«, dann kann .NET Framework ohne Probleme die Methoden und Eigenschaften über `locItem` anbieten, das vom Typ `flexiblerDatentyp` ist, die durch `Körperbasis` implementiert werden.

Diese Änderungen nehmen wir als Nächstes vor, allerdings nicht in der Klasse `DynamicList` selbst, denn: Damit wir nun nicht unsere `DynamicList` für alle Zeiten nur noch auf die Verwaltung von `Körper`-Objekten limitieren, implementieren wir diese Änderungen in einer Klasse, die identisch zur `DynamicList`-Klasse ist, jedoch nur die benötigten Änderungen noch zusätzlich innehat. Diese Klasse nennen wir `DynamicListKörper`, und die sieht folgendermaßen aus (aus Platzgründen enthält dieses Listing lediglich die Änderungen im Vergleich zur Klasse `DynamicList`):

```
Class DynamicListKörper(Of flexiblerDatentyp As KörperBasis)
    Implements IEnumerable

    Protected myStep As Integer = 4          ' Schrittweite, um die das Array erhöht wird.
    Protected myCurrentArraySize As Integer  ' Aktuelle Array-Größe
    Protected myCurrentCounter As Integer    ' Zeiger auf aktuelles Element
    Protected myArray() As flexiblerDatentyp ' Array mit den Elementen.

    Sub New()
        myCurrentArraySize = myStep
        ReDim myArray(myCurrentArraySize - 1)
    End Sub

    Sub Add(ByVal Item As flexiblerDatentyp)
    .
    .
    .
    End Sub

    Public ReadOnly Property Gesamtvolumen() As Double
        Get
            Dim locVolumen As Double
            For Each locItem As flexiblerDatentyp In Me
                locVolumen += locItem.Volumen
            Next
            Return locVolumen
        End Get
    End Property
.
.
.
```

Sie können in der ersten Zeile des Klassenlistings sehen, wie Beschränkungen implementiert werden: Neben der Erweiterung (`Of flexiblerDatentyp`, die den Typparameter für die weitere Verwendung des generischen Typs in der Klasse festlegt, gibt es obendrein den Zusatz `As KörperBasis`), der nun bestimmt, dass ausschließlich Klassen, die von `KörperBasis` abgeleitet wurden, als Basis für die Erstellung des Datentyps `DynamicListKörper` dienen dürfen.

Das befähigt uns jetzt auch, die Eigenschaft `Gesamtvolumen` zu implementieren. Da wir die Datentypen für die generische Klasse auf `KörperBasis` und deren Ableitungen beschränken, weiß .NET Framework, dass es alle Methoden, die `KörperBasis` anbietet, sicher für alle Objekte zur Verfügung stellen kann, die auf `flexiblerDatentyp` basieren.

Nach der Implementierung dieser neuen Klasse stellen wir das Testprogramm im Modul *mdlMain.vb* entsprechend um:

```
Module mdlMain

    Sub Main()
        Dim locKörperliste As New DynamicListKörper(Of KörperBasis)
        With locKörperliste
            .Add(New Quader(10, 20, 30))
            .Add(New Pyramide(300, 30))
            .Add(New Pyramide(300, 30))
            .Add(New Quader(20, 10, 30))
        End With
        Console.WriteLine("Das Gesamtvolumen aller Körper beträgt:" & _
            locKörperliste.Gesamtvolumen)

        Console.WriteLine()
        Console.WriteLine("Taste drücken zum Beenden!")
        Console.ReadKey()
    End Sub

End Module
```

Übrigens: Dass sich die Klasse wirklich nur noch verwenden lässt, wenn Sie diese tatsächlich auf einer Ableitung von `KörperBasis` basieren lassen, zeigt Abbildung 18.6.

Abbildung 18.6 Eine beschränkte generische Klasse können Sie tatsächlich nur noch auf Basis eines Datentyps instanziieren, der die Beschränkung erfüllt

HINWEIS .NET Framework erlaubt es nicht, mehrere Basisklassen als Beschränkung für einen generischen Typ zu definieren. Denn das würde die Technik der Mehrfachvererbung implizieren, die .NET Framework in der vorliegenden 4.0-Version nicht beherrscht (und wahrscheinlich auch nie beherrschen wird).

Beschränkungen auf Klassen, die bestimmte Schnittstellen implementieren

Ungleich flexibler können Sie generische Klassen gestalten, wenn sich diese in ihren Beschränkungen nicht auf bestimmte Basisklassen, sondern nur auf bestimmte Schnittstellen beschränken.

Darüber hinaus haben Schnittstellen den Vorteil, sich bereits in vielen »fertigen« Typen von .NET Framework »zu befinden«, sodass Sie generische Klassen erstellen können, die nicht nur auf Ihren eigenen Typen basieren (deren Einschränkungen Sie ja gut steuern können, da Sie über deren Quellcode verfügen); vielmehr können Sie auch die Typen von .NET Framework verwenden, die sich, da sie die unterschiedlichsten Schnittstellen implementieren, ebenfalls sehr gut selektieren lassen.

Ein Beispiel: Sie möchten die `DynamicList`-Klasse um eine Sortierfunktion erweitern. Zu diesem Zweck müssen Sie die Elemente, die die `DynamicList`-Klasse speichert, miteinander vergleichen können. .NET Framework stellt für Typen, deren einzelne Instanzen sich miteinander vergleichen lassen sollen, die `IComparable`-Schnittstelle bereit. Wenn ein Typ diese Schnittstelle einbindet, zwingt ihn diese Schnittstelle damit auch, eine Funktion namens `CompareTo` einzubinden. Und wenn Sie eine generische Klasse schaffen, dann können Sie die Typen, aus denen diese hervorgehen soll, auch auf die `IComparable`-Schnittstelle beschränken. Damit bleibt die Klasse generisch und kann dennoch jeden beliebigen Datentyp typsicher speichern – jedenfalls solange er die `IComparable`-Schnittstelle selbst implementiert. *Wenn* er das allerdings macht, können Sie auch vom Vorhandensein einer `CompareTo`-Funktion sicher ausgehen und damit beispielsweise eine Sortierfunktion in der generischen Klasse implementieren. Die `IComparable`-Schnittstelle wird übrigens von allen primitiven Datentypen wie `String`, `Long`, `Decimal`, `Date` etc. eingebunden.

BEGLEITDATEIEN Ein Beispiel, das diesen Sachverhalt verdeutlichen soll, finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 18\Generics03
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Schauen wir uns die veränderte (und aus Platzgründen wieder gekürzte) Version der `DynamicList` an, die nun den Namen `DynamicListSortable` trägt und die – neben der Einschränkung bei der Definition des Klassennamens – um eine `Sort`-Methode ergänzt wurde:

```
Class DynamicListSortable(Of flexiblerDatentyp As IComparable)
    Implements IEnumerable

    Protected myStep As Integer = 4          ' Schrittweite, um die das Array erhöht wird.
    Protected myCurrentArraySize As Integer  ' Aktuelle Array-Größe
    Protected myCurrentCounter As Integer    ' Zeiger auf aktuelles Element
    Protected myArray() As flexiblerDatentyp ' Array mit den Elementen.

    Sub New()
        myCurrentArraySize = myStep
        ReDim myArray(myCurrentArraySize - 1)
    End Sub

    Sub Add(ByVal Item As flexiblerDatentyp)
    .
    .
    .
    End Sub
```

```
    'Sortiert die Elemente, die die DynamicListSortable speichert
    Public Sub Sort()
        Dim locÄußererZähler, locInnererZähler As Integer
        Dim locDelta As Integer
        Dim locItemTemp As flexiblerDatentyp

        locDelta = 1

        'Größten Wert der Distanzfolge ermitteln
        Do
            locDelta = 3 * locDelta + 1
        Loop Until locDelta > myCurrentCounter

        Do
            locDelta \= 3

            'Shellsort's Kernalgorithmus
            For locÄußererZähler = locDelta To myCurrentCounter - 1
                locItemTemp = Me.Item(locÄußererZähler)
                locInnererZähler = locÄußererZähler
                Do While (Me.Item(locInnererZähler - locDelta).CompareTo(locItemTemp) > 0)
                    Me.Item(locInnererZähler) = Me.Item(locInnererZähler - locDelta)
                    locInnererZähler = locInnererZähler - locDelta
                    If (locInnererZähler <= locDelta) Then Exit Do
                Loop
                Me.Item(locInnererZähler) = locItemTemp
            Next
        Loop Until locDelta = 0
    End Sub
```

Möglich wird die Implementierung eines Sortieralgorithmus erst wegen der Beschränkung auf Typen, die die `IComparable`-Schnittstelle einbinden. Doch da die Typen die Schnittstelle einbinden müssen (denn anderenfalls wäre gar keine Instanzerstellung der `DynamicListSortable` möglich), kann die `Sort`-Methode auch gefahrlos auf die `CompareTo`-Methode jedes Elements zurückgreifen (siehe fett hervorgehobene Zeile im oben stehenden Listing).

Da, wie schon gesagt, beispielsweise alle primitiven Datentypen in .NET `IComparable` einbinden, können wir nun diese auch mit unserer Liste verwenden, wie der Modulcode im Folgenden zeigt:

```
Module mdlMain

    Sub Main()
        Dim locDoubleList As New DynamicListSortable(Of Double)
        locDoubleList.Add(124)
        locDoubleList.Add(1243)
        locDoubleList.Add(24)
        locDoubleList.Add(14)
        locDoubleList.Add(1)
        locDoubleList.Add(-32)
        locDoubleList.Add(231)
        locDoubleList.Add(143)
```

```
        locDoubleList.Sort()
        For Each locItem As Double In locDoubleList
            Console.WriteLine(locItem)
        Next
        Console.WriteLine()

        Dim locStringList As New DynamicListSortable(Of String)
        locStringList.Add("Klaus")
        locStringList.Add("Arnold")
        locStringList.Add("Sarah")
        locStringList.Add("Christiane")
        locStringList.Add("Jürgen")
        locStringList.Add("Uta")
        locStringList.Add("Helge")
        locStringList.Add("Uwe")

        locStringList.Sort()
        For Each locItem As String In locStringList
            Console.WriteLine(locItem)
        Next

        Console.WriteLine()
        Console.WriteLine("Taste drücken zum Beenden!")
        Console.ReadKey()
    End Sub

End Module
```

Auch hier sind die Zeilen in Fettschrift die eigentlich interessanten. Sie demonstrieren einerseits, dass die generische Liste auf unterschiedlichen Datentypen basieren kann und andererseits, dass dennoch solch komplexe Funktionen wie das Sortieren funktionieren, obwohl zum Zeitpunkt der Entwicklung der Liste noch gar nicht bekannt ist, mit welchen Typen es die Liste später zu tun haben wird.

Dass dieses Konzept auch funktioniert, zeigt die Ausführung des Programms, die folgendes Ergebnis ins Konsolenfenster zaubert:

```
-32
1
14
24
124
143
231
1243

Arnold
Christiane
Helge
Jürgen
Klaus
Sarah
Uta
Uwe

Taste drücken zum Beenden!
```

HINWEIS Einen Sortieralgorithmus in eigenen Auflistungsklassen zu implementieren ist im Übrigen genau so überflüssig, wie eigene Auflistungsklassen von Grund auf neu zu kreieren. .NET Framework kennt nämlich eine Vielzahl von Auflistungsklassen für die unterschiedlichsten Zwecke. Doch es ist allemal interessant zu sehen, wie das Prinzip von Auflistungsklassen an sich funktioniert, und für das bessere Verständnis von Kapitel 22, das die wichtigsten Auflistungen in .NET Framework vorstellt, bestimmt von Vorteil.

Beschränkungen auf Klassen, die über einen Standardkonstruktor verfügen

In einigen Fällen ist es notwendig, dass eine generische Klasse in der Lage ist, den Typ, auf dem sie basieren soll, auch zu instanziieren. Das kann sie dann nicht, wenn es sich bei dem Typ, auf dem sie basiert, um eine abstrakte Klasse handelt, um eine Schnittstelle oder um eine Klasse, die ausschließlich über parametrisierte Konstruktoren verfügt.

Wenn Sie diese Fälle für den Typ ausschließen möchten, den eine generische Klasse einbindet, müssen Sie eine Beschränkung definieren, die vom Typ, auf dem sie basieren soll, einen Standardkonstruktor erfordert, und das geht folgendermaßen:

```
Public Class GenerischeKlasseMitInstanziierbaremTyp(Of flexiblerDatentyp As New)

    Public Sub TestMethode()
       'Das geht nur auf Grund der angegebenen Beschränkung:
        Dim locTest As New flexiblerDatentyp

        'Und hier ist locTest jetzt als Datentyp instanziiert!
        Console.WriteLine(locTest.ToString)
    End Sub
End Class
```

Beschränkungen auf Wertetypen

Möchten Sie eine generische Klasse auf Wertetypen beschränken, verfahren Sie auf ähnliche Weise, wie im vorherigen Abschnitt beschrieben. Sie bestimmen durch `As Structure`, dass nur noch Wertetypen (in Visual Basic also Strukturen) innerhalb einer generischen Klasse als Typ zur Anwendung kommen dürfen, die auf `ValueType` basieren. Ein Beispiel:

```
Public Class GenerischeKlasseNurMitWertetypen(Of flexiblerDatentyp As Structure)

    Public Sub TestMethode()
        'Ist Wertetyp – keine Instanziierung durch New erforderlich!
        Dim locWerteTyp As flexiblerDatentyp

        'Und hier ist locTest jetzt als Datentyp instanziiert.
        Console.WriteLine(locWerteTyp.ToString)
    End Sub
End Class
```

Kombinieren von Beschränkungen und Bestimmen mehrerer Typparameter

In Visual Basic sind alle Beschränkungen für generische Datentypen untereinander kombinierbar. Im Gegensatz zu Beschränkungen bei Basisdatentypen können Sie darüber hinaus auch Beschränkungen für mehrere Schnittstellen bestimmen. Wenn Sie verschiedene Beschränkungen oder mehrere Schnittstellen für einen generischen Datentyp einrichten, fassen Sie die verschiedenen Vorschriften in geschweiften Klammern zusammen. Ein Beispiel soll auch diesen Sachverhalt verdeutlichen:

```
Public Class GenerischeBeschränkungskombi(Of flexiblerDatentyp As {Structure, IComparable, IDisposable})

    Public Sub TestMethode()
        Dim locWerteTyp As flexiblerDatentyp
        Dim locWerteTyp2 As flexiblerDatentyp

        'Direkt verwendbar, da Wertetyp durch Struktur
        'Vergleichbar, dank IComparable
        locWerteTyp.CompareTo(locWerteTyp2)

        'Disposable, dank IDisposable
        locWerteTyp.Dispose()
        locWerteTyp2.Dispose()
    End Sub
End Class
```

Zusätzlich können Sie eine generische Klasse auch für die Verwendung von mehreren Typparametern einrichten. Falls Sie beispielsweise eine Auflistung entwickeln möchten, die als ein Wörterbuch fungiert, dann benötigen Sie einen Typ zum Nachschlagen (den Schlüssel) und einen für den eigentlichen Wert. (Programmieren Sie das aber nicht selbst, denn auch das gibt es schon beispielsweise mit der generischen `KeyedCollection`-Klasse, die sich im `System.Collections.ObjectModel`-Namespace befindet.) Die Einschränkungen lassen sich dann für jeden Typparameter einzeln festlegen:

```
Public Class GenerischesWörterbuch(Of Schlüsseltyp As {Structure, IComparable}, _
                        Werttyp As {New, IComparable, IDisposable})

    Public Sub TestMethode()
        Dim locWerteTyp As Schlüsseltyp
        Dim locWerteTyp2 As Werttyp

        'Direkt verwendbar, da Wertetyp durch Struktur
        'Vergleichbar, dank IComparable
        locWerteTyp.CompareTo(locWerteTyp2)

        'Disposable, dank IDisposable
        locWerteTyp2.Dispose()
    End Sub
End Class
```

Kapitel 19

Typen für Fortgeschrittene

In diesem Kapitel:

Nullable Wertetypen 580
Generische Delegaten 586
Tuple(Of T) 590
Typvarianz 591
Erweiterungsmethoden 595

Seit Visual Basic 2005 gibt es Generics – das vorherige Kapitel hat Ihnen das Konzept von Generics ausführlich dargelegt. Mit Generics führte damals .NET Framework 2.0 einen sehr wichtigen Datentyp ein, der auf dieser Technik basierte – die *Nullable*-Wertetypen, die mit .NET Framework 3.5 bei Visual Basic 2008 sogar fest in der Sprache verankert wurden. Sie haben Nullables in der Einführung dieses Buchs kurz kennen gelernt – dieses Kapitel nimmt sie ein wenig genauer unter die Lupe.

Delegaten kommen ab Visual Basic 2010 eine erweiterte Bedeutung zu, denn Visual Basic 2010 führt die so genannten mehrzeiligen Anweisungs-Lambdas ein, die Sie in Kapitel 16 kennen gelernt haben. Die Weise, wie sich Problemstellungen lösen lassen, ändert sich dabei eklatant – man könnte fast schon von einem Paradigmenwechsel im Aufbau von Programmcode oder beim Strukturieren von Programmen sprechen. Mit Visual Basic 2010 und .NET Framework 4.0 gibt es für diesen Zweck einige neuen Klassen, die Sie bestimmt schon bald täglich verwenden werden, denn sie erleichtern Ihnen das Entwickeln in den unterschiedlichsten Szenarien enorm – das gilt umso mehr mit der immer wichtiger werdenden Multithreading-Programmierung, die ganz besonders von Lambdas und Delegaten profitiert. Die Rede hier ist von den Klassen `Action(Of T)` und `Function(Of T)`; in diesem Kapitel finden Sie sie im Abschnitt »Generische Delegaten« ab Seite 586 beschrieben.

Neben den beschriebenen Klassen erfahren Sie in diesem Kapitel auch noch Weiteres zum Thema Typen, im Anspruch vielleicht ein wenig höher angesiedelt, als Sie es bislang gewohnt waren.

Nullable Wertetypen

Nullable-Typen gibt es schon seit Visual Basic 2005, aber dort waren sie nur – sagen wir einmal – halbherzig in Visual Basic implementiert. Zwar erfuhren sie schon eine (notwendige!) Sonderbehandlung durch die CLR (siehe Abschnitt »Besonderheiten bei Nullable beim Boxen« ab Seite 583), aber anders als in C# waren sie noch nicht mit einem eigenen Typliteral implementiert. Das hat sich in Visual Basic 2008 geändert.

`Nullable` ist ein generischer Datentyp mit einer Einschränkung auf Wertetypen. Er ermöglicht, dass ein beliebiger Wertetyp neben seinem eigentlichen Wert einen weiteren Zustand »speichern« kann – nämlich `Nothing`.

Und wozu braucht man das? Beispielsweise in der Datenbankprogrammierung. Wenn Sie bereits Erfahrungen in der Datenbankprogrammierung haben, wissen Sie auch sicherlich, dass Ihre Datenbanktabellen über Datenfelder verfügen können, die den »Wert« `Null` »speichern« können – als Zeichen dafür, dass eben *nichts* (auch nicht die Zahl 0) in diesem Feld gespeichert wurde.

Ein anderes Beispiel sind `CheckBox`-Steuerelemente in Windows Forms-Anwendungen: Sie verfügen über einen Zwischenzustand, der den Zustand »nicht definiert« anzeigen soll. Eine einfache boolesche Variable könnte alle möglichen Zustände nicht aufnehmen – `True` und `False` sind dafür einfach zu wenig. Anders ist es, wenn Sie eine Variable vom Typ `Boolean` definieren könnten, die auch den »Nichts ist gespeichert«-Wert widerspiegeln könnte.

Abbildung 19.1 Ein `Boolean` eignet sich beispielsweise dazu, auch Zwischenzustände eines CheckBox-Steuerelements– wie hier im Bild zu sehen – zu speichern

Und das geht: Seit Visual Basic 2008 definiert man eine Variable als Nullable-Wertetyp, indem man ihrem Variablennamen ein Fragezeichen als Typliteral anhängt. Das sieht entweder so …

```
Private myCheckBoxZustand As Boolean?
```

… oder so aus:

```
Private myCheckBoxZustand? As Boolean
```

Alternativ können Sie die Definition eines Nullables immer noch so vornehmen, wie Sie es in Visual Basic 2005 sogar noch mussten:

```
Private myCheckBoxZustand As Nullable(Of Boolean)
```

Und keine Angst: Sie müssen sich jetzt nicht durch Hunderte, vielleicht Tausende Zeilen 2005er-Code hangeln und für nachfolgende Basic-Versionen die entsprechenden Änderungen mit dem Fragezeichen vornehmen. Bei allen drei Varianten kommt nämlich exakt die gleiche Variante heraus.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 19\NullableUndCheckbox
```

Dieses Beispiel demonstriert, wie alle Zustände eines `CheckBox`-Steuerelements, dessen `ThreeState`-Eigenschaft zur Anzeige aller *drei* Zustände auf `True` gesetzt wurde, in einer Feldvariablen vom Typ `Boolean?` gespeichert werden können. Klicken Sie auf *Zustand speichern,* um den Zustand des `CheckBox`-Steuerelements in der Feldvariablen zu sichern, verändern Sie anschließend den Zustand, und stellen Sie den ursprünglichen Zustand des `CheckBox`-Steuerelements mit der entsprechenden Schaltfläche wieder her. Der entsprechende Code dazu lautet folgendermaßen:

```
Public Class Form1

    Private myCheckBoxZustand As Boolean?

    'Das ginge auch:
    'Private myCheckBoxZustand? As Boolean

    'Und das auch:
    'Private myCheckBoxZustand As Nullable(Of Boolean)
```

```
    Private Sub btnOK_Click(ByVal sender As System.Object, _
                            ByVal e As System.EventArgs) Handles btnOK.Click
        Me.Close()
    End Sub

    Private Sub btnSpeichern_Click(ByVal sender As System.Object, _
                                   ByVal e As System.EventArgs) Handles btnSpeichern.Click

        If chkDemo.CheckState = CheckState.Indeterminate Then
            myCheckBoxZustand = Nothing
        Else
            myCheckBoxZustand = chkDemo.Checked
        End If
    End Sub

    Private Sub btnWiederherstellen_Click(ByVal sender As System.Object, _
                                  ByVal e As System.EventArgs) Handles btnWiederherstellen.Click
        If Not myCheckBoxZustand.HasValue Then
            chkDemo.CheckState = CheckState.Indeterminate
        Else
            If myCheckBoxZustand.Value Then
                chkDemo.CheckState = CheckState.Checked
            Else
                chkDemo.CheckState = CheckState.Unchecked
            End If
        End If
    End Sub
End Class
```

Die Zeilen, in denen die Feldvariable `Boolean?` zum Einsatz kommt, sind im Listing fett markiert. Dabei fällt Folgendes auf:

- **Wertzuweisung** Wenn Sie einen Wert des zugrunde liegenden Typs an `type?` zuweisen wollen, können Sie die implizite Konvertierung verwenden, den entsprechenden Wert also direkt zuweisen, etwa wie in der folgenden Zeile zu sehen:

  ```
  myCheckBoxZustand = chkDemo.Checked
  ```

- **Auf `Nothing` zurücksetzen** Möchten Sie eine `Nullable`-Instanz auf `Nothing` zurücksetzen, weisen Sie ihr einfach den »Wert« `Nothing` zu – wie im Listing an dieser Stelle zu sehen:

  ```
  myCheckBoxZustand = Nothing
  ```

- **Auf Wert prüfen** Möchten Sie wissen, ob eine `Nullable`-Instanz einen Wert oder `Nothing` enthält, verwenden Sie deren Eigenschaft `HasValue`. Auch dafür gibt es ein Beispiel im Listing:

  ```
  If Not myCheckBoxZustand.HasValue Then
      chkDemo.CheckState = CheckState.Indeterminate
  Else
  .
  .
  .
  ```

- **Wert des Basistyps abrufen** Und schließlich müssen Sie natürlich auch den Wert, den eine `Nullable`-Instanz trägt, wenn sie *nicht* `Nothing` ist, ermitteln können. Dazu dient die Eigenschaft `Value`. Ein Beispiel dafür:

```
If myCheckBoxZustand.Value Then
    chkDemo.CheckState = CheckState.Checked
Else
    chkDemo.CheckState = CheckState.Unchecked
End If
.
.
```

Alternativ rufen Sie den Wert auch ohne die `Value`-Eigenschaft ab, wobei Sie natürlich dann immer davon ausgehen müssen, dass die Variable `Nothing` enthält, Sie sie also wieder nur einer Variable vom gleichen Typ (nämlich Nullable des Basistyps) zuweisen können.

HINWEIS Erst in diesem Beispiel fiel auf, dass man offensichtlich ein `CheckBox`-Steuerelement, dessen `ThreeState`-Eigenschaft gesetzt ist und das den `Indeterminate`-Zustand (also »unbestimmt«) momentan trägt, nicht mit seiner `Checked`-Eigenschaft in einen anderen Zustand versetzen kann (`Checked` oder `Unchecked`). Sie können in diesem Fall nur die `CheckState`-Eigenschaft verwenden, um das `CheckBox`-Steuerelement programmgesteuert wieder aus dem `Indeterminate`-Zustand herauszuholen!

Aufpassen beim Umgang mit booleschen Ausdrücken auf Basis von Nullables!

Besondere Vorsicht ist geboten, wenn Sie mit booleschen Ausdrücken auf Basis von Nullables umgehen – im Grunde genommen sollten Sie versuchen, dies zu vermeiden. Das Problem ist, dass Sie bei normalen booleschen Ausdrücken annehmen können, dass wenn ein Wert nicht `False` ist, dieser automatisch `True` sein muss. Es gibt keine Abweichung von dieser Regel.

Das ist bei Nullable-Booleans jedoch durchaus anders: Wenn ein Wert vom Typ `Boolean?` den »Wert« `Nothing` hat, dann ist er nicht *falsch* und auch nicht *true*. Da wir es aber aufgrund unserer Erfahrung gewohnt sind, dass wenn der Ausdruck `Not booleanValue` als `False` ausgewertet wird, ihn im Umkehrschluss als `True` anzusehen, schleichen sich bei solchen Konstrukten schnell Fehler ein. Das folgende Beispiel zeigt dies im Detail:

```
Dim regularBoolean As Boolean = False
If regularBoolean Then
    'In diesem Fall ist, wenn der Wert nicht wahr ist, ...
    Console.WriteLine("Wert ist wahr!")
Else
    '... er notwendigerweise falsch.
    Console.WriteLine("Wert ist falsch!")
End If


'Jetzt ist der Wert ein nullable-boolean und kann auch Nothing sein.
Dim booleanNullable As Boolean? = Nothing
```

```
        If booleanNullable Then
            'Wenn also der Wert nicht wahr ist ...
            MessageBox.Show("Wert ist wahr!")
        Else
            '...dann muss er NICHT notwendigerweise falsch sein!
            MessageBox.Show("Wert ist nicht unbedingt falsch!")
            If Not booleanNullable Then
                'Nur hier wissen wir, dass er falsch war.
                MessageBox.Show("Wert ist mit Sicherheit falsch!")
            End If
        End If
```

Besonderheiten bei Nullable beim Boxen

Der Datentyp `Nullable (type?)` ist das, was man in Visual Basic als Struktur programmieren würde, also ein Wertetyp. Doch Sie könnten diesen Wertetyp nicht eins zu eins nachprogrammieren, denn er erfährt durch die Common Language Runtime eine besondere Behandlung – und das ist auch gut so.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 19\NullableDemo
```

Wenn Sie eine Instanz einer beliebigen Struktur – also eines beliebigen Wertetyps – verarbeiten, kommt irgendwann der Zeitpunkt, an dem Sie diesen Wertetyp in einer Objektvariablen boxen müssen – beispielsweise wenn Sie ihn als Bestandteil eines Arrays oder einer Auflistung (Collection) speichern.

Wann immer Sie einen definierten Wertetyp in einem Objekt boxen, kann dieses Objekt logischerweise nicht `Nothing` sein, ganz egal, welchen »Wert« diese Struktur hat. Im Falle des `Nullable`-Typs ist das anders, wie das folgende Beispiel zeigt:

```
Module NullableDemo

    Sub Main()
        Dim anObj As Object
        Dim aNullOfInt As Integer? = Nothing

        'Es gibt natürlich eine verwendbare Instanz, denn
        'Integer? ist ein Wertetyp!
        Console.WriteLine("Hat aNullOfInt einen Wert: " & aNullOfInt.HasValue)

        'Und dennoch ergibt das folgende Konstrukt True,
        'als würde locObj keine Referenz haben!
        anObj = aNullOfInt
        Console.WriteLine("Ist anObj Nothing? " & (anObj Is Nothing).ToString)
        Console.WriteLine()

        'Und auch das "Entboxen" geht!
        'Es gibt keine Null-Exception!
        aNullOfInt = DirectCast(anObj, Integer?)
```

```
        'Und geht das dann auch? - Natürlich!
        aNullOfInt = DirectCast(Nothing, Integer?)

        'Und noch weiter. Wir boxen einen Integer?
        aNullOfInt = 10
        '
        anObj = aNullOfInt
        Dim anInt As Integer = DirectCast(anObj, Integer)

        Console.WriteLine("Taste drücken zum Beenden!")
        Console.ReadKey()

        'Das geht übrigens nicht, obwohl Nullable die
        'Constraints-Einschränkung im Grunde genommen erfüllt!
        'Dim anotherNullOfInt As Nullable(Of Integer?)

    End Sub
End Module
```

Wenn Sie dieses Beispiel ausführen, gibt es die folgenden Zeilen aus:

```
Hat aNullOfInt einen Wert: False
Ist anObj Nothing? True

Taste drücken zum Beenden!
```

Das, was hier passiert, ist beileibe keine Selbstverständlichkeit – aber dennoch sauberes Design der CLR, denn: Zwar wird `aNullOfInt` nicht initialisiert (oder, um es in diesem Beispiel deutlich zu machen, mit `Nothing` – aber das kommt auf dasselbe raus), aber natürlich existiert dennoch eine Instanz der Struktur. Sie spiegelt eben nur den Wert `Nothing` wider. Gemäß den bekannten Regeln müsste das anschließende Boxen in der Variablen `anObj` auch ergeben, dass `anObj` einen Zeiger auf eine Instanz der `Nothing` widerspiegelnden Variable `aNullOfInt` enthält und keinen Null-Zeiger. Doch das ist nicht der Fall, denn die anschließende Ausgabe von

```
Console.WriteLine("Ist anObj Nothing? " & (anObj Is Nothing).ToString)
```

zeigt Folgendes auf dem Bildschirm an:

```
Ist anObj Nothing? True
```

Das »Zurückcasten« von `Nothing` in `Nullable` ist damit natürlich genauso gestattet, wie ebenfalls im Listing zu sehen. Und noch eine Unregelmäßigkeit erfahren `Nullable`-Datentypen, nämlich wenn es darum geht, einen geboxten Typ (vorausgesetzt, er ist eben nicht `Nothing`) in seinen Grundtyp zurückzucasten, wie der folgende Codeausschnitt zeigt:

```
        'Und noch weiter. Wir boxen einen Integer?
        aNullOfInt = 10
        '
        anObj = aNullOfInt
        Dim anInt As Integer = DirectCast(anObj, Integer)
```

Hier wird ein `Nullable`-Datentyp in einem Objekt geboxt, aber später zurück in seinen *Grunddatentypen* gewandelt. Ein, wie ich finde, logisches Design, was allerdings dem »normalen« Vorgehen beim Boxen von Wertetypen in Objekten völlig widerspricht.

Unterschied zwischen Nothing und Nothing als Standardwert

In diesem Zusammenhang ist es wichtig, sich folgendes Verhalten nochmal zu veranschaulichen, das auf den ersten Blick inkonsistent wirkt (Abbildung 19.2).

Abbildung 19.2 `Nothing` lässt sich einer `Integer`-Variablen direkt zuweisen, `Nothing` aus einem `Nullable`-Integer aber nicht!

Hier stellt sich die Frage: Wieso lässt sich `Nothing` einer `Integer`-Variablen direkt zuweisen, aber nicht das `Nothing`, was in einem Nullable-Integer »gespeichert« ist? (Was deswegen in Anführungszeichen steht, weil eben nichts gespeichert ist, *weshalb* die Abfrage des Inhalts ja eben `Nothing` ergibt).

Im ersten Fall handelt es sich bei `Nothing` um den *Standardwert* des entsprechenden Datentyps. Wenn Sie `Nothing` einer Variablen zuweisen, dann bekommt diese immer ihren Standardwert zugewiesen. Wenn Sie `Nothing` einem Typ zuweisen, ist dies nur bei Referenztypen quasi zufällig ein 0-Referenz-Zeiger, in Wahrheit wird auch hier der Standardwert des Typs zugewiesen – bei Referenztypen eben ein 0-Referenz-Zeiger. Bei numerischen Datentypen ist es der Wert 0.

Im zweiten Fall ist es anders, denn hier gibt es tatsächlich keinen Inhalt; und ein nicht vorhandener Inhalt ist eben etwas anderes als der Standardwert des Typs – deswegen gibt Visual Basic zurecht zur Laufzeit eine entsprechende Fehlermeldung aus, denn einem Wertetyp kann man nicht *nichts* zuweisen.

Generische Delegaten

Mit der Einführung von mehrzeiligen Anweisungs-Lambdas wird unser Programmcode in Zukunft deutlich anders ausschauen. Mehrzeilige Anweisungs-Lambdas haben beispielsweise den Vorteil, ereignisbehandelnden Code an die Stelle zu schreiben, an der das Ereignis gebunden wird. Ein Beispiel soll das verdeutlichen:

```
Public Class Form1

    Sub New()

        ' Dieser Aufruf ist für den Designer erforderlich.
        InitializeComponent()

        Dim MessageText As String = "Button1 wurde geklickt"

        ' Fügen Sie Initialisierungen nach dem InitializeComponent()-Aufruf hinzu.
        AddHandler Button1.Click, Sub(sender As Object, e As EventArgs)
                                      MessageBox.Show(MessageText)
                                  End Sub
    End Sub

End Class
```

Sie sehen hier, dass schon im Konstruktor des Formulars das `Click`-Ereignis kodiert ist und das mehrzeilige Lambda direkt das Argument für `AddHandler` ist an der Stelle, an der normalerweise mit dem `AddressOf`-Operator auf eine signaturkompatible Methode verwiesen wird.

Der Vorteil dieser Art der Codegestaltung ist, dass lokale Variablen, die Sie innerhalb der Methode verwenden, die das Lambda enthalten, innerhalb des Lambdas verwendet werden können – im Beispiel die lokale Variable `MessageText`, die den Ausgabetext für `MessageBox` enthält, die angezeigt wird, wenn der Benutzer die Schaltfläche betätigt. Und dass das erlaubt ist, ist gar nicht selbstverständlich, denn der Zeitpunkt der Ausführung des Lambdas ist natürlich ein völlig anderer als der Zeitpunkt der Definition der lokalen Variablen.

Die direkte Angabe eines Lambdas anstelle eines Delegaten in der Form, wie Sie sie oben im Beispiel sehen, ist schon eine angenehme Sache, wie ich finde. Doch es geht noch besser, nämlich wenn Sie Lambdas Delegatenvariablen einer bestimmten Signatur zuordnen können und sie damit austauschbar machen.

Action(Of T)

Mit dem generischen Delegaten `Action(Of T)` können Sie einen neuen Delegaten auf dessen Basis erstellen, der kompatibel zu bestimmten Methodensignaturen ist, die keine Rückgabewerte haben. Angenommen, Sie haben eine Methode, die Sie einem Delegaten zuweisen möchten und welche die folgende Signatur aufweist:

```
    Private Sub NachtClickHandler(ByVal sender As Object, ByVal e As EventArgs)
        MessageBox.Show("Gute Nacht, lieber Anweder!" &
                        vbNewLine & "Der Button wurde gedrückt")

    End Sub
```

Dann erstellen Sie entweder einen neuen Delegaten wie in Kapitel 16 beschrieben, oder Sie vereinfachen sich die Sache und verwenden den generischen Delegaten `Action(Of T)`, wies es in Abbildung 19.3 zu sehen ist.

```
'Definiert einen Delegaten auf eine Methode, die einen Parameter vom
'Typ Object und einen vom Typ EventArgs entgegen nimmt.
Private myClickHandler As Action(Of Object, EventArgs)

Private Sub NachtClickHandler(ByVal sender As Object, ByVal e As EventArgs)
    MessageBox.Show("Gute Nacht, lieber Anweder!" &
                    vbNewLine & "Der Button wurde gedrückt")

End Sub
```

Abbildung 19.3 Die Typparameter der **`Action`**-Klasse müssen in der Reihenfolge den Typen der Methode entsprechen, die eine Instanz der Action-Klasse als Delegat aufnehmen soll

Wie die Abbildung zeigt, wird durch die Deklaration von `myClickHandler` als `Action(Of Object, EventArgs)` dieser signaturkompatibel zur Methode `NachtClickHandler`. Hätte die Methode (`NachtClickHandler`), deren Adresse in `myClickHandler` gespeichert werden soll, einen weiteren Parameter beispielsweise vom Typ `String`, dann würden Sie der `Action(Of T)`-Klasse auch einen weiteren Typparameter mitgeben, und die Definition sähe folgendermaßen aus:

```
Private myClickHandler As Action(Of Object, EventArgs, String)
```

HINWEIS Der `Action(Of T)`-Delegat funktioniert in Verbindung mit Methoden im Bereich von 0 bis zu 16 Parametern. Sollten Sie wirklich einen Delegaten für eine Methodensignatur mit mehr als 16 Parametern benötigen, müssen Sie einen eigenen Delegatentyp definieren.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 19\GenericDelegates
```

Ab diesem Zeitpunkt arbeitet `myClickHandler` wie jede andere Delegatenvariable auch. Das folgende Beispiel demonstriert die Einsatzmöglichkeiten auch mit mehrzeiligen Anweisungs-Lambdas:

```
Public Class Form1

    'Definiert einen Delegaten auf eine Methode, die einen Parameter vom
    'Typ Object und einen vom Typ EventArgs entgegennimmt.
    Private myClickHandler As Action(Of Object, EventArgs)

    'Ein EventHandler, der aufgerufen wird, wenn die aktuelle Uhrzeit nach 18:00 liegt.
    Private Sub NachtClickHandler(ByVal sender As Object, ByVal e As EventArgs)
        MessageBox.Show("Gute Nacht, lieber Anweder!" &
                        vbNewLine & "Der Button wurde gedrückt")

    End Sub
```

```
    Sub New()

        ' Dieser Aufruf ist für den Designer erforderlich.
        InitializeComponent()

        Dim MessageText As String = "Button1 wurde geklickt"

        'Vor 12 --> dann dieser Lambda als Delegate
        If Now.TimeOfDay < New TimeSpan(12, 0, 0) Then
            'Beachtenswert: die Typen von sender und e leiten
            'sich aus der Signatur von myClickHandler ab (Typrückschluss).
            ' 'as Object' und 'as EventArgs' brauchen Sie also nicht mit anzugeben.
            myClickHandler = Sub(sender, e)
                                MessageBox.Show("Guten Morgen lieber Anwender!" &
                                                vbNewLine & MessageText)
                            End Sub

            'Vor 18 Uhr, dann dieser Lambda als Delegate
        ElseIf Now.TimeOfDay < New TimeSpan(18, 0, 0) Then
            'auch hier: Typrückschluss für Parameter des Lambdas.
            myClickHandler = Sub(sender, e)
                                MessageBox.Show("Guten Tag lieber Anwender!" &
                                                vbNewLine & MessageText)
                            End Sub
        Else
            'Sonst die NachtClickHandler Sub dem Action-Delegate zuweisen.
            myClickHandler = AddressOf NachtClickHandler
        End If


        'Hier wird das Click-Ereignis verdrahtet, das seinerseits
        'den Action-Delegaten aufruft.
        AddHandler Button1.Click, Sub(sender, e)
                                      myClickHandler(sender, e)
                                  End Sub
    End Sub

End Class
```

Function(Of T)

Die `Action(Of T)`-Klasse funktioniert prima für Methoden, die keine Rückgabewerte an die aufrufende Instanz zurückliefern. Für Methoden, die dagegen Rückgabewerte zurückgeben, verwenden Sie den generischen Delegaten `Function(Of T)`. Dieser nimmt im Vergleich zu `Action(Of T)` immer einen Typparameter mehr entgegen, der dann den Typ des Rückgabewerts darstellt. Davon abgesehen funktioniert `Function(Of T)` genau wie `Action(Of T)`. Abbildung 19.4 demonstriert, wie Sie eine Delegatenvariable vom Typ `Function(Of T)` signaturkompatibel zu einer Methode mit Rückgabewert machen.

```
Private myFunction As Func(Of Integer, Double, String) = AddressOf SomeFunction

Private Function SomeFunction(ByVal par1 As Integer, ByVal par2 As Double) As String
    Return "Result"
End Function
```

Abbildung 19.4 Die Typparameter der Func-Klasse müssen in der Reihenfolge den Typen der Methode entsprechen, die eine Instanz der **Func**-Klasse als Delegat aufnehmen soll. Der letzte Typparameter von Func entspricht dem Rückgabetyp der Methode.

Tuple(Of T)

Bei Methoden, die Funktionsergebnisse zurückgeben, stellt sich in bestimmten Situationen ein Problem, das in der Regel einiges an Zusatzaufwand verursacht. Was ist zu tun, wenn mehr als ein Wert als Ergebnis von der Funktion zurückgegeben werden soll? Bislang gab es zwei Möglichkeiten, dieses Problem zu umgehen:

- Sie arbeiten mit der Übergabe von Parametern als Referenz, bei der Sie die übergebene Variable auch als Rückgabewert »missbrauchen«, ähnlich wie es beispielsweise die `TryParse`-Methoden vieler primitiver Datentypen macht, um herauszufinden, ob eine Zeichenkette in einen primitiven Datentyp umgewandelt werden kann oder nicht
- Sie haben eine Hilfsklasse oder Hilfsstruktur erstellt, die die Parameter, die zurückgegeben werden, zusammenfasst

Mit `Tuple(Of T)` tut sich eine dritte Möglichkeit auf. Die generische `Tuple`-Klasse vereinigt so viele Werte unter einem Hut, wie Sie Typparameter (maximal 8) bei der Deklaration der Variable angeben. Werte für das `Tuple` lassen sich ausschließlich bei der Instanziierung der Klasse im Konstruktor spezifizieren.

Wenn Sie die Werte, die das `Tuple` speichert, auslesen möchten, bedienen Sie sich der entsprechenden *Item*-Eigenschaften, die für jeden angegebenen Typparameter durchnummeriert als entsprechender Typ vorliegen – allerdings nicht als Index, sondern als feste Namen (`Item1`, `Item2`, `Item3` …). Das folgende Beispiel demonstriert den Umgang mit der `Tuple`-Klasse:

```
Sub TupleTest()

    'Definiert ein neues Tuple mit drei Werten.
    Dim t As New Tuple(Of Double, Integer, String)(3.1415926, 20, "Test")
    Debug.Print("Erster Wert des Tupels:" & t.Item1.ToString)
    Debug.Print("Zweiter Wert des Tupels:" & t.Item2.ToString)
    Debug.Print("Dritter Wert des Tupels:" & t.Item3)

    t = TupelFunktion
End Sub

Function TupelFunktion() As Tuple(Of Double, Integer, String)
    'Diese Funktion liefert drei Werte in einem Tuple gekapselt zurück.
    Return New Tuple(Of Double, Integer, String)(3.141592657, 20, "String")
End Function
```

Typvarianz

Unter Typvarianz (*Type Variance*) versteht man im Zusammenhang mit der objektorientierten Programmierung, dass eine Objektvariable eines bestimmten Typs auch auf eine Instanz einer Ableitung dieses Typs zeigen kann (Kovarianz) bzw. eine Objektvariable vom Typ einer abgeleiteten Klasse auf die Instanz einer Basisklasse (Kontravarianz).

Ein einfaches Beispiel von Kovarianz sehen wir immer dann, wenn wir uns die Polymorphie bei überschriebenen Methoden einer Basisklasse zunutze machen. Wenn eine ListBox beispielsweise in ihrer `Items`-Auflistung Variablen vom Typ `Object` erwartet, können Sie hier natürlich auch von `Object` abgeleitete Typen (also jeden anderen) Typen übergeben. `Object` verhält sich dabei *kovariant*, und damit ist dieses Beispiel ein einfaches für Kovarianz.

Bei Auflistungen stellt sich jedoch ein Problem. Angenommen, Sie haben einen Basistyp namens `Basisklasse` und einen davon abgeleiteten Typ namens `AbgeleiteteKlasse`. Dann können wir, soviel wissen wir schon, Basistypen in vielen Fällen kovariant verwenden.

HINWEIS Für das Verständnis dieses Abschnitts ist es wichtig, dass Sie sich sowohl ausreichend mit dem Thema Generics beschäftigt haben (siehe Kapitel 18) als auch mit dem Thema Auflistungen (Kapitel 20). Kapitel 12 und insbesondere die Funktionsweise von Polymorphie und Grundkenntnisse von dem in Kapitel 13 Besprochenen helfen ebenfalls bei diesem anspruchsvollen Thema.

Wenn sich aber nun `BasisTyp` zu `AbgeleiteterTyp` kovariant verhalten kann, sollte dann nicht `BasisTyp` in `List(Of BasisTyp)` ebenfalls kovariant zu `List(Of AbgeleiteterTyp)` sein, sodass folgende Zeile denkbar wäre?

```
    Dim basisListe As List(Of AdressenBasis)
    Dim abgeleiteteListe As New List(Of Mitarbeiter)

    basisListe = abgeleiteteListe
```

Die Antwort lautet nein – und Abbildung 19.5 verrät dann auch sogleich, worin Abhilfe liegen könnte, wenngleich sie keine Erklärung dafür parat hält, *wieso* diese Zuweisung verboten ist.

Abbildung 19.5 Obwohl `basisListe` für sich kovariant in Bezug auf beispielsweise eine Instanz von `Mitarbeiter` verwendet werden kann, weil `Mitarbeiter` von `AdressenBasis` abgeleitet ist, sind Listen aus `List(Of T)` mit entsprechenden Typparametern nicht automatisch kovariant

Lassen Sie uns dieses Problem noch mal im Kontext eines Beispiels aus der Praxis zeigen:

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 19\CoContraVariance
```

Wir haben hier ein nettes kleines Programm, das im Moment durch eine hauptsächliche Eigenart hervorsticht: Es läuft nämlich nicht. Der Grund ist folgender: Es sind zwei Listen vorhanden, einmal mit Mitarbeitern und einmal mit Lieferanten.

```
'Mitarbeiter mithilfe von Auflistungs-Initialisierern füllen
Dim mitarbeiter As New List(Of Mitarbeiter) From {
        New Mitarbeiter With {.IDMitarbeiter = 1, .Nachname = "Leenings", .Vorname = "Ramona"},
        New Mitarbeiter With {.IDMitarbeiter = 3, .Nachname = "Calla", .Vorname = "Sarika"},
        New Mitarbeiter With {.IDMitarbeiter = 2, .Nachname = "Löffelmann", .Vorname = "Klaus"},
        New Mitarbeiter With {.IDMitarbeiter = 3, .Nachname = "Pusch", .Vorname = "Marcus"},
        New Mitarbeiter With {.IDMitarbeiter = 2, .Nachname = "Belke", .Vorname = "Andreas"}}

Dim lieferanten As New List(Of Lieferant) From {
        New Lieferant With {.IDLieferant = 1, .Nachname = "Woerrlein", .Vorname = "Hartmut"},
        New Lieferant With {.IDLieferant = 2, .Nachname = "Löffelmann", .Vorname = "Klaus"},
        New Lieferant With {.IDLieferant = 3, .Nachname = "Grauthoff", .Vorname = "Kristin"}}
```

Beide dieser Ausgangsklassen basieren auf einer gemeinsamen Basisklasse, wie man hier unten sieht, nämlich der Klasse `AdressenBasis`:

```
Public Class AdressenBasis

    Property Vorname As String
    Property Nachname As String

End Class

Public Class Mitarbeiter
    Inherits AdressenBasis
    Implements IComparable(Of Mitarbeiter)

    Property IDMitarbeiter As Integer

    Public Function CompareTo(ByVal other As Mitarbeiter) As Integer _
        Implements System.IComparable(Of Mitarbeiter).CompareTo
        Return (Me.Nachname & Me.Vorname).CompareTo(other.Nachname & other.Vorname)
    End Function

End Class

Public Class Lieferant
    Inherits AdressenBasis

    Property IDLieferant As Integer
End Class
```

Jetzt hatte der Entwickler die Idee – Entwickler sind schließlich bequem –, eine *gemeinsame* Methode zu schreiben, mit der er die Lieferanten oder die Mitarbeiter anzeigen lassen kann:

```
    Sub ListeAusgeben(ByVal adressenListe As List(Of AdressenBasis))
        For Each item In adressenListe
            Debug.Print(item.Nachname & ", " & item.Vorname)
        Next
    End Sub
```

Und genau das ist momentan das Problem. Die Methode, die diese Elemente (Mitarbeiter und Lieferanten) ausgeben soll, nimmt nämlich eine *Liste* vom Typ `AdressenBasis` entgegen. Die erste Liste, die ihr, wie in Abbildung 19.6 zu sehen ist, übergeben wird, ist aber vom Typ `Mitarbeiter`. Und ihr dürften natürlich keine Lieferanten hinzugefügt werden, denn sonst wäre die Typsicherheit und die Homogenität der Liste verletzt.

Bei einem pauschalen Casting einer `List(Of Mitarbeiter)` in `List(Of AdressenBasis)` könnte aber genau das passieren. Es steht ja dem Hinzufügen von Lieferanten einer `List(Of AdressenBasis)` eigentlich nichts im Weg. Nur, wenn wir die Methode `ListeAusgeben` aufrufen und in dieser Methode lediglich Elemente *auslesen*, – sie also inhaltlich *nicht* verändern – dann bleibt es ja bei einer `List(Of Mitarbeiter)` die lediglich über die »Basisklasse« verarbeitet werden soll.

```
'Co-Variance Demo
Public Class Form1

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

        'Mitarbeiter mithilfe von Auflistungs-Initialisierern füllen
        Dim mitarbeiter As New List(Of Mitarbeiter) From {
                    New Mitarbeiter With {.IDMitarbeiter = 1, .Nachname = "Leenings", .Vorname = "Ramona"},
                    New Mitarbeiter With {.IDMitarbeiter = 3, .Nachname = "Calla", .Vorname = "Sarika"},
                    New Mitarbeiter With {.IDMitarbeiter = 2, .Nachname = "Löffelmann", .Vorname = "Klaus"},
                    New Mitarbeiter With {.IDMitarbeiter = 3, .Nachname = "Pusch", .Vorname = "Marcus"},
                    New Mitarbeiter With {.IDMitarbeiter = 2, .Nachname = "Belke", .Vorname = "Andreas"}}

        Dim lieferanten As New List(Of Lieferant) From {
                    New Lieferant With {.IDLieferant = 1, .Nachname = "Woerrlein", .Vorname = "Hartmut"},
                    New Lieferant With {.IDLieferant = 2, .Nachname = "Löffelmann", .Vorname = "Klaus"},
                    New Lieferant With {.IDLieferant = 3, .Nachname = "Grauthof", .Vorname = "Kristin"}}

        ListeAusgeben(mitarbeiter)
        ListeAusgeben(lieferanten)
```

Abbildung 19.6 Gleiches Problem in Variation: `ListeAusgeben` nimmt eine `List(Of adressenBasis)` entgegen – ein Casten von den abgeleiteten Typen als Typparameter beim Aufruf der Methode ist damit allerdings nicht möglich. Der Typparameter von `List(Of )` ist nicht kovariant.

Gäbe es also eine Einschränkung, dass man einer solchen – ich nenne sie mal – *Basisklassenliste* – keine Elemente hinzufügen könnte, dann – und nur dann – wäre garantiert, dass sie homogen bliebe, und nur dann könnte sie als Parameter für eine Methode, so wie wir sie hier zur Ausgabe verwenden möchten, infrage kommen.

In Visual Basic 2010 geht genau das. Zwar nur für Typparameter generischer Schnittstellen, aber das spielt ja letzten Endes keine Rolle. Es kann uns ja egal sein, ob wir über eine Klasse oder eine Schnittstellenvariable auf unsere Liste zugreifen wollen.

Möglich wird das, weil es bei der Definition von Schnittstellen ab Visual Basic 2010 bzw. .NET Framework 4.0 die Möglichkeit gibt, durch das Schlüsselwort `Out` für Typparameter eine Schnittstelle für die kovariante Benutzung und mit dem Schlüsselwort `In` für die kontravariante Benutzung zu definieren.

`Out` sorgt also dafür, dass eine Schnittstelle nur Funktionen oder Eigenschaften implementieren darf, die den angegebenen Typen zurückliefern dürfen, und `In` dafür, dass Methoden oder Eigenschaften Parameter dieses Typs nur entgegennehmen, aber eben nicht zurückliefern dürfen.

.NET Framework in der Version 4.0 macht genau das mit `IEnumerable`, also mit der Schnittstelle, die vorschreibt, dass eine Klasse eine Auflistung darstellt und damit mit `For Each` durchiterierbar ist. `IEnumerable` ist folgendermaßen definiert:

```
Public Interface IEnumerable(Of Out typ)
   .
   .
   .
End Interface
```

Und genau dieses neue Schlüsselwort `Out` bestimmt, dass Klassen, über die man über diese Schnittstelle zugreift, unsere Typen nur ausspucken, sie aber nicht entgegen nehmen dürfen. Also:

```
Function EineMethode() As typ
```

ginge in Ordnung –

```
Sub NochNeMethode(ByVal einParameter As typ)
```

allerdings nicht, denn hier wird der Schnittstelle ein Typ zugeführt. In unserem Fall führt jetzt das Umbauen unserer Ausgabe der Liste in ein `IEnumerable` dazu, dass wir für homogene Listen an dieser Stelle sorgen. Der Typparameter für `IEnumerable` ist, wie gesagt, mit `Out` definiert, und damit darf die `Items`-Methode der Schnittstelle zwar Elemente zurückgeben, aber eine `Add`-Methode, die diesen generischen Typ entgegennehmen würde, darf es eben in der Schnittstellendefinition nicht geben. Und wie gesagt: Das bedeutet nicht, dass die Klasse, die die Schnittstelle implementiert, diese Methode nicht haben darf, sondern nur, dass die Schnittstelle eine solche Methode nicht binden kann, was aber als Garant für die Typsicherheit ausreicht. Und nach dem Umbauen läuft das Programm dann auch genau so wie wir es wollen:

```
Sub ListeAusgeben(ByVal adressenListe As IEnumerable(Of AdressenBasis))
    For Each item In adressenListe
        Debug.Print(item.Nachname & ", " & item.Vorname)
    Next
End Sub
```

HINWEIS Das Beispiel zeigt darunter ebenfalls anhand der Methode `Distinct` den umgekehrten Fall, also Kontravarianz für generische Schnittstellen, was vom Prinzip genauso funktioniert, nur dass bei der Definition das Schlüsselwort `In` zur Anwendung kommt und damit bestimmt wird, dass der Typparameter nur der Klasse *übergeben*, aber nicht zurückgeliefert werden darf.

Erweiterungsmethoden

Prinzipiell gab es schon immer Möglichkeiten, Klassen um neue Methoden oder Eigenschaften zu erweitern. Sie vererbten sie und fügten ihnen anschließend im vererbten Klassencode neue Methoden oder Eigenschaften hinzu. Das funktionierte so lange, wie es sich um Typen handelte, die …

- … nicht mit dem `NotInheritable`-Modifizierer (etwa: *nicht vererbbar*; der Vollständigkeit halber: `Sealed` in C#, etwa: *versiegelt*) gekennzeichnet waren, oder
- … keine Wertetypen waren – denn mit `Structure` erstellte Typen sind automatisch Wertetypen und die sind implizit nicht vererbbar.

Für eine bessere Strukturierung Ihres Codes wurden daher in Visual Basic 2008 so genannte Erweiterungsmethoden eingeführt, mit deren Hilfe Entwickler bereits definierten Datentypen benutzerdefinierte Funktionen hinzufügen können, ohne eben einen neuen, vererbten Typ zu erstellen. Erweiterungsmethoden sind eine besondere Art von statischen Methoden, die Sie jedoch wie Instanzmethoden für den erweiterten Typ aufrufen können. Für in Visual Basic (und natürlich auch C#) geschriebenen Clientcode gibt es keinen sichtbaren Unterschied zwischen dem Aufrufen einer Erweiterungsmethode und den Methoden, die in einem Typ tatsächlich definiert sind.

Allerdings gibt es dabei Einschränkungen bzw. Konventionen, was das Erweitern von Klassen auf diese Weise anbelangt:

- Erweiterungsmethoden müssen in Visual Basic mit einem Attribut besonders gekennzeichnet werden. Dazu dient das `ExtensionAttribute`, das Sie im Namespace `System.Runtime.CompilerServices` finden.
- Bei einer Erweiterungsmethode kann es sich ausschließlich um eine `Sub` oder eine `Function` handeln. Sie können eine Klasse um Erweiterungsmethoden, *nicht* aber um Eigenschaften, neue Feldvariablen oder Ereignisse erweitern.
- Eine Erweiterungsmethode muss sich in einem gesonderten Modul befinden
- Im Bedarfsfall muss das Modul, sollte es in einem anderen Namespace enthalten sein als die Instanz, die es verwendet, wie jede andere Klasse auch, entsprechend importiert werden
- Eine Erweiterungsmethode weist mindestens einen Parameter auf, der den Typ (die Klasse, die Struktur) bestimmt, die sie erweitert. Möchten Sie beispielsweise eine Erweiterungsmethode für den `String`-Datentyp erstellen, muss der erste Parameter, den die Erweiterungsmethode entgegennimmt, vom Typ `String` sein. Aus diesem Grund kann ein `Optional`-Parameter oder ein `ParamArray`-Parameter nicht der erste Parameter in der Parameterliste sein, da diese zur Laufzeit variiert, dem Compiler aber bereits zur Entwurfszeit bekannt sein muss.

Das Haupteinsatzgebiet von Erweiterungsmethoden

Der Hauptgrund für die Einführung von Erweiterungsmethoden in die wichtigen .NET-Sprachen war es, eine LINQ-kompatible Sprachinfrastruktur für die Compiler schaffen zu können, mit denen sich die Verwendung allgemein gültiger, generischer und statischer Funktionen typgerecht »anfühlt«.

Für das tiefere Verständnis brauchen Sie allerdings gute Kenntnisse von Schnittstellen; den Abschnitt über Lambda-Ausdrücke sollten Sie ebenfalls bereits durchgearbeitet haben.

Die Entwickler von .NET Framework 3.5 standen vor der Aufgabe, in vorhandene Typen quasi eine zusätzliche Funktionalität einzubauen, ohne aber den vorhandenen Code dieser Typen – insbesondere waren dabei alle generischen Auflistungsklassen betroffen – in irgendeiner Form auch nur um ein einzelnes Byte zu verändern. Das hätte nämlich zwangsläufig zu so genannten *Breaking Changes* (Änderungen mit unberechenbaren Auswirkungen auf vorhandene Entwicklungen) geführt. So war es prinzipiell nur möglich, zusätzliche Funktionalität in statischem, generischem Code unterzubringen, der sich allerdings sehr »uncool« angefühlt hätte.

Erweiterungsmethoden sind im Grunde genommen nur eine Mogelpackung, denn sie bleiben das, was sie sind: simple statische Methoden, die sich, wie gesagt, lediglich wie Membermethoden der entsprechenden Typen anfühlen. Ein kleiner Codeausschnitt soll das verdeutlichen:

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 19\ErweiterungsmethodenDemo
```

```
'Wichtig: Das Extension-Attribut ist in diesem Namespace versteckt
Imports System.Runtime.CompilerServices

'Wichtig: Der Namespace mit der Erweiterungsmethode für einen Typ muss importiert werden!
```

```
Imports ErweiterungsmethodenDemo.MeineErweiterungsmethode

Module Hauptmodul

    Sub Main()

        Dim t = "Dies ist die erste Zeile!\nUnd dieses hier die zweite."
        Console.WriteLine(t.Formatted)

        'Auf Taste warten.
        Console.ReadKey()
    End Sub

End Module

Namespace MeineErweiterungsmethode

    Module ErweiterungsmethodenModul

        'Das Extension-Attribut kennzeichnet eine Methode
        'als Erweiterungsmethode. Der Typ des ersten Parameters
        'bestimmt den Typ, der erweitert werden soll.
        <Extension()> _
        Public Function Formatted(ByVal text As String) As String
            'Aus \n ein CR+LF machen.
            Return text.Replace("\n", vbNewLine)
        End Function

    End Module
End Namespace
```

Dieses kleine Beispiel erweitert den `String`-Datentyp um eine Methode `Formatted`. Diese Methode soll einen Text, der eine Formatierungsanweisung à la C# und C++ enthält (nämlich »\n« für das Einfügen eines Zeilenumbruchs) umsetzen. Im Beispiel kann also über die `Formatted`-Methode ein String ausgegeben werden, der einen Zeilenumbruch als Steuerzeichen enthält.[1] Damit der `String`-Datentyp eine neue Methode erhalten kann, ist Folgendes notwendig:

- Die Methode muss als statische Methode definiert werden. Da in Visual Basic alle Methoden statisch sind, die sich in einem Modul befinden, packt man Erweiterungsmethoden in Visual Basic am besten in ein solches Modul, wie hier im Beispiellisting zu sehen.
- Damit die Methode als Erweiterungsmethode zum Datentyp erkannt wird, benötigt man ein Attribut und zwar das `ExtensionAttribute`. Mit diesem Attribut wird die Erweiterungsmethode markiert. Damit Sie auf das `ExtensionAttribute` zugreifen können, müssen Sie den Namespace `System.Runtime.CompilerServices` einbinden.
- Damit der Typ auf die Erweiterungsmethode zugreifen kann, muss der Namespace, der die Erweiterungsmethode enthält, importiert werden. Das gilt natürlich nur für den Fall, dass sich die Namespaces des Kontextes, in dem sich der erweiterte Typ befindet, von der Erweiterungsmethode, um die erweitert wird, unterscheiden. Deswegen finden Sie die entsprechende `Imports`-Anweisung am Anfang der Codedatei.

[1] Wichtig zu wissen: Das ist damit aber nicht dasselbe wie in C# oder C++, bei denen die Umsetzung dieser Steuerzeichen schon zur Kompilierungszeit »ausgetauscht« werden. Hier in unserem Beispiel verschwenden wir Prozessorzeit, um diese Umsetzung der Steuerzeichen zur Laufzeit vorzunehmen.

Das Erweitern von vorhandenen Typen auf diese Weise mag ja für »Convenience Purposes« (siehe auch Kapitel 15) schon einigermaßen sinnvoll sein. Aber was die Infrastrukturschaffung für LINQ anbelangt, kommt jetzt der geniale Trick: Da Erweiterungsmethoden alle Typen als ersten Parameter aufweisen können, können Ihnen auch Schnittstellen als Typ übergeben werden. Das führt aber zwangsläufig dazu, dass alle Typen, die diese Schnittstelle implementieren, in Folge auch mit den entsprechenden scheinbar zusätzlichen Methoden ausgestattet sind. In Kombination mit Generics und Lambda-Ausdrücken ist das natürlich eine coole Sache, da die eigentliche Funktionalität durch – je nach Ausbaustufe der Überladungsversionen der statischen Erweiterungsmethoden – einen oder mehrere Lambda-Ausdrücke gesteuert wird und der eigentlich zu verarbeitende Typ eben durch die Verwendung von Generics erst beim späteren Typisieren (Typ einsetzen) ans Tageslicht kommt. Mit diesem Kunstgriff schafft man im Handumdrehen eine komplette, typsichere Infrastruktur beispielsweise für das Gruppieren, Sortieren, Filtern, Ermitteln oder sich gegenseitige Ausschließen von Elementen aller Auflistungen, die eine bestimmte, um Erweiterungsmethoden ergänzte Schnittstelle implementieren – beispielsweise bei `IEnumerable(T)` um die statischen Methoden der Klasse `Enumerable`. Oder kurz zusammengefasst: *Alle* Auflistungen, die `IEnumerable(T)` einbinden, werden scheinbar um Membermethoden ergänzt, die in `Enumerable` zu finden sind, und das sind genau die, die für LINQ benötigt werden, um Selektionen, Sortierungen, Gruppierungen und weitere Funktionalitäten von Daten durchzuführen. Es ist aber nur scheinbar so – die ganze Funktionalität von LINQ wird letzten Endes an eine rein prozedural aufgebaute Klasse mit einer ganzen Menge statischer Funktionen delegiert.

Erweiterungsmethoden zur Vereinfachung von Auflistungsinitialisieren

Seit Visual Basic 2010 und dem Konzept von Auflistungsinitialisierern kommt Erweiterungsmethoden eine weitere Rolle zu, und zwar bei der Vereinfachung im Umgang mit den Parametern, die Sie hinter der `From`-Klausel für Auflistungsinitialisierer in geschweiften Klammern übergeben können. Kapitel 20 hält dazu genauere Infos in den entsprechenden Abschnitten bereit.

Teil C

Programmieren von und mit Datenstrukturen aus .NET Framework

In diesem Teil:

Arrays und Auflistungen 601
Serialisierung 665
Attribute und Reflection 693

Kapitel 20

Arrays und Auflistungen

In diesem Kapitel:

Grundsätzliches zu Arrays 602
Initialisierung von Arrays 603
Enumeratoren 618
Grundsätzliches zu Auflistungen (Collections) 622
Initialisierung von Auflistungen 626
Wichtige Auflistungen von .NET Framework, die Sie kennen sollten 628
Generische Auflistungen (Generic Collections) 653

Arrays kennt fast jedes Basic-Derivat seit Jahrzehnten – und natürlich können Sie auch in Visual Basic auf diese (Daten-)Felder (so der deutsche Ausdruck, der aber in diesem Buch nicht weiter verwendet wird) zurückgreifen. Doch .NET Framework wäre nicht .NET Framework, wenn nicht auch Arrays viel mehr Möglichkeiten bieten würden als nur den Zugriff auf Daten über einen numerischen Index.

Das bedeutet, dass die Leistung von Arrays weit über die reine Zurverfügungstellung von Containern für die Speicherung verschiedener Elemente eines Datentyps hinausreicht. Da Arrays auf `Object` basieren und damit eine eigene Klasse darstellen (`System.Array` nämlich), bietet .NET Framework über `Array`-Objekte weit reichende Funktionen zur Verwaltung ihrer Elemente an.

So können Arrays beispielsweise quasi auf Knopfdruck sortiert werden. Liegen sie in sortierter Form vor, können Sie auch binär nach ihren Elementen suchen und viele weitere Dinge mit ihnen anstellen, ohne selbst den entsprechenden Code dafür entwickeln zu müssen.

Zu guter Letzt bildet der Typ `System.Array` aber auch die Basis für weitere Datentypen – zum Beispiel `ArrayList` –, die Datenelemente in einer bestimmten Form dynamisch verwalten können, aber auch die Grundlage für viele der generischen Auflistungstypen, die wiederum die Basis für *LINQ to Objects* bilden.

Dieses Kapitel zeigt Ihnen, was Sie mit Arrays und den von ihnen abgeleiteten Klassen alles anstellen können, und bereitet Sie nicht zuletzt damit auf den Umgang mit *LINQ to Objects* vor.

Grundsätzliches zu Arrays

Arrays im ursprünglichen Basic-Sinne dienen dazu, *mehrere* Elemente desselben Datentyps unter einem bestimmten Namen verfügbar zu machen. Um die einzelnen Elemente zu unterscheiden, bedient man sich eines Indexes (ganz einfach ausgedrückt: einer Nummerierung der Elemente) – der im Übrigen nichts mit einem Index für die Datensätze einer Datenbanktabelle zu tun hat –, damit man auf die verschiedenen Array-Elemente zugreifen kann.

BEGLEITDATEIEN Viele der größeren nun folgenden Beispiele sind im Projekt *Arrays* in verschiedenen Methoden zusammengefasst. Das Projekt befindet sich im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 20\Arrays
```

Sie können dieses Projekt verwenden, um die Beispiele an Ihrem eigenen Computer nachzuvollziehen oder um eigene Experimente mit Arrays durchzuführen.

Ein Beispiel:

```
Sub Beispiel1()

    'Array mit 10 Elementen fest deklarieren.
    'Wichtig: Anders als in C# oder C++ wird der Index
    'des letzten Elements, nicht die Anzahl der Elemente
    'festgelegt! Elementzählung beginnt bei 0.
    'Die folgende Anweisung definiert also 10 Elemente:
    Dim locIntArray(9) As Integer

    'Zufallsgenerator initialisieren,
    Dim locRandom As New Random(Now.Millisecond)
```

```
        For count As Integer = 0 To 9
            locIntArray(count) = locRandom.Next
        Next

        For count As Integer = 0 To 9
            Console.WriteLine("Element Nr. {0} hat den Wert {1}", count, locIntArray(count))
        Next

        Console.ReadLine()

    End Sub
```

Wenn Sie dieses Beispiel ausführen, erscheint im Konsolenfenster eine Liste mit Werten, etwa wie im nachstehenden Bildschirmauszug zu sehen (die Werte sollten sich natürlich von Ihren unterscheiden, da es zufällige sind).[1]

```
Element Nr. 0 hat den Wert 1074554181
Element Nr. 1 hat den Wert 632329388
Element Nr. 2 hat den Wert 1312197477
Element Nr. 3 hat den Wert 458430355
Element Nr. 4 hat den Wert 1970029554
Element Nr. 5 hat den Wert 503465071
Element Nr. 6 hat den Wert 112607304
Element Nr. 7 hat den Wert 1507772275
Element Nr. 8 hat den Wert 1111627006
Element Nr. 9 hat den Wert 213729371
```

Dieses Beispiel demonstriert die grundsätzliche Anwendung von Arrays. Natürlich sind Sie bei der Definition des Elementtyps nicht auf `Integer` festgelegt. Es gilt der Grundsatz: Jedes Objekt in .NET kann Element eines Arrays sein.

Initialisierung von Arrays

Arrays werden statisch im Speicher gehalten. Einmal angelegt, können sie nicht mehr verändert werden. Aber: Eine Objektvariable, die auf einen ganz bestimmten Array-Typ zeigt, kann auf ein neues Array gleichen Typs mit einer anderen Anzahl an Elementen umgebogen werden. Es gibt mehrere Möglichkeiten, das zu erreichen. In dem Moment, in dem Sie ein Array definieren und gleichzeitig die Anzahl der Elemente bestimmen, wird gleichzeitig der Speicher für diese Elemente reserviert. Das bedeutet:

```
'Byte-Array mit 11 Elementen (0-10)
Dim myByteArray(10) As Byte
```

Diese Anweisung legt das Array vom Typ `Byte` an und reserviert gleichzeitig den Speicherplatz für 11 `Byte`-Elemente.

[1] Kleine Anmerkung am Rande: Ganz so zufällig sind die Werte nicht – `Random` stellt nur sicher, dass generierte Zahlenfolgen zufällig *verteilt* sind. Bei gleicher Ausgangsbasis (definiert durch den Parameter `Seed`, den Sie der `Random`-Klasse beim Instanziieren übergeben) produziert `Random` auch gleiche Zahlenfolgen. Da wir hier die Millisekunde als Basis übergeben und eine Sekunde aus 1000 Millisekunden besteht, gibt es eine Wahrscheinlichkeit von 1:1000, dass Sie dieselbe wie die hier abgedruckte Zahlenfolge generieren lassen. Es ist immer schwer, einer deterministischen Maschine so etwas Indeterministisches wie Zufall beizubringen…

Änderung der Array-Dimensionen zur Laufzeit

In einigen Fällen kann es aber sinnvoll sein, zunächst ein Array nur von einem bestimmten Typ zu definieren, die Anzahl der Elemente aber offen zu halten. Denn die Elementanzahl wird vielleicht erst später berechnet oder sonst wie ermittelt und soll deswegen auch erst später definiert werden. Und das funktioniert dann auf folgende Weise:

```
Public Class ArrayInitializeDemo

    'Erstmal nur definieren als Array vom Typ Integer
    Private myIntegerArray() As Integer

    Sub New(ByVal ElementCount As Integer)
        'Hier wissen wir, wie viele wir brauchen,
        'und nehmen erst die Reservierung der Elemente vor:
        myIntegerArray = New Integer(ElementCount - 1) {}
    End Sub

End Class
```

In der ersten in Fettschrift gesetzten Zeile des oben stehenden Listings sehen Sie, dass `myIntegerArray` zunächst deklariert, aber noch nicht definiert wird. Wie jede andere Referenzvariable zeigt sie deshalb auf `Nothing`. Erst in der zweiten Fettschriftzeile wird der Platz für eine entsprechende Anzahl an Elementen geschaffen.

HINWEIS Wichtig ist, dass Sie bei der Neudefinition von Arrays daran denken, dass zum einen auch hier nicht die Anzahl der Elemente, sondern die Array-Obergrenze angegeben wird, und zum anderen, dass Sie nicht vergessen dürfen, die geschweiften Leerklammern hinter der Dimensionsangabe anzugeben. Für Letzteres sieht der Compiler nämlich anderenfalls den Wert in den Klammern als Argument für einen parametrisierten Konstruktor des entsprechenden Wertetyps, den es aber gar nicht gibt, und er generiert eine Fehlermeldung.

Die Magie von ReDim

Eigentlich mehr aus Gründen der Abwärtskompatibilität gibt es auch in .NET-Versionen von Visual Basic eine weitere Möglichkeit, ein Array nach seiner ersten Deklaration neu zu dimensionieren. Auch hier soll ein Beispiel den Umgang verdeutlichen:

```
    Sub Beispiel3()

        Dim locAnzahlStrings As Integer = 15
        Dim locStringArray As String()
        locStringArray = GeneriereStrings(locAnzahlStrings, 30)
        Console.WriteLine("Ausgangsgröße: {0} Elemente. Es folgt der Inhalt:", locStringArray.Length)
        Console.WriteLine(New String("="c, 40))
        DruckeStrings(locStringArray)
        'Wir brauchen 10 weitere, die alten sollen aber erhalten bleiben!
        ReDim Preserve locStringArray(locStringArray.Length + 9)
```

```
        'Bleiben die alten wirklich erhalten?
        Console.WriteLine()
        Console.WriteLine("Inhaltsüberprüfung:", locStringArray.Length)
        Console.WriteLine(New String("="c, 40))
        DruckeStrings(locStringArray)

        '10 weitere Elemente generieren.
        Dim locTempStrings(9) As String

        '10 Zeichen mehr pro Element, sodass wir die neuen leicht erkennen können.
        locTempStrings = GeneriereStrings(10, 40)

        'In das "alte" Array kopieren, aber ab Index 15,
        locTempStrings.CopyTo(locStringArray, 15)

        'und nachschauen, was nun wirklich drinsteht!
        Console.WriteLine()
        Console.WriteLine("Inhaltsüberprüfung:", locStringArray.Length)
        Console.WriteLine(New String("="c, 40))
        DruckeStrings(locStringArray)

        Console.ReadLine()
    End Sub
```

Auch dieses Beispiel macht sich die Möglichkeit zunutze (direkt in der ersten Codezeile), die Dimensionierung und Deklaration eines Arrays zeitlich voneinander trennen zu können. Das Array `locStringArray` wird zunächst nur als Array deklariert – wie groß es sein soll, wird zu diesem Zeitpunkt noch nicht bestimmt. Dabei spielt es in Visual Basic übrigens keine Rolle, ob Sie eine Variable in diesem

```
Dim locStringArray As String()
```

oder in dem folgenden Stil als Array deklarieren:

```
Dim locStringArray() As String
```

Die Größe des Arrays wird das erste Mal von der Prozedur `GeneriereStrings` festgelegt. In dieser erfolgt zwar die Dimensionierung eines zunächst völlig anderen Arrays (`locStrings`); da dieses Array aber als Rückgabewert der aufrufenden Instanz überlassen wird, lebt der hier erstellte Array-Inhalt unter anderem Namen (`locStringArray`) weiter. Das durch beide Objektvariablen referenzierte Array ist dasselbe (im ursprünglichen Sinne des Wortes).

Übrigens: Diese Vorgehensweise entspricht eigentlich schon dem Neudimensionieren eines Arrays. Wie Sie beim ersten Array-Beispiel gesehen haben, spielt es natürlich keine Rolle, ob Sie eine Array-Variable zur Aufnahme eines Arrays als Rückgabewert einer Funktion verwenden, die zuvor mit einer festen Anzahl an Elementen oder ohne die Angabe der Arraygröße dimensioniert wurde. Allerdings: Sie verlieren bei dieser Vorgehensweise den Inhalt des ursprünglichen Arrays (falls es ihn gibt), denn die Methode erstellt ein neues Array, und mit der Zuweisung an die Objektvariable `locStringArray` wird intern nur ein Zeiger umgebogen. Der Speicherbereich der alten Elemente ist nicht mehr referenzierbar und würde bei nächster Gelegenheit vom Garbage Collector entsorgt werden.

WICHTIG Diese Tatsache hat Konsequenzen, denn: Anders, als es bei Visual Basic 6.0 zum Beispiel noch der Fall war und bei VBA der Fall ist, werden Arrays bei einer Zuweisung an eine andere Objektvariable *nicht* automatisch kopiert. Array-Variablen verhalten sich so wie jeder andere Referenztyp auch unter .NET: Ein Zuweisen einer Array-Variablen an eine andere biegt nur ihren Zeiger auf den Speicherbereich der eigentlichen Daten im Managed Heap um. Die Elemente bleiben an ihrem Platz im Managed Heap, und es wird kein neuer Speicherbereich mit einer Kopie der Array-Elemente für die neue Objektvariable erzeugt!

Das Redimensionieren kann – wie im Beispielcode zu sehen – auch mit der `ReDim`-Anweisung erfolgen. Mit dem zusätzlichen Schlüsselwort `Preserve` haben Sie darüber hinaus die Möglichkeit zu bestimmen, dass die alten Elemente dabei erhalten bleiben. Man möchte meinen, dass diese Verhaltensweise die Regel sein sollte, doch mit dem Wissen im Hinterkopf, was beim Neudimensionieren mit `ReDim` genau passiert, sieht die Sache schon anders aus:

- Wird `ReDim` aufgerufen, wird ein komplett neuer Speicherbereich dafür reserviert.
- Der Zeiger für die Objektvariable auf den Bereich für die zuvor zugeordneten Array-Elemente wird auf den neuen Speicherbereich umgebogen.
- Der Speicherbereich, der die alten Array-Elementezeiger enthielt, fällt dem nächsten Garbage-Collector-Durchlauf zum Opfer.
- Wenn `Preserve` hinter der `ReDim`-Anweisung platziert wird, bleiben die alten Array-Elemente erhalten. Doch das entspricht nicht der vollständigen Erklärung des Vorgangs. In Wirklichkeit wird auch hier ein neuer Speicherbereich erstellt, der den Platz für die neu angegebene Anzahl an Array-Elementen bereithält. Auch bei `Preserve` wird der Zeiger auf den Speicherbereich mit den alten Elementen für die betroffene Objektvariable auf den neuen Speicherbereich umgebogen. Doch bevor der alte Speicherbereich freigegeben wird und sich der Garbage Collector über die alten Elemente hermachen kann, werden die vorhandenen Elemente in den neuen Bereich kopiert.

Sie können übrigens mit `Preserve` nur Elemente retten, die in eindimensionalen Arrays gespeichert sind oder die durch die letzte Dimension eines mehrdimensionalen Arrays angesprochen werden.

Wertevorbelegung von Array-Elementen im Code

Alle Arrays, die in den vorangegangenen Beispielen verwendet wurden, sind zur Laufzeit mit Daten gefüllt worden. In vielen Fällen möchten Sie aber Arrays erstellen, die Sie automatisch mit Daten vorbelegen, die das Programm fest vorgibt:

```
Sub Beispiel5()

    'Deklaration und Definition von Elementen mit Double-Werten
    Dim locDoubleArray As Double() = {123.45F, 5435.45F, 3.14159274F}

    'Deklaration und spätere Definition von Elementen mit Integer-Werten
    Dim locIntegerArray As Integer()
    locIntegerArray = New Integer() {1I, 2I, 3I, 3I, 4I}
    'Deklaration und spätere Definition von Elementen mit Date-Werten
    Dim locDateArray As Date()
    locDateArray = New Date() {#12/24/2005#, #12/31/2005#, #3/31/2006#}
```

```
        'Deklaration und Definition von Elementen im Char-Array:
        Dim locCharArray As Char() = {"V"c, "B"c, "."c, "N"c, "E"c, "T"c, " "c, _
                                      "r"c, "u"c, "l"c, "e"c, "s"c, "!"c}

        'Zweidimensionales Array
        Dim locZweiDimensional As Integer(,)
        locZweiDimensional = New Integer(,) {{10, 10}, {20, 20}, {30, 30}}

        'Oder verschachtelt (das ist nicht zwei-dimensional)!
        Dim locVerschachtelt As Date()()
        locVerschachtelt = New Date()() {New Date() {#12/24/2004#, #12/31/2004#}, _
                                         New Date() {#12/24/2005#, #12/31/2005#}}
End Sub
```

Häufigste Fehlerquelle bei dieser Vorgehensweise, die sich Array-Initialisierer (engl. »Initializer«) nennt: Der Zuweisungsoperator wird falsch gesetzt. Bei der kombinierten Deklaration/Definition wird er benötigt; *definieren* Sie nur neu, lassen Sie ihn dagegen weg. Beachten Sie auch den Unterschied zwischen mehrdimensionalen und verschachtelten Arrays, auf den ich im Abschnitt »Mehrdimensionale und verschachtelte Arrays« (Seite 609) noch genauer eingehen werde.

Typrückschluss bei der Verwendung von Array-Initialisierern

Wenn Arrays auf Prozedurebene verwendet werden (also beispielsweise in `Function`- und `Sub`-Methoden oder Eigenschaftenprozeduren), erlaubt der Compiler auch hier lokalen Typrückschluss, was Ihnen als Entwickler eine Menge Tipparbeit erspart:

```
    Private Sub SomeThing()

        'String-Array mit 5 Elementen (nur auf Prozedur-Ebene definierbar).
        Dim myStringArray = {"Andreas", "Ramona", "Klaus", "Adriana", "Magges"}

    End Sub
```

Sie suchen hier vergeblich nach einer Typdefinition für `myStringArray`, und dennoch ist die verwendete Array-Objektvariable eine stark definierte, nämlich vom Typ `Array` aus Strings. Der Compiler erkennt, dass es sich bei den einzelnen Instanzen in den geschweiften Klammern um Strings handelt, und deswegen muss es sich bei der Gesamtmenge eben um ein Array aus Strings handeln.

Typrückschluss gibt es für Arrays aber noch in einem anderen Szenario, und Sie können ihn immer dann anwenden, wenn Sie Methodenaufrufe durchführen, die als Parameter (auch) ein leeres Array eines bestimmten Typs erwarten. Das folgende Beispiel zeigt einen solchen Fall und demonstriert dabei auch gleichzeitig einmal mehr die Überlegenheit von Visual Basic 2010 gegenüber C#; hier soll per Reflection (siehe Kapitel 22) der Konstruktor eines Typs ermittelt und dieser dann über seine `Invoke`-Methode aufgerufen werden. Beide Methoden verlangen, dass Arrays als Parameter übergeben werden, und in diesem Beispiel müssen diese Arrays zwar von einem bestimmten Typ sein, aber dürfen keine Elemente enthalten:

```
Dim tk = GetType(Testklasse). _
          GetConstructor(New Type() {}). _
            Invoke(New Object() {})
```

Hier wird zunächst das Typobjekt der Klasse `Testklasse` in Erfahrung gebracht. `GetConstructor` ermittelt anschließend den Konstruktor mit der Signatur, der den Typen entspricht, die durch das `Type`-Array übergeben werden, und `Invoke` ruft schließlich den Konstruktor mit den entsprechenden Parametern auf. Die Übergabe eines leeren Arrays eines bestimmten Typs erfolgt also auf die bis dato einzig mögliche Weise: `New ArrayTyp() { }`.

In C# sieht das Ganze übrigens ähnlich aus, und zwar bis heute, bis zur Visual C# 2010-Version:

```
var f = typeof(Testklasse).
        GetConstructor(new Type[0]).
         Invoke(new Object[] { });
```

Anders als in Visual Basic können in C#, dadurch dass bei Array-Definitionen die Anzahl der Elemente und nicht die obere Array-Grenze angegeben wird, leere aber definierte Arrays auf zwei Weisen bestimmt werden: Entweder durch die Angabe des Typs und 0 in eckigen Klammern oder durch die Übergabe des Typs und einer eckigen Leerklammer zum Zeichen dafür, dass es sich um ein Array handelt, sowie einer leeren Liste als Objektinitialisierer mit zwei geschweiften Klammern.

Mit Visual Basic 2010 geht das noch einfacher:

```
Dim tk = GetType(Testklasse).
           GetConstructor({}).
              Invoke({})
```

Der Visual Basic-Compiler macht an dieser Stelle etwas ziemlich Bemerkenswerts: Die `GetConstructor`-Methode definiert durch ihre Signatur, welchen Typ Argument sie erwartet, und leitet den zwar leeren, aber vorhandenen Array-Initialisierer aus dem Kontext in den richtigen Typen ab – in diesem Fall in ein Array vom Typ *Type*. Und das Gleiche passiert anschließend bei der *Invoke*-Methode, für die aus dem Array-Initialisierer-Literal abermals der korrekte Typ abgeleitet wird, nämlich ein leeres Array vom Typ *Object*.

Dominierende Typen

In diesem Zusammenhang – schauen Sie sich mal die folgenden Array-Definitionen an, die auf lokalem Typrückschluss basieren:

```
        'Neu seit Visual Basic 2010
        'Typrückschluss für Arrays:
        Dim locWasWirdEsWohl = {2, 4, 8, 16, 32, 64, 128}

        'Ein Typ dominiert.
        Dim locUndDasHier = {2, 4, 5.6F, 10.266212122326667R}

        'Das geht nicht - Object-Arrays gibt nur bei Option Strict Off!
        'Dim nocheinVersion = {10, "c"c, "Ein Text", #7/24/1969#}
```

Die erste Zeile ist klar – da wir hier nur Integer-Konstanten verwenden, ist das resultierende Array auch vom Typ `Integer`.

Bei der zweiten Definition ist dies nicht so offensichtlich, denn hier gibt es `Integer`, `Single` und `Double`-Werte in der Definition. Hier gilt: Es gibt einen definierten dominierenden Typen und das ist hier `Double`. Es kann nichts schief gehen beim Konvertieren zwischen `Integer` in `Double` und `Single` in `Double` – `Double` ist daher der dominierende Typ:

HINWEIS Obwohl der `Decimal`-Datentyp genauer ist als `Double`, bleibt `Double` dennoch dominierender Typ, wenn Sie auch mit Typliteral erzwungene `Decimal`-Konstanten (zum Beispiel 3.14159265754D) in die Liste aufnehmen würden. Das resultierende Array wäre also `Double`, wenn nur ein `Double`-Wert in der Liste wäre.

Die dritte Definition ist nicht möglich. Der Visual Basic-Compiler müsste hier `Object` als dominierenden Typ nehmen, aber zugunsten von Typsicherheit macht er das nicht und bringt stattdessen lieber eine Fehlermeldung. Nur wenn Sie `Option Strict` ausschalten würden, nimmt der Visual Basic-Compiler einen Fallback nach `Object` vor.

Mehrdimensionale und verschachtelte Arrays

Bei der Definition von Arrays sind Sie nicht auf eine Dimension beschränkt – auch das ist ein Feature, das schon bei jahrzehntealten Basic-Dialekten zu finden ist. Sie können ein mehrdimensionales Array erstellen, indem Sie bei der Deklaration die Anzahl der Elemente für jede Dimension durch Komma getrennt angeben:

```
Dim DreiDimensional(5, 10, 3) As Integer
```

Möchten Sie die Anzahl der zu verwaltenden Elemente bei der Deklaration des Arrays nicht festlegen, verwenden Sie die folgende Deklarationsanweisung:

```
Dim AuchDreiDimensional As Integer(,,)
```

Mit `ReDim` oder dem Aufruf von Funktionen, die ein entsprechend dimensioniertes Array zurückliefern, können Sie anschließend das Array definieren.

Auch hier ist wieder folgende Syntax alternativ zu `ReDim` möglich:

```
        'Verschachtelt anstelle von Redim:
        Dim AuchEinfachVerschachtelt As Integer()()
        AuchEinfachVerschachtelt = New Integer(10)() {}
```

Verschachtelte Arrays

Verschachtelte Arrays sind etwas anders konzipiert als mehrdimensionale Arrays. Bei verschachtelten Arrays (*Jagged Arrays*) ist ein Array-Element selbst ein Array (welches auch wieder Arrays beinhalten kann usw.). Anders als bei mehrdimensionalen Arrays können die einzelnen Elemente unterschiedlich dimensionierte Arrays enthalten und diese Zuordnung lässt sich auch im Nachhinein noch ändern.

Verschachtelte Arrays definieren Sie, indem Sie die Klammerpaare mit der entsprechenden Array-Dimension hintereinander schreiben – anders als bei mehrdimensionalen Arrays, bei denen die Dimensionen in einem Klammerpaar mit Komma getrennt angegeben werden.

Die folgenden Beispiel-Codezeilen (aus der `Sub Beispiel6`) zeigen, wie Sie verschachtelte Arrays definieren, deklarieren und ihre einzelnen Elemente abrufen können:

```
'Einfach verschachtelt; Tiefe wird nicht definiert.
Dim EinfachVerschachtelt(10)() As Integer

'Erstes Element hält ein Integer-Array mit drei Elementen.
EinfachVerschachtelt(0) = New Integer() {10, 20, 30}

'Zweites Element hält ein Integer-Array mit acht Elementen.
EinfachVerschachtelt(1) = New Integer() {10, 20, 30, 40, 50, 60, 70, 80}

'Druckt das dritte Element des zweiten Elements (30) des Arrays.
Console.WriteLine(EinfachVerschachtelt(1)(2))

'In einem Rutsch alles neu zuweisen.
EinfachVerschachtelt = New Integer()() {New Integer() {30, 20, 10}, _
                                        New Integer() {80, 70, 60, 50, 40, 30, 20, 10}}

'Druckt das dritte Element des zweiten Elements (jetzt 60) des Arrays.
Console.WriteLine(EinfachVerschachtelt(1)(2))
Console.ReadLine()
```

Die wichtigsten Eigenschaften und Methoden von Arrays

In den vorangegangenen Beispielprogrammen ließ es sich nicht vermeiden, die eine oder andere Eigenschaft oder Methode des `Array`-Objekts bereits anzuwenden. Dieser Abschnitt soll sich ein wenig genauer mit den zusätzlichen Möglichkeiten dieses Objekts beschäftigen – denn sie sind mächtig und können Ihnen, richtig angewendet, eine Menge Entwicklungszeit ersparen.

Anzahl der Elemente eines Arrays ermitteln mit Length

Wenn Sie wissen wollen, wie viele Elemente ein Array beherbergt, bedienen Sie sich seiner `Length`-Eigenschaft. Bei zwei- und mehrdimensionalen Arrays stellt `Length` ebenfalls die Anzahl aller Elemente fest.

Aufgepasst bei verschachtelten Arrays: Hier ermittelt `Length` nämlich nur die Elementanzahl des umgebenden Arrays. Sie können die Array-Länge eines Elements des umgebenden Arrays etwa so ermitteln:

```
'Verschachtelte Arrays
Dim locVerschachtelt As Date()()
locVerschachtelt = New Date()() {New Date() {#12/24/2004#, #12/31/2004#}, _
                                 New Date() {#12/24/2005#, #12/31/2005#}}

Console.WriteLine("Äußeres Array hat {0} Elemente.", locVerschachtelt.Length)
Console.WriteLine("Array des 1. Elements hat {0} Elemente.", locVerschachtelt(0).Length)
```

Sortieren von Arrays mit Array.Sort

Arrays lassen sich durch eine ganz simple Methode sortieren. Das folgende Beispiel soll demonstrieren, wie es geht:

```
Sub Beispiel7()

    'Array-Erstellen:
    Dim locNamen As String() = {"Jürgen", "Martina", "Hanna", "Gaby", "Michaela", "Miriam", "Ute", _
                        "Leonie-Gundula", "Melanie", "Uwe", "Andrea", "Klaus", "Anja", _
                         "Myriam", "Daja", "Thomas", "José", "Kricke", "Flori", "Katrin", "Momo", _
                         "Gareth", "Anne"}
    System.Array.Sort(locNamen)
    DruckeStrings(locNamen)
    Console.ReadLine()
End Sub
```

Wenn Sie dieses Beispiel starten, produziert es folgendes Ergebnis im Konsolenfenster:

```
Andrea
Anja
Anne
Daja
Flori
Gaby
Gareth
Hanna
José
Jürgen
Katrin
Klaus
Kricke
Leonie-Gundula
Martina
Melanie
Michaela
Miriam
Momo
Myriam
Thomas
Ute
Uwe
```

Die `Sort`-Methode ist eine statische Methode von `System.Array` und sie kann noch eine ganze Menge mehr. So haben Sie beispielsweise die Möglichkeit, einen korrelierenden Index mitsortieren zu lassen, oder Sie können bestimmen, zwischen welchen Indizes eines Arrays die Sortierung stattfinden soll. Die Onlinehilfe zu .NET Framework verrät Ihnen, welche Überladungen die `Sort`-Methode genau anbietet.

Umdrehen der Array-Anordnung mit Array.Reverse

Wichtig in diesem Zusammenhang ist eine weitere statische Methode von `System.Array` namens `Reverse`, die die Reihenfolge der einzelnen Array-Elemente umkehrt. Im Zusammenhang mit der `Sort`-Methode erreichen Sie durch den anschließenden Einsatz von `Reverse` die Sortierung eines Arrays in absteigender Reihenfolge. Wenn Sie das vorherige Beispiel um diese Zeilen

```
        Console.WriteLine()
        Console.WriteLine("Absteigend sortiert:")
        Array.Reverse(locNamen)
        DruckeStrings(locNamen)
        Console.ReadLine()
```

ergänzen, sehen Sie schließlich die Namen in umgekehrter Reihenfolge im Konsolenfenster, etwa:

```
Absteigend sortiert:
Uwe
Ute
Thomas
.
.
.
Anja
Andrea
```

Durchsuchen eines sortierten Arrays mit Array.BinarySearch

Auch bei der Suche nach bestimmten Elementen eines Arrays ist Ihnen .NET Framework behilflich. Dazu stellt die `Array`-Klasse die statische Funktion `BinarySearch` zur Verfügung.

WICHTIG Damit eine binäre Suche in einem Array durchgeführt werden kann, muss das Array in sortierter Form vorliegen – ansonsten wird die Funktion höchstwahrscheinlich falsche Ergebnisse zurückliefern. Wirklich brauchbar ist die Funktion überdies nur dann, wenn Sie sicherstellen, dass es keine Dubletten in den Elementen gibt. Da eine *binäre* Suche erfolgt, ist nicht gewährleistet, dass die Funktion das *erste* zutreffende Objekt findet und nicht ein beliebiges, das dem Gesuchten entsprach.

Beispiel:

```
   Sub Beispiel8()

        'Array-Erstellen:
        Dim locNamen As String() = {"Jürgen", "Martina", "Hanna", "Gaby", "Michaela", "Miriam", "Ute", _
                              "Leonie-Gundula", "Melanie", "Uwe", "Andrea", "Klaus", "Anja", _
                              "Myriam", "Daja", "Thomas", "José", "Kricke", "Flori", "Katrin", "Momo", _
                                    "Gareth", "Anne", "Jürgen", "Gaby"}
        System.Array.Sort(locNamen)
        Console.WriteLine("Jürgen wurde gefunden an Position {0}", _
                System.Array.BinarySearch(locNamen, "Jürgen"))
        Console.ReadLine()
    End Sub
```

Wie `Sort` ist auch `BinarySearch` eine überladene Funktion und bietet weitere Optionen, die die Suche beispielsweise auf bestimmte Indexbereiche beschränkt. IntelliSense und die Onlinehilfe geben hier genauere Hinweise für die Verwendung.

Implementierung von Sort und BinarySearch für eigene Klassen

.NET Framework erlaubt es, Arrays von beliebigen Typen zu erstellen, auch von solchen, die Sie selbst erstellt haben. Bei der Erstellung einer Klasse, die als Element eines Arrays fungieren soll, brauchen Sie dabei nichts Besonderes zu beachten.

Anders wird das allerdings, wenn Sie eine Funktion auf das Array anwenden wollen, die einen Elementvergleich erfordert, oder wenn Sie sogar steuern wollen, nach welchen Kriterien Ihr Array beispielsweise sortiert werden soll oder auch nach welchem Kriterium Sie es mit `BinarySearch` durchsuchen lassen möchten. Dazu ein Beispiel:

Sie haben eine Klasse entwickelt, die die Adressen einer Kontaktdatenbank speichert. Der nachfolgend gezeigte Code demonstriert, wie sie funktioniert und wie man sie anwendet:

BEGLEITDATEIEN Im Folgenden sehen Sie zunächst den Klassencode, der einen Adresseneintrag verwaltet – dieses Projekt finden Sie im Verzeichnis

```
...\VB 2010 Entwicklerbuch\Kapitel 20\IComparer01
```

```
Public Class Adresse

    Sub New(ByVal Name As String, ByVal Vorname As String, ByVal Plz As String, ByVal Ort As String)
        Me.Name = Name
        Me.Vorname = Vorname
        Me.PLZ = Plz
        Me.Ort = Ort
    End Sub

    'Automatisch implementierte Eigenschaften
    Public Property Name() As String
    Public Property Vorname() As String
    Public Property PLZ() As String
    Public Property Ort() As String

    Public Overrides Function ToString() As String
        Return Name + ", " + Vorname + ", " + PLZ + " " + Ort
    End Function
```

Sie sehen selbst: einfachstes Visual Basic. Die Klasse stellt ein paar Eigenschaften für die von ihr verwalteten Daten zur Verfügung, und sie überschreibt die `ToString`-Methode der Basisklasse, damit sie eine Adresse als kompletten String ausgeben kann.[2]

Das folgende kleine Beispielprogramm definiert ein Array aus dieser Klasse, richtet ein paar Adressen zum Experimentieren ein und druckt diese anschließend in einer eigenen Unterroutine aus:

[2] Den Vorgang des Überschreibens konnten Sie bereits im Kapitel 11 zu abstrakten Klassen und Polymorphie kennen lernen. Im Bedarfsfall führen Sie sich zunächst dieses Kapitel zu Gemüte.

```
Module ComparerBeispiel

    Sub Main()
        Dim locAdressen(5) As Adresse

        locAdressen(0) = New Adresse("Löffelmann", "Klaus", "11111", "Soest")
        locAdressen(1) = New Adresse("Heckhuis", "Jürgen", "99999", "Gut Uhlenbusch")
        locAdressen(2) = New Adresse("Sonntag", "Miriam", "22222", "Dortmund")
        locAdressen(3) = New Adresse("Sonntag", "Christian", "33333", "Wuppertal")
        locAdressen(4) = New Adresse("Ademmer", "Uta", "55555", "Bad Waldholz")
        locAdressen(5) = New Adresse("Kaiser", "Wilhelm", "12121", "Ostenwesten")

        Console.WriteLine("Adressenliste:")
        Console.WriteLine(New String("="c, 40))
        DruckeAdressen(locAdressen)
        'Array.Sort(locAdressen)
        Console.ReadLine()
    End Sub
    Sub DruckeAdressen(ByVal Adressen As Adresse())
        For Each locString As Adresse In Adressen
            Console.WriteLine(locString)
        Next
    End Sub

End Module
```

Auch hier werden Sie erkennen: Es passiert nichts wirklich Aufregendes. Wenn Sie das Programm starten, produziert es, wie zu erwarten, folgende Ausgabe im Konsolenfenster:

```
Adressenliste:
========================================
Löffelmann, Klaus, 11111 Soest
Heckhuis, Jürgen, 99999 Gut Uhlenbusch
Sonntag, Miriam, 22222 Dortmund
Sonntag, Christian, 33333 Wuppertal
Ademmer, Uta, 55555 Bad Waldholz
Kaiser, Wilhelm, 12121 Ostenwesten
```

Die Liste, wie wir Sie anzeigen lassen, ist allerdings ein ziemliches Durcheinander. Beobachten Sie, was passiert, wenn wir das Programm um eine `Sort`-Anweisung ergänzen und wie die Liste anschließend ausschaut. Dazu nehmen Sie die Auskommentierung der `Sort`-Methode im Listing einfach zurück. Nach dem Start des Programms warten Sie vergeblich auf die zweite, sortierte Ausgabe der Liste. Stattdessen löst .NET Framework eine Ausnahme aus, etwa wie in Abbildung 20.1 zu sehen.

Der Grund dafür: Die `Sort`-Methode der `Array`-Klasse versucht, die einzelnen Elemente des Arrays miteinander zu vergleichen. Dazu benötigt es einen so genannten *Comparer* (etwa: *Vergleicher*). Da wir nicht explizit angeben, dass wir einen speziellen Comparer verwenden wollen (welchen auch? – wir haben noch keinen!), erzeugt es einen Standard-Comparer, der aber wiederum die Einbindung einer bestimmten Schnittstelle in der Klasse verlangt, deren Elemente er miteinander vergleichen soll. Diese Schnittstelle nennt sich `IComparable`. Leider haben wir auch diese Schnittstelle nicht implementiert und die Ausnahme ist die Folge.

Wir haben nun drei Möglichkeiten. Wir binden die `IComparable`-Schnittstelle ein, dann können Elemente unserer Klasse auch ohne die Nennung eines expliziten Comparers verglichen und im Endeffekt sortiert werden.

```
IComparerBeispiel.vb
ComparerBeispiel
Module ComparerBeispiel

    Sub Main()
        Dim locAdressen(5) As Adresse

        locAdressen(0) = New Adresse("Löffelmann", "Klaus", "11111", "Soest")
        locAdressen(1) = New Adresse("Heckhuis", "Jürgen", "99999", "Gut Uhlenbusch")
        locAdressen(2) = New Adresse("Sonntag", "Miriam", "22222", "Dortmund")
        locAdressen(3) = New Adresse("Sonntag", "Christian", "33333", "Wuppertal")
        locAdressen(4) = New Adresse("Ademmer", "Uta", "55555", "Bad Waldholz")
        locAdressen(5) = New Adresse("Kaiser", "Wilhelm", "12121", "Ostenwesten")

        Console.WriteLine("Adressenliste:")
        Console.WriteLine(New String("="c, 40))
        DruckeAdressen(locAdressen)

        Console.WriteLine()
        Console.WriteLine("Adressenliste (sortiert):")
        Console.WriteLine(New String("="c, 40))
        Array.Sort(locAdressen)
        DruckeAdressen(locAdressen)

        Console.ReadLine()
    End Sub

    Sub DruckeAdressen(ByVal Adre
        For Each locString As Ad
            Console.WriteLine(loc
        Next
    End Sub

End Module

Public Class Adresse
    'Implements IComparable

InvalidOperationException wurde nicht behandelt.
Fehler beim Vergleichen von zwei Elementen im Array.
Hinweise zur Fehlerbehebung:
Für diese Ausnahme die allgemeine Hilfe aufrufen.
Für die interne Ausnahme die allgemeine Hilfe aufrufen.
Weitere Hilfeinformationen online suchen...
Aktionen:
Details anzeigen...
Ausnahmedetail in die Zwischenablage kopieren
```

Abbildung 20.1 Wenn Sie das Programm starten, löst es eine Ausnahme aus, sobald die `Sort`-Methode der `Array`-Klasse erreicht ist – in der Detailbeschreibung zur Ausnahme sehen Sie, was falsch läuft!

Oder wir stellen der Klasse einen *expliziten* Comparer zur Verfügung; in diesem Fall müssen wir ihn beim Einsatz von `Sort` (oder auch `BinarySearch`) benennen. Dieser Comparer hätte den Vorteil, dass sich durch ihn steuern ließe, nach welchem Kriterium die Adressenliste durchsucht bzw. sortiert werden soll.

Die dritte Möglichkeit: Wir machen beides. Damit wird unsere Klasse universell nutzbar und läuft nicht Gefahr, eine Ausnahme auslösen zu können. Und genau das werden wir in den nächsten beiden Abschnitten in die Tat umsetzen.

Implementieren der Vergleichsfähigkeit einer Klasse durch IComparable

Die Implementierung der `IComparable`-Schnittstelle, damit Instanzen unserer Klasse miteinander verglichen werden können, ist vergleichsweise simpel. Erinnern wir uns: Damit bestimmter Code allgemein gültig verwendbar sein kann, kann man sich des Konzeptes von Schnittstellen (engl. *Interfaces*) bedienen. Das Einbinden einer Schnittstelle in eine Klasse oder eine Struktur erzwingt, dass in der Klasse die Elemente (Eigenschaften, Methoden, Ereignisse) vorhanden sein müssen, die die Schnittstelle vorschreibt. Gleichzeitig reicht es aus, beispielsweise eine Methode, die eine Schnittstelle für eine Klasse vorschreibt, über eine Schnittstellenvariable aufzurufen. Damit spielt es für die aufrufende Instanz keine Rolle mehr, mit was sie es genau zu tun hat – ob es beispielsweise ein `Integer`- oder ein `Decimal`-Datentyp ist. Die aufrufende Instanz arbeitet lediglich mit einer Schnittstellenvariablen und ruft über diese die gewünschte Methode auf. Bei einem `Integer`-Datentyp, der diese Schnittstelle einbindet, werden so beispielsweise `Integer`-Werte verglichen, bei einem `String`-Datentyp, der die gleiche Schnittstelle einbindet, eben Zeichenketten. Doch das

merkt und interessiert die aufrufende Instanz gar nicht (in diesem Fall die `Array`-Klasse von .NET Framework). Sie ist nur am Ergebnis interessiert.

Das Implementieren der Schnittstelle in unserer Beispielklasse erfordert übrigens zusätzlich lediglich das Vorhandensein einer `CompareTo`-Methode, die die aktuelle Instanz der Klasse mit einer weiteren vergleicht, und das Anzeigen, dass es sich bei der Methode um die von der Schnittstelle vorgeschriebene Methode handelt.

Da durch den Einsatz von `IComparable` keine Möglichkeit besteht festzulegen, welches Datenfeld das Vergleichskriterium sein soll, wird unser Kriterium eine Zeichenkette sein, die aus Namen, Vornamen, Postleitzahl und Ort besteht – genau die Zeichenkette also, die `ToString` in der jetzigen Version bereits zurückliefert. Deswegen brauchen wir den eigentlichen Vergleich noch nicht einmal selbst durchzuführen, sondern können ihn an die `CompareTo`-Funktion des `String`-Objekts, das wir von `ToString` zurückerhalten, weiterreichen. Die Modifizierungen an der Klasse sind also denkbar gering (aus Platzgründen nur gekürzter Code, der die Änderungen widerspiegelt):

```
Public Class Adresse
    Implements IComparable
    .
    .
    .
    Public Function CompareTo(ByVal obj As Object) As Integer Implements System.IComparable.CompareTo

     Dim locAdresse As Adresse

     Try
         locAdresse = DirectCast(obj, Adresse)
     Catch ex As InvalidCastException
      Dim up As New InvalidCastException("'CompareTo' der Klasse 'Adresse' kann keine Vergleiche " + _
                                         "mit Objekten anderen Typs durchführen!")
         Throw up
     End Try
     Return ToString.CompareTo(locAdresse.ToString)
    End Function
End Class
```

An dieser Stelle vielleicht erwähnenswert ist der fett gekennzeichnete mittlere Bereich im Programm. Auf den ersten Blick mag es unsinnig erscheinen, einen möglichen Fehler abzufangen und ihn anschließend mehr oder weniger unverändert wieder auszulösen. Aber: Der Fehlertext ist entscheidend, und Sie tun sich selbst einen Gefallen, wenn Sie den Fehlertext einer Ausnahme, die Sie generieren, so formulieren, dass Sie eindeutig wissen, wer oder was sie ausgelöst hat. Wenn Sie das Programm anschließend starten, liefert es das gewünschte Ergebnis, wie im Folgenden zu sehen:

```
Adressenliste:
=====================================
Löffelmann, Klaus, 11111 Soest
Heckhuis, Jürgen, 99999 Gut Uhlenbusch
Sonntag, Miriam, 22222 Dortmund
Sonntag, Christian, 33333 Wuppertal
Ademmer, Uta, 55555 Bad Waldholz
Kaiser, Wilhelm, 12121 Ostenwesten

Adressenliste (sortiert):
=====================================
Ademmer, Uta, 55555 Bad Waldholz
```

```
Heckhuis, Jürgen, 99999 Gut Uhlenbusch
Kaiser, Wilhelm, 12121 Ostenwesten
Löffelmann, Klaus, 11111 Soest
Sonntag, Christian, 33333 Wuppertal
Sonntag, Miriam, 22222 Dortmund
```

Verwenden von Lambdas mit Arrays-Methoden

Mit Visual Studio 2010 ist das Anwenden von Lambdas dank der Fähigkeit, Lambdas auch über mehrere Zeilen schreiben zu können, sehr praktikabel geworden. Das gilt insbesondere für die Methoden von .NET Framework, die generische Delegaten als Parameter erwarten – hier lassen sich nämlich direkt mehrzeilige Lambdas einsetzen. Die `Array`-Klasse stellt dazu gleich mehrere Methoden zur Verfügung, und die Sortierung des Arrays damit vorzunehmen, wird zum Kinderspiel:

```
Array.Sort(locAdressen, Function(value1, value2)
                            Return value1.ToString.CompareTo(value2.ToString)
                        End Function)
```

Der statischen Methode `Sort` übergeben Sie als erstes Argument das Array, das sortiert werden soll, und als zweites Argument einen Delegaten, dessen Methode, auf die er zeigt, zwei Parameter vom Typ des Arrays entgegennimmt – oder eben ein Lambda, das das direkt macht, wie im Beispiel zu sehen.

Die Frage ist: Woher wissen Sie, welche Signatur sich hinter welchem Delegaten befindet? Das ist zugegeben nicht auf den ersten Blick ersichtlich – aber wird einfach in der Hilfe nachschlagbar, wie ein Blick auf Abbildung 20.2 offenbart.

Abbildung 20.2 Konsultieren Sie die Hilfe (F1), um herauszufinden, welche Signatur ein Lambda haben muss, um es als Parameter für einen generischen Delegaten einzusetzen

Gemäß der Hilfe müsste also die korrekte Signatur des Lambdas folgendermaßen lauten:

```
'Alternativ: Sortieren mit Comparision-Delegat und Lambda:
Array.Sort(Of Adresse)(locAdressen, Function(value1 As Adresse, value2 As Adresse)
                                        Return value1.ToString.CompareTo(value2.ToString)
                                    End Function)
```

Diese Zeile funktioniert natürlich genauso und beinhaltet alle erforderlichen Typinformationen im Detail. Aber auch hier nimmt Ihnen der Visual Basic-Compiler eine Menge Arbeit ab, indem er alle Typen, die Sie in den oberen Codezeilen in Fettschrift gesetzt sehen, aus dem Typen der Variablen ableitet, den Sie als ersten Parameter übergeben: `locAdressen`. Da `locAdressen` vom Typ `Adressen` ist, *müssen* alle anderen Typdefinitionen und Typparameter sich daraus ergeben, und deswegen können Sie sie an den entsprechenden Stellen auch weglassen.

Eine weitere Methode, die Sie mit Lambdas verwenden können, ist die `ForEach`-Methode der `Array`-Klasse. Die `ForEach`-Methode der `Array`-Klasse macht prinzipiell dasselbe, was die `ForEach`-Schleife macht – sie erledigt diese Aufgabe nur erheblich schneller. Auch hier findet sich im Projekt ein entsprechendes Beispiel:

```
'Auch hier die ALternative mit einer Methode, die einen
'Delegaten erwartet, der als Lambda formuliert werden kann:
Sub DruckeAdressenLambda(ByVal Adressen As Adresse())
    Array.ForEach(Adressen, Sub(item)
                                Console.WriteLine(item.ToString)
                            End Sub)
End Sub
```

Weitere Beispiele für das Verwenden von Lambdas in statischen Methoden finden Sie in diesem Kapitel zur Erklärung der generischen `List(Of )`-Auflistung, die Sie im Abschnitt »List(Of)-Auflistungen und Lambda-Ausdrücke« ab Seite 655 beschrieben finden.

Enumeratoren

Wenn Sie Arrays oder – wie Sie später sehen werden – Auflistungen für die Speicherung von Daten verwenden, dann müssen Sie in der Lage sein, diese Daten abzurufen. Mit *Indexern* gibt Ihnen .NET Framework eine einfache – *die* einfachste – Möglichkeit: Sie verwenden eine Objektvariable und versehen sie mit einem Index, der auch durch eine andere Variable repräsentiert werden kann. Ändern Sie diese Variable, die als Index dient, können Sie dadurch programmtechnisch bestimmen, welches Element eines Arrays Sie gerade verarbeiten wollen. Typische Zählschleifen, mit denen Sie durch die Elemente eines Arrays iterieren, sind die Folge – etwa im folgenden Stil:

```
For count As Integer = 0 To Array.Length - 1
    TuWasMit(Array(count))
Next
```

HINWEIS Enumeratoren haben nichts mit Enums zu tun. Lediglich die Namen sind sich etwas ähnlich. Enums sind aufgezählte Benennungen von bestimmten Werten im Programmlisting, während Enumeratoren die Unterstützung von `For Each` zur Verfügung stellen!

Wenn ein Objekt allerdings die Schnittstelle `IEnumerable` implementiert, gibt es eine elegantere Methode, die verschiedenen Elemente, die das Objekt zur Verfügung stellt, zu durchlaufen. Glücklicherweise implementiert `System.Array` die Schnittstelle `IEnumerable`, sodass Sie auf Array-Elemente mit dieser eleganten Methode – namentlich mit `For Each` – zugreifen können. Ein Beispiel:

```
'Deklaration und Definition von Elementen im Char-Array
Dim locCharArray As Char() = {"V"c, "B"c, "."c, "N"c, "E"c, "T"c, " "c,
                              "r"c, "u"c, "l"c, "e"c, "s"c, "!"c}
For Each c As Char In locCharArray
    Console.Write(c)
Next
Console.WriteLine()
Console.ReadLine()
```

Wenn Sie dieses Beispiel laufen lassen, sehen Sie im Konsolenfenster den folgenden Text:

```
VB.NET rules!
```

Enumeratoren werden von allen typdefinierten Arrays unterstützt und ebenso von den meisten Auflistungen. Sie können aber Enumeratoren auch in Ihren eigenen Klassen einsetzen, wenn Sie erlauben möchten, dass der Entwickler, der mit Ihrer Klasse arbeitet, durch Elemente mit `For Each` iterieren kann.

WICHTIG Enumeratoren sind lebenswichtig für LINQ (siehe ab Kapitel 25). Nur eine Auflistungsklasse, die `IEnumerable(Of T)` implementiert, kann als Abfrageausdruck in LINQ verwendet werden. Denken Sie daran, wenn Sie eigene Auflistungsklassen entwerfen.

Benutzerdefinierte Enumeratoren durch Implementieren von IEnumerable

Enumeratoren können allerdings nicht nur für die Aufzählung von gespeicherten Elementen in Arrays oder Auflistungen eingesetzt werden. Sie können auf Enumeratoren auch dann zurückgreifen, wenn eine Klasse ihre Enumerationselemente durch Algorithmen zurückliefert. Als Beispiel dafür möchte ich Ihnen zunächst einen Codeausschnitt zeigen, der nicht funktioniert – bei dem es aber in einigen Fällen wünschenswert wäre, wenn er funktionieren würde:

```
'Das würde nicht funktionieren:
For d As Date=#24/12/2010# To #31/12/2010#
    Console.WriteLine("Datum in Aufzählung: {0}", d)
Next d
```

Dabei könnte es für bestimmte Anwendungen durchaus sinnvoll sein, tageweise einen bestimmten Datumsbereich zu durchlaufen. Etwa wenn Ihre Anwendung herausfinden muss, wie viele Mitarbeiter, deren Daten Ihre Anwendung speichert, in einem bestimmten Monat Geburtstag haben.

Wenn das, was wir vorhaben, nicht mit `For … Next` funktioniert, vielleicht können wir dann aber eine Klasse schaffen, die einen Enumerator zur Verfügung stellt, sodass das Vorhaben mit `For Each` gelingt. Diese Klasse sollte beim Instanziieren Parameter übernehmen, mit denen wir bestimmen können, welcher Datumsbereich in welcher Schrittweite durchlaufen werden soll. Damit Sie mit `For Each` durch die Elemente einer Klasse iterieren können, muss die Klasse die Schnittstelle `IEnumerable` einbinden.

Das kann sie nur, wenn sie gleichzeitig eine Funktion `GetEnumerator` zur Verfügung stellt, die erst das Objekt mit dem eigentlichen Enumerator liefert. Doch eines nach dem anderen.

BEGLEITDATEIEN Schauen wir uns zunächst die Basisklasse an, die die Grundfunktionalität zur Verfügung stellt – Sie finden das Projekt im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 20\Enumerators
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei); starten Sie das Programm mit Strg + F5.

```
Public Class Datumsaufzählung
    Implements IEnumerable
    Dim locDatumsaufzähler As Datumsaufzähler

    Sub New(ByVal StartDatum As Date, ByVal EndDatum As Date, ByVal Schrittweite As TimeSpan)
        locDatumsaufzähler = New Datumsaufzähler(StartDatum, EndDatum, Schrittweite)
    End Sub
    Public Function GetEnumerator() As System.Collections.IEnumerator _
                    Implements System.Collections.IEnumerable.GetEnumerator
        Return locDatumsaufzähler
    End Function
End Class
```

Sie sehen, dass diese Klasse selbst nichts Großartiges macht – sie schafft durch die ihr übergebenen Parameter lediglich die Rahmenbedingungen und stellt die von `IEnumerable` verlangte Funktion `GetEnumerator` zur Verfügung. Die eigentliche Aufgabe wird von der Klasse `Datumsaufzähler` erledigt, die auch als Rückgabewert von `GetEnumerator` zurückgeliefert wird.

Eine Instanz dieser Klasse wird bei der Instanziierung von `Datumsaufzählung` erstellt. Was in dieser Klasse genau passiert, zeigt der folgende Code:

```
Public Class Datumsaufzähler
    Implements IEnumerator

    Private myAktuellesDatum As Date

    Sub New(ByVal StartDatum As Date, ByVal EndDatum As Date, ByVal Schrittweite As TimeSpan)
        Me.StartDatum = StartDatum
        Me.EndDatum = EndDatum
        Me.Schrittweite = Schrittweite
        myAktuellesDatum = StartDatum
    End Sub

    Public Property StartDatum() As Date
    Public Property EndDatum() As Date
    Public Property Schrittweite() As TimeSpan

    Public ReadOnly Property Current() As Object Implements System.Collections.IEnumerator.Current
        Get
            Return myAktuellesDatum
        End Get
    End Property
```

```
    Public Function MoveNext() As Boolean Implements System.Collections.IEnumerator.MoveNext
        myAktuellesDatum = myAktuellesDatum.Add(Schrittweite)
        If myAktuellesDatum > EndDatum Then
            Return False
        Else
            Return True
        End If
    End Function

    Public Sub Reset() Implements System.Collections.IEnumerator.Reset
        myAktuellesDatum = StartDatum
    End Sub
End Class
```

Der Konstruktor und die Eigenschaftsprozeduren sollen hier nicht so sehr interessieren. Vielmehr von Interesse ist, dass diese Klasse eine weitere Schnittstelle namens `IEnumerator` einbindet und die eigentliche Aufzählungsfunktionalität zur Verfügung stellt. Sie muss dazu die Eigenschaft `Current`, die Funktion `MoveNext` sowie die Methode `Reset` implementieren.

Wenn eine `For Each`-Schleife durchlaufen wird, dann wird das aktuell bearbeitete Objekt der Klasse durch die `Current`-Eigenschaft des eigentlichen Enumerators ermittelt. Anschließend zeigt `For Each` mit dem Aufruf der Funktion `MoveNext` dem Enumerator an, dass es auf das nächste Objekt zugreifen möchte. Erst wenn `MoveNext` mit `False` als Rückgabewert signalisiert, dass es keine weiteren Objekte mehr zur Verfügung stellen kann (oder will), ist die umgebende `For Each`-Schleife beendet.

In unserem Beispiel müssen wir bei `MoveNext` nur dafür sorgen, dass unsere interne Datums-Zählvariable um den Wert erhöht wird, den wir bei ihrer Instanziierung als Schrittweite bestimmt haben. Hat die Addition der Schrittweite auf diesen Datumszähler den Endwert noch nicht überschritten, liefern wir `True` als Funktionsergebnis zurück – `For Each` darf mit seiner Arbeit fortfahren. Ist das Datum allerdings größer als der Datumsendwert, wird die Schleife abgebrochen – der Vorgang wird beendet.

Das Programm, das von dieser Klasse Gebrauch machen kann, lässt sich nun sehr elegant einsetzen, wie die folgenden Beispielcodezeilen zeigen:

```
Module Enumerators
    Sub Main()
        Dim locDatumsaufzählung As New Datumsaufzählung(#12/24/2004#,
                                                        #12/31/2004#,
                                                        New TimeSpan(1, 0, 0, 0))
        For Each d As Date In locDatumsaufzählung
            Console.WriteLine("Datum in Aufzählung: {0}", d)
        Next

    End Sub
End Module
```

Das Ergebnis sehen Sie anschließend in Form einer Datumsfolge im Konsolenfenster.

Grundsätzliches zu Auflistungen (Collections)

Arrays haben in .NET Framework einen entscheidenden Nachteil. Sie können zwar dynamisch zur Laufzeit vergrößert oder verkleinert werden, aber der Programmieraufwand dazu ist doch eigentlich recht aufwändig. Wenn Sie sich schon früher mit Visual Basic beschäftigt haben, dann ist Ihnen »Auflistung« sicherlich ein Begriff, und darüber hinaus sind Auflistungen in Kapitel 2 auch schon vorgestellt worden.

Auflistungen erlauben es dem Entwickler, Elemente genau wie Arrays zu verwalten. Im Unterschied zu Arrays wachsen Auflistungen jedoch mit Ihren Speicherbedürfnissen. Damit ist aber auch klar, dass das Indizieren mit Nummern zum Abrufen der Elemente nur bedingt funktionieren kann. Wenn ein Array 20 Elemente hat und Sie möchten das 21. Element hinzufügen, dann können Sie das dem Array nicht einfach so mitteilen. Ganz anders bei Auflistungen: Hier fügen Sie ein Element mit der `Add`-Methode hinzu.

Intern werden (fast alle) Auflistungen ebenfalls wie (oder besser: als) Arrays verwaltet. Rufen Sie eine neue Auflistung ins Leben, dann hat dieses Array, wenn nichts anderes gesagt wird, eine Größe von 16 Elementen. Wenn die Auflistungsklasse später, sobald Ihr Programm richtig in Aktion ist, merkt, dass ihr die Puste mengentechnisch ausgeht, dann legt sie ein Array mit nunmehr 32 Elementen an, kopiert die vorhandenen Elemente in das neue Array, arbeitet fortan mit dem neuen Array und tut ansonsten so, als wäre nichts gewesen.

Dieser anfängliche Load-Faktor erhöht sich bei jedem Neuanlegen des Arrays, um die Kopiervorgänge zu minimieren. Wenn das Limit erreicht wird, geht man einfach davon aus, dass ein Programm für die kommenden Elemente noch einmal den gleichen Speicherplatz brauchen wird. Reicht dies nicht aus, ist anzunehmen, dass die Elementanzahl in der Auflistung weiter wächst, möglicherweise sogar exponentiell. Um nun ausufernde komplette Kopiervorgänge zu vermeiden, die das jeweils neu angelegte Array immer nur um wenige Elemente anwachsen lassen und daher ineffektiv sind, versucht man dem entgegenzuwirken, indem man auch die Schrittweite exponentiell anwachsen lässt –sie wird deswegen jeweils verdoppelt. Somit wird im Normalfall erreicht, dass die Schrittweite irgendwann schneller wächst, als das Array selbst »nachlegen« kann. Ergo sinkt die Anzahl kompletter Kopiervorgänge (und wird optimiert) Der Load-Faktor geht dabei bis maximal 2048.

In etwa entspricht also die Grundfunktionsweise einer Auflistung stark vereinfacht der folgenden Klasse:

```
Class DynamicList
    Implements IEnumerable

    Private myStepIncreaser As Integer
    Private myCurrentArraySize As Integer
    Private myCurrentCounter As Integer
    Private myArray() As Object

    Sub New()
        MyClass.New(16)
    End Sub

    Sub New(ByVal StepIncreaser As Integer)
        myStepIncreaser = StepIncreaser
        myCurrentArraySize = myStepIncreaser
        ReDim myArray(myCurrentArraySize)
    End Sub
```

```
    Sub Add(ByVal Item As Object)

        'Prüfen, ob aktuelle Arraygrenze erreicht wurde
        If myCurrentCounter = myCurrentArraySize - 1 Then
            'Neues Array mit mehr Speicher anlegen,
            'und Elemente hinüberkopieren. Dazu:

            'Neues Array wird größer:
            myCurrentArraySize += myStepIncreaser

            'temporäres Array erstellen
            Dim locTempArray(myCurrentArraySize - 1) As Object

            'Elemente kopieren
            'Wichtig: Um das Kopieren müssen Sie sich,
            'anders als bei VB6, selber kümmern!
            Array.Copy(myArray, locTempArray, myArray.Length)

            'temporäres Array dem Memberarray zuweisen
            myArray = locTempArray

            'Beim nächsten Mal werden mehr Elemente reserviert!
            myStepIncreaser *= 2
        End If

        'Element im Array speichern
        myArray(myCurrentCounter) = Item

        'Zeiger auf nächstes Element erhöhen
        myCurrentCounter += 1

    End Sub

    'Liefert die Anzahl der vorhandenen Elemente zurück
    Public Overridable ReadOnly Property Count() As Integer
        Get
            Return myCurrentCounter
        End Get
    End Property

    'Erlaubt das Zuweisen und Abfragen
    Default Public Overridable Property Item(ByVal Index As Integer) As Object
        Get
            Return myArray(Index)
        End Get

        Set(ByVal Value As Object)
            myArray(Index) = Value
        End Set
    End Property

    'Liefert den Enumerator der Basis (dem Array) zurück
    Public Function GetEnumerator() As System.Collections.IEnumerator Implements
System.Collections.IEnumerable.GetEnumerator
        Return myArray.GetEnumerator
    End Function
End Class
```

Nun könnte man meinen, diese Vorgehensweise könnte sich zu einer Leistungsproblematik entwickeln, da alle paar Elemente der komplette Array-Inhalt kopiert werden muss.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 20\DynamicList
```

Wie lange, glauben Sie, dauert es, ein Array mit 200.000 Zufallszahlen (mit Nachkommastellen) auf diese Weise anzulegen? 3 Sekunden? 2 Sekunden?

```
Module DynamicListDemo

    Private Const SAMPLESIZE = 200000

    Sub Main()

        Dim locDynamicList As New DynamicList
        Dim locRandom As New Random(Now.Millisecond)

        Console.WriteLine("Anlegen von {0} zufälligen Double-Elementen...", SAMPLESIZE)
        Dim locZeitmesser = Stopwatch.StartNew
        For count As Integer = 1 To SAMPLESIZE
            locDynamicList.Add(locRandom.NextDouble * locRandom.Next)
        Next
        locZeitmesser.Stop()
        Console.WriteLine("...in {0:#,##0} Millisekunden!", locZeitmesser.ElapsedMilliseconds)
        Console.ReadLine()
    End Sub
.
.
.
```

Finden Sie es selbst heraus, indem Sie das Programm starten:

```
Anlegen von 200000 zufälligen Double-Elementen...
...in 21 Millisekunden!
```

Gerade mal 21 Millisekunden benötigt das Programm für diese Operation[3] – beeindruckend, wie schnell Basic dieser Tage ist, finden Sie nicht?

[3] Auf einem Intel Core 2 Quad Q6600-Prozessor übrigens. Lesern des Vor-Vorgängerbuchs *Visual Studio 2005 – das Entwicklerbuch* (wie mehrfach schon erwähnt unter *http://www.activedevelop.de* kostenlos herunterladbar!) wird auffallen, dass das Ergebnis sogar eine Millisekunde langsamer ist als auf dem Computer, auf dem ich vor 5 Jahren das Visual Basic 2005-Buch schrieb. Sehr auffällig ist dabei zu sehen, dass neuere Computer ihre Leistung kaum noch »vertikal«, über die Taktfrequenz, sondern nur noch horizontal über mehrere Prozessoren skalieren können – beim Laufenlassen dieses Beispiels wird nur einer von vier Prozessorkernen genutzt – ¾ des Prozessors liegen ungenutzt (schlimmer: in diesem Beispiel sogar unnutzbar!) brach. Um nicht nur alles, sondern überhaupt mehr an Leistung aus modernen Prozessoren herauszukitzeln, ist daher Multithreading-Entwicklung sehr entscheidend und sollte bei Programmportierungen – beispielsweise wenn Sie ältere Borland-, MFC- oder Visual Basic 6.0-Anwendungen auf .NET portieren wollen oder müssen – *unbedingt* berücksichtigt werden! Ergänzung an dieser Stelle: Mit dem Update auf Visual Basic 2010 war es dann ein Samsung Notebook mit Core 2 Duo P8600-Prozessor und 2,4 GHz Taktfrequenz. Das Ergebnis dieses Mal unter Windows 7: 28 Millisekunden.

Nun muss ich Ihnen leider verraten: Die Klasse `DynamicList` werden Sie nie benötigen. Bestandteil von .NET Framework ist nämlich eine Klasse, die das schon kann. Sogar noch ein kleines bisschen schneller. Und: Sie hat zusätzlich noch andere Möglichkeiten, die Ihnen die selbst gestrickte Klasse nicht bietet. Im Beispielprojekt finden Sie eine `Sub Beispiel2`, die Ihnen die gleiche Prozedur mit der Klasse `ArrayList` demonstriert:

```
Sub Beispiel2()

    Dim locDynamicList As New ArrayList
    Dim locRandom As New Random(Now.Millisecond)

    Console.WriteLine("Anlegen von {0} zufälligen Double-Elementen...", SAMPLESIZE)
    Dim locZeitmesser = Stopwatch.StartNew
    For count As Integer = 1 To SAMPLESIZE
        locDynamicList.Add(locRandom.NextDouble * locRandom.Next)
    Next
    locZeitmesser.Stop()
    Console.WriteLine("...in {0:#,##0} Millisekunden!", locZeitmesser.ElapsedMilliseconds)
    Console.ReadLine()
End Sub
```

Dessen Geschwindigkeit ist auch nicht von schlechten Eltern:

```
Anlegen von 200000 zufälligen Double-Elementen...
...in 19 Millisekunden!
```

Im Prinzip arbeitet `ArrayList` nach dem gleichen Verfahren, das Sie in `DynamicList` kennen gelernt haben. `ArrayList` verfährt auch mit dem gleichen Trick, um möglichst viel Leistung herauszuholen. Es verdoppelt die jeweils nächste Größe des neuen Arrays im Vergleich zu der vorherigen Größe des Arrays. Damit reduziert sich der Gesamtaufwand des Kopierens erheblich. Da die Methode aber Bestandteil von .NET Framework ist, muss sie nicht zur Laufzeit »geJITted« werden, was der Geschwindigkeit zusätzlich zugute kommt.

Im Übrigen werden die Daten der einzelnen Elemente ja nicht wirklich bewegt. Lediglich die Zeiger auf die Daten werden kopiert – vorhandene Elemente bleiben im Managed Heap an ihrem Platz. Kapitel 9 erklärte Ihnen im Abschnitt »New oder nicht New – Wieso es sich bei Objekten um Verweistypen handelt« übrigens mehr zu diesem Thema.

WICHTIG Die Geschwindigkeit, mit der `ArrayList` für primitive Wertetypen und normale Wertetypen (wie `Double` hier im Beispiel) arbeitet, ist dabei *tatsächlich* nicht optimal, denn da es sich bei `ArrayList` intern um ein Array von Objekten handelt, müssen die (primitiven) Wertetypen, wie in Kapitel 13 beschrieben, geboxt werden, und das kostet Zeit und Speicherplatz. Erst mit Generics ist es sinnvoll möglich, Auflistungen auf Basis von typisierten Arrays und nicht auf `Object`-Arrays aufzubauen, und die Geschwindigkeits- und Platzersparnis dabei ist enorm, zumal sie auch die Arbeit des Garbage Collectors erleichtern, denn dieser hat beim Neuanlegen des Arrays, wenn der interne Workload überschritten ist, nicht mehr Hunderte einzelner geboxter Elemente, sondern nur noch ein »altes« Array zu entsorgen. Aber wie gesagt: Dieser Zusatz gilt nur für (primitive) Wertetypen. Bei normalen Referenztypen ist der Unterschied irrelevant.

Faustregel: Wenn Sie Wertetypen in Auflistungen einsetzen, sollten Sie *auf jeden Fall* mit generischen Auflistungen arbeiten, allen voran `List(Of Type)` und `Collection(Of Type)` – Letztere bitte nicht mit der Visual Basic 6.0/VBA-kompatiblen, nicht generischen Klasse `Collection` verwechseln!

Initialisierung von Auflistungen

Seit Visual Basic 2010 gibt eine elegante Möglichkeit, Auflistungen mit Werten vorzubelegen, ähnlich, wie Sie das mit Arrays schon im Abschnitt »Wertevorbelegung von Array-Elementen im Code« auf Seite 606 kennen gelernt haben. Die Syntax um das zu erreichen, ist allerdings ein wenig anders, wie das folgende Beispiel zeigt:

```
        'Collection-Initializers:
        Dim locHundeNamenList As New ArrayList From {"Waldi", "Basko", "Hasso",
                                                     "Wauzi", "Bello", "Minka"}
```

Dabei generiert der Compiler aus den Elementen, die Sie hinter dem Schlüsselwort `From` in den geschweiften Klammern angeben, für jedes Element einen `Add`-Befehl, mit dem das Element der Auflistung hinzugefügt wird, sodass das Ganze intern in etwa wie folgt ausschauen würde:

```
Dim VB$t_ref$S0 As New ArrayList
VB$t_ref$S0.Add("Waldi")
VB$t_ref$S0.Add("Basko")
VB$t_ref$S0.Add("Hasso")
VB$t_ref$S0.Add("Wauzi")
VB$t_ref$S0.Add("Bello")
VB$t_ref$S0.Add("Minka")
Dim locHundeNamenList As ArrayList = VB$t_ref$S0
```

Vereinfachung von Auflistungsinitialisierern durch Erweiterungsmethoden

Warum ist es so wichtig zu wissen, wie Auflistungsinitialisierer intern arbeiten? Ganz einfach: Was bei einzelnen, primitiven Datentypen noch so einfach funktioniert, wird bei komplexeren Klassen nicht wirklich zu einer Arbeitserleichterung, da Sie jedes Element, das Sie in die geschweifte Klammer schreiben, als vollständig initialisierte Instanz angeben müssen, so wie es das folgende Beispiel zeigt:

```
Module Module1

    Sub Main()

        Dim Adressen As New ArrayList() From
            {New Adresse With
                {.Vorname = "Ramona", .Nachname = "Leenings"},
            New Adresse With
                {.Vorname = "Arnold", .Nachname = "Löffelmann"},
            New Adresse With
                {.Vorname = "Margarete", .Nachname = "Schindler"}}

    End Sub

    Public Class Adresse
        Public Property Vorname As String
        Public Property Nachname As String
```

```
        Public Property PLZ As String
        Public Property Ort As String
    End Class

End Module
```

Da wir wissen, dass die Add-Methode ausschlaggebend ist für das Hinzufügen von Elementen aus der geschweiften Klammer hinter From, würden wir den Umgang mit Auflistungsinitialisierern vereinfachen, wenn wir steuern könnten, in welcher Form die Add-Methode Parameter für das Hinzufügen von Auflistungselementen entgegennimmt. Mit Erweiterungsmethoden, wie wir sie in Kapitel 19 kennen gelernt haben, wird aber das genau möglich, wie das folgende Beispiel zeigt:

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 20\CollectionInitializers
```

Zum schon vorhandenen Beispiel, dessen Listing Sie oben sehen, fügen wir ein Modul mit einer Erweiterungsmethode hinzu. Wichtig: Da sie das Extension-Attribut verwendet, muss der Namespace, in der sich dieses Attribut befindet (System.Runtime.CompilerServices) mit eingebunden werden.

```
Module AdressenCollectionInitializer

    'Das Extension-Attribut kennzeichnet eine Methode
    'als Erweiterungsmethode. Der Typ des ersten Parameters
    'bestimmt den Typ, der erweitert werden soll.
    <Extension()> _
    Public Sub Add(ByVal adressenListe As ArrayList,
                   ByVal Vorname As String,
                   ByVal Nachname As String)

        adressenListe.Add(New Adresse With {.Vorname = Vorname,
                                            .Nachname = Nachname})
    End Sub

End Module
```

Da der Add-Methode als erster Parameter eine ArrayList übergeben wird und die Methode mit dem Attribut Extension gekennzeichnet ist, erweitert sie die Klasse ArrayList um eine Add-Methode, die zwei String-Parameter übernimmt. Im Grunde genommen überlädt sie also die Add-Methode der ArrayList mit einer neuen Version. Innerhalb dieser Methode wird dann erst die eigentliche Instanz der Adresse-Klasse angelegt und schließlich der ArrayList hinzugefügt. Im Ergebnis erlaubt uns das, die neue Add-Methode indirekt auch in Auflistungsinitialisierern anzugeben:

```
        Dim andereAdressen As New ArrayList() From
            {{"Klaus", "Löffelmann"},
             {"Andreas", "Belke"},
             {"Marcus", "Pusch"}}
```

HINWEIS `ArrayList` ist eine sehr allgemein gehaltene Klasse, die `Object` als Elementtyp entgegennimmt und deswegen nicht typsicher arbeitet. Sie sollten deswegen, gerade wenn Sie Auflistungsinitialisierer mit Erweiterungsmethoden vereinfachen, immer das generische Äquivalent vorziehen – in diesem Fall also beispielsweise `List(Of Adressen)`. Bei der Verwendung von Wertetypen hat das obendrein einen Performancevorteil, weil diese Typen, anders als bei `ArrayList`, nicht geboxt werden müssen.

Wichtige Auflistungen von .NET Framework, die Sie kennen sollten

.NET Framework enthält eine ganze Reihe Auflistungen, von denen Sie eine der wichtigsten – `ArrayList` – schon im Einsatz gesehen haben. In diesem Abschnitt möchte ich Ihnen die wichtigsten dieser Auflistungen kurz vorstellen und darauf hinweisen, für welchen Einsatz sie am besten geeignet sind oder welche Besonderheiten Sie bei ihrem Gebrauch beachten sollten.

ArrayList – universelle Ablage für Objekte

`ArrayList` können Sie als Container für Objekte aller Art verwenden. Sie ist zwar nicht die modernste Auflistung von .NET Framework, aber die grundlegendste und dienstälteste: Damit gibt sie quasi das Konzept von Auflistungen vor, und wenn Sie verstanden haben, wie `ArrayList` funktioniert, verstehen Sie die anderen Auflistungen ebenfalls.

Sie instanziieren ein `ArrayList`-Objekt und weisen ihm mithilfe seiner `Add`-Funktion das jeweils nächste Element zu. Mit der `Default`-Eigenschaft `Item` können Sie schon vorhandene Elemente abrufen oder neu definieren. `AddRange` erlaubt Ihnen, die Elemente einer vorhandenen `ArrayList` einer anderen `ArrayList` hinzuzufügen. Mit der `Count`-Eigenschaft eines `ArrayList`-Objekts finden Sie heraus, wie viele Elemente es beherbergt.

`Clear` löscht alle Elemente einer `ArrayList`. Mit `Remove` löschen Sie ein Objekt aus der `Arraylist`, das Sie als Parameter übergeben. Wenn mehrere gleiche Objekte (die `Equals`-Methode jedes Objekts wird dabei verwendet) existieren, wird das erste gefundene Objekt gelöscht. Mit `RemoveAt` löschen Sie ein Element an einer bestimmten Position. `RemoveRange` erlaubt Ihnen schließlich, einen ganzen Bereich von Array-Elementen ab einer bestimmten Position im `ArrayList`-Objekt zu löschen.

`ArrayList`-Objekte können in einfache Arrays umgewandelt werden. Dabei ist jedoch einiges zu beachten: Die Elemente der `ArrayList` müssen ausnahmslos alle dem Typ des Arrays entsprechen, in den sie umgewandelt werden sollen. Die Konvertierung nehmen Sie mit der `ToArray`-Methode des entsprechenden `ArrayList`-Objekts vor. Dabei bestimmen Sie, wenn Sie in ein typdefiniertes Array (wie `Integer()` oder `String()`) umwandeln, den Grundtyp (nicht den Array-Typ!) als zusätzlichen Parameter. Wenn Sie ein Array in eine `ArrayList` umwandeln wollen, verwenden Sie den jeweiligen Konstruktor der `ArrayList` – die entsprechende Konstruktorroutine nimmt anschließend die Konvertierung in eine `ArrayList` vor.

HINWEIS Bitte schauen Sie sich dazu auch das weiter unten gezeigte Listing an, insbesondere was die Konvertierungshinweise von `ArrayList`-Objekten in Arrays betrifft.

`ArrayList` implementiert die Schnittstelle `IEnumerable`. Aus diesem Grund stellt die Klasse einen Enumerator zur Verfügung, über den Sie mithilfe von `For Each` durch die Elemente der `ArrayList` iterieren können. Beachten Sie dabei, den richtigen Typ für die Schleifenvariable zu verwenden. `ArrayList`-Elemente sind nicht typsicher, und eine Typverletzung ist nur dann ausgeschlossen, wenn Sie genau wissen, welche Typen gespeichert sind (das nachfolgende Beispiel demonstriert diesen Fehler recht anschaulich). Hier die Beispiele, die das gerade Gesagte näher erläutern und den Code dazu dokumentieren:

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 20\CollectionsDemo
```

```
Sub ArrayListDemo()
    Dim locMännerNamen As String() = {"Jürgen", "Uwe", "Klaus", "Christian", "José"}
    Dim locFrauenNamen As New ArrayList
    Dim locNamen As ArrayList

    'ArrayList aus vorhandenem Array erstellen.
    locNamen = New ArrayList(locMännerNamen)

    'ArrayList mit Add auffüllen.
    locFrauenNamen.Add("Ute") : locFrauenNamen.Add("Miriam")
    locFrauenNamen.Add("Melanie") : locFrauenNamen.Add("Anja")
    locFrauenNamen.Add("Stephanie") : locFrauenNamen.Add("Heidrun")

    'Arraylist einer anderen Arraylist hinzufügen:
    locNamen.AddRange(locFrauenNamen)

    'Arraylist in eine Arraylist einfügen.
    Dim locHundenamen As String() = {"Hasso", "Bello", "Wauzi", "Wuffi", "Basko", "Franz"}
    'Einfügen *vor* dem 6. Element
    locNamen.InsertRange(5, locHundenamen)

    'ArrayList in ein Array zurückwandeln.
    Dim locAlleNamen As String()

    'Vorsicht: Fehler!
    'locAlleNamen = DirectCast(locNamen.ToArray, String())

    'Vorsicht: Ebenfalls Fehler!
    'locAlleNamen = DirectCast(locNamen.ToArray(GetType(String())), String())

    'So ist es richtig.
    locAlleNamen = DirectCast(locNamen.ToArray(GetType(String)), String())

    'Repeat legt eine Arraylist aus wiederholten Items an.
    locNamen.AddRange(ArrayList.Repeat("Dublettenname", 10))

    'Ein Element im Array ändern.
    locNamen(10) = "Fiffi"
    'Mit der Item-Eigenschaft geht es auch:
    locNamen.Item(13) = "Miriam"
```

```
        'Löschen des ersten zutreffenden Elements aus der Liste.
        locNamen.Remove("Basko")

        'Löschen eines Elements an einer bestimmten Position.
        locNamen.RemoveAt(4)

        'Löschen eines bestimmten Bereichs aus der ArrayList mit RemoveRange.
        'Count ermittelt die Anzahl der Elemente in der ArrayList.
        locNamen.RemoveRange(locNamen.Count - 6, 5)

        'Ausgeben der Elemente über die Default-Eigenschaft der ArrayList (Item).
        For i As Integer = 0 To locNamen.Count - 1
            Console.WriteLine("Der Name Nr. {0} lautet {1}", i, locNamen(i).ToString)
        Next

        'Etwas anderes als ein String-Objekt der ArrayList hinzufügen,
        'um den folgenden Fehler "vorzubereiten".
        locNamen.Add(New FileInfo("C:\TEST.TXT"))

        'Diese Schleife kann nicht bis zum Ende ausgeführt werden,
        'da ein Objekt nicht vom Typ String mit von der Partie ist!
        For Each einString As String In locNamen
            'Hier passiert irgendetwas mit dem String.
            'nicht von Interesse, deswegen kein Rückgabewert.
            einString.EndsWith("Peter")
        Next
        Console.ReadLine()
    End Sub
```

Wenn Sie dieses Beispiel laufen lassen, dann sehen Sie zunächst die erwarteten Ausgaben auf dem Bildschirm. Doch der Programmcode erreicht nie die Anweisung `Console.ReadLine`, um auf Ihre letzte Bestätigung zu warten. Stattdessen löst er eine Ausnahme aus, etwa wie in Abbildung 20.3 zu sehen.

Abbildung 20.3 Achten Sie beim Iterieren mit `For Each` durch eine Auflistung darauf, dass die Elemente dem Schleifenvariablentyp entsprechen, um solche Ausnahmen zu vermeiden!

Der Grund dafür: Das letzte Element in der `ArrayList` ist kein String. Aus diesem Grund wird die Ausnahme ausgelöst. Achten Sie deshalb stets darauf, dass die Schleifenvariable eines `For Each`-Konstrukts immer den Typen entspricht, die in einer Auflistung gespeichert sind.

HINWEIS Da `ArrayList` auf `Object` basiert, sind Instanzen dieser Klasse zum einen nicht typsicher, zum anderen muss sie Wertetypen aufwändig boxen, um sie zu speichern. Aus diesem Grund sollten Sie, wo es geht, auf die generischen Äquivalente `List(Of Type)` oder `Collection(Of Type)` zurückgreifen, die sich in den Namespaces `System.Collections.Generic` und `System.Collections.ObjectModel` befinden.

`ArrayList`-Instanzen verwenden Sie hingegen bei inhomogenen Auflistungen, also bei Auflistungen, die verschiedene Elemente komplett unterschiedlichen Typs speichern.

Typsichere Auflistungen auf Basis von CollectionBase

Sie können dem Problem aus dem vorherigen Abschnitt entgehen, indem Sie eine Auflistung mithilfe von *Generics* (*Generika*, auf Microsoft-Deutsch) dazu zwingen, homogen zu sein (also nur Elemente des gleichen Typs zu verarbeiten). Für den späteren Einsatz von *LINQ to Objects* (siehe Kapitel 26) ist das sogar eine Notwendigkeit. Es gibt aber eine andere, klassische Möglichkeit, die eine Technologie unter Zuhilfenahme von Schnittstellen nutzt, um eine Basisklasse zu implementieren, mit der Sie Ihre Ableitungen typsicher machen, und die eben nicht auf Generics basieren. Diese Technik können Sie immer dann einsetzen, wenn Sie aus welchen Gründen auch immer auf Generics verzichten müssen – beispielsweise wenn Sie eine ältere .NET Framework-Version verwenden müssen, die Generics nicht unterstützt, oder für bestimmte andere Plattformen entwickeln, bei denen Generics ebenfalls nicht zur Verfügung stehen.

.NET Framework bietet zu diesem Zweck eine abstrakte Klasse namens `CollectionBase` als Vorlage an, die Sie für diese Zwecke ableiten und erweitern können.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 20\TypeSafeCollections
```

Der folgende Code zeigt, wie die `Item`-Eigenschaft und die `Add`-Methode in einer benutzerdefinierten Auflistung realisiert sind:

```
Public Class Adressen
    Inherits CollectionBase

    Public Overridable Function Add(ByVal Adr As Adresse) As Integer
        Return MyBase.List.Add(Adr)
    End Function

    Default Public Overridable Property Item(ByVal Index As Integer) As Adresse
        Get
            Return DirectCast(MyBase.List(Index), Adresse)
        End Get
        Set(ByVal Value As Adresse)
            MyBase.List(Index) = Value
        End Set
    End Property
End Class
```

Das entsprechende Beispielprogramm, das die Klasse testet, sieht folgendermaßen aus:

```
Module TypesafeCollections

    Sub Main()
        Dim locAdressen As New Adressen
        Dim locAdresse As New Adresse("Christian", "Sonntag", "99999", "Trinken")
        Dim locAndererTyp As New FileInfo("C:\Test.txt")

        'Kein Problem:
        locAdressen.Add(locAdresse)

        'Schon der Editor mault!
        'locAdressen.Add(locAndererTyp)

        'Auch kein Problem.
        locAdresse = locAdressen(0)

        For Each eineAdresse As Adresse In locAdressen
            Console.WriteLine(eineAdresse)
        Next
        Console.ReadLine()
    End Sub

End Module
```

Sie sehen anhand des Testprogramms, dass Typsicherheit in dieser Auflistung gewährleistet ist. Allerdings sollte dieses Programm, wenn Sie den OOP-Teil dieses Buchs aufmerksam studiert haben, auch Fragen aufwerfen: Wenn Sie sich die Beschreibung zu `CollectionBase` in der Onlinehilfe ansehen, finden Sie dort folgenden Prototyp:

```
<Serializable>
MustInherit Public Class CollectionBase
   Implements IList, ICollection, IEnumerable
.
.
.
```

`CollectionBase` bindet unter anderem die Schnittstelle `IList` ein. Die Schnittstelle `IList` schreibt vor, dass eine `Add`-Funktion mit der folgenden Signatur in der Klasse implementiert sein muss, die sie implementiert:

```
Function Add(ByVal value As Object) As Integer
```

Erkennen Sie die Unregelmäßigkeit? Unsere Klasse erbt von `CollectionBase`, Letztere bindet `IList` ein, `IList` wiederum verlangt die Methode `Add`, in unserer Klasse müsste demzufolge eine `Add`-Methode vorhanden sein, die wir zu überschreiben hätten! Aber sie ist es nicht, und das ist auch gut so, denn: Wäre sie vorhanden, dann müssten wir ihre Signatur zum Überschreiben übernehmen. Als Parameter nimmt sie aber ein `Object` entgegen – unsere Typsicherheit wäre dahin. Alternativ könnten wir sie überladen, aber in diesem Fall könnte man ihr dennoch ein `Object` übergeben – wieder wäre es aus mit der Typsicherheit.

Wir kommen dem Geheimnis auf die Spur, wenn wir uns die `Add`-Funktion unserer neuen `Adressen`-Auflistung ein wenig genauer anschauen. Dort heißt es:

```
    Public Overridable Function Add(ByVal Adr As Adresse) As Integer
        Return MyBase.List.Add(Adr)
    End Function
```

`MyBase` greift auf die Basisklasse zurück und verwendet letzten Endes die `Add`-Funktion des Objekts, das die `List`-Eigenschaft zurückliefert, um das hinzuzufügende Element weiterzureichen. Was aber macht `List`? Welches Objekt liefert es zurück? Um Sie vielleicht zunächst komplett zu verwirren: `List` liefert die Instanz unserer Klasse zurück[4] – und wir betreiben hier Polymorphie in Vollendung!

Schauen wir, was die Onlinehilfe von Visual Studio zur Beschreibung von `List` zu sagen hat: »Ruft eine `IList` mit der Liste der Elemente in der `CollectionBase`-Instanz ab.« – Wow, das ist informativ!

Der ganze Umstand wird klar, wenn Sie sich das folgende Konstrukt anschauen. Behalten Sie dabei im Hinterkopf, dass eine Möglichkeit gefunden werden muss, die `Add`-Funktion einer Schnittstelle in einer Klasse zu implementieren, ohne einer sie einbindenden Klasse die Möglichkeit zu nehmen, eine `Add`-Funktion mit einer ganz anderen Signatur zur Verfügung zu stellen:

```
Interface ITest
    Function Add(ByVal obj As Object) As Integer
End Interface

MustInherit Class ITestKlasse
    Implements ITest

    Private Function ITestAdd(ByVal obj As Object) As Integer Implements ITest.Add
        Trace.WriteLine("ITestAdd:" + obj.ToString)
    End Function

End Class

Class ITestKlasseAbleitung
    Inherits ITestKlasse

    Public ReadOnly Property Test() As ITest
        Get
            Return DirectCast(Me, ITest)
        End Get
    End Property

    Public Sub Add(ByVal TypeSafe As Adresse)
        Test.Add(TypeSafe)
    End Sub
End Class
```

[4] Dieses Konstrukt erinnerte mich spontan an eine Star Trek-Voyager-Episode, in der Tom Paris mit B'Elanna Torres das Holodeck besucht, um mit ihr in einem holografisch projizierten Kino einen »altertümlichen« Film mit 3-D-Brille zu sehen. Ihr Kommentar dazu: »Lass mich das mal klarstellen: Du hast diesen Aufwand betrieben, um eine dreidimensionale Umgebung so zu programmieren, dass sie ein zweidimensionales Bild projiziert, und nun bittest Du mich, diese Brille zu tragen, damit es wieder dreidimensional ausschaut?« …

Die Schnittstelle dieser Klasse schreibt vor, dass eine Klasse, die diese Schnittstelle einbindet, eine Add-Funktion implementieren muss. Die abstrakte Klasse ITestKlasse bindet diese Schnittstelle auch ordnungsgemäß ein – allerdings stellt sie die Funktion nicht der Öffentlichkeit zur Verfügung; die Funktion ist dort nämlich als Private definiert. Außerdem – und das ist der springende Punkt – nennt sich die Funktion nicht Add, sondern ITestAdd – möglich wird das, da Schnittstelleneinbindungen in Visual Basic grundsätzlich explizit funktionieren, genauer gesagt durch das Schlüsselwort Implements am Ende der Funktionsdeklaration.

Halten wir fest: Wir haben nun eine saubere Schnittstellenimplementierung, *und* wir können die der Schnittstelle zugeordnete Funktion von außen nicht sehen. Wie rufen wir die Funktion ITestAdd der Basisklasse ITestKlasse dann überhaupt auf? Des Rätsels Lösung ist: Die Funktion ist nicht ganz so privat, wie sie scheint. Denn wir können über eine Schnittstellenvariable auf sie zugreifen. Wenn wir die eigene Instanz der Klasse in eine Schnittstellenvariable vom Typ ITest casten, dann haben wir über die Schnittstellenfunktion ITest.Add dennoch auf die private Funktion ITestAdd der Klasse ITestKlasse Zugriff – und wir sind am Ziel!

Zu unserer Bequemlichkeit stellt die abgeleitete Klasse ITestKlasseAbleitung bereits eine Eigenschaft (Property Test) zur Verfügung, die die aktuelle Klasseninstanz als ITest-Schnittstelle zurückliefert. Wir können uns dieser also direkt bedienen.

Genau das Gleiche haben wir in unserer aus CollectionBase entstandenen Klasse Adressen gemacht – um zum alten Beispiel zurückzukommen. Die List-Eigenschaft der Klasse CollectionBase entspricht der Test-Eigenschaft der Klasse ITestKlasse unseres Erklärungsbeispiels.

Hashtables – für das Nachschlagen von Objekten

Hashtable-Objekte sind das ideale Werkzeug, wenn Sie eine Datensammlung aufbauen wollen, aber die einzelnen Objekte nicht durch einen numerischen Index, sondern durch einen Schlüssel abrufen wollen. Ein Beispiel soll das verdeutlichen.

Angenommen, Sie haben eine Adressenverwaltung programmiert, bei der Sie einzelne Adressen durch eine Art Matchcode abrufen wollen (Kundennummer, Lieferantennummer, was auch immer). Bei der Verwendung einer ArrayList müssten Sie schon einigen Aufwand betreiben, um an ein Array-Element auf Basis des Matchcode-Namens zu gelangen: Sie müssten zum Finden eines Elements in der Liste für die verwendete Adressklasse eine CompareTo-Methode implementieren, damit die Liste mittels Sort sortiert werden könnte. Anschließend könnten Sie mit BinarySearch das Element finden – vorausgesetzt, die CompareTo-Methode würde eine Instanz der Adressklasse über ihren Matchcode-Wert vergleichen.

Hashtable-Objekte vereinfachen ein solches Szenario ungemein: Wenn Sie einer Hashtable ein Objekt hinzufügen, dann nimmt dessen Add-Methode nicht nur das zu speichernde Objekt (den hinzuzufügenden Wert) entgegen, sondern auch ein weiteres. Dieses zusätzliche Objekt (das genau genommen als erster Parameter übergeben wird) stellt den Schlüssel – den Key – zum Wiederauffinden des Objekts dar. Sie rufen ein Objekt aus einer Hashtable anschließend nicht wie bei der ArrayList mit

```
Element = eineArrayList(5)
```

ab, sondern mit dem entsprechenden Schlüssel, etwa:

```
Element = eineHashtable("ElementKey")
```

Voraussetzung bei diesem Beispiel ist natürlich, dass der Schlüssel zuvor ein entsprechender String gewesen ist.

Anwenden von Hashtables

BEGLEITDATEIEN Im Verzeichnis

`...\VB 2010 Entwicklerbuch\Kapitel 20\HashtableDemo01`

finden Sie ein Beispielprojekt, das den Einsatz der `Hashtable`-Klasse demonstrieren soll. Es enthält eine Klasse, die eine Datenstruktur – eine Adresse – abbildet. Eine statische Funktion erlaubt darüber hinaus, eine beliebige Anzahl von Zufallsadressen zu erstellen, die zunächst in einer `ArrayList` gespeichert werden. Eine solche mit zufälligen Daten gefüllte `ArrayList` soll zum Experimentieren mit der `Hashtable`-Klasse dienen. Der Vollständigkeit halber (und des späteren einfacheren Verständnisses) möchte ich Ihnen diese `Adresse`-Klasse kurz vorstellen – wenn auch in verkürzter Form):

WICHTIG Sie benötigen mindestens 1 GB Arbeitsspeicher unter Windows XP und 2 GB unter Windows Vista oder Windows 7, damit Sie die folgenden Hashtable-Beispiele nachvollziehen können. Die Menge des zu reservierenden Speichers für das Beispiel ist deshalb so groß, weil durch die hohen Prozessorgeschwindigkeiten eine genügend große Anzahl von Elementen vorhanden sein muss, um überhaupt Geschwindigkeitsvergleiche anstellen zu können. Gleichzeitig muss aber so viel echter Speicher vorhanden sein, dass durch die schiere Masse an Daten die Auslagerungsdatei von Windows nicht in Anspruch genommen werden muss, was die Ergebnisse natürlich komplett verzerren würde.

```
Public Class Adresse

    'Konstruktor - legt eine neue Instanz an
    Sub New(ByVal Matchcode As String, ByVal Name As String,
            ByVal Vorname As String, ByVal Plz As String, ByVal Ort As String)
        Me.Matchcode = Matchcode
        Me.Name = Name
        Me.Vorname = Vorname
        Me.PLZ = Plz
        Me.Ort = Ort
    End Sub

    Public Overridable Property Matchcode() As String
    Public Overridable Property Name() As String
    Public Overridable Property Vorname() As String
    Public Overridable Property PLZ() As String
    Public Overridable Property Ort() As String

    Public Overrides Function ToString() As String
        Return Matchcode + ": " + Name + ", " + Vorname + ", " + PLZ + " " + Ort
    End Function

    Public Shared Function ZufallsAdressen(ByVal Anzahl As Integer) As ArrayList

        Dim locArrayList As New ArrayList(Anzahl)
        Dim locRandom As New Random(Now.Millisecond)

        Dim locNachnamen As String() = {"Heckhuis", "Löffelmann", "Thiemann", "Müller",
                    "Meier", "Tiemann", "Sonntag", "Ademmer", "Westermann", "Vüllers",
                    "Hollmann", "Vielstedde", "Weigel", "Weichel", "Weichelt", "Hoffmann",
                    "Rode", "Trouw", "Schindler", "Neumann", "Jungemann", "Hörstmann",
                    "Tinoco", "Albrecht", "Langenbach", "Braun", "Plenge", "Englisch",
                    "Clarke"}
```

```
        Dim locVornamen As String() = {"Jürgen", "Gabriele", "Uwe", "Katrin", "Hans",
                    "Rainer", "Christian", "Uta", "Michaela", "Franz", "Anne", "Anja",
                    "Theo", "Momo", "Katrin", "Guido", "Barbara", "Bernhard", "Margarete",
                    "Alfred", "Melanie", "Britta", "José", "Thomas", "Daja", "Klaus", "Axel",
                    "Lothar", "Gareth"}
        Dim locStädte As String() = {"Wuppertal", "Dortmund", "Lippstadt", "Soest",
                    "Liebenburg", "Hildesheim", "München", "Berlin", "Rheda", "Bielefeld",
                    "Braunschweig", "Unterschleißheim", "Wiesbaden", "Straubing",
                    "Bad Waldliesborn", "Lippetal", "Stirpe", "Erwitte"}

        For i As Integer = 1 To Anzahl
            Dim locName, locVorname, locMatchcode As String
            locName = locNachnamen(locRandom.Next(locNachnamen.Length - 1))
            locVorname = locVornamen(locRandom.Next(locNachnamen.Length - 1))
            locMatchcode = locName.Substring(0, 2)
            locMatchcode += locVorname.Substring(0, 2)
            locMatchcode += i.ToString("00000000")
            locArrayList.Add(New Adresse(
                            locMatchcode,
                            locName,
                            locVorname,
                            locRandom.Next(99999).ToString("00000"),
                            locStädte(locRandom.Next(locStädte.Length - 1))))

        Next
        Return locArrayList
    End Function

    Shared Sub AdressenAusgeben(ByVal Adressen As ArrayList)
        For Each Item As Object In Adressen
            Console.WriteLine(Item)
        Next
    End Sub

End Class
```

Die eigentliche Klasse zum Speichern einer Adresse ist Kinderkram. Wichtig ist, dass Sie wissen, welche besondere Bewandtnis es mit der Eigenschaft `Matchcode` dieser Adressenklasse hat. `Matchcode` ist in dieser Klasse quasi der Primärschlüssel und damit in einem Satz von Adressen immer eindeutig. Die Prozedur in diesem Beispiel, die zufällige Adressen erzeugt, stellt das sicher. Der Matchcode setzt sich aus den ersten beiden Buchstaben des Nachnamens, den ersten beiden Buchstaben des Vornamens und einer fortlaufenden Nummer zusammen. Würden Sie 15 verschiedene Zufallsadressen mit diesem Code

```
    Sub AdressenTesten()

        '15 zufällige Adressen erzeugen.
        Dim locDemoAdressen As ArrayList = Adresse.ZufallsAdressen(15)

        'Adressen im Konsolenfenster ausgeben.
        Console.WriteLine("Liste mit zufällig erzeugten Personendaten")
        Console.WriteLine(New String("="c, 30))
        Adresse.AdressenAusgeben(locDemoAdressen)

    End Sub
```

erstellen und ausgeben, sähen Sie etwa folgendes Ergebnis im Konsolenfenster:

```
Liste mit zufällig erzeugten Personendaten
==============================
HeMo00000001: Heckhuis, Momo, 06549 Straubing
SoGu00000002: Sonntag, Guido, 21498 Liebenburg
ThAl00000003: Thiemann, Alfred, 51920 Bielefeld
HöJü00000004: Hörstmann, Jürgen, 05984 Liebenburg
TiMa00000005: Tiemann, Margarete, 14399 München
TiAn00000006: Tiemann, Anja, 01287 Dortmund
ViGu00000007: Vielstedde, Guido, 72762 Wuppertal
RoMe00000008: Rode, Melanie, 94506 Hildesheim
TiDa00000009: Tiemann, Daja, 54134 Lippstadt
BrJo00000010: Braun, José, 14590 Soest
WeJü00000011: Westermann, Jürgen, 83128 Wuppertal
HeKa00000012: Heckhuis, Katrin, 13267 Bad Waldliesborn
TrJü00000013: Trouw, Jürgen, 54030 Lippstadt
PlGa00000014: Plenge, Gabriele, 97702 Braunschweig
WeJü00000015: Weichel, Jürgen, 39992 Unterschleißheim
```

Falls Sie sich nun fragen, wieso ich Ihnen so viel Vorgeplänkel erzähle: Es ist wichtig für die richtige Anwendung von `Hashtable`-Objekten und das richtige Verständnis, wie Elemente in Hashtables gespeichert werden. Denn wenn Sie ein Objekt in einer Hashtable speichern, *muss* der Schlüssel eindeutig sein – anderenfalls hätten Sie nicht die Möglichkeit, wieder an alle Elemente heranzukommen. (Welches von zwei Elementen sollte die Hashtable schließlich durch einen Schlüssel indiziert zurückliefern, wenn die Schlüssel die gleichen wären?)

Verarbeitungsgeschwindigkeiten von Hashtables

Jetzt lassen Sie uns eine Hashtable in Aktion sehen und schauen, was sie zu leisten vermag. Aus den Elementen der `ArrayList`, die uns die `ZufallsAdressen`-Funktion liefert, bauen wir eine Hashtable mit nicht weniger als 1.000.000 Elementen auf. Gleichzeitig erzeugen wir ein weiteres Array mit 51 zufälligen Elementen der `ArrayList` und merken uns deren Matchcode. Diese Matchcodes verwenden wir anschließend, um uns Elemente aus der Liste herauszupicken und messen dabei die Zeit, die das Zugreifen benötigt. Das Programm dazu sieht folgendermaßen aus:

```
Module HashtableDemo

    Sub Main()

        ''Für die weiteren Beispiele kommentieren Sie die folgenden drei Zeilen aus.
        'AdressenTesten()
        'Console.ReadLine()
        'Return

        Dim locAnzahlAdressen As Integer = 1000000
        Dim locZugriffsElemente As Integer = 2
        Dim locMessungen As Integer = 3
        Dim locZugriffe As Integer = 1000000
        Dim locVorlageAdressen As ArrayList
        Dim locAdressen As New Hashtable
        Dim locTestKeys(locZugriffsElemente) As String
        Dim locRandom As New Random(Now.Millisecond)
```

```
        'Warten auf Startschuss
        Console.WriteLine("Drücken Sie Return, um zu beginnen")
        Console.ReadLine()

        'Viele Adressen erzeugen:
        Console.Write("Erzeugen von {0} zufälligen Adresseneinträgen...", locAnzahlAdressen)
        Dim locZeitmesser = Stopwatch.StartNew
        locVorlageAdressen = adresse.ZufallsAdressen(locAnzahlAdressen)
        locZeitmesser.Stop()
        Console.WriteLine("fertig nach {0} ms", locZeitmesser.ElapsedMilliseconds)
        locZeitmesser.Reset()

        'Aufbauen der Hashtable
        Console.Write("Aufbauen der Hashtable mit zufälligen Adresseneinträgen...", locAnzahlAdressen)
        locZeitmesser.Start()
        For Each adresse As Adresse In locVorlageAdressen
            locAdressen.Add(adresse.Matchcode, adresse)
        Next
        locZeitmesser.Stop()
        Console.WriteLine("fertig nach {0} ms", locZeitmesser.ElapsedMilliseconds)
        locZeitmesser.Reset()

        'n zufällige Adressen 'rauspicken
        For i As Integer = 0 To locZugriffsElemente
            locTestKeys(i) = DirectCast(locVorlageAdressen(locRandom.Next(locAnzahlAdressen)),
                             Adresse).Matchcode
        Next

        Dim locTemp As Object
        Dim locTemp2 As Object = Nothing

        'Zugreifen und Messen, wie lange das dauert
        'Das ganze 5 Mal, um die Messung zu bestätigen
        For z As Integer = 1 To locMessungen
            Console.WriteLine()
            Console.WriteLine("{0}. Messung:", z)
            For i As Integer = 0 To locZugriffsElemente
                Console.Write("{0} Zugriffe auf: {1} in ", locZugriffe, locTestKeys(i))
                locTemp = locTestKeys(i)
                locZeitmesser.Start()
                For j As Integer = 1 To locZugriffe
                    locTemp2 = locAdressen(locTemp)
                Next j
                locZeitmesser.Stop()
                locTemp = locTemp2.GetType
                Console.WriteLine("{0} ms", locZeitmesser.ElapsedMilliseconds)
                locZeitmesser.Reset()
            Next

            'Zugriff auf Arraylist für Vergleich
            For i As Integer = 0 To locZugriffsElemente
                Console.Write("{0} Zugriffe auf ArrayList-Element in ", locZugriffe)
                locZeitmesser.Start()
                For j As Integer = 1 To locZugriffe
                    locTemp2 = locVorlageAdressen(0)
```

```
                Next j
                locZeitmesser.Stop()
                locTemp = locTemp2.GetType
                Console.WriteLine("{0} ms", locZeitmesser.ElapsedMilliseconds)
                locZeitmesser.Reset()
            Next

        Console.ReadLine()
    End Sub
```

Wenn Sie das Programm starten, erzeugt es eine Ausgabe, etwa wie im folgenden Bildschirmauszug:[5]

```
Drücken Sie Return, um zu beginnen

Erzeugen von 1000000 zufälligen Adresseneinträgen...fertig nach 2105 ms
Aufbauen der Hashtable mit zufälligen Adresseneinträgen...fertig nach 584 ms

1. Messung:
1000000 Zugriffe auf: WeMo00182238 in 112 ms
1000000 Zugriffe auf: PlMa00662840 in 132 ms
1000000 Zugriffe auf: WeLo00244369 in 131 ms
1000000 Zugriffe auf ArrayList-Element in 26 ms
1000000 Zugriffe auf ArrayList-Element in 30 ms
1000000 Zugriffe auf ArrayList-Element in 27 ms

2. Messung:
1000000 Zugriffe auf: WeMo00182238 in 142 ms
1000000 Zugriffe auf: PlMa00662840 in 134 ms
1000000 Zugriffe auf: WeLo00244369 in 111 ms
1000000 Zugriffe auf ArrayList-Element in 25 ms
1000000 Zugriffe auf ArrayList-Element in 26 ms
1000000 Zugriffe auf ArrayList-Element in 25 ms

3. Messung:
1000000 Zugriffe auf: WeMo00182238 in 104 ms
1000000 Zugriffe auf: PlMa00662840 in 133 ms
1000000 Zugriffe auf: WeLo00244369 in 102 ms
1000000 Zugriffe auf ArrayList-Element in 25 ms
1000000 Zugriffe auf ArrayList-Element in 26 ms
1000000 Zugriffe auf ArrayList-Element in 26 ms
```

Nach dem Programmstart legt das Programm hier im Beispiel 1.000.000 Testelemente an und baut daraus die Hashtable auf. Anschließend pickt es sich drei Beispieleinträge heraus und misst die Zeit, die es für 1.000.000 Zugriffe auf jeweils diese Elemente der Hashtable benötigt. Das Ganze macht es dreimal, um eine Konstanz in den Verarbeitungsgeschwindigkeiten sicherzustellen. Um im Gegenzug nachzuweisen, wie schnell ein indizierter Zugriff auf ein `ArrayList`-Element erfolgt, führt es eine Messung dazu anschließend durch. Das Ergebnis ist beeindruckend: Der Zugriff auf ein Element über seinen Schlüssel ist nur rund 4 Mal so langsam wie der direkte Zugriff über den Index. Dieses Ergebnis wäre nicht möglich, wenn man die gesamte Hashtable nach dem Key von vorne bis hinten durchsuchen würde. Und was auch auffällt:

[5] Notebook mit 2,5 GHz Dual Core T9300, mit 2,5 GHz Takt, 4 GB Hauptspeicher unter der 64 Bit-Version von Windows 7.

Wieso die Zugriffszeit auf Hashtable-Elemente nahezu konstant ist ...

Sie können die Parameter am Anfang des Programms verändern, um weitere Eindrücke der unglaublichen Geschwindigkeit von .NET zu sammeln. Sie werden bei allen Experimenten jedoch eines herausfinden: Ganz gleich, wie Sie auch an den Schrauben drehen, die Zugriffsgeschwindigkeit auf ein einzelnes Element bleibt nahezu konstant. In den seltensten Fällen gibt es Ausreißer, bei denen der Zugriff auf ein Element das Doppelte der Durchschnittszeit benötigt.

Das Geheimnis für die auf der einen Seite sehr hohe, auf der anderen Seite durchschnittlich gleich bleibende Geschwindigkeit beim Zugriff liegt am Aufbau der Hashtable und an der Behandlung der Schlüssel. Die schnellste Art und Weise, auf ein Element eines Arrays zuzugreifen, ist das direkte Auslesen des Elements über seinen Index (auch das hat das vorherige Beispiel mit dem Zugriff auf die `ArrayList`-Elemente gezeigt). Daraus folgt, dass es am günstigsten ist, die Elemente der Hashtable nicht nach einem Schlüssel durchsuchen zu müssen, sondern die Positionsnummer eines Elements der Hashtable irgendwie zu berechnen. Und genau hier setzt das *Hashing*-Konzept an. *Hashing* bedeutet eigentlich »zerhacken«. Das klingt sehr negativ, doch das Zerhacken des Schlüssels, das auf eine bestimmte Weise tatsächlich stattfindet, dient hier einem konstruktiven Zweck: Wenn der Schlüssel, der beispielsweise in Form einer Zeichenkette vorliegt, *gehashed* wird, geht daraus eine Kennzahl hervor, die Aufschluss über die Wertigkeit der Zeichenkette gibt. *Wertigkeit* in diesem Zusammenhang bedeutet nicht nur, dass gleiche Zeichenketten gleiche Hashwerte ausweisen, sondern auch, dass größere Zeichenketten größere Hashwerte bedeuten.

Ein einfaches Beispiel soll diese Zusammenhänge klarstellen: Angenommen, Sie haben 26 Wörter, die alle mit einem anderen Buchstaben beginnen. Diese Wörter liegen in unsortierter Reihenfolge vor. Nehmen wir weiter an, dass Ihr Hashalgorithmus ganz einfach gestrickt ist: Der Hashcode einer Zeichenfolge entspricht der Nummer des Anfangsbuchstabens jedes Wortes. In dieser vereinfachten Konstellation haben Sie das Problem des Positionsberechnens bereits gelöst. Ihr Hashcode ist die Indexnummer im Array; sowohl das Einsortieren als auch das spätere Auslesen funktioniert in Windeseile über die Zeichenfolge selbst (genauer über seinen Anfangsbuchstaben).

Das Problem gestaltet sich in der Praxis natürlich nicht ganz so einfach. Mehr Zeichen (um beim Beispiel Zeichenfolgen für Schlüssel zu bleiben) müssen berücksichtigt werden, um den Hashcode zu berechnen, und bei langen Zeichenketten und begrenzter Hashcodepräzision kann man nicht ausschließen, dass unterschiedliche Zeichenketten gleiche Hashcodes ergeben. In diesem Fall müssen Kollisionen bei der Indexberechnung berücksichtigt werden. So groß gestaltet sich das Problem aber gar nicht, denn wenn beispielsweise beim Einsortieren der Elemente der sich durch den Hashcode des Schlüssels ergebende Index bereits belegt ist, nimmt man eben den nächsten freien.

Um beim Beispiel zu bleiben: Angenommen, Sie haben nun 26 Elemente, von denen alle mit einem anderen Anfangsbuchstaben beginnen, mit einer Unregelmäßigkeit: Es gibt zwei A-Wörter, ein B-Wort und kein C-Wort. Der Hash-Algorithmus bleibt der gleiche. Sie sortieren das B-Wort in Slot 2, anschließend das erste A-Wort in Slot 1. Nun bekommen Sie das zweite A-Wort zum Einsortieren, und laut Hashcode zeigt es auch auf Slot 1. In diesem Fall fangen Sie an zu suchen und testen Slot 2, der durch das B-Wort schon belegt ist, aber Sie finden anschließend einen freien Slot 3. Der gehört eigentlich zu C, aber in diesem Fall ist er momentan nicht nur frei, sondern wird auch nie beansprucht werden, da es kein C-Wort in der einzusortierenden Liste gibt. Im ungünstigsten Fall gibt es in diesem Beispiel zwar ein C-Wort, dafür aber kein Z-Wort, und das zweite A-Wort wird als letztes Element einsortiert. Jetzt verläuft die Suche über alle Elemente und landet schließlich auf dem letzten Elemente für das *Z*.

... und wieso Sie das wissen sollten!

Sie können sich die Reduzierung solcher Fälle mit zusätzlichem Speicherplatz erkaufen. Angenommen, Sie reservieren in unserem Beispiel doppelt so viel Speicher für die Schlüssel, dann sind zwar ganz viele Slots leer, aber das zweite A-Wort muss nicht den ganzen Weg bis zum Z-Index antreten. Was ganz wichtig ist: Sie wissen jetzt, was ein *Load-Faktor* ist. So nennt man nämlich den Faktor, der das Verhältnis zwischen Wahrscheinlichkeiten von Zuordnungskollisionen und benötigtem Speicher angibt. Weniger Kollisionswahrscheinlichkeit erfordert höheren Speicherbedarf und umgekehrt. Und Sie können diesen Zusammenhang auch am Beispielprogramm austesten.

Am Anfang des Moduls in der `Sub Main` des Beispielprogramms finden Sie einen Block mit auskommentierten Deklarationsanweisungen. Tauschen Sie die Auskommentierung der beiden Blöcke, um folgende Parameter für den nächsten Versuch zugrunde zu legen:

```
Dim locAnzahlAdressen As Integer = 1000000
Dim locZugriffsElemente As Integer = 25
Dim locMessungen As Integer = 3
Dim locZugriffe As Integer = 1000000
Dim locVorlageAdressen As ArrayList
Dim locAdressen As New Hashtable(100000, 1)
Dim locTestKeys(locZugriffsElemente) As String
Dim locZeitmesser As New HighSpeedTimeGauge
Dim locRandom As New Random(Now.Millisecond)
```

Mit diesem Block erhöhen wir die Anzahl der Elemente, die es zu testen gilt, auf 50, und damit erhöhen wir natürlich auch die Wahrscheinlichkeit, Elemente zu finden, die beim Anlegen kollidiert sind. Gleichzeitig verändern wir – im Codelisting fett markiert – den Load-Faktor der Hashtable. Bei der Instanziierung einer Hashtable können Sie, neben der Anfangskapazität, als zweiten Parameter den Load-Faktor bestimmen. Gültig sind dabei Werte zwischen 0.1 und 1. Für den ersten Durchlauf bestimmen wir einen Load-Faktor von 1. Dabei wird auf Speicherplatz auf Kosten von vielen zu erwartenden Kollisionen und damit auf Kosten von Geschwindigkeit verzichtet (dieser Wert entspricht übrigens der Voreinstellung, die dann verwendet wird, wenn Sie keinen Parameter für den Load-Faktor angeben).

Starten Sie das Programm mit diesen Einstellungen und beenden Sie es zunächst ***nicht***, nachdem der Messdurchlauf abgeschlossen wurde! Je nach Leistungsfähigkeit Ihres Computers sehen Sie bei der Ausgabe der Testergebnisse Messungen, von denen ich nur einige ausschnittsweise im Folgenden zeigen möchte:

```
1000000 Zugriffe auf: BrFr00803543 in 127 ms
1000000 Zugriffe auf: HoKl00971779 in 105 ms
1000000 Zugriffe auf: TiAn00984539 in 160 ms
1000000 Zugriffe auf: TiKl00353084 in 104 ms
1000000 Zugriffe auf: HoKa00471671 in 109 ms
1000000 Zugriffe auf: ViBe00059525 in 106 ms
1000000 Zugriffe auf: WeKa00412662 in 119 ms
1000000 Zugriffe auf: TiBr00982230 in 153 ms
1000000 Zugriffe auf: BrBa00457658 in 103 ms
1000000 Zugriffe auf: HoAn00031026 in 106 ms
1000000 Zugriffe auf: TrMo00777434 in 177 ms
1000000 Zugriffe auf: ThMi00652362 in 112 ms
1000000 Zugriffe auf: WeUw00237110 in 117 ms
```

```
1000000 Zugriffe auf: HeBr00835839 in 106 ms
1000000 Zugriffe auf: VüBr00254760 in 134 ms
1000000 Zugriffe auf: LöMa00309629 in 127 ms
1000000 Zugriffe auf: ScUt00344455 in 102 ms
1000000 Zugriffe auf: WeBa00812888 in 104 ms
```

Sie erkennen, dass die Zugriffsdauer auf die Elemente in dieser Liste erheblich streut. Die Zugriffszeit auf einige Elemente liegt teilweise doppelt so hoch wie im Durchschnitt.

Rufen Sie nun, ohne das Programm zu beenden, den Task-Manager. Dazu klicken Sie mit der rechten Maustaste auf die Task-Leiste (auf einen freien Bereich neben dem Startmenü) und wählen den entsprechenden Menüeintrag aus.

Wenn der Task-Manager-Dialog dargestellt ist, wechseln Sie auf die Registerkarte *Prozesse*. Suchen Sie den Eintrag *HashtableDemo*, und merken Sie sich zunächst den Wert, der dort als Speicherauslastung angegeben wurde (etwa wie in Abbildung 20.4 zu sehen).

Abbildung 20.4 Bei einem großen Wert für den Parameter Load-Faktor einer Hashtable hält sich der Speicherbedarf in Grenzen, dafür gibt es mehr Zuordnungskollisionen, die Zeit kosten ...

Beenden Sie das Programm anschließend. Jetzt verändern Sie den Load-Faktor der Hashtable auf den Wert 0.1 und wiederholen die Prozedur:

```
        Dim locAdressen As New Hashtable(100000, 0.1)
```

Beobachten Sie anschließend, wie sich der kleinere Load-Faktor auf die Zugriffsgeschwindigkeit der Elemente ausgewirkt hat:

```
1000000 Zugriffe auf: MüAl00837928 in 103 ms
1000000 Zugriffe auf: HoDa00414300 in 103 ms
1000000 Zugriffe auf: JuGu00310325 in 105 ms
1000000 Zugriffe auf: AlBr00045195 in 104 ms
1000000 Zugriffe auf: NeUw00268129 in 103 ms
1000000 Zugriffe auf: TrFr00286598 in 104 ms
1000000 Zugriffe auf: HöJü00041436 in 106 ms
1000000 Zugriffe auf: HeJü00963087 in 108 ms
1000000 Zugriffe auf: WeCh00740139 in 104 ms
1000000 Zugriffe auf: AlGa00720899 in 104 ms
1000000 Zugriffe auf: BrUw00572080 in 104 ms
1000000 Zugriffe auf: WeRa00608848 in 102 ms
1000000 Zugriffe auf: AdMi00350918 in 107 ms
1000000 Zugriffe auf: TrKa00955107 in 104 ms
```

Abbildung 20.5 ... auf der anderen Seite bedeutet ein kleiner Load-Faktor zwar konstantere Zugriffszeiten, aber auf Kosten des Speichers

Sie sehen, dass im Gegensatz zum vorherigen Beispiel die Zugriffszeiten nahezu konstant sind. Natürlich kann es bei einer sehr unglücklichen Verteilung immer noch vorkommen, dass es den einen oder anderen Ausreißer gibt. Aber die Anzahl der Ausreißer ist deutlich gesunken.

Betrachten Sie anschließend den Speicherbedarf im Task-Manager, dann sehen Sie sofort, auf wessen Kosten die konstantere Zugriffsgeschwindigkeit erkauft wurde (Abbildung 20.5). Der Speicherbedarf der Beispielapplikation hat sich nahezu verfünffacht!

Es gibt allerdings noch weiteres Beachtenswerts, das Sie im Hinterkopf behalten sollten, wenn Sie mit `Hashtable`-Objekten programmieren. Das Beachten dieser Dinge wird Ihnen jetzt, da Sie wissen, wie Hashtables prinzipiell funktionieren, abermals leichter fallen.

Verwenden eigener Klassen als Schlüssel (Key)

In allen bisherigen Beispielen zur Hashtable wurde eine Zeichenkette als Schlüssel eingesetzt. Der Index auf die Tabelle hat sich dabei aus dem Hashcode der als Schlüssel verwendeten Zeichenkette ergeben.

In unserem konkreten Beispiel ist das Ermitteln des Hashcodes des Strings eigentlich ein zu komplizierter und damit zu lange dauernder Algorithmus. Da wir eine eindeutige »Kundennummer« im Matchcode versteckt haben, können wir rein theoretisch auch diese Nummer als Hashcode verwenden, und die Zugriffsgeschwindigkeit auf die Elemente sollte damit – unabhängig vom Load-Faktor – gleich bleibend sein, denn die Kundennummer ist immer eindeutig (im Gegensatz zum Hashcode des Matchcodes, bei dem leicht Dubletten und damit Kollisionen beim Einsortieren in die Hashtable auftreten können).

Wenn Sie Objekte eigener Klassen als Schlüssel verwenden wollen, müssen Sie die beiden folgenden Punkte beherzigen:

- Die Klasse muss ihre `Equals`-Funktionen überschreiben, damit die `Hashtable`-Klasse die Gleichheit zweier Schlüssel prüfen kann
- Die Klasse muss die `GetHashcode`-Funktion überschreiben, damit die `Hashtable`-Klasse überhaupt einen *Hashcode* ermitteln kann

WICHTIG Wenn Sie die `Equals`-Funktion überschreiben, achten Sie darauf, die richtige Überladungsversion dieser Funktion zu überschreiben. Da die Basisfunktion (die `Equals`-Funktion von `Object`) sowohl als nicht statische als auch als statische Funktion implementiert ist, müssen Sie das `Overloads`-Schlüsselwort anwenden. Da Sie die Funktion überschreiben wollen, müssen Sie `Overrides` ebenfalls verwenden. `Overrides` alleine lässt der Compiler nicht zu, da die Funktion schon in die Basisklasse überladen ist. Allerdings – und das ist das Gefährliche – lässt er ein einzelnes `Overloads` zu, und das führt zu einer Überschattung der Basisfunktion, ***ohne*** sie zu überschreiben. Das Ergebnis: Der Compiler zeigt keine Fehlermeldung (nicht einmal eine Warnung, obwohl er das sollte!), doch Ihre Funktion wird nicht polymorph behandelt und damit nie aufgerufen.

Mit diesem Wissen können wir nun unsere eigene `Key`-Klasse in Angriff nehmen. Bei unserem stark vereinfachten Beispiel bietet sich ein Teil der Matchcode-Zeichenkette an, direkt als *Hashcode* zu fungieren. Da die laufende Nummer einer Adresse Teil des Matchcodes ist, können wir genau diese als *Hashcode* verwenden. Der Code für eine eigene `Key`-Klasse könnte sich folgendermaßen gestalten:

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 20\HashtableDemo02
```

```
Public Class AdressenKey

    Private myMatchcode As String
    Private myKeyValue As Integer
```

```
    Sub New(ByVal Matchcode As String)
        myKeyValue = Integer.Parse(Matchcode.Substring(4))
        myMatchcode = Matchcode
    End Sub

    'Wird benötigt, um bei Kollisionen den richtigen
    'Schlüssel zu finden.
    Public Overloads Overrides Function Equals(ByVal obj As Object) As Boolean
        'If Not (TypeOf obj Is AdressenKey) Then
  '    Dim up As New InvalidCastException("AdressenKey kann nur mit Objekten gleichen Typs verglichen werden")
        '   Throw up
        'End If
        Return myKeyValue.Equals(DirectCast(obj, AdressenKey).KeyValue)
    End Function

    'Wird benötigt, um den Index zu "berechnen".
    Public Overrides Function GetHashcode() As Integer
        Return myKeyValue
    End Function
    Public Overrides Function ToString() As String
        Return myKeyValue.ToString
    End Function

    Public Property KeyValue() As Integer
        Get
            Return myKeyValue
        End Get
        Set(ByVal Value As Integer)
            myKeyValue = Value
        End Set
    End Property

End Class
```

Am ursprünglichen Testprogramm selbst sind nur einige kleinere Änderungen notwendig, um die neue `Key`-Klasse zu implementieren. Die `Adresse`-Klasse musste dabei überhaupt nicht verändert werden. Lediglich am Hauptprogramm müssen einige Zeilen geändert werden, um die Änderung wirksam werden zu lassen.

```
Module HashtableDemo

    Sub Main()

        Dim locAnzahlAdressen As Integer = 1000000
        Dim locZugriffsElemente As Integer = 50
        Dim locMessungen As Integer = 3
        Dim locZugriffe As Integer = 1000000
        Dim locVorlageAdressen As ArrayList
        Dim locAdressen As New Hashtable(100000, 0.1)
        Dim locTestKeys(locZugriffsElemente) As AdressenKey
        Dim locRandom As New Random(Now.Millisecond)


        'Warten auf Startschuss
        Console.WriteLine("Drücken Sie Return, um zu beginnen")
        Console.ReadLine()
```

```
        'Viele Adressen erzeugen:
        Console.Write("Erzeugen von {0} zufälligen Adresseneinträgen...", locAnzahlAdressen)
        Dim locZeitmesser = Stopwatch.StartNew
        locVorlageAdressen = adresse.ZufallsAdressen(locAnzahlAdressen)
        locZeitmesser.Stop()
        Console.WriteLine("fertig nach {0} ms", locZeitmesser.ElapsedMilliseconds)
        locZeitmesser.Reset()

        'Aufbauen der Hashtable
        Console.Write("Aufbauen der Hashtable mit zufälligen Adresseneinträgen...", locAnzahlAdressen)
        locZeitmesser.Start()
        For Each adresse As Adresse In locVorlageAdressen
            'Änderung: Nicht den String, sondern ein Key-Objekt verwenden
            locAdressen.Add(New AdressenKey(adresse.Matchcode), adresse)
        Next
        locZeitmesser.Stop()
        Console.WriteLine("fertig nach {0} ms", locZeitmesser.ElapsedMilliseconds)
        locZeitmesser.Reset()

        '51 zufällige Adressen 'rauspicken.
        'Anderung: Die Key werden abgespeichert, nicht der Matchcode
        For i As Integer = 0 To locZugriffsElemente
            locTestKeys(i) = New AdressenKey(
                DirectCast(locVorlageAdressen(locRandom.Next(locAnzahlAdressen)), Adresse).Matchcode)
        Next

        'Änderung: Kein Object mehr, sondern direkt ein AdressenKey
        Dim locTemp As AdressenKey
        Dim locTemp2, locTemp3 As Object

        'Zugreifen und Messen, wie lange das dauert
        'Das ganze 5 Mal, um die Messung zu bestätigen
        For z As Integer = 1 To locMessungen
            Console.WriteLine()
            Console.WriteLine("{0}. Messung:", z)
            For i As Integer = 0 To locZugriffsElemente
                Console.Write("{0} Zugriffe auf: {1} in ", locZugriffe, locTestKeys(i))
                locTemp = locTestKeys(i)
                locZeitmesser.Start()
                For j As Integer = 1 To locZugriffe
                    locTemp2 = locAdressen(locTemp)
                Next j
                locZeitmesser.Stop()
                'Besser ist besser: Nicht, dass der Compiler merkt,
                'dass locTemp2 gar nicht benötigt wird, und es wegoptimiert!
                locTemp3 = locTemp2.GetType
                Console.WriteLine("{0} ms", locZeitmesser.ElapsedMilliseconds)
                locZeitmesser.Reset()
            Next

            'Zugriff auf Arraylist für Vergleich
.
.
.
    End Sub
```

Die entscheidende Zeile im Beispielcode hat übrigens gar keine Änderung erfahren müssen. `locTemp` dient nach wie vor als Objektvariable für den Schlüssel, nur ist sie nicht mehr vom Typ `String` definiert, sondern als `AdressenKey`. Die Variable `locTemp3` (und ehemals `locTemp`) dient übrigens nur dazu, sicherzustellen, dass der Compiler die innere Schleife nicht wegoptimiert[6] und damit Messergebnisse verfälscht.

Wenn Sie dieses Programm starten, werden Sie zwei Dinge feststellen. Der Zugriff auf die Daten ist spürbar schneller geworden, und: Der Zugriff auf die Daten erfolgt unabhängig vom Load-Faktor immer gleich schnell. Da der Schlüssel auf die Daten nun eindeutig ist, braucht sich die Hashtable nicht mehr um Kollisionen zu kümmern – es gibt nämlich keine mehr. Folglich bringt die Reservierung zusätzlicher Elemente auch keinen nennenswerten Vorteil mehr. Ganz im Gegenteil: Sie würden Speicher verschwenden, der gar nicht benötigt würde.

In diesem Beispiel sind die zu verwaltenden Datensätze nicht sonderlich groß gewesen. Wenn Sie überlegen, wie viele Zugriffe auf die Objekte der Hashtable tatsächlich notwendig gewesen sind, um überhaupt in einen messbaren Bereich zu gelangen, dann wird deutlich, dass sich der betriebene Aufwand für dieses Beispiel eigentlich nicht gelohnt hat. Dennoch: Denken Sie immer daran, dass Computer, auf denen Ihre Software später läuft, in der Regel nicht so leistungsfähig sind wie die Computer, auf denen sie entwickelt wird.

Schlüssel müssen unveränderlich sein!

Wenn Sie sich dazu entschlossen haben, eigene Klassen für die Verwaltung von Schlüsseln in einer Hashtable zu entwickeln, sollten Sie zusätzlich zum Gesagten Folgendes unbedingt beherzigen: Schlüssel müssen unveränderlich sein. Achten Sie daher darauf, dass Sie den Inhalt eines `Key`-Objekts nicht von außen verändern, solange er einer Hashtable zugeordnet ist. Ansonsten würden Sie riskieren, dass die `GetHashcode`-Funktion unter Umständen einen falschen Hashcode für ein `Key`-Objekt zurückliefert. Der Nachschlagealgorithmus der Hashtable hätte dann keine Chance mehr, das Objekt in der Datentabelle wieder zu finden.

Enumerieren von Datenelementen in einer Hashtable

Die Enumeration einer Hashtable (das Iterieren durch sie mit `For Each`) ist prinzipiell möglich. Allerdings müssen sie zwei Dinge dabei beachten:

- Datenelemente werden in einer Hashtable nicht in sequenzieller Reihenfolge gespeichert. Da der Hashcode eines zu speichernden Objekts ausschlaggebend für die Position des Objekts innerhalb der Datentabelle ist, kann die wirkliche Position eines Objekts nur bei ganz einfachen Hashcode-Algorithmen vorausgesagt werden. Wenn Sie – und das wird wahrscheinlich am häufigsten der Fall sein – ein `String`-Objekt als Schlüssel verwenden, wirken die zu speichernden Objekte schon mehr oder weniger zufällig in der Tabelle verteilt.
- Objekte werden innerhalb einer Hashtable in so genannten `Bucket`-Strukturen gespeichert. Ein Schlüssel gehört untrennbar zu seinem eigentlichen Objekt, und beide werden in einem `Bucket`-Element in der Tabelle abgelegt. Eine Iteration durch die Datentabelle kann aus diesem Grund nur mit einem speziellen Objekt erfolgen – nämlich vom Typ `DictionaryEntry` (etwa: *Wörterbucheintrag*).

[6] Dieser Handgriff dient nur als Vorsichtsmaßnahme. Ich gebe zu, nicht wirklich überprüft zu haben, ob die Zeile wegoptimiert werden würde. Bei der Intelligenz moderner Compiler (oder JITter) ist das aber durchaus denkbar.

Eine Iteration durch eine Hashtable könnte für unser Beispiel folgendermaßen aussehen:

```
'Iterieren durch die Hashtable
For Each locDE As DictionaryEntry In locAdressen
    'in unserem Beispiel für den Key
    Dim locAdressenKey As AdressenKey = DirectCast(locDE.Key, AdressenKey)
    'in unserem Beispiel für das Objekt
    Dim locAdresse As Adresse = DirectCast(locDE.Value, Adresse)
Next
```

TIPP Schnelle, typsichere Auflistungen können Sie beispielsweise mithilfe der `KeyedCollection` implementieren, die Ihnen .NET Framework ab der Version 2.0 als Basisklasse zur Verfügung stellt. Ein Beispiel für den Einsatz der `KeyedCollection` finden Sie in Abschnitt »KeyedCollection – Schlüssel/Wörterbuch-Auflistung mit zusätzlichen Index-Abrufen« ab Seite 660.

DictionaryBase

Die Hashtable-Auflistung ermöglicht es, wie `ArrayList` selbst in ihrer Grundausstattung lediglich typunsichere Auflistungen von Objekten zu verwalten. Bis zu .NET Framework 2.0 war es lediglich mithilfe der Klasse `DictionaryBase` möglich, durch Vererbung eine typsichere Hashtable-Auflistung zu implementieren. Die Vorgehensweise dabei entspricht im Prinzip der gleichen, wie Sie sie für `CollectionBase` im Abschnitt »Typsichere Auflistungen auf Basis von CollectionBase« ab Seite 631 beschrieben finden – nur eben mit einer Hashtable-Auflistung als `InnerList` und eben nicht mit einer `ArrayList`-Auflistung.

Seit .NET Framework 2.0 gibt es dank Generics (siehe Kapitel 18) eine sehr viel bequemere Möglichkeit, typsichere Wörterbuch/Hashtable-Implementierungen zu realisieren. Buchstäblicher Schlüssel zum Glück ist hier einerseits die generische `Dictionary`-Auflistung (`Dictionary Of` in Visual Basic), andererseits die so genannte `KeyedCollection`. Diese kann übrigens nur durch Vererbung konkret implementiert werden, und dadurch kümmert sie sich selbstständig darum, dass es eine homogene Schlüssel/Element-Auflistung gibt, indem sie dafür sorgt, den Schlüssel aus dem Element selbstständig zu ermitteln, das der Auflistung hinzugefügt wird. Ein Beispiel zu `KeyedCollection` finden Sie im Abschnitt »List(Of)-Auflistungen und Lambda-Ausdrücke« ab Seite 655; bitte beachten Sie dabei unbedingt die Implementierungs-»Unzulänglichkeiten«, auf die dort auch im Zusammenhang mit `KeyedCollection`-Serialisierungen eingegangen wird.

Queue – Warteschlangen im FIFO-Prinzip

»First in, first out« (als Erstes rein, als Erstes raus) – nach diesem Muster arbeitet die `Queue`-Klasse von .NET Framework. Angewendet haben Sie dieses Prinzip selbst schon sicherlich einige Male in der Praxis – und zwar immer dann, wenn Sie unter Windows mehrere Dokumente hintereinander gedruckt haben. Das Drucken unter Windows funktioniert gemäß dem Warteschlangenprinzip. Das Dokument, das als Erstes in die Warteschlange eingereiht (*enqueue* – einreihen) wurde, wird als Erstes verarbeitet und anschließend wieder aus ihr entfernt (*dequeue* – ausreihen). Aus diesem Grund verwenden Sie die Methoden `Enqueue`, um Elemente der Queue hinzuzufügen, und `Dequeue`, um sie zurückzubekommen und gleichzeitig aus der Warteschlange zu entfernen.

BEGLEITDATEIEN Falls Sie mit der `Queue`-Klasse experimentieren möchten, verwenden Sie dazu am besten nochmals das *CollectionsDemo*-Projekt, das Sie unter

```
...\VB 2010 Entwicklerbuch\Kapitel 20\CollectionsDemo
```

finden. Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei). Verändern Sie das Programm so, dass es die `Sub QueueDemo` aufruft, um das folgende Beispiel nachzuvollziehen:

```
Sub QueueDemo()

    Dim locQueue As New Queue

    'Die typsichere Version mit Generics würde lauten:

    Dim typsafeQueue As New Queue(Of String)

    Dim locString As String

    locQueue.Enqueue("Erstes Element")
    locQueue.Enqueue("Zweites Element")
    locQueue.Enqueue("Drittes Element")
    locQueue.Enqueue("Viertes Element")

    'Nachschauen, was am Anfang steht, ohne es zu entfernen.
    Console.WriteLine("Element am Queue-Anfang:" + locQueue.Peek().ToString)
    Console.WriteLine()

    'Iterieren funktioniert auch.
    For Each locString In locQueue
        Console.WriteLine(locString)
    Next
    Console.WriteLine()

    'Alle Elemente aus Queue entfernen und Ergebnis im Konsolenfenster anzeigen.
    Do
        locString = CStr(locQueue.Dequeue)
        Console.WriteLine(locString)
    Loop Until locQueue.Count = 0
    Console.ReadLine()
End Sub
```

Wenn Sie dieses Programm ablaufen lassen, produziert es folgende Ausgabe im Konsolenfenster:

```
Element am Queue-Anfang:Erstes Element

Erstes Element
Zweites Element
Drittes Element
Viertes Element

Erstes Element
Zweites Element
Drittes Element
Viertes Element
```

HINWEIS Die typsichere Version dieser Auflistung nennt sich `Queue(Of Type)`. Sie bietet, dadurch dass sie typsicher ist und damit für robusteren Code sorgt, für Wertetypen deutliche Geschwindigkeitsvorteile, da sie intern auf Wertetypen-Arrays basiert, deren Elemente nicht geboxt werden müssen.

Stack – Stapelverarbeitung im LIFO-Prinzip

Die `Stack`-Klasse arbeitet nach dem Prinzip »Last in, first out« (»als Letztes rein, als Erstes raus«), also genau umgekehrt zum FIFO–Prinzip der `Queue`-Klasse. Mit der `Push`-Methode legen Sie ein Element auf den Stapel, mit `Pull` ziehen Sie es wieder herunter und erhalten es damit zurück. Das Element, das Sie zuletzt auf den Stapel gelegt haben, wird also mit `Pull` auch als Erstes wieder entfernt.

BEGLEITDATEIEN Sie finden die folgende Demo im gleichen Projekt wie die vorherige. Verändern Sie das Programm so, dass es die `Sub StackDemo` aufruft, um das folgende Beispiel nachzuvollziehen:

```
Sub StackDemo()
    Dim locStack As New Stack

    'Die typsichere Version mit Generics würde lauten:
    Dim typeSafeStack As New Stack(Of String)

    Dim locString As String

    locStack.Push("Erstes Element")
    locStack.Push("Zweites Element")
    locStack.Push("Drittes Element")
    locStack.Push("Viertes Element")

    'Nachschauen, was oben auf dem Stapel liegt, ohne das Element zu entfernen.
    Console.WriteLine("Element zu oberst auf dem Stapel: " + locStack.Peek.ToString)
    Console.WriteLine()

    'Iterieren funktioniert auch.
    For Each locString In locStack
        Console.WriteLine(locString)
    Next
    Console.WriteLine()

    'Alle Elemente vom Stack ziehen und Ergebnis im Konsolenfenster anzeigen.
    Do
        locString = CStr(locStack.Pop)
        Console.WriteLine(locString)
    Loop Until locStack.Count = 0
    Console.ReadLine()
End Sub
```

Wenn Sie dieses Programm ablaufen lassen, produziert es folgende Ausgabe im Konsolenfenster:

```
Element zu oberst auf dem Stapel: Viertes Element

Viertes Element
Drittes Element
Zweites Element
Erstes Element
```

```
Viertes Element
Drittes Element
Zweites Element
Erstes Element
```

HINWEIS Die typsichere Version dieser Auflistung nennt sich `Stack(Of Type)`. Sie bietet, dadurch dass sie typsicher ist und damit für robusteren Code sorgt, für Wertetypen deutliche Geschwindigkeitsvorteile, da sie intern auf Wertetypen-Arrays basiert, deren Elemente nicht geboxt werden müssen.

SortedList – Elemente ständig sortiert halten

Wenn Sie Elemente schon direkt nach dem Einfügen in der richtigen Reihenfolge in einer Auflistung halten wollen, dann ist die `SortedList`-Klasse das richtige Werkzeug für diesen Zweck. Allerdings sollten Sie beachten: Von allen bislang vorgestellten Auflistungsklassen ist die `SortedList`-Klasse diejenige, die die meisten Ressourcen verschlingt. Für zeitkritische Applikationen sollten Sie überlegen, ob Sie Ihre Daten auch anders organisieren und stattdessen lieber auf eine unsortierte Hashtable oder gar auf `ArrayList` zurückgreifen können.

Der Vorteil der `SortedList` ist, dass sie quasi aus einer Mischung von `ArrayList`- und `Hashtable`-Funktionen besteht (obwohl sie algorithmisch gesehen überhaupt nichts mit `Hashtable` zu tun hat). Sie können auf der einen Seite über einen Schlüssel, auf der anderen Seite aber auch über einen Index auf die Elemente von `SortedList` zugreifen. Das folgende erste Beispiel zeigt den generellen Umgang mit `SortedList`.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 20\SortedListDemo
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei). Das Beispiel besteht aus drei Codedateien. In der Datei *Daten.vb* finden Sie die schon bekannte `Adresse`-Klasse (bekannt, falls Sie die vorherigen Abschnitte ebenfalls durchgearbeitet haben) – allerdings in leicht veränderter Form. Der Matchcode der Zufallsadressen beginnt in dieser Version mit einer laufenden Nummer und endet mit der Buchstabenkombination des Nach- und Vornamens. Damit wird vermieden, dass eine Sortierung des Matchcodes grob auch die Adressen nach Namen und Vornamen sortiert und etwaige Nachweise eines bestimmten Programmverhaltens nicht geführt werden können.

```
Module SortedListDemo
    Sub Main()
        Dim locZufallsAdressen As ArrayList = Adresse.ZufallsAdressen(6)
        Dim locAdressen As New SortedList

        Console.WriteLine("Ursprungsanordnung:")
        For Each locAdresse As Adresse In locZufallsAdressen
            Console.WriteLine(locAdresse)
            locAdressen.Add(locAdresse.Matchcode, locAdresse)
        Next
```

```
        'Zugriff per Index:
        Console.WriteLine()
        Console.WriteLine("Zugriff per Index:")
        For i As Integer = 0 To locAdressen.Count - 1
            Console.WriteLine(locAdressen.GetByIndex(i).ToString)
        Next

        Console.WriteLine()
        Console.WriteLine("Zugriff per Enumerator:")
        'Zugriff per Enumerator
        For Each locDE As DictionaryEntry In locAdressen
            Console.WriteLine(locDE.Value.ToString)
        Next
        Console.ReadLine()
    End Sub
End Module
```

Wenn Sie dieses Programm starten, generiert es in etwa die folgenden Ausgaben im Konsolenfenster (die Adressen werden zufällig generiert, deswegen kann die Darstellung in Ihrem Konsolenfenster natürlich wieder von der hier gezeigten abweichen):

```
Ursprungsanordnung:
00000005PlKa: Plenge, Katrin, 26201 Liebenburg
00000004PlKa: Plenge, Katrin, 93436 Liebenburg
00000003AlMa: Albrecht, Margarete, 65716 Bad Waldliesborn
00000002HoBa: Hollmann, Barbara, 96807 Liebenburg
00000001LöLo: Löffelmann, Lothar, 21237 Lippetal
00000000AdKa: Ademmer, Katrin, 49440 Unterschleißheim

Zugriff per Index:
00000000AdKa: Ademmer, Katrin, 49440 Unterschleißheim
00000001LöLo: Löffelmann, Lothar, 21237 Lippetal
00000002HoBa: Hollmann, Barbara, 96807 Liebenburg
00000003AlMa: Albrecht, Margarete, 65716 Bad Waldliesborn
00000004PlKa: Plenge, Katrin, 93436 Liebenburg
00000005PlKa: Plenge, Katrin, 26201 Liebenburg

Zugriff per Enumerator:
00000000AdKa: Ademmer, Katrin, 49440 Unterschleißheim
00000001LöLo: Löffelmann, Lothar, 21237 Lippetal
00000002HoBa: Hollmann, Barbara, 96807 Liebenburg
00000003AlMa: Albrecht, Margarete, 65716 Bad Waldliesborn
00000004PlKa: Plenge, Katrin, 93436 Liebenburg
00000005PlKa: Plenge, Katrin, 26201 Liebenburg
```

Sie erkennen, dass die Liste in der Tat nach dem Schlüssel umsortiert wurde. Dies gilt sowohl für den Zugriff über den Index als auch über den Enumerator mit `For Each`.

HINWEIS Seit .NET Framework 2.0 gibt es die generische Auflistungsklasse `SortedList(Of Key, Type)`, die Sie als typsichere Alternative in Erwägung ziehen sollten. Verwechseln Sie sie bitte nicht mit der generischen Klasse `SortedSet(Of Type)`, die einen sortierten Satz von Elementen bereithält, in dem keine Dubletten vorkommen dürfen.

Generische Auflistungen (Generic Collections)

Generische Auflistungen haben im Vergleich zu »herkömmlichen« Auflistungen, die Sie in den vorherigen Abschnitten kennen gelernt haben, einen entscheidenden Vorteil: Sie sind grundsätzlich typsicher. Anders als »normale« Auflistungen wie beispielsweise die `ArrayList`-Klasse, nehmen sie, da sie nicht auf `Object` basieren, nicht jeden Datentyp als Element entgegen, sondern beschränken sich auf den Datentyp, der ihrer Definition zugrunde liegt.

Das bedeutet: Definieren Sie beispielsweise eine `Collection` auf Basis von `Integer`, etwa mit

```
Dim locGenColl As New Collection(Of Integer)
```

dann laufen Sie nicht Gefahr, später versehentlich der Auflistung ein Element hinzuzufügen, das nicht vom Typ `Integer` ist – oder mit anderen Worten: Die folgende Zeile würde bereits zur Entwurfszeit im Editor als fehlerhaft gekennzeichnet werden:

```
locGenColl.Add("Ein Element")
```

Ihre Programme werden somit durch den Einsatz von generischen Auflistungen robuster, und auch die Entwicklungszeit reduziert sich, da Sie sich nicht mit Laufzeitfehlern herumärgern müssen, sondern bereits zur Entwurfszeit Fehler korrigieren können. Und weniger Programmtests bedeuten weniger Entwicklungszeit und -kosten.

WICHTIG Für viele generische Auflistungen gilt auch, dass sie Wertetypen – also primitive Datentypen wie `Integer`, `Date`, `Double` oder `Decimal` und auch Wertetypen wie `Point` oder `Size` – viel schneller verarbeiten können als ihre nicht generischen Varianten. Das liegt daran, dass generische Auflistungen intern meistens auf typisierte Arrays zurückgreifen können, während ihre nicht generischen Auflistungen die Träger der Daten auf `Object`-Arrays basieren lassen. Die Folge dabei ist, dass die Elemente, wie in Kapitel 13 beschrieben, aufwändig geboxt werden müssen, was zusätzlichen Speicher kostet und lange dauert.

Nun gibt es nicht wenige generische Auflistungen seit .NET Framework 2.0, und sie alle im Detail zu beschreiben, ist nicht nur unnötig – schließlich funktionieren viele von ihnen wie ihre nicht generischen Verwandten – es würde auch den Rahmen des Buchs sprengen. Aus diesem Grund möchte ich mich auf die in meinen Augen wichtigen Besonderheiten beschränken, die Sie bei nicht generischen Auflistungen antreffen.

Tabelle 20.1 gibt darüber hinaus Auskunft, welche wichtigen generischen Auflistungen Sie für welche Zwecke verwenden können. Die Auflistungen, mit denen Sie es am häufigsten zu tun haben werden, finden Sie anschließend detaillierter und mit entsprechenden Beispielen beschrieben.

Namespace	Auflistung	Beschreibung
System.Collections.ObjectModel	Collection(Of Type)	Stellt eine Standardauflistung für die einfache, unsortierte Verwaltung von Elementen eines bestimmten Typs zur Verfügung. **Besonderheiten:** Im Gegensatz zu List(Of Type) gibt es überschreibbare Methoden, mit denen man das Verhalten beim Einfügen, Löschen und Neuzuweisen von Elementen der Auflistung in abgeleiteten Klassen beeinflussen kann. Intern basiert die Collection(Of Type) ebenfalls auf einer List(Of Type) – sie ist also nur unwesentlich langsamer als die List(Of Type), eignet sich aber insbesondere in Szenarien, bei denen man das Hinzufügen oder Entfernen von Elementen in der Auflistung »mitbekommen« muss, um zum Beispiel Ereignisse der hinzugefügten Elemente zu binden oder wieder zu lösen oder eine Parent-Eigenschaft zu pflegen, damit ein Objekt weiß, welcher Auflistung es hinzugefügt wurde, es also auch einen Verweis »nach oben« gibt.
System.Collections.ObjectModel	KeyedCollection(Of Key, Type)	Stellt eine Auflistung von Elementen zur Verfügung, die sowohl das Nachschlagen über einen Schlüssel (Key) eines bestimmten Typs als auch den Einsatz eines Indexes erlaubt. **Besonderheiten:** Diese Auflistung können Sie nur in Ableitungen verwenden, da der Typ, den Sie speichern, selbst einen Standardschlüssel erzeugen muss, und für diesen Umstand müssen Sie in der Ableitung sorgen. **WICHTIG:** Vermeiden Sie den Einsatz von Integer-Schlüsseln, da es hier beim Serialisieren der Auflistung zu Problemen kommt.
System.Collections.ObjectModel	ReadOnlyCollection(Of Type)	Stellt eine Auflistung dar, deren Elemente nur gelesen werden können. **Besonderheiten:** Elemente, die in einer anderen generischen Auflistung gleichen Typs vorliegen, lassen sich dieser Auflistung nur bei der Instanziierung im Konstruktor übergeben. Ansonsten sind die Elemente nur lesbar und können nach der Instanziierung nicht mehr verändert werden.
System.Collections.Generic	Dictionary(Of Key, Type)	Stellt eine Auflistung von Schlüsseln und Werten dar. **Besonderheiten:** Stellt eine Zuordnung von einem Satz von Schlüsseln zu einem Satz von Werten bereit. Jede Hinzufügung zum Dictionary besteht aus einem Wert und dem zugeordneten Schlüssel. Ein Wert kann anhand des zugehörigen Schlüssels sehr schnell abgerufen werden, da die Dictionary-Klasse in Form einer Hashtable implementiert ist. Mehr zum Konzept von Hashtables finden Sie im Abschnitt »Hashtables – für das Nachschlagen von Objekten« ab Seite 634.
System.Collections.Generic	LinkedList(Of Type)	Stellt eine doppelt verknüpfte Liste dar. **Besonderheiten:** Es handelt sich um eine verknüpfte Liste mit einzelnen Knoten vom Typ LinkedListNode; das Einfügen und Entfernen einzelner Elemente geht extrem schnell vonstatten. ▶

Namespace	Auflistung	Beschreibung
System.Collections.Generic	List(Of Type)	Stellt eine Standardauflistung für die einfache, unsortierte Verwaltung von Elementen eines bestimmten Typs zur Verfügung. **Hinweis:** Diese Klasse eignet sich nicht für Ableitungen in eigenen Auflistungsklassen, bei denen Sie mit Code Einfluss auf die Bearbeitung der Liste nehmen müssen. Verwenden Sie stattdessen die Collection(Of)-Auflistung (siehe oben).
System.Collections.Generic	Queue(Of Type)	Stellt eine FIFO-Auflistung *(First in, first out)* von Objekten dar. **Hinweis:** Prinzipiell funktioniert diese Auflistung wie ihr nicht generischer Verwandter. Mehr zur nicht generischen Version dieser Auflistung finden Sie im Abschnitt »Queue – Warteschlangen im FIFO-Prinzip« auf Seite 648.
System.Collections.Generic	SortedDictionary(Of Key, Type)	Stellt eine Auflistung von Schlüssel-Wert-Paaren dar, deren Reihenfolge anhand der Schlüsselsortierung bestimmt wird
System.Collections.Generic	SortedList(Of Key, Type)	Verwaltet eine sortierte Liste, deren Elemente über Schlüssel abrufbar sind. **Hinweis:** Prinzipiell funktioniert diese Auflistung wie ihr nicht generischer Verwandter. Mehr zur nicht generischen Version dieser Auflistung finden Sie im Abschnitt »SortedList – Elemente ständig sortiert halten« auf Seite 651. Im Gegensatz zu SortedDictionary erfolgt die Sortierung über das Element und nicht über den Schlüssel!
System.Collections.Generic	Stack(Of Type)	Stellt eine LIFO-Auflistung (*Last in, first out*) von Objekten dar. **Hinweis:** Prinzipiell funktioniert diese Auflistung wie ihr nicht generischer Verwandter. Mehr zur nicht generischen Version dieser Auflistung finden Sie im Abschnitt »Stack – Stapelverarbeitung im LIFO-Prinzip« auf Seite 650.
System.Collections.Generic	SortedSet(Of Type)	Verwaltet einen sortierten Satz von Elementen, in dem keine Dubletten vorkommen dürfen. **HINWEIS:** Das Hinzufügen eines bereits vorhandenen Elements löst KEINE Ausnahme aus; das Element wird einfach nur nicht eingefügt.

Tabelle 20.1 Die wichtigsten generischen Auflistungstypen

List(Of)-Auflistungen und Lambda-Ausdrücke

Die List(Of)-Klasse ist eine der am häufigsten (wenn nicht sogar die am häufigsten) eingesetzte generische Auflistungsklasse. Wenn Sie primitive Datentypen in einer Auflistung speichern, sollten Sie den Einsatz von List(Of Type) anderen Auflistungsklassen vorziehen – alleine schon aus Performancegründen. Denn .NET Framework ist in der Lage, den eigentlich notwendigen Boxing-Vorgang bei List(Of Type) zu umgehen, und damit ist List(Of Type) die schnellere Alternative.

Und obwohl es sich bei dieser generischen Auflistung um solch eine populäre Klasse handelt, hält sie Funktionalitäten bereit, über die man – obwohl sie bereits in .NET Framework 2.0 vorhanden waren – jedenfalls im Rahmen von Visual Basic 2005 bis zu .NET Framework 2.0 wenig Brauchbares erfahren konnte; wohl auch, weil sie förmlich nach der Verwendung von Lambdas schreien, die erst mit der 2008er-Version Einzug in Visual Basic hielten.

Das folgende Beispiel setzt das Verständnis von Lambda-Funktionen übrigens voraus – und falls Sie sich mit diesen noch nicht auseinander gesetzt haben sollten: Kapitel 16 klärt deren grundsätzliches Verständnis. Sie werden dabei auch sehen, dass sich mehrzeilige Anweisungs-Lambdas besonders für die Zusammenarbeit mit vielen Methoden der `List(Of Type)`-Klasse eignen!

Abbildung 20.6 Sortierung und Suche steuern Sie über die Spaltenköpfe in dieser Adressenliste

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 20\ListOfDemo
```

Wenn Sie das Programm starten, sehen Sie einen Dialog, etwa wie in Abbildung 20.6 zu sehen. Die im Formular vorhandene `ListView` wird mit 50 Zufallsadressen gefüllt. Sie können die Liste sortieren, indem Sie auf den entsprechenden Spaltenkopf klicken – so, wie Sie es vom Windows-Explorer in der *Details*-Ansicht gewohnt sind.

Eine Suchfunktion, die Sie über die entsprechende Schaltfläche erreichen, gestattet es Ihnen, nach dem Begriff innerhalb der Liste zu suchen. Gefunden wird dabei der Eintrag, dessen Text in der Spalte, nach der zuletzt sortiert wurde, dem eingegebenen Suchbegriff entspricht.

Soweit ist dieses Beispiel noch nichts Besonderes. Doch wenn Sie einen Blick in die Umsetzung riskieren, wird klar, wie alternativ der Lösungsansatz ist. Das geht schon beim Schreiben der Zufallsadressen in die Liste los:

ForEach und der generische Action-Delegat

Wenn Sie den Inhalt eines Arrays oder einer Auflistung in einer Liste darstellen möchten, liegt die Vorgehensweise, um das zu erreichen, eigentlich auf der Hand: Sie iterieren mit einem `For Each`-Konstrukt durch die Auflistung, verarbeiten damit jedes Element und bringen es mit geeigneten Methoden oder Eigenschaften in die sichtbare Liste eines Formulars. Dieses Beispiel nutzt einen anderen Weg, wie im folgenden Listingausschnitt zu sehen:

```
'Für jedes Element der Liste wird der folgende Lambda aufgerufen.
myAdressen.ForEach(Sub(element)
                       Dim locLvwItem As New ListViewItem(element.Name)

                       'Die Untereinträge setzen
                       With locLvwItem.SubItems
                           .Add(element.Vorname)
                           .Add(element.PLZ)
                           .Add(element.Ort)
                       End With

                       'Zum Wiederfinden Referenz in Tag
                       locLvwItem.Tag = element

                       'Zur Listview hinzufügen
                       lvwAdressen.Items.Add(locLvwItem)

                   End Sub)
```

Sie können die `ForEach`-Methode verwenden, um durch eine Auflistung iterieren *zu lassen* und dabei für jedes Element einen Delegaten aufrufen, der eine bestimmte Aktion ausführt. Genau das Verfahren nutzen wir an dieser Stelle, um die Liste aufzubauen. Dank der Fähigkeit von Visual Basic, seit der Version 2010 auch mehrzeilige Lambdas zu verarbeiten, brauchen wir hier nicht mehr, wie in früheren Versionen, einen Delegaten anzugeben, sondern können direkt ein Lambda verwenden, das den Code zur Verfügung stellt, der von `ForEach` für jedes Element angewendet werden soll.

Aber natürlich könnten an dieser Stelle auch »manuelle« Delegatvariablen zum Einsatz kommen, die in Abhängigkeit von bestimmten Programmzuständen andere Prozeduren verwenden, und genau darin besteht der Reiz des Einsatzes dieser Verfahren.

Wichtig ist in jedem Fall, dass die Methoden den Signaturanspruch des `Action`-Delegaten erfüllen, den `ForEach` verlangt: Es muss sich hier um eine Methode handeln, die kein Funktionsergebnis hat (also um eine Visual Basic-`Sub`) und als Parameter ein Element entgegennimmt, dessen Typ auch der Typbasis der generischen Liste entspricht – `Adresse` in unserem Beispiel.

Im Ergebnis erreichen wir also, dass für jedes Element des `List(Of Adresse)`-Objekts `myAdressen` die entsprechende Lambda-Funktion (fett im obenstehenden Listing) aufgerufen wird.

Sort und der generische Comparison-Delegat

Prinzipiell funktioniert das nächste Pärchen ähnlich, das Sie verwenden, wenn Sie ein Array mit der Methode `Sort` ohne den Einsatz einer speziellen `Comparer`-Klasse (wie im Abschnitt »Implementierung von Sort und BinarySearch für eigene Klassen« ab Seite 613 gezeigt) sortieren möchten. In Abwandlung zur ersten Verwendung kommen hier jedoch Lambda-Ausdrücke zum Einsatz, sodass wir uns das buchstäbliche Delegieren an die richtige Sortierfunktion mit einer Delegatenvariable sparen können. Den relevanten Codeausschnitt des Beispiels finden Sie im Folgenden, und Sie sehen dort auch, wie Sie im Kontext sowohl mit einzeiligen Ausdrucks-Lambdas als auch mit mehrzeiligen Anweisungs-Lambdas arbeiten können:

```
'Wird aufgerufen, wenn eine der Spalten angeklickt wird.
Private Sub lvwAdressen_ColumnClick(ByVal sender As Object,
               ByVal e As System.Windows.Forms.ColumnClickEventArgs) Handles lvwAdressen.ColumnClick

    'Spaltennummer, die in e.Column steht, in AdressenSortierenNach konvertieren
    mySortierenNach = CType(e.Column, AdressenSortierenNach)

    Select Case mySortierenNach
        Case AdressenSortierenNach.Name
            myAdressen.Sort(Function(adr1, adr2) String.Compare(adr1.Name, adr2.Name))

        Case AdressenSortierenNach.Vorname
            myAdressen.Sort(Function(adr1, adr2) String.Compare(adr1.Vorname, adr2.Vorname))

        Case AdressenSortierenNach.PLZ
            myAdressen.Sort(Function(adr1, adr2) String.Compare(adr1.PLZ, adr2.PLZ))

        Case AdressenSortierenNach.Ort
            'Mehrzeilige ginge natürlich auch ...
            myAdressen.Sort(Function(adr1, adr2)
                                '... dann aber Return nicht vergessen!
                                Return String.Compare(adr1.Ort, adr2.Ort)
                            End Function)

    End Select

    'Die Elemente neu sortiert darstellen
    ElementeDarstellen()
End Sub
```

`Sort` arbeitet hier in diesem Beispiel mit Lambda-Ausdrücken, bei denen jeder der vier Lambda-Ausdrücke nach einer anderen Eigenschaft der Adresse sortiert – in Abhängigkeit davon, welche der vier Spalten in der `ListView` zum Sortieren eingestellt wurde.

Find und der generische Predicate-Delegat

Das letzte generische »Dreamteam« schließlich kommt zum Einsatz, wenn es um das Finden eines bestimmten Objekts in einer generischen Liste geht: `Find` und der Delegat, der durch die generische `Predicate`-Klasse eingerichtet wird. Auch hier entspricht die grundsätzliche Vorgehensweise dem bereits Bekannten, wie der entsprechende Code im Beispiel zeigt:

```
Private Sub btnSuchen Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
                              Handles btnSuchen.Click

    'Suchbegriff abfragen
    Dim suchFormular As New frmSuchen
    Dim aktuellerSuchbegriff As String    ' Der zuletzt erfragte Suchbegriff

    'Den Suchbegriff merken, damit der Predicate-Delegat darauf
    'zugreifen kann.
    aktuellerSuchbegriff = suchFormular.Suchbegriff
    If String.IsNullOrEmpty(aktuellerSuchbegriff) Then
        Return
    End If

    'Hier kann die Suche beginnen!
    Dim locGefAdr As Adresse = Nothing

    Select Case mySortierenNach
        Case AdressenSortierenNach.Name
            locGefAdr = myAdressen.Find(Function(adr) adr.Name = aktuellerSuchbegriff)
        Case AdressenSortierenNach.Vorname
            locGefAdr = myAdressen.Find(Function(adr) adr.Vorname = aktuellerSuchbegriff)
        Case AdressenSortierenNach.PLZ
            locGefAdr = myAdressen.Find(Function(adr) adr.PLZ = aktuellerSuchbegriff)
        Case AdressenSortierenNach.Ort
            'Mehrzeilig ginge natürlich auch hier wieder ...
            locGefAdr = myAdressen.Find(Function(adr)
                                            '...dann aber - wieder - 'Return' nicht vergessen!
                                            Return adr.Ort = aktuellerSuchbegriff
                                        End Function)
    End Select

    'Wenn ein Element gefunden wurde, dieses markieren.
    If locGefAdr IsNot Nothing Then

        'Alle ListView-Elemente durchsuchen und überprüfen, ob ...
        For Each locLvwItem As ListViewItem In Me.lvwAdressen.Items

            '... die Tag-Referenz der Referenz des gesuchten Objekts entspricht.
            If locLvwItem.Tag Is locGefAdr Then

                'Gefunden! ListView-Element markieren,
                locLvwItem.Selected = True

                'und dafür sorgen, dass es im sichtbaren Bereich liegt.
                locLvwItem.EnsureVisible()
                Return
            End If
        Next
    End If
End Sub
```

Abbildung 20.7 Der Suchbegriff bezieht sich immer auf die Spalte, nach der zuletzt sortiert wurde

Auch hier machen Lambda-Ausdrücke, wie in den fett hervorgehobenen Codezeilen zu sehen, die Suche nach den richtigen Adressen vergleichsweise einfach. Je nach selektierter Sortierspalte wird einfach ein anderer Lambda-Ausdruck verwendet, der den Vergleich mit dem eingegebenen Suchbegriff übernimmt.
Nur eine kleine Herausforderung ist dann noch das Selektieren des gefundenen Begriffs in der Liste – und das ist im Übrigen auch der Grund, wieso wir beim Aufbauen der Liste eine Referenz jedes Elements in der `Tag`-Eigenschaft jedes `ListViewItem`-Elements speichern: Wenn der Begriff durch `Find` gefunden wurde, liegt uns das entsprechende `Adresse`-Objekt vor. Mit diesem können wir anschließend durch die `ListViewItem`-Auflistung iterieren und auf Objektübereinstimmung durch Abfrage der `Tag`-Eigenschaft testen. Dieser Aufwand ist nötig, da es keine andere Möglichkeit gibt, das richtige `ListViewItem`-Element zu finden, und nur dieses erlaubt es aber, die richtige Zeile in der Liste durch seine `Select`-Eigenschaft auszuwählen.

KeyedCollection – Schlüssel/Wörterbuch-Auflistung mit zusätzlichen Index-Abrufen

Ich gebe zu, ich bin ein großer Fan von `KeyedCollection` – sie ist in meinen eigenen Projekten neben der `Collection(Of Type)` am häufigsten anzutreffen. Warum? Sie hat zwei entscheidende Vorteile:

- Sie erlaubt es, einen beliebigen Typ (na ja – *fast* beliebigen Typ, doch dazu gleich mehr) als Schlüssel anzugeben. Damit können Sie sie als Wörterbuchauflistung verwenden und ihre Elemente über einen Schlüssel abfragen.
- Sie können sie zusätzlich wie eine ganz normale `Collection` behandeln, also mit `For Each` durch ihre Elemente iterieren oder auch einzelne Elemente über einen numerischen Index abrufen.

Da `KeyedCollection` auch einen Schlüsselwert zum Abrufen jedes ihrer Elemente braucht, benötigt sie einen Mechanismus, der aus dem Element, das es hinzuzufügen gilt, einen Schlüssel erzeugt. Aus diesem Grund ist die `KeyedCollection`-Klasse auch eine abstrakte Basisklasse: Sie müssen die `KeyedCollection` vererben und ihre Methode `GetKeyForItem` überschreiben, in der Sie dann regeln, wie aus dem Element, das es hinzuzufügen gilt, ein eindeutiger Schlüssel kreiert werden soll. Die `KeyedCollection`-Klasse sollte deswegen nur Elemente verwalten, die über wenigstens eine Eigenschaft verfügen, mit der sich ein Element eindeutig von einem anderen unterscheiden lässt. Das kann beispielsweise eine Personalnummer, eine Kundennummer, ein eindeutiger Matchcode oder Ähnliches sein.

WICHTIG `KeyedCollection` hat aber auch einen entscheidenden Nachteil, oder besser: einen Designfehler. Wenn Sie ihren Inhalt serialisieren – also beispielsweise mit einem bestimmten in .NET Framework eingebauten Mechanismus als XML-Datei speichern –, können Sie als Schlüssel *keinen* `Integer`-Datentyp verwenden, da .NET Framework beim Serialisierungsversuch mit einer Ausnahme »aussteigt«. Kapitel 22 geht näher auf dieses Problem ein und hält auch einen Workaround zu dieser Designunzulänglichkeit bereit.

BEGLEITDATEIEN Sie finden das Projekt für das folgende Beispiel im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 20\KeyedCollectionDemo
```

Im Projekt finden Sie zwei Codedateien, *Daten.vb* und *KeyedCollection.vb*. Erstere enthält eine Adressenklasse, die Sie aus diesem Kapitel schon kennen. Aufgabe dieser Klasse ist es, die Daten einer Adresse zu speichern. Sie stellt darüber hinaus auch eine statische Funktion bereit, die eine bestimmbare Anzahl von Zufallsadressen generiert und in einer `ArrayList` zurückliefert.

Damit die Klasse `Adresse` in diesem Beispiel als Verwaltungselement für die `KeyedCollection` infrage kommt, muss sie eine Anforderung erfüllen: Sie muss über eine Eigenschaft verfügen, mit der sich ein `Adresse`-Element von jedem anderen `Adresse`-Element unterscheiden lässt.

HINWEIS Die Funktionalität zur Prüfung auf Eindeutigkeit kann die `Adresse`-Klasse natürlich nicht selbst implementieren, da sie die anderen Elemente einer Auflistung nicht kennt; deswegen ist die Bestimmung einer solchen Dateneigenschaft an dieser Stelle nur eine theoretische Definition. Es ist Aufgabe des späteren Hauptprogramms, sicherzustellen, dass es keine Schlüsseldubletten in einer Elementauflistung gibt, bevor ein neues Element (wie `Adresse`) der Auflistung hinzugefügt wird.

`KeyedCollection` ist eine *abstrakte* Basisklasse, was bedeutet, dass Sie sie nicht direkt verwenden können. Sie müssen sie vererben und um eine bestimmte Funktionalität erweitern. Deswegen definieren Sie – wie Sie es von abstrakten Klassen gelernt haben – zunächst einen neuen Klassennamen, etwa mit

```
Public Class Adressen
```

und fügen darunter die Zeile

```
    Inherits KeyedCollection(Of String, Adresse)
```

ein. Mit dieser Zeile leiten Sie von `KeyedCollection` ab und bestimmten gleichzeitig, dass `String` der Typ für den Schlüssel und `Adresse` der Typ der zu verwaltenden Elemente sein soll.

In dem Moment, in dem Sie die Zeile mit ↵ abschließen, fügt der Visual Basic-Editor automatisch den Rumpf der Methode ein, den Sie in der Basisklasse überschreiben müssen:

```
    Protected Overrides Function GetKeyForItem(ByVal item As Adresse) As String

    End Function
```

Mit dieser Funktion »holt« sich `KeyedCollection` den Wert, der vom Schlüsseldatentyp sein muss, immer dann, wenn Sie ein neues Element der Auflistung (beispielsweise durch `Add`) hinzufügen oder ein Element verändern (zum Beispiel mit `Item(Index)=New Adresse(…)`). So wird sichergestellt, dass jedes Element über einen eindeutigen Schlüssel verfügt.

In unserem Beispiel gibt es eine Eigenschaft in der Klasse `Adresse`, die ein Element vom Typ `Adresse` eindeutig kennzeichnet: `Matchcode`. Diese Eigenschaft ist vom Typ `String` – also ist der Schlüsseltyp für die `KeyedCollection` ebenfalls String. Die komplett implementierte `KeyedCollection` sieht also folgendermaßen aus:

```
''' <summary>
''' Verwaltet Objekte vom Typ Adresse als Wörterbuch-Auflistung.
''' Abgeleitet von der abstrakten Basisklasse KeyedCollection.
''' </summary>
''' <remarks></remarks>
Public Class Adressen
```

```
    Inherits KeyedCollection(Of String, Adresse)

    'Diese Methode muss überschrieben werden, um zu garantieren,
    'dass für jedes Element ein Schlüssel existiert.
    Protected Overrides Function GetKeyForItem(ByVal item As Adresse) As String
        'Hier wird bestimmt, dass der eindeutige Key der Matchcode ist.
        Return item.Matchcode
    End Function
End Class
```

Diese Implementierung ist auch schon alles, was Sie brauchen, um anschließend mit der `KeyedCollection` arbeiten zu können. Das Hauptmodul *KeyedCollection.vb* demonstriert den Umgang und stellt insbesondere heraus, dass Elemente einer auf `KeyedCollection` basierenden Klasse sowohl über einen Index als auch über den entsprechenden Schlüssel abgerufen und bearbeitet werden können:

```
Imports System.Collections.ObjectModel

Module KeyedCollection

    Sub Main()
        'Ersteinmal eine normale Arraylist definieren,
        'die aus 100 Zufallsadressen besteht.
        Dim locArrayList As ArrayList = Adresse.ZufallsAdressen(100)

        'Das ist die "selbst gestrickte" KeyedCollection
        Dim locKeyedAdressen As New Adressen

        'Mit den Elementen der Arraylist füllen wir die KeyedCollection
        For Each locAdressItem As Adresse In locArrayList
            locKeyedAdressen.Add(locAdressItem)
        Next

        'Abrufen der Elemente aus der KeyedCollection:
        'Variation Nr.1 - abrufen über den Index
        For c As Integer = 0 To 10
            Console.WriteLine(locKeyedAdressen(c).ToString)
        Next

        'Leerzeile - der besseren Übersicht wegen:
        Console.WriteLine()

        'Irgendeine herauspicken, um an den Matchcode zu kommen:
        Dim locMatchcode As String = locKeyedAdressen(10).Matchcode

        'Variation Nr. 2: Eine Adresse kann auch über den Key
        '(in diesem Fall über den Matchcode) abgerufen werden.
        Dim locAdresse As Adresse = locKeyedAdressen(locMatchcode)
        Console.WriteLine("Die Adresse mit dem Matchcode " & locMatchcode & " lautet:")
        Console.WriteLine(locAdresse.ToString)
        Console.WriteLine()

        Console.WriteLine("Taste drücken zum Beenden!")
        Console.ReadKey()
    End Sub

End Module
```

Elementverkettungen mit LinkedList(Of)

Erwähnenswert bei den generischen Auflistungen ist die `LinkedList`-Klasse, die das erste Mal mit .NET Framework 2.0 die Möglichkeit zur Verwaltung von Elementen in Form von verketteten Listen einführt.

`LinkedList` stellt eine verknüpfte Liste mit einzelnen Knoten vom Typ `LinkedListNode` dar. Durch das Konzept von `LinkedList`, bei dem die zu speichernden Elemente jeweils selbst auf das nächste und das vorherige Elemente »zeigen«, kann das Einfügen bzw. Löschen von Elementen sehr schnell vonstatten gehen, da – sinnbildlich – die Kette an einer Stelle nur aufgeknüpft werden muss, um ein neues Glied in die Kette einzufügen.

Durch die Art der Elementverwaltung bietet `LinkedList` Methoden und Eigenschaften, die Sie in anderen Auflistungen nicht finden. Die folgende Tabelle gibt Aufschluss über die besonderen Methoden und Eigenschaften der `LinkedList`-Klasse.

Elemente werden listenintern nicht wie bei anderen Auflistungsklassen direkt gespeichert, sondern in einem Knotenobjekt namens `LinkedListNode` abgelegt, das für die Verkettung zum nächsten bzw. vorherigen Knoten der Liste sorgt. Auch diese Klasse ist generisch ausgelegt. Knoten, die der Liste hinzugefügt werden, müssen dabei auf Basis desselben Typs definiert werden wie die Liste selbst.

Methode	Beschreibung
`AddAfter`	Fügt nach einem vorhandenen Knoten in der `LinkedList` einen neuen Knoten oder Wert hinzu
`AddBefore`	Fügt vor einem vorhandenen Knoten in der `LinkedList` einen neuen Knoten oder Wert hinzu
`AddFirst`	Fügt am Anfang der `LinkedList` einen neuen Knoten oder Wert hinzu
`AddLast`	Fügt am Ende der `LinkedList` einen neuen Knoten oder Wert hinzu
`Find`	Sucht den ersten Knoten, der den angegebenen Wert enthält
`FindLast`	Sucht den letzten Knoten, der den angegebenen Wert enthält
`First`	Ruft den ersten Knoten der `LinkedList` ab. Diese Eigenschaft kann nur gelesen werden.
`Last`	Ruft den letzten Knoten der `LinkedList` ab. Diese Eigenschaft kann nur gelesen werden.
`Remove`	Entfernt das erste Vorkommen eines Knotens oder Werts aus der `LinkedList`
`RemoveFirst`	Entfernt den Knoten am Anfang der `LinkedList`
`RemoveLast`	Entfernt den Knoten am Ende der `LinkedList`

Tabelle 20.2 Die besonderen Methoden und Eigenschaften der `LinkedList`-Klasse

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 20\LinkedList
```

```
Module LinkedList

    Sub Main()
        Dim locLinkedList As New LinkedList(Of Adresse)
        Dim locAdressen As ArrayList = Adresse.ZufallsAdressen(50)
        Dim locAdresse As Adresse = Nothing

        'Acht Adressen an das jeweilige Ende der LinkedList anfügen
        For c As Integer = 0 To 49

            'den 25. merken wir uns für die spätere Suche in der Liste
            If c = 25 Then
                locAdresse = DirectCast(locAdressen(c), Adresse)
            End If
            locLinkedList.AddLast(DirectCast(locAdressen(c), Adresse))
        Next

        Dim locLinkedListNode As LinkedListNode(Of Adresse)
        'Den Knoten finden, der dem 25. Adresseneintrag entsprach.
        locLinkedListNode = locLinkedList.Find(locAdresse)
        Console.WriteLine("Vor " & locLinkedListNode.Value.ToString & " kommt " &
                          locLinkedListNode.Previous.Value.ToString &
                          " und danach kommt " & locLinkedListNode.Next.Value.ToString)
        Console.WriteLine()

        'Eine neue Adresse vor dem 25. einfügen:
        Console.WriteLine("Sarah Halek vorher einfügen!")
        locLinkedList.AddBefore(locLinkedListNode,
                New Adresse("SasiMatch", "Halek", "Sarah", "99999", "Musterhausen"))

        Console.WriteLine()

        'Das Gleiche nochmal ausgeben, es sollte die Änderungen jetzt widerspiegeln.
        Console.WriteLine("Vor " & locLinkedListNode.Value.ToString & " kommt " &
                          locLinkedListNode.Previous.Value.ToString &
                          " und danach kommt " & locLinkedListNode.Next.Value.ToString)
        Console.WriteLine()
        Console.WriteLine("Taste drücken zum Beenden")
        Console.ReadKey()
    End Sub
End Module
```

Wenn Sie dieses Programm ausführen, gibt es in etwa Folgendes auf dem Bildschirm aus:

```
Vor "Hörstmann, Hans" kommt "Englisch, Margarete" und danach kommt "Löffelmann, Katrin"

Sarah Halek vorher einfügen!

Vor "Hörstmann, Hans" kommt "Halek, Sarah" und danach kommt "Löffelmann, Katrin"


Taste drücken zum Beenden
```

Kapitel 21

Serialisierung

In diesem Kapitel:

Einführung in Serialisierungstechniken 666
Flaches und tiefes Klonen von Objekten 674
Serialisieren von Objekten mit Zirkelverweisen 680
XML-Serialisierung 683

Wenn Sie Anwendungen entwickeln, kommen Sie irgendwann zwangsläufig an den Punkt, an dem Sie die Objekte, mit deren Hilfe Sie die Daten Ihrer Anwendung verwalten und speichern, für den späteren Gebrauch sichern oder zur Weiterverarbeitung an eine andere Instanz übertragen müssen. Vom aktuellen »Zustand« eines Objekts muss in diesem Fall eine Art Momentaufnahme gemacht werden; der Speicherinhalt aller Eigenschaften und aller untergeordneten Objekte muss dabei gesichert werden. Dabei spielt es natürlich erst einmal keine Rolle, auf welche Weise die Daten gesichert werden: Sie können byteweise – so, wie sie im Arbeitsspeicher stehen – direkt dort ausgelesen und in eine Datei geschrieben werden; denkbar wäre auch, dass sie zuvor durch entsprechende Algorithmen komprimiert und erst dann in einer Datei gespeichert werden. Die Daten eines Objekts, wie Zahlen oder Datumswerte, könnten auch zuvor in ein vom Benutzer lesbares Format umgewandelt und speziell formatiert werden, sodass ein Transfervorgang sie auch beispielsweise im *XML*- oder *SOAP*-Format direkt über das Internet zu einem anderen Server transportieren könnte – beispielsweise mithilfe eines Webdienstes.

Ganz gleich, wie die Daten aus einem Objekt »geholt« und anschließend an eine andere Stelle verfrachtet werden – die Prozedur, die diese Aufgabe erfüllt, kann nur nach einem bestimmten Schema vorgehen: Sie muss *der Reihe nach* alle Eigenschaften und Variablen des Objekts abfragen, sie aufbereiten und anschließend mit den aufbereiteten Daten irgendetwas anstellen. Bei diesem Vorgang spricht man vom Serialisieren von Objekten.

Auch der umgekehrte Weg ist notwendig: Wenn Sie beispielsweise eine Reihe von Adressobjekten in eine Textdatei serialisiert haben, damit die Daten nach Beenden des Adressprogramms und Ausschalten des Computers erhalten bleiben, muss der umgekehrte Prozess stattfinden, sobald der Benutzer den Computer einschaltet, die Adressverwaltung startet und mit den zuvor erfassten Adressen weiterarbeiten möchte. In diesem Fall müssen die Objekte wieder den gleichen Zustand annehmen, den sie vor dem Serialisieren hatten. Sie müssen jetzt aus der Textdatei *deserialisiert* werden.

Nun wäre .NET Framework nicht .NET Framework, wenn es Sie bei diesen Vorgängen, die natürlich in jeder Anwendung vorkommen, nicht unterstützen würde. Sie brauchen also nicht selbst Hand anzulegen und jede einzelne Eigenschaft eines Objekts auszulesen und in eine Textdatei zu schreiben. Umgekehrt müssen Sie Adressobjekte auch nicht selbst in Ihrer Anwendung komplett neu instanziieren, während Sie sie aus einer Textdatei wieder einlesen. Für diesen Zweck hält .NET Framework einige geniale Werkzeuge bereit, die Sie in den folgenden Abschnitten kennen lernen werden. Es empfiehlt sich, diese genauer anzuschauen, da sie Ihnen die Arbeit erheblich erleichtern können – nicht nur, wenn Sie die Adressen Ihrer Adressverwaltung auf der Festplatte Ihres Computers speichern wollen.

HINWEIS Wenn Sie Instanzen von Klassen serialisieren und an anderer Stelle wieder deserialisieren, benötigen Sie in jedem Fall die Klassendefinition in beiden Entitäten. Sie können mit den hier vorgestellten Techniken also nicht auch den kompletten Klassencode selbst übertragen, sondern nur die Zustände innerhalb eines Objekts.

Einführung in Serialisierungstechniken

Bevor Sie sich anschauen, was .NET Framework an Serialisierungstechniken zu bieten hat, lassen Sie uns zunächst einen Blick auf ein ganz simples Beispiel werfen. Dieses Beispielprogramm macht nichts weiter, als den Benutzer eine Adresse eingeben zu lassen und diese durch Klick auf eine Schaltfläche in eine Datei zu serialisieren (sprich: zu sichern).

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 21\Serialization01
```

Abbildung 21.1 Die Nur-Lesen-Eigenschaften werden beim Serialisieren nicht übernommen

Das Beispiel nutzt dabei zunächst noch keine Serialisierungstechniken von .NET Framework, sondern liest die einzelnen Eigenschaften sozusagen manuell aus und speichert sie als Text in einer Datei. Beim Klicken auf die Schaltfläche *Serialisieren* generiert die Prozedur, die das `Click`-Ereignis behandelt, ein `Adresse`-Objekt aus den Feldinhalten des Dialogs. Dieses Objekt übergibt die Routine einer weiteren Prozedur, die eine Datei zum Schreiben öffnet, die Eigenschaften nacheinander ausliest und sie als String in die Datei schreibt:

```
    Private Sub btnSerialisieren_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
        Handles btnSerialisieren.Click
        Dim locAdresse As New Adresse(txtVorname.Text, _
                                      txtNachname.Text, _
                                      txtStraße.Text, _
                                      txtPLZOrt.Text)
        'Einen "unmöglichen" Dateinamen verwenden, damit, wie es der Zufall will,
        'nicht eine andere wichtige Datei gleichen Namens überschrieben wird.
        Adresse.SerializeToFile(locAdresse, "C:\serializedemo_f4e3w21.txt")

        'Info über den Datensatz anzeigen.
        txtErstelltAm.Text = locAdresse.ErfasstAm.ToString("dd.MM.yyyy HH:mm:ss")
        txtErstelltVon.Text = locAdresse.ErfasstVon
    End Sub
```

Die `Adresse`-Klasse hält die notwendigen Elemente zum Speichern der Adressdaten und die statische Prozedur zum Serialisieren der Objektdaten in eine Datei bereit:

```
Imports System.IO

Public Class Adresse

    Private myErfasstAm As DateTime
    Private myErfasstVon As String

    Sub New(ByVal Vorname As String, ByVal Nachname As String, ByVal Straße As String, ByVal PLZOrt As
String)
        'Konstruktor legt alle Member-Daten an
        Me.Nachname = Nachname
```

```
        Me.Vorname = Vorname
        Me.Straße = Straße
        Me.PLZOrt = PLZOrt
        myErfasstAm = DateTime.Now
        myErfasstVon = My.User.Name
    End Sub

    'Alle öffentlichen Felder in die Datei schreiben
    Public Shared Sub SerializeToFile(ByVal adr As Adresse, ByVal Filename As String)
        Dim locStreamWriter As New StreamWriter(Filename, False, System.Text.Encoding.Default)
        With locStreamWriter
            .WriteLine(adr.Vorname)
            .WriteLine(adr.Nachname)
            .WriteLine(adr.Straße)
            .WriteLine(adr.PLZOrt)
            .Flush()
            .Close()
        End With
    End Sub

    'Aus der Datei lesen und daraus ein neues Adressobjekt erstellen
    Public Shared Function SerializeFromFile(ByVal Filename As String) As Adresse
        Dim locStreamReader As New StreamReader(Filename, System.Text.Encoding.Default)
        Dim locAdresse As Adresse
        With locStreamReader
            locAdresse = New Adresse(.ReadLine, .ReadLine, .ReadLine, .ReadLine)
        End With
        locStreamReader.Close()
        Return locAdresse
    End Function

    Public Property Nachname() As String
    Public Property Vorname() As String
    Public Property Straße() As String
    Public Property PLZOrt() As String

    'Die beiden folgenden Eigenschaften haben "Nur-Lesen-Status", da auch
    'der Entwickler das Anlegen-Datum nicht manipulieren darf!
    Public ReadOnly Property ErfasstAm() As DateTime
        Get
            Return myErfasstAm
        End Get
    End Property

    Public ReadOnly Property ErfasstVon() As String
        Get
            Return myErfasstVon
        End Get
    End Property
End Class
```

Sie sehen anhand dieses Codelistings, dass es zwei Eigenschaften gibt, die zwar gelesen, aber nicht geschrieben werden dürfen: Diese Eigenschaften, die Auskunft darüber geben, wer das `Adresse`-Objekt zu welchem Zeitpunkt angelegt hat, dürfen ausschließlich bei der Objekterstellung definiert werden.

Beim Serialisieren auf die herkömmliche Art und Weise, wie hier im Beispiel zu sehen, ergibt sich hier schon ein Problem: Wenn Sie das Programm nämlich beenden, anschließend erneut starten und die `Deserialisieren`-Schaltfläche betätigen, dann wird zwar aus der Datei ein neues `Adresse`-Objekt erzeugt. Dieses Objekt entspricht dem ursprünglichen aber nicht in allen Details, denn: Das Erstellungsdatum und der »Ersteller« des Objekts selbst hätten zwar noch mitgesichert werden können; der Deserialisierungsalgorithmus hat aber aufgrund der Beschaffenheit dieser zusätzlichen Eigenschaften keine Möglichkeit, die Originalzustände dieser Eigenschaft wieder einzulesen. Er könnte diese Zusatzinformationen dem Objekt schlichtweg nicht zuordnen.

Diese Tatsache spiegelt sich im Programm wider: Wann immer Sie das ursprünglich gespeicherte Objekt durch Klick auf die entsprechende Schaltfläche deserialisieren, steht in den unteren Infofeldern nicht das ursprüngliche Erstellungsdatum, sondern das Erstellungsdatum des Objekts zum Zeitpunkt des Deserialisierens.

Serialisieren mit SoapFormatter und BinaryFormatter

Anders sieht es aus, wenn Sie das Serialisieren und das Deserialisieren mit bestimmten Hilfsmitteln aus .NET Framework durchführen. Sie brauchen sich in diesem Fall nämlich nicht um das Auslesen der Eigenschaften der Objekte selbst zu kümmern – .NET Framework macht das automatisch für Sie. Und noch mehr: Einige Klassen von .NET Framework, die für das Serialisieren von Objektdaten zuständig sind, erlauben auch das Serialisieren und Deserialisieren von privaten Eigenschaften einer Klasse.

WICHTIG Für alle Objekte, die serialisiert werden sollen, gilt: Sie müssen mit einem besonderen Attribut namens `Serializable` gekennzeichnet werden. Wenn diese Voraussetzungen erfüllt sind, können Sie zwei der `Serializer`-Klassen für die Serialisierung und Deserialisierung einer so gekennzeichneten Objektinstanz verwenden, die auch private Feldvariablen einer Klasse berücksichtigen.

- **`SoapFormatter`-Klasse** Stellt Serialisierungs- und Deserialisierungsfunktionen zur Verfügung, die das *SOAP*-Format verwenden. Die Dateninhalte eines Objekts werden dabei in reinen Text umgewandelt, der auf der einen Seite auch für den Benutzer verständlich mit einem Texteditor gelesen und angezeigt und auf der anderen Seite – da er das *SOAP*-Format einhält – auch problemlos über das Internet transportiert werden kann. Der Nachteil: Diese Art der Datenspeicherung ist nicht sonderlich effizient, eben durch die Konvertierung nativer Daten in lesbaren Text.
- **`BinaryFormatter`-Klasse** Stellt Serialisierungs- und Deserialisierungsfunktionen zur Verfügung, die das Binärformat verwenden. Die Dateninhalte eines Objekts werden dabei so verwendet, wie sie im Arbeitsspeicher vorliegen. Werden die Daten eines Objekts unter Zuhilfenahme dieses *Serializers* beispielsweise in einer Datei gespeichert, erfolgt die Datenspeicherung sehr kompakt. Eine auf diese Weise generierte Datei ist aber für den Benutzer direkt nicht lesbar, da sie die Daten des Objekts im binären Format enthält.

BEGLEITDATEIEN Das folgende Beispiel demonstriert den Einsatz mit diesen beiden Klassen. Sie finden dieses Beispiel unter:

```
...\VB 2010 Entwicklerbuch\Kapitel 21\Serialization02
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Abbildung 21.2 In dieser Version können Sie zwischen SOAP- und Binärserialisierung wählen. Die Nur-Lesen-Eigenschaften bleiben jetzt beim Deserialisieren erhalten.

Wenn Sie dieses Programm starten, wählen Sie für das Serialisieren bzw. Deserialisieren das gewünschte Format aus. Wenn Sie sich für das *SOAP*-Format entscheiden, generiert die `SoapFormatter`-Klasse eine Datei, die der folgenden entspricht (vorausgesetzt natürlich, Sie haben die gleichen Daten eingegeben, wie in Abbildung 21.2 zu sehen):

```
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:clr="http://schemas.microsoft.com/soap/encoding/clr/1.0" SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<a1:Adresse id="ref-1"
xmlns:a1="http://schemas.microsoft.com/clr/nsassem/Serialization02/Serialization02%2C%20Version%3D1.0.0.
0%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3Dnull">
<myName id="ref-3">Thiemann</myName>
<myVorname id="ref-4">Uwe</myVorname>
<myStraße id="ref-5">Autorenstraße 34</myStraße>
<myPLZOrt id="ref-6">99999 Buchhausen</myPLZOrt>
<myErfasstAm>2008-10-15T09:43:45.1295355+01:00</myErfasstAm>
<myErfasstVon id="ref-7">NOTEVISTA\Uwe</myErfasstVon>
</a1:Adresse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Sie werden überrascht sein, wenn Sie sehen, mit wie wenig Aufwand die Serialisierung im Programm realisiert wurde. Wenn Sie einen Blick in den Projektmappen-Explorer werfen, finden Sie dort im Gegensatz zum vorherigen Beispiel zwei weitere Klassendateien. Sie enthalten die Kapselungen von `SoapFormatter` und `BinaryFormatter` zur Serialisierung (und Deserialisierung) beliebiger, serialisierbarer Objekte in Dateien. Die Klasse `SoapSerializer` sieht wie folgt dargestellt aus:

Universeller SOAP-Datei-De-/Serializer

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

Public Class SoapSerializer

    Shared Sub SerializeToFile(ByVal FileInfo As FileInfo, ByVal [Object] As Object)

        Dim locFs As FileStream = New FileStream(FileInfo.FullName, FileMode.Create)
        Dim locSoapFormatter As New SoapFormatter(Nothing, _
                  New StreamingContext(StreamingContextStates.File))
```

```
        locSoapFormatter.Serialize(locFs, [Object])
        locFs.Flush()
        locFs.Close()

    End Sub

    Shared Function DeserializeFromFile(ByVal FileInfo As FileInfo) As Object

        Dim locObject As Object

        Dim locFs As FileStream = New FileStream(FileInfo.FullName, FileMode.Open)
        Dim locSoapFormatter As New SoapFormatter(Nothing,
                  New StreamingContext(StreamingContextStates.File))
        locObject = locSoapFormatter.Deserialize(locFs)
        locFs.Close()
        Return locObject

    End Function
End Class
```

Wenn Sie `SoapFormatter` verwenden wollen, müssen Sie einen Verweis in das entsprechende Projekt einbauen. Dazu klicken Sie im Projektmappen-Explorer den Projektnamen oder den Ordner *Verweise* (wenn das Symbol *Alle Dateien anzeigen aktiviert* ist) mit der rechten Maustaste an und rufen aus dem Kontextmenü den Eintrag *Verweis hinzufügen* auf. Im Dialog, der sich jetzt öffnet, wählen Sie den Eintrag `System.Runtime.Serialization.Formatters.Soap` per Doppelklick aus (etwa wie in Abbildung 21.3 zu sehen).

Verlassen Sie den Dialog mit *OK*. Achten Sie ebenfalls darauf, die Anweisungen

```
Imports System.Runtime.Serialization
Imports System.Runtime.Serialization.Formatters.Soap
```

als erste Zeilen in der Codedatei zu platzieren, in der Sie `SoapFormatter` verwenden möchten.

Abbildung 21.3 Wenn Sie mit der `SoapFormatter`-Klasse Objektdaten im SOAP-Format serialisieren wollen, binden Sie diesen Verweis in Ihr Projekt ein

Universeller Binary-Datei-De-/Serializer

Die Klasse `BinarySerializer` sieht folgendermaßen aus:

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

Public Class BinarySerializer

    Shared Sub SerializeToFile(ByVal FileInfo As FileInfo, ByVal [Object] As Object)

        Dim locFs As FileStream = New FileStream(FileInfo.FullName, FileMode.Create)
        Dim locBinaryFormatter As New BinaryFormatter(Nothing, _
                    New StreamingContext(StreamingContextStates.File))
        locBinaryFormatter.Serialize(locFs, [Object])
        locFs.Flush()
        locFs.Close()

    End Sub

    Shared Function DeserializeFromFile(ByVal FileInfo As FileInfo) As Object

        Dim locObject As Object

        Dim locFs As FileStream = New FileStream(FileInfo.FullName, FileMode.Open)
        Dim locBinaryFormatter As New BinaryFormatter(Nothing, _
                     New StreamingContext(StreamingContextStates.File))
        locObject = locBinaryFormatter.Deserialize(locFs)
        locFs.Close()
        Return locObject

    End Function
End Class
```

Achten Sie darauf, die Anweisungen

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

als erste Zeilen in der Codedatei zu platzieren, in der Sie `BinaryFormatter` verwenden möchten. Einen speziellen Verweis brauchen Sie, anders als bei `SoapFormatter`, nicht in das Projekt einzubinden.

Funktionsweise der Datei-Serializer-Klassen

Beide Klassen funktionieren exakt nach dem gleichen Prinzip, sie verwenden lediglich unterschiedliche Formatter (also Klassen, die Datenaufbereitungslogiken zur Verfügung stellen), um unterschiedliche Formate zu erzeugen bzw. für die Rekreation des Ursprungobjekts.

Sie öffnen beim Serialisieren zunächst einen Dateistream, über den die Daten unter Verwendung des jeweiligen Formatters aus dem Objekt in die Datei gelangen. In der anschließenden Instanziierung des Formatters wird dieser durch die Übergabe einer `StreamingContext`-Instanz darauf vorbereitet, welches Ziel die Objektserialisierung haben wird (in diesem Fall werden die Objektdaten in eine Datei serialisiert). Die eigentliche Serialisierung des Objekts geschieht dann durch die einzige Zeile:

```
locSoapFormatter.Serialize(locFs, [Object])
```

Der Rest der Prozedur ist obligatorisch: Alle Streamdaten werden mit `Flush` aus dem internen Puffer geleert, und die Datei wird daraufhin geschlossen. Das Ergebnis: Die Objektdaten wurden einschließlich ihrer privaten Member in die Datei geschrieben. Der umgekehrte Weg beim Deserialisieren erfolgt äquivalent.

TIPP Wenn Sie diese Art der Serialisierung in Dateien in Ihren eigenen Programmen verwenden wollen, kopieren Sie die Codedateien `BinarySerializer.vb` bzw. `SoapSerializer.vb` einfach in Ihr Projektverzeichnis und fügen sie anschließend Ihrem Projekt hinzu. Sie können sie dann auf genauso einfache Weise verwenden, wie sie im Beispielprojekt vom Adressobjekt verwendet wurden (siehe in Fettschrift gesetzte Zeilen am Ende des Codelistings):

```
<Serializable()> _
Public Class Adresse

    Private myErfasstAm As DateTime
    Private myErfasstVon As String

    Sub New(ByVal Vorname As String, ByVal Nachname As String, ByVal Straße As String,
            ByVal PLZOrt As String)
        'Konstruktor legt alle Member-Daten ein
        Me.Nachname = Nachname
        Me.Vorname = Vorname
        Me.Straße = Straße
        Me.PLZOrt = PLZOrt
        myErfasstAm = DateTime.Now
        myErfasstVon = My.User.Name
    End Sub

    'Alle öffentlichen Felder in die Datei schreiben - Soap-Format
    Public Shared Sub SerializeSoapToFile(ByVal adr As Adresse, ByVal Filename As String)
        SoapSerializer.SerializeToFile(New FileInfo(Filename), adr)
    End Sub

    'Aus der Datei lesen und daraus ein neues Adressobjekt erstellen - Soap-Format
    Public Shared Function SerializeSoapFromFile(ByVal Filename As String) As Adresse
        Return DirectCast(SoapSerializer.DeserializeFromFile(New FileInfo(Filename)), Adresse)
    End Function

    'Alle öffentlichen Felder in die Datei schreiben - Binary-Format
    Public Shared Sub SerializeBinToFile(ByVal adr As Adresse, ByVal Filename As String)
        BinarySerializer.SerializeToFile(New FileInfo(Filename), adr)
    End Sub

    'Aus der Datei lesen und daraus ein neues Adressobjekt erstellen - Binary-Format
    Public Shared Function SerializeBinFromFile(ByVal Filename As String) As Adresse
        Return DirectCast(BinarySerializer.DeserializeFromFile(New FileInfo(Filename)), Adresse)
    End Function
```

WICHTIG Wenn Sie die Objektserialisierung anwenden, achten Sie darauf, dass alle Objekte, die Sie serialisieren wollen, das `Serializable`-Attribut tragen – so wie auch im vorherigen Codelisting zu sehen (siehe erste beiden Zeilen). Wenn die Klasse, die Sie serialisieren möchten, oder eine Objektinstanz, die diese Klasse einbindet, dieses Attribut nicht aufweist, löst .NET Framework beim Serialisierungsversuch durch eines der zu serialisierenden Objekte eine Ausnahme aus, etwa wie in Abbildung 21.4 zu sehen.

Abbildung 21.4 Wenn eine zu serialisierende Klasse oder ein Objekt, das sie einbindet, nicht mit dem Attribut `Serializable` gekennzeichnet ist, löst .NET Framework beim Serialisierungsversuch diese Ausnahme aus

Flaches und tiefes Klonen von Objekten

Der Vorteil beim Serialisieren über Funktionen von .NET Framework ist, dass die Serialisierungsalgorithmen in der Lage sind, automatisch so genannte »tiefe Klons« (vollständige Kopien) eines Objekts zu erstellen.

Dazu folgender Hintergrund: Angenommen, Sie haben ein Objekt, das die Daten Ihrer Applikation speichert. Dieses Objekt verfügt dann beispielsweise über eine Eigenschaft, die eine `ArrayList` zur Verfügung stellt, in der weitere Elemente gespeichert sind. Um eine komplette Kopie dieses Objekts zu erstellen, würde es nicht ausreichen, die Elemente, die diese `ArrayList`-Eigenschaft beinhaltet, zu kopieren, wie das folgende Beispiel zeigt:

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 21\DeepCloning
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
Module mdlMain

    Sub Main()
        Dim locAdrOriginal As New Adresse("Marcus", "Pusch", "Musterstraße 22", "59555 Lippstadt")
        Dim locAdrKopie As Adresse
        locAdrOriginal.BefreundetMit = New List(Of Adresse) From {
                            New Adresse("Ramona", "Leenings", "Autorstr. 33", "49595 Buchhausen"),
                            New Adresse("Cate", "Hora", "Autorstr. 34", "49595 Buchhausen"),
                            New Adresse("Gabriele", "Löffelmann", "Autorstr. 35", "49595 Buchhausen")}

        'Originaladresse ausgeben
        Console.WriteLine("Original:")
        Console.WriteLine("=========")
        Console.WriteLine(locAdrOriginal)

        'Kopie anlegen
        With locAdrOriginal
```

```
            locAdrKopie = New Adresse(.Vorname, .Name, .Straße, .PLZOrt)
            locAdrKopie.Name += " (Kopie)"
            locAdrKopie.BefreundetMit = .BefreundetMit
        End With

        'Kopie ausgeben
        Console.WriteLine("Kopie:")
        Console.WriteLine("=========")
        Console.WriteLine(locAdrKopie)

        'Änderungen im Original:
        CType(locAdrOriginal.BefreundetMit(1), Adresse).Name = "Löffelmann-Beck"
        CType(locAdrOriginal.BefreundetMit(2), Adresse).Name = "Löffelmann-Beck"

        'Kopie nach Änderungen im Original
        Console.WriteLine("Kopie nach Änderung im Original:")
        Console.WriteLine("==============================")
        Console.WriteLine(locAdrKopie)
        Console.ReadLine()

    End Sub

End Module
```

Dieses Beispiel nutzt die leicht abgewandelte `Adresse`-Klasse, die Sie schon aus dem vorherigen Beispiel kennen. Sie verfügt über eine zusätzliche Eigenschaft namens `BefreundetMit`. Diese Eigenschaft entspricht einer generischen Liste (`List(Of Adresse)`) mit der Aufgabe, andere Adressen aufzunehmen, die bestimmen, mit wem dieser Kontakt befreundet ist.

```
<Serializable()>
Public Class Adresse

    Private myErfasstAm As DateTime
    Private myErfasstVon As String

    Sub New(ByVal Vorname As String, ByVal Name As String, ByVal Straße As String,
            ByVal PLZOrt As String)
        'Konstruktor legt alle Member-Daten an
        Me.Name = Name
        Me.Vorname = Vorname
        Me.Straße = Straße
        Me.PLZOrt = PLZOrt
        myErfasstAm = DateTime.Now
        myErfasstVon = Environment.UserName
    End Sub

    Public Property Name() As String
    Public Property Vorname() As String
    Public Property Straße() As String
    Public Property PLZOrt() As String

    Public ReadOnly Property ErfasstAm() As DateTime
        Get
            Return myErfasstAm
        End Get
    End Property

    Public ReadOnly Property ErfasstVon() As String
```

```
        Get
            Return myErfasstVon
        End Get
    End Property

    Public Property BefreundetMit() As List(Of Adresse)

    Public Overrides Function ToString() As String
        Dim locTemp As String
        locTemp = Name + ", " + Vorname + ", " + Straße + ", " + PLZOrt + vbNewLine
        locTemp += "--- Befreundet mit: ---" + vbNewLine
        For Each locAdr As Adresse In BefreundetMit
            locTemp += "   * " + locAdr.ToShortString() + vbNewLine
        Next
        locTemp += vbNewLine
        Return locTemp
    End Function

    Public Function ToShortString() As String
        Return Name + ", " + Vorname
    End Function
End Class
```

Zusätzlich gibt es in dieser Klasse zwei Funktionen – `ToString` und `ToShortString` –, die eine `Adresse`-Instanz in einen String umwandeln, damit die Ausgabe einfacher wird.

Das Programm macht nun Folgendes: Es legt eine Originaladresse an und fügt ihr weitere Adressobjekte hinzu, die in der `ArrayList`, die durch `BefreundetMit` offen gelegt wird, gespeichert werden. Anschließend erzeugt es eine identische Kopie der Originaladresse im Objekt `locAdrKopie`.

Das Problem: An dieser Stelle wird eine so genannte flache Kopie des Objekts erstellt. Nur die Eigenschaften der oberen Ebene werden in die Kopie übernommen. Das wird auch deutlich, wenn Sie das Beispielprogramm starten:

```
Original:
=========
Pusch, Marcus, Musterstraße 22, 59555 Lippstadt
--- Befreundet mit: ---
   * Leenings, Ramona
   * Hora, Cate
   * Löffelmann, Gabriele


Kopie:
=========
Pusch (Kopie), Marcus, Musterstraße 22, 59555 Lippstadt
--- Befreundet mit: ---
   * Leenings, Ramona
   * Hora, Cate
   * Löffelmann, Gabriele


Kopie nach Änderung im Original:
================================
Pusch (Kopie), Marcus, Musterstraße 22, 59555 Lippstadt
--- Befreundet mit: ---
   * Leenings, Ramona
   * Löffelmann-Beck, Cate
   * Löffelmann-Beck, Gabriele
```

Beim Ändern der Elemente der `ArrayList` `BefreundetMit` werden auch die Elemente der Kopie verändert. Und das muss so sein, denn: Die Eigenschaft `BefreundetMit` stellt ja nicht wirklich selbst eine Instanz von `List(Of Adresse)` dar, sondern verweist lediglich auf sie. Tatsächlich gibt es die Elemente in der generischen Liste nur ein einziges Mal. In diesem Beispiel ist also nur eine flache Kopie (»shallow clone« bzw. »shallow copy« auf Englisch) des `Adresse`-Objekts erstellt worden.

Anders wird das, wenn Sie eine tiefe Kopie (»deep clone« bzw. »deep copy« auf Englisch) erzeugen. Hier werden die Elemente der `ArrayList`, um beim Beispiel zu bleiben, wirklich kopiert – es gibt nach Abschluss der Kopieerstellung tatsächlich zwei völlig unabhängige Objekte, mit ebenso unabhängigen Elementen.

Was bei diesem Beispiel mit noch vergleichsweise wenig Aufwand durchführbar wäre, sieht bei wirklich komplexen Objekten schon anders aus. Wollten Sie eine `DeepCopy`-Routine selbst implementieren, würde dies einen enormen Aufwand bedeuten. Obendrein könnten Sie eine solche Routine mit normalen Mitteln nicht universal gestalten; sie würde sich nur auf das aktuelle Objekt beziehen. Bei einer Klassenableitung, die die Basisklasse um weitere Eigenschaften ergänzen würde, müssten Sie die `DeepCopy`-Routine schon wieder modifizieren.

Doch diesen Aufwand müssen Sie auch gar nicht betreiben, denn mit den Serialierungsfunktionen nimmt Ihnen .NET Framework diesen kompletten Aufwand ab. Beim Serialisieren erstellt .NET Framework nämlich eine tiefe Kopie.[1] Und da Serialisierung natürlich nicht notwendigerweise bedeutet, das Objekt in einer Datei zwischenzuspeichern, können Sie mit einem kleinen Trick eine universelle `DeepCopy`-Routine kreieren, die mit jedem serialisierbaren Objekt funktioniert, wie das folgende Beispiel zeigt.

Universelle DeepClone-Methode

Das folgende Beispiel verwendet ebenfalls den `BinarySerializer` für das Serialisieren und das Deserialisieren eines Objekts, doch nutzt es einen anderen Träger im Vergleich zum letzten Beispiel. Ein Objekt, das es zu serialisieren gilt, wird nicht in eine Datei, sondern in ein `MemoryStream`-Objekt geschrieben, das schließlich in ein `Byte`-Array konvertiert wird. Im umgekehrten Fall erzeugt der `BinarySerializer` aus einem `Byte`-Array, das in ein `MemoryStream`-Objekt umgewandelt wird, wieder das ursprüngliche Objekt. Da der `BinarySerializer` grundsätzlich auch verschachtelte Eigenschaften verarbeitet, lässt sich daraus auf einfache Art und Weise eine Klasse erstellen, die von jedem beliebigen Objekt, das selbst als serialisierbar gekennzeichnet ist und auch nur selbst serialisierbare Objekte verwendet, eine tiefe Kopie anfertigen kann.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 21\UniversalDeepCloning
```

[1] Wobei diese Aussage nicht hundertprozentig richtig ist, denn eigentlich erstellt .NET Framework ja keine Kopie des Objekts, sondern liest beim Serialisieren zunächst nur seine Daten aus. Das macht es aber bis in die unterste Ebene, sodass man den kombinierten Vorgang von Serialisieren eines Objekts in einen Datenstream und Deserialisieren dieses Datenstreams in ein neues Objekt durchaus »tiefes Kopieren« des Objekts nennen kann.

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

Public Class ADObjectCloner

    Public Shared Function DeepCopy(ByVal [Object] As Object) As Object
        Return DeserializeFromByteArray(SerializeToByteArray([Object]))
    End Function

    Shared Function SerializeToByteArray(ByVal [Object] As Object) As Byte()

        Dim retByte() As Byte
        Dim locMs As MemoryStream = New MemoryStream
        Dim locBinaryFormatter As New BinaryFormatter(Nothing, _
                    New StreamingContext(StreamingContextStates.Clone))
        locBinaryFormatter.Serialize(locMs, [Object])
        locMs.Flush()
        locMs.Close()
        retByte = locMs.ToArray()
        Return retByte

    End Function

    Shared Function DeserializeFromByteArray(ByVal by As Byte()) As Object

        Dim locObject As Object

        Dim locFs As MemoryStream = New MemoryStream(by)
        Dim locBinaryFormatter As New BinaryFormatter(Nothing, _
                    New StreamingContext(StreamingContextStates.File))
        locObject = locBinaryFormatter.Deserialize(locFs)
        locFs.Close()
        Return locObject

    End Function

End Class
```

Das Hauptprogramm verwendet im folgenden Beispiel nun nicht mehr eigenen Code zum Kopieren des Objekts, sondern benutzt die `DeepCopy`-Funktion der Klasse (die geänderte Passage ist fett hervorgehoben):

```
Module mdlMain

    Sub Main()
        Dim locAdrOriginal As New Adresse("Hans", "Mustermann", "Musterstraße 22", "59555 Lippstadt")
        Dim locAdrKopie As Adresse
        locAdrOriginal.BefreundetMit = New List(Of Adresse) From {
                            New Adresse("Ramona", "Leenings", "Autorstr. 33", "49595 Buchhausen"),
                            New Adresse("Cate", "Hora", "Autorstr. 34", "49595 Buchhausen"),
                            New Adresse("Gabriele", "Löffelmann", "Autorstr. 35", "49595 Buchhausen")}

        'Originaladresse ausgeben
        Console.WriteLine("Original:")
```

```
        Console.WriteLine("=========")
        Console.WriteLine(locAdrOriginal)

        'Kopie anlegen
        locAdrKopie = DirectCast(ADObjectCloner.DeepCopy(locAdrOriginal), Adresse)

        'Kopie ausgeben
        Console.WriteLine("Kopie:")
        Console.WriteLine("=========")
        Console.WriteLine(locAdrKopie)

        'Änderungen im Original:
        DirectCast(locAdrOriginal.BefreundetMit(1), Adresse).Nachname = "Löffelmann-Beck"
        DirectCast(locAdrOriginal.BefreundetMit(2), Adresse).Nachname = "Löffelmann-Beck"

        'Kopie nach Änderungen im Original
        Console.WriteLine("Kopie nach Änderung im Original:")
        Console.WriteLine("================================")
        Console.WriteLine(locAdrKopie)
        Console.ReadLine()

    End Sub

End Module
```

Wenn Sie das Programm anschließend starten, erkennen Sie den Unterschied zum vorherigen Beispiel. Die beiden erzeugten Objekte sind wirklich komplett unabhängig voneinander. Änderungen an der ArrayList des Ausgangsobjekts beeinflussen das Ergebnis der Ausgabe der Objektkopie in keiner Weise:

```
Original:
=========
Mustermann, Hans, Musterstraße 22, 59555 Lippstadt
--- Befreundet mit: ---
   * Leenings, Ramona
   * Hora, Cate
   * Löffelmann, Gabriele


Kopie:
=========
Mustermann, Hans, Musterstraße 22, 59555 Lippstadt
--- Befreundet mit: ---
   * Leenings, Ramona
   * Hora, Cate
   * Löffelmann, Gabriele


Kopie nach Änderung im Original:
================================
Mustermann, Hans, Musterstraße 22, 59555 Lippstadt
--- Befreundet mit: ---
   * Leenings, Ramona
   * Hora, Cate
   * Löffelmann, Gabriele
```

Serialisieren von Objekten mit Zirkelverweisen

So genannte Zirkelverweise bereiten Speicheralgorithmen erfahrungsgemäß die größten Schwierigkeiten. Zirkelverweise kennen Sie vielleicht aus der Tabellenkalkulation. Sie treten dann auf, wenn beispielsweise Zelle A auf Zelle B, diese auf Zelle C und diese wiederum auf Zelle A verweisen soll. Was bei Tabellenkalkulationen grundsätzlich verboten ist, gestattet .NET Framework mit Objektreferenzen dagegen sehr wohl. Und – um bei unserem bisherigen Beispiel zu bleiben – im Szenario des Programms aus dem letzten Abschnitt könnte das sogar recht schnell passieren, denn: Es ist nicht nur denkbar, sondern wahrscheinlich, dass Person A Person B zu ihrem Freundeskreis zählt und umgekehrt.

Welche Auswirkungen Zirkelverweise normalerweise haben, zeigt das folgende Beispiel.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 21\Zirkelverweise
```

In diesem Beispielprogramm sind im Gegensatz zum vorherigen die folgenden Änderungen vorgenommen worden, was die `Adresse`-Klasse betrifft. Die `ToString`-Funktion benutzt selbst nicht mehr die Kurzform der Adressen für die Ausgabe der `BefreundetMit`-Eigenschaft, sondern erzeugt den Ausgabestring ebenfalls mit der `ToString`-Funktion, die allerdings eine kleine Änderung erfahren hat. Welche das ist, sehen Sie, wenn Sie das Programm starten und sich anschließend das Ergebnis anschauen:

```
Erste Adresse:
==============
Hora, Cate, Musterstraße 24, 32132 Buchhausen
--- Befreundet mit: ---
    Doe, Joan, Autorenstr. 12, 32132 Buchhausen

    Leenings, Ramones, Autorenstr. 22, 32132 Buchhausen
    --- Befreundet mit: ---
        Hora, Oli, Autorenstr. 22, 32132 Buchhausen



Zweite Adresse:
==============
Leenings, Ramones, Autorenstr. 22, 32132 Buchhausen
--- Befreundet mit: ---
    Hora, Oli, Autorenstr. 22, 32132 Buchhausen
```

Das Programm, das diese Ausgabe hervorgebracht hat, sieht folgendermaßen aus:

```
Module mdlMain

    Sub Main()
        Dim locErsteAdr As New Adresse("Cate", "Hora", "Musterstraße 24", "32132 Buchhausen")
        Dim locZweiteAdr As New Adresse("Ramones", "Leenings", "Autorenstr. 22", "32132 Buchhausen")
        locErsteAdr.BefreundetMit = New List(Of Adresse) From
            {New Adresse("Joan", "Doe", "Autorenstr. 12", "32132 Buchhausen"),
             locZweiteAdr}
```

```
        locZweiteAdr.BefreundetMit = New List(Of Adresse) From
            {New Adresse("Oli", "Hora", "Autorenstr. 22", "32132 Buchhausen")}

        'Wenn Sie diese Zeile einbauen, erstellen Sie einen Zirkelverweis
        'locZweiteAdr.BefreundetMit.Add(locErsteAdr)

        'Originaladresse ausgeben
        Console.WriteLine("Erste Adresse:")
        Console.WriteLine("==============")
        Console.WriteLine(locErsteAdr)

        Console.WriteLine("Zweite Adresse:")
        Console.WriteLine("==============")
        Console.WriteLine(locZweiteAdr)

        'Kopie anlegen
        Dim locAdrKopie As Adresse = DirectCast(ObjectCloner.
            DeepCopy(locErsteAdr), Adresse)
        Console.ReadLine()
    End Sub
End Module
```

Sie erkennen am Ergebnis und dem Programmtext, dass das Programm auch verschachtelte Daten in der Eigenschaft `BefreundetMit` (und der dahinter steckenden `List(Of Adresse)`) bei der Ausgabe berücksichtigt. Wenn es die Liste der befreundeten Personen erstellt und eine Person dieser Liste wieder Personeneinträge unter `BefreundetMit` führt, werden diese bei der Ausgabe ebenfalls berücksichtigt.

Eine solche Vorgehensweise kann für ein Programm allerdings wirklich fatale Folgen haben. Denn wenn eine der Personen der `BefreundetMit`-Liste in ihrer Liste die Person führt, die ihr ebenfalls zugeordnet ist (Person A ist befreundet mit Person B, und Person B ist ebenfalls befreundet mit Person A – was ja durchaus nichts Ungewöhnliches ist), dann haben Sie es mit einem Zirkelverweis zu tun.

Der `ToString`-Algorithmus versagt in diesem Fall, weil er sich durch den Zirkelverweis immer wieder selbst aufruft. Sie können dieses Verhalten testen, indem Sie die Auskommentierung der im Listing fett gesetzten Zeile aufheben und den Auflösungsversuch eines Zirkelverweises initiieren. Wenn Sie das Programm anschließend starten, erscheint nach einer Weile im Ausgabefenster die Meldung

```
Process is terminated due to StackOverflowException
```

Das Programm »hängt« eine Weile, und je nachdem, wie gut oder schlecht Ihr System mit der nur noch ungenügend zur Verfügung stehenden Menge an Stack-Speicher klarkommt, sehen Sie unter Umständen noch kurz darauf die folgende Ausnahme aus Abbildung 21.5.[2]

[2] Lassen Sie es niemals soweit kommen. Nehmen Sie… - Scherz bei Seite: Wenn eine `StackOverflowException` ausgelöst wird, ist bereits alles zu spät – sie lässt sich nur in Ausnahmefällen abfangen. Und diese Ausnahmefälle sind nur dann gegeben, wenn Sie sie selber ausgelöst haben. Kommt Sie aus der CLR, haben Sie verloren. Seien sie also sehr vorsichtig, und vermeiden Sie Rekursionen ohne Abbruchbedingungen unbedingt!

Abbildung 21.5 Durch den Zirkelverweis versagt `ToString` der Beispielklasse, da es sich immer wieder selbst aufruft und dabei allen verfügbaren Stack-Speicher verbraucht

Viele Serialisierungsalgorithmen versagen ebenfalls durch eine solche Konstellation der Datenklassen. Dass die `DeepCopy`-Methode jedoch funktioniert und der .NET-Serializer keine Fehlermeldung auslöst, können Sie ganz einfach testen, indem Sie alle `Console.WriteLine`-Befehle auskommentieren. Das Programm passiert nach einem erneuten Start anstandslos die Zeile

```
Dim locAdrKopie As Adresse = DirectCast(ObjectCloner.
                            DeepCopy(locErsteAdr), Adresse)
```

beendet anschließend ordnungsgemäß und beweist so, dass der Serialisierungsalgorithmus von .NET Framework über Zirkelverweise erhaben ist.

Serialisierung von Objekten unterschiedlicher Versionen beim Einsatz von BinaryFormatter- oder SoapFormatter-Klassen

Einer sehr wichtigen Sache sollten Sie sich bewusst sein: Wenn Sie zur Serialisierung `BinaryFormatter` oder `SoapFormatter` verwenden, um Objekte in Dateien zu serialisieren, dann können Sie bei einem Update eines Programms leicht Gefahr laufen, in Versionskonflikte zu geraten.

Angenommen, Sie verfügen über eine Datenebene in Ihrer Applikation, die durch ein Objekt repräsentiert wird. Dieses Objekt kapselt alle Daten Ihrer Anwendung. Da Sie dem Benutzer natürlich die Möglichkeit geben müssen, die Daten als Datei auf einem Datenträger zu speichern, nutzen Sie dafür die Serialisierungsfunktionen von .NET.

Nun erweitern Sie durch ein Update Ihr Programm, und es kommen einige Eigenschaften zur Datenklasse hinzu. Da sich die Deserialisierungsfunktionen beider Serialisierungsklassen am Objekt orientieren, erwartet es die Daten zu einigen Eigenschaften, die es im alten Objektmodell natürlich nicht gegeben hat. Der Deserialisierungsversuch schlägt in diesem Fall mit einer Ausnahme fehl.

Eine Möglichkeit, das zu verhindern, besteht durch den Einsatz der so genannten `SerializationBinder`-Klasse, die für die Lösung des Versionsproblems konzipiert wurde. Mehr über dieses Objekt erfahren Sie in der Visual Studio-Onlinehilfe.

Eine andere Möglichkeit, das Versionsproblem in den Griff zu bekommen, ist die Anwendung der XML-Serialisierung, die im folgenden Abschnitt besprochen wird.

XML-Serialisierung

Die XML-Serialisierung erfolgt generell nach dem gleichen Prinzip, wie Sie es beim Einsatz von `BinaryFormatter` und `SoapFormatter` kennen gelernt haben. Die XML-Serialisierung hat im Gegensatz dazu entscheidende Vorteile:

- Das Problem mit der Serialisierung von Klassen unterschiedlicher Versionen betrifft sie nicht, denn es werden nur die Daten deserialisiert, die einerseits vorhanden sind, andererseits auch in den zu deserialisierenden Klassen zur Verfügung stehen
- XML-Code ist nicht nur leichter lesbar, sondern kann auch von den verschiedensten Programmen importiert und weiterverarbeitet werden

Allerdings müssen Sie beim Erstellen von Klassen, die Sie im XML-Format serialisieren wollen, auch einige Kompromisse eingehen:

- Wie bei anderen Klassen, die Sie mit .NET-Serialisierungstechniken serialisieren, müssen die Klassen mit dem `Serializable`-Attribut gekennzeichnet werden
- Eine Klasse, die im XML-Format serialisiert werden soll, muss über einen parameterlosen Konstruktor (`Sub New()`) verfügen
- Nur Feldvariablen bzw. Eigenschaften, die *öffentlichen* Zugriff haben, können serialisiert werden

Wenn Ihre Klassen diese Voraussetzungen erfüllen, steht einer Serialisierung nichts mehr im Weg.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 21\XMLSerialization
```

Dieses Beispielprojekt entspricht im Großen und Ganzen dem `List(Of Type)`-Beispiel aus Kapitel 20. Es erlaubt das Generieren von Zufallsadressen und deren Darstellung in einer Liste. Im Gegensatz zum Beispiel aus Kapitel 20 verfügt es jedoch über eine richtige Menüstruktur und in dieser Ausbaustufe über ein paar zusätzliche Funktionen, die über dieses Menü erreichbar sind:

Abbildung 21.6 Die Adressenverwaltung »Adresso.NET« erlaubt es in dieser Ausbaustufe, Zufallsadressen zu generieren und erstellte Listen in Dateien zu speichern und zu laden

Sie können eine beliebige Anzahl zufälliger Adressen erstellen, indem Sie den Menüeintrag *Datei/Zufallsadressen anfügen* wiederholt aufrufen. Mit *Datei/Adressliste speichern* erstellen Sie die so generierte Adressliste als Datei im XML-Format. Über den umgekehrten Weg lässt sich eine Liste, die als XML-Datei vorliegt, über den Menüeintrag *Datei/Adressliste öffnen* in die entsprechenden `Adresse`-Objekte umwandeln, die dann wieder im `ListView`-Steuerelement des Programms angezeigt werden.

Neben der Darstellung der Liste und der Auswertung der Menüereignisse besteht der Code dieses Beispiels aus zwei zentralen Klassen. Die erste Klasse `Adresse` kennen Sie aus vorherigen Beispielen bereits, und ihre Codebesonderheiten sind schnell erklärt:

```
<Serializable()>
Public Class Adresse

    ''' <summary>
    ''' Erstellt eine neue Instanz dieser Klasse.
    ''' </summary>
    ''' <remarks>Ein parameterloser Konstruktor wird benötigt,
    ''' um XML-Serialisierung zu ermöglichen.</remarks>
    Sub New()
        MyBase.New()
    End Sub

    'Konstruktor - legt eine neue Instanz an
    Sub New(ByVal Matchcode As String, ByVal Name As String, ByVal Vorname As String, _
            ByVal Straße As String, ByVal Plz As String, ByVal Ort As String)
        Me.Matchcode = Matchcode
```

```
        Me.Name = Name
        Me.Vorname = Vorname
        Me.Straße = Straße
        Me.PLZ = Plz
        Me.Ort = Ort
    End Sub

    Public Overridable Property Matchcode() As String
    Public Overridable Property Name() As String
    Public Overridable Property Vorname() As String
    Public Overridable Property Straße() As String
    Public Overridable Property PLZ() As String
    Public Overridable Property Ort() As String

    Public Overrides Function ToString() As String
        Return Matchcode & ":" & Name & ", " & Vorname
    End Function
End Class
```

Drei Dinge sind wichtig, damit die XML-Serialisierung mit Auflistungen von Elementen dieses Typs funktioniert:

- Das Attribute `Serializable` muss auf die Klasse angewendet werden (siehe oberste, fett hervorgehobene Listingzeile)
- Die Klasse muss über einen parameterlosen Konstruktor verfügen (darunter, ebenfalls in Fettschrift)
- Alle Eigenschaften, die serialisiert werden sollen, müssen öffentlich zugänglich sein. Im Beispiellisting sind die Eigenschaften deswegen auch mit dem `Public`-Modifizierer ausgestattet.

Gespeichert werden die einzelnen `Adresse`-Elemente in einer Auflistung namens `Adressen`, die aus der generischen Klasse `List(Of Adresse)` abgeleitet wird (mehr zur generischen Auflistung erfahren Sie ebenfalls in Kapitel 20). Diese Klasse sieht folgendermaßen aus:

```
<Serializable()>
Public Class Adressen
    Inherits List(Of Adresse)

    ''' <summary>
    ''' Erstellt eine neue Instanz dieser Klasse.
    ''' </summary>
    ''' <remarks></remarks>
    Sub New()
        MyBase.New()
    End Sub

    ''' <summary>
    ''' Erstellt eine neue Instanz dieser Klasse
    ''' und ermöglicht das Übernehmen einer vorhandenen Auflistung in diese.
    ''' </summary>
    ''' <param name="collection">Die generische Adresse-Auflistung, deren Elemente
    ''' in diese Auflistungsinstanz übernommen werden sollen.</param>
    ''' <remarks></remarks>
    Sub New(ByVal collection As ICollection(Of Adresse))
        MyBase.New(collection)
    End Sub
```

```
    ''' <summary>
    ''' Serialisiert alle Elemente dieser Auflistung in eine XML-Datei.
    ''' </summary>
    ''' <param name="Dateiname">Der Dateiname der XML-Datei.</param>
    ''' <remarks></remarks>
    Sub XMLSerialize(ByVal Dateiname As String)
        Dim locXmlWriter As New XmlSerializer(GetType(Adressen))
        Dim locXmlDatei As New StreamWriter(Dateiname)
        locXmlWriter.Serialize(locXmlDatei, Me)
        locXmlDatei.Flush()
        locXmlDatei.Close()
    End Sub

    ''' <summary>
    ''' Generiert aus einer XML-Datei eine neue Instanz dieser Auflistungsklasse.
    ''' </summary>
    ''' <param name="Dateiname">Name der XML-Datei, aus der die Daten für
    ''' diese Auflistungsinstanz entnommen werden sollen.</param>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public Shared Function XmlDeserialize(ByVal Dateiname As String) As Adressen
        Dim locXmlLeser As New XmlSerializer(GetType(Adressen))
        Dim locXmlDatei As New StreamReader(Dateiname)
        Return CType(locXmlLeser.Deserialize(locXmlDatei), Adressen)
    End Function

    ''' <summary>
    ''' Liefert eine Instanz dieser Klasse mit Zufallsadressen zurück.
    ''' </summary>
    ''' <param name="Anzahl">Anzahl der Zufallsadressen, die erzeugt werden sollen.</param>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public Shared Function ZufallsAdressen(ByVal Anzahl As Integer) As List(Of Adresse)
.
.   ' Ausgelassener Code; unwichtig an dieser Stelle
.
    End Function
End Class
```

Auch diese Klasse, die die `Adresse`-Elemente speichert, erfüllt die Voraussetzungen für die XML-Serialisierung – sie verfügt über das notwendige Attribut, über einen parameterlosen Konstruktor und über öffentliche Eigenschaften. Die einzige öffentliche Eigenschaft, auf die es bei dieser Klasse ankommt, ist übrigens im Code nicht zu erkennen: Es ist `Items`, die durch die Basisklasse `List(Of Adresse)` definiert wurde und per Vererbung natürlich auch in der neuen `Adressen`-Klasse zur Verfügung steht. Sie erlaubt dem XML-Serialisierer Zugriff auf die einzelnen `Adresse`-Elemente und damit letzten Endes auf jedes einzelne Datenfeld von `Adresse`.

Beim ersten Blick auf den Code dieser Klasse mag es vielleicht verwirren, dass die Methode zum Serialisieren des Klasseninhalts eine nicht statische, die zum Deserialisieren hingegen eine statische ist. Doch das ist klar: Beim Deserialisieren gibt es noch keine Instanz der `Adressen`-Klasse; also existiert auch keine Membermethode, die aufgerufen werden könnte, denn die Instanz mit den `Adresse`-Elementen geht ja erst durch den Deserialisierungsprozess hervor!

Anders ist es beim Serialisieren: Hier sind die einzelnen Elemente bereits in der Instanz von `Adressen` vorhanden, und aus diesem Grund kann es auch eine Membermethode sein, die das Serialisieren vornimmt; sie greift auf ihre eigenen Elemente zu.

Das Anwenden dieser beiden Methoden ist schließlich eine Kleinigkeit. Die beiden folgenden Ereignisbehandlungsmethoden, die das Auswählen der entsprechenden Menüeinträge auswerten, sind dafür verantwortlich (relevante Codeteile sind wieder fett hervorgehoben):

```
    'Wird aufgerufen, wenn der Anwender Datei/Adressliste öffnen wählt.
    Private Sub tsmAdresslisteLaden_Click(ByVal sender As System.Object,
                        ByVal e As System.EventArgs) Handles tsmAdresslisteLaden.Click

        Dim dateiÖffnenDialog As New OpenFileDialog
        With dateiÖffnenDialog
            .CheckFileExists = True
            .CheckPathExists = True
            .DefaultExt = "*.xml"
            .Filter = "Adresso XML-Dateien" & _
                  " (" & "*.adrxml" & ")|" & "*.adrxml" & "|Alle Dateien (*.*)|*.*"
            Dim dialogErgebnis As DialogResult = .ShowDialog
            If dialogErgebnis = Windows.Forms.DialogResult.Cancel Then
                Exit Sub
            End If

            'Adressen deserialisieren
            myAdressen = Adressen.XmlDeserialize(.FileName)

            'Liste neu darstellen
            ElementeDarstellen()
        End With
    End Sub

    'Wird aufgerufen, wenn der Anwender Datei/Adressliste speichern wählt.
    Private Sub tsmAdresslisteSpeichern_Click(ByVal sender As System.Object,
                        ByVal e As System.EventArgs) Handles tsmAdresslisteSpeichern.Click

        Dim dateiSpeichernDialog As New SaveFileDialog

        With dateiSpeichernDialog
            .CheckPathExists = True
            .DefaultExt = "*.xml"
            .Filter = "Adresso XML-Dateien" & " (" & "*.adrxml" & ")|" & _
                   "*.adrxml" & "|Alle Dateien (*.*)|*.*"
            Dim dialogErgebnis As DialogResult = .ShowDialog
            If dialogErgebnis = Windows.Forms.DialogResult.Cancel Then
                Exit Sub
            End If

            'Adressen serialisieren
            myAdressen.XMLSerialize(.FileName)
        End With
    End Sub
End Class
```

Prüfen der Versionsunabhängigkeit der XML-Datei

Die XML-Datei, die dieses Programm generiert, sieht auszugsweise etwa wie in Abbildung 21.7 aus, und Sie können sich »Ihre« Version mit Notepad anschauen.

Wenn Sie ein Datenfeld aus dieser XML-Datei entfernen, etwa wie in Abbildung 21.7 zu sehen, und anschließend die Datei speichern, können Sie diese XML-Datei dennoch deserialisieren lassen – für die `Ort`-Eigenschaft des betroffenen `Adresse`-Objekts wird dann beim Deserialisieren eben keine Zuordnung vorgenommen. Der Deserialisierungsprozess schlägt aber nicht fehl.

Abbildung 21.7 Auch wenn Sie ein Datenfeld aus der XML-Datei entfernen, lässt sich die Datei anschließend noch deserialisieren

Genauso könnten Sie der `Adresse`-Klasse eine zusätzliche Eigenschaft hinzufügen. Ältere Versionen der Serialisierungsdateien ließen sich dennoch weiterverwenden.

Serialisierungsfehler bei KeyedCollection

Die generische `KeyedCollection`-Klasse eignet sich als Auflistungsbasis für Geschäftsobjekte ideal, denn:

- sie stellt, da sie generisch ist, eine typsichere Auflistung des jeweiligen Elements dar
- Objekte lassen sich wahlweise über einen Index *oder* über einen Schlüssel abrufen. Den Letzteren muss das Objekt allerdings selbst zur Verfügung stellen – Kapitel 20 liefert im entsprechenden Abschnitt mehr zu diesem Thema.
- Da die `KeyedCollection` auch über einen Indexer verfügt, stellt sie auch eine Enumerierungstechnik für `ForEach` zur Verfügung, obwohl Sie zusätzlich Wörterbuchfunktionalität implementiert.

An Flexibilität ist diese Klasse also in Sachen Auflistung kaum zu übertreffen. Allerdings ist diese Kombination auch gerade dann eine Krux, wenn sich Schlüsseldatentyp und Indexer typtechnisch ins Gehege kommen, denn: Stellen Sie sich vor, Sie haben eine Klasse wie die folgende, die den eindeutigen Schlüssel eines Geschäftsobjekts über eine ID zur Verfügung stellt (und dieser Anwendungsfall ist keinesfalls konstruiert, denn gerade in der Datenbankprogrammierung haben Tabellen oft den Datentyp `Integer` als primären Schlüssel und Hauptindex):

BEGLEITDATEIEN Das Projekt zu diesem Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 21\KeyedCollection
```

```
Public Class Adresse
.
.
.
    Sub New(ByVal IDAdresse As Integer, ByVal Nachname As String, ByVal Vorname As String)
        myIDAdresse = IDAdresse
        myNachname = Nachname
        myVorname = Vorname
    End Sub
.
.
.
    Public Property IDAdresse() As Integer
.
.
.
    Public Property Nachname() As String
.
.
.
    Public Property Vorname() As String
.
.
.
End Class
```

Dieses Geschäftsobjekt speichert also eine simple Adresse – wesentliche Eigenschaften sind hier aus Platzgründen ausgelassen.

Was viel wichtiger ist: Ein Geschäftsobjekt dieses Typs identifiziert sich eindeutig durch seine `IDAdresse`-Eigenschaft, und das Problem dabei ist: Diese ist vom Typ `Integer`. Warum das ein Problem darstellt, zeigt die Implementierung einer `KeyedCollection`-Auflistung auf Basis dieser Klasse, denn es bietet sich mehr als an, dass `IDAdresse` auch als Quelle für die Schaffung des eindeutigen Schlüssels verwendet wird:

```
Public Class Adressen
    Inherits System.Collections.ObjectModel.KeyedCollection(Of Integer, Adresse)

    Protected Overrides Function GetKeyForItem(ByVal item As Adresse) As Integer
        Return item.IDAdresse
    End Function
End Class
```

Während der Aufbau einer `KeyedCollection` in dieser Struktur noch ohne Probleme möglich ist

```
    Protected Overrides Sub OnLoad(ByVal e As System.EventArgs)
        MyBase.OnLoad(e)
        myAdressen.Add(New Adresse(2, "Löffelmann", "Klaus"))
        myAdressen.Add(New Adresse(4, "Leenings", "Ramona"))
        myAdressen.Add(New Adresse(6, "Sonntag", "Miriam"))
        myAdressen.Add(New Adresse(8, "Dröge", "Ruprecht"))
        myAdressen.Add(New Adresse(10, "Vielstedde", "Anja"))
    End Sub
```

zeigt sich beim Abrufen der Elemente bereits eine Besonderheit, wenn es sich beim Schlüssel um den Datentyp `Integer` handelt (Abbildung 21.8).

```
Private Sub btnObjekteAusgeben_Click(ByVal sender As System.Object
    For count As Integer = 0 To 4
        Debug.Print(myAdressen(
    Next
End Sub
```

Item(**key As Integer**) As KeyedCollection.Adresse
Ruft das Element mit dem angegebenen Schlüssel ab.
key: *Der Schlüssel des abzurufenden Elements.*

Abbildung 21.8 Handelt es sich um einen `Integer`-Schlüssel, bietet die `KeyedCollection`-Auflistung nur eine einzige Signatur zum Abrufen der Elemente an

Soweit macht das auch Sinn. Da Indexer und Schlüssel in diesem Fall vom gleichen Datentyp sind, könnten weder Compiler noch .NET Framework unterscheiden, welche »Version« der Abruffunktion sie verwenden sollen – die durch den Indexer oder die über den Schlüssel. Einzig und allein die Methodensignatur (also welche Parametertypen übergeben werden) entscheidet nämlich darüber, und da es in diesem Fall die gleichen Typen sind, kann keine Signaturenunterscheidung stattfinden. Ergo: Es gibt nur eine Möglichkeit, an das Element heranzukommen.

Anders wäre das, wenn unser Schlüssel beispielsweise vom Datentyp `Long` wäre.[3] In diesem Fall würde der Indexer über einen `Integer` und der Schlüssel über den Datentyp `Long` angesprochen werden, und dementsprechend würde Ihnen bereits IntelliSense zwei Möglichkeiten zum Abrufen der Elemente anbieten (Abbildung 21.9).

```
Private Sub btnObjekteAusgeben_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handl
    For count As Integer = 0 To 4
        Debug.Print(myAdressen(
    Next
End Sub

'For Each locAdresse As Adresse
'    Debug.Print(locAdresse.Nac
'Next
```

▲ 1 von 2 ▼ Item(**index As Integer**) As KeyedCollection.Adresse
Ruft das Element am angegebenen Index ab oder legt dieses fest.
index: *Der nullbasierte Index des Elements, das abgerufen oder festgelegt werden soll.*

▲ 2 von 2 ▼ Item(**key As Long**) As KeyedCollection.Adresse
Ruft das Element mit dem angegebenen Schlüssel ab.
key: *Der Schlüssel des abzurufenden Elements.*

perty AcceptButton
:haltfläche im Form

Abbildung 21.9 Unterscheiden sich Schlüssel und Indexer im Datentyp, gibt es bei der `KeyedCollection` tatsächlich zwei Möglichkeiten, an ein einzelnes Element heranzukommen

Diese `Integer`/`Integer`-Problematik wäre noch nicht weiter tragisch, denn durch den von `KeyedCollection` implementierten Enumerator gäbe es immer noch die Möglichkeit, die Objekte einerseits per Schlüssel, andererseits nacheinander abzurufen. Für letztere Methode würden Sie einfach eine `ForEach`-Schleife verwenden.

Richtig problematisch wird es aber, wenn Sie versuchen, die Daten zu serialisieren. Denn dann passiert das in Abbildung 21.10 Gezeigte.

[3] Lesen Sie aber bitte diesen Abschnitt zunächst zu Ende, bevor Sie jetzt schon alle Ihre `KeyedCollection` basierenden Auflistungsklassen auf den Datentyp `Long` als Schlüssel umstellen!

Abbildung 21.10 Beim Serialisieren stellen sich massive Probleme ein, wenn Schlüsseltyp und Indexer gleichermaßen vom Typ `Integer` sind!

Der Serialisierungsalgorithmus kommt hier offensichtlich nicht mehr mit der Doppelbelegung der Datentypen zurecht. Offensichtlich versucht er über eine Zählvariable der Reihe nach auf die Objekte zuzugreifen, die aber der Indexer von `KeyedCollection` fälschlicherweise als Schlüssel interpretiert. Da in unserem Beispiel die IDs nicht der Nummerierungsreihenfolge der Objekte entsprechen, schlägt dieser Versuch fehl, und das Programm steigt mit einer Ausnahme, wie in der Abbildung zu sehen, aus.

Wenn Sie also die `KeyedCollection` verwenden und Ihr Key aus einem `Integer`-Wert besteht, achten Sie darauf, dass Sie beispielsweise den Datentyp `Long` für den Key verwenden, damit der Compiler `Integer` als Indexer erkennen kann, oder schreiben Sie eine Key-Hilfsklasse, mit der Sie auf die Elemente zugreifen.

Kapitel 22

Attribute und Reflection

In diesem Kapitel:

Genereller Umgang mit Attributen 694
Einführung in Reflection 697
Erstellung benutzerdefinierter Attribute und deren Erkennen zur Laufzeit 705

Wenn Sie größere Anwendungen entwickeln, kommen Sie an Attributen nicht vorbei. Attribute sind besondere Klassen, die keine spezielle Funktion ausführen, sondern nur ein Hinweis für eine andere Instanz sind, ein bestimmtes Element Ihrer Anwendung auf besondere Weise zu behandeln.

Sie können nicht nur auf die in .NET Framework vorhandenen Attribute zurückgreifen, sondern auch eigene Attributklassen entwerfen. In diesem Fall müssen Sie allerdings auch Techniken beherrschen, die das Erkennen von Attributen zur Laufzeit ermöglichen – und an dieser Stelle kommt *Reflection* ins Spiel.

Die Reflection-Techniken in .NET Framework stellen im Prinzip den Psychologen Ihres Programms dar, und sie helfen Ihnen, Informationen über bestimmte Assemblys, Klassen, Methoden, Eigenschaften und weitere Elemente zur Laufzeit Ihres Programms zu erhalten. Zu diesen Elementen gehören auch Attribute. Wenn Sie herausfinden wollen, ob eine bestimmte Methode einer Klasse oder Klasseninstanz beispielsweise mit einem bestimmten Attribut ausgestattet ist, verwenden Sie die Techniken der Reflection, um diese Attribute zu ermitteln. Gerade benutzerdefinierte Attribute und Reflection sind also zwei Themenbereiche, die eng miteinander verknüpft sind, und aus diesem Grund finden Sie diese beiden Themen auch in einem gemeinsamen Kapitel an dieser Stelle.

Damit klarer wird, welche enormen Möglichkeiten Ihnen Attribute und Reflection bieten können, finden Sie im Folgenden ein paar praktische Beispiele:

- Sie möchten, dass eine bestimmte Klasse Ihrer Datenbankanwendung automatisch die vorhandenen Datenbankfelder synchronisiert: Bestimmte Klassen sollen Tabellen darstellen, die Eigenschaften dieser Klassen die Datenfelder. Durch Attribute haben Sie die Möglichkeit, diese Klassen zu kennzeichnen und die Datenbankdatei zur Laufzeit zu synchronisieren.
- Formulare könnten sich in Abhängigkeit bestimmter Klassen selbst erstellen und die Eingabe bzw. Änderung einer Klasseninstanz übernehmen
- Eine Ableitung des `ListView`-Steuerelements könnte ein Array mit Instanzen besonders gekennzeichneter Klassen automatisch anzeigen. Attribute würden dann bestimmen, welche Eigenschaften eines Array-Elements als Spalte verwendet würden. Die Reihenfolge der Spalten ließe sich durch weitere Attribute bestimmen.

Sie sehen: Beispiele gibt es viele, und vielleicht ahnen Sie schon, welch mächtiges Werkzeug Ihnen .NET Framework mit Attributen und Reflection in die Hände legt.

Genereller Umgang mit Attributen

Wie in der Einführung schon kurz angerissen, und wie Sie es in den zahlreichen vergangenen Beispielprogrammen schon zigfach gesehen haben, werden Attribute ganz anders als herkömmliche Klassen verwendet (wenn sie auch auf dieselbe Weise erstellt werden). Attribute statten Klassen oder Prozeduren mit besonderen Eigenschaften aus. Genauso, wie die Verwendung von **Fettschrift innerhalb eines Absatzes dieses Buchs** selbst nicht den Sinn des Geschriebenen verändert, so erfüllt das Attribut Fettschrift dennoch seinen Zweck: Es ist der Hinweis für Sie, die so markierte Textstelle aufmerksamer zu lesen.

Attribute »markieren« eine Klasse oder eine Prozedur in Visual Basic, indem sie in Kleiner-/Größerzeichen eingeschlossen vor die Klassen- bzw. Prozedurdefinition gesetzt werden, etwa:

```
<MeinAttribute(MöglicherParameter)> Class MeineKlasse

End Class
```

Da Zeilen, in denen Attribute verwendet werden, auf diese Weise unnötig lang und damit schlecht lesbar sind, umbricht man die Zeilen am besten wie in diesem Beispiel zu sehen. Seit Visual Basic 2010 brauchen Sie dazu den Unterstrich (»_«) nicht mehr zu verwenden, um die Zeile zu umbrechen (siehe auch in Kapitel 5):

```
<MeinAttribute(MöglicherParameter)>
Class MeineKlasse

End Class
```

Die Auswirkungen, die ein Attribut auf den folgenden Code hat, werden nicht durch die Attributklasse gesteuert, sondern ausschließlich von den Instanzen, die die Elemente unter die Lupe nehmen, die mit Attributen versehen sind. Attributklassen sind in der Regel Klassen, die keine wirkliche Funktionalität zur Verfügung stellen; sie dienen lediglich als Container für Informationen. Behalten Sie diese wichtige Aussage im Hinterkopf, wenn Sie den Einsatz von Attributen planen.

WICHTIG Attributklassen enden grundsätzlich auf den Namen ...`Attribute`. Dennoch müssen Sie den Namenszusatz `Attribute` nicht mit angeben, um eine attributbasierende Klasse für Kennzeichnungszwecke zu verwenden.

Einsatz von Attributen am Beispiel von ObsoleteAttribute

Ein Beispiel soll das verdeutlichen: Die `ObsoleteAttribute`-Klasse dient beispielsweise dazu, eine Prozedur oder Klasse als »überholt« (das heißt, sie wird in zukünftigen Versionen eventuell nicht mehr verfügbar sein) zu kennzeichnen. Gesetzt den Fall, Sie haben vor einem Jahr eine Klassenbibliothek für .NET entwickelt, die Sie Ihren Kunden nun in erweiterter und überarbeiteter Form zugänglich machen wollen. Im Rahmen der Umbauarbeiten haben Sie festgestellt, dass Sie bestimmte Funktionen nicht mehr benötigen, weil Ihre Klassenbibliothek entweder sehr viel stärker automatisierter arbeiten kann, oder es Sinn ergibt, bestimmte Funktionen durch neuere Funktionen zu ersetzen, da diese viel effizienter arbeiten.

Natürlich können Sie in der neuen Version die alten, überflüssigen Funktionen nicht einfach entfernen. Würde der Benutzer Ihrer Klassenbibliothek nämlich anschließend sein Programm mit der neuen Version kompilieren, wären Fehlfunktionen vorprogrammiert (das Programm ließe sich wahrscheinlich gar nicht erst kompilieren). Mithilfe des `Obsolete`-Attributs können Sie den Entwickler aber gefahrlos darauf aufmerksam machen, eine bestimmte Funktion nicht mehr zu verwenden. Sie setzen in diesem Fall das `ObsoleteAttribute` vor die entsprechende Klasse bzw. Prozedur und geben zusätzlich eine Hinweismeldung an.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 22\AttributesDemo
```

Bei diesem Beispielprogramm kommt es gar nicht darauf an, das Programm zu starten. Wenn Sie es geladen haben, betrachten Sie vielmehr den Quellcode und wie die Fehlerliste auf die Verwendung einer bestimmten Eigenschaft reagiert:

Abbildung 22.1 Die Verwendung eines durch das `Obsolete`-Attribut gekennzeichneten Elements ruft mindestens eine entsprechende Warnmeldung hervor

Die Eigenschaft `AlteEigenschaft` der Klasse `TestKlasse` ist hier im Beispiel mit dem `Obsolete`-Attribut ausgestattet. Der Visual Basic-Compiler erkennt dieses Attribut und zeigt in der Fehlerliste eine Warnung, die den Entwickler auf diese Tatsache hinweist. Die `ObsoleteAttribute`-Klasse selbst hat aber nichts mit der Ausgabe des Textes zu tun, außer dass sie den Text, der ausgegeben werden soll, zur Verfügung stellt. Die eigentliche Ausgabe des Textes erfolgt durch den Visual Basic-Compiler bzw. die Entwicklungsumgebung.

TIPP Obwohl die `ObsoleteAttribute`-Klasse im Beispiel zur Kennzeichnung der `AlteEigenschaft`-Eigenschaft verwendet worden ist, lässt sich die Eigenschaft verwenden und das Programm damit auch kompilieren und starten. Wenn Sie möchten, dass der Einsatz einer veralteten Eigenschaft zum Compilerfehler führt, der dafür sorgt, dass sich das Programm nicht mehr starten lässt, ändern Sie hinter dem Meldungsstring im `ObsoleteAttribute`-Konstruktor den zweiten (booleschen) Parameter in `True`. Auf diese Weise wird die Kompilierung eines Programms verhindert, indem auf diese alte Version der Eigenschaft zurückgegriffen wird.

Die speziell in Visual Basic verwendeten Attribute

Die wichtigsten Attribute haben Sie im Laufe der vergangenen Kapitel themenbezogen schon kennengelernt. Es gibt allerdings einige spezielle Attribute für Visual Basic selbst,[1] die in der folgenden Tabelle zusammengefasst sind:

[1] Das bedeutet nicht, dass Sie nur diese Attribute in Visual Basic verwenden dürfen. Sie können natürlich alle Attribute von .NET Framework auch in Ihren eigenen Visual Basic-Anwendungen einsetzen. Um eine Liste aller in .NET Framework enthaltenen Attribute zu erhalten, rufen Sie die Onlinehilfe für die `Attribute`-Klasse auf und lassen sich anschließend alle abgeleiteten Klassen anzeigen.

Attribut	Zweck
`COMClassAttribute`-Klasse	Weist den Compiler an, die Klasse als COM-Objekt anzuzeigen
`VBFixedStringAttribute`-Klasse	Gibt die Größe einer Zeichenfolge mit fester Länge in einer Struktur an, die mit Dateiein- und -ausgabefunktionen verwendet werden soll. Spezifisch für Visual Basic.
`VBFixedArrayAttribute`-Klasse	Gibt die Größe eines festen Arrays in einer Struktur an, die mit Dateiein- und -ausgabefunktionen verwendet werden soll. Spezifisch für Visual Basic.
`SerializableAttribute`-Klasse	Gibt an, dass eine Klasse serialisiert werden kann
`MarshalAsAttribute`-Klasse	Legt fest, wie ein Parameter zwischen dem verwalteten Code von Visual Basic und nicht verwaltetem Code zum Beispiel einer Windows-API gemarshallt werden soll. Wird von der Common Language Runtime verwendet.
`AttributeUsageAttribute`-Klasse	Gibt die Verwendungsweise eines Attributs an
`DllImportAttribute`-Klasse	Gibt an, dass die Methode, die mit dem Attribut markiert ist, in einer nicht verwalteten DLL implementiert ist. P/Invoke-Betriebssystemaufrufe lassen sich damit beispielsweise realisieren.

Tabelle 22.1 Die speziellen Visual Basic-Attribute

Einführung in Reflection

Bevor wir im nächsten Schritt die beiden Themengebiete Reflection und Attribute miteinander verheiraten, lassen Sie uns zunächst einen Blick auf die Möglichkeiten der Reflection-Klassen selbst werfen.

Reflection selbst ist, wie am Anfang des Kapitels schon kurz zu erfahren war, der Oberbegriff für Techniken, die es einem Programm ermöglichen, etwas über Klassen zu erfahren, Klassen zu analysieren und auch Instanzen von Klassen programmtechnisch zu erstellen.

Natürlich ist Letzteres keine Kunst. Sie machen das schließlich immer, wenn Sie den Konstruktor einer Klasse verwenden. Aber darum geht es in diesem Fall auch gar nicht. Das Ziel ist vielmehr, Klassen zu verwenden, von denen das Programm zu dem Zeitpunkt, an dem es gestartet wird, noch nichts weiß.

Angenommen, Sie möchten eine Funktion zur Verfügung stellen, die ein beliebiges Objekt als Parameter übernimmt und den Wert jeder einzelnen Eigenschaft auf dem Bildschirm ausgibt. Mit herkömmlichen Mitteln könnten Sie dieses Vorhaben nicht realisieren, denn: Da Sie im Vorfeld nicht wissen, welchen Objekttyp Sie zu erwarten haben, kennen Sie dessen Eigenschaftennamen auch nicht. Folglich können Sie die Werte dieser Eigenschaften auch nicht abrufen.

Sie müssen also Mittel und Wege finden, ein Objekt zu analysieren und zunächst zu ermitteln, über welche Eigenschaften es verfügt. Erst im zweiten Schritt können Sie, wenn Ihr Programm die Namen der Eigenschaften herausgefunden hat, die Inhalte der Eigenschaften in Erfahrung bringen und schließlich auf dem Bildschirm ausgeben.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 22\Reflection01
```

Zentraler Bestandteil dieses Programms ist die `Adresse`-Klasse, die Sie aus vergangenen Kapiteln kennen. Die `Main`-Prozedur dieser Konsolenanwendung macht nichts weiter, als eine neue `Adresse`-Instanz zu erstellen und die Untersuchungsergebnisse im Konsolenfenster darzustellen, etwa wie im folgenden Bildschirmauszug zu sehen:

```
Attribute der Klasse:Reflection01.Adresse
Standardattribute:
    *AutoLayout, AnsiClass, Class, Public, Serializable

Member-Liste:
    *get_Name, Method
     --------
    *set_Name, Method
     --------
    *get_Vorname, Method
     -----------
    *set_Vorname, Method
     -----------
    *get_Straße, Method
     ----------
    *set_Straße, Method
     ----------
    *get_PLZOrt, Method
     ----------
    *set_PLZOrt, Method
     ----------
    *get_BefreundetMit, Method
     -----------------
    *set_BefreundetMit, Method
     -----------------
    *get_ErfasstAm, Method
    *get_ErfasstVon, Method
    *ToString, Method
    *ToShortString, Method
    *Equals, Method
     ------
    *GetHashCode, Method
     -----------
    *GetType, Method
     -------
    *.ctor, Constructor
    *.ctor, Constructor
    *Name, Property
     Wert: Adriana
    *Vorname, Property
     Wert: Beck
    *Straße, Property
     Wert: Buchstraße 223
    *PLZOrt, Property
     Wert: 32154 Autorhausen
    *BefreundetMit, Property
     Wert: System.Collections.Generic.List`1[Reflection01.Adresse]
    *ErfasstAm, Property
     Wert: 16.06.2010 19:59:31
    *ErfasstVon, Property
     Wert: klauslo
```

Sie sehen, dass dieses Programm nicht nur in der Lage ist, eine (fast) vollständige Memberliste der übergebenden Objektinstanz zu ermitteln; es kann darüber hinaus auch die Inhalte der Eigenschaften des Objekts anzeigen.

Bevor wir die Funktionsweise dieses Programms genauer unter die Lupe nehmen, lassen Sie mich zunächst ein paar Grundlagen zum Thema Typen und Reflection zwecks späteren besseren Verständnisses klären.

Die Type-Klasse als Ausgangspunkt für alle Typenuntersuchungen

Den Schlüssel dazu bilden Klassen und Methoden, die Sie im `Reflection`-Namespace von .NET Framework finden. Ausgangspunkt für alle Reflection-Operationen bildet dabei die so genannte `Type`-Klasse, die den Typ einer Objektvariablen zur Laufzeit ermitteln kann.

Die `Type`-Klasse selbst können Sie nicht instanziieren, da sie eine abstrakte Klasse darstellt. Sie können sie aber verwenden, um den Typ eines bestehenden Objekts zu ermitteln und festzuhalten. Zwar gehört die `Type`-Klasse selbst nicht zum `Reflection`-Namespace – Sie benötigen sie aber, um mit weiteren Klassen dieses Namespaces und deren Funktionen Informationen über die jeweilige Typdeklaration eines zu untersuchenden Objekts abzurufen, zum Beispiel Konstruktoren, Methoden, Felder, Eigenschaften und Ereignisse. Auch Informationen über Modul und Assembly, in denen die Klasse bereitgestellt wird, lassen sich anschließend mit diesen Funktionen ermitteln.

Jedes Objekt in .NET Framework stellt eine `GetType`-Funktion zur Verfügung, mit der ihr zugrunde liegendes `Type`-Objekt ermittelt werden kann. Darüber hinaus können Sie mit der statischen Funktionsvariante auch ein `Type`-Objekt erstellen, wenn nur der voll qualifizierte Name des Typs bekannt ist. Mit `TypeOf` in Zusammenhang mit dem `Is`-Operator können Sie einen Typenvergleich durchführen. Alternativ funktioniert das auch mit `GetType`, welches auch als Operator eingesetzt werden kann. Die folgenden Codeauszüge demonstrieren den generellen Einsatz dieser Funktionen:

```
'Ein paar Type-Experimente:
Dim locFreund As New Adresse("Beck", "Adriana", "Buchstraße 223", "32154 Autorhausen")

Dim locTest As New Adresse("Klaus", "Löffelmann", "Urlaubsgasse 17", "59555 Lippende")
Dim locType1, locType2 As Type
locType1 = locTest.GetType

'Wichtig: TypeOf funktioniert nur zusammen mit dem Is-Operator:
If TypeOf locTest Is Reflection01.Adresse Then
    Console.WriteLine("Adresse-Typ erkannt!")
Else
    Console.WriteLine("Adresse-Typ nicht erkannt!")
End If

'Und auch das ist eine Alternative, die dasselbe bewirkt:
If locTest.GetType Is GetType(Reflection01.Adresse) Then
    Console.WriteLine("Adresse-Typ erkannt!")
Else
    Console.WriteLine("Adresse-Typ nicht erkannt!")
End If

'So funktioniert's, wenn zwei Typobjekte
'miteinander verglichen werden sollen:

locType2 = GetType(Adresse)                     ' Der Normalfall, GetType als Operator
locType2 = GetType(Reflection01.Adresse)        ' Alternativ: mit Namespace
locType2 = Type.GetType("Reflection01.Adresse") ' Alternativ: aus String
```

```
        Console.WriteLine("Der Typ " + locType1.FullName +
                CStr(IIf(locType1 Is locType2, " entspricht ", " entspricht nicht ")) +
                "dem Typ " + locType2.FullName)

        'Rausfinden, ob eine Typvariable (zum Beispiel auch Basisklasse oder Interface)
        'Kompatibel zum eigentlich instanziierten Objekt ist.
        Dim adresseMitGebDatum As New AbgeleiteteAdresse With
            {.Name = "Leenings", .Vorname = "Ramona", .Straße = "Irgendwoweg 12",
             .PLZOrt = "Someplace-Hausen",
             .BefreundetMit = New List(Of Adresse) From {locTest, locFreund},
             .Geburtsdatum = #7/24/1991#}

        If GetType(Adresse).IsAssignableFrom(adresseMitGebDatum.GetType) Then
            Console.WriteLine(adresseMitGebDatum.ToShortString &
                              " kann in Typ 'Adresse' gespeichert werden.")
        End If

        Console.ReadLine()
```

Dieses Programm produziert die folgende Ausgabe:

```
Adresse-Typ erkannt!
Adresse-Typ erkannt!
Der Typ Reflection01.Adresse entspricht dem Typ Reflection01.Adresse
Leenings, Ramona kann in Typ 'Adresse' gespeichert werden.
```

HINWEIS Dieser Codeschnipsel ist ebenfalls im besprochenen Beispielprojekt vorhanden (direkt im Anschluss an die *Main*-Prozedur, die mit `Exit Sub` endet). Möchten Sie mit ihm experimentieren, kommentieren Sie das `Exit Sub` einfach aus.

Wenn Sie auf diese Weise ein `Type`-Objekt ermittelt haben, können Sie mithilfe seiner Funktionen weitere Informationen über den entsprechenden Typen erhalten.

Klassenanalysefunktionen, die ein Type-Objekt bereitstellt

Die folgende Tabelle gibt Ihnen einen Überblick über die wichtigsten Funktionen und Klassentypen, die von der `Type`-Klasse bereitgestellt werden und die dazu dienen, nähere Informationen zum entsprechenden Typ zu erhalten:

Funktion der Type-Klasse	Aufgabe
`Assembly`	Ruft die Assembly ab, die den Typ definiert. Rückgabetyp: `System.Reflection.Assembly`
`AssemblyQualifiedName`	Ruft den voll qualifizierten Assemblynamen ab. Rückgabetyp: `String`
`Attributes`	Ruft eine Bitkombination (*Flags-Enum*) ab, die Auskunft über alle nicht benutzerdefinierten Attribute gibt. Rückgabetyp: `TypeAttributes`
`BaseType`	Ruft den Typ ab, von dem der angegebene Typ direkt erbt. Rückgabetyp: `Type`
`FullName`	Ruft den voll qualifizierten Namen (einschließlich Namespace also) des angegebenen Typs ab. Rückgabetyp: `String`
`GetCustomAttributes`	Ruft ein Array mit allen benutzerdefinierten Attributen ab. Wurden für den Typ keine benutzerdefinierten Attribute definiert, wird `Nothing` zurückgegeben. Rückgabetyp: `Object()` ▶

Funktion der Type-Klasse	Aufgabe
GetEvent	Ruft das EventInfo-Objekt eines bekannten Ereignisses ab. Der Ereignisname wird als String übergeben. Rückgabetyp: EventInfo
GetEvents	Ruft eine Liste (als Array) mit allen Ereignissen des Objekts ab. Rückgabetyp: EventInfo()

GetField	Ruft das FieldInfo-Objekt eines bekannten öffentlichen Felds ab. Der Feldname wird als String übergeben. Rückgabetyp: FieldInfo
	
	WICHTIG: Felder sind einerseits oft nicht öffentlich (public) definiert und andererseits, anders als Eigenschaften, natürlich nur verfügbar, wenn es eine Instanz einer Klasse gibt, die Sie untersuchen. Wichtig ist deshalb, dass Sie mithilfe der so genannten BindingFlags, die steuern, *was* Sie ermitteln wollen, bestimmen, dass Sie eine Instanz der Klasse unter die Lupe nehmen (Reflection.Binding Flags.Instance) und dass Sie nicht öffentliche Elemente abrufen möchten (Reflection.BindingFlags.NonPublic). Anderenfalls werden Sie für die definierten privaten Feldvariablen einer Instanz kein Ergebnis zurückerhalten:
	
	`Dim retField=KlassenInstanz.GetType().GetField(`
	`  Reflection.BindingFlags.NonPublic Or Reflection.BindingFlags.Instance)`
	HINWEIS: Falls Sie alles außer private Felder zurückbekommen möchten, lassen Sie den BindingFlags-Parameter einfach ganz weg.

GetFields	Ruft eine Liste (als Array) mit allen öffentlichen Feldern des Objekts ab.
	Rückgabetyp: FieldInfo()
	WICHTIG: Es gilt das unter GetField (siehe oben) Gesagte – Sie müssen also BindingFlags angeben, die auf Instanz und nicht öffentliche Member abfragen. Eine Schleife, die durch die Felder iteriert, sollte also folgendermaßen ausschauen:
	`For Each item In Me.GetType().GetFields(`
	`  Reflection.BindingFlags.NonPublic Or Reflection.BindingFlags.Instance)`
	`  Debug.Print(item.ToString)`
	`Next`

GetMember	Ruft das MemberInfo-Objekt eines *Members*, dessen Namen Sie kennen, des Objekts ab. Ein *Member* ist eine Oberkategorie eines Elements eines Objekts, kann also beispielsweise eine Eigenschaft, eine Methode, ein Ereignis oder ein Feld sein. Mit der MemberType-Eigenschaft eines MemberInfo-Objekts können Sie feststellen, um was für ein Objektelement es sich handelt. Rückgabetyp: MemberInfo
GetMembers	Ruft eine Liste (als Array) aller Elemente (*Member*) des Objekts ab. Rückgabetyp: MemberInfo()
GetProperty	Ruft das PropertyInfo-Objekt einer Eigenschaft ab, dessen Namen Sie kennen. Der Eigenschaftenname wird als String übergeben. Rückgabetyp: PropertyInfo
GetProperties	Ruft eine Liste (als Array) aller Eigenschaften des Objekts ab. Rückgabetyp: PropertyInfo()

Tabelle 22.2 Die wichtigsten Funktionen, mit denen Sie Informationen über einen Typ abrufen können

Mit diesen Informationen können wir nun das Beispielprogramm betrachten. Es nutzt im Wesentlichen die GetMembers-Funktion des Type-Objekts, um die Informationen über ein beliebiges Objekt zu ermitteln:

```
Module mdlMain

    Sub Main()
        Dim locAdresse As New Adresse("Beck", "Adriana", "Buchstraße 223", "32154 Autorhausen") With
            {.BefreundetMit = New List(Of Adresse) From
                {New Adresse("Klaus", "Löffelmann", "One Microsoftway", "99999 Redmond")}}
```

```
        PrintObjectInfo(locAdresse)
        Console.ReadLine()
    End Sub

    Sub PrintObjectInfo(ByVal [Object] As Object)
        'Den Objekttypen ermitteln, um auf die Objektinhalte zuzugreifen
        Dim locTypeInstanz As Type = [Object].GetType

        'Die nicht benutzerdefinierten Attribute ausgeben:
        Console.WriteLine("Attribute der Klasse:" + locTypeInstanz.FullName)
        Console.WriteLine("Standardattribute:")
        Console.WriteLine("    *" + locTypeInstanz.Attributes.ToString())
        Console.WriteLine()

        'Members und deren mögliche Attribute ermitteln
        Dim locMembers() As MemberInfo

        'Holt alles (außer nicht-öffentliche Elemente);
        'BindingFlags kann dabei auch ausgelassen werden:
        locMembers = locTypeInstanz.GetMembers(BindingFlags.Default)

        'Führt dazu, dass nur nicht-öffentliche Felder geholt werden:
        'locMembers = locTypeInstanz.GetMembers(BindingFlags.Instance Or BindingFlags.NonPublic)
        Console.WriteLine("Member-Liste:")
        For Each locMember As MemberInfo In locMembers
            Console.WriteLine("    *" + locMember.Name + ", " _
                              + locMember.MemberType.ToString)
            If locMember.GetCustomAttributes(True).Length > 0 Then
                Console.WriteLine("     " + New String("-"c, locMember.Name.Length))
            End If

            'Wenn es sich um Eigenschaften handelt, holen wir auch den Wert.
            If locMember.MemberType = MemberTypes.Property Then
                Dim locPropertyInfo As PropertyInfo = DirectCast(locMember, PropertyInfo)
                Dim value = locPropertyInfo.GetValue([Object], Nothing)
                If value IsNot Nothing Then
                    Console.WriteLine("     Wert: " + value.ToString)
                End If
            End If

            'Feldinformationen ausgeben, wenn entsprechende
            'Bindingflags angegeben wurden.
            If locMember.MemberType = MemberTypes.Field Then
                Dim locFieldInfo As FieldInfo = DirectCast(locMember, FieldInfo)
                Dim value = locFieldInfo.GetValue([Object])
                If value IsNot Nothing Then
                    Console.WriteLine("     Wert: " + value.ToString)
                End If
            End If
        Next
    End Sub
End Module
```

TIPP Wenn Sie alle Elemente eines Typs ermitteln wollen, verwenden Sie die Funktion `GetMembers`, wie hier im Beispiel gezeigt (erster in Fettschrift gesetzter Block). Sie können anschließend `MemberType` jedes einzelnen Elements überprüfen, um herauszufinden, um was für einen Elementtyp es sich genau handelt (Eigenschaft, Methode etc.).

HINWEIS Beachten Sie auch, dass zu jeder Eigenschaft auch Methoden existieren. .NET Framework-intern werden Eigenschaften wie solche behandelt; sie bekommen dann entweder das Präfix `set_` oder `get_` verpasst, um die Zugriffsart unterscheiden zu können. Aus diesem Grund finden Sie in der Elementliste, die Sie mit `GetMembers` ermitteln, drei Elemente für jede Eigenschaft (zwei Methoden, eine Eigenschaft – vorausgesetzt natürlich, dass es sich dabei nicht um eine Nur-Lesen- oder um eine Nur-Schreiben-Eigenschaft handelt).

WICHTIG Sie können über eine `BindingsFlags`-Enum-Flag-Kombination angeben, wie die Elemente, die Sie ermitteln wollen, beschaffen sein müssen. Im Listing sind zwei Zeilen vordefiniert. Wenn Sie die Auskommentierung der zweiten Definition

```
locMembers = locTypeInstanz.GetMembers(BindingFlags.Instance Or BindingFlags.NonPublic)
```

zurücknehmen, so werden beispielsweise alle nicht öffentlichen Member der Instanz samt Inhalten ausgegeben – und dazugehören neben den beiden `Protected`-definierten Methoden, die `Adresse` von `Object` erben, auch alle nicht öffentlichen Feldvariablen. Das ist ganz interessant zu beobachten, denn hier lässt sich schön erkennen, wie der Visual Basic-Compiler durch Verwendung von automatisch implementierten Eigenschaften in unserer Testklasse `Adresse` tatsächlich die entsprechenden Hilfsfelder – auch Backing Fields genannt – (Feldvariablen) selbstständig angelegt hat:

```
Attribute der Klasse:Reflection01.Adresse
Standardattribute:
    *AutoLayout, AnsiClass, Class, Public, Serializable

Member-Liste:
    *Finalize, Method
     --------
    *MemberwiseClone, Method
     ---------------
    *myErfasstAm, Field
     Wert: 16.06.2010 22:29:56
    *myErfasstVon, Field
     Wert: klauslo
    *_Name, Field
     -----
     Wert: Adriana
    *_Vorname, Field
     --------
     Wert: Beck
    *_Straße, Field
     -------
     Wert: Buchstraße 223
    *_PLZOrt, Field
     -------
     Wert: 32154 Autorhausen
    *_BefreundetMit, Field
     --------------
     Wert: System.Collections.Generic.List`1[Reflection01.Adresse]
```

Objekthierarchie von MemberInfo und Casten in den spezifischen Info-Typ

`MemberInfo` ist die Basisklasse für alle spezifischeren Reflection-`XXXInfo`-Objekte. Von ihr abgeleitet sind:

- `EventInfo` zur Speicherung von Informationen über Ereignisse,
- `FieldInfo` zur Speicherung von Informationen über öffentliche Felder einer Klasse,
- `MethodInfo` zur Speicherung von Informationen über Methoden einer Klasse,
- `PropertyInfo` zur Speicherung von Informationen über die Eigenschaften einer Klasse.

Wenn Sie Informationen über eine Klasse oder ein Objekt mit der `GetMembers`-Funktion ermitteln, dann sind die spezifischeren Info-Objekte in den einzelnen `MemberInfo`-Objekten des `MemberInfo`-Arrays geboxt. Sie können sie mit einer `CType`- oder `DirectCast`-Anweisung in den eigentlichen Infotyp zurückwandeln, um auf spezielle Eigenschaften des Infotyps zugreifen zu können, etwa mit:

```
If locMember.MemberType = MemberTypes.Property Then
    Dim locPropertyInfo As PropertyInfo = DirectCast(locMember, PropertyInfo)
End If
```

Ermitteln von Eigenschaftswerten über PropertyInfo zur Laufzeit

Wenn Sie auf die beschriebene Weise ein `PropertyInfo`-Objekt ermittelt haben, können Sie auch den eigentlichen Wert der Eigenschaft abrufen. Voraussetzung dafür ist, dass es einen entsprechenden Typ gibt, der zuvor instanziiert wurde. Zum Abruf des Eigenschaftenwerts verwenden Sie die `GetValue`-Methode des `PropertyInfo`-Objekts. Das Beispielprogramm nutzt diese Vorgehensweise, um den aktuellen Inhalt einer Eigenschaft als Text anzuzeigen.

TIPP Da jede Klasse über eine zumindest standardmäßig implementierte `ToString`-Funktion verfügt, ist die Ermittlung des Werts als String sicher. Inwieweit `ToString` ein brauchbares Ergebnis zurückliefert, hängt natürlich von der Implementierung der `ToString`-Methode des jeweiligen Objekts ab.

Ein Beispiel für die Ermittlung von Eigenschafteninhalten oder Feldvariableninhalten von Objekten eines zuvor nicht bekannten Typs finden Sie ebenfalls im Beispielprogramm:

```
'Wenn es sich um Eigenschaften handelt, holen wir auch den Wert.
If locMember.MemberType = MemberTypes.Property Then
    Dim locPropertyInfo As PropertyInfo = DirectCast(locMember, PropertyInfo)
    Dim value = locPropertyInfo.GetValue([Object], Nothing)
    If value IsNot Nothing Then
        Console.WriteLine("     Wert: " + value.ToString)
    End If
End If

'Feldinformationen ausgeben, wenn entsprechende
'Bindingflags angegeben wurde.
If locMember.MemberType = MemberTypes.Field Then
    Dim locFieldInfo As FieldInfo = DirectCast(locMember, FieldInfo)
    Dim value = locFieldInfo.GetValue([Object])
    If value IsNot Nothing Then
        Console.WriteLine("     Wert: " + value.ToString)
    End If
End If
```

`GetValue` für Eigenschaften erfordert mindestens zwei Parameter: Der erste Parameter bestimmt, von welchem Objekt die angegebene Eigenschaft ermittelt werden soll. Da es Eigenschaften mit Parametern gibt, übergeben Sie im zweiten Parameter ein `Object`-Array, das diese Parameter enthält. Wenn die Eigenschaft parameterlos verwendet wird, übergeben Sie `Nothing` als zweiten Parameter, wie im Beispiel zu sehen.

Bei Feldern benötigt `GetValue` nur einen Parameter – nämlich den, der auf das Objekt zeigt, dessen Feldwert Sie ermitteln möchten.

Erstellung benutzerdefinierter Attribute und deren Erkennen zur Laufzeit

Neben der manuellen Serialisierung von Daten ist das Erkennen und Reagieren auf benutzerdefinierte Attribute zur Laufzeit die wohl häufigste Anwendung von Reflection-Techniken. Attribute dienen, wie eingangs erwähnt, zur Kennzeichnung von Klassen oder Klassenelementen; Attributklassen erfüllen in der Regel aber keine weitere wirkliche Funktionalität, da nur in seltenen Fällen ihr Klassencode ausgeführt wird.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 22\Reflection02
```

Innerhalb der Codedatei *mdlMain.vb* finden Sie die Definition einer `Attribute`-Klasse, die folgendermaßen ausschaut:

```
'Benutzerdefiniertes Attribut erstellen
<AttributeUsage(AttributeTargets.All)>
Public Class MeinAttribute
    Inherits Attribute

    Private myName As String

    Public Sub New(ByVal name As String)
        myName = name
    End Sub 'New

    Public ReadOnly Property Name() As String
        Get
            Return myName
        End Get
    End Property
End Class
```

Zwei Sachen fallen hier auf: Zunächst ist der Klassenname nicht wirklich deutsch (*Attribut* wird im Deutschen bekanntlich nicht mit »e« am Ende geschrieben). Dies ist aber hier beabsichtigt. Sie sollten darauf achten, wenn Sie selbst `Attribute`-Klassen entwerfen, dass die Klassen grundsätzlich auf … `Attribute` enden. Der Entwickler, der das Attribut anschließend verwendet, kann in diesem Fall den verkürzten Namen verwenden. Anstelle von `MeinAttribute` reichte also die Angabe von `Mein`.

Die zweite Auffälligkeit: Eine `Attribute`-Klasse kann für den Gebrauch von nur bestimmten Elementen einer Klasse reglementiert werden, und dazu dient `AttributeUsageAttribute`. Sie bestimmen, wie im oben gezeigten Beispielcode, mit der `AttributeTargets`-Enum, für welche Elemente einer Klasse Ihre `Attribute`-Klasse zutreffend ist.

Wichtig ist auch, dass Sie eine benutzerdefinierte `Attribute`-Klasse mit `Inherits` von der `Attribute`-Basisklasse ableiten, damit die Reflection-Funktionen diese später überhaupt als Attribute ausmachen können.

Für das erweiterte Beispielprogramm kommt ebenfalls wieder die bekannte `Adresse`-Klasse zum Einsatz; allerdings verfügt sie an einigen Stellen über `Attribute`-Kennzeichnungen, und wir haben die neue `MeinAttribute`-Klasse dafür verwendet. Wie Sie im oben gezeigten Codeausschnitt sehen können, übernimmt die `MeinAttribute`-Klasse einen Parameter, der in diesem Beispiel nur informative Zwecke hat. In der `Adresse`-Klasse nutzen wir den `String`-Parameter, um darüber zu informieren, in welchem Kontext wir das Attribut eingesetzt haben. Die mit dem benutzerdefinierten Attribut versehene `Adresse`-Klasse sieht wie folgt dargestellt aus (die `Attribute`-Verwendungen sind fett markiert).

HINWEIS Sie sehen anhand dieses Beispiels, dass der Namenszusatz `Attribute` weggelassen werden kann, wenn die Attributklasse wie vorhin bereits erwähnt korrekt auf `Attribute` endet.

```
<Serializable(), Mein("Über der Klasse")>
Public Class Adresse

    Private myErfasstAm As DateTime
    Private myErfasstVon As String

    <Mein("Über dem Konstruktor")>
    Sub New(ByVal Vorname As String, ByVal Name As String,
            ByVal Straße As String, ByVal PLZOrt As String)
        'Konstruktor legt alle Member-Daten ein
        Me.Name = Name
        Me.Vorname = Vorname
        Me.Straße = Straße
        Me.PLZOrt = PLZOrt
        myErfasstAm = DateTime.Now
        myErfasstVon = Environment.UserName
    End Sub

    <Mein("Über einer Eigenschaft")>
    Public Property Name() As String
    Public Property Vorname() As String
    Public Property Straße() As String
    Public Property PLZOrt() As String
    Public Property BefreundetMit() As List(Of Adresse)

    Public ReadOnly Property ErfasstAm() As DateTime
        Get
            Return myErfasstAm
        End Get
    End Property

    Public ReadOnly Property ErfasstVon() As String
        Get
            Return myErfasstVon
        End Get
    End Property
```

```
    <Mein("Über einer Methode")>
    Public Overrides Function ToString() As String
        Dim locTemp As String
        locTemp = Name + ", " + Vorname + ", " + Straße + ", " + PLZOrt + vbNewLine
        locTemp += "--- Befreundet mit: ---" + vbNewLine
        For Each locAdr As Adresse In BefreundetMit
            locTemp += "   * " + locAdr.ToShortString() + vbNewLine
        Next
        locTemp += vbNewLine
        Return locTemp
    End Function

    Public Function ToShortString() As String
        Return Name + ", " + Vorname
    End Function
End Class
```

Ziel von Reflection ist es nun, die Attribute zur Laufzeit zu erkennen. Schauen wir uns zunächst das Ergebnis vorweg an, damit das eigentliche Auswertungsprogramm, das die Attribute findet, anschließend leichter zu verstehen ist. Wenn Sie das Programm starten, produziert es die folgende Bildschirmausgabe:

```
Attribute der Klasse:Reflection02.Adresse
Standardattribute:
    *AutoLayout, AnsiClass, Class, Public, Serializable

Benutzerattribute:
    * System.SerializableAttribute
    * Reflection02.MeinAttribute

Member-Liste:
    *get Name, Method
     --------
         * System.Diagnostics.DebuggerNonUserCodeAttribute
          TypeId: System.Diagnostics.DebuggerNonUserCodeAttribute
    *set Name, Method
     --------
         * System.Diagnostics.DebuggerNonUserCodeAttribute
          TypeId: System.Diagnostics.DebuggerNonUserCodeAttribute
    *get Vorname, Method
     -----------
         * System.Diagnostics.DebuggerNonUserCodeAttribute
          TypeId: System.Diagnostics.DebuggerNonUserCodeAttribute
    *set Vorname, Method
     -----------
         * System.Diagnostics.DebuggerNonUserCodeAttribute
          TypeId: System.Diagnostics.DebuggerNonUserCodeAttribute
    *get Straße, Method
     ----------
         * System.Diagnostics.DebuggerNonUserCodeAttribute
          TypeId: System.Diagnostics.DebuggerNonUserCodeAttribute
    *set Straße, Method
     ----------
         * System.Diagnostics.DebuggerNonUserCodeAttribute
          TypeId: System.Diagnostics.DebuggerNonUserCodeAttribute
    *get PLZOrt, Method
     ----------
         * System.Diagnostics.DebuggerNonUserCodeAttribute
          TypeId: System.Diagnostics.DebuggerNonUserCodeAttribute
    *set_PLZOrt, Method
```

```
 ----------
     * System.Diagnostics.DebuggerNonUserCodeAttribute
      TypeId: System.Diagnostics.DebuggerNonUserCodeAttribute
*get BefreundetMit, Method
 -----------------
     * System.Diagnostics.DebuggerNonUserCodeAttribute
      TypeId: System.Diagnostics.DebuggerNonUserCodeAttribute
*set BefreundetMit, Method
 -----------------
     * System.Diagnostics.DebuggerNonUserCodeAttribute
      TypeId: System.Diagnostics.DebuggerNonUserCodeAttribute
*get ErfasstAm, Method
*get ErfasstVon, Method
*ToString, Method
 --------
     * Reflection02.MeinAttribute
      Name: Über einer Methode
      TypeId: Reflection02.MeinAttribute
*ToShortString, Method
*Equals, Method
 ------
     * System.Runtime.TargetedPatchingOptOutAttribute
      Reason: Performance critical to inline across NGen image boundaries
      TypeId: System.Runtime.TargetedPatchingOptOutAttribute
*GetHashCode, Method
 -----------
     * System.Runtime.TargetedPatchingOptOutAttribute
      Reason: Performance critical to inline across NGen image boundaries
      TypeId: System.Runtime.TargetedPatchingOptOutAttribute
*GetType, Method
 -------
     * System.Security.SecuritySafeCriticalAttribute
      TypeId: System.Security.SecuritySafeCriticalAttribute
*.ctor, Constructor
 -----
     * Reflection02.MeinAttribute
      Name: Über dem Konstruktor
      TypeId: Reflection02.MeinAttribute
*Name, Property
 ----
     * Reflection02.MeinAttribute
      Name: Über einer Eigenschaft
      TypeId: Reflection02.MeinAttribute
 Wert: Adriana
*Vorname, Property
 Wert: Beck
*Straße, Property
 Wert: Buchstraße 223
*PLZOrt, Property
 Wert: 32154 Autorhausen
*BefreundetMit, Property
 Wert: System.Collections.Generic.List`1[Reflection02.Adresse]
*ErfasstAm, Property
 Wert: 16.06.2010 23:04:33
*ErfasstVon, Property
 Wert: klauslo
```

Die Passagen, bei denen sowohl das Attribut selbst als auch sein jeweils gültiger Parameter erkannt wurden, sind im Bildschirmauszug fett markiert.

Ermitteln von benutzerdefinierten Attributen zur Laufzeit

Um benutzerdefinierte Attribute zu ermitteln, gibt es die Funktion `GetCustomAttributes`. Diese Funktion ist sowohl auf einen Typ (die Klasse oder Struktur selbst) als auch auf einzelne Member anwendbar. Als Parameter übernimmt sie entweder einen booleschen Wert, der entweder bestimmt, ob in der Hierarchieliste des Objekts vorhandene Typen ebenfalls auf Attribute untersucht werden sollen (`True`), oder aber den Typ des Attributes, nach dem gezielt gesucht werden soll, und zusätzlich den booleschen Wert für das Durchsuchen der Hierarchieliste festlegt.

Da die Parameter einer Attributklasse in der Regel als Eigenschaften abrufbar sind, können Sie, nachdem Sie das Attribut ermittelt haben, mit `GetType` seine Typinstanz abrufen, anschließend seine Eigenschaften mit `GetProperties` auflisten und schließlich mit `GetValue` den eigentlichen Wert einer Attributeigenschaft auslesen.

Das modifizierte Beispielprogramm zeigt, wie es geht. Es liest am Anfang ein mögliches, benutzerdefiniertes Attribut aus, das für die gesamte Klasse gilt, und untersucht anschließend jedes einzelnen Klassenmember auf Attribute:

```
Module mdlMain

    Sub Main()

        Dim locAdresse As New Adresse("Beck", "Adriana", "Buchstraße 223", "32154 Autorhausen") With
            {.BefreundetMit = New List(Of Adresse) From
                {New Adresse("Klaus", "Löffelmann", "One Microsoftway", "99999 Redmond")}}

        PrintObjectInfo(locAdresse)
        Console.ReadLine()

    End Sub

    Sub PrintObjectInfo(ByVal [Object] As Object)
        'Den Objekttypen ermitteln, um auf die Objektinhalte zuzugreifen
        Dim locTypeInstanz As Type = [Object].GetType

        'Die Nicht Benutzerdefinierten Attribute ausgeben:
        Console.WriteLine("Attribute der Klasse:" + locTypeInstanz.FullName)
        Console.WriteLine("Standardattribute:")
        Console.WriteLine("    *" + locTypeInstanz.Attributes.ToString())
        Console.WriteLine()

        'Benutzerdefinierte Attribute der Klasse ermitteln
        '(Es können auf diese Weise *nur* benutzerdefinierte Attribute ermittelt werden)
        Console.WriteLine("Benutzerattribute:")
        For Each locAttribute As Attribute In locTypeInstanz.GetCustomAttributes(True)
            Console.WriteLine("    * " + locAttribute.ToString())
        Next
        Console.WriteLine()

        'Members und deren mögliche Attribute ermitteln
        Dim locMembers() As MemberInfo
        locMembers = locTypeInstanz.GetMembers()
        Console.WriteLine("Member-Liste:")
        For Each locMember As MemberInfo In locMembers
            Console.WriteLine("    *" + locMember.Name + ", " _
                              + locMember.MemberType.ToString)
            If locMember.GetCustomAttributes(True).Length > 0 Then
                Console.WriteLine("     " + New String("-"c, locMember.Name.Length))
```

```
                For Each locAttribute As Attribute In locMember.GetCustomAttributes(False)
                    Console.WriteLine("         * " + locAttribute.ToString())
                    For Each locPropertyInfo As PropertyInfo In locAttribute.GetType.GetProperties
                        Console.Write("          " + locPropertyInfo.Name)
                        Console.WriteLine(": " + locPropertyInfo.GetValue(
                                locAttribute, Nothing).ToString)
                    Next
                Next
            End If

            If locMember.MemberType = MemberTypes.Property Then
                Dim locPropertyInfo As PropertyInfo = CType(locMember, PropertyInfo)
                Console.WriteLine("     Wert: " + locPropertyInfo.GetValue([Object], Nothing).ToString)
            End If
        Next
    End Sub

End Module
```

HINWEIS Beachten Sie, dass Sie mit `GetCustomAttributes` ausschließlich benutzerdefinierte Attribute auslesen können. Möchten Sie wissen, ob eine Klasse beispielsweise serialisierbar ist, können Sie entweder mit einer der `IsXxx`-Funktionen ihres `Type`-Objekts (`IsSerializable` beispielsweise) oder mit der `Attributes`-Eigenschaft an diese Informationen gelangen.

Teil D

Entwicklungsvereinfachungen in Visual Basic 2010

In diesem Teil:

My als Abkürzung zu Framework-Funktionen verwenden 713
Das Anwendungsframework 739

Kapitel 23

My als Abkürzung zu Framework-Funktionen verwenden

In diesem Kapitel:

Die Visual Basic 2010-Vereinfachungen am Beispiel DotNetCopy 715
Die prinzipielle Funktionsweise von DotNetCopy 721
Der My-Namespace 722
Formulare ohne Instanziierung aufrufen 724
Auslesen der Befehlszeilenargumente mit My.Application.CommandLineArgs 726
Gezieltes Zugreifen auf Ressourcen mit My.Resources 727
Internationalisieren von Anwendungen mithilfe von Ressourcendateien und dem My-Namespace 730
Vereinfachtes Durchführen von Dateioperationen mit My.Computer.FileSystem 733
Verwenden von Anwendungseinstellungen mit My.Settings 736

Es gibt exklusiv in Visual Basic 2010 einige Hilfsmittel (man könnte sie pessimistisch betrachtet auch als »Eigentümlichkeiten« bezeichnen), die auf den ersten Blick das OOP-Konzept von .NET Framework zu umgehen scheinen. Da scheint man direkt auf Formulare zugreifen zu können, ohne sie instanziieren zu müssen, da gibt es Funktionen wie in Visual Basic 6.0 nur für die .NET-Version von Visual Basic, nach denen man in C# vergeblich sucht, und da verhält sich der Compiler an einigen Stellen, als würde er von Geisterhand von jemand anderem programmiert werden. Das alles sind Dinge, bei denen OOP-Puristen der alten Visual Basic 6.0-Riege vorwerfen, man hätte diese Hilfsmittel, die es erstmalig in Visual Basic 2005 und .NET 2.0 gab, eigentlich nicht implementieren dürfen, weil es Visual Basic 2005 wieder zurück in den Pool der »Baby-Programmiersprachen« wirft, und – Zitat – »man dürfe die Blödheit oder Sturheit der prozeduralen Fraktion nicht noch unterstützen«.

Ich gebe zu, ich sah mich eine Weile viel mehr in der letzteren Fraktion der OOP-Puristen als in der alten prozeduralen Visual Basic 6.0-Riege. Und auch ich wollte nicht, dass mein schönes neues jetzt endlich auch erwachsenes .NET-Visual Basic durch »nachgemachte« Visual Basic 6.0-Features aufgeweicht und der Ruf »meines« OOP-fähigen Visual Basic 2005 dadurch vielleicht wieder angeknackst wird – Visual Basic hatte (und hat!) es schließlich in bestimmten Gemeinden ohnehin schwer genug, sich als professionelle Programmiersprache zu etablieren.

Doch wissen Sie was: Diese ganze Diskussion ist eigentlich völlig überflüssig, denn: Visual Basic hat all das, was zum OOP-Programmieren benötigt wird. Und ich persönlich, der in C# und Visual Basic gleichermaßen gut zurechtkommt, programmiere in VB lieber, weil meiner Meinung nach Visual Basic-Editor und -Hintergrundcompiler ein schnelleres und effizienteres Arbeiten als in Verbindung mit den C#-Äquivalenten gestatten. Wer Funktionalitäten, die in diesem Buchteil besprochen werden, wie beispielsweise das Anwendungsframework oder den `My`-Namespace, nicht nutzen will – niemand zwingt ihn dazu! Soll er sich die Arbeit halt schwerer machen, als sie sein müsste – dann braucht derjenige eben länger zum Ziel.

Ein viel genanntes Hauptargument, dass für die vereinfachte Programmierung in Visual Basic Komponenten verwendet werden, die eigentlich nicht zu .NET Framework gehören, ist – mit Verlaub gesagt – ziemlicher Unsinn. Die Datei *VisualBasic.dll* ***ist*** Bestandteil von .NET Framework, und alle Vereinfachungen, mit denen wir vielleicht auch die »alte« Visual Basic 6.0-Riege überzeugen könnten, bei »uns mitzumachen«, sind dort implementiert. Und wer sagt, er will auf diese Vereinfachungen verzichten, weil er Gerüchten glaubt, die sagen, dass diese Visual Basic-eigenen Dinge in Zukunft aus .NET Framework verbannt werden, der darf streng genommen gar nicht mehr in Visual Basic programmieren. Denn, und das ist wichtig: Ohne die Datei *VisualBasic.dll* läuft überhaupt kein Visual Basic-Programm, unabhängig davon, welchen Ansatz (einfach oder .NET-puristisch) man verfolgt oder welcher Komponenten aus .NET Framework man sich bedient. Und wer weiß, dass die Datei *VisualBasic.dll* erstens ohnehin genauso zentraler Bestandteil von .NET Framework ist wie alle anderen .NET-Assemblys und sich zweitens diese Vereinfachungen wie Anwendungsframework und `My`-Namespace auch dort befinden, der soll doch mal einen Blick in die folgenden Seiten riskieren: Viele Probleme lassen sich damit wirklich einfach und schnell lösen – ich kann es am Beispiel beweisen!

WICHTIG In der Tat gibt es eine Assembly, in der der Name *VisualBasic* vorkommt, und zwar die `VisualBasic.Compatibility`-Assembly. Sie ist von vornherein für .NET Framework 1.0 und 1.1 ausschließlich für 32-Bit-Plattformen konzipiert worden und enthält ausschließlich Hilfsfunktionen für die Abwärtskompatibilität zu Visual Basic 6.0. Aus diesem Grund ist jede ihrer Methoden auch mit dem `Obsolete`-Attribut gekennzeichnet, und Sie sollten wirklich (auch für 32-Bit-Anwendungen) davon absehen, diese DLL bzw. ihre Funktionen zu verwenden.

Davon abgesehen kann es auch bei den Visual Basic 6.0-Stil-Funktionen, so möchte ich sie jetzt mal nennen, in Zukunft dazu kommen, dass sie aus technischen Gründen nicht mehr so funktionieren, wie sie eigentlich sollten. Ein Beispiel mag die `FileOpen`-Methode sein: Dateihandles, die hier verwendet werden, müssen beispielsweise assemblyübergreifend funktionieren und abgestimmt werden – da kann es beispielsweise durch ein sehr aggressives Inlineverhalten des 64-Bit-JITters schon mal dazu kommen, dass nicht sauber bestimmt werden kann, aus welcher Assembly ein Aufruf kam, und die Zuweisung der Dateihandles kann nicht zuverlässig garantiert werden. Sie müssen also damit rechnen, dass diese Methoden in der Zukunft nicht mehr verwendet werden können, und Sie sollten von ihrer Verwendung ebenfalls absehen.

Die Visual Basic 2010-Vereinfachungen am Beispiel DotNetCopy

Jeder, der mit seinem Computer regelmäßig viele neue und wichtige Dateien generiert – und zu dieser Riege gehören wir Softwareentwickler ja zwangsläufig wohl alle –, weiß, wie wichtig Datensicherungen sind. Aber getreu dem Motto »Der Schuster hat die schlechtesten Schuhe« gibt es hier und da den einen oder anderen, der dieses Gebiet ein wenig vernachlässigt – ist es nicht so? Der Autor dieser Zeilen will sich gar nicht davon freisprechen. Ich persönlich kopiere meine Daten jeden Abend auf ein externes Laufwerk, und das mithilfe einer simplen Stapeldatei, die in der Eingabeaufforderung von Windows 7 läuft. Eine regelrechte differenzierende Bandsicherung mit einem Band für jeden Wochentag empfehle ich lediglich meinen Kunden und wäre für mich beispielsweise auch momentan schlecht durchführbar, da über 60% dieses Buchs beispielsweise nicht zuhause in unserer Firma, sondern rund 9000 Kilometer weiter westlich in einem wunderschönen Appartment im Stadtteil Belltown von Seattle entstanden sind. Für die Datensicherung war das folgende Beispiel eine echte Hilfe, denn:

Es wäre in Abwandlung zu meiner Stapelkopierdatei schon ein erheblicher Fortschritt, gäbe es ein Programm, das die Dateien bestimmter Verzeichnisse nicht nur kopieren, sondern auch beliebige Backuphistorien der Dateien erstellen würde, die vor einem erneuten Sichern einer aktualisierten Datei versionsmäßig »nach oben« rutschen würden, wie etwa in Abbildung 23.3 (Seite 717) zu sehen.

Das Beispiel zu diesem Buchteil kann genau das, und es kann noch viel, viel mehr: Sie können DotNetCopy so einstellen, dass es nur Dateien kopiert, die im Zielverzeichnis entweder nicht vorhanden oder älter sind als die Dateien, die es zu sichern gilt.

TIPP Ab Windows Vista haben Sie die Möglichkeit, auch lokal – also in einer Nicht-Serverumgebung – auf so genannte Schattenkopien zurückzugreifen. Auf dem Bootlaufwerk (C: in der Regel) ist dies standardmäßig eingestellt; wenn Sie es für weitere Laufwerke verwenden wollen, regeln Sie das über den folgenden Dialog, das Sie über die *Systemsteuerung* und *System/Erweiterte Systemeinstellungen* und die Registerkarte *Computerschutz* erreichen (Abbildung 23.1).

Abbildung 23.1 Ab Windows Vista können Sie einen Teil Ihrer Festplatte für Schattenkopien einrichten – Sie kommen dann über die Eigenschaften einer Datei ...

Abbildung 23.2 **...** an die Registerkarte *Vorgängerversionen*, über die Sie die unterschiedlichen Versionsstände Ihrer Dateien aufrufen können. Auf externen (Backup-)Platten funktioniert das leider nicht oder nicht zuverlässig – hier ist DotNetCopy eine gute Alternative!

Und im Bedarfsfall werden die im Zielverzeichnis vorhandenen Dateien nicht einfach ersetzt, sodass die alte Version auf Nimmerwiedersehen verschwindet. Vielmehr können Sie bestimmen, wie viele Backupstufen der älteren Dateien automatisch vorgehalten werden sollen – Abbildung 23.3 zeigt in Form einer Dateien-Backuphistorie, wie das später auf dem Backuplaufwerk aussieht.

BEGLEITDATEIEN Sie finden dieses Projekt übrigens unter dem Namen *DotNetCopy.sln* im Verzeichnis

```
...\VB 2010 Entwicklerbuch\Kapitel 23\DotNetCopy
```

Abbildung 23.3 DotNetCopy kopiert Dateien nur, wenn sie neuer sind als die im Zielverzeichnis vorhandenen Versionen oder dort noch nicht existieren. Bei Dateien, die schon vorhanden waren, legt das Programm automatisch Backuphistorien der betroffenen Dateien an.

Der Dialog *Optionen* von *DotNetCopy* erlaubt es, das Kopierverhalten genau zu kontrollieren (Abbildung 23.4).

Abbildung 23.4 DotNetCopy erlaubt es, bestimmte Kopiereinstellungen im *Optionen*-Dialog zu konfigurieren

Sie erstellen mit DotNetCopy anschließend eine Kopierliste, die das Programm darüber informiert, welches Quellverzeichnis wohin kopiert (also gesichert) werden soll. Kopierlisten, die Sie auf diese Art und Weise erstellen, lassen sich sogar speichern und laden, was die Anwendung des Programms recht flexibel macht (Abbildung 23.5).

Abbildung 23.5 Erstellen Sie mit DotNetCopy Kopierlisten, die beliebig viele Quell- und Zielpfade enthalten und geladen sowie gespeichert werden können

Und wenn Sie eine Kopierliste erstellt und für die spätere Wiederverwendung gespeichert haben, drücken Sie einfach aufs Knöpfchen, und der Kopiervorgang kann beginnen, wie in Abbildung 23.6 zu sehen.

Abbildung 23.6 DotNetCopy nach erfolgreichem Backuplauf

DotNetCopy mit /Autostart und /Silent-Optionen

Damit dieses Tool aber auch für Administratoren interessant ist, die es vielleicht in eigene Stapeldateien einbinden möchten, lässt es sich auch im Autostart- sowie im Silent-Modus starten.

- Der Autostart-Modus erlaubt es, eine Kopierlistendatei beim Aufruf des Programms anzugeben. Diese Kopierlistendatei wird dann anschließend geladen und der Backupvorgang beginnt unmittelbar. Der Aufruf von DotNetCopy im Kopierlistenmodus erfolgt dazu an der Eingabeaufforderung oder über Ausführen im Start-Menü. Sie geben zusätzlich zum Programmnamen *DotNetCopy* die Option */autostart:kopierlistendatei.ols* an, also beispielsweise

```
Dotnetcopy /autostart:backup.ols
```

- Zusätzlich können Sie das Programm so starten, dass es ohne sichtbare Benutzeroberfläche läuft. In diesem Fall verwenden Sie die Option */silent*, die Sie durchaus mit der zuvor beschriebenen Option */autostart* kombinieren können. Um also die obige als Beispiel gezeigte Kopierliste automatisch anzeigen zu lassen, können Sie folgende Befehlszeile verwenden:

```
Dotnetcopy /autostart:backup.ols /silent
```

Nachdem Sie die Funktionsweise von DotNetCopy kennen gelernt haben, geht es als Nächstes darum, zu erfahren, was die Kapitel dieses Buchs mit der ganzen Geschichte zu tun haben.

Und falls Sie denken, Sie müssten zum Verständnis des Programms tief in die Abgründe von .NET Framework hinabsteigen – weit gefehlt. DotNetCopy benötigt dank der Visual Basic-eigenen Vereinfachungen überraschend wenig Code.

Und selbst Visual Basic 6.0-Umsteiger werden lediglich mit zwei neuen Klassen aus .NET Framework konfrontiert – fast die gesamte andere Funktionalität basiert komplett auf Merkmalen, die durch den `My`-Namespace und das Anwendungsframework zur Verfügung gestellt werden.

Exkurs: Die Klassen FileInfo- und DirectoryInfo zur Pflege von Verzeichnissen und Dateien

Um Zugriff auf Dateien und Verzeichnisse der Festplatte oder eines anderen Datenträgers Ihres Computers zu nehmen, stellt Ihnen .NET Framework die Klassen `FileInfo`- und `DirectoryInfo` zur Verfügung.

Die `FileInfo`-Klasse dient zur »Bearbeitung« von Dateien im Sinne des Windows-Explorers; `DirectoryInfo` stellt im Gegensatz dazu die äquivalente Funktionalität für Verzeichnisse zur Verfügung: Stellen Sie sich beide Klassen am besten wie einen Windows-Datei-Explorer vor, der nur eben keine sichtbare Benutzeroberfläche hat: Mit der `FileInfo`-Klasse haben Sie beispielsweise die Möglichkeit, Attribute von Dateien abzurufen, Dateien umzubenennen, sie an einen anderen Ort zu verschieben, auf deren Vorhandensein zu überprüfen und Ähnliches. Mit `DirectoryInfo` arbeiten Sie prinzipiell genau so – nur eben mit Verzeichnissen.

Das Schwierige bei Dateioperationen ist in der Regel gar nicht so sehr das Aufrufen bestimmter Funktionen, wenn Sie erst einmal den exakten Pfad und Dateinamen einer Datei kennen. Dabei geht es um das richtige Zusammenbauen des Pfades, das Extrahieren des eigentlichen Dateinamens aus der Zeichenkettenkombination von Pfad und Dateinamen oder auch das Ermitteln der Dateiendung einer kompletten Pfad-/Dateinamenkombination. Und auch dabei helfen Ihnen die Klassen `FileInfo` und `DirectoryInfo`. Beim Erstellen neuer Unterordner in mehreren Verzeichnisebenen brauchen Sie sich obendrein auch nicht darum zu kümmern, ob Ordner in Teilebenen schon existieren – wenn Sie beispielsweise mit der `DirectoryInfo`-Klasse ein Verzeichnis namens *C:\Ordner1\UnterOrdner\EigentlicherOrdner* anlegen wollen, werden im Bedarfsfall alle benötigten Ordner (*Ordner1*, *UnterOrdner*) gleich mit angelegt. Die folgenden Codezeilen geben einige Beispiele für die Anwendung der `FileInfo`- bzw. `DirectoryInfo`-Klasse:

Beispiel DirectoryInfo-Objekt:

```
 'Neues DirectoryInfo-Objekt aus Pfad-Zeichenkette erstellen:
Dim einDirInfo As New DirectoryInfo("C:\Ordner1\Unterordner")
'Existiert das Verzeichnis?
If Not einDirInfo.Exists Then
    Debug.Print("Ordner existiert nicht, wird erstellt:")
    'Verzeichnis mit allen benötigten Unterverzeichnissen erstellen
    einDirInfo.Create()
End If
```

Beispiel FileInfo-Objekt:

```
'Neues FileInfo-Objekt aus Pfad-/Dateinamenzeichenkette erstellen
Dim einFileInfo As New FileInfo("c:\Ordner1\Unterordner\textfile.txt")
'Jedes FileInfo-Objekt enthält auch ein DirectoryInfo-Objekt, das
'über die Directory-Eigenschaft abrufbar ist:
If Not einFileInfo.Directory.Exists Then
'Pfad zur Datei im Bedarfsfall erstellen
    einFileInfo.Directory.Create()
End If
```

▶

```
'Auf Vorhandensein der Datei prüfen:
If Not einFileInfo.Exists Then
'Eine simple Textdatei erstellen.
    My.Computer.FileSystem.WriteAllText(einFileInfo.FullName, "Dateiinhalt", False)
End If
Debug.Print("Die Erweiterung der Datei lautet: " & einFileInfo.Extension)
Debug.Print("Der reine Dateiname lautet: " & einFileInfo.Name)
Debug.Print("Voller Pfad und Dateiname zur Datei lauten: " & einFileInfo.FullName)
Debug.Print("Die Datei hat folgende Attribute: " & einFileInfo.Attributes)
Debug.Print("Die Datei wird jetzt ins Hauptverzeichnis kopiert!")
'Wichtig: Pfad und neuer Dateiname müssen als Ziel beide angegeben werden!
einFileInfo.CopyTo("C:\" & einFileInfo.Name)
Debug.Print("Die Ausgangsdatei wird gelöscht!")
einFileInfo.Delete()
```

Dieser Code erzeugt folgende Ausgabe:

```
Die Erweiterung der Datei lautet: .txt
Der reine Dateiname lautet: textfile.txt
Voller Pfad und Dateiname zur Datei lauten: c:\Ordner1\Unterordner\textfile.txt
Die Datei hat folgende Attribute: 32
Die Datei wird jetzt ins Hauptverzeichnis kopiert!
Die Ausgangsdatei wird gelöscht!
```

Die prinzipielle Funktionsweise von DotNetCopy

Wie gesagt: DotNetCopy dient dazu, die beiden Besonderheiten von Visual Basic in Vergleich zu den anderen .NET Framework-Programmiersprachen zu demonstrieren. Dazu gehören die Funktionalitäten, die durch das so genannte Anwendungsframework zur Verfügung gestellt werden, sowie der `My`-Namespace, der die Aufgabe hat, dass sich bestimmte thematisch kategorisierte Probleme, die Ihnen bei den täglichen Entwicklungsaufgaben immer wieder begegnen, einfacher lösen lassen.

Beim Anwendungsframework, dessen Funktionalität und Handhabung das folgende Kapitel beschreibt, bedient sich DotNetCopy dreierlei Dinge: der Zurverfügungstellung der Visual Styles von Windows XP und späterer Betriebssysteme, der Anwendungsereignisse und einer Funktion, die verhindert, dass eine zweite Instanz von DotNetCopy gestartet werden kann, wenn eine Instanz von DotNetCopy bereits aktiv ist.[1]

Da DotNetCopy die Option */silent* kennt, muss es einen Mechanismus geben, der quasi noch vor dem Anzeigen des ersten Formulars greift und der eben in diesem Modus verhindert, dass das erste Formular angezeigt wird. Das Anwendungsframework stellt zu diesem Zweck eine Reihe von Ereignissen zur Verfügung, und unter anderem auch ein Ereignis, das ausgelöst wird, nachdem das Programm gestartet ist. Ob es zur Anzeige des Hauptformulars kommt oder nicht, wird in dieser Ereignisbehandlungsroutine namens `MyApplication_Startup` geregelt, die sich in der Codedatei *ApplicationEvents.vb* des Projekts befindet.

[1] Ob dieser letzte Punkt für das richtige Funktionieren von DotNetCopy tatsächlich erforderlich ist, darüber lässt sich sicherlich streiten – vielleicht möchte auch der ein oder andere, dass DotNetCopy in mehreren Instanzen gleichzeitig laufen kann. Für die Demonstration des Anwendungsframeworks ist es allemal geeignet.

Im Silent-Modus wird anschließend zwar das Hauptformular, das auch die komplette Funktionalität für den Backupvorgang in *frmMain.vb* enthält, instanziiert, aber nicht dargestellt. Für die korrekte Funktionsweise ist das jedoch kein Nachteil – selbst der Code für die Formularaktualisierungen (Fortschrittsanzeige und Protokollausgabe aktualisieren, aktuell bearbeitetes Verzeichnis anzeigen etc.) funktioniert, nur dass der Benutzer sie während des Backupvorgangs nicht sieht, da das Formular selbst eben nicht anzeigt wird.

Die weitere Funktionsweise ist daher auch in allen Modi gleich. Nachdem eine Kopierliste geladen wurde – die Funktion `LoadCopyEntryList` ist dafür zuständig, und sie wird entweder durch die Funktion `HandleAutoStart` (ihrerseits durch `MyApplication_Startup` aufgerufen) oder durch das Auswählen des entsprechenden Menüeintrags oder Symbols vom Benutzer getriggert –, startet der Kopiervorgang in der Methode `CopyFiles` (auch entweder automatisch durch `HandleAutoStart` oder durch das Auswählen des entsprechenden Menüeintrags oder Symbols durch den Benutzer).

Und in `CopyFiles` und allen Methoden, die dort aufgerufen werden, kommt das Thema dieses Kapitels intensiv zum Einsatz – Funktionen aus dem `My`-Namespace.

Wenn Sie einen Blick in die Projektdateien werfen, werden Sie sehen, dass das Programm ausführlich kommentiert ist. Zudem werden Sie die wichtigen Ausschnitte in den nächsten Abschnitten und im nächsten Kapitel wieder finden.

Obendrein haben Sie durch DotNetCopy nicht nur ein anschauliches Beispiel, wie einfach sich auch komplexere Anwendungen mit Visual Basic und den »Visual Basic-Vereinfachungen« entwickeln lassen, sondern auch ein Tool, das der eine oder andere von Ihnen vielleicht auch wirklich gebrauchen kann. Bei mir hat DotNetCopy jedenfalls seit seiner Fertigstellung die alten Stapeldateiprozesse abgelöst, und die Backuphistorie von Dateien, die es generiert, habe ich schon mehr als einmal gut gebrauchen können.

Und vielleicht noch eine letzte Anmerkung zum Thema OOP und den Visual Basic-Vereinfachungen: DotNetCopy ist sicherlich kein Paradebeispiel für OOP. Aber es ist brauchbar, stabil und war schnell entwickelt. Das gleiche Tool in der Sprache C#, in der es die Visual Basic-Vereinfachungen nicht gibt, hätte sicherlich mindestens einen Tag mehr Entwicklungsaufwand bedeutet. OOP ist gerade bei größeren Anwendungen sicherlich ein Muss, auch in Visual Basic und trotz der Tatsache, dass es `My` und das Anwendungsframework gibt. Aber OOP verleitet auch dazu, mehr zu machen, als eigentlich gefordert ist. Denken Sie daran, dass Sie für ein Einfamilienhaus auch kein Fundament für einen Wolkenkratzer ausheben würden, und diese Analogie sollten Sie im Hinterkopf behalten, wenn Sie die Klassenplanung für ein größeres Projekt in Angriff nehmen.

Es ist nichts Verwerfliches dabei, die speziellen Visual Basic-Vereinfachungen zu verwenden. Wenn Sie damit Zeit sparen und absehen können, dass Sie sich damit für spätere Erweiterungen den Weg nicht verbauen, nutzen Sie sie. Und lassen Sie die C#- oder Visual Basic-Entwickler, die Ihnen etwas anderes weismachen wollen, ruhig reden …

Der My-Namespace

Der `My`-Namespace wurde in Visual Basic 2005 eingeführt, um den Umgang mit bestimmten Funktionalitäten zu vereinfachen – um sozusagen »Abkürzungen« zu bestimmten Zielen mit Funktionalitäten in .NET Framework zur Verfügung zu stellen, die Sie normalerweise nur auf verschlungenen Pfaden erreichen würden.

Wenn Sie das Beispielszenario aus Kapitel 5 komplett durchexerziert haben, sind Sie auch schon in den Genuss dieser Abkürzungen gekommen – Sie haben dort nämlich gelernt, wie einfach Sie auf selbst speichernde Anwendungseinstellungen durch `My.Settings` zurückgreifen können.

Doch der `My`-Namespace verfügt über noch weit mehr wirklich coole Features. Welche das allerdings genau sind, lässt sich nicht verbindlich sagen, denn die Features, die Ihnen durch `My` zur Verfügung stehen, hängen von dem Projekttyp ab, den Sie gerade bearbeiten.

Generell teilt sich die Funktionalität im `My`-Namespace in die folgenden Kategorien auf:

- **My.Application** stellt Zugriffsmöglichkeiten auf Informationen über die aktuelle Anwendung bereit, wie beispielsweise den Pfad zur ausführbaren Datei, die Programmversion, derzeitige Kulturinformationen oder den Benutzerauthentifizierungsmodus
- **My.Computer** ermöglicht Ihnen den Zugriff auf mehrere untergeordnete Objekte, die Ihnen wiederum das Abrufen von computerbezogenen Informationen gestatten, wie beispielsweise über das verwendete Dateisystem, über verwendete Audio- und Videospezifikationen, angeschlossene Drucker oder generelle I/O-Hardware wie Maus, Tastatur, Speicher, die verwendete Netzwerkumgebung, serielle Schnittstellen und Weiteres
- **My.Forms** gestattet Ihnen das Abrufen von Standardinstanzen aller Formulare einer Windows Forms-Anwendung
- **My.Resources** ermöglicht den einfachen Zugriff auf eingebettete Ressourcen, wie beispielsweise auf Zeichenkettentabellen, Bitmaps und Ähnliches
- **My.Settings** ermöglicht einerseits, auf Anwendungseinstellungen Zugriff zu nehmen, und sorgt andererseits dafür, dass diese Anwendungseinstellungen im Bedarfsfall benutzerabhängig beim Programmstart wiederhergestellt und beim Programmende gesichert werden
- **My.User** ermöglicht das Abrufen von Infos über den zurzeit angemeldeten Benutzer und erlaubt ferner das Implementieren benutzerdefinierter Authentifizierungsmechanismen
- **My.WebServices** stellt eine Eigenschaft für jeden Webdienst zur Verfügung, den das aktuelle Projekt referenziert, und erlaubt so, auf Webdienste Zugriff zu nehmen, ohne eine explizite Proxyklasse für den jeweiligen Webdienst erstellen zu müssen

In einer Konsolenanwendung, die nun einmal keine Formulare verwendet, steht Ihnen natürlich die Kategorie `My.Forms` nicht zur Verfügung. Und so hängt die tatsächliche Verfügbarkeit der `My`-Features durchweg von den Projekttypen ab, mit denen Sie es gerade zu tun haben, wie Tabelle 23.1 zeigt.

My-Objekt	Windows Forms-Anwendung	Klassenbibliothek	Konsolenanwendung	Windows Forms-Steuerelementbibliothek	Windows-Dienst
`My.Application`	Ja	Ja	Ja	Ja	Ja
`My.Computer`	Ja	Ja	Ja	Ja	Ja
`My.Forms`	Ja	Nein	Nein	Ja	Nein
`My.Resources`	Ja	Ja	Ja	Ja	Ja
`My.Settings`	Ja	Ja	Ja	Ja	Ja
`My.User`	Ja	Ja	Ja	Ja	Ja
`My.WebServices`	Ja	Ja	Ja	Ja	Ja

Tabelle 23.1 Verfügbarkeit der Bereiche des `My`-Namespace bei wichtigen Projekttypen

Nun finde ich, es würde an dieser Stelle keinen Sinn machen, die komplette Referenz aller Funktionalitäten jedes einzelnen Bereichs herunterzubeten – im Gegenteil: Schließlich dient `My` eben genau dazu, auch ohne großes Blättern und nur mithilfe von IntelliSense und dynamischer Hilfe schnell an die gewünschte Funktionalität zu gelangen.

Stattdessen sollten Sie gerade von einem Entwicklerbuch erwarten können, dass Ihnen der Autor Sachverhalte im Rahmen einer Softwareentwicklung näher bringt – und das hat er für die Demonstration von `My` auch getan. Die in den folgenden Abschnitten beschriebenen `My`-Funktionalitäten erheben deshalb keinen Anspruch auf Vollständigkeit – sie sollen es auch gar nicht. Ziel ist es vielmehr, zum einen generelle Vorgehensweisen beim Umgang mit dem `My`-Namespace zu vermitteln. Die richtigen Funktionen für Ihre eigenen Erfordernisse zu finden, wird dank IntelliSense und Onlinehilfe sicherlich nicht *das* Problem sein und würde an dieser Stelle nur wertvollen Platz kosten.

Es gibt zum anderen auch einiges Erwähnenswerts zum `My`-Namespace und seinen Funktionalitäten, das Sie nicht in der Onlinehilfe finden, und auch diesen Punkten widmen sich die folgenden Abschnitte. Viele der hier gezeigten Features können Sie sich direkt am Beispiel von DotNetCopy anschauen, und innerhalb der Abschnitte werden Sie auch immer wieder Codeausschnitte aus DotNetCopy finden.

Formulare ohne Instanziierung aufrufen

Die Änderungen an Visual Basic, um die es jetzt geht, haben in der bis dato etablierten Visual Basic-Gemeinde für viel Aufsehen gesorgt, denn: Es geht um die Vereinfachung von Formularaufrufen, und das Hauptargument dabei ist: Damit nähere sich Visual Basic wieder einer »Kindersprache« an, bei der man dem »unmündigen« Visual Basic-Programmierer nicht zumuten könne, die buchstäblichen Basics der OOP zu kennen und anzuwenden.

Worum geht es genau? Auch schon in Visual Basic 6.0 oder in Visual Basic for Applications (VBA) mussten Sie, um mit einem Objekt arbeiten zu können, es zunächst instanziieren. Haben Sie also eine `Collection` verwendet, in der Sie andere Objekte speichern wollten, waren folgende Zeilen notwendig:

```
Dim auflistung As Collection
Set auflistung = New Collection
auflistung.Add 5
```

Es ist klar, dass diese Zeilen nicht funktioniert hätten:

```
Collection.Add 10
Collection.Add 15
Collection.Add 20
```

Bei Visual Basic 6.0-Formularen passiert aber genau das: Hier können Klassenname und Instanz das Gleiche sein. Diese Version funktioniert

```
Dim f As Form1
f.Show
```

genau wie die Verwendung der Klasse `Form1` als Instanzvariable:

```
Form1.Show
```

Das entspricht natürlich nicht den Vorschriften der objektorientierten Programmierung, war aber wahrscheinlich für viele Visual Basic 6.0-Programmierer einfacher zu verstehen, und aus diesem Grund wurde der Basic-Compiler so frisiert, dass eine solche Verwendung einer Formularklasse möglich war.

Schon in Visual Basic 6.0 passierte »unter der Haube« dazu etwas, was nunmehr auch seit Visual Basic 2005 wieder möglich ist.

Der Visual Basic-Compiler sorgt nämlich bei der Kompilierung einer Windows Forms-Anwendung dafür, dass versteckter Quellcode, der im Compiler fest verdrahtet ist, mit in Ihre Anwendung kompiliert wird. Der Visual Basic-Compiler »erfindet« also quasi einen Haufen Quellcode und fügt diesen, unsichtbar für den Entwickler, dem eigentlichen Windows-Projekt hinzu.

So können Sie auch in Visual Basic 2010 wieder folgende Zeile verwenden, um das unter dem Klassennamen `Form2` erstellte Formular beispielsweise als modalen Dialog ins Leben zu rufen:

```
Form2.ShowDialog
```

Doch es sieht hier nur so aus, als würden Sie eine statische Klassenmethode direkt verwenden, denn in Wirklichkeit ist `Form2` eine Eigenschaft, die eine *Instanz* vom Typ `Form2` zurückliefert. Denn was hier eigentlich passiert, ist, dass der Compiler diese Zeile nur vervollständigt und er im Grunde genommen folgenden Quellcode für die Generierung der eigentlichen ausführbaren Datei kompiliert:

```
My.Forms.Form2.ShowDialog
```

Der Compiler baut intern aus diesem Konstrukt `MyProject.Forms`. Die Klasse `MyProject` ist dabei allerdings eine Klasse, die Sie unter diesem Namen nicht erreichen können, weil sie mit entsprechenden Attributen so gekennzeichnet ist, dass der Compiler sie nicht »sieht«. Sie ist aber dennoch vorhanden. Quasi versteckt legt der Compiler also für jedes Formular, das Sie Ihrem Projekt hinzufügen, eine Eigenschaft vom Typ des jeweiligen Formulars in der (ebenfalls unsichtbaren) `MyForms`-Klasse an, die eine eingebettete Klasse der Klasse `MyProject` selbst ist, deren Instanz durch die `Forms`-Eigenschaft der `MyProject`-Klasse abgerufen werden kann. Und erst diese Eigenschaft, die den Namen des Formulars trägt, sorgt dafür, dass die gewünschte Formularklasse vor Gebrauch instanziiert und dann ohne Probleme (bzw. `NullReference`-Ausnahmen) verwendet werden kann.

Zur Veranschaulichung: Die Programmzeile

```
My.Forms.Form2.ShowDialog
```

bedeutet im Grunde genommen:

```
(MyProject-Klasse).(Forms-Eigenschaft der MyProject-Klasse vom Typ MyForms). _
(Form2-Eigenschaft der MyForms-Klasse vom Typ Form2).(ShowDialog-Methode der Form2-Klasse).
```

Auslesen der Befehlszeilenargumente mit My.Application.CommandLineArgs

Anwendungen, die unter Windows laufen, können Sie auf unterschiedliche Weise starten. Die einfachste und bekannteste ist, sie durch einen Doppelklick auf eine Verknüpfung beispielsweise aus dem Startmenü zum Laufen zu bewegen. Sie können jede Windows- oder Befehlszeilenanwendung aber auch direkt durch die Angabe des Namens der ausführbaren Datei aus der Befehlszeile oder durch den *Ausführen*-Befehl des Startmenüs heraus in Gang bekommen. Und viele Anwendungen erlauben es hier, weitere Parameter anzugeben, die das Startverhalten steuern. So starten Sie beispielsweise den Windows-Explorer mit

```
Explorer c:\
```

um zu definieren, dass der Windows-Explorer nicht nur gestartet wird, sondern dass er auch direkt den Inhalt des Stammverzeichnisses von Laufwerk *C:* anzeigt.

Parameter, die Sie Anwendungen auf diese Weise übergeben, nennt man Befehlszeilenargumente, und mithilfe des `My`-Namespace können Sie Befehlszeilenargumente auf einfache Weise ermitteln und auswerten.

Das `My`-Beispiel DotNetCopy arbeitet ebenfalls mit solchen Befehlszeilenargumenten, nämlich um die Anwendung in den Autostart- oder Silent-Modus zu versetzen. Der folgende Code, der sich im `StartUp`-Ereignis des `MyApplication`-Objekts befindet und der ausgelöst wird, sobald die Anwendung startet, demonstriert den Umgang mit Befehlszeilenargumenten.

HINWEIS Sie finden diesen Code in der Datei *ApplicationEvents.vb* des Projekts. Mehr zu Anwendungsereignissen erfahren Sie übrigens im nächsten Kapitel.

```
Partial Friend Class MyApplication

    Private Sub MyApplication_Startup(ByVal sender As Object, ByVal e As _
         Microsoft.VisualBasic.ApplicationServices.StartupEventArgs) Handles Me.Startup

        'Das Verzeichnis für die Protokolldatei beim ersten Mal setzen...
        If String.IsNullOrEmpty(My.Settings.Option_AutoSaveProtocolPath) Then
            My.Settings.Option_AutoSaveProtocolPath = _
                My.Computer.FileSystem.SpecialDirectories.MyDocuments & "\DotNetCopy Protokolle"
            Dim locDi As New DirectoryInfo(My.Settings.Option_AutoSaveProtocolPath)

            'Überprüfen und
            If Not locDi.Exists Then
                'im Bedarfsfall anlegen
                locDi.Create()
            End If

            'Settings speichern
            My.Settings.Save()
        End If

        Dim locFrmMain As New frmMain
```

```
            'Kommandozeile auslesen
            'Sind überhaupt Befehlszeilenargumente vorhanden?
            If My.Application.CommandLineArgs.Count > 0 Then
                'Durch jedes einzelne Befehlszeilenargument durchiterieren
                For Each locString As String In My.Application.CommandLineArgs

                    'Alle unnötigen Leerzeichen entfernen und
                    'Groß-/Kleinschreibung 'Unsensibilisieren'
                    'HINWEIS: Das funktioniert nur in der Windows-Welt;
                    'kommt die Kopierlistendatei von einem Unix-Server, bitte darauf achten,
                    'dass der Dateiname dafür auch komplett in Großbuchstaben gesetzt ist,
                    'da Unix- (und Linux-) Derivate Groß-/Kleinschreibung berücksichtigen!!!
                    locString = locString.ToUpper.Trim

                    If locString = "/SILENT" Then
                        locFrmMain.SilentMode = True
                    End If

                    If locString.StartsWith("/AUTOSTART") Then
                        locFrmMain.AutoStartCopyList = locString.Replace("/AUTOSTART:", "")
                        locFrmMain.AutoStartMode = True
                    End If
                Next
            End If
.
.
.
```

Gezieltes Zugreifen auf Ressourcen mit My.Resources

Ressourcendateien speichern bestimmte Elemente einer Anwendung in einer separaten Datei. Bei solchen Elementen handelt es sich beispielsweise um Bitmaps, Symbole, Zeichenketten oder Ähnliches. Diese Elemente lassen sich mithilfe von `My.Resources` zur Laufzeit aus Ressourcendateien lesen und anschließend im Kontext verwenden.

Der Vorteil dieser Vorgehensweise: Anwendungen können auch im Nachhinein angepasst werden, ohne dem Bearbeiter den eigentlichen Code der Anwendung zur Verfügung zu stellen. Denken Sie dabei nur zum Beispiel an die Lokalisierung einer Anwendung, etwa bei Microsoft Office-Programmen wie Word, in eine andere Sprache. Wären die Texte der Anwendung fest verdrahtet, müsste man den Übersetzern die kompletten Quellen der Anwendung zur Verfügung stellen und die entsprechenden Texte, die dann als Zeichenketten irgendwo im Quellcode eingebettet wären, heraussuchen. Ein weiterer Nachteil wäre, dass es verschiedene Versionen des Quellcodes gäbe, von denen jede einzelne bei Änderungen oder Fehlerbehebungen in der Anwendung bearbeitet werden müsste – ein Aufwand, der bei größeren Projekten nicht zu leisten ist.

Anlegen und Verwalten von Ressourcenelementen

In unserer Beispielanwendung werden die Texte für die Spaltennamen der `ListView`, die die Kopierlisteneinträge beinhaltet, aus einer Ressourcendatei entnommen. Die Ressourcendatei selbst wird ähnlich verwaltet, wie Sie es bei den Anwendungseinstellungen (den *Application Settings*) bereits im 5. Kapitel kennen gelernt haben.

- Rufen Sie, um die Ressourcendatei einer Anwendung zu bearbeiten, einfach die Projekteigenschaften über das Kontextmenü (rechte Maustaste) des Projektmappen-Explorers auf.
- Wählen Sie anschließend die Registerkarte *Ressourcen*.
- Wählen Sie aus der linken Schaltfläche, die Sie öffnen können (Abbildung 23.7), den Typ der Ressource, den Sie einpflegen möchten. Bei Zeichenketten erfassen Sie die Texte wie bei Anwendungseinstellungen in Form von Texttabellen. Bei anderen Ressourcentypen, wie beispielsweise Bitmaps, Symbolen oder Audioclips, verwenden Sie die Schaltfläche *Ressource hinzufügen*, um eine entsprechende Datei in die Ressourcendatei aufzunehmen.

Abbildung 23.7 Bestimmen Sie wie hier den Typ Ressource, den Sie der Ressourcendatei hinzufügen möchten. Bei anderen Typen als Zeichenketten verwenden Sie *Ressource hinzufügen*, um Ressourcenelemente wie Bitmaps, Symbole oder Audioclips in die Ressourcendatei aufzunehmen.

Abrufen von Ressourcen mit My.Resources

Wenn Sie Ressourcen auf die im letzten Abschnitt beschriebene Weise Ihrem Projekt hinzugefügt haben, ist das Abrufen der Ressourcenelemente zur Laufzeit dank *My* ein Einfaches. Im Beispielprojekt DotNetCopy wird das beispielsweise im Ereignishandler `form_Load` des Formulars gemacht; hier werden beispielhaft die Spaltentexte der `ListView` eingerichtet:

```
''' <summary>
''' Wird aufgerufen, wenn das Formular geladen wird, und enthält die
''' Initialisierung der ListView sowie das Anstoßen des Kopiervorgangs
''' (und das zuvor notwendige Laden der Kopierliste), wenn das Programm
''' im Autostart (aber nicht im Silent) -Modus läuft.
''' (Silent-Modus wird in ApplicationEvents.vb gehandelt).
''' </summary>
''' <param name="sender"></param>
''' <param name="e"></param>
''' <remarks></remarks>
Private Sub frmMain_Load(ByVal sender As Object, ByVal e As System.EventArgs) _
        Handles Me.Load
```

```
        'Listview einrichten
        With Me.lvwCopyEntries.Columns
            'Die Texte für die ListView-Spalten aus der Ressource-Datei entnehmen
            .Add(My.Resources.ListViewSourcePathColumnName)
            .Add(My.Resources.ListViewSearchMaskColumnName)
            .Add(My.Resources.ListViewDestinationPathColumnName)
            'Spalten ausrichten
            AlignColumns()
        End With
.
.
.
```

Natürlich könnte man dieses Verfahren auch beispielsweise auf die angezeigten Texte im Backupprotokoll ausweiten. Der Einfachheit halber habe ich die Verwendung von Textressourcen auf diesen Bereich beschränkt.

HINWEIS Im Übrigen machen WinForms-Anwendungen beim Zuweisen von Texten oder Bilddaten ebenfalls implizit von Ressourcendateien Gebrauch. Zwar verwendet der Windows Forms-Designer für den generierten Code nicht die Funktionalität aus dem `My`-Namespace, aber das Holen beispielsweise von Bitmapdateien aus einer Ressource, um dieser ein Symbol etwa einer Symbolleistenschaltfläche hinzuzufügen, funktioniert prinzipiell auf die gleiche Weise. Sie können sich selbst ein Bild davon machen: Wenn Sie im Projektmappen-Explorer die ausgeblendeten Dateien des Projekts mit dem Symbol *Alle Dateien anzeigen* (der ToolTip hilft beim Finden des Symbols) einblenden, können Sie einen Blick in den Code werfen, der zum Aufbau des Formulars führt. Dazu klicken Sie anschließend die Datei *frmMain.Designer.vb* doppelt an, die sich jetzt im Zweig unterhalb der Formulardatei *frmMain.vb* befindet.

```
.
.
.
        '
        'tsbLoadCopyList – Auszug aus der FormsDesigner generierten Datei zum Zuweisen
        'eines Images an ein Symbol in DotNetCopy:
        Me.tsbLoadCopyList.DisplayStyle = System.Windows.Forms.ToolStripItemDisplayStyle.Image
        Me.tsbLoadCopyList.Image = CType(resources.GetObject("tsbLoadCopyList.Image"), _
                            System.Drawing.Image)

        Me.tsbLoadCopyList.ImageTransparentColor = System.Drawing.Color.Magenta
.
.
.
```

Die Zuweisung eines Symbols mithilfe des `My`-Namespaces würde folgendermaßen ausschauen:

```
Me.tsbLoadCopyList.Image = My.Resources.LoadCopyList
```

Voraussetzung dafür wäre, dass eine entsprechende Image-Datei auf die zuvor beschriebene Weise der Ressourcendatei hinzugefügt wurde, etwa wie in Abbildung 23.8 zu sehen.

Abbildung 23.8 Öffnen Sie die Schaltfläche *Ressource hinzufügen* (mit einem Klick auf das Dreiecksymbol neben dem Schaltflächennamen, nicht auf die Schaltfläche selbst klicken!), und wählen Sie aus der Auswahlliste *Datei hinzufügen*, um beispielsweise eine Bitmap in die Ressourcendatei aufzunehmen, die Sie dann zur Laufzeit mit `My.Resource` und dem Laden von Elementen zuweisen können

Internationalisieren von Anwendungen mithilfe von Ressourcendateien und dem My-Namespace

Der Einsatz von Ressourcendateien macht natürlich nur dann Sinn, wenn Sie planen, eine Anwendung tatsächlich auch in einer anderen Sprache zur Verfügung zu stellen. Für diesen Vorgang des Lokalisierens müssen Sie weitere Ressourcendateien erstellen, die auf Basis eines bestimmten Schemas benannt und dem Projekt hinzugefügt werden.

Leider gibt es in Visual Studio 2010 für Windows Forms-Anwendungen keine Designerunterstützung, die Ihnen wirklich dabei hilft, neue Ressourcendateien für andere Kulturen aus schon vorhandenen zu erstellen. Sie müssen dazu selbst Hand anlegen und die IDE auch ein wenig austricksen. Die Unterstützung im Designer beschränkt sich lediglich auf Formulare selbst, bei denen Sie einfach mit ihrer `Localizable`-Eigenschaft bestimmen können, ob das Formular lokalisiert werden soll oder nicht.

HINWEIS Wenn Sie die folgenden Schritte nachvollziehen möchten, empfehle ich Ihnen, die Projektdateien zum »Kaputtexperimentieren« zunächst an einen anderen Ort auf Ihrer Festplatte zu kopieren. Sollte bei den Versuchen dann etwas schief gehen, können Sie das Quellprojekt einfach wieder dort rüberkopieren und ohne Verlust von vorne beginnen.

1. Öffnen Sie als Erstes die »Experimentierkopie« von DotNetCopy und lassen Sie den Projektmappen-Explorer mit Strg+Alt+L darstellen, falls dieser noch nicht angezeigt wird.
2. Klicken Sie auf das entsprechende Symbol im Projektmappen-Explorer, um alle Dateien des Projekts anzeigen zu lassen. Öffnen Sie den Zweig, der sich neben dem jetzt sichtbaren Eintrag *My Project* befindet.
3. Die Datei *Resources.resx*, in der sich die Ressourcen für die deutsche Kultur befinden, wird nun neben anderen angezeigt.
4. Klicken Sie mit der rechten Maustaste auf diese Datei, um das Kontextmenü zu öffnen, und wählen Sie dort den Eintrag *Kopieren*.
5. Im nächsten Schritt müssen Sie die IDE ein wenig austricksen: Sie müssen nun die Datei exakt an der Stelle einfügen – und damit eine Kopie der Ressourcendatei erstellen –, an der die Ausgangsdatei ebenfalls platziert war. Klicken Sie allerdings wieder mit der rechten Maustaste, dann befindet sich im Kontextmenü kein Eintrag namens *Einfügen*. Aus diesem Grund klicken Sie die Datei *Resources.resx* mit der *linken* Maustaste an, um sie zu selektieren, und wählen Sie anschließend *Einfügen* aus dem Menü *Bearbeiten* der Visual Studio-IDE. Anschließend befindet sich die angestrebte Kopie der Datei ebenfalls im Zweig *My Project*.
6. Nun benennen Sie die Ressourcendatei um. Später wird .NET Framework dann, wenn es Ihr Programm ausführen muss, anhand des Namens der Ressourcendatei wissen, welche Datei es für welche zugrunde liegende Kultur verwenden soll. Auf deutschen Systemen wird, wenn Sie auf einer deutschen Windows-Plattform entwickelt haben, immer die Ressourcendatei verwendet, mit der Sie die Anwendung entwickelt haben. Für ein englisches System (bzw. US-amerikanisches) benennen Sie die gerade erstellte Kopie der Ressourcendatei in *Resources.en.resx* um, für ein französisches in *Resources.fr.resx*, für ein italienisches in *Resources.it.resx* usw.

TIPP Die gängigen Sprachkürzel entsprechen denen, wie sie bei der Programmierung von kulturabhängigen Formatproviders verwendet werden.

Abbildung 23.9 Wenn Sie die zusätzliche Ressourcendatei ins Projekt kopiert haben, können Sie sich an das Übersetzen der Ressourcen machen

7. Wenn Sie die Datei umbenannt haben,[2] müssen Sie noch mit ein paar Handgriffen dafür sorgen, dass diese neue Ressourcendatei nicht dafür verantwortlich ist, dass eine weitere Codedatei erzeugt wird, bei der dann, bedingt durch die schon vorhandene deutsche Version, Codeelemente für das Abrufen der Ressourcen doppelt definiert werden würden, was zu Fehlern führen würde. Dazu klicken Sie die umbenannte Datei an und löschen im Eigenschaftenfenster die Werte für *Benutzerdefiniertes Tool* und *Namespace des benutzerdefinierten Tools*.
8. Klicken Sie anschließend doppelt auf die unbenannte Datei, um die Einträge die sie beinhaltet, zu lokalisieren – etwa wie in Abbildung 23.9 zu sehen.
9. Speichern Sie die Änderungen anschließend und lassen Sie das komplette Projekt neu erstellen.

Sie werden anschließend sehen, dass im Verzeichnis der ausführbaren Dateien des Projekts (standardmäßig ist das der Pfad *Projekt\bin\Debug*) ein weiteres Unterverzeichnis hinzugekommen ist, das eine so genannte Satellitenassembly beinhaltet. Diese Satellitenassembly beinhaltet die lokalisierten Ressourcen für jede Ausgangsressourcendatei und wird von .NET Framework automatisch verwendet, wenn der Name des Unterordners mit dem entsprechenden Kulturkürzel der Kultur des Systems übereinstimmt, auf dem die Anwendung ausgeführt wird (*en* für Englisch, *fr* für Französisch usw.). Das bedeutet: Wenn Sie die Satellitenassemblys auf diese Weise erzeugt haben, dann brauchen Sie sich selbst um nichts Weiteres mehr zu kümmern – sorgen Sie lediglich dafür, dass sie in entsprechenden Unterverzeichnissen vorhanden sind.

Abbildung 23.10 Auf diesem ohne Zweifel englischen Betriebssystem wird automatisch – so vorhanden – die englische Satellitenassembly verwendet; in der Ressourcendatei zuvor lokalisierte Texte erscheinen dann auch in der Landessprache

[2] Laut Aussagen des Entwicklerteams ist dieser Schritt seit der finalen Version von Visual Studio 2005 nicht mehr nötig; ein Auslassen dieses Schrittes hat aber bei einigen Experimenten Probleme verursacht, und so kann es nicht schaden, diesen zusätzlichen Schritt dennoch zu erledigen.

Auf einem englischen System würde die Anwendung dann automatisch so erscheinen, wie es Abbildung 23.10 auch wiedergibt (achten Sie hier auf die Bezeichnungen der `ListView`-Spalten, denn nur diese werden beispielhaft bei DotNetCopy aus der Ressourcendatei entnommen).

Vereinfachtes Durchführen von Dateioperationen mit My.Computer.FileSystem

`FileInfo` und `DirectoryInfo` unterstützen Sie bei der Ausführung von Dateioperationen bereits auf eine sehr komfortable Art und Weise. Doch bestimmte Funktionen im `My`-Namespace können das noch besser, und sie machen es erst möglich, dass unser Backuptool DotNetCopy trotz der vergleichsweise großen Flexibilität dennoch relativ wenig Code benötigt.

Die Funktionen, von denen DotNetCopy gerade bei der Durchführung des eigentlichen Backupvorgangs sehr intensiv Gebrauch macht, finden sich in `My.Computer.FileSystem`. Besonders bemerkenswert sind hier Funktionen wie `GetFiles`, mit denen ganze Verzeichnisstrukturen bzw. die Namen der sich dort befindlichen Dateien rekursiv eingelesen werden können. Sie kennen das vielleicht von dem *Dir*-Befehl der Befehlszeile von Windows: Nutzen Sie diesen mit der Option */S*, werden nicht nur die Verzeichnisse bzw. Dateien des Verzeichnisses aufgelistet, das Sie angegeben haben bzw. in dem Sie sich derzeit befinden, sondern es werden auch alle Unterverzeichnisse berücksichtigt.

Die eigentliche Kopierroutine von DotNetCopy macht von dieser `GetFiles`-Funktion Gebrauch, indem sie alle Unterverzeichnisse samt der darin enthaltenen Dateien ermittelt, zwischenspeichert und gemäß der so erstellten Liste im zweiten Schritt all diese Dateien kopiert. Der folgende Codeausschnitt demonstriert den Umgang:

```
    ''' <summary>
    ''' Die eigentliche Kopierroutine, die durch die My.Settings-Optionen gesteuert wird.
    ''' </summary>
    ''' <remarks></remarks>
    Public Sub CopyFiles()
.
.  'Aus Platzgründen gekürzt
.
        'In dieser Schleife werden zunächst alle Dateien ermittelt,
        'die es daraufhin zu untersuchen gilt, ob sie kopiert werden
        'müssen oder nicht.
        For locItemCount As Integer = 0 To lvwCopyEntries.Items.Count - 1
            Dim locCopyListEntry As CopyListEntry
            'CType ist notwendig, damit .NET weiß, welcher Typ
            'in der Tag-Eigenschaft gespeichert war.
            locCopyListEntry = CType(lvwCopyEntries.Items(locItemCount).Tag, CopyListEntry)
            'UI aktualisieren
            lblCurrentPass.Text = "Pass 1: Kopiervorbereitung durch Zusammenstellung der Pfade..."
            lblSourceFileInfo.Text = "Unterverzeichnisse suchen in:"
            lblDestFileInfo.Text = ""
            lblProgressCaption.Text = "Fortschritt Vorbereitung:"
            lblCurrentSourcePath.Text = locCopyListEntry.SourceFolder.ToString
            LogInProtocolWindow(locCopyListEntry.SourceFolder.ToString)
            pbPrepareAndCopy.Value = locItemCount + 1

            'Windows die Möglichkeit geben, die Steuerelemente zu aktualisieren
```

```
            My.Application.DoEvents()

            'GetFiles aus My.Computer.FileSystem liefert die Namen aller
            'Dateien im Stamm- und in dessen Unterverzeichnissen.
            Dim locFiles As ReadOnlyCollection(Of String)
            Try
                locNotCaught = False
                locFiles = My.Computer.FileSystem.GetFiles( _
                        locCopyListEntry.SourceFolder.ToString, _
                        FileIO.SearchOption.SearchAllSubDirectories, _
                        locCopyListEntry.SearchMask)
            Catch ex As Exception
                LogError(ex)
                'Wenn schon beim Zusammenstellen der Dateien Fehler aufgetreten sind,
                'sollte das Verzeichnis ausgelassen werden. In diesem Fall sollte dieser
                'schwerwiegende Fakt im Anwendungsprotokoll protokolliert werden!
                My.Application.Log.WriteException(ex, TraceEventType.Error, _
                    "DotNetCopy konnte das Verzeichnis " & _
                    locCopyListEntry.SourceFolder.ToString & " nicht durchsuchen., " & _
                    "Das Verzeichnis wurde daher nicht berüscksichtigt!")

                ' Es müsste wie "Catch" einen "NotCaught"-Zweig geben...
                locNotCaught = True
            End Try
.
.
.
```

Eine weitere Funktionalität, die ebenfalls sehr sinnvoll und praktisch ist, stellt `My.Computer.FileSystem` in Form der `CopyFile`-Methode zur Verfügung. Anders als die Methode `CopyTo` der `FileInfo`-Klasse greift diese exakt auf die gleiche Betriebssystemfunktion von Windows zurück, die auch beispielsweise der Windows-Explorer verwendet. Und das bedeutet – auch wenn DotNetCopy in diesem Beispiel davon keinen Gebrauch macht –, dass Sie bei der Verwendung von `CopyFile` aus dem `My`-Namespace automatisch den bekannten Fortschrittsdialog anzeigen lassen können.

```
    ''' <summary>
    ''' Interne Kopierroutine, die den eigentlichen 'Job' des Kopierens übernimmt.
    ''' </summary>
    ''' <param name="SourceFile">Quelldatei, die kopiert werden soll.</param>
    ''' <param name="DestFile">Zielpfad, in den die Datei hineinkopiert werden soll.</param>
    ''' <remarks></remarks>
    Private Sub CopyFileInternal(ByVal SourceFile As FileInfo, ByVal DestFile As FileInfo)
        'Falls die Zieldatei noch gar nicht existiert,
        'dann in jedem Fall kopieren
        If Not DestFile.Exists Then
            Try
                'Datei kopieren, ohne Windows-Benutzeroberfläche anzuzeigen.
                My.Computer.FileSystem.CopyFile(SourceFile.ToString, DestFile.ToString)
                LogInProtocolWindow("Kopiert, OK: " & SourceFile.ToString)
            Catch ex As Exception
                LogError(ex)
            End Try
            Exit Sub
        End If
        'Datei nur kopieren, wenn sie jünger als die zu überschreibende ist
        If My.Settings.Option_OnlyOverwriteIfOlder Then
```

```
            If SourceFile.LastWriteTime > DestFile.LastWriteTime Then
                'Wenn das Historien-Backup aktiviert ist...
                If My.Settings.Option_EnableBackupHistory Then
                    '...dieses durchführen.
                    ManageFileBackupHistory(DestFile)
                End If
                Try
                    'Datei kopieren.
                    My.Computer.FileSystem.CopyFile(SourceFile.ToString, DestFile.ToString)
                    LogInProtocolWindow("Kopiert, OK: " & SourceFile.ToString)
                Catch ex As Exception
                    LogError(ex)
                End Try
            End If
        End If
    End Sub
```

HINWEIS Leider werden bei der Verwendung von `CopyFile` in Windows-Anwendungen, bei denen – wie es standardmäßig bei Windows Forms-Anwendungen geschieht – Formulare an eine Meldungswarteschlange gebunden werden, auch Fehlermeldungen als modale Dialoge dargestellt. Gerade bei Fehlern, die das Programm behandeln soll, kann das lästig und unerwünscht sein. Wichtig ist es aber auch zu wissen, dass bei Anwendungen, die keine Meldungswarteschlange initiieren, Fehlermeldungen nicht vom Betriebssystem in Form eines sichtbaren modalen Dialogs abgefangen werden, oder mit anderen Worten: Wenn Sie dafür sorgen, dass Ihr Programm keine sichtbare Benutzeroberfläche hat, dann erscheint auch keine sichtbare Fehlermeldung, selbst wenn bei `CopyFile` ein Fehler auftritt.

Im Falle unseres Beispiels DotNetCopy bedeutet das: Wir können im Silent-Modus keine sichtbaren Fehlermeldungen gebrauchen. Die gibt es aber auch bei `CopyFile` in diesem Modus ohnehin nicht, denn es ist keine sichtbare Benutzeroberfläche vorhanden. Zwar verwenden wir die gleiche Formularklasse, auf die wir auch dann zurückgreifen, wenn DotNetCopy nicht im Silent-Modus läuft, aber wir zeigen das Formular nicht an.

Der folgende Code, der sich im `StartUp`-Ereignis des `MyApplication`-Objekts befindet, und der ausgelöst wird, sobald die Anwendung startet, demonstriert das:

```
        'Hinweis: locFrmMain wurde weiter oben im Code schon instanziiert, aber nicht dargestellt!
        'Silentmode bleibt nur "an", wenn AutoStart aktiv ist.
        locFrmMain.SilentMode = locFrmMain.SilentMode And locFrmMain.AutoStartMode

        'Und wenn Silentmode, erfolgt keine Bindung des Formulars an den Anwendungskontext!
        If locFrmMain.SilentMode Then
            'Alles wird in der nicht sichtbaren Instanz des Hauptformulars durchgeführt,
            locFrmMain.HandleAutoStart()
            'und bevor das "eigentliche" Programm durch das Hauptformular gestartet wird,
            'ist der ganze Zauber auch schon wieder vorbei.
            e.Cancel = True
        Else
            'Im Nicht-Silent-Modus wird das Formular an die Anwendung gebunden,
            'und los geht's!
            My.Application.MainForm = locFrmMain
        End If
    End Sub
End Class
```

Verwenden von Anwendungseinstellungen mit My.Settings

DotNetCopy verwendet Anwendungseinstellungen (*ApplicationSettings*), um die Einstellungen auch nach dem Beenden des Programms zu erhalten, die der Benutzer vornehmen kann, wenn er aus dem Menü *Extras* den Menüeintrag *Optionen* auswählt. Da schon Kapitel 5 sich intensiv mit Anwendungseinstellungen auseinandergesetzt hat, erfolgt an dieser Stelle lediglich eine kompakte Zusammenfassung des dazu Bekannten:

Anwendungseinstellungen pflegen Sie – ähnlich wie die Zeichenkettentabellen der Ressourcen – über eine Tabelle, die Sie über die Projekteigenschaften erreichen (Abbildung 23.11).

Abbildung 23.11 Anwendungseinstellungen, die Sie über `My.Settings` abrufen können, verwalten Sie wie Ressourcen in den Projekteigenschaften auf der Registerkarte *Einstellungen*

Für jeden Namen, den Sie in der Tabelle anlegen, wird dabei eine über `My.Settings` zu erreichende Objektvariable von dem Typ angelegt, den Sie in der Tabelle unter *Typ* bestimmt haben. Sie können für die Variablen, die Sie an dieser Stelle erstellen, aus zwei Bereichen auswählen: Für *Benutzer* entscheiden Sie sich, wenn Sie den Variableninhalt zur Laufzeit verändern wollen und der neue Inhalt nach Programmende auch automatisch gesichert werden soll; *Anwendung* wählen Sie, wenn es sich um eine Konstante handeln soll, die nur Sie zur Entwurfszeit in den Projekteigenschaften (in der in Abbildung 23.11 gezeigten Tabelle) einstellen können, die man aber zur Laufzeit nicht verändern darf.

Das Speichern oder Auslesen von Anwendungseinstellungen ist schließlich eine Kleinigkeit. Der folgende Code zeigt, wie von Anwendungseinstellungen intensiv Gebrauch gemacht wird, wenn der Benutzer im Optionsdialog Änderungen vornimmt. Die nachstehende Routine, die aufgerufen wird, wenn der Optionsdialog geladen wird, sorgt zunächst dafür, dass die Steuerelemente die Anwendungseinstellungen widerspiegeln:

```
Private Sub frmOptions_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.Load
    'Anwendungseinstellungen im Dialog widerspiegeln lassen:
    'Auch dazu wird My.Settings verwendet.
    chkCopyFoldersRecursive.Checked = My.Settings.Option_CopyFoldersRecursive
    chkCopyHiddenFiles.Checked = My.Settings.Option_CopyHiddenFiles
    chkCopySystemFiles.Checked = My.Settings.Option_CopySystemFiles
    chkEnableBackupHistory.Checked = My.Settings.Option_EnableBackupHistory
    nudHistoryLevels.Value = My.Settings.Option_HistoryLevels
    chkNeverOverwriteFiles.Checked = My.Settings.Option_NeverOverwriteFiles
    chkOnCopyErrorContinue.Checked = My.Settings.Option_OnCopyErrorContinue
    chkOnlyOverwriteIfOlder.Checked = My.Settings.Option_OnlyOverwriteIfOlder
    chkBackupInHiddenFiles.Checked = My.Settings.Option_BackupInHiddenFiles
    chkAutoSaveProtocol.Checked = My.Settings.Option_AutoSaveProtocol
    lblProtocolPath.Text = My.Settings.Option_AutoSaveProtocolPath
End Sub
```

Beim Verlassen des Dialogs mit *OK* erfolgt der umgekehrte Weg: Die Werte oder Zustände der Steuerelemente werden ausgelesen und in den Anwendungseinstellungen gespeichert.

```
Private Sub btnOK_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnOK.Click
    'Die Einstellungen des Options-Dialogs in den Anwendungseinstellungen speichern.
    'Auch dazu wird My.Settings verwendet.
    My.Settings.Option_CopyFoldersRecursive = chkCopyFoldersRecursive.Checked
    My.Settings.Option_CopyHiddenFiles = chkCopyHiddenFiles.Checked
    My.Settings.Option_CopySystemFiles = chkCopySystemFiles.Checked
    My.Settings.Option_EnableBackupHistory = chkEnableBackupHistory.Checked
    My.Settings.Option_HistoryLevels = CInt(nudHistoryLevels.Value)
    My.Settings.Option_NeverOverwriteFiles = chkNeverOverwriteFiles.Checked
    My.Settings.Option_OnCopyErrorContinue = chkOnCopyErrorContinue.Checked
    My.Settings.Option_OnlyOverwriteIfOlder = chkOnlyOverwriteIfOlder.Checked
    My.Settings.Option_BackupInHiddenFiles = chkBackupInHiddenFiles.Checked
    My.Settings.Option_AutoSaveProtocol = chkAutoSaveProtocol.Checked
    My.Settings.Option_AutoSaveProtocolPath = lblProtocolPath.Text

    'Das Setzen des Dialogergebnisses bei einem modal aufgerufenen Dialog
    'bewirkt gleichzetig, dass dieser geschlossen wird!
    Me.DialogResult = Windows.Forms.DialogResult.OK
End Sub
```

Das Ablegen von Informationen in den Anwendungseinstellungen wird auch vom Designer der Windows-Anwendungen und der dort benutzten Steuerelemente verwendet. Wenn Sie beispielsweise die Beschriftung eines Formulars (also die `Text`-Eigenschaft einer Windows Forms-Klasse) außerhalb des Quellcodes der Anwendung festlegen und ändern wollen, ist dies über die Oberfläche einfach zu realisieren. Öffnen Sie ein beliebiges Formular und wechseln Sie dort ins Eigenschaftsfenster (F4). Wenn Sie die Eigenschaften nach Kategorien anzeigen lassen, gehen Sie zur Kategorie *Daten*, dort ist der erste Eintrag, in alphabetischer Sortierung ist es von vornherein der erste Eintrag: (*Application Settings*). Die wichtigsten Eigenschaften der Klasse lassen sich direkt (wie `Text` bei Windows Forms) eintragen, wählen Sie *(Neu)* im Listenfeld ganz unten. (Unter *(PropertyBinding)* lassen sich aber beinahe alle Eigenschaften auslagern.) Im folgenden Dialog geben Sie einfach einen Namen an, unter dem Sie die Einstellung speichern wollen. Unter *Scope* wählen Sie, ob die Einstellungen immer für die Anwendung gelten sollen oder ob diese für jeden Benutzer konfiguriert werden können. Dazu mehr im folgenden Kapitel.

Speichern von Anwendungseinstellungen mit dem Bereich Benutzer

Den Code zum Speichern der Anwendungseinstellungen werden Sie übrigens im Optionsdialog vergebens suchen – es gibt ihn nicht, da er im Grunde genommen nicht notwendig ist. Wenn eine Windows Forms-Anwendung beendet wird, für die sowohl das Anwendungsframework als auch die Option *My.Settings beim Herunterfahren speichern* in den Projekteigenschaften unter *Anwendung* aktiviert ist, dann regelt .NET Framework das Speichern der Anwendungseinstellungen für den Bereich *Benutzer* automatisch.

In einigen Fällen kann es aber sinnvoll sein, die Anwendungseinstellungen selbst oder schon vor dem Beenden des Programms zu speichern. In DotNetCopy ist das beispielsweise der Fall, wenn das Programm zum allerersten Mal gestartet wird. In diesem Fall werden die Einstellungen für die Position und die Größe des Hauptformulars festgelegt und direkt gespeichert – und das erfolgt mit der Methode `My.Settings.Save`, wie das folgende Beispiel zeigt:

```
Private Sub frmMain_Load(ByVal sender As Object, ByVal e As System.EventArgs) _
.
. ' Aus Platzgründen ausgelassen
.
    'Fenstergröße und -position wiederherstellen, wenn
    'nicht im Silent-Modus
    If Not mySilentMode Then
        'Beim ersten Aufruf ist die Größe -1, dann bleibt die Default-
        'Position, die Settings werden aber sofort übernommen...
        If My.Settings.frmMain_LastPosition.X = -1 Then
            My.Settings.frmMain_LastPosition = Me.Location
            Me.Size = My.Settings.frmMain_LastSize
            My.Settings.Save()

            'anderenfalls werden die Settings-Einstellungen in die Formular-
            'Position übernommen
        Else
            Me.Location = My.Settings.frmMain_LastPosition
        End If
        Me.Size = My.Settings.frmMain_LastSize
    End If
```

Kapitel 24

Das Anwendungsframework

In diesem Kapitel:

Die Optionen des Anwendungsframeworks 741

Eine Codedatei implementieren, die Anwendungsereignisse (Starten, Beenden, Netzwerkzustand, generelle Fehler) behandelt 742

Das, was Microsoft als »Anwendungsframework« bezeichnet, ist eine Reihe von Funktionalitäten, die unter anderem auch dafür verantwortlich sind, dass Anwendungseinstellungen nach dem Beenden eines Programms gespeichert oder auch dass Anwendungsereignisse zur Verfügung gestellt werden (siehe Abschnitt »Eine Codedatei implementieren, die Anwendungsereignisse (Starten, Beenden, Netzwerkzustand, generelle Fehler) behandelt« ab Seite 742).

Um das zu erreichen, bedient sich Visual Basic übrigens zweier Mittel: Zum einen ergänzt der Visual Basic-Compiler Ihren Quellcode um weiteren Code, den Sie selbst aber nie zu Gesicht bekommen. Dieser Quellcode ist quasi fest im Compiler hinterlegt, und werden bestimmte Funktionalitäten benötigt, die Sie durch die Optionen des Anwendungsframeworks in den Projekteigenschaften für Ihr Projekt aktivieren, ergänzt der Compiler dafür während der Kompilierung den entsprechenden Quellcode, der dann wiederum dafür sorgt, dass Ihr Programm die entsprechenden Funktionalitäten bereitstellt. Zum anderen benötigt Visual Basic dazu bestimmte Funktionsaufrufe, die durch die *VisualBasic.dll* von .NET Framework zur Verfügung gestellt werden.

Für Windows-Anwendungen ist das Anwendungsframework standardmäßig aktiviert. Sie können die Aktivierung des Anwendungsframeworks in den Projekteigenschaften auf der Registerkarte *Anwendung* steuern (Abbildung 24.1).

Abbildung 24.1 Erst durch das aktivierte Anwendungsframework stehen viele der hier beschriebenen Funktionalitäten zur Verfügung. Bei Windows Forms-Anwendungen ist das Anwendungsframework standardmäßig aktiviert.

Wenn das Kontrollkästchen *Anwendungsframework aktivieren* eingeschaltet ist, verwendet die Anwendung dann eine quasi »versteckte« `Sub Main` (die durch den versteckten Quellcode des Compilers in Ihr Projekt »gelangt«), mit der Ihre Windows-Anwendung startet und die es erst ermöglicht, dass alle weiteren Optionen des Anwendungsframeworks funktionieren können. In diesem Fall müssen Sie ein Startformular auswählen, das von dieser versteckten `Sub Main` aufgerufen wird, wenn alle nötigen Vorbereitungen für die Zurverfügungstellung aller anderen Funktionen des Anwendungsframeworks abgeschlossen sind.

Wenn dieses Kontrollkästchen deaktiviert ist, verwendet die Anwendung die benutzerdefinierte `Sub Main`, die Sie unter *Startformular* angegeben haben und die Sie in einem Modul Ihres Projekts platzieren müssen. In diesem Fall können Sie entweder ein Startobjekt (eine benutzerdefinierte `Sub Main` in einer Methode oder Klasse) oder ein Formular festlegen. Die Optionen im Bereich *Eigenschaften des Windows-Anwendungsframeworks* sind in diesem Fall nicht verfügbar.

Die Optionen des Anwendungsframeworks

Die folgenden Einstellungen dienen der Konfiguration des Abschnitts *Eigenschaften des Windows-Anwendungsframeworks*. Diese Optionen sind nur verfügbar, wenn das Kontrollkästchen *Anwendungsframework aktivieren* eingeschaltet ist.

Windows XP Look and Feel für eigene Windows-Anwendungen – Visuelle XP-Stile aktivieren

Aktivieren oder deaktivieren Sie die visuellen Windows XP-Stile, die auch Windows XP-Designs genannt werden. Die visuellen Windows XP-Stile lassen zum Beispiel Steuerelemente mit gerundeten Ecken und dynamischen Farben zu. Die Option ist standardmäßig aktiviert; eine entsprechende visuelle Anpassung wird auch für Vista und Windows 7 vorgenommen.

Verhindern, dass Ihre Anwendung mehrfach gestartet wird – Einzelinstanzanwendung erstellen

Aktivieren Sie dieses Kontrollkästchen, um zu verhindern, dass Benutzer mehrere Instanzen der Anwendung ausführen. Dieses Kontrollkästchen ist standardmäßig deaktiviert, wodurch mehrere Instanzen der Anwendung ausgeführt werden können.

Eigene Einstellungen beim Herunterfahren automatisch sichern – My.Settings beim Herunterfahren speichern

Durch das Aktivieren dieses Kontrollkästchens wird festgelegt, dass die `My.Settings`-Einstellungen der Anwendung beim Herunterfahren gespeichert werden. Die Option ist standardmäßig aktiviert. Wenn diese Option ausgeschaltet wird, können Sie diese Festlegung durch Aufrufen von `My.Settings.Save` manuell durchführen.

Bestimmen, welcher Benutzer-Authentifizierungsmodus verwendet wird

Wählen Sie unter *Authentifizierungsmodus* den Eintrag *Windows* (es ist die Standardeinstellung) aus, um die Verwendung der Windows-Authentifizierung für die Identifikation des gerade angemeldeten Benutzers festzulegen. Diese Informationen können zur Laufzeit mit dem `My.User`-Objekt abgerufen werden. Wählen Sie *Anwendungsdefiniert* aus, wenn Sie eigenen Code anstatt der Windows-Standardmethoden für die Authentifizierung von Benutzern verwenden möchten.

Festlegen, wann eine Anwendung »zu Ende« ist – Modus für das Herunterfahren

Wählen Sie *Beim Schließen des Startformulars* (Standardeinstellung) aus, um festzulegen, dass die Anwendung beendet wird, wenn das als Startformular festgelegte Formular geschlossen wird. Dies gilt auch, wenn andere Formulare geöffnet sind. Wählen Sie *Beim Schließen des letzten Formulars* aus, um zu erreichen, dass die Anwendung beendet wird, wenn das letzte Formular geschlossen oder wenn `My.Application.Exit` bzw. die `End`-Anweisung explizit aufgerufen wird.

Einen Splash-Dialog beim Starten von komplexen Anwendungen anzeigen – Begrüßungsbildschirm

Wählen Sie das Formular aus, das als Begrüßungsbildschirm angezeigt werden soll. Zuvor müssen Sie einen Begrüßungsbildschirm mithilfe eines Formulars oder einer Vorlage (indem Sie zum Beispiel im Projektmappen-Explorer das Projekt markieren und per Rechtsklick *Hinzufügen/Neues Element/Begrüßungsbildschirm* auswählen) erstellt haben.

Eine Codedatei implementieren, die Anwendungsereignisse (Starten, Beenden, Netzwerkzustand, generelle Fehler) behandelt

In den vorangegangenen Abschnitten haben Sie schon hier und da am Rande mitbekommen, dass es offensichtlich möglich sein muss, bestimmte Anwendungsereignisse »mithören« zu können. *DotNetCopy* klinkt sich so beispielsweise in den Programmstart ein und steuert sogar das weitere Schicksal des Programms aufgrund der übergebenen Optionen.

Wenn bei bestimmten Anwendungsereignissen Ihr eigener Code ausgeführt werden soll, müssen Sie dazu eine kleine Vorbereitung treffen:

- Rufen Sie dazu die Eigenschaften Ihres Projekts auf, und wählen Sie die Registerkarte *Anwendung*.
- Klicken Sie auf die Schaltfläche *Anwendungsereignisse anzeigen* (Abbildung 24.1 auf Seite 740).
- Die Visual Studio-IDE fügt Ihrem Projekt nun eine Codedatei namens *ApplicationEvents.vb* hinzu und öffnet diese im Editor.

- Um den Funktionsrumpf für die Behandlungsroutine in eines der fünf existierenden Anwendungsereignisse einzufügen, wählen Sie im Codeeditor aus dem linken Listenfeld am oberen Rand des Editors zunächst den Eintrag *(MyApplication-Ereignisse)*.
- Wählen Sie anschließend in der rechten Liste das Ereignis, das Sie behandeln wollen und für das der Funktionsrumpf in den Editor eingefügt werden soll. Orientieren Sie sich dabei auch an Abbildung 24.2.

Abbildung 24.2 Wählen Sie zum Einfügen des Rumpfs der Ereignisbehandlungsroutine eines der fünf existierenden Ereignisse aus dem Listenfeld aus

- Formulieren Sie den Behandlungscode aus.

HINWEIS Beachten Sie auch, dass einige Ereignisse besondere Ableitungen der `EventArgs`-Klasse als Instanz übergeben, mit denen Sie das weitere Programmverhalten beeinflussen können. Die folgende Tabelle verrät mehr zu den vorhandenen Ereignissen und wie Sie in ihrem Rahmen den weiteren Programmverlauf beeinflussen können.

Ereignisname	Beschreibung	Steuerungsmöglichkeiten
`Startup`	Wird beim Starten der Anwendung noch vor dem Erstellen des Startformulars und dessen Binden an den Anwendungskontext ausgelöst	Ja: Der Start der Anwendung kann durch Setzen von `Cancel` der `StartupEventArgs`-Instanz verhindert werden. In `CommandLineArgs` dieser Instanz werden ferner die Befehlszeilenargumente an die Anwendung als `String`-Array übergeben.
`Shutdown`	Wird nach dem Schließen aller Anwendungsformulare ausgelöst, es sei denn, die Anwendung wird nicht normal beendet (beispielsweise durch einen Fehler, der eine Ausnahme ausgelöst hat)	Nein ▶

Ereignisname	Beschreibung	Steuerungsmöglichkeiten
StartupNextInstance	Wird beim Starten einer Einzelinstanzanwendung ausgelöst, wenn diese bereits aktiv ist. HINWEIS: Sie bestimmen, dass eine Anwendung nur in einer Instanz (also nicht mehrmals gleichzeitig) gestartet werden darf, wenn Sie in den Projekteigenschaften Ihres Windows Forms-Projekts auf der Registerkarte *Anwendung* die Option *Einzelinstanzanwendung erstellen* aktivieren.	Ja: Mit dem Setzen der BringToForeground-Eigenschaft der StartupNextInstanceEventArgs-Instanz können Sie festlegen, dass nach Verlassen der Ereignisbehandlungsroutine die schon laufende Instanz Ihrer Anwendung wieder zur aktiven Anwendung wird. In CommandLineArgs dieser Instanz werden ferner die Befehlszeilenargumente an die Anwendung als String-Array übergeben (und zwar die Befehlszeilenargumente des Aufrufs, der zum erneuten Startversuch und damit auch zum Auslösen des Ereignisses führte).
NetworkAvailabilityChanged	Wird beim Herstellen oder Trennen der Netzwerkverbindung ausgelöst	Nein. Sie können jedoch mit der Nur-Lesen-Eigenschaft IsNetworkAvailable abfragen, ob das Netzwerk zur Verfügung steht oder nicht.
UnhandledException	Wird ausgelöst, wenn in der Anwendung aufgrund eines nicht abgefangenen Fehlers eine unbehandelte Ausnahme auftritt	Ja: Das Beenden der Anwendung aufgrund des Fehlers kann durch Setzen von Cancel der UnhandledExceptionEventArgs-Instanz verhindert werden. Darüber hinaus steht Ihnen durch die Exception-Eigenschaft dieser Instanz ein Exception-Objekt zur Verfügung, das Auskunft über die aufgetretene unbehandelte Ausnahme gibt.

Tabelle 24.1 Diese Ereignisse des Anwendungsframeworks stehen Ihnen zur Verfügung

Der folgende Beispielcode demonstriert den Umgang mit den Anwendungsereignissen:

```
Private Sub MyApplication NetworkAvailabilityChanged(ByVal sender As Object,
        ByVal e As Microsoft.VisualBasic.Devices.NetworkAvailableEventArgs)
        Handles Me.NetworkAvailabilityChanged
    MessageBox.Show("Die Netzwerkverfügbarkeit hat sich geändert!" &
        "Das Netzwerk ist " & IIf(e.IsNetworkAvailable, "verfügbar",
        "nicht verfügbar").ToString)
End Sub

Private Sub MyApplication Shutdown(ByVal sender As Object, ByVal e As System.EventArgs)
       Handles Me.Shutdown
    Debug.Print("Die Anwendung wird jetzt beendet!")
End Sub

Private Sub MyApplication StartupNextInstance(ByVal sender As Object,
        ByVal e As Microsoft.VisualBasic.ApplicationServices.StartupNextInstanceEventArgs)
        Handles Me.StartupNextInstance
    MessageBox.Show("Sie können keine 2. Instanz dieser Anwendung starten!")
    e.BringToForeground = True
End Sub

Private Sub MyApplication UnhandledException(ByVal sender As Object,
        ByVal e As Microsoft.VisualBasic.ApplicationServices.UnhandledExceptionEventArgs)
        Handles Me.UnhandledException
    'Eine bislang nicht behandelte Ausnahme ist aufgetreten!
    Dim locDr As DialogResult = _
```

```
                MessageBox.Show("Eine unbehandelte Ausnahme ist aufgetreten!" & _
                        vbNewLine & "Soll die Anwendung beendet werden?", _
                        "Unbehandelte Ausnahme", MessageBoxButtons.YesNo, _
                        MessageBoxIcon.Error, MessageBoxDefaultButton.Button1)
            If locDr = DialogResult.No Then
                e.ExitApplication = False
            End If
        End Sub
```

Ein dokumentiertes Beispiel für das Ereignis `StartUp`, das beim Starten einer Windows Forms-Anwendung ausgelöst wird, finden Sie auch im Beispielprogramm DotNetCopy – der Vollständigkeit halber ist der komplette Code dieser Ereignisbehandlungsroutine im Folgenden abgedruckt:

BEGLEITDATEIEN Viele der hier gezeigten Features können Sie sich direkt am Beispiel von DotNetCopy (vorgestellt im vorherigen Kapitel) anschauen. Sie finden dieses Projekt unter dem Namen *DotNetCopy.sln* im Verzeichnis

```
...\VB 2010 Entwicklerbuch\Kapitel 24\DotNetCopy
```

```
        Private Sub MyApplication_Startup(ByVal sender As Object,
               ByVal e As Microsoft.VisualBasic.ApplicationServices.StartupEventArgs) Handles Me.Startup


            'Das Verzeichnis für die Protokolldatei beim ersten Mal setzen...
            If String.IsNullOrEmpty(My.Settings.Option_AutoSaveProtocolPath) Then
                My.Settings.Option_AutoSaveProtocolPath = _
                    My.Computer.FileSystem.SpecialDirectories.MyDocuments & "\DotNetCopy Protokolle"
                Dim locDi As New DirectoryInfo(My.Settings.Option_AutoSaveProtocolPath)

                'Überprüfen und
                If Not locDi.Exists Then
                    'im Bedarfsfall anlegen
                    locDi.Create()
                End If

                'Settings speichern
                My.Settings.Save()
            End If

            Dim locFrmMain As New frmMain

            'Kommandozeile auslesen
            If My.Application.CommandLineArgs.Count > 0 Then
                For Each locString As String In My.Application.CommandLineArgs

                    'Alle unnötigen Leerzeichen entfernen und
                    'Groß-/Kleinschreibung 'Unsensibilisieren'
                    'HINWEIS: Das funktioniert nur in der Windows-Welt;
                    'kommt die Kopierlistendatei von einem Unix-Server, bitte darauf achten,
                    'dass der Dateiname dafür auch komplett in Großbuchstaben gesetzt ist,
                    'da Unix- (und Linux-) Derivate Groß-/Kleinschreibung berücksichtigen!!!
                    locString = locString.ToUpper.Trim
```

```
                    If locString = "/SILENT" Then
                        locFrmMain.SilentMode = True
                    End If

                    If locString.StartsWith("/AUTOSTART") Then
                        locFrmMain.AutoStartCopyList = locString.Replace("/AUTOSTART:", "")
                        locFrmMain.AutoStartMode = True
                    End If
                Next
            End If

            'Silentmode bleibt nur "an", wenn AutoStart aktiv ist.
            locFrmMain.SilentMode = locFrmMain.SilentMode And locFrmMain.AutoStartMode

            'Und wenn Silentmode, erfolgt keine Bindung des Formulars an den Anwendungskontext!
            If locFrmMain.SilentMode Then
                'Alles wird in der nicht sichtbaren Instanz des Hauptformulars durchgeführt,
                locFrmMain.HandleAutoStart()
                'und bevor das "eigentliche" Programm durch das Hauptformular gestartet wird,
                'ist der ganze Zauber auch schon wieder vorbei.
                e.Cancel = True
            Else
                'Im Nicht-Silent-Modus wird das Formular an die Anwendung gebunden,
                'und los geht's!
                My.Application.MainForm = locFrmMain
            End If
        End Sub
    End Class

End Namespace
```

Teil E

Language Integrated Query – LINQ

In diesem Teil:

Einführung in Language integrated Query (LINQ) 749
LINQ to Objects 763
LINQ to XML 787
LINQ to Entities – Programmieren mit Entity Framework 797

Kapitel 25

Einführung in Language integrated Query (LINQ)

In diesem Kapitel:

Wie »geht« LINQ prinzipiell 751
Die Where-Methode 755
Die Select-Methode 756
Kombinieren von LINQ-Erweiterungsmethoden 760
Vereinfachte Anwendung von LINQ – Erweiterungsmethoden mit der LINQ-Abfragesyntax 761

Programme verarbeiten irgendwelche Daten – lässt man den Bezug zur realen Welt einmal außer Acht, was in der Softwareentwicklung eigentlich die Verletzung des obersten Gebotes ist – im Grunde kann man also den Sinn und Zweck jedes Programms auf diesen Nenner herunterbrechen.

Doch Daten sind nicht gleich Daten. Vom Prinzip her zwar schon, aber wir Entwickler benötigen eigentlich die meiste Zeit, Daten so zu transformieren, dass wir damit arbeiten können. Und diese Transformationen kosten Zeit. Beispiel:

- Wenn Sie Daten aus einer XML-Datei laden, müssen Sie den Aufwand betreiben (oder betreiben lassen), die wirklichen Daten aus dem XML-Format herauszuparsen und die Daten, die Sie wirklich benötigen, in entsprechenden Feldern, Arrays, Auflistungen oder eigenen Klasseninstanzen zu speichern und damit weiterzuverarbeiten.
- Wenn Sie Daten selbst ermitteln, beispielsweise von Hardwaregeräten wie Kameras, Scannern oder Multi-IO-Karten, müssen Sie Treiber abfragen oder sogar selbst implementieren, um die entsprechenden analogen Spannungszustände zu messen und wiederum in die Daten so umzuwandeln, wie Sie sie benötigen.
- Wenn Sie Daten einer Datenbank benötigen, müssen Sie eine Verbindung zu ihr aufbauen, entsprechende Abfragebefehle formulieren, an sie senden und die Daten, die Sie dann zurückerhalten, abermals so aufbereiten und in eigenen Objekten speichern, damit Sie sie in Ihrer Anwendung weiterverarbeiten können.

Sie sehen: Wir können diese Liste eigentlich beliebig fortführen. Wann immer Sie programmieren, Sie hantieren mit Daten. Und das Schlimme ist: Vor dem Erscheinen von Visual Studio 2008 funktionierte das für jede Datenquelle unter Umständen uneinheitlich und vor allen Dingen oft nicht typsicher: Wollten Sie beispielsweise noch in Visual Basic 2005 die Daten aus einer Tabelle von Microsoft SQL Server (oder eines anderen SQL Servers) bekommen, mussten Sie die Abfrage zwangsläufig als Zeichenfolge an den SQL Server schicken. Die Visual Studio-IDE konnte Ihnen wenn überhaupt nur über Assistentendialoge dabei helfen, die Abfrage so zu formulieren, dass Sie auch die richtigen Feldnamen oder Tabellennamen wählten. Im Editor waren Sie komplett auf sich alleine gestellt.

Typsicherheit auch bei der Abfrage von Daten innerhalb des Editors für Entwickler anzubieten, das war das Ziel bei der Entwicklung einer neuen Technologie, wie sie ungefähr ab 2003 schon von Microsoft vorangetrieben wurde (und die ersten Ansätze reichen sogar schon im Rahmen von *ObjectSpaces* – googeln Sie mal danach – bis vor der Veröffentlichung der ersten .NET-Version zurück). Es ging also letzten Endes darum, Datenabfragen, erst einmal völlig unabhängig davon, welche Datenquelle dahintersteckte, irgendwie in die Programmiersprache zu integrieren, mit der man die Daten abfragen wollte. Gesucht war also eine sprachintegrierte Abfragemöglichkeiten, auf Englisch: *Language Integrated Query*, kurz *LINQ*, was sich im übrigens »Link«, und nicht etwa »Linkju« ausspricht.

Die Idee dahinter war es, die Datenquelle mehr oder weniger zu egalisieren, sodass sich – je nach verwendetem LINQ-Provider – die Abfragen einer Datenbank genauso, oder zumindest, fast genauso anfühlen würden wie die Abfrage von Daten aus einem Array oder einer Auflistung.

Entsprechende LINQ-Provider sollten dann einerseits dafür sorgen, dass bei einer Abfrage gegen im Arbeitsspeicher befindliche Datenobjekte – also Auflistungen oder Arrays – entsprechende Suchschleifen ausgeführt werden. Andererseits sollten die LINQ-Provider bei einer ähnlich formulierten Abfrage gegen eine Datenbank dann eben entsprechende T-SQL-Statements produzieren, die schließlich wiederum an den SQL Server geschickt werden und dort die eigentliche (SQL-)Abfrage ausführen lassen. Und so gibt es heute folgende unterschiedliche Einsatzgebiete für LINQ:

- **LINQ to Objects** Damit können Sie SQL-ähnliche Abfragen an Auflistungen oder Arrays stellen. Und zwar an alle Auflistungen, die die Schnittstelle `IEnumerable` implementieren – und das sind die meisten. Sie haben also die Möglichkeit, Auflistungen die aus Instanzen ihrer eigenen Klassen bestehen, quasi so zu verwenden, als wären es ihre eigenen, kleinen lokalen Datenbanken, deren Engine direkt in Ihre Anwendung eingebaut ist.
- **LINQ to SQL** Damit können Sie SQL-ähnliche Abfragen direkt an einen SQL Server stellen. LINQ to SQL ist eine schnelle Möglichkeit, Microsoft SQL Server-Datenbanken abzufragen. Dazu gibt es ein Tool, das zunächst über einen so genannten OR-Mapper Code für Objektklassen generiert, die dann den Datentabellen in SQL Server entsprechen. Damit »kennt« quasi auch der Visual Basic-Editor und sein IntelliSense den Aufbau (das so genannte »Schema«) der Datenbank und kann Ihnen nun beim Schreiben der (LINQ-)Abfragen helfen. *LINQ to SQL* meint aber dabei nicht nur die Möglichkeit, in LINQ formulierte Abfragen in T-SQL-Abfragen umzuwandeln, sondern auch die Funktionalitäten zur Verfügung zu stellen, mit denen sich auf einfache Weise Daten in SQL Server hinzufügen oder verändern lassen. Leider gibt es seit Visual Studio 2008 SP1 zwei konkurrierende Technologien (siehe *LINQ to Entities*), und LINQ to Entities ist die Technologie, die in Zukunft weiterentwickelt wird, sodass wir auf LINQ to SQL im Rahmen dieses Buchs nicht weiter eingehen werden.

HINWEIS Dass LINQ to SQL in seiner Entwicklung von den Teams nicht fortgeführt wird, bedeutet nicht, dass es aus .NET Framework rausfliegen wird. Es hat nicht nur seine Existenzberechtigung, sondern durchaus auch seine Vorteile: So ist es, da es ein wenig »stromlinienförmiger« entwickelt wurde, beispielsweise ein wenig schneller bei der Abfrage großer Datenmengen aus SQL Server. Es ist aber andererseits auf SQL Server beschränkt und lange nicht so flexibel wie LINQ to Entities.

- **LINQ to XML** Mit XML als Zielprovider können Sie Abfragen direkt auf XML-Dokumente loslassen, um eine Auflistung entsprechender Elemente zu bekommen. Kapitel 27 zeigt Ihnen, wie das funktioniert.
- **LINQ to DataSets** Funktioniert in etwa wie *LINQ to Objects*, nur dass eben DataSets abgefragt werden können und dass es zusätzliche Abfragemöglichkeiten gibt, bei der eine LINQ-Abfrage ein DataSet bzw. eine DataTable zurückliefert. Aus Platzgründen werden wir auch auf diese Technologie nicht näher eingehen; wenn Sie allerdings Erfahrungen mit ADO.NET haben, werden Sie nach dem Studium dieses und des nächsten Kapitels problemlos LINQ to DataSets einsetzen können.
- **LINQ to Entities** Funktioniert prinzipiell wie *LINQ to SQL*, ist aber in Sachen SQL eine andere, alternative Implementierung, die zusätzlich den Vorteil hat, dass nicht nur SQL Server der Abfrageempfänger sein muss, sondern es auch weitere auf ADO.NET basierende Provider dafür gibt oder geben wird. Darüber hinaus lassen sich bei diesem LINQ-Provider viel detailliertere Mappingszenarien zwischen dem verwendeten Objektmodell (konzeptionelles Modell) und dem Datenbankmodell (Speichermodell) konstruieren. LINQ to Entities finden Sie in Kapitel 28 beschrieben.

Wie »geht« LINQ prinzipiell

LINQ erweitert die .NET-Sprachen um Abfrageausdrücke. Was dabei genau abgefragt wird, ist Sache des jeweiligen LINQ-Providers – das haben wir im vergangenen Abschnitt schon erfahren. Das *Wie* ist dabei aber eher entscheidend, denn es ist für alle LINQ-fähigen Entitäten (Objects, XML, SQL etc.) gleich.

BEGLEITDATEIEN Im Verzeichnis

```
...\VB 2010 Entwicklerbuch\Kapitel 25\ErsteLinqDemo
```

finden Sie eine Konsolenbeispielanwendung, die mit Business-Objekten die Fähigkeiten von LINQ to Objects eindrucksvoll demonstriert. Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Relation zwischen Artikel und Käufer-Tabellen

ID	Nachname	Vorname	Straße	PLZ	Ort
1	Heckhuis	Jürgen	Wiedenbrücker Str. 47	59555	Lippstadt
2	Wördehoff	Angela	Erwitter Str. 33	01234	Dresden
3	Dröge	Ruprecht	Douglas-Adams-Str. 42	55544	Ratingen
4	Dröge	Ada	Douglas-Adams-Str. 42	55544	Ratingen
5	Halek	Gaby „Doc“	Krankenhausplatz 1	59494	Soest
6					

IdGekauftVon	ArtikelNummer	ArtikelName	Kategorie	Einzelpreis	Anzahl
1	9-445	Die Tore der Welt	Bücher, Belletristik	19,90	2
3	3-537	Visual Basic 2005 - Das Entwicklerbuch	Bücher, EDV	59,00	2
3	3-123	SQL Server 2000 - So gelingt der Einstieg	Bücher, EDV	19,90	1
5	5-312	SQL Server 2008 - Das Profi-Buch	Bücher, EDV	39,90	2

Abbildung 25.1 Das LINQ-Beispielprogramm legt zwei Tabellen an, die nur logisch über die Spalten-ID miteinander verknüpft sind

Das Programm konstruiert dabei folgendes Szenario: Es erstellt eine Kundenauflistung – die einzelnen Elemente dieser Auflistung werden dabei in entsprechenden Business-Objekten gespeichert, deren Schema genau dem der Tabelle aus Abbildung 25.1 entspricht. Es definiert ferner eine zweite Tabelle, die Daten mit gekauften Artikeln enthält. In dieser Tabelle ist jedoch der Name des Käufers nicht mehr enthalten; stattdessen gibt es in der Tabelle nur eine ID, über die man in der ersten Tabelle nachschlagen und so die eigentlichen Käuferdaten ermitteln kann.

Und jetzt kommt das eigentliche Problem bzw. die eigentliche Aufgabe: Das Programm soll die Daten so aufbereiten, dass die Kunden nicht nur hintereinander nach Nachnamen sortiert angezeigt werden; das Programm soll ebenfalls ermitteln, welcher Kunde wie viel Umsatz gemacht hat, und es soll die gekauften Artikel untereinander auflisten. Dabei soll es aber nur die Kunden berücksichtigen, die mehr als 1.000 Euro erzielt haben und in einem definierten Postleitzahlgebiet wohnen, etwa wie folgt:

```
5: Lehnert, Theo - 11 Artikel zu insgesamt 1064,10 Euro
     Details:
     1-234: Das Leben des Brian(3 Stück für insgesamt 254,85 Euro)
     1-234: Das Leben des Brian(1 Stück für insgesamt 14,95 Euro)
     3-123: Das Vermächstnis der Tempelritter(1 Stück für insgesamt 29,95 Euro)
     3-537: Visual Basic 2005 - Das Entwicklerbuch(2 Stück für insgesamt 169,90 Euro)
     3-537: Visual Basic 2005 - Das Entwicklerbuch(1 Stück für insgesamt 89,95 Euro)
     5-506: Visual Basic 2010 - Das Entwicklerbuch(1 Stück für insgesamt 79,95 Euro)
     5-518: Visual Basic 2008 - Neue Technologien - Crashkurs(1 Stück für insgesamt 4,95 Euro)
```

```
     7-321: Das Herz der Hölle(2 Stück für insgesamt 99,90 Euro)
     9-009: Die Wächter(2 Stück für insgesamt 39,90 Euro)
     9-009: Die Wächter(3 Stück für insgesamt 254,85 Euro)
     9-646: Was diese Frau so alles treibt(1 Stück für insgesamt 24,95 Euro)
10: Weigel, Momo - 14 Artikel zu insgesamt 1118,70 Euro
     Details:
      4-444: Harry Potter und die Heiligtümer des Todes(1 Stück für insgesamt 34,95 Euro)
     1-234: Das Leben des Brian(1 Stück für insgesamt 49,95 Euro)
     2-424: 24 - Season 6 [UK Import - Damn It!](2 Stück für insgesamt 89,90 Euro)
     2-424: 24 - Season 6 [UK Import - Damn It!](3 Stück für insgesamt 74,85 Euro)
     2-424: 24 - Season 6 [UK Import - Damn It!](1 Stück für insgesamt 4,95 Euro)
     3-534: Mitten ins Herz(3 Stück für insgesamt 59,85 Euro)
     3-537: Visual Basic 2005 - Das Entwicklerbuch(3 Stück für insgesamt 74,85 Euro)
     3-543: Microsoft Visual C# 2005 - Das Entwicklerbuch(1 Stück für insgesamt 44,95 Euro)
     3-543: Microsoft Visual C# 2005 - Das Entwicklerbuch(1 Stück für insgesamt 29,95 Euro)
     5-401: Programmieren mit dem .NET Compact Framework(1 Stück für insgesamt 29,95 Euro)
     7-321: Das Herz der Hölle(1 Stück für insgesamt 89,95 Euro)
     9-009: Die Wächter(3 Stück für insgesamt 179,85 Euro)
     9-423: Desperate Housewives - Staffel 2, Erster Teil(2 Stück für insgesamt 99,90 Euro)
     9-445: Die Tore der Welt(3 Stück für insgesamt 254,85 Euro)
```

LINQ ermöglicht es, aus zwei Auflistungen, die mit den Schemainformationen, wie sie auch in Abbildung 25.1 zu sehen sind, *eine* neue Auflistung zu erstellen, die das hier Gezeigte mit den genannten Konventionen einfach hintereinander ausgibt. Das entsprechende Listing dazu sieht folgendermaßen aus:

```
Module LinqDemo

    Sub Main()

        Dim adrListe = Kontakt.ZufallsKontakte(10)
        Dim artListe = Artikel.Zufallsartikel(adrListe)

        Dim adrListeGruppiert = From adrElement In adrListe
                                Join artElement In artListe
                                On adrElement.ID Equals artElement.IDGekauftVon
                                Select adrElement.ID, adrElement.Nachname,
                                       adrElement.Vorname, adrElement.PLZ,
                                       artElement.ArtikelNummer, artElement.ArtikelName,
                                       artElement.Anzahl, artElement.Einzelpreis,
                                       Postenpreis = artElement.Anzahl * artElement.Einzelpreis
                                Order By Nachname, ArtikelNummer
                                Where (PLZ > "0" And PLZ < "50000")
                                Group ArtikelNummer, ArtikelName,
                                      Anzahl, Einzelpreis, Postenpreis
                                By ID, Nachname, Vorname
                                Into Artikelliste = Group, AnzahlArtikel = Count(),
                                     Gesamtpreis = Sum(Postenpreis)
                                Where Gesamtpreis > 1000

        For Each KundenItem In adrListeGruppiert
            With KundenItem
                Console.WriteLine(.ID & ": " & .Nachname & ", " & .Vorname & " - " & _
                          .AnzahlArtikel & " Artikel zu insgesamt " & .Gesamtpreis & " Euro")
                Console.WriteLine("     Details:")
                For Each ArtItem In KundenItem.Artikelliste
```

```
                    With ArtItem
                        Console.WriteLine("     " & .ArtikelNummer & ": " & .ArtikelName & _
                                          "(" & .Anzahl & " Stück für insgesamt " _
                                          & (.Einzelpreis * .Anzahl).ToString("#,##0.00") & " Euro)")
                    End With
                Next
            End With

        Next
        Console.ReadKey()

    End Sub

End Module
```

Wenn Sie sich dieses Listing anschauen, dann geht es Ihnen wahrscheinlich wie jedem, der sich das erste Mal mit SQL-Abfragen oder – wie hier im Beispiel von LINQ – mit an SQL angelehnten Abfragen beschäftigt, und es stellen sich zwei Fragen:

- Wie formuliere ich eine Abfrage, um genau das Datenresultat zu bekommen, was ich mir vorstelle?
- Wie funktioniert LINQ, und wie gliedert der Compiler die LINQ-Abfrage in das Klassenkonzept und die Sprachelemente von .NET Framework ein?

Wir beginnen mit der Beantwortung der letzten Frage, denn die erste wird sich automatisch erledigen, wenn Sie verstanden haben, wie der Compiler diese Art von Abfragen verarbeitet und was im Grunde genommen dabei passiert. Sie müssen das nicht unbedingt wissen. Aber auf lange Sicht gesehen wird Ihnen das Wissen um die genaue Funktionsweise helfen, schneller und bessere Abfragen zu formulieren.

LINQ und LINQ-Abfragen leben von Lambda-Ausdrücken. Und um zu verstehen, wie LINQ funktioniert, sollten Sie Lambda-Ausdrücke (Kapitel 16), Generics (Kapitel 18 – die Nullable-Datentypen in Kapitel 19 sind auch einen Blick wert!), Auflistungen und gerade auch generische Auflistungen (Kapitel 20) verstanden haben.

Dabei sollten Sie im Auge behalten, dass LINQ für seine Ausdrücke eine Syntax benutzt, die zwar reservierte Wörter der SQL-Sprache benutzt (`Select`, `From` etc.), aber nicht T-SQL ist – auch kein T-SQL-Dialekt. Auch die Logik, die Art und Weise, wie Abfragen formuliert werden, weicht spätestens bei komplexen Abfragen doch sehr stark von SQL ab.

Erweiterungsmethoden als Basis für LINQ

Wie in Kapitel 19 schon erwähnt, basiert LINQ als allererstes auf einem Satz an Funktionen, die als Erweiterungsmethoden implementiert sind. Mit ihrer Hilfe können Entwickler bereits definierte Datentypen um benutzerdefinierte Funktionen erweitern, *ohne* eben einen neuen, vererbten Typ zu erstellen. Erweiterungsmethoden sind eine besondere Art von statischen Methoden, die Sie jedoch wie Instanzmethoden für den erweiterten Typ aufrufen können: Für in Visual Basic (und natürlich auch C#) geschriebenen Clientcode gibt es keinen sichtbaren Unterschied zwischen dem Aufrufen einer Erweiterungsmethode und den Methoden, die in einem Typ tatsächlich definiert sind.

Das Erweitern von vorhandenen Typen auf diese Weise mag ja für »Convenience-Zwecke« (siehe Kapitel 20, Abschnitt »Vereinfachung von Auflistungsinitialisierern durch Erweiterungsmethoden«) schon einigermaßen sinnvoll sein. Aber was die Infrastrukturschaffung für LINQ anbelangt, greift man auf einen coolen Trick zurück: Da Erweiterungsmethoden *alle* Typen als ersten Parameter aufweisen können, können ihnen auch Schnittstellen als Typ übergeben werden. Das führt aber zwangsläufig dazu, dass alle Typen, die diese Schnittstelle implementieren, in Folge auch mit den entsprechenden scheinbar zusätzlichen Methoden ausgestattet sind. In Kombination mit Generics und Lambda-Ausdrücken ist das natürlich eine coole Sache, da die eigentliche Funktionalität durch – je nach Ausbaustufe der Überladungsversionen der statischen Erweiterungsmethoden – einen oder mehrere Lambda-Ausdrücke gesteuert wird und der eigentlich zu verarbeitende Typ eben durch die Verwendung von Generics erst beim späteren Typisieren (Typ einsetzen) ans Tageslicht kommt. Mit diesem Kunstgriff schafft man im Handumdrehen eine komplette, typsichere Infrastruktur beispielsweise für das Gruppieren, Sortieren, Filtern, Ermitteln oder sich gegenseitige Ausschließen von Elementen aller Auflistungen, die eine bestimmte, um Erweiterungsmethoden ergänzte Schnittstelle implementieren – beispielsweise bei `IEnumerable(T)` um die statischen Methoden der Klasse `Enumerable`. Oder kurz zusammengefasst: *Alle* Auflistungen, die `IEnumerable(T)` einbinden, werden scheinbar um Methoden ergänzt, die in `Enumerable` zu finden sind, und das sind genau die, die für *LINQ to Objects* benötigt werden, um Selektionen, Sortierungen, Gruppierungen und weitere Funktionalitäten von Daten durchzuführen. Es ist aber nur scheinbar so – die ganze Funktionalität von LINQ wird letzten Endes in eine rein prozedural aufgebaute Klasse mit einer ganzen Menge statischer Funktionen delegiert.

Bei Abfragen, die von anderen als *LINQ to Object*-Providern ausgeführt werden, funktioniert das ähnlich. Hier kommt allerdings die Schnittstelle `IQueryable(Of T)` zum Einsatz, und bei der Verwendung der Erweiterungsmethoden, aus deren Kombinationen sich eine Abfrage zusammensetzt, werden so genannte Ausdrucksbäume aufgebaut. Diese enthalten anschließend, vereinfacht ausgedrückt, für jede Abfrage eine Art To-do-Liste, aus der der entsprechende Provider dann beispielsweise eine neue Abfrage erstellen kann, die dann wiederum beispielsweise von Microsoft SQL Server oder von einem Oracle-Server-System verstanden werden kann.

Bevor wir uns im Detail mit der LINQ-typischen Abfragesyntax beschäftigen, wollen wir als Erstes diese Erweiterungsmethoden für LINQ to Objects, also die auf Basis von `IEnumerable(Of T)`, ein wenig genauer unter die Lupe nehmen.

Dabei sei Folgendes angemerkt: Die kommenden Abschnitte beschreiben nur ein paar der Erweiterungs-Methoden beispielhaft, um die Funktionsweise zu erklären; sie erheben aber natürlich keinen Anspruch auf Vollständigkeit. Vielmehr geht es in den Abschnitten hauptsächlich darum, ein Verständnis für den Kontext zu vermitteln, in dem diese Methoden später bei der Verwendung der konkreten LINQ-Syntax durch den Visual Basic-Compiler zum Einsatz gelangen, und Sie damit auf das folgende Kapitel vorbereiten.

Die Where-Methode

Das folgende Listing demonstriert eine der wichtigsten Erweiterungsmethoden der `IEnumerable`-Schnittstelle: die `Where`-Methode. Da auch normale Arrays `IEnumerable` implementieren, steht `Where` damit auch »normalen« Arrays zu Verfügung. Sie iteriert automatisch durch alle Elemente und ruft für jedes Element einen `Predicate`-Delegaten (mehr dazu finden Sie in der Beschreibung von `List(Of Type)` in Kapitel 20) auf. Anstelle einer Delegatenvariablen kann hier natürlich auch ein Lambda-Ausdruck angegeben werden. Wird dieser Ausdruck als *wahr* ausgewertet – trifft die entsprechende Bedingung also zu –, wird das betroffene Element in eine neue Ergebnisliste überführt, die vom generischen Typ `IEnumerable` des entsprechenden Quelltyps ist: Im ersten Beispiel ist der Quelltyp ein `Integer`, im zweiten ein `String`:

```
    Public Sub WhereBeispiel()
        ' Where iteriert durch alle Elemente der angegebenen
        ' Auflistung und führt für jedes Element die Prüfung in der Lambda-Funktion
        ' durch. Trifft sie zu, wird das entsprechende Element
        ' in die neue Auflistung überführt.
        ' Hier der erste Versuch mit Integer-Elementen ...
        Dim fnums = ziffern.Where(Function(einIntElement) einIntElement < 6)
        Console.WriteLine("Ziffern < 6")
        For Each intElement In fnums
            Console.WriteLine(intElement)
        Next
        Console.WriteLine("-----------------")

        ' ... und hier ein weiterer mit Strings.
        Dim fStrings = Namen.Where(Function(einName) einName.StartsWith("K"c))
        Console.WriteLine("Namen die mit 'K' beginnen")
        For Each strElement In fStrings
            Console.WriteLine(strElement)
        Next
    End Sub
```

Die Select-Methode

Weiter geht es mit der `Select`-Methode. Diese ist in der Lage, ein Element der einen Auflistung gegen das entsprechende Element einer anderen Auflistung anderen Typs aufgrund seiner Ordinalnummer auszutauschen. Auch hierzu ein Beispiel. Wir haben zwei Arrays, nämlich die folgenden beiden:

```
    Private ziffern As Integer() = New Integer() {9, 1, 4, 2, 5, 8, 6, 7, 3, 0}
    Private ziffernNamen As String() = New String() {"null", "eins", "zwei", "drei",
                                                     "vier", "fünf", "sechs", "sieben",
                                                     "acht", "neun"}
```

Mithilfe der `Select`-Anweisung können wir nun eine Auflistung aufbauen, die aufgrund der Inhalte der Elemente des ersten Arrays die Elemente des zweiten Arrays findet. Die folgende Methode gibt daher …

```
    Public Sub SelectBeispiel()
        ' Hier wird Select verwendet, um ein Element auf Grund seiner
        ' Wertigkeit durch ein zweites in einer anderen Auflistung zu ersetzen.
        Dim ziffernListe = ziffern.Select(Function(einIntElement) ziffernNamen(einIntElement))

        Console.WriteLine("Ziffern:")
        For Each s As String In ziffernListe
            Console.WriteLine(s)
        Next
    End Sub
```

… das folgende Ergebnis aus …

```
Ziffern:
neun
eins
vier
zwei
fünf
acht
```

```
sechs
sieben
drei
null
```

… das exakt der Reihenfolge der Inhalte des ersten Arrays entsprach.

Anonyme Typen

Nun können wir dieses Spielchen mit der `Select`-Methode noch ein kleines bisschen weitertreiben und lernen in diesem Zusammenhang das erste Mal den sinnvollen Einsatz anonymer Typen kennen. Anonyme Typen sind Klassen, die keinen Namen haben und direkt als Rückgabetypen – beispielsweise im Rahmen einer `Select`-Methode – definiert werden können.

Nun werden Sie denken: Wenn der Typ keinen Namen hat – wie soll ich dann eine Variable definieren von diesem nicht benannten Typen oder gar eine Auflistung, die auf Basis eines anonymen Typs definiert wird? Möglich wird das auf Basis von lokalen Typrückschlüssen (siehe Kapitel 2) und Generics (Kapitel 18). Wir erinnern uns: Lokaler Typrückschluss bewirkt, dass Typen aufgrund ihrer initialen Zuweisung während der Deklarierung definiert werden – und das funktioniert natürlich auch für Typen, die keinen Namen haben, anonyme Typen eben (Abbildung 25.2).

```
Public Sub AnonymeKlasseDemo()

    Dim element = New With {.Elementname = "Ein Element", _
        Dim element As <anonymer Typ>    .ElementID = 42}

    element.
            ElementID
            Elementname

End Sub
```

Abbildung 25.2
Anonyme Klassen erlauben das Definieren von Typen, die keine Namen haben. Dank lokalem Typrückschluss lassen sich solche Typen aber dennoch an Objektvariablen zuweisen.

Typrückschluss für generische Typparameter

Für unser erweitertes `Select`-Beispiel brauchen wir eine weitere Technik, die sich Typrückschluss für generische Typparameter nennt. Dabei geht es darum, dass generische Funktionen ihre Typparameter über die ihnen tatsächlich übergebenen Parameter rückschließen, beispielsweise wie es in der folgenden Methode passiert:

```
Public Function ListeMitEinemElement(Of TSource)(ByVal dasEineElement As TSource) As
                                                                        List(Of TSource)
    Dim lst As New List(Of TSource)
    lst.Add(dasEineElement)
    Return lst
End Function
```

Über den Sinn oder Unsinn dieser Routine kann man natürlich streiten, denn ihre Aufgabe ist folgende: Ihr wird eine Instanz beliebigen Typs übergeben, und die Methode macht daraus eine Auflistung, der sie exakt dieses Element hinzufügt. Es geht dabei auch weniger um die Aufgabe der Methode, sondern wie ihr Parameter übergeben werden können. Um diese Routine ohne Typrückschluss für generische Typparameter zu verwenden, müssten Sie sie folgendermaßen verwenden:

```
Dim dasEineElement As Integer = 10
Dim liste As List(Of Integer) = ListeMitEinemElement(Of Integer)(dasEineElement)
```

Nun wird `dasEineElement` als `Integer` definiert, und die Tatsache, dass Sie der Methode eine `Integer`-Variable übergeben, reicht dem Compiler, die Information zu ergänzen, um aus der generischen Methode eben eine vom Typ `Integer` zu machen. Ausreichend ist also:

```
Dim dasEineElement As Integer = 10
Dim liste As List(Of Integer) = ListeMitEinemElement(dasEineElement)
```

Dank Typrückschluss kann es jetzt noch einen Schritt weitergehen: Die im oben gezeigten Listing angegebenen Typdefinitionen sind eigentlich auch redundant: Die Konstante 10 definiert `dasEineElement` ausreichend als `Integer`. Und damit funktioniert der Typrückschluss für Generics natürlich auch bei der Zuweisung der zurückgegebenen Auflistung. Damit ergibt sich folgende, immer noch typmäßig ausreichend aufschlussreiche Sparversion des Beispiels:

```
Dim dasEineElement = 10
Dim liste = ListeMitEinemElement(dasEineElement)
```

Und da wir auf diese Weise sämtliche explizite Typangaben eliminieren konnten, funktioniert diese Sparversion natürlich auch für anonyme Typen, wie Abbildung 25.3 zeigt.

```
Dim liste = ListeMitEinemElement(New With {.ID = 42, .Elementname = "Adams"})
    Dim liste As System.Collections.Generic.List(Of <anonymer Typ>)
```

Abbildung 25.3 Ist der übergebene Parameter ein anonymer Typ, wird auch der generische Typparameter der generischen Methode anonym – genau wie die Auflistung, die die Methode in dessen Abhängigkeit zurückliefert

Mit diesem Wissen ist das Verständnis der folgenden Routine ebenfalls kein Problem mehr. Wenn Sie das folgende Beispiel laufen lassen …

```
Public Sub SelectBeispielAnonym()
    ' Hier wird Select verwendet, um ein Element auf Grund seiner
    ' Wertigkeit durch ein zweites, anonymes in einer anderen Auflistung zu ersetzen.
    Dim anonymeListe = ziffern.Select(Function(einIntElement) New With
                                          {.OriginalWert = einIntElement,
                                                   .Ziffernname = ziffernNamen(einIntElement)})
    Console.WriteLine("Inhalt der anonymen Klassenelemente:")
    For Each anonymerTyp In anonymeListe
        Console.WriteLine(anonymerTyp.OriginalWert & ", " & anonymerTyp.Ziffernname)
    Next
End Sub
```

erhalten Sie das folgende Ergebnis:

```
Inhalt der anonymen Klassenelemente:
9, neun
1, eins
4, vier
```

```
2, zwei
5, fünf
8, acht
6, sechs
7, sieben
3, drei
0, null
```

Und für diejenigen, die es genau interessiert, wie Methoden wie `Select` implementiert werden, sei die folgende Methode als Beispiel gezeigt, mit deren Hilfe sich Elemente einer Auflistung beliebigen Typs nummerieren lassen:

```
    Public Function ItemNumberer(Of TSource, TResult)(ByVal source As IEnumerable(Of TSource),
                        ByVal selector As Func(Of TSource, Integer, TResult)) As IEnumerable(Of TResult)

        Dim retColl As New List(Of TResult)
        Dim count = 1
        For Each item In source
            retColl.Add(selector(item, count))
            count += 1
        Next
        Return retColl.AsEnumerable
    End Function
```

Diese Methode übernimmt eine generische `IEnumerable`-Auflistung eines beliebigen Typs und einen Lambda-Ausdruck und liefert eine `IEnumerable`-Auflistung zurück, die durch den Rückgabewert des Lambda-Ausdrucks definiert wurde. Ziel ist es, durch alle Elemente der Quellauflistung hindurchzuiterieren und der aufrufenden Methode über den Lambda-Ausdruck die Möglichkeit zu geben, eine aktuelle Elementnummer (`count`) in die Zielauflistung einzubauen. Mithilfe eines anonymen Typs könnte die Verwendung dieser Methode folgendermaßen aussehen:

```
    Public Sub ItemNumbererDemo()
        Dim numList = ItemNumberer(New String() {"eins", "zwei", "drei", "vier", "fünf",
                                                 "sechs", "sieben", "acht", "neun", "zehn"},
                                   Function(element, count) New With {.ID = count,
                                                                      .Element = element})

        For Each numListItem In numList
            Console.WriteLine(numListItem.ID & ": " & numListItem.Element)
        Next

    End Sub
```

Auch bei Aufruf dieser Methode wird wieder mit Rückschließen generischer Typparameter gearbeitet. Die Methode `ItemNumberer` wird nicht mit (`Of Irgendwas, Irgendwas`) genauer spezifiziert. Stattdessen leitet sich der Typ `TSource` durch das übergebene `String`-Array und der Rückgabewert durch den übergebenen, anonymen Typ ab. Die anschließende Ausführung der Methode ergibt folgendes Ergebnis:

```
1: eins
2: zwei
3: drei
4: vier
```

```
5: fünf
6: sechs
7: sieben
8: acht
9: neun
10: zehn
```

Kombinieren von LINQ-Erweiterungsmethoden

Nachdem Sie nunmehr ein paar der wichtigsten LINQ-Erweiterungsmethoden kennen gelernt haben, lassen Sie uns als Nächstes einen Blick darauf werfen, wie sich diese Methoden miteinander vertragen.

Wie wir gelernt haben, geben alle diese Methoden Auflistungen zurück. Und diese Auflistungen haben alle eines gemeinsam: Sie implementieren, genau wie ihre Quellauflistungen – also die, aus denen sie entstanden sind – `IEnumerable(Of T)`. Das bedeutet aber weiterhin, dass die Ergebnisse der Erweiterungsmethoden sich wieder als Argument für eine weitere LINQ-Erweiterungsmethode nutzen lassen. Ein Beispiel soll auch dies verdeutlichen:

```
    Public Sub KombinierteErweiterungsmethoden()

        Dim ergebnisListe = Kontakte.Where(Function(einKontakt) einKontakt.Ort = "Lippstadt"
                                           ).Select(Function(einKontakt) New With {.ID =
einKontakt.ID,
                                                    .LastName = einKontakt.Nachname,
                                                    .FirstName = einKontakt.Vorname,
                                                    .City = einKontakt.Ort}
                                                    ).OrderBy(Function(Contact) Contact.LastName)

        For Each contactItem In ergebnisListe
            Console.WriteLine(contactItem.ID & ": " & _
                        contactItem.LastName & ", " & _
                        contactItem.FirstName & " living in " & _
                        contactItem.City)

        Next
    End Sub
```

Wenn Sie die Erweiterungsmethoden dieses Beispiels in dieser Reihenfolge nacheinander angewendet sehen, ahnen Sie bestimmt bereits, was bei LINQ-Abfragen, wie Sie sie am Anfang dieses Kapitels kennen gelernt haben, wirklich passiert.

Aber lassen Sie uns nicht zwei Schritte auf einmal machen, sondern die Methodenkombination, die Sie im obigen Listing fett hervorgehoben sehen, erklärungstechnisch Schritt für Schritt auseinander nehmen:

- Als Erstes verarbeitet das Programm die Ausgangsliste `Kontakte` mithilfe der `Where`-Erweiterungsmethode. In diesem konkreten Beispiel generiert die `Where`-Methode eine neue, `IEnumerable(Of T)`-implementierende Auflistung, die nur jene Elemente enthält, deren Ort der Zeichenkette »Lippstadt« entspricht.
- Das Ergebnis dieser Auflistung ist Gegenstand der nächsten Erweiterungsmethode: `Select`. Diese macht nichts anderes, als ein Element der ersten Auflistung durch ein anderes Element auszutauschen und dieses neue Element in einer abermals neu erstellten Auflistung einzugliedern. Die `Select`-Methode dieses konkreten Beispiels erstellt dazu eine anonyme Klasse, die prinzipiell den alten Elementen ähnlich

ist, aber englische Eigenschaftenbezeichnungen enthält. Die alten Objekte, auf Basis der Klasse `Kontakte`, werden also als Basis für eine Reihe von neuen Objekten verwendet, die von der `Select`-Methode nacheinander angelegt werden. Wir erhalten so eine Auflistung mit anonymen Klasseninstanzen, die sich durch englische Eigenschaftenbezeichner auszeichnen.

- Diese Auflistung wird abermals Gegenstand einer Methode – dieses Mal ist es die Erweiterungsmethode `OrderBy`. Sie nimmt nun die anonymen Klasseninstanzen mit den englischen Eigenschaftenbezeichnern und sortiert diese nach Nachnamen (`Contact.LastName`) in einer wiederum neuen Auflistung ein.
- Diese zuletzt generierte Auflistung ist schließlich die Ergebnisauflistung, die der Objektvariablen `ergebnisListe` zugewiesen und in der anschließenden `For Each`-Schleife ausgegeben wird:

```
4: Belke, Andreas living in Lippstadt
2: Dröge, Ruprecht living in Lippstadt
11: Sonntag, Miriam living in Lippstadt
7: Vielstedde, Anja living in Lippstadt
```

Vereinfachte Anwendung von LINQ – Erweiterungsmethoden mit der LINQ-Abfragesyntax

O.k. – viel zu sagen gibt es jetzt nicht mehr, ich hoffe, dass Ihnen – um es mit einer Metapher zu sagen – mein vielleicht etwas von hinten aufgezäumtes Pferd didaktisch genehm war.

Ein Blick auf das folgende Listing reicht meiner Meinung nach jetzt aus, um zu verstehen, was der Visual Basic-Compiler aus einem LINQ-Ausdruck macht und wie er ihn – jedenfalls im Fall von *LINQ to Objects* – in eine Kombination aus Erweiterungsmethoden umsetzt.

Die Abfrage des folgenden Listings entspricht nämlich exakt …

```
Public Sub KombinierteErweiterungsmethoden_à_la_LINQ()

    Dim ergebnisListe = From einKontakt In Kontakte
                        Where einKontakt.Ort = "Lippstadt"
                        Select ID = einKontakt.ID,
                                LastName = einKontakt.Nachname,
                                FirstName = einKontakt.Vorname,
                                City = einKontakt.Ort
                        Order By LastName

    For Each contactItem In ergebnisListe
        Console.WriteLine(contactItem.ID & ": " & _
                    contactItem.LastName & ", " & _
                    contactItem.FirstName & " living in " & _
                    contactItem.City)

    Next

End Sub
```

… der Methodenkombination des Listings im vorherigen Abschnitt. Und natürlich liefert es auch das exakt gleiche Ergebnis – probieren Sie es aus!

Kapitel 26

LINQ to Objects

In diesem Kapitel:

Einführung in LINQ to Objects 764
Der Aufbau einer LINQ-Abfrage 764
Performance von LINQ-Abfragen 770
Kombinieren und verzögertes Ausführen von LINQ-Abfragen 771
Verbinden mehrerer Auflistungen zu einer neuen 778
Gruppieren von Auflistungen 781
Aggregatfunktionen 784

Einführung in LINQ to Objects

Nach der Lektüre des vorherigen Kapitels wissen Sie ja im Grunde genommen schon, wie LINQ to Objects funktioniert, was gibt es also in Sachen Einführung zu LINQ to Objects noch zu sagen? – Was die generelle, interne Funktionsweise anbelangt, im Grunde gar nichts. Was die Auswirkungen von LINQ to Objects auf Ihre tägliche Routine anbelangt, vielleicht eine ganze Menge!

Letzten Endes geht es nämlich beim Einsatz von LINQ to Objects nicht primär darum, möglichst eine Datenbank mit Business-Objekten quasi im Managed Heap Ihrer .NET-Anwendung nachzubilden – auch wenn die Beispiele dieses Kapitels das vielleicht vermuten lassen.

Nein? – Nein!

LINQ to Objects soll es Ihnen ermöglichen, die im letzten Kapitel vorgestellte, an die SQL-Abfragesprache angelehnte Syntax auch für ganz andere Zwecke, an die Sie im Moment vielleicht noch gar nicht denken, in Ihre Programme einzubauen. Doch dazu müssen Sie wissen, mit welchen Komponenten einer Abfrage Sie welche Ergebnisse erzielen können – und dabei soll Ihnen dieses Kapitel behilflich sein.

Aus diesem Grund wählen wir als Beispielträger auch einfachste Objektauflistungen (übrigens exakt die des letzten Kapitels), damit Sie sich bei den Beispielen auf LINQ konzentrieren können und nicht von Funktionalitäten anderer Auflistungsklassen des Systems abgelenkt werden. Dennoch können Sie LINQ nicht nur für Ihre selbst geschriebenen Business-Klassen und –Auflistungen einsetzen. Natürlich lassen sich auch Verzeichnis-, Grafik- oder Steuerelementauflistungen mit LINQ bearbeiten und organisieren.

HINWEIS Dieses Kapitel kann übrigens dabei keinen Anspruch auf Vollständigkeit erheben – dazu ist das Thema LINQ einfach zu mächtig und umfangreich. Es wird Ihnen aber auf jeden Fall ausreichend Informationen, Beispiele und Anregungen zum Thema liefern, sodass Sie Ihr Wissen durch eigenes Ausprobieren und aufmerksames Lesen der Onlinehilfe sicherlich perfektionieren können.

Aber: Ich möchte Ihnen an dieser Stelle die Lektüre eines Buches empfehlen, das Sie umfassend über das Thema LINQ informiert. Jim Wooley, ein sehr netter MVP-Kollege, hat mit einigen anderen Autoren das Buch »LINQ im Einsatz« verfasst, in dem Sie alle Infos finden, die Sie über diese Kapitel hinaus zum Thema benötigen. Sie finden alle Infos, die Sie benötigen, unter diesem Link: *http://tinyurl.com/33jvb6h*.

Der Aufbau einer LINQ-Abfrage

LINQ-Abfragen beginnen stets[1] mit der Syntax

```
From bereichsVariable In DatenAuflistung
```

und offen gesagt: Wenn ich auch die Syntax aus Programmiersicht nachvollziehen konnte, so verstand ich zunächst nicht, warum es unter rein sprachlichen Aspekten ausgerechnet `From` heißen musste – was wird denn da eigentlich *von* der Bereichsvariablen genommen, entwendet oder verwendet?

[1] Ausnahmen bilden reine Aggregatabfragen, die auch mit der Klausel `Aggregate` beginnen, wie im entsprechenden Abschnitt dieses Kapitels beschrieben.

Das Visual Basic-Team hat darauf folgende Antwort: *»From« deutet auf Quelle oder Ursprungsort hin. Der* `From`*-Ausdruck als solcher ist die Stelle, an der man die Abfragequelle bestimmt (beispielsweise den Datenkontext, eine Auflistung im Speicher, XML etc.). Aus diesem Grund fanden wir diesen Terminus angemessen. Darüber hinaus handelt es sich bei »From« auch um ein SQL-Schlüsselwort. Wir haben uns bei LINQ um größtmöglichen SQL-Erhalt bemüht, sodass LINQ-Neueinsteiger, die bereits über SQL-Erfahrung verfügten, die Ähnlichkeit bereits »erfühlen« konnten.*

Ein interessanter Unterschied zwischen LINQ und SQL, den wir jedoch einbauen mussten, ist die Reihenfolge der Elemente innerhalb des Abfrageausdrucks. Bei SQL steht `Select` *vor* `From`*. In LINQ ist es genau umgekehrt.*

Einer der großen Vorteile von LINQ ist, dass man mit verschiedenen Datenquellen in einem gemeinsamen Modell arbeiten kann. Innerhalb einer Abfrage arbeitet man nur mit Objekten, und wir können IntelliSense für diese Objekte zur Verfügung stellen. Der Grund dafür, dass wir `From` *als Erstes benötigen ist, dass wir als Erstes wissen müssen, von woher die Quelle stammt, über die wir die Abfrage durchführen, bevor wir die IntelliSense-Information zur Verfügung stellen können.*[2]

Sprachlich zu verstehen ist `From` also im Kontext des gesamten ersten Abfrageteilausdrucks – also quasi `From (bereichsVariable In Auflistung)` – und bezieht sich *nicht* nur auf die Bereichsvariable, die hinter `From` steht. Und nachdem dieser Ausdruck nun auch sprachlich geklärt ist, fassen wir zusammen:

`From` leitet eine LINQ-Abfrage ein, und er bedeutet ins »Menschliche« übersetzt: »Hier, laufe mal bitte durch alle Elemente, die hinter dem Schlüsselwort `In` stehen, und verwende die Bereichsvariable hinter `From`, um mit deren Eigenschaften bestimmen zu können, wie die Daten der Auflistung selektiert, sortiert, gruppiert und dabei in eine neue Elementliste (mit – bei Bedarf – Instanzen anderer Elementklassen) überführt werden können.

WICHTIG Wichtig für die Profi-T-SQL-Fraktion unter Ihnen ist übrigens: LINQ-Abfragen beginnen niemals mit `Select`, ja, sie müssen noch nicht einmal ein `Select` aufweisen. `DELETE`, `INSERT` und `UPDATE` gibt es im Übrigen auch nicht – auch nicht wie im übernächsten Kapitel zu lesen, in *LINQ to Entities*.

Mit dem Wissen des vorherigen Kapitels schauen wir uns jetzt mal die folgenden beiden Abfragen an und überlegen uns, wie sie sich im Ergebnis unterscheiden.

BEGLEITDATEIEN Im Verzeichnis

```
...\VB 2010 Entwicklerbuch\Kapitel 26\LinqToObjectSamples
```

finden Sie eine Konsolenbeispielanwendung, die mit Business-Objekten weitere Möglichkeiten von *LINQ to Objects* erklären soll. Das Programm entspricht wieder dem aus dem letzten Kapitel: Es generiert per Zufallsgenerator zwei Auflistungen mit Business-Objekten (eine Kunden-, eine Artikelbestellauflistung), die logisch zueinander in Relation stehen. Das vorherige Kapitel erklärt mehr zum Aufbau des Beispiels.

[2] Quelle: Lisa Feigenbaum E-Mail vom 01.02.2008. Lisa Feigenbaum ist Kollegin der Co-Autorin dieses Buchs, Sarika Calla, und ebenfalls im Visual Basic-Team.

O.k. Wir schauen uns zunächst das folgende Listing des Beispielprogramms an, und das schaut so aus:

```
Sub LINQAbfragenAufbau()

    Console.WriteLine("Ergebnisliste 1:")
    Console.WriteLine("----------------")

    Dim ergebnisListe = From adrElement In adrListe
                        Order By adrElement.Nachname, adrElement.Vorname
                        Select KontaktId = adrElement.ID,
                               adrElement.Nachname,
                               adrElement.Vorname,
                               PlzOrtKombi = adrElement.PLZ & " " & adrElement.Ort
                        Where KontaktId > 50 And KontaktId < 100 And
                                Nachname Like "L*"

    For Each ergebnisItem In ergebnisListe
        With ergebnisItem
            Console.WriteLine(.KontaktId & ": " & _
                              .Nachname & ", " & .Vorname & _
                              " - " & .PlzOrtKombi)
        End With
    Next

    Console.WriteLine()
    Console.WriteLine("Ergebnisliste 2:")
    Console.WriteLine("----------------")


    Dim ergebnisListe2 = From adrElement In adrListe
                Where adrElement.ID > 50 And adrElement.ID < 100 And
                        adrElement.Nachname Like "L*"
                Select KontaktId = adrElement.ID,
                       adrElement.Nachname,
                       adrElement.Vorname,
                       PlzOrtKombi = adrElement.PLZ & " " & adrElement.Ort
                Order By Nachname, Vorname

    For Each ergebnisItem In ergebnisListe2
        With ergebnisItem
            Console.WriteLine(.KontaktId & ": " & _
                              .Nachname & ", " & .Vorname & _
                              " - " & .PlzOrtKombi)
        End With
    Next
End Sub
```

Sie sehen, dass sich die Abfragen syntaktisch und damit auf den ersten Blick schon einmal gar nicht gleichen. Und dennoch, wenn Sie das Beispiel starten, so sehen Sie ein Ergebnis etwa wie im folgenden Bildschirmauszug…

```
Ergebnisliste 1:
----------------
81: Langenbach, Barbara - 78657 Wiesbaden
76: Langenbach, Klaus - 61745 Dortmund
54: Lehnert, Katrin - 82730 Wiesbaden
96: Löffelmann, Barbara - 63122 Rheda
68: Löffelmann, Gabriele - 54172 Bad Waldliesborn
```

```
Ergebnisliste 2:
----------------
81: Langenbach, Barbara - 78657 Wiesbaden
76: Langenbach, Klaus - 61745 Dortmund
54: Lehnert, Katrin - 82730 Wiesbaden
96: Löffelmann, Barbara - 63122 Rheda
68: Löffelmann, Gabriele - 54172 Bad Waldliesborn
```

… und obwohl diese Liste laut Codelisting definitiv auf zwei verschiedenen Ergebnisauflistungen beruht, die aus zwei verschiedenen Abfragen entstanden sind, ist das Ergebnis dennoch augenscheinlich dasselbe. Zufall? Keinesfalls – im Gegenteil, pure Absicht.

Denn dieses Beispiel demonstriert sehr schön, wie LINQ-Abfragen im Code funktionieren und wie sehr sie im Grunde genommen nichts mit den klassischen SQL-Datenbankabfragen zu tun haben. Gut, zugegeben, Ähnlichkeiten sind vorhanden, aber das war es auch schon.

Nun lassen Sie uns mal beide Abfragen auseinander nehmen und dabei schauen, wieso so unterschiedlich Formuliertes dennoch zum gleichen Ergebnis führen kann. Los geht es:

- Die erste Abfrage startet, wie jede LINQ-Abfrage, mit der `From`-Klausel, und diese definiert `adrElement` als *Bereichsvariable* für alle kommenden Parameter bis, so vorhanden, zum ersten `Select`. Die Bereichsvariable ist also bis zum ersten `Select`-Befehl die Variable, mit der quasi intern durch die komplette Auflistung hindurchiteriert wird und an der damit auch die Eigenschaften bzw. öffentlichen Felder der Auflistungselemente »hängen«.
- Es folgt hier im Beispiel (aber natürlich nicht notwendigerweise) die Sortieranweisung `Order By`, der die Felder bzw. Eigenschaften, nach denen die Elemente der Auflistung sortiert werden sollen, als Argumente über die Bereichsvariable übergeben werden: Das sind in diesem Beispiel `adrElement.Nachname` und `adrElement.Vorname`. Die Bereichsvariable dient also dazu, auf die Eigenschaften zuzugreifen, über die durch die `Order By`-Klausel festgelegt werden kann, nach welchen Feldern sortiert wird.

TIPP Wenn nichts anderes gesagt wird, sortiert `Order By` *aufsteigend*. Durch die Angabe des Schlüsselwortes `Descending` können Sie den Sortierausdruck abändern, sodass die Elementauflistung *absteigend* sortiert wird. Wollte man – um beim Beispiel zu bleiben – die erste Liste nach Namen aufsteigend und nach Vornamen absteigend sortieren, hieße der LINQ-Abfrageausdruck folgendermaßen:

```
Dim ergebnisListe = From adrElement In adrListe
                    Order By adrElement.Nachname, adrElement.Vorname Descending
                    Select KontaktId = adrElement.ID,
                           adrElement.Nachname,
                           adrElement.Vorname,
                           PlzOrtKombi = adrElement.PLZ & " " & adrElement.Ort
                    Where KontaktId > 50 And KontaktId < 100 And
                            Nachname Like "L*"
```

- Jetzt folgt ein `Select`, und Achtung: Wir haben ja im letzten Kapitel erfahren, dass `Select` dazu dient, ein Element einer Auflistung durch ein anderes zu ersetzen, das in einer anderen Auflistung »landet«. Und dieses zu ersetzende Element wird eine Instanz einer neuen anonymen Klasse mit exakt den Feldern sein, die Sie hinter dem `Select` bestimmen.

TIPP Spätestens hier sehen Sie, dass ein `Select` für eine Abfrage, die sich auf nur eine Tabelle bezieht und die keine Gruppierungen durchführt, eigentlich ein reiner Zeitkiller ist, denn Sie können natürlich mit *den* Ausgangselementen weiterarbeiten, die *ohnehin* schon da sind und in der späteren Ergebnisliste *schlimmstenfalls* ein paar redundante Informationen aufweisen. Anders als bei SQL-Abfragen benötigen Sie das `Select`-Schlüsselwort bei LINQ in Visual Basic[3] *nicht*, um Abfragen für Filterungen, Selektierungen oder Gruppierungen einzuleiten, sondern *nur*, um eine *neue Auflistung* mit Elementen zu erstellen, die aus einem neuen, anonymen Typ bestehen, der nur die Eigenschaften offen legt, die Sie als Feldnamen angeben. Beherzigen Sie deswegen auch das, was der nächste Abschnitt über `Select` zu berichten weiß!

- Nachdem wir nun das `Select` hinter uns gelassen und der Abfrage-Engine damit mitgeteilt haben, dass wir nur noch mit `KontaktId`, `Nachname`, `Vorname` und einem `PlzOrtKombi`-Feld weitermachen wollen, können sich weitere LINQ-Abfrageelemente auch nur noch auf diese, durch `Select` neu eingeführte anonyme Klasse beziehen – dementsprechend würde hinter der `Where`-Klausel ein `adrElement.Id` als Argument nicht mehr funktionieren; `Select` hat `adrElement` der Auflistung schließlich durch Instanzen einer anonymen Klasse ersetzt, und die haben nur noch die `KontaktID`-Eigenschaft. `Where` dient jetzt dazu, die neue Auflistung mit den Elementen der anonymen Klasse zu filtern: Nur Elemente mit `KontaktID > 50` und `KontaktID < 100` sind mit von der Partie sowie die Elemente, deren Nachname mit »L« beginnt.

Damit ist die erste Abfrage durch. Und jetzt schauen wir uns die zweite an.

- Hier geht es nach der obligatorischen Festlegung von Auflistung und Bereichsvariable, die als Elementiterator dienen soll, zunächst los mit der `Where`-Klausel, die die Elemente auf jene IDs einschränkt, die größer als 50 und kleiner als 100 sind. Die Einschränkung wird dann noch weitergeführt, nämlich auf Nachnamen, die mit dem Buchstaben »L« beginnen. Hier kann `Where` direkt auf die Eigenschaften zurückgreifen, die über die Bereichsvariable erreichbar sind, denn sie entsprechen einem Element der Ausgangsauflistung.
- Das geht so lange, bis es auch hier wieder ein `Select` gibt, das dem ein Ende setzt. Auch hier wird wieder ein Ersetzungsvorgang durchgeführt – unnötigerweise und nur zu Demonstrationszwecken –, der die Elemente der vorhandenen Auflistung durch neue einer anonymen Klasse ersetzt, die wiederum die angegebenen Eigenschaften haben.
- Das `Where` schränkt diese Auflistung wieder ein, und zwar prinzipiell in gleicher Weise wie im ersten Beispiel – lediglich der Zeitpunkt, wann welche Elemente wie eingeschränkt werden, ist ein anderer.

Eines ist in diesem Zusammenhang sehr wichtig zu erwähnen:

WICHTIG LINQ-Abfragen werden immer verzögert ausgeführt und niemals direkt. Das bedeutet: Nicht die eigentliche Abfrage ist ausschlaggebend für das »Anstoßen« der Verarbeitung, sondern erst der erste Zugriff auf die Ergebnisliste löst das Ausführen der eigentlichen Abfrage aus. Mehr dazu erfahren Sie im Abschnitt »Performance von LINQ-Abfragen« ab Seite 770.

Die Ausführung von `Select` ergibt bei der Anwendung von lediglich einer Auflistung nur eingeschränkt Sinn. Das folgende Beispiel zeigt, wieso `Select` Rechenleistung kosten kann, ohne dabei einen nachvollziehbaren Nutzen nach sich zu ziehen.

[3] In C# übrigens schon, und zwar am Ende einer jeden LINQ-Abfrage.

Dieses Beispiel arbeitet mit einem Hochgeschwindigkeitszeitmesser, um die Laufdauer bestimmter Auswertungen zu messen. Dabei wird eine Auflistung mit 2.000.000 Kundenelementen einmal mit und einmal ohne `Select`-Abfrage ausgeführt.

HINWEIS Achten Sie beim Einrichten dieses Beispiels darauf, die ersten Zeilen des Beispielprojekts folgendermaßen abzuändern:

```
Module LinqDemo

    Private adrListe As List(Of Kontakt) = Kontakt.ZufallsKontakte(2000000)
    Private artListe As List(Of Artikel) = Artikel.Zufallsartikel(adrListe)
```

Achten Sie darauf, diese Zeilen für die anderen Beispiele wieder zurückzubauen.

Die eigentliche Methode, die das Beispiel enthält, sieht wie folgt aus:

```
    Sub SelectSpeedCompare()

        Dim sw = Stopwatch.StartNew
        Console.WriteLine("Starte Test")
        sw.Start()
        Dim ergebnisListe = From adrElement In adrListe
                            Order By adrElement.Nachname, adrElement.Vorname
                            Select KontaktId = adrElement.ID,
                                   adrElement.Nachname,
                                   adrElement.Vorname,
                                   PlzOrtKombi = adrElement.PLZ & " " & adrElement.Ort

        Dim ersteAnzahl = ergebnisListe.Count
        sw.Stop()
        Dim dauer1 = sw.ElapsedMilliseconds

        sw.Reset()
        sw.Start()
        Dim ergebnisListe2 = From adrElement In adrListe
                             Order By adrElement.Nachname, adrElement.Vorname

        Dim zweiteAnzahl = ergebnisListe2.Count
        sw.Stop()
        Dim dauer2 = sw.ElapsedMilliseconds

        Console.WriteLine("Abfragedauer mit Select: " & dauer1 & " für " & ersteAnzahl & "
Ergebniselemente.")
        Console.WriteLine("Abfragedauer ohne Select: " & dauer2 & " für " & zweiteAnzahl & "
Ergebniselemente.")
    End Sub
```

Das Ergebnis dieses Tests offenbart den Unterschied. Bei 2.000.000 Elementen kommt es …

```
Starte Test
Abfragedauer mit Select: 16939 für 2000000 Elemente.
Abfragedauer ohne Select: 16061 für 2000000 Elemente.
```

… bis zum Unterschied von einer Sekunde. Natürlich ist das kein Wert, an dem man sich für die Praxis in irgendeiner Form orientieren sollte; das Beispiel ist zudem natürlich auch praxisfern, denn wer zwei Millionen Datensätze im Speicher hält, hat entweder eine Datenbankengine programmiert oder das Konzept von Datenbanken und Datenbankabfragen nicht verstanden. Aber es geht nur um Tendenzen und den Tipp, Prozessorleistung nicht an Stellen unnötig zu verheizen, an denen es nicht nötig ist.

Übrigens, und besser könnte diese Überleitung zum nächsten Abschnitt nicht sein, wenn Sie sich das vorherige Listing anschauen, sollte Ihnen etwas auffallen, das Sie erstaunen sollte. Sie sehen es nicht? O.k.– dann überlegen Sie sich mal, ob es Zufall ist, dass die `Count`-Eigenschaft noch innerhalb der Zeitmessung abgefragt wird. Denn: Das ist nicht nur bewusst so gemacht, anderenfalls würden Sie komplett andere Ergebnisse bekommen, denn wenn Sie das Listing folgendermaßen abändern …

```
    Sub SelectSpeedCompare()

.
.
.

        sw.Stop()
        Dim ersteAnzahl = ergebnisListe.Count
        Dim dauer1 = sw.ElapsedMilliseconds

        sw.Reset()
        sw.Start()
        Dim ergebnisListe2 = From adrElement In adrListe
                             Order By adrElement.Nachname, adrElement.Vorname

        sw.Stop()
        Dim zweiteAnzahl = ergebnisListe2.Count
        Dim dauer2 = sw.ElapsedMilliseconds
.
.
.
    End Sub
```

… erhalten Sie auf einmal das folgende Ergebnis:

```
Starte Test
Abfragedauer mit Select: 1 für 2000000 Elemente.
Abfragedauer ohne Select: 0 für 2000000 Elemente.
```

Ein Fehler? Nein – das ist genau das Ergebnis, was bei der Ausführung herauskommen muss!

Performance von LINQ-Abfragen

Übrigens: Die Demo aus dem vorherigem Abschnitt ist ein Paradebeispiel für schlechte Kombinationen von Abfrageausdrücken. Schauen Sie sich das Beispiel an, und überlegen Sie, was passiert:

```
    Sub PerformanceTest()

        Console.WriteLine("Abfrage durchführen:")
        Console.WriteLine("--------------------")
```

```
        Dim sw = Stopwatch.StartNew
        Dim ergebnisListe = From adrElement In adrListe
                            Order By adrElement.Nachname, adrElement.Vorname
                            Where adrElement.ID > 50 And adrElement.ID < 100 And
                                    adrElement.Nachname Like "L*"

        Dim liste = ergebnisListe.ToList
        sw.Stop()
        Console.WriteLine("Die Abfrage benötigte: " & sw.ElapsedMilliseconds.ToString &
                          " Millisekunden.")
    End Sub
```

Hier werden zunächst alle Elemente sortiert und anschließend gefiltert. Das ist natürlich Unsinn, denn wenn wir die Elemente erst filtern, gibt es anschließend weniger Elemente zu sortieren. Sie sollten sich also gut überlegen, in welcher Reihenfolge Sie Abfrageausdrücke platzieren, denn dort ist einiges an Performance rauszukitzeln.

Der oben stehende Abfrageausdruck benötigt auf einem I7-Notebook mit Q720-Prozessor bei 200.000 Elementen ca. 1.400 Millisekunden. Wenn wir den Abfrageausdruck umstellen, sodass das Programm erst filtert und anschließend sortiert …

```
        Dim ergebnisListe = From adrElement In adrListe
                            Where adrElement.ID > 50 And adrElement.ID < 100 And
                                    adrElement.Nachname Like "L*"
                            Order By adrElement.Nachname, adrElement.Vorname
```

…braucht es nur noch sensationelle 90 Millisekunden – das ist mehr als 14 mal schneller!

TIPP Achten Sie deswegen darauf, nach Möglichkeit erst die Filteroperationen durchzuführen, bevor Sie andere Abfragekriterien zur Abfrage hinzufügen!

Kombinieren und verzögertes Ausführen von LINQ-Abfragen

Eines vorweg: LINQ-Abfragen werden immer verzögert ausgeführt, die Überschrift könnte also insofern in die Irre führen, als dass sie impliziert, der Entwickler, der sich LINQ-Abfragen bedient, hätte eine Wahl. Er hat sie nämlich nicht.

Wann immer Sie eine Abfrage definieren, und sei sie wie die folgende …

```
        Dim adrListeGruppiert = From adrElement In adrListe
                                Join artElement In artListe
                                On adrElement.ID Equals artElement.IDGekauftVon
                                Select adrElement.ID, adrElement.Nachname,
                                       adrElement.Vorname, adrElement.PLZ,
                                       artElement.ArtikelNummer, artElement.ArtikelName,
                                       artElement.Anzahl, artElement.Einzelpreis,
                                       Postenpreis = artElement.Anzahl * artElement.Einzelpreis
                                Order By Nachname, ArtikelNummer
                                Where (PLZ > "0" And PLZ < "50000")
```

```
                                Group ArtikelNummer, ArtikelName,
                                      Anzahl, Einzelpreis, Postenpreis
                                By ID, Nachname, Vorname
                                Into Artikelliste = Group, AnzahlArtikel = Count(),
                                     Gesamtpreis = Sum(Postenpreis)
                                Where Gesamtpreis > 1000
```

… noch so lang; die Abfragen selbst werden nicht zum Zeitpunkt ihres »Darüberfahrens« ausgeführt. Im Gegenteil: In `adrListeGruppiert` wird – um bei diesem Beispiel zu bleiben – im Prinzip nur eine Art *Ausführungsplan* generiert, also eine Liste der Methoden, die nacheinander ausgeführt werden, sobald ein Element aus der (noch zu generierenden!) Ergebnisliste abgerufen wird oder Eigenschaften bzw. Funktionen der Ergebnisliste abgerufen werden, die in unmittelbarem Zusammenhang mit der Ergebnisliste selbst stehen – wie beispielsweise die `Count`-Eigenschaft.

Und es wird noch besser: Verschiedene LINQ-Abfragen lassen sich so nacheinander »aufreihen«, wie das folgende Beispiel eindrucksvoll zeigt.

HINWEIS Achten Sie beim Einrichten dieses Beispiels darauf, die ersten Zeilen des Beispielprojekts folgendermaßen abzuändern:

```
Module LinqDemo

    Private adrListe As List(Of Kontakt) = Kontakt.ZufallsKontakte(500000)
    Private artListe As List(Of Artikel) = Artikel.Zufallsartikel(adrListe)
```

Achten Sie auch darauf, diese Zeilen für die anderen Beispiele wieder zurückzubauen.

```
    Sub SerialLinqsCompare()

        Dim sw = Stopwatch.StartNew
        Console.WriteLine("Starte Test")
        sw.Start()
        Dim ergebnisListe = From adrElement In adrListe
                            Order By adrElement.Nachname, adrElement.Vorname

        Dim ergebnisListe2 = From adrElement In ergebnisListe
                             Where adrElement.Nachname Like "L*"

        ergebnisListe2 = From adrElement In ergebnisListe2
                         Where adrElement.ID > 50 And adrElement.ID < 200

        Dim ersteAnzahl = ergebnisListe2.Count
        sw.Stop()
        Dim dauer1 = sw.ElapsedMilliseconds

        sw.Reset()
        sw.Start()
        Dim ergebnisListe3 = From adrElement In adrListe
                             Order By adrElement.Nachname, adrElement.Vorname
                             Where adrElement.Nachname Like "L*" And
                             adrElement.ID > 50 And adrElement.ID < 200
        Dim zweiteAnzahl = ergebnisListe3.Count
        sw.Stop()
        Dim dauer2 = sw.ElapsedMilliseconds
```

```
        Console.WriteLine("Abfragedauer serielle Abfrage: " & dauer1 & " für " & ersteAnzahl &
                    " Ergebniselemente.")
        Console.WriteLine("Abfragedauer kombinierte Abfrage: " & dauer2 & " für " & zweiteAnzahl &
                    " Ergebniselemente.")

        adrListe.Add(New Kontakt(51, "Löffelmann", "Klaus", "Wiedenbrücker Straße 47",
                    "59555", "Lippstadt"))
        Console.WriteLine("Elemente der seriellen Abfrage: " & ergebnisListe2.Count)

        Dim reineGenerischeListe = ergebnisListe.ToLookup(Function(einKontakt) einKontakt.ID)
    End Sub
```

Der zweite, in Fettschrift markierte Block entspricht im Grunde genommen dem ersten, nur dass hier Abfragen hintereinandergeschaltet, aber eben nicht ausgeführt werden. Die eigentliche Ausführung findet statt, wenn auf die zu entstehende Elementauflistung zugegriffen wird – im Beispiel also die `Count`-Eigenschaft der »Auflistung« abgerufen wird, die die Anzahl der Elemente nur dann zurückgeben kann, wenn es eine Anzahl an Elementen gibt.

Dass sich die beiden Ausführungspläne nicht nennenswert unterscheiden, zeigt das folgende Ergebnis …

```
Starte Test
Abfragedauer serielle Abfrage: 3531 für 17 Ergebniselemente.
Abfragedauer kombinierte Abfrage: 3561 für 17 Ergebniselemente.
```

… das mit 30 ms Unterschied bei 500.000 Elementen wirklich nicht nennenswert ist.

WICHTIG Die Ausführungspläne, die Sie durch die Abfragen erstellen, werden jedes Mal ausgeführt, wenn Sie auf eine der Ergebnislisten zugreifen. Wiederholen Sie die letzte fett geschriebene Zeile im obigen Listing ...

```
.
.
        Dim zweiteAnzahl = ergebnisListe3.Count
        zweiteAnzahl = ergebnisListe3.Count
        hsp.Stop()
        Dim dauer2 = hsp.DurationInMilliSeconds
.
.
```

... ergibt sich das folgende Ergebnis:

```
Starte Test
Abfragedauer serielle Abfrage: 3675 für 12 Ergebniselemente.
Abfragedauer kombinierte Abfrage: 7104 für 12 Ergebniselemente.
```

Parallelisieren von LINQ-Abfragen mit AsParallel

Seit dem .NET 4.0 erlaubt LINQ auch das Parallelisieren von Abfragen, und damit die Durchführung einer Abfrage auf mehreren Prozessoren – sofern diese auch vorhanden sind.

Doch aufgepasst: Das bloße Hinzufügen von `AsParallel` zur Datenquelle sorgt nicht dafür, dass bei vier vorhandenen Prozessoren die Abfrage auch grundsätzlich viermal so schnell läuft – es hängt von einer Vielzahl von Faktoren ab, ob und wenn in welcher Form eine Abfrage parallelisierbar, also auf mehrere Prozessoren oder Prozessorkerne verteilt werden kann.

Bei zeitkritischen Abfragen sollten Sie auf jeden Fall experimentieren, und auch wie hier in den Beispielen immer mal wieder die Stopwatch-Klasse verwenden, um ein Gefühl für die Optimierungsmöglichkeiten zu bekommen.

Das folgende Beispiel demonstriert den Umgang mit `AsParallel`, um eine Abfrage auf mehrere Prozessorkerne oder Prozessoren zu verteilen - so vorhanden:

```
Module LinqDemo

    Private Const ELEMENTCOUNT = 20000000

    Private adrListe As List(Of Kontakt) = Kontakt.ZufallsKontakte(ELEMENTCOUNT)
    Private artListe As List(Of Artikel) '= Artikel.Zufallsartikel(adrListe)
.
.
.
    Sub AsParallel()

        Dim sw = Stopwatch.StartNew
        Console.WriteLine("Starte Abfrage 1 (as Parallel)")
        Dim ergebnisListe1 = From adrElement In adrListe.AsParallel
                             Where adrElement.Nachname Like "L*"
                             Order By adrElement.Nachname, adrElement.Vorname

        Dim ersteList = ergebnisListe1.ToList
        Dim lastElement = ersteList(ersteList.Count - 1)
        Console.WriteLine("Ende Abfrage 1")
        Console.WriteLine()
        sw.Stop()
        Dim dauer1 = sw.ElapsedMilliseconds
        sw.Reset()

        'List neu aufbauen
        adrListe = Kontakt.ZufallsKontakte(ELEMENTCOUNT)
        sw.Start()
        Console.WriteLine("Starte Abfrage 2 (seriell)")

        Dim ergebnisListe2 = From adrElement In adrListe
                             Where adrElement.Nachname Like "L*"
                             Order By adrElement.Nachname, adrElement.Vorname

        Dim zweiteList = ergebnisListe2.ToList
        sw.Stop()
        Dim dauer2 = sw.ElapsedMilliseconds

        Console.WriteLine("Abfragedauer parallele Abfrage: " & dauer1 & " für " &
                          ersteList.Count & " Ergebniselemente.")
        Console.WriteLine("Abfragedauer serielle Abfrage: " & dauer2 & " für " &
                          zweiteList.Count & " Ergebniselemente.")
    End Sub
```

Sie sehen, dass das Parallelisieren selbst ein Kinderspiel ist: Sie hängen einfach die Erweiterungsmethode `AsParallel` an die Datenquelle an, um aus der Liste eine parallelisierbare Abfrage zu machen.

Im folgenden Bild haben wir ein I7-Notebook mit Q720-Prozessor richtig gefordert: Nicht weniger als 20.000.000 Datensätze – wie schon aus dem Listing hervorgeht – musste er sortieren, und er brauchte in der Spitze dafür nicht weniger als 5 GByte.

Solche Mengen an Daten kommen in der freien Wildbahn aber durchaus vor: Vielleicht weniger bei einer Kontakt- oder Kundenverwaltung, aber sicherlich in der Industrie im Bereich Messdaten-, Zeit- oder Betriebsdatenerfassung. Und dementsprechend viele Daten müssen hier gefiltert, sortiert oder sonst wie aufgearbeitet werden.

Abbildung 26.1 Im Prozessorverlauf erkennt man, dass die erste Abfrage parallelisiert ausgeführt wurde: Auf allen 8 Kernen ergibt sich ein sekundenlanger Ausschlag; die Abfrage wird auf allen Kernen und Prozessoren ausgeführt

Im Bild schön zu erkennen: Einen Moment lang sind tatsächlich alle Prozessoren ausgelastet; das Parallelisieren der Abfrage bringt bei einer derartigen Datenmenge einen deutlichen Geschwindigkeitsvorteil. Für die zweite, nicht parallelisierte Abfrage ist im Grunde genommen nur ein Kern/Prozessor 100%ig beschäftigt. Der Scheduler von Windows verteilt diese 100% zwar über mehrere physische Kerne, aber die laufen alle nur zwischen gar keiner und halber Last – im Grunde ist nämlich nur ein Thread und damit ein Kern mit Volllast unterwegs, und dieser springt zwischen den Cores hin und her und ergibt in der Summe niemals mehr als 100%.

ACHTUNG Parallelisieren von Abfragen kann auch nach hinten losgehen – unter Umständen braucht eine paralellisierte Abfrage länger als eine nicht-parallelisierte. Das liegt dann daran, dass der Overhead zum Aufteilen der Daten auf mehrere Threads größer als das reine parallele Ausführen der Abfrage. Gutes Testen auch mit unterschiedlichen Datenmengen ist hier unverzichtbar. Auch sollten Sie Kapitel 29 lesen, dass sich mit dem Parallelisieren generell beschäftigt, um ein Gefühl für die Materie zu bekommen.

Faustregeln für das Erstellen von LINQ-Abfragen

Die Faustregeln für das Erstellen von Abfragen lauten:

- LINQ-Abfragen bestehen nur aus Ausführungsplänen – die eigentlichen Abfragen werden durch ihre Definition nicht ausgelöst!
- Wenn das Ergebnis einer Abfrage als Ergebnisliste Gegenstand einer weiteren Abfrage wird, kommt der erste Abfrageplan auch nicht zur Auslösung; beide Abfragepläne werden miteinander kombiniert.
- Erst der Zugriff auf ein Element der Ergebnisauflistung (über `For Each` oder den Indexer) löst die Abfrage und damit das Erstellen der Ergebnisliste aus.
- Ein erneuter Zugriff auf eine elementeabhängige Eigenschaft oder auf die Elemente selbst löst – und das ist ganz wichtig – abermals das Erstellen der Ergebnisliste aus. Das gilt auch für Abfragen, die durch mehrere Abfragen kaskadiert wurden.
- Achten Sie auf das richtige Zusammenstellen der Abfrage: Filtern Sie erst (`Where`) bevor Sie weiter Sortieren, Selektieren oder Gruppieren.
- Versuchen Sie Abfragen zu parallelisieren, aber testen Sie gut aus, ob das wirklich etwas bringt.

Kaskadierte Abfragen

Das Beispiel, was ich Ihnen im letzten Listing vorgestellt habe, hat nämlich noch zwei weitere, auskommentierte Zeilen, die die Kaskadierungsfähigkeit (das Hintereinanderschalten) von Abfragen eindrucksvoll demonstrieren. Wenn Sie das Ergebnis dieser beiden zusätzlichen Zeilen des Listings …

```
adrListe.Add(New Kontakt(51, "Löffelmann", "Klaus", "Wiedenbrücker Straße 47", "59555", "Lippstadt"))
Console.WriteLine("Elemente der seriellen Abfrage: " & ergebnisListe2.Count)
```

… verstanden haben, dann haben Sie auch das Prinzip von LINQ verstanden!

```
Starte Test
Abfragedauer serielle Abfrage: 3515 für 7 Ergebniselemente.
Abfragedauer kombinierte Abfrage: 7101 für 7 Ergebniselemente.
Elemente der seriellen Abfrage: 8
```

Hier wird der Originalauflistung ein weiteres Element hinzugefügt, das exakt der Kriterienliste der kaskadierten Abfrage entspricht. Durch das Abfragen der `Count`-Eigenschaft von `ergebnisListe2` wird die *komplette* Abfragekaskade ausgelöst, denn von vorher 7 Elementen befinden sich anschließend 8 Elemente in der Ergebnismenge, was nicht der Fall wäre, würde LINQ nur die letzte Ergebnismenge, nämlich `ergebnisListe2` selbst auswerten, denn dieser haben wir das Element nicht hinzugefügt.

Gezieltes Auslösen von Abfragen mit ToArray oder ToList

Nun haben wir gerade erfahren, dass Abfragen bei jedem Zugriff auf die Ergebnisliste mit `For Each` oder direkt über eine Elementeigenschaft bzw. den Index zu einer neuen Ausführung des Abfrageplans führen. Das kann, wie im Beispiel des vorherigen Abschnitts, durchaus wünschenswert sein; in vielen Fällen können sich daraus aber echte Performanceprobleme entwickeln.

Aus diesem Grund implementierten die Entwickler von LINQ spezielle Methoden, die eine auf `IEnumerable` basierende Ergebnisliste wieder in eine »echte« Auflistung bzw. in ein »echtes« Array umwandeln können.

Am häufigsten wird dabei sicherlich die Methode `ToList` zur Anwendung kommen, die das Ergebnis einer wie auch immer gearteten LINQ-Abfrage in eine generische `List(Of t)` umwandelt – und dabei ist es völlig gleich, ob es sich um eine *LINQ to Objects*, *LINQ to SQL*, *LINQ to XML* oder eine *LINQ to sonstwas*-Abfrage handelt. Das Ausführen von `ToList` auf eine solche Ergebnisliste hat zwei Dinge zur Folge:

- Die LINQ-Abfrage wird ausgeführt
- Eine `List(Of {Ausgangstyp|Select-Anonymer-Typ})` wird zurückgeliefert

Die Elemente, die anschließend zurückkommen, sind völlig ungebunden – Sie können sie anschließend so oft indizieren, mit `For Each` durchiterieren und ihre `Count`-Eigenschaft abfragen, wie Sie wollen – mit der ursprünglichen LINQ-Abfrage haben sie nichts mehr zu tun. `ToList` funktioniert dabei denkbar einfach:

```
Dim reineGenerischeListe = ergebnisListe.ToList
```

Und `ToList` ist auch nicht die einzige Methode, mit der Sie eine Ergebnisliste von ihrer Ursprungsabfrage trennen können. Das können Sie – in Form von Ergebnislisten unterschiedlichen Typs – auch mit den folgenden Methoden machen. `t` ist dabei immer der Typ eines Elements der Ausgangsauflistung oder ein anonymer Typ, der in der Abfrage beispielsweise durch die `Select`-Klausel entstanden ist.

- `ToList`: überführt die Abfrageergebnisliste in eine generische `List(Of t)`
- `ToArray`: überführt die Abfrageergebnisliste in ein Array vom Typ `t`
- `ToDictionary`: überführt die Abfrageliste in eine generische Wörterbuchauflistung vom Typ `Dictionary(Of t.key, t)`. Dabei muss `t.key` durch die Angabe eines Lambda-Ausdrucks festgelegt werden, also etwa so, um beim Beispiel zu bleiben und die ID zum Nachschlageschlüssel zu machen:

  ```
  Dim reineGenerischeListe = ergebnisListe.ToDictionary(Function(einKontakt) einKontakt.ID)
  ```

- `ToLookup`: überführt die Abfrageliste in eine generische Lookup-Auflistung vom Typ `Lookup(Of t.key, t)`. Dabei muss `t.key` durch die Angabe eines Lambda-Ausdrucks festgelegt werden, also etwa so:

  ```
  Dim reineGenerischeListe = ergebnisListe.ToLookup(Function(einKontakt) einKontakt.ID)
  ```

HINWEIS Die `ToLookup`-Methode gibt ein generisches `Lookup`-Element zurück, ein 1:*n*-Wörterbuch, das Auflistungen von Werten Schlüssel zuordnet. Ein Lookup unterscheidet sich von `Dictionary`, das eine 1:*1*-Zuordnung von Schlüsseln zu einzelnen Werten ausführt.

Verbinden mehrerer Auflistungen zu einer neuen

LINQ kennt mehrere Möglichkeiten, Auflistungen miteinander zu verbinden. Natürlich ist es nicht unbedingt sinnig, das willkürlich zu tun: Die Auflistungen müssen schon in irgendeiner Form miteinander in logischer Relation stehen – dann aber kann man sich durch geschicktes Formulieren eine Menge an Zeit sparen. Wie eine solche Relation ausschauen kann, in der zwei Auflistungen zueinander stehen, demonstriert Ihnen der Abschnitt »Wie ›geht‹ LINQ prinzipiell« des vorherigen Kapitels – und diese Relation soll auch noch mal Gegenstand der Beispiele dieses Abschnitts sein (Abbildung 26.2).

Relation zwischen Artikel und Käufer-Tabellen

ID	Nachname	Vorname	Straße	PLZ	Ort
1	Heckhuis	Jürgen	Wiedenbrücker Str. 47	59555	Lippstadt
2	Wördehoff	Angela	Erwitter Str. 33	01234	Dresden
3	Dröge	Ruprecht	Douglas-Adams-Str. 42	55544	Ratingen
4	Dröge	Ada	Douglas-Adams-Str. 42	55544	Ratingen
5	Halek	Gaby „Doc“	Krankenhausplatz 1	59494	Soest
6					

IdGekauftVon	ArtikelNummer	ArtikelName	Kategorie	Einzelpreis	Anzahl
1	9-445	Die Tore der Welt	Bücher, Belletristik	19,90	2
3	3-537	Visual Basic 2005 - Das Entwicklerbuch	Bücher, EDV	59,00	2
3	3-123	SQL Server 2000 - So gelingt der Einstieg	Bücher, EDV	19,90	1
5	5-312	SQL Server 2008 - Das Profi-Buch	Bücher, EDV	39,90	2

Abbildung 26.2 Das LINQ-Beispielprogramm legt zwei Tabellen an, die nur logisch über die Spalten-ID miteinander verknüpft sind

Implizite Verknüpfung von Auflistungen

Die einfachste Möglichkeit, zwei Auflistungen zu gruppieren, zeigt Abbildung 26.3.

```
Sub Auflistungsvereinigung()

    Dim ergebnisliste = From adrElement In adrListe, artElement In artListe
    'Where adrElement.ID = artElement.IDGekauftVon

    For Each element In ergebnisliste
        element.
        With ele
            Cons                .adrElement.ID & ": " & .adrElement.Nachname & _
                                ", " & .adrElement.Vorname & ": " & _
                                .artElement.IDGekauftVon & ": " & .artElement.ArtikelName)
        End With
    Next
End Sub
```

adrElement
artElement
Equals
GetHashCode
GetType
ReferenceEquals
ToString
Allgemein Alle

Abbildung 26.3 Schon mit der `From`-Klausel können Sie zwei Auflistungen per LINQ miteinander kombinieren

`From` kombiniert zwei Auflistungen miteinander. Im Ergebnis erhalten Sie eine neue Auflistung, bei der jedes Element der ersten mit jedem Element der zweiten Auflistung kombiniert wurde. Die Ausgabe dieser Auflistung, die über zwei Eigenschaften verfügt, die jeweils Zugriff auf die Originalelemente gestatten, zaubert dann folgendes Ergebnis auf den Bildschirm (aus Platzgründen gekürzt):

```
6: Clarke, Christian: 1: Visual Basic 2010 - Das Entwicklerbuch
6: Clarke, Christian: 1: Das Herz der Hölle
6: Clarke, Christian: 2: Visual Basic 2005 - Das Entwicklerbuch
6: Clarke, Christian: 2: Das Vermächstnis der Tempelritter
6: Clarke, Christian: 3: Das Leben des Brian
6: Clarke, Christian: 3: Die Tore der Welt
.
.
.
7: Ademmer, Lothar: 1: Visual Basic 2010 - Das Entwicklerbuch
7: Ademmer, Lothar: 1: Das Herz der Hölle
7: Ademmer, Lothar: 2: Visual Basic 2005 - Das Entwicklerbuch
7: Ademmer, Lothar: 2: Das Vermächstnis der Tempelritter
7: Ademmer, Lothar: 3: Das Leben des Brian
7: Ademmer, Lothar: 3: Die Tore der Welt
```

An der Ergebnisliste können Sie erkennen, wie redundant und wenig informativ diese Liste ist, denn jeder Artikel der Artikelliste wird einfach und stumpf mit jedem Kontakt kombiniert. Wichtiger wäre es, Zuordnungen ausdrücklich bestimmen zu können, um zu sagen, dass ein Artikel mit einer bestimmten ID auch nur dem logisch dazugehörigen Kontakt zugeordnet werden soll. Und das funktioniert folgendermaßen:

```
Sub Auflistungsvereinigung()
    Dim ergebnisliste = From adrElement In adrListe, artElement In artListe
                        Where adrElement.ID = artElement.IDGekauftVon

    For Each element In ergebnisliste
        With element
            Console.WriteLine(.adrElement.ID & ": " & .adrElement.Nachname & _
                              ", " & .adrElement.Vorname & ": " & _
                              .artElement.IDGekauftVon & ": " & .artElement.ArtikelName)
        End With
    Next
End Sub
```

In dieser Version werden durch die `Where`-Klausel nur die Elemente in die Auflistung übernommen, die die gleiche `ID` haben wie das korrelierende Element der anderen Auflistung (`IDGekauftVon`). Das Ergebnis, was dann zu sehen ist, macht natürlich viel mehr Sinn, da es eine Aussagekraft hat (nämlich: welcher Kunde hat welche Artikel gekauft):

```
7: Sonntag, Uwe: 7: Visual Basic 2010 - Das Entwicklerbuch
7: Sonntag, Uwe: 7: The Da Vinci Code
7: Sonntag, Uwe: 7: O.C., California - Die komplette zweite Staffel (7 DVDs)
7: Sonntag, Uwe: 7: Desperate Housewives - Staffel 2, Erster Teil
7: Sonntag, Uwe: 7: Die Rache der Zwerge
8: Vüllers, Momo: 8: Programmieren mit dem .NET Compact Framework
9: Tinoco, Daja: 9: Das Herz der Hölle
9: Tinoco, Daja: 9: Mitten ins Herz
```

```
9: Tinoco, Daja: 9: The Da Vinci Code
9: Tinoco, Daja: 9: Der Schwarm
9: Tinoco, Daja: 9: Desperate Housewives - Staffel 2, Erster Teil
9: Tinoco, Daja: 9: Harry Potter und die Heiligtümer des Todes
9: Tinoco, Daja: 9: Der Teufel trägt Prada
9: Tinoco, Daja: 9: O.C., California - Die komplette zweite Staffel (7 DVDs)
10: Lehnert, Michaela: 10: Abgefahren - Mit Vollgas in die Liebe
10: Lehnert, Michaela: 10: Das Herz der Hölle
```

HINWEIS Die Verknüpfung zweier oder mehrerer Auflistungen mit `In` als Bestandteil der `From`-Klausel einer Abfrage nennt man implizite Verknüpfung von Auflistungen, da dem Compiler nicht ausdrücklich mitgeteilt wird, welche Auflistung auf Basis welchen Elements mit einer anderen verknüpft wird. Eine ausdrückliche oder explizite Verknüpfung stellen Sie mit `Join` her, das im nächsten Abschnitt beschrieben wird. Explizit oder implizit hier im Beispiel ist aber letzten Endes einerlei – das Ergebnis ist in beiden Fällen dasselbe.

Explizite Auflistungsverknüpfung mit Join

Im Gegensatz zu impliziten Auflistungsverknüpfungen, die Sie, wie im letzten Abschnitt beschrieben, mit `In` als Teil der `From`-Klausel bestimmen, erlaubt Ihnen die `Join`-Klausel, so genannte explizite Auflistungsverknüpfungen festzulegen. Die generelle Syntax der `Join`-Klausel lautet:

```
Dim ergebnisliste = From bereichsVariable In ersterAuflistung
                    Join verknüpfungsElement In zweiterAuflistung
                    On bereichsVariable.JoinKey Equals verknüpfungsElement.ZweiterJoinKey
```

`Join` verknüpft die erste mit der zweiten Tabelle über einen bestimmten Schlüssel (`JoinKey`, `ZweiterJoinKey`), der beide Tabellen zueinander in Relation stellt. In unserem Beispiel wird für jede Bestellung in der Artikeltabelle ein Schlüssel (ein *Key*, eine *ID*) definiert, der der Nummer der *ID* in der Kontakttabelle entspricht.

Im Vergleich zur impliziten Verknüpfung von Tabellen ändert sich im Ergebnis nichts; die Umsetzung des vorherigen Beispiels mit `Join` gestaltet sich folgendermaßen:

```
    Sub JoinDemo()
        Dim ergebnisliste = From adrElement In adrListe
                            Join artElement In artListe
                            On adrElement.ID Equals artElement.IDGekauftVon

        For Each element In ergebnisliste
            With element
                Console.WriteLine(.adrElement.ID & ": " & .adrElement.Nachname & _
                                  ", " & .adrElement.Vorname & ": " & _
                                  .artElement.IDGekauftVon & ": " & .artElement.ArtikelName)
            End With
        Next
    End Sub
```

Es gibt die Möglichkeit, mit der Klausel `Group Join` Verknüpfungen mehrerer Tabellen auf bestimmte Weise in Gruppen zusammenzufassen. Wie das funktioniert, erfahren Sie im Abschnitt »Group Join« ab Seite 783.

Gruppieren von Auflistungen

Die Klausel `Group By` erlaubt es, Dubletten von Elementen einer oder mehrerer Auflistungen in Gruppen zusammenzufassen. Sie möchten also beispielsweise eine Auflistung von Kontakten nach Nachnamen gruppieren und dann in einer geschachtelten Schleife außen durch die Namen und innen durch alle den Namen zugeordneten Kontakte iterieren. Mit der `Group By`-Klausel können Sie genau das erreichen, wie das folgende Beispiel zeigt:

```
Sub GroupByDemo()
    Dim ergebnisliste = From adrElement In adrListe
                        Group By adrElement.Nachname Into Kontaktliste = Group
                        Order By Nachname

    For Each element In ergebnisliste
        With element
            Console.WriteLine(element.Nachname)
            For Each Kontakt In element.Kontaktliste
                With Kontakt
                    Console.WriteLine(.ID & ": " & .Nachname & ", " & .Vorname)
                End With
            Next
            Console.WriteLine()
        End With
    Next
End Sub
```

Wörtlich ausformuliert hieße die `Group By`-Klausel dieses Beispiels: »Erstelle eine Liste aller Nachnamen (`Group By adrElement.Nachname`) und mache diese unter der `Nachname`-Eigenschaft in der Liste zugänglich.[4] Fasse alle Elemente der Ausgangsliste in den jeweiligen Auflistungen zusammen, die dem Nachnamen zugehörig sind (`Into … = Group`), und mache diese Auflistung über die Eigenschaft `Kontaktliste` verfügbar.«

HINWEIS Falls Sie keine Aliasbenennung der Gruppe vornehmen (der Ausdruck würde dann einfach

```
    Dim ergebnisliste = From adrElement In adrListe
                        Group By adrElement.Nachname Into Group
                        Order By Nachname
```

heißen), würde die Eigenschaft, mit der Sie die zugeordneten Elemente erreichen können, einfach `Group` heißen.

Wenn Sie das Beispiel starten, sehen Sie das gewünschte Ergebnis, etwa wie im folgenden Bildschirmauszug (aus Platzgründen gekürzt):

```
Weichelt
19: Weichelt, Anne
39: Weichelt, Gabriele
47: Weichelt, Uta
```

[4] Wenn nichts anderes gesagt wird, heißt das Feld bzw. die Eigenschaft, nach der Sie gruppieren, in der späteren Auflistung so wie das Ausgangsfeld. Wenn Sie das nicht wünschen, können Sie das durch das Davorsetzen eines neuen Namens etwa mit `Group By Lastname = adrElement.Nachname Into Kontaktliste = Group` – an Ihre Wünsche anpassen. Anders als im Beispiel wäre hier `Lastname` die Eigenschaft, mit der Sie später die Nachnamen abfragen könnten.

```
69: Weichelt, Franz
97: Weichelt, Hans

Weigel
37: Weigel, Lothar
40: Weigel, Rainer
43: Weigel, Lothar
58: Weigel, Hans
60: Weigel, Anja

Westermann
11: Westermann, Margarete
21: Westermann, Alfred
28: Westermann, Alfred
41: Westermann, Alfred
77: Westermann, Guido
98: Westermann, José
100: Westermann, Michaela

Wördehoff
14: Wördehoff, Bernhard
```

Gruppieren von Listen aus mehreren Auflistungen

`Group By` eignet sich auch sehr gut dazu, mit `Join` kombinierte Listen zu gruppieren und auszuwerten. Stellen Sie sich vor, Sie möchten eine Liste mit Kontakten erstellen, mit der Sie über jeden Kontakt wieder auf eine Liste mit Artikeln zugreifen können, um auf diese Weise herauszufinden, welche Kunden welche Artikel gekauft hatten. Die entsprechende Abfrage und die anschließende Iteration durch die Ergebniselemente sähen dann folgendermaßen aus:

```
    Sub GroupByJoinedCombined()
        Dim ergebnisliste = From adrElement In adrListe
                            Join artElement In artListe
                            On adrElement.ID Equals artElement.IDGekauftVon
                            Group artElement.IDGekauftVon, Artikelnummer = artElement.ArtikelNummer,
                                artElement.ArtikelName
                            By artElement.IDGekauftVon, adrElement.Nachname, adrElement.Vorname
                            Into Artikelliste = Group, AnzahlArtikel = Count() Order By Nachname

        For Each kontaktElement In ergebnisliste
            With kontaktElement

                Console.WriteLine(.IDGekauftVon & ": " & .Nachname & ", " & .Vorname)
                For Each Artikel In .Artikelliste
                    With Artikel
                        Console.WriteLine("   " & .Artikelnummer & ": " & .ArtikelName)
                    End With
                Next
                Console.WriteLine()
            End With
        Next
    End Sub
```

Hier sehen Sie eine Zusammenfassung dessen, was wir in den letzten Abschnitten kennen gelernt haben. Die Abfrage beginnt mit einem `Join` und vereint Artikel und Kundenliste zu einer flachen Auflistung, die sowohl die Namen als auch die Artikel für jeden Namen beinhaltet. Und dann wird gruppiert: Anders als beim ersten Gruppierungsbeispiel, in dem alle Elemente in der untergeordneten Liste einbezogen werden, geben wir hier zwischen `Group` und `By` die Felder an, die in der inneren Auflistung als Eigenschaften erscheinen sollen, wir ändern also, ähnlich dem `Select`-Befehl, das Schema der inneren Auflistung. Die Elemente, die wir anschließend hinter dem `By` angeben, sind die, nach denen gruppiert wird und die damit auch in der äußeren Auflistung verfügbar sind. Das Ergebnis entspricht dann unseren Erwartungen:

```
21: Wördehoff, Theo
   4-444: The Da Vinci Code
   2-424: 24 - Season 6 [UK Import - Damn It!]
   2-134: Abgefahren - Mit Vollgas in die Liebe
   3-534: Mitten ins Herz
   3-333: Der Schwarm
   3-537: Visual Basic 2005 - Das Entwicklerbuch
   4-444: Harry Potter und die Heiligtümer des Todes
   5-554: O.C., California - Die komplette zweite Staffel (7 DVDs)
   2-134: Abgefahren - Mit Vollgas in die Liebe
   7-321: Das Herz der Hölle

75: Wördehoff, Katrin
   2-134: Abgefahren - Mit Vollgas in die Liebe
   2-134: Abgefahren - Mit Vollgas in die Liebe
   2-134: Abgefahren - Mit Vollgas in die Liebe
   3-123: Das Vermächstnis der Tempelritter
   2-134: Abgefahren - Mit Vollgas in die Liebe
   1-234: Das Leben des Brian
   3-543: Microsoft Visual C# 2005 - Das Entwicklerbuch
   3-543: Microsoft Visual C# 2005 - Das Entwicklerbuch
   9-423: Desperate Housewives - Staffel 2, Erster Teil
   7-321: Das Herz der Hölle
   5-506: Visual Basic 2010 - Das Entwicklerbuch
   5-513: Microsoft SQL Server 2010 - Einführung in Konfiguration, Administration, Programmierung
   9-646: Was diese Frau so alles treibt
   5-506: Visual Basic 2010 - Das Entwicklerbuch
   2-321: Die Rache der Zwerge
   9-445: Die Tore der Welt
   4-444: Harry Potter und die Heiligtümer des Todes

77: Wördehoff, Theo
   7-321: Das Herz der Hölle
   1-234: Das Leben des Brian
   9-009: Die Wächter
   3-123: Das Vermächtnis der Tempelritter
   5-518: Visual Basic 2010 - Neue Technologien - Crashkurs
```

Group Join

Exakt das gleiche Ergebnis bekommen Sie, allerdings mit sehr viel weniger Aufwand, wenn Sie sich der `Group Join`-Klausel bedienen, die `Join` und `Group By` miteinander kombiniert – das folgende Beispiel zeigt, wie es geht:

```
Sub GroupJoin()
   Dim ergebnisliste = From adrElement In adrListe
                       Group Join artElement In artListe
                       On adrElement.ID Equals artElement.IDGekauftVon Into Artikelliste = Group
```

```
        For Each kontaktElement In ergebnisliste
            With kontaktElement

                Console.WriteLine(.adrElement.ID & ": " &
                                  .adrElement.Nachname & ", "
                                  & .adrElement.Vorname)
                For Each Artikel In .Artikelliste
                    With Artikel
                        Console.WriteLine("   " & .ArtikelNummer & ": " & .ArtikelName)
                    End With
                Next
                Console.WriteLine()
            End With
        Next
    End Sub
```

Aggregatfunktionen

Aggregatfunktionen sind Funktionen, die ein bestimmtes Feld einer Auflistung aggregieren – zum Beispiel aufsummieren, den Durchschnitt berechnen, das Maximum oder das Minimum ermitteln oder einfach nur zählen, wie viele Elemente in einer Auflistung vorhanden sind. Anders als bei »normalen« Abfragen werden beim Aggregieren also die Elemente der Auflistung einer Aggregatfunktion alle nacheinander übergeben, und diese Funktion arbeitet bzw. rechnet dann mit jedem einzelnen dieser Elemente, betrachtet sie aber als Menge.

HINWEIS LINQ-Abfragen fangen zwar grundsätzlich mit der `From`-Klausel an – ein reines Aggregieren von Auflistungen ist allerdings die einzige Ausnahme. Wenn Sie eine reine Aggregierung auf eine Auflistung ausführen wollen, leiten Sie die LINQ-Abfrage nicht mit `From`, sondern mit der Klausel `Aggregate` ein; die folgenden Beispiele werden es gleich demonstrieren.

Lassen Sie uns ganz einfach beginnen. Die erste Demo …

```
    Sub AggregateDemo()

        Dim scheinbarerUmsatz = Aggregate artElement In artListe
                                Into Summe = Sum(artElement.Einzelpreis)
        Console.WriteLine("Scheinbar beträgt der Umsatz {0:#,##0.00} Euro", scheinbarerUmsatz)
```

… aggregiert unsere Artikelliste in eine Gesamtsumme und ermittelt so scheinbar, wie viel Gesamtumsatz mit den Artikeln erzielt wurde, etwa wie hier zu sehen:

```
Scheinbar beträgt der Umsatz 47.117,60 Euro
```

Doch diese Aussage stimmt wirklich nur scheinbar. Denn bei der Abfrage wurden nur die Artikelpreise in einer `Decimal`-Variable (`scheinbarerUmsatz`) summiert – die Anzahl eines Artikels, also wie oft dieser jeweils gekauft wurde, blieb unberücksichtigt. Dieses Manko behebt das direkt anschließende Beispiel …

```
        Dim gesamtUmsatz = Aggregate artElement In artListe
                           Let Artikelumsatz = artElement.Einzelpreis * artElement.Anzahl
                           Into Summe = Sum(Artikelumsatz)
        Console.WriteLine("Der Umsatz beträgt {0:#,##0.00} Euro", gesamtUmsatz)
```

… das mit einem sehr, sehr alten Bekannten – nämlich der `Let`-Klausel – eine neue Variable einführt, die das *Produkt* von Anzahl und Preis aufnimmt und anschließend als Ergebnis aufsummiert:

```
Der Umsatz beträgt 93.306,00 Euro
```

Zurückliefern mehrerer verschiedener Aggregierungen

Wenn eine Aggregatabfrage nur eine einzige Aggregatfunktion verwendet, dann ist das Abfrageergebnis eine einzige Variable, die dem Rückgabetyp der Aggregatfunktion entspricht. Liefert beispielsweise die `Sum`-Aggregatfunktion ein Ergebnis vom Typ `Decimal`, dann ist das Abfrageergebnis ebenfalls vom Typ `Decimal`. Sie können aber auch mehrere Aggregate innerhalb einer einzigen Abfrage verwenden, wie das dritte Beispiel zeigt:

```
Dim mehrereInfos = Aggregate artElement In artListe
                   Let Artikelumsatz = artElement.Einzelpreis * artElement.Anzahl
                   Into Summe = Sum(Artikelumsatz),
                        Minimalpreis = Min(artElement.Einzelpreis),
                        MaximalPreis = Max(artElement.Einzelpreis)
Console.WriteLine("Der Umsatz beträgt {0:#,##0.00} Euro" & vbNewLine & _
                  "Minimal- und Maximalpreis jeweils " & _
                  "{1:#,##0.00} und {2:#,##0.00} Euro",
                  mehrereInfos.Summe, mehrereInfos.Minimalpreis,
                  mehrereInfos.MaximalPreis)
```

In diesem Beispiel werden in einem Rutsch Artikelpostensumme, Maximalpreis und Minimalpreis ermittelt; die Rückgabevariable ist in dem Fall vom Typ anonyme Klasse mit den drei Eigenschaften, die über die entsprechenden Variablen innerhalb der Abfrage (`Summe`, `Minimalpreis` und `Maximalpreis`) definiert wurden. Hätten Sie diese Variablendefinitionen weggelassen und stattdessen nur die Aggregatfunktionen angegeben, hätte der Compiler für die Eigenschaften des anonymen Rückgabetypen Namen gewählt, die den Namen der Aggregatfunktionen entsprächen.

WICHTIG Anders als bei Standardabfragen werden reine Aggregatabfragen, wie Sie sie hier in den letzten drei Beispielen kennen gelernt haben, *nicht* verzögert, sondern sofort ausgeführt.

Kombinieren von gruppierten Abfragen und Aggregatfunktionen

Das Gruppieren von Abfragen eignet sich naturgemäß gut für den Einsatz von Aggregatfunktionen, denn beim Gruppieren von Datensätzen werden verschiedene Elemente zusammengefasst, auf die sich dann die unterschiedlichsten Aggregatfunktionen anwenden lassen.

Ein Beispiel soll auch das verdeutlichen: Angenommen, Sie möchten herausfinden, welcher Kunde durch seine Käufe wie viel Umsatz produziert hat. In diesem Fall würden Sie die Abfrage nach Artikeln gruppieren, und zwar nach dem Feld `IDGekauftVon` – denn dieses Feld repräsentiert »*wer* hat gekauft«. Damit in der Ergebnisliste nicht nur eine nichts sagende Nummer, sondern ein greifbarer Vor- sowie ein Nachname zu finden sind, verknüpfen Sie Artikeltabelle und Kundentabelle mit einer `Join`-Abfrage oder – noch besser – kombinieren Gruppierung und Zusammenfassung der Tabellen mit einem `Group Join`. In die `Group Join`-Abfrage bauen Sie gleichzeitig noch die schon bekannte `Order By`-Klausel ein, mit der Sie die Ergebnisliste nach dem errechneten

Umsatz sortieren – durch die Angabe von `Descending` als Schlüsselwort sogar absteigend, um eine Top-Ranking-Umsatzliste zu bekommen. Die eigentliche Aggregierung der Umsätze erfolgt dann mit der hinter `Into` stehenden Klausel, in der ebenfalls bestimmt wird, mit welcher Eigenschaft später die innere Liste (`JoinedList`) im Bedarfsfall durchiteriert werden kann (und dabei dann einzelne `artListe`-Elemente vom Typ `Artikel` zurückliefert). Die Aggregatfunktion `Sum`, die dafür notwendig ist, summiert im Beispiel das Produkt von Menge und Artikeleinzelpreis auf, was – für jeden Kontakt gruppiert – exakt dem Umsatz jedes Kunden entspricht.

TIPP Wie ebenfalls im folgenden Beispiel zu sehen, verwendet die Abfrage die `Take`-Klausel, um die Ergebnisliste auf 10 Elemente zu beschränken. Es gibt weitere Klauseln, die Sie für solche Einschränkungen verwenden können, wie beispielsweise `Take While` oder `Skip` – deren Anwendungsweise ist ähnlich und über dieses Beispiel leicht ableitbar. Die Onlinehilfe verrät Ihnen zu diesen Klauseln mehr.

```
Sub JoinAggregateDemo()
    Dim ergebnisListe = From adrElement In adrListe
                        Group Join artElement In artListe
                        On adrElement.ID Equals artElement.IDGekauftVon
                        Into JoinedList = Group,
                        KundenUmsatz = Sum(artElement.Einzelpreis * artElement.Anzahl)
                        Order By KundenUmsatz Descending
                        Take 10

    For Each item In ergebnisListe
        With item.adrElement
            Console.WriteLine(.ID & ": " & .Nachname & ", " &
                              .Vorname & " --- Umsatz:" &
                              item.KundenUmsatz.ToString("#,##0.00") &
                              " Euro")
        End With
    Next
End Sub
```

Wenn Sie dieses Beispiel ausführen, produziert es in etwa die folgenden Zeilen:

```
3: Braun, Franz --- Umsatz:2.358,00 Euro
34: Plenge, Franz --- Umsatz:2.287,80 Euro
91: Jungemann, Katrin --- Umsatz:2.183,05 Euro
94: Tinoco, Anne --- Umsatz:2.128,00 Euro
100: Hollmann, Gareth --- Umsatz:2.108,15 Euro
18: Westermann, Anja --- Umsatz:2.098,05 Euro
66: Hörstmann, Katrin --- Umsatz:1.923,00 Euro
54: Albrecht, Uta --- Umsatz:1.843,40 Euro
97: Hollmann, Daja --- Umsatz:1.798,50 Euro
27: Westermann, Alfred --- Umsatz:1.788,15 Euro
```

Kapitel 27

LINQ to XML

In diesem Kapitel:

Einführung in LINQ to XML 788
XML-Dokumente verarbeiten – damals und heute 789
XML-Literale – XML direkt im Code ablegen 790
Erstellen von XML mithilfe von LINQ 791
Abfragen von XML-Dokumenten mit LINQ to XML 793
IntelliSense-Unterstützung für LINQ To XML-Abfragen 794

Einführung in LINQ to XML

XML steht für *Extensible Markup Language,* was auf Deutsch so viel wie *erweiterbare Auszeichnungssprache* bedeutet. XML ist eine Untermenge von SGML, welches bereits 1986 vom World Wide Web Consortium (W3C) standardisiert wurde.

Markup Languages (deutsch: *Auszeichnungssprachen*) dienen dazu, mit einem Satz von Attributen eine Dokumentenstruktur inhaltlich zu gestalten, oder zu definieren, wie die Inhalte dieses Dokuments dargestellt werden sollen – oder natürlich auch beides. Die *HyperText Markup Language* (wir kennen sie eher unter ihrer Abkürzung *HTML*) ist dabei wohl die derzeit am häufigsten angewendete Auszeichnungssprache für das Speichern und Darstellen von Inhalten im Web; aber auch die *Extensible Application Markup Language* (kurz XAML) ist Ihnen im Zusammenhang mit dem Thema *Windows Presentation Foundation* im Laufe dieses Buchs schon hier und da begegnet. Generell können XML-Dokumente simpler Struktur auch dazu dienen, einfache Daten wie Adressen oder Artikel strukturiert zu speichern – das X (das wie schon erwähnt für *extensible,* also etwa *erweiterbar* steht) in XML beschreibt schon, dass die Extensible Markup Language eben die Basis für konkrete Implementierungen für Auszeichnungssprachen bildet. XML hat aber noch einen weiter gehenden Anspruch. Mit XML lassen sich prinzipiell alle Arten von Daten beschreiben und übertragen.

LINQ to XML wiederum ist, vereinfacht ausgedrückt, eine Implementierung, die dazu dient, dass Daten, die in XML-Dokumenten gespeichert werden, mithilfe der schon vorgestellten Language Integrated Query verarbeitet und selektiert werden können.

Bevor wir uns mit dem eigentlichen Kapitelthema *LINQ to XML* beschäftigen, lassen Sie uns kurz XML als solches rekapitulieren: Ein typisches XML-Dokument besteht, vereinfacht gesagt, aus Elementen, Attributen und Werten.

BEGLEITDATEIEN Sie finden die Begleitdateien für die folgenden Abschnitte im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 27\LinqToXml
```

Beispiel:

```
<software Installierbar="Ja">Visual Studio 2010</software>
```

Das Element `software` besitzt hier ein Attribut `Installierbar` und den Wert `Visual Studio 2010`. Elemente in XML beginnen mit einem Starttag `<name>` und enden mit dem Endtag `</name>`, wobei *name* durch ein beliebiges Wort getauscht werden kann (Achtung: Dabei dürfen keine Leerzeichen verwendet werden!).

Zwischen `<name>` und `</name>` steht der Wert. Ist kein Wert vorhanden, kann anstelle von `<name></name>` auch `<name/>` geschrieben werden.

Attribute werden in dem Element notiert, etwa:

```
<person name="Müller" vorname="Hans"/>
```

Dieses Beispiel definiert ein Element ohne Wert, das zudem die Attribute `name` und `vorname` beinhaltet. Bei all diesen Beispielen handelt es sich genau genommen um ein XML-Fragment, kein vollständiges XML-Dokument. Jedes XML-Dokument sollte zum Beispiel mit einer Auszeichnung beginnen, die deutlich macht, dass es sich um ein XML Dokument handelt: *<?xml version="1.0"?>*

Eine detaillierte Beschreibung von XML würde den Umfang dieses Buchs sprengen, daher sei hier auf die gute deutsche Einführung von *SELFHTML*[1] bzw. auf die Seiten von *W3C*[2] verwiesen.

XML-Dokumente verarbeiten – damals und heute

Um bis Visual Basic 2005 eine XML-Struktur zu erstellen oder zu ändern, mussten Sie schon einiges an Aufwand betreiben. Um das zu demonstrieren, schauen wir uns an, was zu tun ist, um folgende XML-Struktur, die Daten für Fahrzeuge eines Fuhrpark speichert, um das Element `baujahr` zu erweitern:

```
 <fuhrpark>
  <kennzeichen>MS - VB 2008</kennzeichen>
  <ladung menge="10 Tonnen">Salz</ladung>
  <hersteller>MAN</hersteller>
  <baujahr>1998</baujahr>
</fuhrpark>
```

Der Code lautete hierzu für Visual Basic 2005:

```
        Dim xml As New XmlDocument
        xml.Load(Application.StartupPath + "\fuhrpark1.xml")
        Dim nodeList As XmlNodeList = xml.GetElementsByTagName("fuhrpark")

        'Für den Fall, dass es mehrere gäbe: Schleife
        For Each node As XmlNode In nodeList
            Dim xmlElement As XmlElement = xml.CreateElement("baujahr")
            xmlElement.InnerText = "1998"
            node.AppendChild(xmlElement)
        Next
```

Seit Visual Basic 2008 gibt es viele Vereinfachungen bezüglich der XML-Verarbeitung, denn .NET Framework wurde um einige recht leistungsfähige Klassen erweitert. Die wichtigsten dabei sind `XDocument`, `XElement` und `XAttribute`.

[1] Mehr dazu unter *http://de.selfhtml.org/*

[2] Weiteres unter *http://www.w3.org/*

Um nunmehr das oben stehende XML-Dokument zu erhalten, reicht unter Zuhilfenahme dieser neuen Klassen folgender Code aus:

```
Dim xml As New XDocument
xml.Add(New XElement("fuhrpark", _
            New XElement("kennzeichen", "MS - VB 2008"), _
            New XElement("ladung", _
                        New XAttribute("menge", "10 Tonnen"), _
                        "Salz" _
                        ), _
            New XElement("hersteller", "MAN") _
))
```

Unterhalb des Elements `fuhrpark` werden die Elemente `kennzeichen`, `ladung` und `hersteller` gespeichert. Zudem erhält das Element `ladung` ein weiteres Attribut namens `menge`. Um nun ein weiteres Element `baujahr` hinzuzufügen, verwenden Sie einfach die `Add`-Methode, etwa wie in folgender Codezeile zu sehen:

```
xml.Element("fuhrpark").Add(New XElement("baujahr", 1998))
```

Möchten Sie den Wert eines `XElements` (zum Beispiel `1998`) ermitteln, verwenden Sie dazu die `Value`-Eigenschaft, etwa auf die folgende Art und Weise:

```
'Mit der Value-Eigenschaft Werte ermitteln
'Gesamter "Wert":
Console.WriteLine(xml.Element("fuhrpark").Value)

'Nur das Baujahr:
Console.WriteLine(xml.Element("fuhrpark").Element("baujahr").Value)
Console.ReadKey()
```

Die letzten Zeilen dieses Beispiel erzeugen folgende Ausgabe auf dem Bildschirm:

```
MS - VB 2008SalzMAN1998
1998
```

XML-Literale – XML direkt im Code ablegen

Seit Visual Basic 2008 gibt es die Möglichkeit, für `XDocument`-und `XElement`-Objekte die entsprechenden Dokumente direkt im Code zu erstellen – diese XML-Dokumente, die im Code abgelegt werden können, nennt man *XML-Literale*. Wie das funktioniert, zeigt das folgende Beispiel:

```
Dim fuhrpark As XElement = <fuhrpark>
                                <kennzeichen>MS - VB 2008</kennzeichen>
                                <ladung menge="10 Tonnen">Salz</ladung>
                                <hersteller>MAN</hersteller>
                            </fuhrpark>
```

Beim `fuhrpark`-Objekt handelt es sich um ein gewöhnliches `XElement` und `baujahr` kann nun wie gewohnt hinzugefügt werden:

```
fuhrpark.Add(New XElement("baujahr", 1998))
```

Mit XML-Literalen geht das sogar noch einfacher:

```
fuhrpark.Add(<baujahr>1998</baujahr>)
```

Einbetten von Ausdrücken in XML-Literalen

XML-Dokumente, die Sie im Code direkt ablegen, müssen nicht komplett statisch sein, sondern können auch sich dynamisch anpassende Elemente erhalten. Es ist so möglich, Ausdrücke mit `<%=  %>` als XML in den Code einzubetten, wie die folgenden beiden Zeilen zeigen:

```
        Dim myBaujahr As Integer = 1998
        fuhrpark.Add(<baujahr><%= myBaujahr %></baujahr>)
```

Beim Aufrufen der `Add`-Methode wird der Wert des `baujahr`-Elements durch die Variable `myBaujahr` ersetzt.

Erstellen von XML mithilfe von LINQ

Mithilfe von LINQ können ebenso XML-Dokumente bzw. -Elemente erstellt werden. Aus einer bestehenden Liste, die durch die folgenden Codezeilen erstellt wird …

```
        Dim fahrzeugliste As New List(Of Fahrzeug)
        fahrzeugliste.Add(New Fahrzeug With {.Kennzeichen = "MS-VB 2008", .Hersteller = "MAN", _
                                             .Ladung = New Fahrzeug.LadungsBeschreibung With
                                              {.Menge = 10000, .Gut = "Salz"}})
        fahrzeugliste.Add(New Fahrzeug With {.Kennzeichen = "MS-C# 2008", .
                                              Hersteller = "Mercedes-Benz",
                                             .Ladung = New Fahrzeug.LadungsBeschreibung With
                                              {.Menge = 20000, .Gut = "Erdnüsse"}})
        fahrzeugliste.Add(New Fahrzeug With {.Kennzeichen = "MS-J# 2008", .Hersteller = "DAF",
                                             .Ladung = New Fahrzeug.LadungsBeschreibung With
                                              {.Menge = 5000, .Gut = "Wackeldackel"}})

    Private Class Fahrzeug
        Public Class LadungsBeschreibung
            Private _menge As Double
            Private _gut As String

            Public Property Menge() As Double
                        …
            End Property

            Public Property Gut() As String
                        …
            End Property
        End Class
```

```
        Private _kennzeichen As String
        Private _ladung As LadungsBeschreibung
        Private _hersteller As String

        Public Property Kennzeichen() As String
                …
        End Property

        Public Property Ladung() As LadungsBeschreibung
                …
        End Property

        Public Property Hersteller() As String
                …
        End Property
    End Class
```

… soll folgendes XML-Element erstellt werden:

```
<fuhrpark>
  <fahrzeug>
    <kennzeichen>MS-VB 2008</kennzeichen>
    <hersteller>MAN</hersteller>
    <ladung menge="10000">Salz</ladung>
  </fahrzeug>
  <fahrzeug>
    <kennzeichen>MS-C# 2008</kennzeichen>
    <hersteller>Mercedes-Benz</hersteller>
    <ladung menge="20000">Erdnüsse</ladung>
  </fahrzeug>
  <fahrzeug>
    <kennzeichen>MS-J# 2008</kennzeichen>
    <hersteller>DAF</hersteller>
    <ladung menge="5000">Wackeldackel</ladung>
  </fahrzeug>
</fuhrpark>
```

Das Anlegen der XML-Strukturen erfolgt dabei mit `XElement` und `XAttribute`:

```
        Dim fuhrpark = New XElement("fuhrpark", _
                From fzg In fahrzeugliste _
                Select New XElement("fahrzeug", _
                                    New XElement("kennzeichen", fzg.Kennzeichen), _
                                    New XElement("hersteller", fzg.Hersteller), _
                                    New XElement("ladung", fzg.Ladung.Gut,
                                         New XAttribute("menge", fzg.Ladung.Menge)) _
                                    )
                        )
```

Dem `XElement` mit dem Namen `fuhrpark` wird eine Liste von XElementen (`fahrzeug`) hinzugefügt. Diese Liste (genaugenommen `IEnumerable(Of XElement)`) wird mit *LINQ to Objects* erstellt. Das `fahrzeug`-Element wird zudem um die entsprechenden Unterelemente erweitert. Auch hier ist wieder die Verwendung von Literalen möglich:

```
        fuhrpark = <fuhrpark>
                       <%= From fzg In fahrzeugliste
                           Select <fahrzeug>
                                      <kennzeichen><%= fzg.Kennzeichen %></kennzeichen>
                                      <hersteller><%= fzg.Hersteller %></hersteller>
```

```
                                <ladung menge=<%= fzg.Ladung.Menge %>>
                                    <%= fzg.Ladung.Gut %>
                                </ladung>
                            </fahrzeug> _
                    %>
                </fuhrpark>
```

Abfragen von XML-Dokumenten mit LINQ to XML

LINQ bietet mit *LINQ to XML* auch eine Abfrageunterstützung für XML-Strukturen an.Ein Beispiel:

```
Dim fuhrpark = <?xml version="1.0"?>
               <fuhrpark>
                   <fahrzeug>
                       <kennzeichen>MS-VB 2008</kennzeichen>
                       <hersteller>MAN</hersteller>
                       <ladung menge="10000">Salz</ladung>
                   </fahrzeug>
                   <fahrzeug>
                       <kennzeichen>MS-C# 2008</kennzeichen>
                       <hersteller>Mercedes-Benz</hersteller>
                       <ladung menge="20000">Erdnüsse</ladung>
                   </fahrzeug>
                   <fahrzeug>
                       <kennzeichen>MS-J# 2008</kennzeichen>
                       <hersteller>DAF</hersteller>
                       <ladung menge="5000">Wackeldackel</ladung>
                   </fahrzeug>
               </fuhrpark>
Dim manFahrzeuge = From fahrzeug In fuhrpark...<fahrzeug> _
Where fahrzeug.<hersteller>.Value = "MAN"
For Each fahrzeug In manFahrzeuge
    Console.WriteLine(fahrzeug)
Next
```

Bei der Variablen `fuhrpark` handelt es sich um ein `XDocument`. In der `From`-Klausel wird auf alle XElemente mit dem Namen `fahrzeug` zugegriffen. Die `Where`-Klausel prüft den Hersteller des Fahrzeugs auf den Eintrag »MAN«. Der Wert eines XElements muss über die `Value`-Property erfragt werden.

Es werden zwei Zugriffmöglichkeiten für Elemente bereitgestellt:

Der Zugriff auf direkte Unterelemente erfolgt über `.<Elementname>`. Zudem kann auf darunter liegende Elemente mit `...<Elementname>` zugegriffen werden, wobei sich das Element durchaus tief in den XML-Strukturen befinden darf.

Auf den Inhalt eines Attributs kann über `.@<Attributname>` zugegriffen werden.

Beispiel: Es sollen alle schwer beladenen Fahrzeuge ermittelt werden:

```
Dim schwerBeladen = From fahrzeug In fuhrpark...<fahrzeug> _
                    Where CInt(fahrzeug.<ladung>.@menge) > 15000
For Each fahrzeug In schwerBeladen
    Console.WriteLine(fahrzeug)
Next
```

ergibt folgende Ausgabe:

```
<fahrzeug>
  <kennzeichen>MS-C# 2008</kennzeichen>
  <hersteller>Mercedes-Benz</hersteller>
  <ladung menge="20000">Erdnüsse</ladung>
</fahrzeug>
```

ACHTUNG **Aufgepasst bei der Typsicherheit bei LINQ to XML:** Auch wenn Sie – wie Sie beim Durchsehen der Beispieldateien erkennen können – für Ihr XML-Dokument mithilfe einer XSD-Datei die Typen des Dokuments definiert haben, gilt: Werte der LINQ-Abfragen bekommen Sie grundsätzlich als String zurück – das gilt sowohl für Elemente als auch für Attribute. Wie im Listing zu sehen, müssen Sie also stets entsprechende Casting-Operatoren einsetzen, um den wirklichen Typ abzufragen, und dabei können natürlich Fehler auftreten.

Achten Sie deswegen besonders darauf, gerade wenn Sie LINQ to XML verwenden, grundsätzlich typsicher zu entwickeln – `Option Strict` muss also eingeschaltet sein! Anderenfalls laufen Sie Gefahr, dass Sie Strings verarbeiten, obwohl Sie eigentlich numerische Typen vergleichen sollten, und dann kann es ruck, zuck passieren, dass 4.500 größer als 15.000 ist!

IntelliSense-Unterstützung für LINQ To XML-Abfragen

Bei der Arbeit mit XML-Daten ist jede Unterstützung der IDE sehr hilfreich. Visual Basic 2010 bietet genau diese an, wie in Abbildung 27.1 zu sehen.

Abbildung 27.1 XML-IntelliSense-Unterstützung

Diese Unterstützung erhält man, sobald eine XML Schema Definition (XSD)-Datei mit `Imports` in die Codedatei importiert wird. Aber fangen wir mal vorne an. Eine XSD-Datei kann Visual Studio selbst erzeugen. Hierzu erstellt man zunächst die komplette XML-Datei mit allen möglichen Ausprägungen, etwa wie in Abbildung 27.2 dargestellt.

```
fuhrpark2.xml
1  <?xml version="1.0"?>
2  <fuhrpark>
3    <fahrzeug>
4      <kennzeichen>MS-VB 2008</kennzeichen>
5      <hersteller>MAN</hersteller>
6      <ladung menge="10000">Salz</ladung>
7    </fahrzeug>
8    <fahrzeug>
9      <kennzeichen>MS-C# 2008</kennzeichen>
10     <hersteller>Mercedes-Benz</hersteller>
11     <ladung menge="20000">Erdnüsse</ladung>
12   </fahrzeug>
13   <fahrzeug>
14     <kennzeichen>MS-J# 2008</kennzeichen>
15     <hersteller>DAF</hersteller>
16     <ladung menge="5000">Wackeldackel</ladung>
17   </fahrzeug>
18 </fuhrpark>
```

Abbildung 27.2 Eine XML-Datei in Visual Studio

Danach kann man mithilfe des gleich lautenden Befehls im Menü *XML* ein *Schema erstellen* (Abbildung 27.3).

Abbildung 27.3 Erstellen einer XML Schema Definition (XSD)

Das erstellte Schema muss nun im Projektverzeichnis gespeichert und in das Projekt mit aufgenommen werden (Abbildung 27.4).

Abbildung 27.4 Hinzufügen einer Datei im Projektmappen-Explorer

In der soeben erstellten XSD-Datei muss nun noch ein Zielzamensbereich (»Targetnamespace«) angegeben werden. Dieser Namensbereich muss nicht existieren, sondern dient lediglich der Klassifikation. In diesem Fall wurde `http://mysample.local.de/fuhrpark2` verwendet (Abbildung 27.5).

Abbildung 27.5 Erweitern einer XSD-Datei um den Zielnamensbereich

In der Visual Basic-Datei, die eine XML-Unterstützung erhalten soll, kann jetzt der soeben festgelegte Zielnamensbereich importiert werden (Abbildung 27.6).

Abbildung 27.6 Hinzufügen einer *Imports*-Anweisung für XML- Schemadateien

Werden mehrere Namensräume verwendet, bietet es sich an, den Namensraum mit einem Präfix zu versehen. Das Präfix wird nach `xmlns` notiert und mit einem Doppelpunkt von diesem getrennt. In diesem Fall wird als Präfix `fuhrpark` verwendet. Bei nur einem Namensraum, der in die Visual Basic-Codedatei importiert wird, ist das nicht erforderlich.

```
Imports <xmlns:fuhrpark="http://mysample.local.de/fuhrpark2">
Imports <xmlns:artikel="http://mysample.local.de/artikel">
```

Verwendung von Präfixen (fuhrpark und artikel)

Die XML-Strukturen müssen in diesem Fall auch um die Präfixe erweitert werden.

```
        Dim fuhrpark = <fuhrpark:fuhrpark>
                           <fuhrpark:fahrzeug>
                               <fuhrpark:kennzeichen>MS-VB 2008</fuhrpark:kennzeichen>
                               <fuhrpark:hersteller>MAN</fuhrpark:hersteller>
                               <fuhrpark:ladung menge="10000">Salz</fuhrpark:ladung>
                           </fuhrpark:fahrzeug>
                           <fuhrpark:fahrzeug>
                               <fuhrpark:kennzeichen>MS-C# 2008</fuhrpark:kennzeichen>
                               <fuhrpark:hersteller>Mercedes-Benz</fuhrpark:hersteller>
                               <fuhrpark:ladung menge="20000">Erdnüsse</fuhrpark:ladung>
                           </fuhrpark:fahrzeug>
                           <fuhrpark:fahrzeug>
                               <fuhrpark:kennzeichen>MS-J# 2008</fuhrpark:kennzeichen>
                               <fuhrpark:hersteller>DAF</fuhrpark:hersteller>
                               <fuhrpark:ladung menge="5000">Wackeldackel</fuhrpark:ladung>
                           </fuhrpark:fahrzeug>
                       </fuhrpark:fuhrpark>
```

Abbildung 27.7 XML-Unterstützung der IDE

Kapitel 28

LINQ to Entities – Programmieren mit Entity Framework

In diesem Kapitel:

Voraussetzungen zum Test der Beispiele 799
LINQ to Entities – ein erstes Praxisbeispiel 811
Abfrage von Daten eines Entitätsmodells 818
Daten verändern, speichern, einfügen und löschen 830
Datenmodell aus Datenbank aktualisieren 838
Model First-Design 839
Vererbung im konzeptionellen Datenmodell 845
T-SQL-Befehle im Objektkontext direkt ausführen 849
Arbeiten mit gespeicherten Prozeduren 850
Ausblick 852

Als eine Entität bezeichnet man in der Datenmodellierung ein eindeutiges Objekt, dem Informationen zugeordnet werden können, und dass diese Begriffsdefinition nicht einer heftigen Abstraktheit entbehrt, impliziert schon, dass es sich bei LINQ to Entities um ein Verfahren handeln muss, dass das, was es erreichen will, auf eine sehr flexible Weise macht.

Dem ist auch so. Nur: Was macht es überhaupt?

Heutzutage werden viele datenorientierte Anwendungen auf der Grundlage von relationalen Datenbanken geschrieben. Diese Anwendungen müssen also irgendwann mit diesen Daten interagieren. Der logisch orientierte Aufbau von Datenbanktabellen und Datensichten eines üblichen Datenbanksystems, wie sie beispielsweise *Microsoft SQL Server*, *Oracle Database* oder *DB2* (von IBM) darstellen, bietet allerdings nur eine suboptimale Grundlage, die konzeptionellen Modelle objektorientiert entwickelter Anwendungen abzubilden: *Impedance Mismatch*, in etwa: *objekt-relationale Unverträglichkeit*, ist der Grund, dass Abbildungsversuche zwischen den beiden verschiedenen Welten eigentlich immer unvollkommen sein müssen, wenn nicht eine wirkliche objektorientierte Datenbank als Grundlage für die Speicherung der Anwendungsobjekte vorhanden ist – und das ist bei den genannten großen, kommerziellen Datenbanksystemen eben nicht der Fall.

Das Entitätsdatenmodell (*Entity Data Modell,* kurz: *EDM*) kann hier Abhilfe schaffen, da es ein konzeptionelles Datenmodell beschreibt, welches es erlaubt, durch flexibles Mapping eine Brücke zwischen den beiden Welten zu bauen.

LINQ to SQL macht das im Ansatz ebenso; auch LINQ to SQL hilft dem Benutzer, eine Reihe von Business-Objekten zu erstellen, die sich weitestgehend an den Tabellen- bzw. Sichtenschemata der Datenbank orientieren. Für den Entwickler eines Datenbanksystems mittlerer Größenordnung ist das sicherlich oftmals genau die Hilfe, die er benötigt, und in vielen Fällen ausreichend.

Doch die Zuordnungsmöglichkeiten zwischen dem logischen, relationalen Datenmodell der Datenbanken und dem konzeptionellen, objektorientierten Modell in Anwendungen gehen in LINQ to SQL einfach nicht weit genug, sowohl was die Zuordnung von Tabellen zu Businessobjekten anbelangt, als auch was die Abstraktion des oder der darunter liegenden Datenprovider betrifft, denn: LINQ to SQL kann lediglich mit den Microsoft SQL Server-Plattformen umgehen (mit den Versionen SQL Server 2000, 2005 und 2008), und Tabellen und Sichten können lediglich 1:1 auf ein Objektmodell abgebildet werden.

HINWEIS LINQ to SQL ist darüber hinaus eine Technik, die zwar für .NET Framework 4.0 nochmals überarbeitet wurde, und – sollte es notwendig sein – sicherlich auch noch weiterhin in Service Packs Berücksichtigung findet. Und für einige Zwecke ist LINQ to SQL auch sicher noch geeigneter als LINQ to Entities, insbesondere, weil es auch in der vorliegenden .NET Framework-Version beim Übertragen großer Datenmengen vom SQL-Server zum Client deutlich schneller ist als LINQ to Entities. Doch es überträgt die Daten leider auch nur von Microsoft SQL Server und wird anders als LINQ to Entities bzw. ADO.NET Entity Framework leider nicht mehr wesentlich um neue Funktionalitäten ergänzt.

Diese Unzulänglichkeiten löst LINQ to Entities, indem es eine weitere Schicht zwischen den generierten Business-Objekten und der ursprünglichen Datenbank einführt – die so genannte konzeptionelle Schicht. Dadurch wird LINQ to Entities gegenüber LINQ to SQL ungleich flexibler, aber auch nicht ganz so einfach zu handhaben wie LINQ to SQL.

Es gibt einige zusätzliche Konzepte, die man zum richtigen Nutzen kennen und begreifen muss. Diese Konzepte jedoch versteht und lernt man, wie so oft, am besten mit der Methode *Learning by Doing*. Und deswegen schauen wir uns in der folgenden Schritt-für-Schritt-Anleitung als Erstes an, wie man ADO.NET Entity Framework nutzt, um ein konzeptionelles Entitätsmodell für eine Datenbank zu erstellen und dieses anschließend einsetzt, um Daten abzufragen.

Die Elemente, die dabei verwendet werden, lernen Sie im entsprechenden Kontext kennen.

Voraussetzungen zum Test der Beispiele

Download und Installation von SQL Server 2008 Express Edition with Advanced Services

Um die Beispiele dieses Kapitels nachvollziehen zu können, benötigen Sie eine funktionsfähige SQL Server 2008-Instanz. Hierbei ist SQL Server 2008 in der Express Edition ausreichend, diese kostet nichts, und reicht für viele Datenbankanwendungen ohnehin aus. Sie können sie sogar mit Ihrer Anwendung, die Sie vielleicht auf Basis von LINQ to Entities entwickeln, weitervertreiben.

HINWEIS Die mit Visual Studio 2010 installierte Version von SQL Server 2008 Express können wir leider nicht verwenden, da die von Microsoft propagierte Beispieldatenbank AdventureWorks zu dieser leider nicht kompatibel ist. SQL Express with Advanced Services ist aber in jedem Fall die ohnehin bessere Version, die obendrein SQL Server Management Studio Basic zum Verwalten von SQL Server-Instanzen und -Datenbanken direkt mitinstalliert.

Falls Sie im Rahmen der Visual Studio 2010-Installation SQL Server 2008 Express bereits mitinstalliert haben, sollten Sie es über die Systemsteuerung (*Programme und Funktionen*) wieder deinstallieren. Die Software, die als Voraussetzung gilt und wie sie im folgenden Achtung-Kasten beschrieben wird, müssen Sie in diesem Fall aber nicht wieder installieren.

- *SQL Server 2008 Express Edition with Advanced Services* können Sie über *http://tinyurl.com/34px86k* herunterladen.

ACHTUNG Je nach Aktualität und bereits installierten Komponenten auf Ihrem Computer benötigen Sie ferner folgende Komponenten, die Sie herunterladen und installieren müssen, *bevor* Sie SQL Server 2008 Express Edition with Advanced Services installieren.

Stellen Sie sicher, dass Sie .NET Framework 3.5 mit Service Pack 1 installiert haben. Der entsprechende Link dazu lautet: *http://tinyurl.com/ycesrto*. Falls Sie Windows 7 einsetzen, ist .NET Framework 3.5 mit Service Pack 1 bereits installiert. Das gleiche gilt für Windows Server 2008 R2.

Laden Sie unter *http://tinyurl.com/6p5myz* den *Microsoft Installer 4.5* herunter und installieren Sie diesen. Beim Herausfinden des richtigen Dateinamens ist ein wenig Puzzelei angesagt: Für Windows Vista und Windows Server 2008 laden Sie die für Ihre Prozessorarchitektur relevante Version herunter, die mit *Windows 6.0* beginnt; Windows XP- und Windows Server 2003-Versionen sind entsprechend abweichend gekennzeichnet (bitte auch auf die Prozessorarchitektur achten! *x64* für 64-Bit-Betriebssysteme, *x86* für 32-Bit-Betriebssysteme verwenden). Für Vista 64 Bit laden Sie z.B. die Datei *Windows6.0-KB942288-v2-x64.msu* herunter.

Laden Sie unter *http://tinyurl.com/yk5h8l* PowerShell herunter, und installieren Sie diese Komponente ebenfalls. Das ist erforderlich, selbst wenn Sie Visual Studio 2010 zuvor installiert haben. Für Windows Server 2008 müssen Sie diesen Download nicht durchführen, da PowerShell hier bereits im System enthalten ist. Für Windows Vista laden Sie vom angegebenen Link die für Sie zutreffende Version, die unter *Windows Vista RTM* angegeben ist (*x86* für 32-Bit-, *x64* für 64-Bit-Betriebssystemversion), und für Windows XP und Windows 2003 die jeweils unter der gleichnamigen Kategorie für Ihre Busbreite aufgeführte Version. Denken Sie auch daran, die für Sie wichtige Sprachversion herunterzuladen.

Abbildung 28.1 Nach dem Start der SQL Server Express-Installationsdatei gelangen Sie zunächst in das SQL Server-Installationscenter. Führen Sie erst eine *Systemkonfigurationsprüfung* durch, bevor Sie die eigentliche Installation starten (jedoch NICHT, wenn Sie Windows 7 verwenden. In diesem Fall überspringen Sie diesen Schritt).

Haben Sie alle erforderlichen Komponenten installiert, können Sie sich an die Installation von *SQL Server 2008 Express Edition with Advanced Services* machen. Leider meldet Windows 7 bei dem Ausführen des Installationspakets Kompatibilitätsprobleme. Sie können die Installation trotzdem durchführen, müssen jedoch auf jeden Fall das Service Pack 1 für SQL Server 2008 nachinstallieren.

1. Zur Installation von *SQL Server 2008 Express Edition with Advanced Services* starten Sie die heruntergeladene Installationsdatei, und Sie sollten anschließend den Dialog vorfinden, den Sie auch in Abbildung 28.1 erkennen können.
2. Klicken Sie anschließend auf den Link *Systemkonfigurationsprüfung*, um abzuklären, ob alle Installationsvoraussetzungen, die Sie in der vorherigen Liste beschrieben finden, auf dem Zielsystem erfüllt sind. Bessern Sie im Bedarfsfall die fehlenden Komponenten nach.

ACHTUNG **Bitte ignorieren Sie diesen Schritt für Windows 7**. Sie sollten bereits eine Warnung erhalten haben, dass es mit dem Programm Probleme gibt. Ignorieren Sie diese Meldung zunächst, und starten Sie den Installationsvorgang dennoch. Sie müssen auf jeden Fall das Service Pack 1 für SQL Server 2008 nachinstallieren. Sie finden es unter *http://tinyurl.com/26k52uy*.

3. Klicken Sie anschließend im linken Bereich des Dialogs aus Abbildung 28.1 auf *Installation*. Der Inhalt des Dialogs ändert sich, etwa wie in Abbildung 28.2 zu sehen.

Abbildung 28.2 Klicken Sie im linken Bereich auf Installation und anschließend im rechten Bereich des Dialogs auf den oberen Link, um die Installation der Express-Edition von SQL Server 2008 zu starten

4. Klicken Sie hier auf *Neue eigenständige SQL Server-Installation oder Hinzufügen von Features zu einer vorhandenen Installation*, um die eigentliche Installation der Express-Edition zu starten.
5. Die Installation startet jetzt die Überprüfung der Setup-Unterstützungsregeln, die keine Fehler mehr ergeben sollte. Verlassen Sie den Dialog mit *OK*.
6. Klicken im SQL Server 2008-Setup-Assistenten, der jetzt erscheint (und sich vielleicht unter dem Ausgangsdialog befindet; holen Sie den Dialog im Bedarfsfall in den Vordergrund) auf *Weiter*, akzeptieren Sie die Lizenzbedingungen im nächsten Schritt und klicken Sie abermals auf *Weiter*.
7. Im Schritt *Setup-Unterstützungsdateien*, der jetzt wie in Abbildung 28.3 erscheinen sollte, klicken Sie auf *Installieren*. Dieser Vorgang kann einige Minuten in Anspruch nehmen.

Abbildung 28.3 Klicken Sie im Schritt *Setup-Unterstützungsdateien* auf *Installieren*, um die Setup-Unterstützungsdateien zu installieren

8. Sie sollten bei der anschließenden Zusammenfassung lediglich eine Windows-Firewall-Warnung erhalten. Diese Warnung können Sie ignorieren, solange Sie Ihre SQL Server-Datenbanken nur lokal, also von Ihrem Computer aus verwenden. Erst wenn Sie auf die SQL Server-Instanzen von einem anderen Computer aus über das Netzwerk zugreifen wollen, müssen Sie die entsprechenden Ports der Windows-Firewall öffnen (SQL Server-Standard: 1433). Klicken Sie daher auf *Weiter*, um zum nächsten Schritt zu gelangen.
9. Sie gelangen anschließend zur Featureauswahl (Abbildung 28.4). Hier bestimmen Sie, welche Features im Rahmen der Installation mitinstalliert werden sollen. Beziehen Sie die *Datenbankmoduldienste* und die *Verwaltungstools* in jedem Fall mit in die Installation ein. Die *SQL Server-Replikation* können Sie optional installieren, wenn Sie Inhalte der Datenbank zu Replikationszwecken regelmäßig in andere Datenbanken übertragen wollen.

WICHTIG Die von uns verwendete AdventureWorks-Datenbank funktioniert nur richtig, sofern Sie die Volltextsuche mit installieren.

Abbildung 28.4 In diesem Schritt wählen Sie die Komponenten aus, die Setup mitinstallieren soll. Die hier markierten müssen Sie in jedem Fall auswählen

10. Klicken Sie anschließend auf *Weiter*.

11. Bestimmen Sie im nächsten Dialog den Namen der SQL Server-Instanz, die Ihre Datenbanken später verwalten soll. Sie können die Voreinstellung *Benannte Instanz*: *SQLExpress* belassen, wie sie ist, wenn es nicht bereits eine SQL Server-Instanz unter diesem Namen auf demselben Computer gibt; dies wäre zum Beispiel dann der Fall, wenn Sie mit Visual Studio auch eine Version von SQL Server 2005 Express Edition installiert haben. Geben Sie dann einfach einen anderen, beliebigen Namen ein.

12. Klicken Sie anschließend auf *Weiter*.

13. Der nächste Schritt zeigt Ihnen eine Zusammenfassung der Installationsparameter. Wenn Sie mit diesen einverstanden sind, bestätigen Sie auch diesen Dialog mit *Weiter*.

14. Damit Sie die SQL Server-Instanz später beispielsweise mit dem Verbindungsdialog des Server-Explorers von Visual Studio finden können, stellen Sie, wie in Abbildung 28.5 zu sehen, den SQL Browser-Dienst auf *Automatisch*. Sie müssen in diesem Dialog ferner die Systemkonten bestimmen, in deren Kontext sowohl die Datenbank-Engine als auch der Browser-Dienst laufen sollen. Setzen Sie den Vorgang mit Klick auf *Weiter* fort.

Abbildung 28.5 Bestimmen Sie hier die Konten, in deren Kontext Datenbank-Engine und Browser-Dienst laufen sollen

15. Im nächsten Dialog (Abbildung 28.6) bestimmen Sie den Authentifizierungsmodus, mit dem sich Clients an der Instanz und an von ihr verwalteten Datenbanken anmelden können. Wenn Ihre Anwendung ausschließlich auf dem gleichen Computer läuft, der auch die SQL Server-Instanz enthält, oder SQL Server auf einem Computer betrieben wird, der Teil einer Active Directory-Domäne ist, wählen Sie die sichere Authentifizierungsmethode *Windows-Authentifizierungsmodus*. Ist keine Domäne vorhanden, müssen Sie zur Anmeldung an einer SQL Server-Instanz Anmeldeinformationen übergeben; das funktioniert nur, wenn die Instanz zuvor in den *gemischten Modus* (*Mixed Mode*) geschaltet wurde. In diesem Fall bestimmen Sie ebenfalls ein Systemadministrator-Kennwort (»sa«), mit dem Sie sich später an der Instanz anmelden können.

HINWEIS Ein sehr bekannter Virus (SQL Slammer) benutzte beim SQL Server 2000 die menschlich verständliche Schwäche, dass Benutzer damals zumeist KEIN Kennwort für das *sa*-Konto angaben; schon deshalb warnt das Setup hier deutlich. Vergeben Sie daher auf jeden Fall ein sicheres Kennwort (mindestens 6 Zeichen, Groß- und Kleinschreibung, Sonderzeichen und Zahlen, kein Wort der deutschen Sprache, also beispielsweise im Stil von »P@$$w0rd[1]«)

[1] Und auch nicht unbedingt dieses Beispielkennwort. ;-)

16. Klicken Sie anschließend auf *Weiter*.

Abbildung 28.6 Bestimmen Sie in diesem Dialog den Authentifizierungsmodus und fügen Sie der Datenbankinstanz mindestens einen Benutzer als SQL Server-Administrator hinzu

17. Bestätigen Sie den nächsten Schritt des Setup-Assistenten mit *Weiter*, nachdem Sie den Dialog nach Belieben eingestellt haben – die Optionen, die sich Ihnen hier bieten, sind durch ausreichende Hilfstexte beschrieben.

18. Sie sollten anschließend die Installationsregeln überprüft vorfinden, etwa wie in Abbildung 28.7 zu sehen. Danach sind die Datenbankdienste zur Installation bereit, und Sie können deren Installation beginnen, indem Sie abermals auf *Weiter* klicken, …

Abbildung 28.7 Die Installationsregeln sollten hier erfolgreich bestätigt worden sein; die Datenbankdienste sind nun zur Installation bereit

19. … nicht allerdings, bevor Sie sich im letzten Dialog die Zusammenfassung anschauen und mit einem beherzten Klick auf *Installieren* bestätigen müssen.

Die Installation dauert nun eine ganze Weile. Nach dem Abschluss der Installation werden Sie in den meisten Fällen aufgefordert, Ihren Computer neu zu starten. Machen Sie das, nachdem Sie zunächst den Dialog mit der Meldung *Der Setupvorgang ist abgeschlossen* mit *Weiter* bestätigt, im Abschlussdialog im entsprechenden Meldungsfeld auf *OK* (Abbildung 28.8) und dann auf *Schließen* geklickt haben. Der Computer wird nicht automatisch neu gestartet, Sie müssen den Vorgang nun selbst anstoßen, da das durch den Abschluss der Installation nicht automatisch initiiert wird.

TIPP Sollten Sie gerade das Kapitel eines Buchs schreiben, sorgen Sie dafür, dass das Kapitel abgespeichert ist, bevor Sie den Computer neu starten. Anderenfalls müssen Sie erfahrungsgemäß gegebenenfalls einige Absätze, die Sie bereits geschrieben haben, noch einmal schreiben. Das gilt natürlich auch für ähnliche andere Arbeiten mit Ihrem Computer, die Sie gerade durchführen, und die Sie in irgendeiner Form erhaltenswert finden.

Abbildung 28.8 Es ist vollbracht! Nach dem Neustart des Computers können Sie SQL Server 2008 Express with Advanced Services verwenden

Installation der AdventureWorks Beispieldatenbanken

Darüber hinaus brauchen Sie für die Beispiele dieses Kapitels auch die Beispieldatenbanken *AdventureWorks*.

HINWEIS Sie können die 2008er-Version der AdventureWorks-Beispiele nur auf dem großen SQL Server 2008 bzw. auf SQL Server 2008 Express with Advanced Services installieren, da nur in diesen Versionen die zur Installation notwendige Volltextsuche funktionell implementiert ist.

- **Beispieldatenbanken für SQL Server 2008 (für die Beispiele in diesem Buch)** Die Beispieldatenbanken für SQL Server 2008 finden Sie unter *http://tinyurl.com/25jqenv*. Sie können, wie schon erwähnt, diese nur dann installieren, wenn Sie eine der großen SQL Server-Versionen verwenden oder SQL Server 2008 with Advanced Services und die Komponente Volltextsuche mitinstalliert haben. Wenn Sie die AdventureWorks-Beispieldatenbanken herunterladen wollen, wird Ihnen nur *AdventureWorks 2008R2 RTM* angeboten. Diese Datenbanken können jedoch auch unter dem »normalen« SQL Server 2008 with Advanced Services installiert werden.

1. Wenn Sie die Installationsdateien der Beispieldatenbanken AdventureWorks heruntergeladen haben, starten Sie das Installerpaket durch Doppelklick.

Abbildung 28.9 Nach dem Start des Installerpakets begrüßt Sie das Setup der Beispieldatenbanken »AdventureWorks« mit diesem Dialog

2. Akzeptieren Sie die Lizenzbedingungen und klicken Sie dann auf *Next* (Abbildung 28.9).

Abbildung 28.10 Auswahl der verschiedenen Datenbankinstanzen, in die die AdventureWorks-Datenbanken eingeklinkt werden soll. Ignorieren Sie die Warnungen, sie sind für uns unwichtig.

3. Wählen Sie unter *Installation Instance* die richtige Instanz, in die die AdventureWorks Datenbank eingehängt werden soll. Zudem können Sie mit dem *Script Install Directory* festlegen, wo die Skripts für die Ersteinrichtung der Datenbank gespeichert werden sollen.
4. Wählen Sie alle Einträge bis auf *AdventureWorks OLTP* ab, und klicken Sie auf *Install* (Abbildung 28.10).

Abbildung 28.11 Das letzte Fenster bei der Installation der AdventureWorks-Datenbanken

Prinzipielle Funktionsweise eines Entity Data Model (EDM)

LINQ to Entities bezeichnet die Möglichkeit, mithilfe der Abfragetechnologie LINQ Datenentitäten abzufragen, die durch ein Entitätsdatenmodell (Entity Data Model) beschrieben werden. LINQ to Entities *ist* aber natürlich nicht das Entitätsdatenmodell, und LINQ to Entities beinhaltet auch nicht die Werkzeuge, dieses Modell zu erstellen.

Zuhause ist ein Entitätsdatenmodell in Entity Framework oder, genauer gesagt, in ADO.NET Entity Framework, und eine Elementvorlage dieses Typs fügen Sie auch ein, wenn Sie ein solches Modell Ihrem Projekt hinzufügen wollen.

Entity Framework möchte in seinem Modell sehr viel mehr erreichen:

- Der Datenprovider, der letzten Endes für die Kommunikation zwischen Ihrer Anwendung und der Datenbankserver zuständig ist, soll austauschbar sein, ohne dass sich Abfragen gegen das Entity-Modell wirklich ändern müssen.
- Die Tabellen, die das Datenbankmodell liefern, sollen sich nicht ausschließlich 1:1 gegen die sich ergebenden Businessobjekte mappen lassen müssen, sondern *beliebig* zuordbar sein.

Aus diesen Gründen basiert das Entity-Modell auf drei Einzelmodellen, nämlich aus dem …

- … **physischen Speichermodell** (*Storage Model*), das all das abbildet, was sich aus den Schemainformationen der eigentlichen Entitäten (Tabellen, Sichten, gespeicherte Prozeduren) der Datenbank ergibt und was zur Kommunikation mit der Datenplattform erforderlich ist.
- … **konzeptionellen Modell** (*Conceptual Model*), aus dem später die Business-Objekte generiert werden, und gegen das Sie mit LINQ to Entities abfragen.
- … **Mapping**, das die Zuordnungen zwischen den beiden Modellen regelt und definiert.

WICHTIG Ganz wichtig zu wissen in diesem Zusammenhang: Diese drei Modelle (bzw. diese zwei Modelle und das Mapping) werden beim Erstellen eines Entitätsdatenmodels (EDM) in Form einer XML-Datei aufgebaut, die die Dateierweiterung *.edmx* trägt – für *Entity Data Model eXtended markup language*. Beim Erstellen des Projekts landen Sie automatisch in drei verschiedenen XML-Dateien, die als eingebettete Ressourcen in der .NET-Assembly abgelegt werden. Auf diese Weise sind die Definitionen zwar nicht frei zugänglich, lassen sich aber durch Austauschen der Ressourcen anpassen. Und nicht nur das: Mehrere dieser Definitionen können natürlich ebenfalls vorhanden sein, denn der eigentliche Vorgang des Mappings zwischen den beiden Modellen findet nicht zur Kompilierungszeit, sondern zur Laufzeit statt – wie diese drei Dateien dabei zur Verwendung »bestimmt« werden, sehen Sie übrigens gleich im Praxisteil.

Diese drei Dateien werden übrigens in der Standardeinstellung genau wie der Gesamtcontainer genannt, unterscheiden sich aber durch ihre Dateierweiterungen:

- **.CSDL:** enthält die XML-Datei, die das konzeptionelle Modell beschreibt (*Conceptual Schema Definition Language*)
- **.SSDL:** enthält die XML-Datei, die das physische Speichermodell beschreibt (*Store Schema Definition Language*)
- **.MSL:** enthält die Mappingdatei (*Mapping Specification Language*)

TIPP **Einzelne Erzeugung der drei Modelldateien:** Falls Sie diese drei Dateien als Projektoutput doch in separaten Dateien ablegen möchten, weil Sie sie beispielsweise für weitere Tools benötigen, öffnen Sie den EDM-Designer. Selektieren Sie das Modell selbst, indem Sie auf einen freien Bereich im Designer klicken. Im Eigenschaftenfenster stellen Sie anschließend die Eigenschaft *Verarbeitung der Metadatenartefakte* auf *In Ausgabeverzeichnis kopieren*.

Abbildung 28.12 Um die Erstellung der drei Modelldateien ins Ausgabeverzeichnis zu forcieren, wählen Sie diese Einstellung

Mit diesen Voraussetzungen wissen wir vorerst ausreichend viel, um mit dem Aufbau des ersten Praxisbeispiels beginnen zu können.

LINQ to Entities – ein erstes Praxisbeispiel

Lassen Sie uns LINQ to Entities mit einem Praxisbeispiel nähern, anhand dessen wir dann die einzelnen Elemente erforschen können.

HINWEIS Denken Sie daran, dass Sie für das Nachvollziehen der Beispiele Zugriff auf eine Instanz von SQL Server haben müssen und dort die *AdventureWorks*-Beispiele eingerichtet sind.

1. Erstellen Sie ein neues Visual Basic-Projekt (als Konsolenanwendung).
2. Fügen Sie mithilfe des Projektmappen-Explorers ein neues Element in die Projektmappe ein – den entsprechenden Dialog erreichen Sie über das Kontextmenü des Projektnamens –, und wählen Sie für das neue Element, wie in Abbildung 28.13 zu sehen, die Vorlage *ADO.NET Entity Data Model* aus.
3. Nennen Sie das neue Datenmodell **AdventureWorks.edmx**.

Abbildung 28.13 Hinzufügen eines ADO.NET Entity Data Model

4. Schließen Sie den Dialog mit einem Klick auf *Hinzufügen* ab.
5. Im nächsten Schritt zeigt Visual Studio Ihnen einen Assistenten, der Ihnen hilft, die Modellinhalte entweder aus einer Datenbank generieren zu lassen oder ein leeres Modell zu erzeugen. Wählen Sie hier *Aus Datenbank generieren* (Abbildung 28.14).

Abbildung 28.14 Bestimmen Sie im ersten Schritt des Assistenten, ob Sie ein Modell aus der Datenbank generieren lassen möchten oder ein leeres Modell verwenden

6. Klicken Sie auf *Weiter*.

Abbildung 28.15 Bestimmen Sie in diesem Schritt die Entitätsverbindungszeichenfolge sowie wie und ob diese Verbindungszeichenfolge hinterlegt werden soll

7. In diesem Assistentenschritt (Abbildung 28.15) bestimmen Sie nun die Verbindung zu Ihrem Datenprovider. Mit anderen Treibern können Sie mithilfe der Schaltfläche *Neue Verbindung* auch andere Datenbank-Plattformen als den SQL Server-Datenprovider nutzen. Richten Sie Ihre Verbindung zur Datenbankinstanz ein, und wählen Sie an dieser Stelle die Datenbank aus, die Sie als Grundlage der ersten Schemaerstellung Ihres neuen Entitätsmodells erstellen lassen wollen. Für unser Beispiel verwenden wir wieder die Instanz der AdventureWorks-Datenbank, deren Einrichtung am Anfang dieses Kapitels beschrieben wurde.

HINWEIS Sie sehen jetzt in der Vorschau der Entitätsverbindungszeichenfolge, wie die Schema- bzw. Spezifikationsdateien des Entitätsmodells und die physische Verbindung zur Datenbankinstanz in einer Verbindungszeichenfolge zusammengefasst werden. Die Verbindungszeichenfolge für das EDM bestimmt also einerseits das konzeptionelle Modell, das Speichermodell sowie das Mapping und andererseits noch die verwendete Datenbank.

Sie können, falls Sie das wünschen, die Entitätsverbindungseinstellungen auch in der *App.Config*-Datei speichern und brauchen sich dann später, beim Instanziieren des Objektkontextes, der unter anderem den Verbindungsaufbau zur Datenbankinstanz managt, nicht mehr um das Übergeben dieser ellenlangen Zeichenfolge zu kümmern. Sie sollten wiederum dieses Verfahren nur dann anwenden, wenn Sie keine sicherheitskritischen Anmeldeinformationen in der Verbindungszeichenfolge verwenden; falls Sie auf die integrierte Windows-Sicherheit zur Anmeldung an einem Microsoft SQL Server-System zurückgreifen, befinden Sie sich auf der sicheren Seite. Sollten Sie SQL Server im gemischten Modus betreiben und Anmeldeinformationen in der SQL Server-Verbindungszeichenfolge übergeben müssen, ist es besser, die Verbindungszeichenfolge in der Anwendung zu hinterlegen oder die entsprechenden sicherheitsrelevanten Abschnitte der Datei *App.Config* zu verschlüsseln.[2]

8. Klicken Sie auf *Weiter*, um zum nächsten Schritt zu gelangen.

Abbildung 28.16 In diesem Schritt wählen Sie aus, welche Objekte als Grundlage für die Erstellung der Schema- und Mappingdateien dienen sollen

[2] Dieser MSDN-Forumsbeitrag, den Sie über *http://tinyurl.com/2vt9yko* erreichen, liefert wertvolle Tipps zur Vorgehensweise.

9. In diesem Schritt (Abbildung 28.16) bestimmen Sie, welche Datenbankobjekte als Grundlage für die Erstellung der Schema- und Mappingdateien dienen sollen. Im einfachsten Fall entspricht die Zuordnung des konzeptionellen Modells über ein 1:1-Mapping genau dem Speichermodell. Ihre Aufgabe im Bedarfsfall wird es dann immer sein, ein entsprechendes Mapping vorzunehmen, das für das generierte Objektmodell Ihrer Anwendung am besten geeignet ist. Für das Beispiel in diesem Kapitel wählen Sie folgende Tabellen aus:
   - **Address** (Adressendetails-Tabelle)
   - **Product** (Produktdetails-Tabelle)
   - **ProductVendor** (Lieferantenprodukte-Tabelle)
   - **Vendor** (Lieferanten-Tabelle)
   - **VendorAddress** (Lieferantenadressen-Tabelle)
10. Der Text des Kontrollkästchens *Generierte Objektnamen in den Singular oder Plural setzen* ist etwas unglücklich gewählt. In Visual Studio 2008 wurde noch von Pluralisierung gesprochen. Ist dieses Feld aktiviert, werden Entitätstypen in den Singular gesetzt, während die Entitätsmengen (auch EntitySet genannt) in den Plural gesetzt werden. Da diese Methode nur sinnvoll mit englischen Entitätsnamen und im Spracheinzugsgebiet des Ruhrgebiets funktioniert, sollten Sie dieses Kontrollkästchen nicht aktivieren, sonst werden aus *Mitarbeitern* ganz schnell *Mitarbeiters*.
11. Wird das Feld *Fremdschlüsselspalten in das Modell einbeziehen* aktiviert, werden neben den Navigationseigenschaften auch die entsprechenden IDs der Fremdschlüssel in den jeweiligen Entitätstyp integriert. Dies ist auch immer dann der Fall, wenn die referenzierte Tabelle nicht mit in das Entitätsmodell aufgenommen wird. Ein gutes Beispiel ist hier die Tabelle *VendorAddress*. Die Tabelle *AddressType* ist derzeit nicht im Modell aufgeführt. Daher besitzt die Tabelle *VendorAddress* das Feld *AddressTypeID*. Wenn Sie nun die Tabelle *AddressType* mit in das Modell aufnehmen, wird das Feld *AddressTypeID* nur dann generiert, wenn *Fremdschlüsselspalten in das Modell einbeziehen* aktiv ist. Die Navigationseigenschaft *AddressType* ist jedoch immer vorhanden.
12. Klicken Sie auf *Fertig stellen*, wenn Sie Ihre Auswahl getroffen haben.

Abbildung 28.17 Anders als beim O/R-Designer von LINQ to SQL können Sie beim Bearbeiten des Entitätsmodells die Zuordnung zwischen konzeptionellem Modell und Speichermodell beliebig umgestalten

In der Designer-Ansicht können Sie übrigens sehen, dass die Darstellung der Eigenschaften der einzelnen Entitäten wesentlich durchsichtiger ist als bei LINQ to SQL. Neben den Skalareigenschaften, die das Mapping auf die eigentlichen Felder (also Spalten) in der Datenbank ermöglichen, sind die Navigationseigenschaften gesondert aufgeführt. Mithilfe der Navigationseigenschaften einer Entität gelingt beispielsweise später, bei der Programmierung mit den korrelierenden Business-Objekten, der Zugriff auf verknüpfte Tabellen, auf Tabellen also, die durch das Speichermodell beschrieben in Relation zueinander stehen.

HINWEIS Der Designer ist dabei übrigens so clever, reine Hilfstabellen, die in der Datenbank dem ausschließlichen Zweck dienen, eine n:m-Verbindung herzustellen, komplett durch die Einstellungen in den Zuordnungsdetails des Mappings ganz aufzulösen. Das funktioniert natürlich dann nicht mehr, wenn es nur ein weiteres Feld in der Hilfstabelle gibt, das eine Skalareigenschaft enthält, wie es leider bei der AdventureWorks-Datenbank durch die `ModifiedDate`-Eigenschaft in jeder solchen Tabelle geschieht. Mithilfe der Northwind-Datenbank, die Sie natürlich trotz ihres fortgeschrittenen Alters immer noch an eine SQL Server 2008-Instanz knüpfen können, haben Sie aber die Möglichkeit, dieses automatische Wegmappen einer Hilfstabelle direkt in Aktion zu sehen.

TIPP Ein Anleitung, die die Northwind-Datenbank SQL Server 2008 installiert, findet man unter *http://tinyurl.com/2574mao*.

Und nachdem diese Grundvoraussetzungen erledigt sind, steht, wie in Abbildung 28.17 zu sehen, das erste Entitätsmodell für unsere zukünftigen Demos, und wir können uns den programmiertechnischen Aspekten zuwenden.

Nachträgliches Ändern des Entitätscontainernamens

In vielen Fällen mögen Sie EDMs so nennen, wie auch die Datenbank heißt. In vielen Fällen macht es aber Sinn, den Namen des Entitätscontainers zu ändern, aus denen sich auch der programmtechnische Objektkontext ergibt, den Sie in Ihren Anwendungen benötigen, um auf die Entitätsobjekte des konzeptionellen Modells beispielsweise für die Erstellung von Abfragen mit LINQ to Entities zuzugreifen.

Die Änderung ist dabei vergleichsweise simpel: Sie öffnen den EDM-Designer, klicken auf einen freien Bereich im Designer, um das Modell selbst »auszuwählen«, und können dann im Eigenschaftenfenster unter *Entitätscontainername* einen neuen Namen festlegen (Abbildung 28.18).

Abbildung 28.18 So ändern Sie den Namen eines Entitätscontainers im EDM-Designer

Gegebenenfalls wird es danach notwendig, auch die Datei *App.Config* anzupassen, falls Sie dort die Verbindungszeichenfolge hinterlegt haben (fett geschrieben im unten stehenden Auszug der Datei).

```
      <!--<add name="EventLog" type="System.Diagnostics.EventLogTraceListener"
initializeData="APPLICATION_NAME"/> -->
    </sharedListeners>
  </system.diagnostics>
  <connectionStrings><add name="AWEntities"
connectionString="metadata=res://*/AdventureWorks.csdl|res://*/AdventureWorks.ssdl|res://*/AdventureWork
s.msl;provider=System.Data.SqlClient;provider connection string=&quot;Data
Source=V64_LOEFFELDEV\SQL2008EXPRESS;Initial Catalog=AdventureWorks;Integrated
Security=True;MultipleActiveResultSets=True&quot;" providerName="System.Data.EntityClient" /><add
name="AWEntities"
connectionString="metadata=res://*/AdventureWorks.csdl|res://*/AdventureWorks.ssdl|res://*/AdventureWork
s.msl;provider=System.Data.SqlClient;provider connection string=&quot;Data
Source=V64_LOEFFELDEV\SQL2008EXPRESS;Initial Catalog=AdventureWorks;Integrated
Security=True;MultipleActiveResultSets=True&quot;" providerName="System.Data.EntityClient"
/></connectionStrings>
</configuration>
```

- Falls Sie einmal mit dem Designer nicht weiterkommen, können Sie die Datei einfach mit dem XML-Editor bearbeiten. Dazu wählen Sie aus dem Kontextmenü der *edmx*-Datei *Öffnen mit*.
- Wählen Sie im jetzt erscheinenden Dialog den Eintrag *XML-Editor* aus der Liste aus (Abbildung 28.19), und klicken Sie auf *OK*, um das Modell als XML-Datei zu editieren.

Abbildung 28.19 Falls der Entitäts-Designer einmal streiken sollte oder Dummheiten macht, schließen Sie das EDM in der Designansicht und öffnen Sie die *edmx*-Datei, die konzeptionelles Modell, Speichermodell und Mapping enthält, mit dem XML-Editor

Abbildung 28.20 Konzeptionelles Modell, Speichermodell und Mapping können Sie im Bedarfsfall auch als XML einsehen und bearbeiten. Bitte nehmen Sie keine Änderungen unterhalb des Abschnittes *EF Designer content* vor!

HINWEIS In Abbildung 28.20 sehen Sie die Abschnitte, in denen die Inhalte der SSDL-Datei (Speichermodell), CSDL-Datei (konzeptionelles Modell) und MSL (Mapping), die Sie bedenkenlos editieren können. Lassen Sie aber unbedingt die Finger vom Abschnitt, der sich unterhalb des Kommentars *EF Designer content (DO NOT EDIT MANUALLY BELOW HERE)* – zu Deutsch: »Ab hier nicht mehr händisch editieren« – befindet. Den hier vermerkten Hinweis sollten Sie sehr, sehr wörtlich nehmen!

Abfrage von Daten eines Entitätsmodells

Die Codedateien, die aus dem konzeptionellen Modell des Entitätsmodells entstehen, werden in einer Codedatei abgelegt, die Sie unterhalb des Datenmodells im Projektmappen-Explorer eingeordnet finden, wenn Sie den Projektmappen-Explorer *alle* Dateien anzeigen lassen.

Auch bei der Programmierung mit Entity Framework gibt es für die Business-Objekte, die aus dem konzeptionellen Modell entstehen, eine zentrale Verwalterklasse, die sich, anders als Sie es vielleicht von LINQ to SQL kennen, nicht *Datenkontext* sondern konsequenterweise *Objektkontext* nennt.

```
'''<summary>
''' No Metadata Documentation available.
'''</summary>
Partial Public Class AdventureWorksEntities
    Inherits ObjectContext
```

Der Objektkontext stellt zum einen die Businessobjekt-Abfrageauflistungen in Form von `ObjectSet(Of Type)`[3]-Klassen dar. Eine solche generische Klasse, wie im unten stehenden Listing zu sehen, repräsentiert also eine Abfrage, die eine Auflistung von Objekten eines bestimmten Typs zurückliefert, wenn diese Abfrage entweder mithilfe einer so genannten Entity SQL-Anweisung oder einer LINQ-Abfrage erstellt und ausgeführt wird. Dabei führt sie die Abfragen immer dann aus, wenn …

- … ihr Iterator verwendet wird, wenn Sie also beispielsweise mit `For Each` durch die Elemente ihrer Ergebnisauflistung hindurch iterieren,
- … sie an eine generische Liste (`List(Of Type)`)zugewiesen wird oder
- … wenn ihre `Execute`-Methode ausdrücklich aufgerufen wurde.

```
    ''' <summary>
    ''' No Metadata Documentation available.
    ''' </summary>
    Public ReadOnly Property Product() As ObjectSet(Of Product)
        Get
            If ( Product Is Nothing) Then
                 Product = MyBase.CreateObjectSet(Of Product)("Product")
            End If
            Return  Product
        End Get
    End Property
```

Die Business-Objekte werden, wie ebenfalls im Listing in der zweiten in Fettschrift gesetzten Zeile zu sehen, so definiert, dass bei der ersten Verwendung automatisch eine Instanz mit einer Abfrage definiert wird, die die gesamten Elemente der zugeordneten Entität (Tabelle, View) abruft.

Zum anderen steuert der Objektkontext natürlich auch den Aufbau zur Datenbank, managt die Änderungsverfolgung und stellt eine Infrastruktur zum Concurrency-Check[4] zur Verfügung.

---

[3] Die Klasse `ObjectSet(of Type)` basiert auf `ObjectQuery(of Type)`, welche unter Visual Studio 2008 / .NET Framework 3.5 noch Basis für Linq to Entities war.

[4] Überprüfen, ob Daten, die geändert wurden und nun in die Datenbank zurückgeschrieben werden sollen, gleichzeitig schon von jemand anderem geändert *und* schon zurückgeschrieben wurden.

Abfrage von Daten mit LINQ to Entities-Abfragen

Eine einfache Abfrage zeigt das folgende Listing, das sich in *Module1.vb* des im Folgenden verwendeten Beispielprojekts befindet.

BEGLEITDATEIEN Die Begleitdateien zu den folgenden Beispielen finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 28\LinqToEntitiesDemo
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
Module Module1

    Sub Main()

        'Wir verwenden in den Beispielen die Verbindungszeichenfolge aus der App.Config.
        'Ändern Sie die Zeichenfolge im Abschnitt ConnectionStrings, um die Beispiele
        'auf Ihre SQL Server-Instanz anzupassen. Sie müssen dazu nur den Provider-
        'ConnectionString innerhalb des Gesamt-ConnectionStrings anpassen.

        Dim awContext As New AWEntities()

        Dim lieferanten = From lieferant In awContext.Vendor _
                          Where lieferant.Name.StartsWith("H") _
                          Order By lieferant.Name

        For Each lieferant In lieferanten
            Console.WriteLine(lieferant.AccountNumber & ": " & lieferant.Name)
            Console.WriteLine(New String("="c, 70))

            'Zu jedem Lieferanten die Produkte ausliefern
            For Each produkt In lieferant.ProductVendor
                Console.Write("ID:" & produkt.ProductID & " ")
                Console.WriteLine("Name:" & produkt.Product.Name)
            Next
            Console.WriteLine()

        Next

    End Sub

End Module
```

Wenn Sie das Programm laufen lassen, werden Sie feststellen, dass es lediglich die folgende Ausgabe auf den Bildschirm zaubert:

```
HILLBIC0001: Hill Bicycle Center
======================================================================

HILLSBI0001: Hill's Bicycle Service
======================================================================

HOLIDAY0001: Holiday Skate & Cycle
======================================================================

HYBRIDB0001: Hybrid Bicycle Center
======================================================================
```

Die innere Schleife wird also gar nicht ausgeführt bzw. liefert keine Ergebnisse. Das liegt an der Voreinstellung für *Lazy Loading,* das standardmäßig deaktiviert ist. Der Begriff bedeutet etwa »faules Laden«, wobei in diesem Fall Daten von untergeordneten Tabellen erst beim Zugriff auf diese automatisch geladen werden, aber eben erst dann. Dieses Voreinstellung wird wohl auch darin begründet liegen, dass mit Visual Studio 2008 und .NET Framework 3.5 SP 1 die Konfiguration das Lazy Loading-Verhaltens vom Designer gar nicht unterstützt wurde. Stattdessen musste Lazy Loading manuell konfiguriert werden. Microsoft hat hier wohl den Nachbesserungsbedarf erkannt und hier Abhilfe geschaffen. Dem Ladeverhalten widmen wir uns später noch ausführlicher.

Zu guter Letzt erlaubt es Entity Framework auch leider nicht, auf einfache Weise Debug-Ausgaben auf der Konsole auszugeben. Alternativ können Sie sich, wenn Sie beispielsweise die Developer Edition von SQL Server zur Hand haben,[5] hier mit SQL Profiler aus der Patsche helfen. Wie Sie SQL Profiler verwenden, um sich generiertes SQL anzuschauen (Hilfe zur Selbsthilfe kann bei Entity Framework nicht schaden!), finden Sie im Abschnitt »Generierte SQL-Anweisungen unter die Lupe nehmen« ab Seite 821 beschrieben.

HINWEIS Wie bei LINQ üblich, werden Abfragen auch bei LINQ to Entites verzögert ausgeführt, und sie lassen sich auch kaskadieren. Das LINQ to Objects-Kapitel 26 bietet hier ausreichend Beispiele. Auch bei LINQ to Entities gilt: Sie können eine Abfrageergebnisliste beispielsweise mit `ToList` oder `ToArray` von ihrem Abfrageobjekt lösen. Im Gegensatz zu LINQ to SQL ist die Liste damit aber nicht automatisch von ihrem Datenkontext gelöst, sondern nur von der Abfragekette, die dann unterbrochen ist.

Wie Abfragen zum Datenprovider gelangen – Entity SQL (eSQL)

Anders als bei LINQ to SQL werden in Entity Framework LINQ-Abfragen nicht direkt zum Datenprovider geschickt, der sich übrigens mit vollem Namen *ADO.NET Entity Client Data Provider* nennt. Vielmehr haben die Redmonder Entwicklerkollegen einen Dialekt namens Entity SQL entwickelt, der das konzeptionelle Modell eines EDM abfragt. Dadurch, dass eSQL nicht direkt auf dem Datenprovider arbeitet, wird eSQL datenbankplattformunabhängig – wenn Sie später also Ihre Programme an andere Datenprovider anpassen müssen, ist der Code, der ja gegen das konzeptionelle Modell des EDM abfragt, davon nicht betroffen. Im günstigsten Fall. Im ungünstigsten Fall kann es sein, dass Sie bei Abfragen gegen Ihr Entitätsmodell Features verwendet haben, die der eine oder andere Datenbanktreiber – wenn er denn erst einmal verfügbar ist – vielleicht nicht umsetzen kann. Das jedoch ist bei den Providern für die großen Anbieter von Datenbanksystemen (Oracle, IBM etc.) wohl eher nicht zu erwarten.

Anpassen des Namens der Entitätenmenge

Der Name einer Entitätsmenge ist mit dem Entitäts-Designer übrigens schnell geändert. Wenn es Ihnen nicht passt, dass aus einer Tabelle *Vendor* die Abfrageeigenschaft `Vendor` des Objektkontexts hervorgegangen ist und Sie den Namen lieber (ordentlich) pluralisiert hätten, ändern Sie den Namen einfach:

[5] Und für professionelle Entwicklungsprojekte sollten Sie die zur Hand haben – im Rahmen eines MSDN-Abos ist diese zum Beispiel verfügbar.

1. Klicken Sie doppelt auf das EDM im Projektmappen-Explorer, um den Designer zu öffnen.
2. Klicken Sie die Entität (zum Beispiel *Vendor*) an.
3. Ändern Sie die Eigenschaft *Name der Entitätenmenge* im Eigenschaftenfenster dementsprechend (Abbildung 28.21).

HINWEIS Um die folgenden Beispiele eins zu eins übernehmen zu können, sollten Sie die Umbenennung *nicht* ausführen. Sollten Sie dennoch umbenennen, müssen Sie die Quellcodes entsprechend anpassen.

Abbildung 28.21 Der Name einer Entitätenmenge lässt sich mit dem EDM-Designer in Nullkommanichts ändern

HINWEIS Die automatische Pluralisierung, die im »Assistent für Entity Data Model« über »Generierte Objektnamen in den Singular oder Plural setzen« gesteuert werden kann, wendet – wie bereits an früherer Stelle erwähnt – nur eine englischsprachige Pluralisierung an, die für deutsche Bezeichner nicht immer ein gutes Ergebnis liefert (etwa Mitarbeiter – Mitarbeiters).

Generierte SQL-Anweisungen unter die Lupe nehmen

In LINQ to SQL können Sie es praktisch ohne Aufwand erreichen, dass die generierten T-SQL-Abfragen, die sich aus einer LINQ to SQL-Abfrage ergeben, automatisch zu protokollieren. Mit LINQ to Entities geht das nicht so einfach. Aber es geht. Sie haben zwei Möglichkeiten:

- Die erste Variante besteht darin, Ihre Abfrage in das ursprüngliche Abfrageobjekt des Objektkontexts zu casten und dann die Methode `ToTraceString` aufzurufen, um herauszufinden, welche Befehle Entity Framework *plant*, zum Datenprovider zu schicken. Sie könnten also das obige Beispielprogramm um diese Zeilen ergänzen …

```
.
.
.
        Dim awContext As New AWEntities()

        Dim lieferanten = From lieferant In awContext.Vendor _
                          Where lieferant.Name.StartsWith("H") _
                          Order By lieferant.Name

        'Mit dieser Anweisung können wir rausfinden,
        'welcher SQL-Text zur Anwendung kommen wird.
        Console.WriteLine(CType(lieferanten, ObjectQuery(Of Vendor)).ToTraceString)
        Console.WriteLine()
.
.
.
```

… um die Ausgabe um folgende Zeilen zu ergänzen:

```
SELECT
[Extent1].[VendorID] AS [VendorID],
[Extent1].[AccountNumber] AS [AccountNumber],
[Extent1].[Name] AS [Name],
[Extent1].[CreditRating] AS [CreditRating],
[Extent1].[PreferredVendorStatus] AS [PreferredVendorStatus],
[Extent1].[ActiveFlag] AS [ActiveFlag],
[Extent1].[PurchasingWebServiceURL] AS [PurchasingWebServiceURL],
[Extent1].[ModifiedDate] AS [ModifiedDate]
FROM [Purchasing].[Vendor] AS [Extent1]
WHERE [Extent1].[Name] LIKE N'H%'
ORDER BY [Extent1].[Name] ASC

HILLBIC0001: Hill Bicycle Center
====================================================================

HILLSBI0001: Hill's Bicycle Service
====================================================================

HOLIDAY0001: Holiday Skate & Cycle
====================================================================

HYBRIDB0001: Hybrid Bicycle Center
====================================================================


Taste drücken zum Beenden
```

- Die zuverlässigere, aber aufwändigere Variante, da sie den tatsächlichen *Ist*-Zustand widerspiegelt: Sie verwenden das Profiler-Werkzeug einer großen SQL Server-Version. Wenn Sie beispielsweise die Möglichkeit haben, in Ihrer Firma auf eine Vollversion von SQL Server per Remote-Desktop zuzugreifen, haben Sie auch die Möglichkeit, den Profiler mit der Instanz Ihrer lokalen SQL Express-Version zu verbinden. Das Profiler-Werkzeug gibt Ihnen genau Auskunft über die gesendeten Daten zwischen der Datenbankinstanz und der auf dem Client geöffneten Session. Ein Beispiel sehen Sie in Abbildung 28.22.

Abbildung 28.22 Das Profiler-Tool eines großes SQL-Server können Sie auch auf Ihre Express-Instanz ansetzen. Dabei muss sich der Profiler nicht mal auf dem gleichen Computer befinden – wenn es Ihre Netzwerk- und Firewalleinstellungen zulassen.

TIPP Damit Sie den Profiler des großen Visual Studio mit Ihrer Express-Instanz verbinden können, muss Ihre Firewall das erlauben und Sie müssen natürlich auch die Netzwerkprotokolle entsprechend eingerichtet haben. Auf dieses Thema möchten wir im Rahmen dieses Buches jedoch nicht näher eingehen – entsprechende Infos bingen oder googeln Sie am besten unter dem Stichwort *SQL Server-Konfigurations-Manager*.

Lazy Loading und Eager Loading in Entity Framework

Mit .NET Framework 3.5 SP1 unterschied sich das Ladeverhalten von LINQ to SQL ganz erheblich von Linq to Entities. Mit .NET Framework 4.0 wurde nun der Generator erweitert, sodass er auch Code für Lazy Loading generieren kann. Damit das *Laden von Daten im Bedarfsfall*, wie man Lazy Loading auch umschreiben könnte, auch zum Tragen kommt, müssen Sie im Designer entsprechende Einstellungen vornehmen.

Wechseln Sie dazu in den Modell-Designer unseres bisherigen Beispielprojekts, und klicken Sie auf einen freien Bereich, um das Modell selbst zu selektieren. Das Eigenschaftenfenster zeigt nun die Eigenschaften des konzeptionellen Entitätsmodells (in unserem Beispiel hat es den Namen *AdventureWorksModel*).

Abbildung 28.23 Im Entitäts-Designer kann ab dem .NET Framework 4.0 für das gesamte Entitätsmodell pauschal Lazy Loading aktiviert werden

- Sobald *Lazy Loading aktiviert* auf *True* gesetzt wird (Abbildung 28.23), werden Daten automatisch nachgeladen (genauer: Der Generator erzeugt Code, um die Daten beim Zugriff auf die entsprechenden Navigationseigenschaften automatisch nachzuladen). Sollten Sie – aus welchen Gründen auch immer – diese Einstellung nicht nutzen wollen, können Sie die Daten auch manuell nachladen, wie der folgende Abschnitt beschreibt.

Daten beim Lazy Loading manuell nachladen

Im Beispielprojekt der Anwendung finden Sie ein weiteres Modul mit dem Namen *LazyEagerLoading.vb*. Es demonstriert, wie Sie vorgehen müssen, um Daten aus Tabellenrelationen manuell nachladen zu können.

HINWEIS Damit das Beispielprojekt jetzt mit dieser `Main`-Methode startet, stellen Sie bitte in den Projekteigenschaften das entsprechende Startobjekt ein. Die Projekteigenschaften erreichen Sie, indem Sie im Projektmappen-Explorer den Projektnamen mit der rechten Maustaste anklicken, *Eigenschaften* aus dem Kontextmenü wählen und das *Startobjekt* entsprechend ändern (etwa wie in Abbildung 28.24 zu sehen).

Abbildung 28.24 So bestimmen Sie, in welchem Modul die *Sub Main* zum Starten der Anwendung aufgerufen werden soll

```
Imports System.Data.Objects

Module LazyEagerLoading

   Sub ManuellesNachladen()
        Dim awContext As New AWEntities()

        Dim lieferanten = From lieferant In awContext.Vendor _
                          Where lieferant.Name.StartsWith("H") _
```

```
                        Order By lieferant.Name

        For Each lieferant In lieferanten
            Console.WriteLine(lieferant.AccountNumber & ": " & lieferant.Name)
            Console.WriteLine(New String("="c, 70))

            If Not lieferant.ProductVendor.IsLoaded Then
                lieferant.ProductVendor.Load()

                'Zu jedem Lieferanten die Produkte ausliefern
                For Each produkt In lieferant.ProductVendor
                    If Not produkt.ProductReference.IsLoaded Then
                        produkt.ProductReference.Load()
                        Console.Write("ID:" & produkt.ProductID & " ")
                        Console.WriteLine("Name:" & produkt.Product.Name)
                    End If
                Next
                Console.WriteLine()
            End If

        Next

        Console.WriteLine()
        Console.WriteLine("Taste drücken zum Beenden")
        Console.ReadKey()

    End Sub

End Module
```

Von Relevanz sind hier drei Dinge:

- Sie stellen mit `IsLoaded` fest, ob eine bestimmte Entität, die eine Verknüpfung von oder zu einer anderen darstellt, schon geladen wurde.
- Sie laden die Inhalte der Verknüpfung mit der `Load`-Methode nach.
- Zu einem »:1«-Endpunkt besteht immer eine *Referenz*. Das bedeutet, dass Sie nicht die eigentliche Entitätsnameneigenschaft zum Nachladen der Daten verwenden, sondern den Namen mit der Endung `Reference`. Da die Tabelle *ProductVendor* (alle Lieferantenprodukte) mit der Tabelle *Product* (Produktdetails) 1:*1* verknüpft ist, verwenden Sie also `Product`*`Reference`* zum Laden der Inhalte. Da hingegen *Vendor* und *ProductVendor* 1:*n* verknüpft sind (erster fett gesetzter Codeblock im Listing), verwenden Sie hier *nicht* das Postfix *Reference*.

Eager Loading für bestimmte Relationen verwenden

Die andere Möglichkeit, die in bestimmten Szenarien sinniger ist – insbesondere dann, wenn vergleichsweise wenig Daten von vornherein zur Verfügung stehen sollen –, besteht darin, das Entity Framework anzuweisen, die Daten für die untergeordneten Tabellen in einem Rutsch durch eine JOIN-Verknüpfung sofort mitzuladen.

Sie erreichen das durch die Verwendung der `Include`-Methode, der Sie den Pfad der untergeordneten Tabellen einfach durch ».« getrennt mitgeben. Wenn Sie also die Tabelle *Vendor* abfragen, und möchten, dass *ProductVendor*-Daten und die denen wiederum zugeordneten *Product*-Daten automatisch mitgeliefert werden, dann lautet der Suchpfad eben »ProductVendor.Product.«

WICHTIG Die Ausgangstabelle (im Beispiel *Vendor*) dürfen Sie nicht mit im Suchpfad angeben.

Das auf diese Weise geänderte Beispiel sieht dann folgendermaßen aus:

```
    Sub EagerLoading()
        Dim awContext As New AWEntities()

        Dim lieferanten = From lieferant In awContext.Vendor.Include("ProductVendor.Product") _
                          Where lieferant.Name.StartsWith("H") _
                          Order By lieferant.Name

        'Mit dieser Anweisungen können wir rausfinden,
        'welcher SQL-Text zur Anwendung kommen wird.
        Console.WriteLine(CType(lieferanten, ObjectQuery(Of Vendor)).ToTraceString)
        Console.WriteLine()

        For Each lieferant In lieferanten
            Console.WriteLine(lieferant.AccountNumber & ": " & lieferant.Name)
            Console.WriteLine(New String("="c, 70))


            'Zu jedem Lieferanten die Produkte ausliefern
            For Each produkt In lieferant.ProductVendor
                Console.Write("ID:" & produkt.ProductID & " ")
                Console.WriteLine("Name:" & produkt.Product.Name)
            Next
            Console.WriteLine()

        Next

        Console.WriteLine()
        Console.WriteLine("Taste drücken zum Beenden")
        Console.ReadKey()
    End Sub
```

Damit ändert sich dementsprechend auch der generierte SQL-Text, der dann wie folgt ausschaut (gekürzt):

```
SELECT
[Project1].[VendorID] AS [VendorID],
[Project1].[AccountNumber] AS [AccountNumber],
… <Ausgabe gekürzt> …
 [Project1].[rowguid] AS [rowguid],
[Project1].[ModifiedDate2] AS [ModifiedDate2]
FROM ( SELECT
        [Extent1].[VendorID] AS [VendorID],
 … <Ausgabe gekürzt> …
        [Extent1].[ModifiedDate] AS [ModifiedDate],
        [Join1].[ProductID1] AS [ProductID],
         … <Ausgabe gekürzt> …
        [Join1].[ModifiedDate2] AS [ModifiedDate2],
        CASE WHEN ([Join1].[ProductID1] IS NULL) THEN CAST(NULL AS int) ELSE 1
ND AS [C1]
        FROM  [Purchasing].[Vendor] AS [Extent1]
        LEFT OUTER JOIN  (SELECT [Extent2].[ProductID] AS [ProductID1], [Exten
```

```
].[VendorID] AS [VendorID], [Extent2].[AverageLeadTime] AS [AverageLeadTime],
… <Ausgabe gekürzt> …
]
VendorID]
        WHERE [Extent1].[Name] LIKE N'H%'
)  AS [Project1]
ORDER BY [Project1].[Name] ASC, [Project1].[VendorID] ASC, [Project1].[C1] ASC

HILLBIC0001: Hill Bicycle Center
====================================================================
ID:524 Name:HL Spindle/Axle

… <Ausgabe gekürzt> …

HYBRIDB0001: Hybrid Bicycle Center
====================================================================
ID:910 Name:HL Mountain Seat/Saddle


Taste drücken zum Beenden
```

Anonymisierungsvermeidung bei Abfragen in verknüpften Tabellen

Sie können bei LINQ to Entites `Join`-Abfragen verwenden, um Daten aus mehreren Tabellen in einer flachen Ergebnisliste zusammenzuführen. In vielen Fällen möchten Sie aber, dass Sie nur deshalb auf verknüpfte Tabellen zugreifen, um bestimmte Datensätze in der Haupttabelle in Abhängigkeit bestimmter Werte der Tabellen zu selektieren, mit denen die Haupttabelle verknüpft ist. Und in diesen Fällen können Sie `Join` nicht gebrauchen, weil `Join` Ihnen automatisch eine Ergebnisliste auf Basis einer anonymen Klasse zurückliefert. Sie möchten in diesen Fällen aber in der Regel eine Ergebnisliste auf Basis einer konkreten Entitätsklasse zurückgeliefert bekommen.

Ein Beispiel: Angenommen, Sie möchten alle Lieferanten ermitteln, die über mindestens ein Produkt verfügen, dessen Name mit *W* beginnt. Das Formulieren dieser Abfrage unter Zuhilfenahme von `Join` wäre, nachdem Sie sich das LINQ to Objects-Kapitel 26 zu Gemüte geführt haben, sicherlich kein Problem.

Doch die Schwierigkeit dabei ist: Sie erhalten, wie eben gesagt, eine Ergebnisliste zurück, die sich auf eine anonyme Klasse stützt. Sie möchten aber eine Ergebnisliste auf Basis von `Vendor` (*Lieferant*) zurückgeliefert bekommen. In diesem Fall verwenden Sie eine Abfragemethode, die Sie im Modul *SpezielleAbfrageMethoden.vb* im Beispielprojekt finden und die mehrere `From`-Abfragen ineinander verschachtelt und schließlich mit `Distinct` dafür sorgt, dass Dubletten in der Ergebnisliste ausgeschlossen werden.

HINWEIS Denken Sie auch hier wieder daran, das Startobjekt auf das entsprechende Modul im Beispielprojekt zu ändern, also in diesem Fall auf *SpezielleAbfrageMethoden*.

HINWEIS Das folgende Beispiel verwendet übrigens Lazy Loading mit manuellem Nachladen der Datensätze, wie wir es im vorherigen Abschnitt kennen gelernt haben, um die untergeordneten Datensätze zu ermitteln.

```
    Sub AnonymisierungsvermeidungBeiVerknüpftenAbfragen()

        Dim awContext As New AWEntities()
```

```
        'Fragt alle Lieferanten ab, die Produkte haben,
        'deren Namen ihrer Produktdetailsbeschreibungen mit "W" beginnen.
        Dim lieferanten = From lieferantItems In awContext.Vendor _
        From lieferantProduktItem In lieferantItems.ProductVendor _
            From productItem In awContext.Product _
                Where lieferantItems.VendorID = lieferantProduktItem.VendorID _
                    Where productItem.ProductID = lieferantProduktItem.ProductID _
                        Where productItem.Name.StartsWith("W") _
                            Select lieferantItems Distinct

        'Manuelles Lazy-Loading für die Gesamtausgabe verwenden:
        For Each lieferant In lieferanten
            Console.WriteLine(lieferant.AccountNumber & ": " & lieferant.Name)
            Console.WriteLine(New String("="c, 70))

            If Not lieferant.ProductVendor.IsLoaded Then
                lieferant.ProductVendor.Load()

                'Zu jedem Lieferanten die Produkte ausliefern
                For Each produkt In lieferant.ProductVendor
                    If Not produkt.ProductReference.IsLoaded Then
                        produkt.ProductReference.Load()
                        Console.Write("ID:" & produkt.ProductID & " ")
                        Console.WriteLine("Name:" & produkt.Product.Name)
                    End If
                Next
                Console.WriteLine()
            End If

        Next

        Console.WriteLine()
        Console.WriteLine("Taste drücken zum Beenden")
        Console.ReadKey()

    End Sub
```

Wenn wir diese Methode laufen lassen, liefert sie die gewünschte Ergebnisliste, wie im folgenden Bildschirmauszug zu sehen:

```
GREENLA0001: Green Lake Bike Company
====================================================================
ID:870 Name:Water Bottle - 30 oz.
ID:871 Name:Mountain Bottle Cage

… <Ausgabe gekürzt> …

ID:854 Name:Women's Tights, L
ID:855 Name:Men's Bib-Shorts, S
ID:856 Name:Men's Bib-Shorts, M
ID:857 Name:Men's Bib-Shorts, L


Taste drücken zum Beenden
```

Kompilierte Abfragen

LINQ to Entities ermöglicht es aus Gründen der Performanceoptimierung, so genannte optimierte Abfragen einzusetzen. Eine kompilierte Abfrage besteht quasi aus einem Delegat zu einer vorkompilierten Instanz eines `ObjectQuery(Of Type)`-Objekts, die dann eine entsprechende Ergebnisliste zurückliefert – natürlich wieder vom Typ `IQueryable(Of Type)`. Mithilfe des Einsatzes entsprechender Parameter in dem dazu notwendigen Lambda-Ausdruck, der beim Einrichten einer kompilierten Abfrage dem Konstruktor der `CompiledQuery`-Klasse übergeben werden muss, können Sie dabei kompilierte Abfragen für verschiedene Abfrageparameter wiederverwendbar gestalten.

Das folgende Beispiel demonstriert den Einsatz einer kompilierten Abfrage, die Lieferanten mit einem bestimmten Anfangsbuchstaben ermittelt. Diese Abfrage wird parametrisiert kompiliert, und sie kann durch Übergabe des Anfangsbuchstabens beim Aufruf der Abfrage immer wieder eingesetzt werden:

```
Sub KompilierteAbfragen()

    Dim awContext As New AWEntities()

    Dim kompilierteLieferantenAbfrage = CompiledQuery.Compile(Of AWEntities, _
                                        String, IQueryable(Of Vendor))( _
                    Function(context, Anfangsbuchstabe) From lieferant In context.Vendor _
                            Where lieferant.Name.StartsWith(Anfangsbuchstabe) _
                            Order By lieferant.Name _
                            Select lieferant)

    Dim abfrageNach_H = kompilierteLieferantenAbfrage(awContext, "H")

    For Each lieferant In abfrageNach_H
        Console.WriteLine(lieferant.AccountNumber & ": " & lieferant.Name)
        Console.WriteLine(New String("="c, 70))

        'Zu jedem Lieferanten die Produkte ausliefern
        For Each produkt In lieferant.ProductVendor
            Console.Write("ID:" & produkt.ProductID & " ")
            Console.WriteLine("Name:" & produkt.Product.Name)
        Next
        Console.WriteLine()
    Next

    Dim abfrageNach_W = kompilierteLieferantenAbfrage(awContext, "W")

    For Each lieferant In abfrageNach_W
        Console.WriteLine(lieferant.AccountNumber & ": " & lieferant.Name)
        Console.WriteLine(New String("="c, 70))

        'Zu jedem Lieferanten die Produkte ausliefern
        For Each produkt In lieferant.ProductVendor
            Console.Write("ID:" & produkt.ProductID & " ")
            Console.WriteLine("Name:" & produkt.Product.Name)
        Next
        Console.WriteLine()
    Next
```

```
        Console.WriteLine()
        Console.WriteLine("Taste drücken zum Beenden")
        Console.ReadKey()

    End Sub

End Module
```

Die erste fett gesetzte Zeile erstellt die kompilierte Abfrage mithilfe einer `CompiledQuery`-Klasse. Diese wird beim Instanziieren auf den Typ des Objektkontexts festgelegt, auf den Typ des Parameters sowie den Rückgabetyp. Der übergebende Lambda-Ausdruck definiert dann nach diesem Typschema die eigentliche Abfrage, die Sie mit dem Lambda-Ausdruck als LINQ-Abfrage definieren.

HINWEIS Mit diesem Schritt haben Sie nicht die Abfrage selbst definiert, sondern nur einen Funktionsdelegaten, der eine Abfrage zurückliefert, wenn Sie ihm die entsprechenden Parameter übergeben.

Die Nutzung der Abfrage ist dann anschließend leicht: Sie definieren eine weitere Abfragevariable (die dann natürlich selbst wieder vom Typ `IQueryable(Of Type)` definiert sein wird), die die vorkompilierte Abfrage zurückliefert, auf die der Funktionsdelegat zeigt, und bei der Zuweisung übergeben Sie die Parameter für die Abfragedurchführung. Wie das in der Praxis genau ausschaut, sehen Sie anhand der beiden weiteren fett gesetzten Codezeilen im Beispiellisting.

Daten verändern, speichern, einfügen und löschen

Wie Sie es vielleicht von LINQ to SQL schon kennen, ist auch bei LINQ to Entities das Abfragen von Daten die eine, das Modifizieren von Daten aber eine ganz andere Geschichte. Auf welche Weise Sie Daten abfragen, haben die letzten Abschnitte von verschiedenen Seiten beleuchtet. Alle Operationen, die sich mit dem Ändern von Daten beschäftigen, sind Thema der folgenden Abschnitte.

Während bei LINQ to SQL der Datenkontext mit den an ihm hängenden `Table(Of )`-Auflistungen für die Änderungsverfolgung zuständig ist, übernehmen diese Aufgabe in LINQ to Entities die Entitätsobjekte selbst. Das können sie deswegen, weil diese nicht nur einfache POCOs (*Plain Old CLR Objects* – also reine, nur von `Object` abgeleitete Business-Objekte) – darstellen, sondern wenn Sie mit den Standardprojektvorlagen arbeiten, von der Klasse `EntityObject` abgeleitet sind.[6]

Die folgenden Abschnitte beschreiben, wie Sie mithilfe von Entity Framework Datenänderungen in dem durch das EDM vorgegebene Objektmodell programmtechnisch vornehmen und die entsprechenden Änderungen in der Datenbank festschreiben.

[6] Es gibt die Möglichkeit, auch im Rahmen von ADO.NET Entity-Modellen mit POCOs zu arbeiten, jedoch würden wir damit deutlich den Rahmen dieses Kapitels sprengen. Für die englischsprachigen Entwickler unter Ihnen hält *http://tinyurl.com/28jo4bo* mehr Informationen bereit.

Datenänderungen mit SaveChanges an die Datenbank übermitteln

Schauen wir uns das Beispiel an, das sich im Modul *DatenBearbeiten* befindet.

HINWEIS Denken Sie auch hier wieder daran, das Startobjekt auf das entsprechende Modul im Beispielprojekt zu ändern, also in diesem Fall auf *DatenBearbeiten*.

Das folgende Beispiel ruft einen Lieferanten ab (und zwar den einzigen, der mit der Zeichenfolge »hybrid« beginnt), liefert ihn aber nicht als Auflistung mit einem Element, sondern durch den Einsatz der Methode `First` (siehe Listing) bereits als Instanz der Klasse `Vendor`.

HINWEIS `Single` wie in LINQ to SQL können Sie bei LINQ to Entities-Abfragen *nicht* verwenden. Sie setzen stattdessen die `First`-Methode ein. Diese sorgt bei der SQL Server-Implementierung des Entity Data Providers dafür, dass die TOP (1)-Klausel vor der eigentlichen SELECT-Abfrage platziert wird.

```
Sub DatenÄndern()

    Dim awContext As New AWEntities()

    Dim lieferant = (From lieferantItem In awContext.Vendor _
                     Where lieferantItem.Name.ToLower.StartsWith("hybrid") _
                     Order By lieferantItem.Name).First

    'Hin und her ändern, damit es nicht nur einmal funktioniert:
    If lieferant.Name = "Hybrid Bicycle Center" Then
        lieferant.Name = "Hybrid Fahrrad Center"
    Else
        lieferant.Name = "Hybrid Bicycle Center"
    End If

    awContext.SaveChanges()

    Console.WriteLine()
    Console.WriteLine("Taste drücken zum Beenden")
    Console.ReadKey()

End Sub
```

Im Anschluss an die Abfrage wird die Eigenschaft `Name` der `Vendor`-Klasse geändert. Und im Prinzip ist das auch schon alles, was Sie machen müssen, wenn Sie Änderungen durchführen möchten: Sie ändern lediglich, wie hier im Beispiel, die korrelierende Eigenschaft, die der Eigenschaft im konzeptionellen Modell entspricht – das Mapping auf die eigentliche Datenbank erledigt Entity Framework.

Um die Änderungen anschließend in die Datenbank zurückzuschreiben, rufen Sie, anders als bei LINQ to SQL, die Methode `SaveChanges` des Objektkontextes auf – die dafür notwendigen UPDATE-Befehle generiert dann der ADO.NET Entity Data Provider selbstständig.

Diese Befehle, die für das Aktualisieren der Datensätze generiert werden, sind bei LINQ to Entities nach derzeitigem Wissensstand nur noch mit dem SQL Profiler-Tool einsehbar (siehe auch Abschnitt »Generierte SQL-Anweisungen unter die Lupe nehmen« auf Seite 821). Die Abfrage des Zieldatensatzes erfolgt im Beispiel mit:

```
SELECT TOP (1)
[Extent1].[VendorID] AS [VendorID],
[Extent1].[AccountNumber] AS [AccountNumber],
[Extent1].[Name] AS [Name],
[Extent1].[CreditRating] AS [CreditRating],
[Extent1].[PreferredVendorStatus] AS [PreferredVendorStatus],
[Extent1].[ActiveFlag] AS [ActiveFlag],
[Extent1].[PurchasingWebServiceURL] AS [PurchasingWebServiceURL],
[Extent1].[ModifiedDate] AS [ModifiedDate]
FROM [Purchasing].[Vendor] AS [Extent1]
WHERE LOWER([Extent1].[Name]) LIKE N'hybrid%'
ORDER BY [Extent1].[Name] ASC
```

Das Aktualisieren übernimmt beim Aufruf von `SaveChanges` dann folgende Anweisung, die dazu von Entity Framework generiert wird:

```
exec sp_executesql N'update [Purchasing].[Vendor]
set [Name] = @0
where ([VendorID] = @1)
',N'@0 nvarchar(50),@1 int',@0=N'Hybrid Bicycle Center',@1=52
```

Einfügen von verknüpften Daten in Datentabellen

Das Einfügen von Daten in Tabellen geschieht durch den Objektkontext bzw. durch die Navigationseigenschaften der Entitätsobjekte. Das heißt im Klartext:

- Sie fügen einen Datensatz in einem Entitätsobjekt ein, indem Sie eine Instanz erstellen und die `AddToEntitätsname`-Methode des Objektkontextes aufrufen, um sie hinzuzufügen. Einen Lieferanten würden Sie also mit `AddToVendor` und einen Kunden mit `AddToCustomer` hinzufügen.
- Sie fügen einen Datensatz in einer Navigationseigenschaft einfach mit der `Add`-Methode ein. Möchten Sie also beispielsweise im Ergebnis nicht nur ein Lieferantenprodukt der Tabelle *ProductVendor* hinzufügen, sondern auch gleichzeitig die Relation zur Lieferanten-Tabelle herstellen, würden Sie eine neue `ProductVendor`-Instanz mit der `Add`-Methode der `ProductVendor`-Navigationseigenschaft einer `Vendor`-Entitätsinstanz hinzufügen.

Auf diese Weise bauen Sie zunächst die Verknüpfungen der neuen Datensätze untereinander auf, und erst ganz zum Schluss sorgen Sie mit der `SaveChanges`-Methode des Objektkontextes dafür, dass die neuen Objekte durch entsprechende SQL-INSERT-Anweisungen in die Datenbank gelangen. Das folgende Beispiel zeigt, wie es geht:

```
    Sub DatenVerknüpftEinfügen()

        Dim awContext As New AWEntities()

        Dim neuerLieferant As New Vendor With {.Name = "ActiveDevelop", .CreditRating = 1, _
                                               .AccountNumber = "ACTDEV00001", .ModifiedDate = Now}

        'Das neue Produkt erstellen
        Dim neuesProdukt As New Product With {.Name = "Visual Basic 2010 - Das Entwicklerbuch", _
                                              .ProductNumber = "", _
```

```
                                                .ListPrice = 59, _
                                                .ModifiedDate = Now, _
                                                .DiscontinuedDate = #12/31/2011#, _
                                                .SellStartDate = #12/10/2008#, _
                                                .SellEndDate = #12/31/2011#, _
                                                .SafetyStockLevel = 5, _
                                                .ReorderPoint = 2}

        'Neues Lieferantenprodukt erstellen, das Produkt mit diesem,
        'und dieses mit dem Lieferanten verknüpfen

        Dim neuesLieferantenProdukt = New ProductVendor With {.Product = neuesProdukt, _
                                                  .StandardPrice = 59, _
                                                  .ModifiedDate = Now, _
                                                  .UnitMeasureCode = "PC", _
                                                  .AverageLeadTime = 19, _
                                                  .MinOrderQty = 1, _
                                                  .MaxOrderQty = 10}

        awContext.AddToVendor(neuerLieferant)
        neuerLieferant.ProductVendor.Add(neuesLieferantenProdukt)
        awContext.SaveChanges()

        Console.WriteLine()
        Console.WriteLine("Taste drücken zum Beenden")
        Console.ReadKey()
    End Sub
```

Sie sehen, dass das Beispiel hier das Anlegen bzw. Hinzufügen der einzelnen neuen Entitätsobjekte bewusst in einer der Relationsvorschrift der Datenbank entgegengesetzten Richtung vornimmt. Ein Einfügen der neuen Datensätze in dieser Reihenfolge auf Datenbankebene würde natürlich sofort zu einer Fremdschlüssel-Relation-Verletzung führen. Doch darum müssen Sie sich in Entity Framework nicht kümmern, denn der Objektkontext kennt natürlich die durch das Storage Modell definierten Relationen der Tabelle und generiert die INSERT-Anweisungen in der Reihenfolge, in der sie diese Beziehungen nicht verletzen können. So produziert – durch den Profiler überprüft – das obige Beispiel folgende INSERT-Anweisungen:

```
exec sp_executesql N'insert [Production].[Product]([Name], [ProductNumber], [MakeFlag],
[FinishedGoodsFlag], [Color], [SafetyStockLevel], [ReorderPoint], [StandardCost], [ListPrice], [Size],
[SizeUnitMeasureCode], [WeightUnitMeasureCode], [Weight], [DaysToManufacture], [ProductLine], [Class],
[Style], [ProductSubcategoryID], [ProductModelID], [SellStartDate], [SellEndDate], [DiscontinuedDate],
[rowguid], [ModifiedDate])
values (@0, @1, @2, @3, null, @4, @5, @6, @7, null, null, null, null, @8, null, null, null, null, null,
@9, @10, @11, @12, @13)
select [ProductID]
from [Production].[Product]
where @@ROWCOUNT > 0 and [ProductID] = scope_identity()',N'@0 nvarchar(50),@1 nvarchar(25),@2 bit,@3
bit,@4 smallint,@5 smallint,@6 decimal(19,4),@7 decimal(19,4),@8 int,@9 datetime2(7),@10
datetime2(7),@11 datetime2(7),@12 uniqueidentifier,@13 datetime2(7)',@0=N'Visual Basic 2010 - Das
Entwicklerbuch',@1=N'',@2=0,@3=0,@4=5,@5=2,@6=0,@7=59.0000,@8=0,@9='2008-12-10 00:00:00',@10='2011-12-31
00:00:00',@11='2011-12-31 00:00:00',@12='00000000-0000-0000-0000-000000000000',@13='2010-07-02
16:24:07.6920277'
```

```
exec sp_executesql N'insert [Purchasing].[Vendor]([AccountNumber], [Name], [CreditRating],
[PreferredVendorStatus], [ActiveFlag], [PurchasingWebServiceURL], [ModifiedDate])
values (@0, @1, @2, @3, @4, null, @5)
select [VendorID]
from [Purchasing].[Vendor]
where @@ROWCOUNT > 0 and [VendorID] = scope_identity()',N'@0 nvarchar(15),@1 nvarchar(50),@2 tinyint,@3
bit,@4 bit,@5 datetime2(7)',@0=N'ACTDEV00001',@1=N'ActiveDevelop',@2=1,@3=0,@4=0,@5='2010-07-02
16:24:07.1139699'

exec sp_executesql N'insert [Purchasing].[ProductVendor]([ProductID], [VendorID], [AverageLeadTime],
[StandardPrice], [LastReceiptCost], [LastReceiptDate], [MinOrderQty], [MaxOrderQty], [OnOrderQty],
[UnitMeasureCode], [ModifiedDate])
values (@0, @1, @2, @3, null, null, @4, @5, null, @6, @7)
',N'@0 int,@1 int,@2 int,@3 decimal(19,4),@4 int,@5 int,@6 nchar(3),@7
datetime2(7)',@0=1003,@1=107,@2=19,@3=59.0000,@4=1,@5=10,@6=N'PC ',@7='2010-07-02 16:24:07.6930278'
```

Wenn Sie sich die Beispielcodedatei einmal ansehen, werden Sie feststellen, dass dort noch etwas mehr passiert, als hier aufgeführt ist. Vor dem Eintragen eines verknüpften Datensatzes werden alle zuletzt eingetragenen Daten gelöscht. Dieses ist notwendig, da zum Beispiel der Lieferant nicht noch mal mit denselben Daten eingetragen werden darf. Zudem machen wir hier etwas, was in einer Produktivumgebung NIEMALS ohne Rücksprache durchgeführt werden darf: Wir deaktivieren einen Trigger.

Daten aus Tabellen löschen

Auch das Löschen von Entitätsobjekten aus bestehenden Auflistungen könnte nicht einfacher sein – wie beim Einfügen von Daten spielt die Reihenfolge, mit der Sie Entitätsobjekte von ihren übergeordneten Entitäten trennen, quasi keine Rolle, denn Entity Framework kümmert sich um die Generierung der richtigen Reihenfolge der DELETE-Statements, damit es keine Fremdschlüsselverletzungen gibt.

In der Regel werden Sie durch irgendeine Abfrage ein Objekt ermitteln, das es aus einer Datenbanktabelle zu löschen gilt. Und dabei wird es oft vorkommen, dass entsprechende untergeordnete Zeilen in anderen Tabellen direkt mitgelöscht werden müssen.

Das Angenehme bei Entity Framework ist, dass es dabei kaskadierendes Löschen unterstützt. Wenn Sie also einen Datensatz einer Haupttabelle löschen, werden in der Standardeinstellung die durch Fremdschlüssel verknüpften Datensätze in untergeordneten Tabellen direkt mitgelöscht.

Im folgenden Beispiel ermitteln wir zunächst ein Lieferantenprodukt anhand seines Namens. Auch hier verwenden wir, wie schon im Abschnitt »Anonymisierungsvermeidung bei Abfragen in verknüpften Tabellen« auf Seite 827, eine verschachtelte `From`-LINQ-Abfrage, um ein Entitätsobjekt zu ermitteln, dessen Abfragekriterium allerdings in einer verknüpften Tabelle liegt (wir selektieren nach dem Namen des Produkts, wollen aber einen Datensatz aus der Lieferantenprodukte-Tabelle ermitteln. In den Lieferantenprodukten gibt es aber kein Namens-Feld, nur in der verknüpften Produkte-Tabelle, die die Details wiedergibt). Wir bekommen also auf diese Weise ein Lieferantenprodukt zurück. Ziel ist es aber nun, nicht nur das Lieferantenprodukt zu löschen, sondern auch die korrelierenden Detailinformationen aus der Tabelle *Product* sowie den Lieferanten aus der Tabelle *Vendor* selbst.

Wir bedienen uns zur Ermittlung der gesuchten Entitätsobjekte wieder des »manuellen Lazy Loading«, speichern dann allerdings die Referenzen auf `Product`- und `Vendor`-Objekt zunächst zwischen. Der Hintergrund: Sie entfernen Objekte mit `DeleteObject` aus den entsprechenden Navigationseigenschaftslisten bzw.

aus dem Objektkontext und lösen damit später die DELETE-Anweisung für SQL Server aus. Allerdings gehen mit `DeleteObject` auch die .NET-Objektreferenzen[7] der Navigationseigenschaftsobjekte verloren. Aus diesem Grund ist es sinnvoll, die .NET-Referenzen mit Hilfsvariablen zu sichern und diese dann `DeleteObject` durch die temporären Objektvariablen zu übergeben. Im unten stehenden Beispiel würden Sie anderenfalls eine Ausnahme auslösen.

```
Sub DatenLöschen()

    Dim awContext As New AWEntities()

    'Die Entitätsobjekte ermitteln, die sich durch das angegebene Produkt ergeben:
    Dim lieferantProduct = (From lieferantProduktItem In awContext.ProductVendor _
                            From productItem In awContext.Product _
                             Where productItem.ProductID = lieferantProduktItem.ProductID _
                             Where productItem.Name = "Visual Basic 2010 - Das Entwicklerbuch" _
                              Select lieferantProduktItem).First

    'Nur zur Kontrolle
    Console.WriteLine(lieferantProduct.ProductID)

    'Lazy-Loading: Die übergeordneten Entitätsobjekte laden.
    'Zuerst das Lieferantenprodukt, dass dem ermittelten Produkt
    'zugeordnet ist.
    lieferantProduct.ProductReference.Load()

    'Und genau so einfach laden wir auch den übergeordneten Lieferanten.
    lieferantProduct.VendorReference.Load()

    'Die Referenzen auf die Entitäten müssen wir uns
    'temporär merken, weil DeleteObject sie mit
    'der übergeordneten Tabelle zerstört.
    Dim tmpProduct = lieferantProduct.Product
    Dim tmpVendor = lieferantProduct.Vendor

    'Reihenfolge des Löschens spielt jetzt ...
    awContext.DeleteObject(tmpVendor)

    '... keine Rolle mehr.
    awContext.DeleteObject(tmpProduct)

    'Entity Framework legt die richtige Reihenfolge fest!
    awContext.SaveChanges()

    Console.WriteLine()
    Console.WriteLine("Taste drücken zum Beenden")
    Console.ReadKey()
```

[7] Ich spreche hier bewusst von .NET-Referenzen, um Verwirrungen zwischen logischen Referenzen zwischen Datentabellen und Referenzen auf Objekte aus CLR-Sicht zu vermeiden.

Das kaskadierende Löschen und die Einhaltung der richtigen Reihenfolge kann man an dieser Stelle übrigens sehr gut beobachten: Obwohl wir mit `DeleteObject` nur zwei Entitäten löschen, zieht dies laut SQL Profiler drei DELETE-Anweisungen nach sich. Das Lieferantenprodukt in der Tabelle *ProductVendor* steht nämlich in Abhängigkeit zu seinem Lieferanten in der Tabelle *Vendor*. So muss also zunächst das Lieferantenprodukt gelöscht werden, bevor der Lieferant aus der *Vendor*-Tabelle entfernt werden kann – Entity Framework kümmert sich darum automatisch.

```
exec sp_executesql N'delete [Purchasing].[ProductVendor]
where (([ProductID] = @0) and ([VendorID] = @1))',N'@0 int,@1 int',@0=1005,@1=109

exec sp_executesql N'delete [Production].[Product]
where ([ProductID] = @0)',N'@0 int',@0=1005

exec sp_executesql N'delete [Purchasing].[Vendor]
where ([VendorID] = @0)',N'@0 int',@0=109
```

HINWEIS Damit das Löschen in diesem Beispiel funktionieren kann, müssen Sie den Trigger, der an der *Vendor*-Tabelle hängt, deaktivieren. Dies geschieht bereits in der Methode *DatenVerknüpftEinfügen*. Diese sollte vorher einmal gelaufen sein, schon allein aus dem Grund, da diese Methode erst die zu löschenden Daten generiert. Sie können den Trigger natürlich auch mithilfe von SQL Server Management Studio deaktivieren oder löschen.

Concurrency-Checks (Schreibkonfliktprüfungen)

Solange Sie Daten nur von einem einzigen Client aus im SQL Server ändern, wird Ihre Anwendung beim Aktualisieren von Daten kaum Probleme bekommen. Was geschieht aber, wenn dieselben Daten in einer Netzwerkumgebung von zwei Benutzern Ihrer Software gleichzeitig geändert wurden?

Standardmäßig implementiert Entity Framework das so genannte optimistische Konfliktprüfungsmodell. Die darunter liegenden Objektdienste speichern dabei Änderungen in den Objekten, ohne eine Konfliktprüfung durchzuführen.

Um dieses Standardverhalten für bestimmte Eigenschaften zu ändern, ist es angebracht, ihre Einstellung mithilfe des Designers im konzeptionellen Modell entsprechend anzupassen, und das ist eine ganz einfache Geschichte. Sie selektieren die Eigenschaft in der Entität und stellen ihren *Parallelitätsmodus* einfach auf *Fixed* (Abbildung 28.25).

Abbildung 28.25 Um die Konfliktprüfung für bestimmte Eigenschaften einer Entität zu aktivieren, stellen Sie ihren *Parallelitätsmodus* einfach auf *Fixed*

Wenn diese Einstellung zur Anwendung kommt, führen die Objektdienste eine Änderungsüberprüfung durch, bevor die Änderungen in die Datenbank übertragen werden. Gibt es dabei Konflikte, wird eine `OptimisticConcurrencyException` ausgelöst.

Wenn Sie in Szenarien, die zu häufigen Änderungskonflikten neigen, Änderungen an die Datenbank übermitteln, ist es empfehlenswert, häufiger zwischendurch die `Refresh`-Methode des Objektkontextes zu verwenden. `Refresh` definiert, wie Änderungen sich auswirken sollen. Übergeben Sie die Option `StoreWins` (in etwa: *die Speichereinheit – also die Datenbank – gewinnt!*), überschreiben die Objektdienste die Daten im Entitätsobjekt. Die Alternative: Sie übergeben die Option `ClientWins`, damit die Daten, die in den Entitätsobjekten lagern, ihren Siegeszug in die Datenbank antreten können. Ein Beispiel dafür kann folgendermaßen ausschauen:

```
Sub DatenÄndernMitKollisionsprüfung()

        Dim awContext As New AWEntities()

        Dim lieferant = (From lieferantItem In awContext.Vendor _
                         Where lieferantItem.Name.ToLower.StartsWith("hybrid") _
                         Order By lieferantItem.Name).First

        'Hin und her ändern, damit es nicht nur einmal funktioniert:
        If lieferant.Name = "Hybrid Bicycle Center" Then
            lieferant.Name = "Hybrid Fahrrad Center"
        Else
            lieferant.Name = "Hybrid Bicycle Center"
        End If
        'Zeitstempel für die letzte Änderung setzen
        lieferant.ModifiedDate = Date.Now
```

```
        Console.WriteLine("Wenn Sie wollen, können Sie nun den Datensatz mit der VendorID {0} mit dem
SQL Server Management Studio ändern.", lieferant.VendorID)
        Console.WriteLine("Vergessen Sie nicht, mindestens ein Feld zu ändern, welches bei dem
Parallelitätsmodus fixed eingestellt hat (typischerweise ModifiedDate).")
        Console.WriteLine("Oder starten Sie dieses Programm einfach nochmals (z.B. mit dem Windows
Explorer aus dem Bin-Verzeichnis)")
        Console.WriteLine("Drücken Sie bitte im Anschluss eine Taste")

        Console.ReadKey()
        Console.WriteLine("")
        Console.WriteLine("")

        Try
            Dim anzahl = awContext.SaveChanges()
            Console.WriteLine("Keine Konflikte bei der Speicherung. " _
                              & anzahl & " Aktualisierungen gespeichert.")
        Catch ex As OptimisticConcurrencyException
            awContext.Refresh(RefreshMode.ClientWins, lieferant)

            awContext.SaveChanges()
            Console.WriteLine("Konfliktfehler wurde abgefangen, Änderungen wurden gespeichert!")
        End Try

        'Änderungen zurückschreiben
        awContext.SaveChanges()

        Console.WriteLine()
        Console.WriteLine("Taste drücken zum Beenden")
        Console.ReadKey()

End Sub
```

Wichtig für das Funktionieren dieses Beispiels ist es, dass Sie den *Parallelitätsmodus* des Felds `ModifiedDate` des Entitätsobjekts `Vendor` zuvor auf `Fixed` gestellt haben, wie in Abbildung 28.25 gezeigt.

Starten Sie das Programm einfach zwei Mal, zum Beispiel einmal im Debug-Modus aus Visual Studio heraus und einmal mithilfe des Windows-Explorers aus dem Bin-Verzeichnis Ihres Projekts. Betätigen Sie bei einem der beiden Programme sofort nach der Meldung *Drücken Sie bitte im Anschluss eine Taste* eine Taste und warten Sie, bis sich das Programm beendet hat oder die Meldung *Taste drücken zum Beenden* erscheint. Nun drücken Sie in dem anderen Programmfenster eine Taste. Sie werden dann auf eine Kollision hingewiesen.

Datenmodell aus Datenbank aktualisieren

Das Aktualisieren des ADO.NET Entity Data Models funktioniert mit Visual Studio 2010 sehr gut – was man von der Vorgängerversion nicht unbedingt sagen konnte. Microsoft hat hier deutlich nachgebessert und damit den Entwicklerkomfort in der .NET Framework 4.0-Version erheblich gesteigert.

Sie können eine Aktualisierung anstoßen, indem Sie das Kontextmenü des Designers aktivieren und dort den Eintrag *Modell aus der Datenbank aktualisieren* auswählen. Ihnen wird im Folgenden ein Dialog angezeigt, der drei Registerkarten besitzt: *Hinzufügen*, *Aktualisieren* und *Löschen* (Abbildung 28.26).

Abbildung 28.26 Beim Aktualisieren des Datenmodells können Sie neue Tabellen, Sichten oder gespeicherte Prozeduren hinzufügen. Zudem werden Sie darüber informiert, welche Objekte aktualisiert werden und welche gelöscht werden, da sie in der Datenbank nicht mehr vorhanden sind

- Die Registerkarte Reiter *Hinzufügen* dient dazu, neue Tabellen, Sichten oder gespeicherte Prozeduren in das Modell aufzunehmen.
- Die Registerkarte *Aktualisieren* zeigt an, welche Tabellen, Sichten oder gespeicherte Prozeduren in dem Modell aktualisiert werden.
- Die Registerkarte *Löschen* gibt an, welche Objekte aus dem Modell entfernt werden, da sie sich nicht mehr in der Datenbank befinden.

Sie könnten übrigens Gefahr laufen, zu versuchen, über die Registerkarte *Löschen* Elemente aus dem Entitätsmodell zu entfernen. Für diesen Zweck ist diese Registerkarte jedoch nicht gedacht. Sie löschen stattdessen eine Entität aus dem Modell, indem Sie im Designer die Entität auswählen und `Entf` drücken. Alternativ rufen Sie das Kontextmenü der Entität auf und wählen dort den Eintrag *Löschen* aus.

Model First-Design

Mit Visual Studio 2008 konnten Sie bisher nur anhand einer Datenbank ein Modell erstellen. Der umgekehrte Weg, erst das Modell zu erstellen und daraus dann die Datenbank, war Ihnen verwehrt. Entity Framework 4.0 bietet Ihnen auch den umgekehrten Weg an: In den folgenden Schritten legen wir nun erst ein neues Modell mit Projekten und Mitarbeitern an und erstellen im Anschluss daraus die Datenbankstrukturen.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis

```
...\VB 2010 Entwicklerbuch\Kapitel 28\ModelFirst
```

1. Legen Sie ein neues Projekt vom Typ *Konsolenanwendung* mit dem Namen **ModelFirst** an.
2. Fügen Sie dem Projekt ein neues Element vom Typ *ADO.NET Einity Data Model* hinzu, und nennen Sie es **modelFirst.edmx**.
3. Wählen Sie im nun folgenden Assistenten *Leeres Modell* aus und bestätigen Sie die Auswahl mit *Fertig stellen*.
4. Sie sehen nun den leeren Entity Data Model Designer. Wählen Sie anschließend aus der Toolbox das Werkzeug *Entität* aus und ziehen Sie es in den Designer.
5. Ändern Sie im Eigenschaftenfenster den Namen der Entität auf **Mitarbeiter**.
6. Ändern Sie ebenfalls den Namen der Entitätenmenge von *MitarbeiterMenge* auf **Mitarbeiter**. Dies wird später der Name der Tabelle in der Datenbank.
7. Ändern Sie das Feld *ID* in **MitarbeiterID**.

TIPP Wenn Sie eine neue Entität über das Kontextmenü des Designers anlegen, können Sie den Namen der Entität, den Namen der Entitätsmenge und den Namen des Primärschlüssels sowie dessen Datentyp direkt angeben.

8. Fügen Sie weitere Felder zu der Mitarbeiterentität hinzu. Hierzu rufen Sie das Kontextmenü auf der Entität *Mitarbeiter* auf und wählen dort *Hinzufügen/Skalareigenschaft* (Abbildung 28.27).

Abbildung 28.27 Hinzufügen von Feldern in den Entity Data Model Designer

9. Achten Sie im Eigenschaftenfenster auf den Datentyp der neuen Entität. Für *Nachname*, *Vorname* wählen Sie *String*, für *Geburtstag* und *EingestelltAm* verwenden Sie *DateTime*.
10. Legen Sie als Nächstes die neue Entität **Projekte** an.
11. Ändern Sie analog dazu auch hier das Feld *ID* in **ProjektID**.
12. Fügen Sie weitere Felder hinzu.

Kommen wir nun zu den Femdschlüsseln bzw. den Navigationseigenschaften. Sie lassen sich sehr einfach über das Zuordnungsobjekt aus der Toolbox erstellen.

13. Wählen Sie das Werkzeug *Zuordnung* in der Toolbox aus.

14. Klicken Sie als Nächstes auf die Projektentität.Eventuell ist Ihnen der geänderte Mauszeiger aufgefallen, als Sie den Bereich der Projektentität betreten haben.

15. Den zweiten Klick geben Sie nun auf der Mitarbeiterentität ab. Sie haben soeben bei beiden Entitäten jeweils eine neue Navigationseigenschaft erstellt.

16. Diese Navigationseigenschaft soll nun den Projektleiter eines Projekts darstellen. Jedes Projekt hat genau einen Projektleiter, ein Projektleiter kann 0-*n* Projekte leiten. Da Sie als Erstes die Projektentität bei der Zuordnung angeklickt haben, ist das nun Ihr »End1«-Punkt, die Mitarbeiter sind der »End2«-Punkt. Somit ergibt sich für die End1-Multiplizität »1 (Auflistung von Projekten)«. Das ist die Relation *ein Mitarbeiter kann mehrere Projekte leiten*. Sobald Sie die »End1«-Navigationseigenschaft auf **Leiter** geändert haben, ändert sich auch in der Projektentität die Navigationseigenschaft auf *Leiter*. Die Navigationseigenschaft des Mitarbeiters ändern wir noch schnell von *Projekte* nach **Leitet**.

Wenn Sie später nun mit dem Entitätsmodell arbeiten, können Sie von einem Projekt durch die Navigationseigenschaft *Leiter* auf den Projektleiter zugreifen. Wenn Sie später einen Datensatz aus der Mitarbeiter Entität bearbeiten, können Sie anhand der Navigationseigenschaft *Leitet* erkennen, ob es sich um ein Projektleiter handelt und dann über die Navigationseigenschaft *Leitet* auf sein Projekt zugreifen.

17. Als nächstes soll die Zuordnung von (nicht leitenden) Mitarbeitern und Projekten abgebildet werden. Ein Mitarbeiter kann an mehreren Projekten arbeiten und an einem Projekt arbeiten mehrere Mitarbeiter. In diesem Fall ist es günstiger, die neue Zuordnung bei dem Mitarbeiter anzufangen.

18. Wählen Sie das Werkzeug *Zuordnung* aus und klicken Sie als Erstes auf die Mitarbeiter Entität und dann auf die Projekte Entität. Es wurde eine Zuordnung angelegt, die als »End1« Multiplizität »1 (Eine von Mitarbeiter)« und als »End2« »* (Auflistung von Projekt)« definiert. Diese Zuordnung lässt sich wie folgt lesen: *Ein Mitarbeiter kann an mehreren Projekten arbeiten und (leider auch) ein Projekt besteht aus (genau) einem Mitarbeiter*. Die Zuordnung ist also noch nicht vollständig.

19. Um die Zuordnung zu komplettieren, wählen Sie die Zuordnung aus und ändern Sie die »End1-Multiplizität« von »1 (Eine von Mitarbeiter)« in »* (Auflistung von Mitarbeiter)«. Durch diese Änderung können mehrere Mitarbeiter an einem Projekt arbeiten. Zur besseren Lesbarkeit ändern Sie noch die soeben angelegten Navigationseigenschaften: Auf Seiten des Projekts heißt die Navigationseigenschaft *Mitarbeiter* und auf Seiten des Mitarbeiters *MitarbeitAn*.

TIPP Die Konfiguration der Zuordnung mit ihren End1- und End2-Eigenschaften ist nicht sehr übersichtlich. Sie können genauso gut die jeweilige Navigationseigenschaft auswählen und dort die Multiplizität (auch Kardinalität genannt) einstellen.

HINWEIS Während der Erstellung des konzeptionellen Modells kann es Fehlermeldungen und Warnungen in der Fehlerliste geben, die sich auf nicht abgebildete (ungemappte) Entitäten beziehen. Sie können diese Fehlermeldungen ignorieren, da diese Meldungen spätestens beim Generieren der Datenbankerstellungsanweisungen verschwinden.

Nun erstellen wir aus unserem konzeptionellen Modell (ADO.Net Entity Data Model) ein Datenbankmodell mit den notwendigen DDL-Anweisungen.

20. Klicken Sie hierzu in einen freien Bereich des konzeptionellen Modells und wählen dort den Menüeintrag *Datenbank aus Modell generieren* aus.

HINWEIS Um die Datenbankerstellungsanweisungen erzeugen zu können, müssen Sie Administrationsrechte in der Datenbankinstanz besitzen. Wenn Sie über administrative Rechte verfügen, haben Sie standardmäßig die Administrationsrechte in der Datenbankinstanz. Unter Windows Vista und Windows 7 müssen Sie Visual Studio dann unter Umständen als Administrator starten. Alternativ können Sie sich natürlich auch mit dem *sa*-Benutzer an der Datenbank anmelden, falls Sie die Datenbank für die Authentifizierung im *gemischten Modus* konfiguriert haben.

Abbildung 28.28 Verbindungseinstellungen für das konzeptionelle Datenmodell

21. Der Generator erstellt daraufhin ein Skript zur Erstellung einer neuen Datenbank. Die Tabellen werden jedoch nicht wirklich in der Datenbankinstanz erstellt – es wird lediglich eine Datei mit den Anweisungen gespeichert. Wählen Sie die richtige Datenbankverbindung aus, oder legen Sie über die Schaltfläche *Neue Verbindung* eine neue Verbindung an (Abbildung 28.28).

Abbildung 28.29 Eingabe des Namens für die neu zu erstellende Datenbank

22. Unter *Servername* wählen Sie die Datenbankinstanz (Abbildung 28.29). Falls Sie eine neue Datenbank anlegen wollen, geben Sie bei *Wählen Sie einen Datenbanknamen, oder geben Sie ihn ein* den neuen Datenbanknamen ein. Ansonsten wählen Sie die betreffende Datenbank aus.

Abbildung 28.30 Konfiguration des Dateinamens für das Datenbankerstellungsskript

23. Im letzten Dialog des Assistenten (Abbildung 28.30) wird der Code angezeigt, mit dem die Datenbankstrukturen generiert werden können (die Datenbank selbst haben Sie zu diesem Zeitpunkt bereits über den Verbindungsmanager angelegt – sie ist nur leer). Geben Sie bei *DDL speichern unter* den Dateinamen an, unter dem das Erstellungsskript gespeichert werden soll.

24. Wechseln Sie nun in die neu erstellte Datei. In der Menüleiste können Sie sich nun mit der Datenbank verbinden (Symbol ganz links), die Zieldatenbank auswählen und dann mit dem »Play«-Symbol das Skript ausführen.

HINWEIS Sollten Sie nun noch Änderungen am Tabellennamen (Name der Entitätsmenge) vornehmen, werden auch Löschanweisungen zum Löschen der alten Tabellen miterstellt. Ihre eventuell vorhandenen Testdaten müssen Sie jedoch vorher selbst sichern und gegebenenfalls in das neue Modell überführen.

Wenn Sie sich die erstellten Tabellen ansehen, fällt Ihnen vielleicht die Tabelle *MitarbeiterProjekte* auf. Sie ist aus der *m*-zu-*n*-Verbindung zwischen Mitarbeiter und Projekte entstanden und wurde automatisch und transparent erstellt. Dieses ist allerdings auch an den Feldern der Tabelle zu erkennen. Sie werden automatisch benannt und zwar als Kombination der beiden Namen der Navigationseigenschaften bzw. IDs (zum Beispiel: *MitarbeitAn_ProjektID*).

So sollte Ihr Modell nun aussehen:

Abbildung 28.31 Das finale Datenmodell für das *model first* Beispiel

Vererbung im konzeptionellen Datenmodell

Vererbung ist ein sehr arbeitssparendes Feature beim konzeptionellen Datenmodell. Mithilfe von Vererbung lassen sich die Entitäten auf ähnliche Weise vererben, wie Sie es bereits bei der objektorientierten Programmierung kennen gelernt haben.

Sie können beispielsweise einen Mitarbeiter definieren und von ihm Entwickler, Projektleiter usw. ableiten und die abgeleiteten Klassen um spezifische Felder erweitern (zum Beispiel die vom Entwickler beherrschten Programmiersprache(n)).

Das Schöne an dieser Vorgehensweise ist, dass sie sehr transparent ist, das heißt beim Umgang mit Entitäten arbeiten Sie mit einer *Entwickler*-Instanz, ohne dass Sie wissen müssen, dass diese von einem *Mitarbeiter* abgeleitet ist. Lediglich beim Hinzufügen eines *Entwicklers* zum Objektkontext müssen Sie die Vererbungshierarchie kennen, denn Sie fügen den Entwickler mit `AddMitarbeiter` dem Kontext hinzu.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis

```
...\VB 2010 Entwicklerbuch\Kapitel 28\Vererbung
```

Wenn Sie sich die Eigenschaften der *.edmx*-Datei ansehen (Abbildung 28.31), haben Sie dort zwei Möglichkeiten, mit denen Sie die Generierung der Datenbankskripts steuern können.

Abbildung 28.32 Einstellungen für die Codegenerierung des Datenmodells

Die DDL-Generierungsvorlage SSDLToSQL10.tt ist eine T4-Generierungvorlage für die Erstellung von Tabellen (Model First) anhand des konzeptionellen Models.

HINWEIS T4-Vorlagen können vom Benutzer erstellt werden und sind eine Mischung aus statischem Text und Programmcode, die als Ergebnis eine Codedatei erstellen. Falls Sie bereits Erfahrungen mit PHP oder Java Server Pages haben, werden Sie sich in diesen Vorlagen einfach zurecht finden. Diese Vorlagen lassen sich sehr einfach anfertigen oder verändern. Visual Studio bringt bereits eine Vorlage hierfür mit. Sie können sie über das Kontextmenü des Modells per *Neues Codegenerierungselement hinzufügen* in das Projekt integrieren.

Die zweite Einstellung nennt sich *Workflow zur Datenbankgenerierung*. Es gibt derzeit nur die *TablePer TypeStrategy*. Mit dieser Einstellung wird bei der Verwendung von Tabellenvererbung pro Typ je eine Tabelle erstellt. In Verbindung mit den Basisklassen ergibt sich so ein vollständiges Abbild einer Entität.

Abbildung 28.33 Hinzufügen der vom Mitarbeiter abgeleiteten Entität *Entwickler* (per Kontextmenü des Designers)

Mit dem Kontextmenü des Designers können Sie eine neue Entität zum konzeptionellen Modell hinzufügen und dabei die Basisentität auswählen (Abbildung 28.33). Somit erhalten Sie die Entität *Mitarbeiter* und *Entwickler*. Durch die Strategie *TablePerTypeStrategy* resultieren daraus zwei Tabellen:

- Mitarbeiter(menge)
- Entwickler(menge)

Die Tabelle *Mitarbeiter* besteht aus den Feldern *MitarbeiterID*, *Nachname* und *Vorname*. Die Tabelle *Entwickler* hat zum jetzigen Zeitpunkt keine weiteren Felder und besteht somit nur aus dem Primärschlüssel der Tabelle *Mitarbeiter* – also *MitarbeiterID*.

Der vollständige Datensatz eines Entwicklers bei einer vererbten Tabelle ergibt sich somit aus einer *gejointen* Abfrage der Mitarbeitertabelle und der Entwicklertabelle. Um die Formulierung dieses Joins brauchen Sie sich jedoch nicht zu kümmern, denn das erledigt Entity Framework bzw. der generierte Code für Sie.

Es ist darüber hinaus auch möglich, die *Entwickler*-Entität mit in die Mitarbeitertabelle auszulagern, sodass *Mitarbeiter* und *Entwickler* aus einer Tabelle befüllt werden. In diesem Fall können Sie jedoch nicht die Datenbank aus dem Modell erstellen lassen (Model first), es sei denn, Sie verwenden eine andere DDL-Generierungsvorlage bzw. eine andere Strategie als die *TablePerTypeStrategy*.

Sie haben jedoch die Möglichkeit, die Datenbankstrukturen von Hand anzulegen und dann in das konzeptionelle Modell zu importieren (*Modell aus der Datenbank aktualisieren* des Modell-Kontextmenüs).

Im Anschluss daran müssen Sie die Vererbungshierarchie im konzeptionellen Modell erstellen und die Daten auf die verschiedenen Entitäten verteilen. Hierzu benötigen Sie mindestens ein Feld, anhand dessen Sie die verschiedenen Entitätstypen unterscheiden können. Im *Mitarbeiter-Entwickler*-Beispiel könnte es

zum Beispiel das Feld *Programmiersprache* (Typ: `String`) sein. Über die *Mappingdetails* ist dann die Verteilung der Daten auf die Entitätstypen möglich. Ist das Feld `Null`, handelt es sich um einen Mitarbeiter, ist es ungleich Null, handelt es sich um einen Entwickler. Ebenso wäre ein *MitarbeiterTyp*-Feld denkbar, wie Sie in Abbildung 28.34 sehen können.

Abbildung 28.34 In den *Mappingdetails* für einen Entwickler: *MitarbeiterTyp 3* ist ein Entwickler, *MitarbeiterTyp 1* ein Projektleiter und der Assistent hat den *MitarbeiterTyp 2*. Wie Sie sehen, fehlen noch die weiteren Felder für die Projektleiter, Entwickler und Assistenten (zum Beispiel Programmiersprachen beim Entwickler usw.)

Im Beispiel werden *Mitarbeiter*, *Entwickler*, *Assistenten* und *Projektleiter* alle auf die Tabelle *Mitarbeiter* abgebildet.

Doch nun zurück zur *TablePerTypeStrategy*. Im Beispiel Model First wurde neben der Entität *Projekt* auch noch die Entität *Mitarbeiter* angelegt. Das Beispiel wurde nun dahingehend erweitert, dass auch die Entität *Entwickler* vorhanden ist und von der Entität *Mitarbeiter* erbt. Der Entwickler verfügt zu den normalen Mitarbeiterdaten auch noch über ein Feld für die *Programmiersprache*.

In dem Module *Module1.vb* können Sie sehen, wie ein (normaler) Mitarbeiter und auch ein Entwickler angelegt werden. Das Verteilen der Daten auf die unterschiedlichen Tabellen wird völlig transparent von dem generierten Code durchgeführt.

Eine vollständige Beschreibung dieser Möglichkeiten würde den Rahmen dieses Kapitels sprengen. Vielleicht sogar die des Buchs.

HINWEIS Bei dieser Art der Vererbung (mehrere Entitäten in einer Tabelle) ist sehr viel Handarbeit notwendig. Hier sind im Designer, Codegenerator usw. leider auch noch einige Fehler enthalten. Ich hoffe, dass hier bald ein Update bzw. Patch verfügbar ist, der diese großartige Funktionalität etwas besser handhabbar macht.

T-SQL-Befehle im Objektkontext direkt ausführen

Wie Sie vielleicht schon im Beispielprojekt *LinqToEntitiesDemo* gesehen haben, kann man mit dem neuen ADO.NET Entity Framework T-SQL-Befehle auch direkt verwenden und zur Ausführung an den SQL Server schicken. In der Methode des Beispiels *LinqToEntitiesDemo* konnten Sie sehen, dass wir den Trigger auf der *Vendor*-Tabelle mit folgender T-SQL-Anweisung deaktiviert haben:

```
awContext.ExecuteStoreCommand("disable trigger Purchasing.dVendor on Purchasing.Vendor")
```

WICHTIG Durch die direkte Verwendung von T-SQL-Befehlen geht ein großer Vorzug von Entity Framework verloren: die Trennung von konzeptionellem Datenmodell und der darunter liegenden Datenbank. Die direkte Verwendung von T-SQL-Anweisungen kann dabei die Kompatibilität zu anderen Datenbanken gefährden.

Mit Entity Framework 4.0 sind folgende Methoden des Objektkontextes neu hinzugekommen:

- **ExecuteFunction:** führt eine gespeicherte Prozedur aus.
- **ExecuteStoreCommand:** führt einen beliebigen Befehl aus.
- **ExecuteStoreQuery:** führt eine SELECT-Anweisung aus.

Die Methoden sind überladen und stellen in mindestens einer Variante eine typsichere, generische Funktion bereit, die als Rückgabewert ein `ObjectResult(of T)` zurückgeben (Ausnahme: `ExcuteStoreCommand`). Die Konvertierung der Rückgabewerte in den Typ `T` wird durch die `Translate`-Methode des Objektkontextes durchgeführt. Falls Parameter übergeben werden, sind diese als datenbankspezifische Parameter anzugeben, wie Sie im folgenden Beispiel für die AdventureWorks-Datenbank sehen können:

```
Dim awContext As New AWEntities()

Dim ret = awContext.ExecuteStoreQuery(Of Vendor) _
                    ("SELECT * FROM Purchasing.Vendor where AccountNumber=@accountNr",
    New System.Data.SqlClient.SqlParameter("accountNr", "METROSP0001")
    )
For Each ds As Vendor In ret
    Console.WriteLine("Ret:" & ds.Name)
Next
```

Innerhalb der SELECT-Anweisung wird der Parameter mit einem vorangestellten @-Zeichen benannt. Für die `SqlParameter`-Instanz wird der Parametername ohne @-Zeichen angegeben. Der Wert des Parameters wird als Zweites spezifiziert (hier: *METROSP0001*).

Die Parametrisierung von Anweisungen bietet für das Datenbankmanagementsystem deutliche Geschwindigkeitsvorteile. Wenn eine neue Anweisung vom Datenbankmanagementsystem ausgeführt wird, wird die Anweisung als Erstes geparst, im Anschluss daran ein Ausführungsplan erstellt und für die Anweisung gespeichert. Wird nun **exakt** dieselbe Anweisung erneut an die Datenbank gesendet, kann auf das Parsen der Anweisung und die Erstellung eines Ausführungsplans verzichtet werden. Der Vorteil bei der Benutzung von Parametern ist also nicht ganz unerheblich.

Arbeiten mit gespeicherten Prozeduren

Gespeicherte Prozeduren oder Funktionen können ebenfalls in das konzeptionelle Modell eingebettet werden. Hierzu wählen Sie die gespeicherten Prozeduren im Assistenten *Modell aus der Datenbank aktualisieren* aus.

Die gespeicherten Prozeduren und Methoden sind im Anschluss im Speichermodell des Modellbrowsers sichtbar. Dort können Sie sie über das Kontextmenü in das konzeptionelle Datenmodell übernehmen (Menüeintrag *Funktionsimport hinzufügen* – dieser ist jedoch nur bei gespeicherten Prozeduren vorhanden). Nach dem erfolgreichen Import sind die Methoden im Modellbrowser im *EntityContainer* unter *Funktionsimporte* zu finden (Abbildung 28.35).

Abbildung 28.35 Im Modellbrowser können Entitäten des konzeptionellen Modells und die Tabellen des Speichermodells betrachtet werden sowie Funktionsimporte und komplexe Typen erstellt werden und vieles mehr

HINWEIS Funktionsimporte können Sie nur für gespeicherte Prozeduren erstellen. Diese Terminologie ist leider an dieser Stelle mehrdeutig. Der Modellbrowser zeigt unterhalb des Knotens *Gespeicherte Prozeduren* nicht nur eben solche (die mit dem SQL-Befehl `CREATE PROCEDURE` erstellt werden), sondern auch Funktionen an (SQL-Befehl `CREATE FUNCTION`). Eine Übersicht erhalten Sie am einfachsten über den Server-Explorer (Abbildung 28.36).

Abbildung 28.36 Mit dem Server-Explorer können Sie die echten gespeicherten Prozeduren von den gespeicherten Funktionen unterscheiden

Ein weiterer wichtiger Aspekt ist die Nutzung von gespeicherten Prozeduren für *Insert*-, *Update*- und *Delete*-Anweisungen an den Datenbankserver für die jeweiligen Tabellen. Hierzu stellt das *Mappingdetails*-Fenster die notwendigen Einstellungsmöglichkeiten bereit, etwa wie in Abbildung 28.37 zu sehen.

Abbildung 28.37 In den Mappingdetails kann für eine Entität auch eingestellt werden, dass für *Insert*-, *Update*- oder *Delete*-Anweisungen gespeicherte Prozeduren verwendet werden

HINWEIS Falls Sie automatisch generierte (Primär-)Schlüssel verwenden, müssen Sie diese als Rückgabewert konfigurieren.

Ausblick

Mit der 2. Version (4.0) von Entity Framework hat Microsoft viele Bitten der Entwicklergemeinde erhört und entsprechend umgesetzt. Im Gegensatz zu LINQ to SQL werden wir aber in den kommenden Jahren sicherlich noch weitere, tolle Features sehen, die es immer leichter werden lassen, professionelle Datenbankanwendungen zu erstellen. Insbesondere beim Umgang mit sehr umfangreichen Datenbanken gibt es noch Wunschfunktionen – beispielsweise die Möglichkeit, ein großes Entitätsmodell auf mehrere Dateien oder Ansichten zu verteilen, um es so im Team, insbesondere beim Ein- und Auschecken mit Team Foundation Server (oder anderen Systemen mit Quellcode-Versionsverwaltung und weiteren kollaborativen Features), besser handhabbar zu machen.

TIPP Kleiner Tipp zum Schluss in diesem Zusammenhang: Große Datenbanken sollten Sie durchaus auf mehrere Entitätsmodelle und mehrere Assemblys verteilen und vorschlagsweise entsprechende Factory-Funktionen für die Entitätsverbindungszeichenfolgen entwickeln.

Im Rahmen dieses Buches können wir natürlich nicht auf alle Features von Entity Framework eingehen. In Entity Framework gehört die Verwendung von gespeicherten Prozeduren natürlich zum täglichen Handwerkszeug. Tabellenvererbung und ausgedehnte Mapping-Funktionalitäten konnten an dieser Stelle aus Platzgründen ebenfalls nicht im Detail beschrieben werden, genauso wenig wie Transaktionen, die sich aber ganz einfach mithilfe von `TransactionScope`-Objekten der Enterprise Services von .NET Framework realisieren lassen.

Wir denken aber, dass Sie dieses Kapitel auf alle Fälle sozusagen in die »richtige Richtung gedreht hat« und Sie nun losmarschieren können. Sie haben mit dem hier erworbenen Wissen sicherlich das Zeug, leistungsfähige Datenbanken auf Entity Framework-Basis zu entwickeln, die vor allen Dingen eines sind: zukunftssicher. Es gibt noch viel zu entdecken, viel auszuprobieren, sicherlich nicht nur für Sie – auch für uns Autoren.

Teil F

Anwendungsausführung parallelisieren

In diesem Teil:

Programmieren mit der Task Parallel Library (TPL) 855

Kapitel 29

Programmieren mit der Task Parallel Library (TPL)

In diesem Kapitel:

Einführung in die Technik des Threading 856
Die verschiedenen Möglichkeiten, Threads zu starten, im Schnelldurchlauf 862
Zugriff auf Steuerelemente von Threads 868
Parallelisierung mit Parallel.For und Parallel.ForEach 873
Arbeiten mit Tasks 888
Threads synchronisieren 902
Wie geht es weiter? 914

Einführung in die Technik des Threading

Dass Gleichzeitigkeit eigentlich eine Illusion ist, hat Einstein mit seiner Relativitätstheorie bewiesen.[1] Wenn auch aus anderen Gründen, besteht dennoch der Eindruck, dass ein normaler Computer Dinge wirklich gleichzeitig erledigen könnte. Auch wenn er im Hintergrund *Black Eyed Peas* spielt, eine seiner Festplatten defragmentiert und mich gleichzeitig diese Zeilen mit Microsoft Word schreiben lässt, so zerlegt er doch alleine diese drei Sachen in klitzekleine Aufgaben, verteilt sie möglichst gut auf die vorhandenen Prozessorkerne und wenn das nicht ausreicht, weil er mehr gleichzeitig tun könnte, als Prozessorkerne da sind, arbeitet er sie im Grunde genommen nacheinander ab. Aus der Geschwindigkeit, mit der er diese kleinen Dinge hintereinander macht, entsteht dann der Eindruck, er mache wirklich alles gleichzeitig.

Szenenwechsel. Gordon Earle Moore war einer der Gründer des Prozessorherstellers Intel und machte 1965 die folgende Aussage:[2]

> *»The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000.«*
>
> *»Die Komplexität für Niedrigpreis-Komponenten hat sich um etwa den Faktor 2 pro Jahr erhöht. Man kann erwarten, dass diese Zuwachsrate auf kurze Sicht bestehen bleibt, wenn sie sich nicht sogar erhöht. Auf lange Sicht gesehen ist die Zuwachsrate etwas unvorhersehbarer, obwohl es eigentlich keinen Grund gibt, zu glauben, dass sie nicht für mindestens die nächsten 10 Jahre annähernd konstant bleiben sollte. Das würde bedeuten, dass 1975 die Anzahl der Komponenten auf einem Niedrigpreis-IC bei 65.000 liegen würde.«*

1975 korrigierte Moore diese Aussage dahingehend, dass er die Verdoppelung der Integrationsdichte bei integrierten Schaltkreisen nur noch alle *24* Monate voraussah – was übrigens der Wahrheit sehr viel näher kam: der 8080-Mikroprozessor von Intel aus dem Jahre 1974 bestand aus ca. 6.000 Transistoren, der 8086-Prozessor, der 1978 erschien, ca. 29.000.

Im Laufe der Zeit bildete sich aus dieser Aussage das so genannte Mooresche Gesetz, das besagt, dass sich »alle 18 Monate die Integrationsdichte der Komponenten eines integrierten Schaltkreises verdoppelt und damit auch dessen Leistungsfähigkeit verdoppelt.«[3]

Bis etwa 2002 bedeutete höhere Leistungsfähigkeit ganz einfach eine höhere Taktfrequenz und damit eine höhere Geschwindigkeit, mit der ein Prozessor rechnete. Wir Softwareentwickler waren im Grunde genommen fein raus: Wollten wir ein bestimmtes Problem mithilfe eines Computers in der doppelten Geschwindigkeit lösen, brauchten wir einfach nur 18 Monate zu warten.

---

[1] Siehe auch die Folge von Alpha Centauri (vom 10.06.2001), zu erreichen unter *http://tinyurl.com/6djw5b.*

[2] Siehe *http://tinyurl.com/2rl973.*

[3] Wobei Moore bestreitet, 18 Monate als Zeitspanne je selbst vorausgesagt zu haben und führt diese Aussage auf einen seiner Kollegen David House zurück.

Die Zeiten sind vorbei. Denn seit etwa 2003 haben sich die maximalen Taktfrequenzen bei Prozessoren bei zwischen 3,4 und 3,8 GHz für das maximal Machbare eingependelt. Zwar sind die Prozessoren durch interne Designänderungen bei gleicher Taktfrequenz sicherlich auch schneller geworden, aber lange nicht in dem Maße, wie es das Mooresche Gesetz eigentlich vorsieht: Würde es immer noch gelten, was die Verdoppelung der Taktfrequenz und damit der Geschwindigkeit anbelangt, müssten wir inzwischen Prozessoren mit einer Taktfrequenz von 20-30 GHz haben, und wer einen solchen hat, soll mal bitte kurz aufzeigen. Gut – dachte ich mir, hat keiner.

Da die Grenze des physikalisch Möglichen in Sachen Taktfrequenz durch Verlustleistung, den Tunneleffekt und andere Faktoren quasi erreicht ist, können Prozessoren nicht mehr wesentlich schneller werden: Man kann ihnen nur beibringen, mehrere Dinge wirklich *gleichzeitig* zu machen. Beim Schreiben der ersten Version dieses Kapitels 2003 hatte ich dies noch als Vermutung angestellt, dass der Trend in diese Richtung gehen würde. Heute wissen wir, dass die Vermutung richtig war: Damals konnte ich meine ersten Multithreading-Gehversuche auf einem der ersten Hyperthreading-Prozessoren machen, die damals verfügbar waren (Intel Pentium 4 mit 3,06 GHz – ein Vermögen hatte ich damals dafür bezahlt!). Wenn wir heute in unserer Firma spezielle Kundenwünsche erfüllen sollen in Sachen Parallelisierung, dann schauen die Computer, mit denen wir es heutzutage zu tun haben, in Sachen Leistungsfähigkeit schon einmal so aus, wie es in Abbildung 29.1 zu sehen ist.

Abbildung 29.1 Moderne Computersysteme haben schon einiges an Rechenleistung zu bieten – und während 2 Prozessoren mit 4 Kernen (Cores) und Hyperthreading zwar beeindruckend sind, sind sie – während diese Zeilen entstehen – sicherlich erst der Anfang der Multicoregeschichte

Denn leider bedeutet es nicht automatisch, dass ein Prozessor, der im Grunde genommen zwei oder vier Prozessoren in sich vereint, auch automatisch das Zwei- bzw. Vierfache an Leistung bringt. Nur wenn eine *Software* auch in der Lage ist, die vorhandenen Prozessorkerne auch wirklich zu nutzen, indem sie Aufgaben (so genannte *Tasks*) in verschiedene Threads verlagert, die dann wiederum unabhängig auf mehreren Prozessorkernen laufen können, können Sie als Benutzer (und als Entwickler zeitkritischer Systeme) auch wirklich von dieser neuen Technologie profitieren.

Sie sehen also, dass dieses Kapitel eines der wirklich wichtigen Themen der jetzigen und kommenden Zeit behandelt.

Unter dem Namen »Multitasking« hat wohl jeder, der sich nur ein wenig mit Computern beschäftigt, diese Fähigkeit schon einmal kennen gelernt. Multitasking ist die Kombination der englischen Wörter »multi« – für »viel« – und »task« – für »Aufgabe« – und bedeutet im Deutschen das, was Frauen beneidenswerterweise und normalerweise eher können als Männer:[4] nämlich mehrere Dinge zur gleichen Zeit erledigen (das gilt insbesondere für das Suchen von verloren gegangenen Dingen, wobei ich ja der Meinung bin, dass es eine Art Paralleluniversum gibt, das sich verzugsweise hinter Kühlschranktüren verbirgt und zu dem gewechselt wird, in dem Moment, in dem eine Frau die Tür öffnet. In diesem alternativen Universum existiert Butter tatsächlich im offensichtlichsten Fach, in meinem Universum hingegen liegt die Butter nicht nur nicht offensichtlich vor dem Sucher, sie existiert schlicht gar nicht).

Ein weiterer, ähnlicher Begriff stammt ebenfalls aus dem Englischen, aber er ist nicht ganz so bekannt wie »Multitasking«. Gemeint ist »Multithreading«,[5] wieder abgeleitet von »multi« – für »viel« – nur im zweiten Teil des Wortes diesmal von »thread« – für »Faden«. Man könnte eine multithreadingfähige Anwendung also als »mehrfädiges« Programm bezeichnen, wollte man den Ausdruck unbedingt übersetzen.

Und was bedeutet dieser Ausdruck jetzt genau? Dazu folgender Hintergrund: Wenn Sie eine Windows-Applikation starten, dann besteht sie aus einem Prozess und der wiederum aus mindestens einem so genannten Thread. In diesem Zusammenhang ist »Thread« eine abstrakte Bezeichnung für einen bestimmten Programmverlauf. Ein Thread startet, löst eine bestimmte Aufgabe und wird beendet, wenn diese Aufgabe erledigt ist. Wenn diese Aufgabe sehr rechenintensiv ist, dann hat das übrigens sehr ungünstige Auswirkungen für den Benutzer, denn es bedeutet für das Programm, dass es nicht weiter bedienbar ist. Der aktuelle Thread beansprucht die gesamte Rechenleistung eines Prozessorkerns (und wenn der Computer nur über einen Prozessorkern verfügt, seine komplette Rechenleistung), sodass für die Behandlung der Bedienungselemente nichts mehr übrig bleibt, wie das folgende kleine Beispielprogramm eindrucksvoll zeigt:

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 29\SingleThreadDemo
```

Mit diesem Programm haben Sie die Möglichkeit, eine Anzahl von Primzahlen unterhalb einer angebbaren Grenzzahl zu finden. Der Algorithmus, der eine Zahl daraufhin überprüft, ob sie eine Primzahl ist, ist dabei sicherlich nicht der am besten optimierte – das soll er aber auch gar nicht sein, denn schließlich geht es ja darum, eine lang andauernde Arbeitsaufgabe zu finden. Wenn Sie das Programm starten, die gewünschten

[4] So sagt es jedenfalls der Volksmund. Ob es wirklich stimmt?

[5] Ausgesprochen etwa »Malti-srädding«, und wenn Sie es ganz perfekt machen wollen, lispeln Sie das S.

Einstellungen vornehmen und dann das Ermitteln der Primzahlen anstoßen, sehen Sie nach einer Weile (nach welcher Weile, hängt maßgeblich von dem Computer ab, auf dem Sie arbeiten) ein Resultat wie in Abbildung 29.2.

Abbildung 29.2 Auf einem Netbook mit Atom-Prozessor (Atom 330 Dual Core, Hyperthreading) dauert das Ermitteln der letzten 20 Primzahlen unterhalb 100 Millionen stolze 206 Sekunden!

```
Private Sub btnZählenStarten Click(ByVal sender As System.Object,
                                   ByVal e As System.EventArgs) Handles btnZählenStarten.Click

    'Liste löschen, falls schon Elemente dort drin stehen:
    PrimesListBox.Items.Clear()

    Dim sw = Stopwatch.StartNew

    Dim PrimesFound = 0
    Dim StartNumber = Integer.Parse(LastPrimeBeforeTextBox.Text)
    Dim leadingZeros = StartNumber.ToString.Length
    For numberToTest = StartNumber To 2 Step -1
        Dim isPrime = True

        InfoLabel.Text = "Teste: " & numberToTest.ToString("#,##0")
        InfoLabel.Refresh()

        For valueToTest = 2 To numberToTest - 1
            If (numberToTest / valueToTest) = (numberToTest \ valueToTest) Then
                isPrime = False
                Exit For
            End If
        Next
        If isPrime Then
            'Kleiner (armseliger, aber unaufwendiger!) Trick:
            'damit die ListBox korrekt numerisch sortiert, packen wir einfach führende Nullen davor.
            PrimesListBox.Items.Add(numberToTest.ToString(New String("0"c, leadingZeros)))
            PrimesListBox.Refresh()

            PrimesFound += 1
            If PrimesFound = PrimesToFindUpDown.Value Then
                Exit For
            End If
        End If
    Next

    sw.Stop()
    InfoLabel.Text = "Dauer in Millisekunden: " & sw.ElapsedMilliseconds.ToString("#,##0")
End Sub
```

Wenn Sie den Task-Manager starten, bevor Sie die Primzahlenberechnung starten, dann werden Sie bei der Verwendung des Programms gleich zwei recht schwer wiegende Probleme feststellen.

- Während die Primzahlenberechnung läuft, können Sie das Programm nicht bedienen.
- Ein Prozessorkern läuft auf Volllast. Leider nutzt das Programm wirklich nur einen einzigen der zur Verfügung stehenden Prozessoren, und Abbildung 29.3 zeigt besonders eindrucksvoll, was das im Grunde genommen bedeutet.

Abbildung 29.3 Bei einem Multicore-System führt der Aufruf einer rechenintensiven Methode dazu, dass gerade mal ein verfügbarer Prozessorkern verwendet wird. Für Systeme mit sehr vielen Kernen ist das ein Desaster: Nur ein Bruchteil der möglichen Leistung wird genutzt.

Beide Probleme sind dazu noch ursächlich miteinander verknüpft. Eine Windows-Anwendung muss regelmäßig eine Nachrichtenschleife durchlaufen, um rauszufinden, ob irgendetwas passiert. Ob der Benutzer beispielsweise eine Schaltfläche anklickt oder versucht, das Fenster zu verschieben. Ganz vereinfacht ausgedrückt, sieht die Hauptschleife einer Windows-Anwendung folgendermaßen aus:

```
Sub WindowsProc()
    Do
        Dim result = DoEvents()
        If Not result Then
            Exit Do
        End If
    Loop
End Sub
```

```
Function DoEvents() As Boolean

    Dim msg = GetWindowsMessage

    If msg = WM_CLOSE Then
        Return True
    End If

    If msg = WM_FENSTERVERSCHIEBEN Then
        'Löst dann das Fenster-Verschieben-Ereignis aus
        OnFensterVerschieben()
    End If
End Function
```

Sie sehen, so lange Ihre Anwendung in der Nachrichtenwarteschleife rotieren kann, ist die Welt in Ordnung. Erst wenn irgendein Ereignis dazu führt, dass eine Methode aufgerufen wird, die unglaublich lange dauert, blockiert dieser Methodenaufruf das ständige Abfragen der Nachrichten durch `DoEvents` – weswegen man oftmals aus der Nummer rauskommt, indem man dann einfach innerhalb der blockierenden Methode selbst `DoEvents` aufruft, um quasi die Nachrichtenverarbeitung wenigstens zwischendurch mal zu erlauben.

Aber so richtig elegant ist das natürlich nicht. Die einzig legitime Lösung: Die lang andauernde Methode sollte nach Möglichkeit auf einem eigenen Thread ablaufen; der UI-Thread kann dann weiter seine Kreise ziehen, und vor allen Dingen: Es ist schon einmal etwas mehr als ein Prozessorkern mit dem Erledigen der Aufgabe beschäftigt: Ein Kern hält die Anwendung bedienbar, der andere führt die eigentliche Aufgabe aus.

Beispiele, wann Threads »angesagt« sind, gibt es viele:

- Ihr Programm muss eine umfangreiche Auswertung von Daten zusammenstellen und drucken. Diese Aufgabe könnte in einem eigenen Thread im Hintergrund passieren, ohne dass es den weiteren Arbeitsablauf des Programms stören würde.
- Ihr Programm muss umfangreiche Datensicherungen durchführen. Ein Thread könnte Momentaufnahmen dieser Dateien erstellen und sie anschließend sozusagen im Hintergrund auf eine andere Ressource kopieren.
- Ihr Programm muss Zustände bestimmter Hardwarekomponenten aufzeichnen – beispielsweise Produktionsdaten von Maschinen abrufen. Auch diese Funktion könnte im Hintergrund ablaufen; Auswertungsfunktionen könnten dennoch zu jeder Zeit parallel laufen und vom Benutzer verwendet werden, wenn die Produktionsdatenerfassung als Thread im Hintergrund läuft.
- Bei komplizierten Berechnungen oder Auswertungen könnten zwei oder mehrere Threads diese Aufgabe übernehmen. In diesem Fall würde auf Hyperthreading- oder Multiprozessorsystemen eine Auslastung mehrerer Prozessoren durch *jeweils* einen eigenen Thread die Verarbeitungsgeschwindigkeit der gesamten Aufgabe deutlich erhöhen.
- In der Bildverarbeitung, beispielsweise beim Konvertieren von Bildern, Anwenden von Filtern auf Bilder oder beim Skalieren von Bildern (also bei der Verkleinerung oder Vergrößerung) kann ein Bild in unterschiedliche Zeilen oder andere Bereiche aufgeteilt werden, und diese Bildteile werden dann auf unterschiedlichen Threads parallel bearbeitet.

Gelegenheiten gibt es also viele, um Threads einzusetzen. Allerdings sind bei Threads auch ein paar Dinge zu beachten: Threads dürfen sich keinesfalls gegenseitig ins Gehege kommen. Und auch nicht gemeinsam eine Ressource so überbeanspruchen, dass es schneller gehen würde, wenn sie ihre Arbeit nacheinander erledigen würden – beispielsweise bei intensiven Zugriffen auf die Festplatte. Doch dazu später mehr.

Betrachten wir zunächst die Grundlagen, also wie wir .NET Framework überhaupt dazu bewegen können, dass bestimmte Teile einer Anwendung als eigener Thread ausgeführt werden können.

Die verschiedenen Möglichkeiten, Threads zu starten, im Schnelldurchlauf

.NET Framework kennt folgende Möglichkeiten, um eine bestimmte Aufgabe auf einem individuellen Thread zu erledigen:

- Die Methode, die die umfangreiche Berechnung ausführen soll, wird über einen Delegaten asynchron aufgerufen. Dies ist die einfachste Möglichkeit, einen Thread zu starten, aber auch die, die Ihnen die geringste Kontrolle über den Thread bietet.
- Sie verwenden eine der speziellen Parallelisierungsmethoden aus der `Parallel`-Klasse, zum Beispiel `Parallel.For` oder `Parallel.ForEach`
- Sie formulieren eine LINQ-Abfrage und versuchen, die Abfrage durch den Zusatz `AsParallel` parallelisieren zu lassen
- In Windows Forms-Anwendungen können Sie die `BackgroundWorker`-Komponente verwenden. Diese hat den Vorteil, dass Sie Ihnen beim Aktualisieren der Benutzeroberfläche hilft, denn Sie dürfen sowohl bei Windows Forms-Anwendungen als auch bei WPF-Anwendungen nur von dem Thread auf Steuerelemente der Benutzeroberfläche zugreifen, der diese Steuerelemente auch erstellt hat. In der Regel ist das der Thread, der standardmäßig gestartet wird, wenn Ihre Anwendung anläuft. Deswegen heißt dieser Thread auch *Benutzeroberflächen-Thread* oder kurz *UI-Thread* (sprich: *Juh Ei*, von Englisch: *User Interface*).
- Sie verwenden direkt einen Thread des so genannten Threadpools. Der Threadpool stellt Ihnen unter .NET bereits eine Reihe vordefinierter `Thread`-Objekte zur Verfügung, die Sie aus dem Threadpool im Bedarfsfall entnehmen können. Alternativ können Sie auch ein neues `Thread`-Objekt erstellen und dieses verwenden. Das Problem liegt lediglich darin, dass Threads sehr ressourcenintensiv sind: Das Erstellen eines neuen Threads erfordert zum Beispiel, dass für diesen Thread auch ein entsprechender Prozessorthread bereitgestellt wird, und der entsprechende Stack benötigt nicht weniger als 1 MB des Hauptspeichers. Sie können ab .NET 4.0 das `Task`-Objekt verwenden, (*Task*, engl. *Aufgabe*), um darüber einen Thread des Threadpools zu steuern. Sie haben auch die Möglichkeit, auf die statische `ThreadPool`-Klasse direkt zuzugreifen, wenn Sie mit vorherigen Versionen von .NET Framework arbeiten.
- Sie verwenden die `Thread`-Klasse direkt. Ein neuer Thread, der über eine Instanz dieser Klasse erstellt wird, dauert, wie gesagt, am längsten und benötigt die meisten Ressourcen und eignet sich deswegen ideal für Threads, die beispielsweise neben dem UI-Thread gestartet werden und sehr lange ablaufen. In der Tat ist es sogar so, dass die `Task`-Klasse bei der Verwendung des Standardschedulers (dazu später mehr) selbst sogar keinen Thread aus dem Threadpool nimmt, sondern ein `Thread`-Objekt anlegt, wenn Sie mit den Optionen angeben, dass es sich um einen lang andauernden Thread handeln soll.

HINWEIS Wann immer Sie mit `Parallel.For`, `Parallel.ForEach` oder durch Parallelisieren von LINQ-Abfragen implizit das Erledigen einer Aufgaben auf mehrere Prozessorkerne verteilen, verwendet .NET Framework dazu ebenfalls Threads aus dem Threadpool.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 29\SimpleThread01
```

Abbildung 29.4 SimpleThread-Demo demonstriert die verschiedenen Möglichkeiten, Threads zu starten

```
'Dies ist der eigentliche Arbeits-Thread (Worker Thread), der das
'Hochzählen und die Werteausgabe übernimmt
Private Sub ComplexCalculation()
    Thread.CurrentThread.Name = "Arbeits-Thread: " + myArbeitsThreadNr.ToString
    For c As Integer = 0 To LOOPCOUNT
        'Dient zur Ausgabe des Werts. TSWriteLine ist eine statische
        'Prozedur, die für die Darstellung des Fensters selbst sorgt,
        'sobald sie das erste Mal verwendet wird.
        TextOutput.TSWriteLine(Thread.CurrentThread.Name + ":: " + c.ToString)
        'Aktuellen Thread um 100ms verzögern, damit die ganze
        'Geschichte nicht zu schnell vorbei ist.
        Thread.Sleep(100)
    Next
End Sub
```

Die Anwendung simuliert auf einfachste Weise eine Methode mit Arbeitslast: `ComplexCalulation`, wie oben zu sehen, iteriert eine bestimmte Anzahl oft und zählt dabei eine Zählvariable hoch (c). Diese Zählvariable wird in einem speziellen Ausgabefenster ausgegeben – in der Abbildung zu sehen sind dabei zwei Threads, die diese Routine gleichzeitig laufen lassen.

HINWEIS Die Ausgabe einer Zeichenfolge beispielsweise in eine TextBox oder ein Label ist weder unter Windows Forms noch unter WPF etwas, was Sie ohne zusätzlichen Aufwand machen könnten. Ein einfaches

```
AusgabeLabel.Text = "Neue Beschriftung"
```

würde immer zu einer Ausnahme führen. Für unsere Experimente verwenden wir deswegen im Folgenden eine Komponente namens `ThreadSafeTextWindowComponent`. Diese Komponente fügen Sie dem Komponentenfach einer Windows Forms Anwendung hinzu (Sie ziehen sie aufs Formular, und sie landet im Komponentenfach). Anschließend können Sie ihre `TSWrite`- und `TSWriteLine`-Methoden verwenden, die bei der ersten Verwendung automatisch dazu führen, dass ein entsprechendes Fenster geöffnet wird, in dem der auszugebende Text dargestellt wird. Die Methoden sind, im Gegensatz zu allen anderen Windows Forms- oder WPF-Steuerelementen threadsicher. Das bedeutet, Sie können sie von beliebigen Threads aus aufrufen, ohne befürchten zu müssen, dass sie eine Ausnahme auslösen. Mehr zum Zugriff auf Steuerelemente unter Windows Forms oder WPF erfahren Sie unter »Zugriff auf Steuerelemente von Threads« ab Seite 868.

Nur wenn Sie die `BackgroundWorker`-Komponente verwenden, brauchen Sie sich um das Fehlschlagen von Zugriffen auf Steuerelemente von anderen Threads aus keine Gedanken zu machen, da diese Komponente den Zugriff intern regelt.

Verwenden der Thread-Klasse

Die klassische Art, einen neuen Thread zu starten, führt über eine `Thread`-Instanz. Diese Möglichkeit steht auf allen .NET Framework-Versionen und auch auf Compact Framework zur Verfügung. Um sie im oben erwähnten Beispielprogramm auszuprobieren, klicken Sie auf die Schaltfläche *Thread mit Thread-Instanz starten*. Der folgende Code wird dann ausgeführt, und dieser bewirkt, dass die Methode `ComplexCalculation` auf einem eigenen Thread ausgeführt wird:

```
Private Sub btnThreadStarten_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
                                   Handles ThreadWithThreadButton.Click

    'Dieses Objekt kapselt den eigentlichen Thread
    Dim locThread As Thread
    'Dieses Objekt benötigen Sie, um die Prozedur zu bestimmen,
    'die den Thread ausführt.
    Dim locThreadStart As ThreadStart

    'Threadausführende Prozedur bestimmen
    locThreadStart = New ThreadStart(AddressOf ComplexCalculation)

    'ThreadStart-Objekt dem Thread-Objekt übergeben
    locThread = New Thread(locThreadStart)

    'Einen Thread daraus machen, der beendet wird, wenn die Hauptanwendung sich beendet:
    locThread.IsBackground = True

    'Thread starten
    locThread.Start()
    'Zähler, damit die Threads durch ihren Namen unterschieden werden können
    myArbeitsThreadNr += 1
End Sub
```

Dies ist die ausführliche Variante: Sie definieren einen neuen `ThreadStart`-Delegaten (mehr zu Delegaten in Kapitel 16), der auf die Methode zeigt, die Sie als eigenständigen Thread ausführen wollen. Diesen `ThreadStart`-Delegaten übergeben Sie dem `Thread`-Objekt während der Instanziierung.

Hintergrundthreads werden stets mit der Hauptanwendung beendet

Das Beispiel setzt anschließend die `IsBackground`-Eigenschaft des `Thread`-Objekts auf `True`. Das macht aus dem neuen Thread einen Hintergrundthread, was wiederum bedeutet, dass der Thread auf jeden Fall mit der Anwendung beendet wird und nicht dazu führt, dass die Anwendung im Speicher hängen bleibt.

WICHTIG Achten Sie darauf, wenn Sie `Thread`-Objekte verwenden: `Thread`-Objekte, deren Eigenschaft `IsBackground` *nicht* gesetzt wird, haben zur Folge, dass der Prozess, der sie erstellt hat, so lange nicht beendet wird, bis die `Thread`-Objekte selbst beendet sind. Auch wenn die Benutzeroberfläche der eigentlichen Anwendung schon nicht mehr dargestellt wird, ist der Prozess der Anwendung erst zu Ende, wenn auch der letzte Thread beendet wird.

Wenn die `IsBackground`-Eigenschaft gesetzt ist, wird der Thread, auch wenn er noch läuft, immer zusammen mit der Anwendung beendet (bzw. abgebrochen).

Vereinfachung des Startens eines Threads mit mehrzeiligen Anwendungs-Lambdas

Seit Visual Basic 2010 gibt es mehrzeilige Anweisungs-Lambdas, die Code enthalten, der direkt einer Delegatenvariable zugewiesen oder direkt anstelle einer Delegatenvariable einer Methode übergeben werden kann. Zusammen mit Eigenschafteninitialisierern kann man den Code der oben stehenden Methode zum Ausführen der Methode `ComplexCalculation` damit folgendermaßen verkürzen:

```
    'Seit Visual Basic 2010 erspart der Compiler durch Multistatement Lambdas
    'einen Haufen Tipparbeit:
    Private Sub ThreadStartShortVersion()

        Dim locThread As New Thread(Sub()
                                        'Gleiche Schleife wie in Methode 'Umfangreiche Berechnung'.
                                        For c As Integer = 0 To LOOPCOUNT
                                            TextOutput.TSWriteLine(Thread.CurrentThread.Name +
                                                                   ": " + c.ToString)
                                            Thread.Sleep(100)
                                        Next
                                    End Sub) With
                                 {.IsBackground = True,
                                  .Name = "Arbeits-Thread: " + myArbeitsThreadNr.ToString}
        locThread.Start()
        myArbeitsThreadNr += 1
    End Sub

Anstelle des Delegaten wird hier dem Thread-Konstruktor die Arbeitsmethode als Lambda direkt übergeben.
Die Eigenschaften des Threads werden zudem mithilfe des With-Schlüsselwortes in Form von
Eigenschafteninitialisierern übergeben. Das Instanziieren des Thread-Objekts, wie hier zu sehen,
entspricht exakt der ausführlichen Version, spart nur – weil Sie dem Compiler eine Menge an
Codegenerierung überlassen – eine Menge Tipparbeit und damit Zeit.
```

Asynchrones Aufrufen von Delegaten

Delegatenvariablen, die auf gültige Methoden zeigen, können direkt wie Methoden aufgerufen werden, so wie es das folgende Beispiel zeigt:

```
Dim helloWorldAction As Action = Sub()
                                     Debug.Print("Hello World!")
                                 End Sub
helloWorldAction()

'Alternativ, bewirkt dasselbe:
helloWorldAction.Invoke()
```

Sie kennen das bereits aus Kapitel 16, das sich mit Delegaten genauer auseinander setzt. Sie haben aber auch die Möglichkeit, einen Delegaten asynchron aufzurufen, und das geschieht dann nicht mit der `Invoke`- sondern mit `BeginInvoke`-Methode. Dabei wird dann automatisch ein Thread aus dem Threadpool entnommen und die Methode auf diesem Thread ausgeführt. `BeginInvoke` übergeben Sie, neben den Parametern, die die Methode erwartet, auch gleichzeitig einen Delegaten (der seit Visual Basic 2010, wie hier im Beispiel zu sehen, der Einfachheit halber auch wieder ein Lambda sein kann), der ausgeführt wird, sobald der asynchrone Vorgang abgeschlossen ist.

Wichtig dabei ist: Wenn eine Methode mit `BeginInvoke` aufgerufen wird, liefert sie ein Objekt zurück, das Ergebnisparameter einer asynchronen Operation beinhaltet und deswegen die `IAsyncResult`-Schnittstelle einbindet:

```
Dim aDelegate As action = AddressOf ComplexCalculation

Dim asyncres As IAsyncResult = Nothing
asyncres = aDelegate.BeginInvoke(Sub(value As Object)
                                     'Dieser Code wird ausgeführt,
                                     'wenn der Task abgeschlossen ist.
                                     TextOutput.TSWriteLine("Worker Thread abgeschlossen!")

                                     'Wichtig: Immer mit EndInvoke abschließen!
                                     aDelegate.EndInvoke(asyncres)
                                     TextOutput.TSWriteLine("EndInvoke abgeschlossen!")
                                 End Sub, New Object)
myArbeitsThreadNr += 1
```

Mithilfe dieses Objekts können Sie den asynchronen Vorgang steuern – worauf wir nicht im Detail eingehen wollen, weil sich die `Task`-Klasse zu diesem Zweck sehr viel besser eignet. Achten Sie aber darauf, dass Sie dieses `IAsyncResult`-Objekt `EndInvoke` mitgegeben, um den asynchronen Vorgang sauber abzuschließen (siehe fett markierte Zeile im Listing) – und dass passiert idealerweise natürlich in der Methode, die aufgerufen wird, wenn die eigentliche Arbeitsmethode abgeschlossen wurde.

HINWEIS In .NET Compact Framework steht Ihnen das asynchrone Aufrufen von Methoden auf die hier beschriebene Weise leider nicht zur Verfügung.

Verwenden der Task-Klasse

Die `Task`-Klasse für das Abarbeiten einer Methode auf einem eigenen Thread zu verwenden, ist die sinnvollste Variante – allerdings steht Ihnen diese Klasse erst im Rahmen der Task Parallel Library zur Verfügung, und diese ist erst seit .NET Framework 4.0 Bestandteil der Common Language Runtime.

Ein Task entspricht in der Regel einem Thread des Threadpools. Abweichend davon können Tasks auch zu einer neuen `Thread`-Instanz führen, wenn Sie beispielsweise den Standardscheduler verwenden und dabei angeben, dass Sie eine lang andauernde Operation ausführen wollen – doch das sind Implementierungsdetails, die im Normalfall nicht interessieren müssen.

Sie erstellen eine `Task`-Instanz entweder durch Instanziieren der `Task`-Klasse oder mithilfe der Factory-Methode `Task.Factory.StartNew`. Die Factory-Methode erzeugt nicht nur den neuen Task, sondern startet ihn auch direkt. Beim manuellen Instanziieren des Tasks muss der Thread, den der Task darstellt, durch die `Start`-Methode gestartet werden. Für unser Beispiel ergeben sich also die folgenden Möglichkeiten:

```
Dim t = Task.Factory.StartNew(AddressOf ComplexCalculation)
```

Möchten Sie die `Task`-Instanz selbst erstellen, verfahren Sie wie folgt:

```
'Alternative:
Dim t2 = New Task(Sub()
                      'Gleiche Schleife wie in Methode 'Umfangreiche Berechnung'.
                      For c As Integer = 0 To LOOPCOUNT
                          TextOutput.TSWriteLine(Thread.CurrentThread.Name + ":: " + c.ToString)
                          Thread.Sleep(100)
                      Next
                  End Sub)
t2.Start()
```

Sie könnten in dieser alternativen Methode auch wieder die Adresse der Methode `ComplexCalculation` angeben; seit Visual Basic 2010 bietet es sich aber auch hier wieder an, den entsprechenden Code als mehrzeiliges Anweisungs-Lambda zu formulieren, so wie es das Beispiel zeigt.

HINWEIS Mit der `ContinueWith`-Methode können Sie einen weiteren Task zum Ausführen einreihen; die Methode, die hinter `ContinueWith` angegeben wird, startet dann, sobald der Task beendet wurde. Auf diese Weise lassen sich auch mehrere Aufgaben als `Task`-Objekte hintereinanderschalten. Im Beispiel wird der `ContinueWith`-Methode lediglich ein Lambda übergeben, das einen Hinweis ausgibt, dass die eigentliche Arbeitsmethode abgeschlossen wurde.

```
t.ContinueWith(Sub()
                   TextOutput.TSWriteLine("Worker Thread abgeschlossen!")
               End Sub)
```

Direktes Verwenden eines Thread des Threadpools

Wenn Sie gegen eine kleinere als die Version 4.0 von .NET Framework entwickeln, steht Ihnen der Komfort und die Zuverlässigkeit der `Task`-Klasse leider nicht zur Verfügung, aber Sie können dennoch Threads des Threadpools verwenden. Mit der Methode `QueueUserWorkItem` übergeben Sie einen Delegaten, und der nächste freie Thread des Threadpools übernimmt die Verarbeitung der Methode:

```
ThreadPool.QueueUserWorkItem(Sub(value)
                                 ComplexCalculation()
                             End Sub)
```

HINWEIS `QueueUserWorkItem` erwartet einen Delegaten, der der Signatur einer Methode mit einem Parameter vom Typ `Object` entspricht. Das ist der Grund, weshalb wir an dieser Stelle nicht direkt `ComplexCalculation` mit `AddressOf` übergeben können.

Zugriff auf Steuerelemente von Threads

Wenn Sie Steuerelemente der Hauptanwendung von einem Thread aus verwenden wollen, müssen Sie Folgendes beachten: Steuerelemente sind nicht threadsicher, das heißt, sie dürfen nur aus dem Thread heraus verwendet werden, der sie erstellt hat. In der Regel ist das der Hauptthread der Anwendung, also der Thread, der die Benutzeroberfläche Ihrer Anwendung regelt. Aus diesem Grund nennt man diesen Thread auch, wie schon erwähnt, den *UI*-Thread (*UI* als Abkürzung für *User Interface*, oder zu Deutsch: *Benutzeroberfläche*).

Erinnern wir uns: Jede Windows-Anwendung läuft in einer riesigen Endlosschleife, wenn Sie so wollen. Erst wenn das Hauptfenster geschlossen wird, ist das die Abbruchbedingung für diese Endlosschleife. Damit die Ereignisverwaltung in geordneten Bahnen verlaufen kann, darf nicht jeder Thread zur gleichen Zeit auf ein Steuerelement zugreifen und dessen Eigenschaften verändern; das muss nacheinander passieren. Also müssen Threads warten, damit der Thread, der ein Steuerelement gerade aktualisiert, seine Arbeit ungestört erledigen kann. Jetzt können sich aber Abhängigkeiten dabei bilden, die dazu führen können, dass gar nichts mehr geht: Der Thread, der ein Steuerelement gerade aktualisiert, wartet auf die Daten eines anderen Steuerelements, das aber ein anderer Thread gerade in Beschlag hat. Er gibt das aktuelle Steuerelement deswegen nicht frei. Der UI-Thread wartet wiederum, dass das 3. Steuerelement frei wird, und die ganze Nachrichtenwarteschleife steht – es geht nicht vor und nicht zurück. Dieser Zustand, den man im Übrigen Deadlock (Verklemmung) nennt, muss unter allen Umständen vermieden werden, und das geht am besten, wenn nur der Thread Steuerelemente aktualisieren darf, der diese Steuerelemente auch erstellt hat. Nur dann ist gewährleistet, dass die Anwendung nicht »festfriert«.

Damit ein anderer Thread dennoch das Steuerelement eines Formulars verwenden kann, muss er sich eines Tricks bedienen: Er muss der Nachrichtenwarteschlage, die ihre Kreise ja auf dem Ausgangsthread, dem UI-Thread zieht, mitteilen, dass sie stellvertretend für ihn die gewünschte Methode zum Aktualisieren des Steuerelements aufrufen soll, und zwar dann, wenn die Nachrichtenwarteschlage meint, dass der Zeitpunkt günstig ist. Und erst diese Methode verändert dann zum Beispiel die Eigenschaften des Steuerelements. Und da das Steuerelement diese Methode – wenn auch indirekt – selbst aufgerufen hat, wurde sie auch vom richtigen Thread (nämlich dem UI-Thread) aus aufgerufen.

Das Steuerelement selbst stellt dafür eine der wenigen threadsicheren Prozeduren zur Verfügung, die es besitzt. Ihr Name: `Invoke`. Ihr wird als Parameter ein Delegat übergeben, der als Zeiger auf die eigentliche Prozedur dient, die aus dem Thread aufgerufen werden soll.

Wie schaut das nun in der Praxis aus? Mit dem Wissen um das Starten von Threads können wir uns unser Primzahlenausgangsbeispiel nochmal zur Brust nehmen und die eigentliche Berechnung der Primzahlen mithilfe des `Task`-Klasse relativ simpel in einen eigenen Thread verbannen. Damit machen wir die Anwendung im ersten Schritt zwar noch nicht schneller, aber immerhin bedienbar, da der UI-Thread und seine Warteschleife nicht auf das Ende der Berechnung warten muss: Wir sehen dabei auch gleich, was passiert, wenn wir das Aktualisieren der Steuerelemente nicht auf den UI-Thread umleiten (engl. übrigens: *to marshal*).

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 29\InvokeDemo
```

```
Dim calcPrimesTask =
    Task.Factory.StartNew(
        Sub()
            For numberToTest = Integer.Parse(LastPrimeBeforeTextBox.Text) To 2 Step -1
                Dim isPrime = True

                InfoLabel.Text = "Teste: " & numberToTest.ToString("#,##0")
                InfoLabel.Refresh()

                For valueToTest = 2 To numberToTest - 1
                    If (numberToTest / valueToTest) = (numberToTest \ valueToTest) Then
                        isPrime = False
                        Exit For
                    End If
                Next
                If isPrime Then
                    PrimesListBox.Items.Add(numberToTest.ToString("#,##0"))
                    PrimesListBox.Refresh()

                    PrimesFound += 1
                    If PrimesFound = PrimesToFindUpDown.Value Then
                        Exit For
                    End If
                End If
            Next

            sw.Stop()
            InfoLabel.Text = "Dauer in Millisekunden: " & _
                         sw.ElapsedMilliseconds.ToString("#,##0")
        End Sub)
```

Der gesamte zwischen `Sub` und `End Sub` stehende Methodenrumpf ist nun als Anweisungs-Lambda formuliert und wird als Delegat dem `Task`-Objekt übergeben, das die Task-Factory-Klasse instanziiert und auch sofort startet. Damit läuft die Primzahlenberechnung bereits auf einem zweiten Thread, und für ein paar Sekunden ist die Welt sogar in Ordnung. Bis der Task den Versuch startet, die Benutzeroberfläche zu aktualisieren, denn dann knallt es (Abbildung 29.5).

Abbildung 29.5 Der Versuch, auf ein Steuerelement von einem anderen Thread aus zuzugreifen, schlägt fehl

Die Lösung dieses Problems liegt nun darin, exakt die Schritte zu unternehmen, damit genau der Code, der die Steuerelemente aktualisiert, nicht auf diesem Arbeitsthread, sondern vom UI-Thread ausgeführt wird. Die Methode `Invoke`, die Sie in Verbindung mit jedem Steuerelement finden, macht genau das: Sie übernimmt mindestens einen Parameter, der den Delegaten widerspiegelt, der wiederum auf die Methode zeigt, die auf dem UI-Thread ausgeführt werden soll. In unserem Beispiel ist klar, dass das Aktualisieren der Steuerelemente mit `Invoke` auf den UI-Thread umgeleitet werden muss – in anderen Konstellationen muss das aber nicht unbedingt der Fall sein: In diesen Fällen überprüfen Sie zunächst mit `InvokeRequired`, ob das Umleiten auf den UI-Thread überhaupt notwendig ist. Das Einfachste ist auch hier wieder der Einsatz von Lambdas, sodass sich folgendes Muster ergibt, das Sie in diesem Zusammenhang immer wieder einsetzen können (natürlich müssen Sie den Inhalt des Lambdas entsprechend ändern, damit er konform zu Ihren Steuerelementen und den Eigenschaften ist, die verändert werden sollen):

HINWEIS Denken Sie daran, den Namespace `System.Threading.Tasks` zu importieren, wenn Sie mit der Task Parallel Library arbeiten, zu der auch die `Task`-Klasse gehört, die wir im Folgenden verwenden:

```
Imports System.Threading.Tasks
```

```
Dim labelUpdaterDel = Sub(numToTest As Integer)
                          InfoLabel.Text = "Teste: " & numToTest.ToString("#,##0")
                          InfoLabel.Refresh()
                      End Sub

If Not InfoLabel.InvokeRequired Then
    labelUpdaterDel(numberToTest)
Else
    InfoLabel.Invoke(labelUpdaterDel, {numberToTest})
End If
```

Dieser Methodenteil definiert dazu eine neue Delegatenvariable, die auf den Lambda-Ausdruck zeigt. Der Lambda-Ausdruck übernimmt übrigens selbst eine Variable als Parameter, die dem Wert entspricht, den das Label als aktuellen Wert anzeigen soll.

HINWEIS Gewöhnen Sie sich beim Parallelisieren von Methoden am besten so schnell wie möglich daran, in Lambdas innerhalb von Schleifen *nicht* auf Variablen zuzugreifen, die sich innerhalb der Schleife, aber außerhalb des Lambdas verändern. Sie können nämlich die Ausführung des Lambdas und damit den Wert der Zählvariablen innerhalb der Schleife nicht mit absoluter Sicherheit voraussagen. In einigen Fällen erkennt der Compiler das auch und gibt eine entsprechende Warnung aus. Mehr zu diesem Thema erfahren Sie im Abschnitt »Vermeiden von Fehlern beim Parallelisieren von Schleifen« ab Seite 885.

Abbildung 29.6 Verwenden Sie in einem Lambda-Ausdruck, der sich innerhalb einer Schleife befindet, nach Möglichkeit keine Iterationsvariable, da das Lambda unter Umständen auch *nach* Abschluss der Schleife ausgeführt werden kann, und der Iterationswert dann falsch ist

Die auf diese Weise umgearbeitete Berechnungsmethode sieht dann im Ganzen folgendermaßen aus. Die im Vergleich zum vorherigen Stand geänderten Zeilen finden Sie im folgenden Listing fett ausgezeichnet.

```
Private Sub StartCalculationButton Click(ByVal sender As System.Object,
                                         ByVal e As System.EventArgs) Handles StartCalculationButton.Click

    Dim sw = Stopwatch.StartNew

    'Für die Threading-Version holen wir uns
    'den Wert des Steuerelements noch im UI-Thread,
    'dann brauchen wir später nicht zu deligieren.
    Dim primesToFind = PrimesToFindUpDown.Value
    Dim StartNumber = Integer.Parse(LastPrimeBeforeTextBox.Text)
    Dim leadingZeros = StartNumber.ToString.Length

    Dim PrimesFound = 0
    Dim calcPrimesTask =
        Task.Factory.StartNew(
            Sub()
                For numberToTest = StartNumber To 2 Step -1
                    Dim isPrime = True

                    Dim labelUpdaterDel = Sub(numToTest As Integer)
                                              InfoLabel.Text = "Teste: " &
                                          numToTest.ToString("#,##0")
                                              InfoLabel.Refresh()
                                          End Sub
```

```
                    If Not InfoLabel.InvokeRequired Then
                        labelUpdaterDel(numberToTest)
                    Else
                        InfoLabel.Invoke(labelUpdaterDel, {numberToTest})
                    End If

                    For valueToTest = 2 To numberToTest - 1
                        If (numberToTest / valueToTest) = (numberToTest \ valueToTest) Then
                            isPrime = False
                            Exit For
                        End If
                    Next
                    If isPrime Then

                        Dim listBoxUpdater = Sub(numToTest As Integer)
                         PrimesListBox.Items.Add(numToTest.ToString(New String("0"c, leadingZeros)))
                         PrimesListBox.Refresh()
                        End Sub

                        If Not PrimesListBox.InvokeRequired Then
                            listBoxUpdater(numberToTest)
                        Else
                            PrimesListBox.Invoke(listBoxUpdater, {numberToTest})
                        End If

                        PrimesFound += 1
                        If PrimesFound = primesToFind Then
                            Exit For
                        End If
                    End If
                Next

                sw.Stop()
                'Hier machen wir's jetzt direkt, ohne nachzufragen:
                InfoLabel.Invoke(
                    Sub()
                        InfoLabel.Text = "Dauer in Millisekunden: " &
                            sw.ElapsedMilliseconds.ToString("#,##0")
                    End Sub)
            End Sub)
    End Sub
```

Probieren Sie es aus: Zwar ist das Programm mit diesen Änderungen nicht schneller als vorher, aber dadurch, dass Berechnung und Bedienung auf unterschiedlichen Threads laufen, bleibt es zu jedem Zeitpunkt voll bedienbar und blockiert nicht.

TIPP Die `Invoke`-Methode ist eine blockierende Methode, was bedeutet, dass sie in jedem Fall ausgeführt wird und so lange dauert, bis die übergebende Methode auf dem UI-Thread ausgeführt wurde. Wenn dieser zu diesem Zeitpunkt aus welchen Gründen auch immer blockiert ist, dann blockiert natürlich auch der Thread an dieser Stelle, der `Invoke` aufruft. Vielleicht blockiert der UI-Thread sogar wegen dieses Threads – in diesem Moment sehen Sie sich mit einer klassische Deadlock-Situation konfrontiert, aus der es kein Entrinnen mehr gibt. Ganz problematisch wird das, wenn Sie auf dem UI-Thread mit einer der `Task.WaitXXX`-Methoden (siehe Abschnitt »Verhindern, dass die Benutzeroberfläche beim Warten auf Tasks blockiert« ab Seite 895) auf das Beenden eines Threads warten, der gleichzeitig über `Control.Invoke` die Benutzeroberfläche aktualisieren will. Dann ist der Deadlock sicher.

In einigen Fällen kann es sinnvoll sein, das Umleiten des Threads auf den UI-Thread nicht blockierend vorzunehmen. Verwenden Sie in diesem Fall die Methode `BeginInvoke` des entsprechenden Steuerelements. `BeginInvoke` hat im Gegensatz zu `Invoke` den Nachteil, dass es nicht deterministisch ist, und das heißt, Sie können nie genau sagen, wann der Methodenaufruf stattfindet.

HINWEIS In WPF gibt besteht ebenfalls die Notwendigkeit, Threads an den UI-Thread zu delegieren, doch das funktioniert ein wenig anders als in Windows Forms. In den Begleitdateien zum Buch finden Sie die Anwendung *InvokeDemo_WPF*, die das Äquivalent dieses Demoprogramms unter WPF darstellt:

```
Dim labelUpdaterDel = Sub(numToTest As Integer)
                          InfoLabel.Text = "Teste: " & numToTest.ToString("#,##0")
                      End Sub

If InfoLabel.Dispatcher.CheckAccess() Then
    labelUpdaterDel(numberToTest)
Else
    InfoLabel.Dispatcher.Invoke(labelUpdaterDel, numberToTest)
End If
```

`CheckAccess` entspricht also dem Windows Forms `InvokeRequired` und `Invoke` eben dem `Invoke`. Auch `BeginInvoke` steht Ihnen über das `Dispatcher`-Objekt des entsprechenden Steuerelements zur Verfügung und wird genau wie in Windows Forms angewendet.

Aber Achtung: Wundern Sie sich nicht, wenn Sie bis inklusive Visual Basic 2010 und .NET Framework 4.0 für die Methode `CheckAccess` keine IntelliSense-Unterstützung bekommen. Die Methode ist dennoch da, und sie wird auch vom Compiler akzeptiert!

Parallelisierung mit Parallel.For und Parallel.ForEach

Die meiste Prozessorzeit innerhalb von Methoden wird in der Regel mit Operationen verbraucht, die wiederholt, teilweise 1000-fach ausgeführt werden müssen, also in Schleifen. Da hier die meiste Prozessorzeit benötigt wird, ist es geschickt, an dieser Stelle anzusetzen, um möglichst viel dieser Last auf die zur Verfügung stehenden Prozessorkerne zu verteilen.

Denken Sie an unser Beispiel aus Kapitel 6: Dort haben wir Bilder pixelweise verkleinert, und ein Prozessorkern musste quasi zeilenweise durch das Bild iterieren und nur jeden x. Pixel einer Spalte dieser Zeile in das neue, kleinere Bild übernehmen. Bei dieser Vorgehensweise spricht nichts dagegen, diesen Vorgang auf mehrere Prozessorkerne zu verteilen: Angenommen, Sie verfügen über ein System mit 4 Kernen, wie es ja bei modernen Computern und selbst Notebooks durchaus schon üblich ist. Dann könnte der 1. Kern die erste Zeile, der 2. Kern die zweite Zeile, der 3. Kern die dritte Zeile etc. übernehmen, bis schließlich alle Kerne damit beschäftigt sind, eine der Zeilen und damit einen der Schleifendurchläufe zu bearbeiten.

Bei `Parallel.For` bzw. `Parallel.ForEach` ist das ähnlich. Hier versucht die Methode den Methodenblock, der für jedes Element einer Auflistung einmal aufgerufen wird, möglichst ideal auf alle vorhandenen Kerne zu verteilen. Das wiederum heißt auch hier, dass zwar alle Elemente, wie beim normalen `For Each`-Konstrukt, einmal drankommen, aber das auch hier die Reihenfolge nicht vorhersagbar ist.

Parallel.For

Soweit klingt das erst einmal ganz toll, und man möchte damit beginnen, einfach alle Schleifen eines Programms einfach auf `Parallel.For` umzustellen, und schon laufen die entsprechenden Algorithmen 2-mal, 4-mal oder vielleicht sogar 8- oder 16-mal schneller! Doch ganz so einfach ist das nicht, denn:

Das große Problem ist, dass die einzelnen Schleifendurchläufe nicht mehr in fortlaufender Reihenfolge passieren und auch nicht mehr nacheinander, sondern gleichzeitig. Wenn der Algorithmus im Inneren Ihrer Schleife also kein Problem damit hat, zwar vollständig jeden Wert, aber durcheinander zu zählen, dann stellt dies kein Problem dar: Die Folge 1, 4, 6, 8, 7, 2, 3, 5, 9, 10 enthält auch alle Zahlen, nur eben nicht nacheinander sortiert. Probieren Sie es mit einer einfachsten Konsolenanwendung aus:

HINWEIS Denken Sie daran, den Namespace `System.Threading.Tasks` zu importieren, wenn Sie mit der Task Parallel Library arbeiten, zu der auch die `Task`-Klasse gehört, die wir im Folgenden verwenden. Darüber hinaus verwenden wir die `Sleep`-Methode, die sich an der `Thread`-Klasse befindet, die wiederum im `System.Threading`-Namespace liegt.

```
Imports System.Threading
Imports System.Threading.Tasks
```

```
Module Module1

    Sub Main()
        Parallel.For(1, 21, Sub(i)
                                Console.WriteLine(i.ToString)
                                'So tun, als würde was 500 ms dauern:
                                Thread.Sleep(500)
                            End Sub)
        Console.Read()
    End Sub
End Module
```

HINWEIS `Thread.Sleep` sorgt dafür, dass der Thread (und ein Task, den Sie initiieren, ist ja nichts anderes als ein Thread des Threadpools) sich die angegebene Anzahl an Millisekunden schlafen legt. Dabei verbraucht er keine Prozessorlast, und er wacht von selbst wieder auf.

HINWEIS Eine Leerschleife wie

```
For z=1 To 1000000000 ' 1 Milliarde Schleifendurchläufe
    'Nichts machen
Next z
```

verbraucht hingegen genau so viel Prozessorlast, als wenn Sie innerhalb dieser Schleife einen komplizierten Algorithmus laufen ließen, um eine neue größte Primzahl zu beweisen. Wenn Sie Prozessorlast simulieren wollen, machen Sie das am besten mit `Thread.SpinWait`, da der Thread dann auch wirklich ausgelastet ist (in etwa, als wenn Sie eine lang laufende `For`-Schleife zählen lassen würden).

Wenn Sie dieses einfachste Beispiel laufen lassen, bekommen Sie in etwa folgendes Ergebnis. Warum nur in etwa? Weil das an der Ausstattung Ihres Computers liegt – je nachdem, wie viele Prozessorkerne dem Programm zur Verfügung stehen:

```
1
6
11
16
2
3
7
12
17
5
4
8
13
18
9
14
19
10
15
20
```

Sie denken es sich sicherlich schon: `Parallel.For` arbeitet prinzipiell wie eine herkömmliche `For … Next`-Schleife, nur dass eben der innere Block parallelisiert wird.

Der so genannte *Scheduler* der Task Parallel Library übernimmt dabei die Aufgabe, zu entscheiden, für welchen Zählwert der `For`-Methode der Delegat wann aufgerufen wird. In der Regel wird die Schleife in Blöcke zerlegt, und jeder Kern ist dann dafür zuständig, dass einer der Schleifenblöcke durchlaufen wird. Bei einer Schleife von 0-99 zählt also Kern 1 von 0-24, Kern 2 von 25-49, Kern 3 von 50-74 und Kern 4 von 75-99. Doch das ist nur die vereinfachte Darstellung zum besseren Verstehen, denn der Scheduler ist viel leistungsfähiger, als die Aufgabenverteilung so starr vorzunehmen. Tatsächlich ist die Verteilung der Schleifen auf die Kerne von mehreren Faktoren abhängig. Wenn ein Teil einer Schleife länger benötigt, wird dynamisch neu verteilt.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 29\SimpleParallelFor
```

Mit einer kleinen Abwandlung unseres vorherigen Programms können wir eine ungefähre Vorstellung davon bekommen, wie eine solche Verteilung ausschauen kann. Sie sehen am Ergebnis, welcher Thread welchen Part der Zählung übernommen hat und wie lange ein Thread insgesamt am Arbeiten war.

```
Imports System.Threading
Imports System.Threading.Tasks

Module Module1

    Private myThreadCount As Integer
    Private myList As System.Collections.Concurrent.ConcurrentBag(Of ItemLogEntry)
```

```
    Sub Main()
        myList = New System.Collections.Concurrent.ConcurrentBag(Of ItemLogEntry)
        Dim sw = Stopwatch.StartNew

        Parallel.For(1, 50, Sub(i)
                                myList.Add(New ItemLogEntry With {.TickTime = sw.ElapsedMilliseconds,
                                                                  .value = i,
                                                                  .threadid = Thread.CurrentThread.ManagedThreadId})
                                'So tun, als würde was lange dauern, aber die
                                'großen Zahlen dauern sehr viel länger:
                                Console.Write(".")
                                Thread.SpinWait(i * 200000)
                            End Sub)
        Console.WriteLine()
        Console.WriteLine("Nach Thread-ID sortiert:")
        Dim sortedById = From item In myList
                         Order By item.ThreadId, item.TickTime

        sortedById.ToList.ForEach(Sub(item)
                                      Console.WriteLine(item.ToString)
                                  End Sub)
        Console.WriteLine("Return")
        Console.ReadLine()

        Console.WriteLine()
        Console.WriteLine("Nach Dauer sortiert:")
        Dim sortedByTime = From item In myList
                           Order By item.TickTime

        sortedByTime.ToList.ForEach(Sub(item)
                                        Console.WriteLine(item.ToString)
                                    End Sub)
        Console.WriteLine("Return")
        Console.ReadLine()

    End Sub

    Public Structure ItemLogEntry
        Public TickTime As Long
        Public ThreadId As Long
        Public Value As Integer

        Public Overrides Function ToString() As String
            Return String.Format("ThreadId {0:0000}: @{1:000000} ms - Value:{2:000000}",
                                 ThreadId, TickTime, Value)
        End Function
    End Structure
End Module
```

Dieses Programm erstellt eine Liste mit der Dauer von Schleifendurchläufen, in der eine länger andauernde Operation mit der Anweisung `Thread.SpinWait` simuliert wird. Die Dauer eines Schleifendurchlaufs verläuft dabei nicht linear, sondern dauert länger, je höher der Zählwert wird. Am Ende gibt das Programm die Messwerte einmal nach Thread und einmal nach dem Dauer bzw. dem Messzeitpunkt sortiert aus. Interessant ist dabei die zweite Liste, die erscheint, wenn Sie einmal [↵] betätigen. Dann nämlich sehen Sie im Konsolenfenster nicht nur, dass auf Mehrprozessorsystemen bestimmte Schleifendurchläufe wirklich gleichzeitig erfolgen (siehe 67 ms), sondern auch, dass die Threads dynamisch, je nach Last und abhängig davon, wann diese wieder frei werden, vergeben werden. Sie sehen aber auch, dass nicht beliebig viele Threads erstellt werden, sondern dass sich die Anzahl der Threads nach der Anzahl der Prozessorkerne im System richtet.

```
Nach Dauer sortiert:
ThreadId 0009: @000004 ms - Value:000001
ThreadId 0006: @000030 ms - Value:000013
ThreadId 0011: @000060 ms - Value:000025
ThreadId 0012: @000067 ms - Value:000037
ThreadId 0010: @000067 ms - Value:000049
ThreadId 0009: @000074 ms - Value:000002
ThreadId 0006: @000133 ms - Value:000014
ThreadId 0009: @000152 ms - Value:000003
ThreadId 0011: @000204 ms - Value:000026
ThreadId 0006: @000206 ms - Value:000015
ThreadId 0012: @000211 ms - Value:000038
ThreadId 0006: @000263 ms - Value:000016
ThreadId 0009: @000285 ms - Value:000004
ThreadId 0011: @000298 ms - Value:000027
ThreadId 0009: @000300 ms - Value:000005
ThreadId 0010: @000301 ms - Value:000008
ThreadId 0009: @000318 ms - Value:000006
ThreadId 0006: @000338 ms - Value:000017
ThreadId 0009: @000341 ms - Value:000007
ThreadId 0010: @000363 ms - Value:000009
ThreadId 0009: @000367 ms - Value:000010
ThreadId 0010: @000396 ms - Value:000020
ThreadId 0011: @000398 ms - Value:000028
ThreadId 0009: @000406 ms - Value:000011
ThreadId 0006: @000407 ms - Value:000018
ThreadId 0009: @000445 ms - Value:000012
ThreadId 0011: @000501 ms - Value:000029
ThreadId 0006: @000504 ms - Value:000019
ThreadId 0010: @000515 ms - Value:000021
ThreadId 0012: @000543 ms - Value:000039
ThreadId 0009: @000571 ms - Value:000024
ThreadId 0006: @000576 ms - Value:000032
ThreadId 0011: @000609 ms - Value:000030
ThreadId 0010: @000623 ms - Value:000022
ThreadId 0009: @000664 ms - Value:000040
ThreadId 0010: @000702 ms - Value:000023
ThreadId 0011: @000720 ms - Value:000031
ThreadId 0009: @000816 ms - Value:000041
ThreadId 0006: @000865 ms - Value:000033
ThreadId 0009: @000966 ms - Value:000042
ThreadId 0006: @000989 ms - Value:000034
ThreadId 0006: @001112 ms - Value:000035
ThreadId 0009: @001114 ms - Value:000043
ThreadId 0006: @001238 ms - Value:000036
ThreadId 0009: @001263 ms - Value:000044
ThreadId 0009: @001407 ms - Value:000045
ThreadId 0009: @001555 ms - Value:000046
ThreadId 0009: @001715 ms - Value:000047
ThreadId 0009: @001869 ms - Value:000048
Return
```

Diese Liste zeigt aber noch eine ganz wesentliche Charakteristik von parallelisierten Schleifen: Sie können `Parallel.For` nicht in Szenarien verwenden, bei denen die Aufgabe eines Schleifendurchgangs zu 100% vom Ergebnis des genau vorherigen Schleifendurchgangs abhängig ist.

HINWEIS `Parallel.For` zählt nur nach oben und nur in 1er-Schritten. Der erste Parameter gibt den Startwert an, der zweite den Endwert+1 (also, aufgepasst, anders als bei `For … Next` *exklusiv*!). Der dritte Parameter ist ein `Action(Of Type)`-Delegat und der Wert, der ihm übergeben wird, ist die Zählvariable. Für jeden Schleifendurchlauf wird also – ähnlich wie Sie bei `List.ForEach` in Kapitel 20 schon erfahren haben – der entsprechende `Action(Of Type)`-Delegat aufgerufen, und jedes Mal ist der Wert des Parameters, der übergeben wird, ein anderer.

Beispiel: Parallelisieren des Image-Resizers aus Kapitel 6

Versuchen wir uns an dem ersten Beispiel. Sie erinnern sich an den Image-Resizer aus Kapitel 6? Der hat viel Potenzial, auf mehreren Kernen zu laufen. Lassen Sie uns einen Blick auf den essenziellen Code werfen:

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 29\ImageResizer
```

```
For y = 0 To mySmallWbm.WriteableBitmap.PixelHeight – 1
    For x = 0 To myReduceToX - 1
        mySmallWbm.SetPixel(x, y,
            currentPicture.GetPixel(CInt(Math.Truncate(pixelFakt * x)),
                      CInt(Math.Truncate(pixelFakt * y))))
    Next
Next
```

Diesen Algorithmus, der ja die eigentliche Größenänderung vornimmt, zu parallelisieren, könnte auf jeden Fall Sinn machen. Natürlich muss das, was mit `Parallel.For` parallelisiert werden soll, auch ausreichend lange dauern, damit das Einrichten der Threads insgesamt nicht länger dauert als das, was die Threads eigentlich parallel nebeneinander machen sollen. Insofern würde es nicht sinnvoll sein, die innere der beiden Schleifen zu parallelisieren – denn soll, um bei diesem Beispiel zu bleiben, ein Bild von sagen wir 12 auf 4 Megapixel verkleinert werden, dann würden wir für jeden Zielpixel einen eigenen Thread des Threadpools bemühen und nicht weniger als 4 Millionen Threads starten. Es zeilenweise zu tun, ist aber schon was anderes. Ein 4:3-Bild wird beispielsweise auf rund 1000 Zeilen verkleinert – das könnte unter Umständen schon eher was bringen. Versuchen wir es:

```
Parallel.For(0, pixelHeight,
             Sub(y)
                 For x = 0 To myReduceToX - 1
                     mySmallWbm.SetPixel(x, y,
                         currentPicture.GetPixel(CInt(Math.Truncate(pixelFakt * x)),
                                                 CInt(Math.Truncate(pixelFakt * y))))
                 Next
             End Sub)
```

HINWEIS Wenn Sie damit auf Ihrem eigenen Computer selbst experimentieren wollen: Sie finden am Anfang der Ereignisbehandlungsroutine für die *Start!*-Schaltfläche folgenden Kommentar:

```
    Private Sub StartButton_Click(ByVal sender As System.Object,
                                  ByVal e As System.Windows.RoutedEventArgs) Handles StartButton.Click

        'Start_DedicatedTaskParallelFor()
        'Return
```

Nehmen Sie hier die Auskommentierung zurück, um die `Parallel.For`-Version zu starten.

TIPP Zum Einsatz von `Math.Truncate` in Ergänzung zu `CInt` lesen Sie bitte auch den entsprechenden Abschnitt »Besondere Funktionen, die für alle numerischen Datentypen gelten« und dort den Unterabschnitt »Performance- und Rundungs-Problematik beim Konvertieren von Fließkomma- zu Integer-Werten « in Kapitel 7.

Und in der Tat: Bei 47 Bildern, die von 12 Megapixel auf 3 Megapixel verkleinert werden, ist die Ausbeute bereits sehr beachtlich: 66 im Gegensatz zu 85 Sekunden auf einem Dual Core-Notebook mit 2,4 GHz ist eine Geschwindigkeitssteigerung von immerhin ca. 40%!

Prozessor/Cores	Ohne Parallelisierung	`Parallel.For`	`Parallel.ForEach` und `ParallelFor`
Atom 330 / 4 Cores/ 2 × HT 47 Bilder mit 4.000 × 3.000 Pixel auf 1.280 Pixel horizontal verkleinert	85 Sekunden	66 Sekunden	58 Sekunden

Parallel.ForEach

`Parallel.ForEach` funktioniert eigentlich ganz ähnlich wie `Parallel.For`, nur dass hier für jedes Element einer Auflistung ein Delegat aufgerufen wird. Auch hier versucht der Scheduler der Task Parallel Library in Abhängigkeit von den vorhandenen Prozessoren möglichst viele Schleifendurchläufe gleichzeitig auszuführen. Das folgende Beispiel ist das abgewandelte Beispiel von `Parallel.For` und demonstriert den grundsätzlichen Umgang mit `Parallel.ForEach`.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 29\SimpleParallelForEach
```

```
Imports System.Threading
Imports System.Threading.Tasks

Module Module1

    Private myThreadCount As Integer
    Private myList As System.Collections.Concurrent.ConcurrentBag(Of ItemLogEntry(Of String))
```

```
    Sub Main()
        Dim Elements As New List(Of String) From {"Peter", "Ramona", "Lisa", "Sarika", "Jürgen",
                                                  "Ruprecht", "Marcus", "Andreas", "Gabriele",
                                                  "Silja", "Uta", "Oli", "Cate", "Arnold"}

        myList = New System.Collections.Concurrent.ConcurrentBag(Of ItemLogEntry(Of String))
        Dim sw = Stopwatch.StartNew
        Dim iterator As Integer
        Parallel.ForEach(Elements,
                         Sub(element As String)
                             iterator += 1
                     myList.Add(New ItemLogEntry(Of String)() With {.TickTime = sw.ElapsedMilliseconds,
                                                                    .value = element,
                                                                    .threadid = Thread.CurrentThread.ManagedThreadId})

                             'So tun, als würde was lange dauern, und je
                             'mehr Elemente da sind, desto länger dauert es.
                             Console.Write(".")
                             Thread.SpinWait(iterator * 2000000)
                         End Sub)

        Console.WriteLine()
        Console.WriteLine("Nach Thread-ID sortiert:")
        Dim sortedById = From item In myList
                     Order By item.ThreadId, item.TickTime

        sortedById.ToList.ForEach(Sub(item)
                                      Console.WriteLine(item.ToString)
                                  End Sub)
        Console.WriteLine("Return")
        Console.ReadLine()

        Console.WriteLine()
        Console.WriteLine("Nach Dauer sortiert:")
        Dim sortedByTime = From item In myList
                     Order By item.TickTime

        sortedByTime.ToList.ForEach(Sub(item)
                                        Console.WriteLine(item.ToString)
                                    End Sub)
        Console.WriteLine("Return")
        Console.ReadLine()

    End Sub

    Public Structure ItemLogEntry(Of vType)
        Public TickTime As Long
        Public ThreadId As Long
        Public Value As vType

        Public Overrides Function ToString() As String
            Return String.Format("ThreadId {0:0000}: @{1:000000} ms - Value:{2}",
                                 ThreadId, TickTime, Value)
        End Function
    End Structure
End Module
```

Das Starten dieser Beispielanwendung führt in etwa zu folgendem Ergebnis:

```
..............
Nach Thread-ID sortiert:
ThreadId 0009: @000118 ms - Value:Peter
ThreadId 0009: @000385 ms - Value:Ramona
ThreadId 0009: @000714 ms - Value:Lisa
ThreadId 0010: @000119 ms - Value:Sarika
ThreadId 0010: @000285 ms - Value:Jürgen
ThreadId 0010: @000471 ms - Value:Ruprecht
ThreadId 0010: @000915 ms - Value:Arnold
ThreadId 0011: @000132 ms - Value:Marcus
ThreadId 0011: @000303 ms - Value:Andreas
ThreadId 0011: @000647 ms - Value:Gabriele
ThreadId 0012: @000133 ms - Value:Silja
ThreadId 0012: @000329 ms - Value:Uta
ThreadId 0012: @000857 ms - Value:Oli
ThreadId 0013: @000343 ms - Value:Cate
Return


Nach Dauer sortiert:
ThreadId 0009: @000118 ms - Value:Peter
ThreadId 0010: @000119 ms - Value:Sarika
ThreadId 0011: @000132 ms - Value:Marcus
ThreadId 0012: @000132 ms - Value:Silja
ThreadId 0010: @000285 ms - Value:Jürgen
ThreadId 0011: @000303 ms - Value:Andreas
ThreadId 0012: @000329 ms - Value:Uta
ThreadId 0013: @000343 ms - Value:Cate
ThreadId 0009: @000385 ms - Value:Ramona
ThreadId 0010: @000471 ms - Value:Ruprecht
ThreadId 0011: @000647 ms - Value:Gabriele
ThreadId 0009: @000714 ms - Value:Lisa
ThreadId 0012: @000857 ms - Value:Oli
ThreadId 0010: @000915 ms - Value:Arnold
Return
```

Auch dabei lässt sich erkennen, dass die Verarbeitung von Elementen wirklich gleichzeitig erfolgt, und genauso zu beobachten ist aber auch, dass die Vorhersagbarkeit, wann letzten Endes welche Element »drankommt«, nicht gegeben ist.

Und für `Parallel.ForEach` gilt dasselbe wie für `Parallel.For`: Sie können `Parallel.ForEach` nicht in Szenarien verwenden, bei denen die Aufgabe eines Schleifendurchgangs zu 100% vom Ergebnis des genau vorherigen Schleifendurchgangs abhängig ist.

CancellationToken – Exit For für Parallel.For und Parallel.ForEach

Sowohl `Parallel.For` als auch `Parallel.ForEach` sind Methoden, die Prozesse auf anderen Threads in Gang bringen und damit ab einem gewissen Zeitpunkt keinen Einfluss mehr darauf haben, was innerhalb dieser Thread-Methoden passiert. Damit ist es klar, dass das vorzeitige Beenden einer Schleife keine triviale Sache ist. Ein üblicher Fehler bei den ersten Gehversuchen mit `Parallel.For` ist es dann auch, mit `Return` eine Schleife vorzeitig zu beenden. Doch `Return` springt ja lediglich aus der Methode, die für jeden Schleifendurchlauf aufgerufen wird, und damit entspricht `Return` der `Continue For`-Methode, nicht aber `Exit For`.

Die einzige Möglichkeit, aus einer parallelisierten `For`-Schleife rauszukommen, besteht darin, dass innerhalb der parallelisierten Schleife (egal, ob es sich dabei um `For` oder `ForEach` handelt) auf eine ganz definierte Art und Weise eine Ausnahme ausgelöst wird. Und das passiert folgendermaßen (und gilt im Übrigen auch gleichermaßen für das vorzeitige Abbrechen eines Tasks):

- Sie verschieben den kompletten `Parallel.For`- bzw. `Parallel.ForEach`-Block in eine `Try … Catch`-Struktur, um eine Ausnahme darin abfangen zu können.
- Sie erstellen ein so genanntes `CancellationTokenSource`-Objekt. Dieses Objekt dient einerseits dazu, später die Operation abzubrechen, und andererseits stellt es der Schleife das Token (eine Art Erkennungsmarke) zur Verfügung, anhand dessen sie erkennen kann, dass die Schleife abgebrochen werden soll.
- Sie übergeben der parallelisierten Schleife dieses Token als Parameter.
- Wenn Sie die Schleife abbrechen wollen – dabei ist es ganz egal, ob das von innen oder von außen erfolgt –, rufen Sie auf dem `CancellationTokenSource`-Objekt die Methode `Cancel` auf. Innerhalb des Delegaten (oftmals das Lambda), den Sie `For` bzw. `ForEach` übergeben haben, wird eine Ausnahme ausgelöst, die bis zum umschließenden `Try … Catch`-Block »nach oben bubbelt« (»hochblubbert« könnte man auf Deutsch sagen) – und an dieser Stelle ist die `For`-Schleife dann beendet.

Das Ganze sieht dann – in Abwandlung zu unserem `Parallel.For`-Beispiel – folgendermaßen aus:

HINWEIS Sie finden die folgenden Beispiele ebenfalls in den zuvor vorgestellten Beispielanwendungen unterhalb der Methoden `CancelableVersion`. Kommentieren Sie in der `Sub Main` einfach die ersten beiden Zeilen aus, um stattdessen diese Methode laufen zu lassen.

```
Sub CancelableVersion()

    myList = New System.Collections.Concurrent.ConcurrentBag(Of ItemLogEntry)
    Dim sw = Stopwatch.StartNew

    Dim ctSource As New CancellationTokenSource()
    Dim cancelToken = ctSource.Token

    Try
        Parallel.For(1, 50, New ParallelOptions With {.CancellationToken = cancelToken},
                     Sub(i)
                         myList.Add(New ItemLogEntry With {.TickTime = sw.ElapsedMilliseconds,
                                                           .value = i,
                                                           .threadid = Thread.CurrentThread.ManagedThreadId})
                         'So tun, als würde was lange dauern, aber die
                         'großen Zahlen dauern sehr viel länger:
                         Console.Write(".")
                         Thread.SpinWait(i * 200000)

                         'bei 25 ist Feierabend!
                         If i = 25 Then
                             ctSource.Cancel()
                         End If
                     End Sub)
    Catch ex As Exception
        Console.WriteLine("Exited For!")
    End Try
```

```
        Console.WriteLine()
        Console.WriteLine("Nach Thread-ID sortiert:")
        Dim sortedById = From item In myList
.
.
.
    End Sub
```

Wenn Sie diese Anwendung laufen lassen, produziert sie folgendes Ergebnis auf dem Bildschirm:

```
.......Exited For!

Nach Thread-ID sortiert:
ThreadId 0008: @000012 ms - Value:000001
ThreadId 0008: @000068 ms - Value:000002
ThreadId 0009: @000012 ms - Value:000013
ThreadId 0009: @000124 ms - Value:000014
ThreadId 0010: @000031 ms - Value:000025
ThreadId 0011: @000031 ms - Value:000037
ThreadId 0012: @000061 ms - Value:000049
Return


Nach Dauer sortiert:
ThreadId 0008: @000012 ms - Value:000001
ThreadId 0009: @000012 ms - Value:000013
ThreadId 0011: @000031 ms - Value:000037
ThreadId 0010: @000031 ms - Value:000025
ThreadId 0012: @000061 ms - Value:000049
ThreadId 0008: @000068 ms - Value:000002
ThreadId 0009: @000124 ms - Value:000014
Return
```

Dabei wird auch hier wieder eine Problematik bei der Schleifenparallelisierung deutlich: Dass das 25. Element das Ausstiegskriterium ist, bedeutet nicht, dass alle 24 Elemente vorher schon bearbeitet wurden! Diese Tatsache müssen Sie unbedingt im Hinterkopf behalten, und im Abschnitt »Vermeiden von Fehlern beim Parallelisieren von Schleifen« ab Seite 885 werden wir auch auf dieses Problem noch mal eingehen und aufzeigen, wie sich etwas derartiges lösen lässt.

Für `Parallel.ForEach` gilt die Vorgehensweise gleichermaßen – im Beispiel, das Sie im Folgenden sehen, sind auch hier die Änderungen im Vergleich zur nicht abbrechbaren Version wieder hervorgehoben:

```
    Sub CancelableVersion()
        Dim Elements As New List(Of String) From {"Peter", "Ramona", "Lisa", "Sarika", "Jürgen",
                                                  "Ruprecht", "Marcus", "Andreas", "Gabriele",
                                                  "Silja", "Uta", "Oli", "Cate", "Arnold"}

        myList = New System.Collections.Concurrent.ConcurrentBag(Of ItemLogEntry(Of String))
        Dim sw = Stopwatch.StartNew
        Dim iterator As Integer
```

```
        Dim ctSource As New CancellationTokenSource()
        Dim cancelToken = ctSource.Token

        Try
            Parallel.ForEach(Elements, New ParallelOptions With {.CancellationToken = cancelToken},
                    Sub(element As String)
                        iterator += 1
                      myList.Add(New ItemLogEntry(Of String)() With {.TickTime = sw.ElapsedMilliseconds,
                                                      .value = element,
                                                      .threadid = Thread.CurrentThread.ManagedThreadId})

                        If element = "Jürgen" Then
                            ctSource.Cancel()
                        End If

                        'So tun, als würde was lange dauern, und je
                        'mehr Elemente da sind, desto länger dauert es.
                        Console.Write(".")
                        Thread.SpinWait(iterator * 2000000)
                    End Sub)
        Catch ex As OperationCanceledException
            Console.WriteLine("Exited For!")
        End Try

        Console.WriteLine()
        Console.WriteLine("Nach Thread-ID sortiert:")
        Dim sortedById = From item In myList
                     Order By item.ThreadId, item.TickTime
.
.
.
    End Sub
```

Das Ergebnis lautet hier:

```
Nach Dauer sortiert:
ThreadId 0010: @000009 ms - Value:Peter
ThreadId 0007: @000009 ms - Value:Sarika
ThreadId 0011: @000024 ms - Value:Marcus
ThreadId 0013: @000033 ms - Value:Silja
ThreadId 0012: @000033 ms - Value:Cate
ThreadId 0013: @000237 ms - Value:Uta
ThreadId 0012: @000274 ms - Value:Arnold
ThreadId 0010: @000312 ms - Value:Ramona
ThreadId 0011: @000391 ms - Value:Andreas
ThreadId 0007: @000418 ms - Value:Jürgen
Return
```

Und auch hier wird deutlich, dass die Ausgangsreihenfolge der Liste nichts mit dem Resultat zu tun hat: Obwohl das Listenelement für die Abbruchbedingung – in diesem Beispiel der Name »Jürgen« – die Position 5 in der Liste hat, werden insgesamt 10 Listenelemente verarbeitet (gleichzeitig, auf anderen Threads nämlich), bevor alle Schleifendurchläufe tatsächlich zum Anhalten kommen.

Vermeiden von Fehlern beim Parallelisieren von Schleifen

Beim Parallelisieren von Schleifen können schnell Fehler entstehen, die dazu führen, dass die Programme nicht nur nicht schneller, sondern gar nicht mehr vernünftig laufen oder sich so verhalten, dass sie auf einmal weit weg von der Problemlösung sind. Einige der Punkte, die dazu führen können, haben Sie bereits kennen gelernt:

- Die Schleifen werden nicht mehr sequenziell ausgeführt, sondern eben parallel. Sie müssen also beispielsweise für `Parallel.For` immer damit rechnen, dass die Abarbeitung eines Schleifenkörpers mit einer kleineren Zählvariable *nicht* vor dem Durchlauf mit einer größeren Zählvariable erfolgt ist. Wann und mit welchem Wert ein Schleifenkörper durchlaufen wird, ist nicht deterministisch. Das gilt für `Parallel.ForEach` und die Abfolge der Elemente der Auflistung, durch die iteriert wird, gleichermaßen. Sie wissen nie, welches Element der Auflistung wann verarbeitet wird, welche schon verarbeitet wurden und welche »noch kommen«.
- Der parallelisierte Schleifenkörper läuft gleichzeitig in mehreren »Ausgaben«, aber anderen Werten als unterschiedliche Tasks und damit auf unterschiedlichen Threads. Threads können aber niemals direkt Steuerelemente aktualisieren. Sie müssen einen Delegaten sozusagen auf den UI-Thread schicken, der dann auf dem UI-Thread ausgeführt wird. Wie das funktioniert, wurde bereits an früherer Stelle beschrieben (Abschnitt »Zugriff auf Steuerelemente von Threads« ab Seite 868).
- Gerade wenn Sie parallelisierte Schleifen verwenden, die die Benutzeroberfläche aktualisieren müssen, kann es schnell zu Deadlocks kommen, bei denen ein Thread darauf wartet, dass der UI-Thread einen Delegaten ausführt, während der UI-Thread wartet, dass alle laufenden Threads beendet werden. Wie Sie diesen Zustand vermeiden, erfahren Sie im Abschnitt »Verhindern, dass die Benutzeroberfläche beim Warten auf Tasks blockiert« ab Seite 895.
- Manchmal ist es notwendig, die maximale Anzahl an Parallelisierungen zu beschränken. Wenn zum Beispiel parallele Schleifen verschachtelt werden, dann initiiert der Scheduler der Task Parallel Library von .NET 4.0 für `Parallel.For` oder `Parallel.ForEach` unter Umständen wesentlich mehr Tasks, als es Cores gibt. Wenn diese dann alle Volllast laufen, scheint der Computer entweder zu stehen, oder es gibt ein Speicherproblem, wenn innerhalb des Schleifenblocks große Mengen an Speicher (beispielsweise bei der Bildverarbeitung reserviert werden), der sich dann entsprechend der Parallelisierung multipliziert.
- Vermeiden Sie es, in Lambdas direkt auf Zählvariablen bzw. Iterationselemente von Schleifenkörpern zuzugreifen. Das ist gerade in verschachtelten Schleifen sehr problematisch. Der Visual Basic-Compiler gibt hier auch eine entsprechende Warnung aus.

Ein Beispiel soll das verdeutlichen. Schleifenkonstruktionen, die weitere Iterationsvariablen von den eigentlichen Schleifenvariablen abhängig machen, kommen in der Praxis häufig vor. Wenn Sie die folgenden Zeilen ausgeben lassen …

```
Const SPINVALUE = 20000000

Dim innerIndex As Integer
Dim outerIndex As Integer

For i = 0 To 3
    innerIndex = 0
    For j = 0 To 3
```

```
                Thread.SpinWait(SPINVALUE)
                Console.WriteLine("iIndex:{0} ;oIndex:{1}",
                          innerIndex, outerIndex)
                innerIndex += 1
            Next
            outerIndex += 1
        Next
```

… dann ergibt sich folgendes Bild im Konsolenfenster:

```
iIndex:0 ;oIndex:0
iIndex:1 ;oIndex:0
iIndex:2 ;oIndex:0
iIndex:3 ;oIndex:0
iIndex:0 ;oIndex:1
iIndex:1 ;oIndex:1
iIndex:2 ;oIndex:1
iIndex:3 ;oIndex:1
iIndex:0 ;oIndex:2
iIndex:1 ;oIndex:2
iIndex:2 ;oIndex:2
iIndex:3 ;oIndex:2
iIndex:0 ;oIndex:3
iIndex:1 ;oIndex:3
iIndex:2 ;oIndex:3
iIndex:3 ;oIndex:3
```

Die gleiche Version parallelisiert mit `Parallel.For`:

```
        innerIndex = 0
        outerIndex = 0

        Parallel.For(0, 4,
                     Sub(i)
                         innerIndex = 0
                         Parallel.For(0, 4,
                                      Sub(j)
                                          Thread.SpinWait(SPINVALUE)
                                          Console.WriteLine("iIndex:{0} ;oIndex:{1}",
                                                    innerIndex, outerIndex)
                                          innerIndex += 1
                                      End Sub)
                         outerIndex += 1
                     End Sub)
        Console.ReadKey()
```

ergibt folgende Ausgabe:

```
iIndex:0 ;oIndex:0
iIndex:1 ;oIndex:0
iIndex:2 ;oIndex:0
iIndex:3 ;oIndex:0
iIndex:4 ;oIndex:0
iIndex:5 ;oIndex:0
iIndex:6 ;oIndex:0
iIndex:0 ;oIndex:0
iIndex:1 ;oIndex:1
```

```
iIndex:2 ;oIndex:1
iIndex:3 ;oIndex:1
iIndex:4 ;oIndex:2
iIndex:5 ;oIndex:2
iIndex:6 ;oIndex:2
iIndex:7 ;oIndex:3
iIndex:8 ;oIndex:3
```

Das Problem dabei ist, dass sich die Iterationsvariablen verändern können (da sie von einem anderen Thread weiterverarbeitet werden), während der Task abläuft, der auf sie zurückgreift. Verwenden Sie also innerhalb von parallelisierten `For`-Schleifen ausschließlich die Zählvariablen der Schleifen selbst, und parallelisieren Sie, wenn die Notwendigkeit besteht, bestimmte Methoden durch neue `Tasks` selbst, und übergeben Sie diesen Iterationsvariablen als Parameter.

Dieses letzte Problem lässt sich also schon nicht mehr durch einfache Maßnahmen in den Griff bekommen – man muss sich gut überlegen, wie man vorgehen kann, um eine parallelisierte Version eines Algorithmus zu schaffen und immer sehr viele Tests und Kontrollen durchführen, um das gewünschte Ergebnis nicht zu verfälschen.

Das gilt auch für die Parallelisierung unseres Primzahlenbeispiels, das die letzten *x* Primzahlen unterhalb eines bestimmten Schwellwerts ermittelt. Hier ergeben sich folgende Probleme:

- Wir zählen im Primzahlenalgorithmus rückwärts – mit `Parallel.For` geht das nicht. Wir können den Bereich auch nicht einfach umdrehen und quasi die Zahlen in der Reihenfolge `MaxCount-currentValue` verarbeiten, da die Verteilung auf die einzelnen, parallel laufenden Schleifenbereiche, was den Wertebereich anbelangt, zu groß sein würde. Wenn wir zum Beispiel den Bereich von 0-99 parallelisieren, würde der erste Wertebereich (das Umdrehen des Werts angenommen) von 99-50, der zweite von 49-0 zählen. Wir wollen aber, dass der erste parallelisierte Block 99 berechnet, der zweite 98, der dritte 97 usw. Jeder neue Thread soll also die nächste Zahl in der Reihe, nicht den nächsten aufgeteilten Block erhalten, denn wir haben nichts davon, wenn wir die 30 Primzahlen direkt unter 100.000 berechnen, wenn einer der Threads die Primzahlen im Block 20.000 bis 30.000 berechnet, denn diese sind sicherlich nicht in den 30 Zahlen unter 100.000 enthalten.
- Wir können die ermittelten Werte nicht einfach in die Ergebnis-ListBox schreiben, da sie nicht zwangsläufig in der richtigen Reihenfolge eintreffen werden. Es kann sein, dass wir mit der zweitletzten Primzahl unterhalb unseres Grenzwerts eher fertig sind und sie als prim eingeordnet haben als mit der drittletzten. Dann aber würde die Reihenfolge der Primzahlen falsch sein; deswegen ist die automatische Sortierung der ListBox eingeschaltet – aber das verschenkt natürlich zusätzliche Rechenleistung, auch wenn es nicht viel ist. Dazu kommt: Die ListBox sortiert alphabetisch. Auch hier müssen wir wieder den Spagat mit den führenden Nullen durchführen, oder eben selbst dafür sorgen, die Zahlenliste in einer sortierten Liste zu halten und dann die gesamte Liste bei jedem neuen Eintreffen einer Primzahl sortiert in die ListBox zu schreiben.
- Wir können nicht sicher sein, dass wir nicht über das Ziel hinausschießen. Angenommen, wir wollen die letzten 10 Primzahlen ermitteln und testen gleichzeitig die Kandidaten, die Nummer 10, 11 und 12 werden könnten. Wenn wir nun mit 11 und 12 eher fertig sind als mit Nummer 10, dann müssen wir dennoch 10 zu Ende testen und haben 11 und 12, weil die ja schon ermittelt waren, quasi umsonst berechnet. Dagegen ist kein Kraut gewachsen, und diesen zusätzlichen Aufwand müssen wir einkalkulieren, weil wir insgesamt dennoch schneller sein werden.
- Wir müssen die maximale Parallelisierung beschränken, um nicht schon den 8. oder 9. Primzahlentest gleichzeitig anzustoßen, wenn nur noch 2 Zahlen fehlen.

BEGLEITDATEIEN Die Begleitdateien zum Beispiel, dass diese Punkte beherzigt, finden Sie im Verzeichnis

```
...\VB 2010 Entwicklerbuch\Kapitel 29\PrimesParallelFor
```

Leider können wir aus Platzgründen an dieser Stelle das Listing nicht abdrucken – um die Details zu erfahren, laden Sie das Beispiel in Visual Studio und lesen Sie die entsprechenden Kommentare. Gönnen Sie sich aber in jedem Fall einen Blick auf Abbildung 29.7 und vergleichen Sie sie mit Abbildung 29.3 auf Seite 860.

Abbildung 29.7 Dank der Task Parallel Library von .NET Framework 4.0 nutzen Sie die zur Verfügung stehende Leistung eines modernen Multicore-/Multiprozessorsystems voll aus. Geschwindigkeitssteigerungen bestimmter Algorithmen um den Faktor 5-10 sind so keine Ausnahme – in diesem Fall waren es 13,5 Sekunden im Vergleich zu 58 Sekunden zur Single-Threading-Version, immerhin fast Faktor 4,5.

Arbeiten mit Tasks

Zu Beginn dieses Kapitels haben Sie ja schon die vielen Möglichkeiten kennen gelernt, Methoden auf eigenen Threads zu initiieren und laufen zu lassen.

Tasks, wie gesagt, stehen erst ab .NET Framework 4.0 und auch nicht in .NET Compact Framework zur Verfügung. Ein Task entspricht einem Thread des Threadpools, und mit seiner Hilfe lässt sich auf einfachste Weise eine Methode auf einem eigenen Thread ausführen. Dabei haben Sie die Möglichkeit, entweder die statische `TaskFactory`-Klasse zu verwenden, die Sie ebenfalls über die `Task`-Klasse erreichen, oder einen Task wie jede andere Klasse auch in ein Objekt zu instanziieren:

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 29\SimpleTask
```

```
'Immer dran denken, diese Imports vorzunehmen,
'wenn Sie mit Task und Thread-Objekten arbeiten!
Imports System.Threading
Imports System.Threading.Tasks

Module Module1

    'Konstante, die bestimmt, wie oft in den Tasks geloopt wird.
    Private Const LOOPS As Integer = 8

    Sub Main()

        'Erstellt einen Task - startet ihn aber nicht.
        Dim firstTask As New Task(
            Sub()
                Console.WriteLine("Starting Task ID: " & Task.CurrentId.ToString)
                Console.WriteLine("Equals Thread ID: " & Thread.CurrentThread.ManagedThreadId.ToString)
                For count = 0 To LOOPS
                    Console.WriteLine(String.Format("ThreadId: {0:000} - Count equals: {1}",
                                      Thread.CurrentThread.ManagedThreadId,
                                      count))
                    'Arbeitslast simulieren
                    Thread.SpinWait(8000000)
                Next
            End Sub)

        'Erstellt einen Task über die Factory-Klasse und startet ihn.
        '(Über die Factory-Klasse geht das auch gar nicht anders.)
        Dim taskByFactory = Task.Factory.StartNew(
            Sub()
                Console.WriteLine("Starting Task ID: " & Task.CurrentId.ToString)
                Console.WriteLine("Equals Thread ID: " & Thread.CurrentThread.ManagedThreadId.ToString)
                For count = 0 To LOOPS
                    Console.WriteLine(String.Format("ThreadId: {0:000} - Count equals: {1}",
                                      Thread.CurrentThread.ManagedThreadId,
                                      count))
                    'Arbeitslast simulieren
                    Thread.SpinWait(10000000)
                Next
            End Sub)

        'Der erste Task wird jetzt erst gestartet:
        firstTask.Start()

        'Wenn Sie eine Taste drücken, während die anderen
        'Tasks noch laufen, wird die Anwendung dennoch beendet.
        Console.WriteLine("Hauptthread ist fertig und wartet!")
        Console.Read()

    End Sub
End Module
```

Wenn Sie diese kleine Beispielanwendung laufen lassen, sehen Sie in etwa das folgende Ergebnis im Konsolenfenster:

```
Hauptthread ist fertig und wartet!
Starting Task ID: 1
Equals Thread ID: 11
ThreadId: 011 - Count equals: 0
Starting Task ID: 2
Equals Thread ID: 10
ThreadId: 010 - Count equals: 0
ThreadId: 011 - Count equals: 1
ThreadId: 010 - Count equals: 1
ThreadId: 011 - Count equals: 2
ThreadId: 010 - Count equals: 2
ThreadId: 011 - Count equals: 3
ThreadId: 010 - Count equals: 3
ThreadId: 011 - Count equals: 4
ThreadId: 011 - Count equals: 5
ThreadId: 010 - Count equals: 4
ThreadId: 011 - Count equals: 6
ThreadId: 010 - Count equals: 5
ThreadId: 011 - Count equals: 7
ThreadId: 010 - Count equals: 6
ThreadId: 011 - Count equals: 8
ThreadId: 010 - Count equals: 7
ThreadId: 010 - Count equals: 8
```

Was auffällt: Die Meldung *Hauptthread ist fertig und wartet!* wird als Allererstes ausgegeben. Und das ist auch richtig. Denn der Hauptthread läuft ja auch unabhängig von den beiden Lambdas weiter, die im Beispiel den beiden `Task`-Instanzen übergeben werden – und diese arbeiten dann ihre Aufgabe ab, während der Hauptthread auf eine Tasteneingabe wartet.

Würde diese übrigens kommen, dann wäre es mit der Abarbeitung der Schleifen vorbei: Tasks entsprechen Threads aus dem Threadpool und Threadpool-Threads werden immer als so genannte *Hintergrundthreads* ausgeführt (bei einem `Thread`-Objekt würden Sie dazu dessen `IsBackground`-Eigenschaft auf `True` setzen, um den gleichen Effekt zu erzielen).

Was ebenfalls auffällt bei dieser Demo, sind die weiteren Zeilen, die ausgegeben werden. Hier sieht man nämlich, dass zunächst der Task seine Arbeit aufnimmt, der durch die Factory-Klasse erstellt wurde, denn er gibt die Meldung des ersten Schleifendurchlaufs aus – und das kurz, bevor der zweite Task (der jedoch vorher erstellt, aber eben nicht gestartet wurde!) mit seiner Arbeit beginnt.

TIPP Wenn Sie einen Task mit einem mehrzeiligen Anweisungs-Lambda definieren, benötigen Sie die wenigste Tipparbeit, wenn Sie erst etwa folgende Zeile schreiben:

```
Dim someTask = Task.Factory.StartNew(Sub())
```

und dann die Einfügemarke zwischen der letzten und vorletzten Klammer dieser Codezeile platzieren und ↵ drücken.

Warten auf das Ende von Tasks – WaitOne, WaitAny und WaitAll

Das Starten eines Tasks ist – das haben wir gerade gesehen – eine vergleichsweise einfache Geschichte. Doch wenn erst einmal mehrere Tasks laufen, dann ist es natürlich sinnvoll zu wissen, wann welcher Task beendet ist. Ebenso möchte man die Möglichkeit haben, auf den ersten (und vielleicht später auf den zweiten, dritten) Task oder sogar auf das Ende aller laufenden Tasks zu warten.

Für `Task`-Instanzen gibt es zu diesem Zweck drei Methoden, mit denen Sie diesen Task (kein Kalauer beabsichtigt) vergleichsweise einfach lösen können: die Methoden `WaitOne`, `WaitAny` und `WaitAll`.

`WaitOne` unterscheidet sich von `WaitAny` und `WaitAll` insofern, dass es sich hierbei um eine Membermethode handelt, während `WaitAny` und `WaitAll` statische Methoden der `Task`-Klasse darstellen. Das heißt: `WaitOne` gilt für die `Task`-Instanz, für die Sie die Methode aufrufen: Haben Sie einen laufenden Task, und rufen Sie vom Thread, der diesen Task initiiert hat, dessen `WaitOne`-Methode auf, dann wartet dieser Thread so lange, bis der Task seine Arbeit erledigt hat.

ACHTUNG Wenn Sie `WaitOne` vom UI-Thread aus aufrufen, ist dieser natürlich ebenfalls bis zum Ende blockiert und kann seine Nachrichtenwarteschleife nicht ausführen. In bestimmten Szenarien müssen Sie Tasks kaskadieren, um ein Blockieren des UI-Threads zu verhindern.

`WaitAny` und `WaitAll` sind statische Methoden, die ein Array mit `Task`-Objekten als Parameter erwarten. `WaitAny` wartet dabei auf dem aktuellen Thread, bis einer der Tasks des Arrays beendet ist, während `WaitAll` wartet, bis *alle* Tasks im Array beendet sind.

Ein Beispiel soll das verdeutlichen. Die Abwandlung des letzten Beispiels wartet darauf, dass der Task, der seine Arbeit zuerst beendet hat, identifiziert wird. Wenn auch der zweite Task seine Arbeit erledigt hat, gibt das Beispiel eine zweite Meldung aus.

```
Sub WaitingForTasks()

    Dim firstTask As New Task(
        Sub()
        .
        .
        .
        End Sub)

    'Erstellt einen Task über die Factory-Klasse und startet ihn.
    '(Über die Factory-Klasse geht das auch gar nicht anders.)
    Dim taskByFactory = Task.Factory.StartNew(
        Sub()
        .
        .
        .
        End Sub)

    'Der erste Task wird jetzt erst gestartet:
    firstTask.Start()

    'Wenn Sie eine Taste drücken, während die anderen
    'Tasks noch laufen, wird die Anwendung dennoch beendet.
    Console.WriteLine("Hauptthread ist fertig und wartet auf Ergebnisse!")
```

```
        'Die Tasks, auf die die Anwendung warten soll, in ein Task-Array packen
        '(Lokaler Typrückschluss macht automatisch ein Array vom Typ Task daraus!)
        Dim tasksToWaitFor = {firstTask, taskByFactory}

        'Warten, bis der erste Task seine Aufgabe beendet hat:
        Dim taskFirstFinishedIndex = Task.WaitAny(tasksToWaitFor)

        'Rausfinden, welcher Task beendet wurde: Der Index steht im Rückgabewert von WaitAny.
        Console.WriteLine(If(tasksToWaitFor(taskFirstFinishedIndex) Is taskByFactory,
                             "Faktorytask wurde zuerst beendet!",
                             "Manueller Task wurde zuerst beendet!"))

        'Warten, dass alle (restlichen) Tasks beendet wurden:
        Task.WaitAll(tasksToWaitFor)
        Console.WriteLine("Alle Tasks wurden beendet.")
        Console.Read()
    End Sub
```

Das Ergebnis des geänderten Beispiels ergibt dann folgende Bildschirmausgabe:

```
Hauptthread ist fertig und wartet auf Ergebnisse!
Starting Task ID: 1
Equals Thread ID: 11
Starting Task ID: 2
Equals Thread ID: 12
ThreadId: 012 - Count equals: 0
ThreadId: 011 - Count equals: 0
ThreadId: 012 - Count equals: 1
ThreadId: 011 - Count equals: 1
ThreadId: 012 - Count equals: 2
ThreadId: 012 - Count equals: 3
ThreadId: 011 - Count equals: 2
ThreadId: 012 - Count equals: 4
ThreadId: 011 - Count equals: 3
ThreadId: 012 - Count equals: 5
ThreadId: 011 - Count equals: 4
ThreadId: 012 - Count equals: 6
ThreadId: 011 - Count equals: 5
ThreadId: 012 - Count equals: 7
ThreadId: 011 - Count equals: 6
ThreadId: 012 - Count equals: 8
Manueller Task wurde zuerst beendet!
ThreadId: 011 - Count equals: 7
ThreadId: 011 - Count equals: 8
Alle Tasks wurden beendet.
```

Task mit und ohne Rückgabeparameter

Bislang haben Sie Tasks kennen gelernt, die eine bestimmte Aufgabe erledigt, die aber keine Resultate in Form von Funktionsergebnissen zurückgeliefert haben. Das entspricht nicht unbedingt dem, was Sie in der Praxis benötigen, denn genau wie »normale« Methoden in vielen Fällen etwas berechnen oder ermitteln und damit Funktionswerte zurückliefern, gilt das für Methoden, die parallelisiert in Tasks laufen, natürlich auch. Nur: Wann und wie kommen Sie an das Funktionsergebnis eines Tasks? Und: Wie übergeben Sie einem Task Parameter?

Mit dem Einsatz von Lambdas ist die »Parameterübergabe« eine recht einfache Geschichte. Tasks, die durch Codeblöcke in Lambdas platziert sind, greifen einfach auf die lokalen Variablen zu, die sie umgeben, etwa so:

```
        Dim localVariable As String = "An diesen String"

        Dim someTask = New Task(Sub()
                                    localVariable &= " etwas dranzuhängen"
                                End Sub)

        localVariable &= " ist nicht schlecht"
        someTask.Start()
        someTask.Wait()
        Console.WriteLine(localVariable)
```

Denken Sie daran, dass das alles andere als selbstverständlich ist, da `localVariable` innerhalb des Lambdas ja nicht unbedingt in der Reihenfolge ausgeführt werden muss, den die Abfolge der Codezeilen in der Codedatei vorgaukelt. Lassen Sie dieses Beispiel laufen, spuckt es folgendes, grammatikalisch nicht so ganz perfektes Deutsch aus:

```
An diesen String ist nicht schlecht etwas dranzuhängen
```

Das frühzeitige Starten des Tasks führt hingegen zu einem komplett anderen Ergebnis:

```
        someTask.Start()
        someTask.Wait()
        localVariable &= " ist nicht schlecht"
        Console.WriteLine(localVariable)
```

obwohl die Reihenfolge des Codes, mit dem der String definiert wird, natürlich dieselbe bleibt:

```
An diesen String etwas dranzuhängen ist nicht schlecht
```

Sie sehen: Lokale Variablen in Lambdas zu verwenden, ist ein wenig »tricky«, aber sicherlich einfach und bequem. Und natürlich durchaus legitim. Aber diese Art der Parameterübergabe bringt Ihnen nichts, wenn Sie die gleiche Codebasis in mehreren Tasks verwenden wollen.

Hier haben Sie mit Tasks die Möglichkeit, einen Parameter zu definieren, den Sie auch übergeben können, wenn Sie die Methode nicht auf herkömmliche Art, sondern als Task aufrufen wollen. Das folgende Beispiel zeigt nicht nur das, sondern auch, wie eine solche Methode einen Parameter zurückgeben kann. Da ist also zunächst die Methode, die die eigentliche Arbeit als Task ausführen soll:

```
    'Methode, die als Task ausgeführt werden soll und einen Parameter entgegennimmt:
    '(Die Methode, die vorher der Statement Lambda war).
    Function WorkingProc(ByVal spinValue As Integer) As Long

        Dim sw = Stopwatch.StartNew

        Console.WriteLine("Starting Task ID: " & Task.CurrentId.ToString)
        Console.WriteLine("Equals Thread ID: " & Thread.CurrentThread.ManagedThreadId.ToString)
        For count = 0 To LOOPS
```

```
        Console.WriteLine(String.Format("ThreadId: {0:000} - Count equals: {1}",
                          Thread.CurrentThread.ManagedThreadId,
                          count))
        'Arbeitslast simulieren
        Thread.SpinWait(spinValue)
    Next

    sw.Stop()
    Return sw.ElapsedMilliseconds
End Function
```

Sie sehen: Der Methode wird ein Parameter (vom Typ `Integer`) übergeben, und diese Methode gibt die Dauer ihres Werkelns als Funktionsergebnis zurück.

Das (mehrfache) Verwenden dieser Methode funktioniert nun auf folgende Weise: Anstelle eines Lambdas übergeben Sie dem Task – entweder über den Konstruktor oder über die Factory-Klasse – den Zeiger der Methode mit dem `AddressOf`-Operator. Gleichzeitig verwenden Sie die generische Task-Klasse (`Task(Of Type)`), die als Typparameter den Typ des Funktionsergebnisses entgegennimmt. Das sieht dann in Abwandlung unseres bisherigen Beispiels folgendermaßen aus:

```
'Erstellt den ersten Task direkt, startet ihn aber noch nicht.
Dim firstTask As New Task(Of Long)(AddressOf WorkingProc, SPIN_ONE)

'Erstellt einen zweiten Task über die Factory-Klasse und startet ihn.
Dim taskByFactory = Task.Factory.StartNew(Of Long)(AddressOf WorkingProc, SPIN_TWO)

'Der erste Task wird jetzt erst gestartet:
firstTask.Start()
```

Den Parameter, den die Methode erhalten soll, übergeben Sie als zweites Argument – in unserem Fall ist der eigentliche Methodenparameter beim ersten Task die Konstante `SPIN_ONE` mit dem entsprechenden Wert für die Arbeitslastsimulation durch `Thread.SpinWait` (siehe oben), im zweiten die Konstante `SPIN_TWO`.

Natürlich kann das Starten des Tasks anders als beim direkten Methodenaufruf noch kein Funktionsergebnis hervorbringen, denn die Methode muss ja schließlich erst ihre Arbeit verrichten. Stattdessen rufen Sie das Funktionsergebnis über die `Result`-Eigenschaft der `Task`-Instanz ab. Und wenn das noch nicht zur Verfügung steht, dann hält `Result` den abfragenden Thread eben so lange an, bis die Methode innerhalb des Tasks das Ergebnis bereitstellt – die Abfrage der `Result`-Eigenschaft verhält sich also diesbezüglich genauso wie `WaitOne` des Task-Objekts. Möchten Sie das Ergebnis des ersten von mehreren Tasks abfragen, müssen Sie `Result` und `WaitAny` kombinieren, so wie es das Beispiel zeigt:

```
Console.WriteLine("Hauptthread ist fertig und wartet auf Ergebnisse!")

'Die Tasks, auf die die Anwendung warten soll, in ein Task-Array packen
'(Lokaler Typrückschluss macht automatisch ein Array vom Typ Task daraus!)
Dim tasksToWaitFor = {firstTask, taskByFactory}

'Warten, bis der erste Task seine Aufgabe beendet hat:
Dim taskFirstFinishedIndex = Task.WaitAny(tasksToWaitFor)
```

```
        'Rausfinden, welcher Task beendet wurde: Der Index steht im Rückgabewert von WaitAny.
        Console.WriteLine(If(tasksToWaitFor(taskFirstFinishedIndex) Is taskByFactory,
                             "Factory Task wurde zuerst beendet nach ",
                             "Manueller Task wurde zuerst beendet nach ") &
                        tasksToWaitFor(taskFirstFinishedIndex).Result.ToString & " ms.")

        'Warten, dass alle (restlichen) Tasks beendet wurden:
        Task.WaitAll(tasksToWaitFor)
        Console.WriteLine("Alle Tasks wurden beendet.")
        Console.Read()
```

Das Ergebnis dieses Beispiels unterscheidet sich vom vorherigen nur durch die Angabe der Dauer des Tasks, der als Erstes beendet wird:

```
...ThreadId: 012 - Count equals: 8
Manueller Task wurde zuerst beendet nach 659 ms.
ThreadId: 011 - Count equals: 8
```

Verhindern, dass die Benutzeroberfläche beim Warten auf Tasks blockiert

Wie in einem der vorherigen Abschnitte schon angedeutet, kann es beim Aktualisieren der Benutzeroberfläche zu klassischen Deadlock-Situationen kommen. Das ist ganz besonders schlimm, wenn der UI-Thread in eine Warteposition gerät, bei der er mit einem der `Wait`-Befehle (siehe vorheriger Abschnitt) auf die Fertigstellung eines anderen Threads wartet, die ihrerseits mit `Invoke` Aktualisierungen an der Benutzeroberfläche vornehmen müssen, wie im folgenden Beispiel zu sehen.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 29\WpfNonBlockingUi
```

Wenn Sie diese Anwendung laufen lassen und auf die `Start Working!`-Schaltfläche klicken, rührt sich nichts mehr, und auf den ersten Blick ist nicht wirklich ersichtlich, wieso das so ist.

```
    Private Sub StartWorkingButton_Click(ByVal sender As System.Object,
                                         ByVal e As System.Windows.RoutedEventArgs) _
                                        Handles StartWorkingButton.Click

        MainProgress.Maximum = MAXVALUE

        Dim t As New Task(Sub()
                              For z = 0 To MAXVALUE
                                  Thread.SpinWait(SPINVALUE)

                                  'Auf UI-Thread delegieren, um den
                                  'ProgressBar zu aktualisieren. Funktioniert
                                  'nur nicht, da Invoke einen funktionierenden
                                  'Message-Loop benötigt, und der wird durch
                                  't.Wait blockiert!!
```

```
                                Dispatcher.Invoke(Sub(insideValue As Integer)
                                                      MainProgress.Value = insideValue
                                                  End Sub, z)
                            Next
                        End Sub)

        'Task starten
        t.Start()

        'Das blockiert den UI-Thread
        t.Wait()

        'Das werden wir nie sehen... :-(
        MessageBox.Show("Task abgeschlossen!")
    End Sub
```

Wenn Sie die Demoanwendung in diesem eingefrorenen Zustand mit der Pause-Schaltfläche der Toolbar von Visual Studio oder der Tastenkombination [Strg]+[Alt]+[Pause/Untbr.] unterbrechen und das *Threads*-Fenster über den Befehl *Threads* des Menüs *Debuggen/Fenster* (auch: [Strg]+[Alt]+[H]) einblenden, dann verrät ein zweiter Blick, was schief gelaufen ist (Abbildung 29.8).

Abbildung 29.8 Beim Öffnen der entscheidenden Threads wird deutlich: Sowohl der Arbeitsthread, dargestellt durch den Task, als auch der Hauptthread warten, und zwar jeweils auf den anderen – das ist die klassische Deadlock-Situation

Im *Threads*-Fenster lässt sich erkennen, dass sowohl der Thread unseres Tasks als auch der Thread der Hauptanwendung in Warteposition verharren: `Invoke` wartet darauf, dass die Windows-Nachrichtenwarteschlange des UI-Threads die Nachricht zum Ausführen des Delegaten weiterleitet, aber es ist genau der UI-Thread (Hauptthread im Fenster), der ebenfalls steht, nämlich durch `t.Wait`, und mit ihm »steht« auch die Warteschlange.

Eine sinnvolle Möglichkeit, dieses Problem zu lösen, liegt darin, den Task, der die lang dauernde Operation durchführt, in einem weiteren Task einzukapseln, der dann wiederum auf das Ende des ersten (inneren) warten und eine entsprechende Meldung auf dem Bildschirm ausgeben (bzw. die Benutzeroberfläche anderweitig aktualisieren) kann. In diesem Fall muss das Aktualisieren der Benutzeroberfläche bzw. das Ausgeben der `MessageBox` ebenfalls an den UI-Thread zurückgeleitet werden. Das folgende Beispiel verdeutlicht diesen Weg aus dem Dilemma:

```
Private Sub StartWorkingButton_Click(ByVal sender As System.Object,
                                     ByVal e As System.Windows.RoutedEventArgs) _
                                    Handles StartWorkingButton.Click

    MainProgress.Maximum = MAXVALUE

    Dim t2 As New Task(
        Sub()
            Dim t As New Task(Sub()
                                  For z = 0 To MAXVALUE
                                      Thread.SpinWait(SPINVALUE)

                                      'Auf UI-Thread delegieren, um den
                                      'ProgressBar zu aktualisieren.
                                      Dispatcher.Invoke(Sub(insideValue As Integer)
                                                            MainProgress.Value = insideValue
                                                        End Sub, z)
                                  Next
                              End Sub)

            'Worker-Task starten
            t.Start()

            'Das blockiert den UI-Thread jetzt nicht mehr, weil es in eigenem Task läuft.
            t.Wait()

            'So wird es funktionieren:
            Dispatcher.Invoke(Sub()
                                  MessageBox.Show("Task abgeschlossen!")
                              End Sub)

        End Sub)

    'Übergeordneten Task starten.
    t2.Start()

End Sub
```

Vorzeitiges Abbrechen von Tasks mit CancellationToken

Tasks können auf einfachste Weise vorzeitig beendet werden. Wenn Sie eine Klassenvariable verwenden, auf die alle Tasks zugreifen können, haben Sie die Möglichkeit, über diese Variable allen Threads zu signalisieren, dass sie ihre Arbeit beenden sollen. In der Regel wird solch ein Muster notwendig, wenn ein Task eine lang andauernde Operation beispielsweise in einer Schleife ausführt und diese Schleife demzufolge abbrechen bzw. vorzeitig verlassen soll, wenn die Tasks insgesamt abgebrochen werden sollen:

```
Private myCancelEverything As Boolean

Private Sub AbbrechenButton_Click(ByVal sender As System.Object,
                  ByVal e As System.Windows.RoutedEventArgs) Handles OkButton.Click
    myCancelEverything = True
End Sub

Sub StartTasks()
    Dim t = Task.Factory.StartNew(Sub()
                                      For tCount = 0 To Long.MaxValue
                                          Thread.SpinWait(SPINVALUE)
                                          If myCancelEverything Then
                                              Exit For

                                              'oder
                                              Exit Sub
                                          End If
                                      Next
                                  End Sub)

End Sub
```

Doch schauen wir uns das genauer an, so wirklich hat das nichts mit dem Abbrechen eines Tasks zu tun, denn: Im Grunde genommen sorgen wir mit einer Logik dafür, dass die Methode alternativ aber ordentlich beendet wird.

Für die meisten Fälle ist die Vorgehensweise die einfachste und vor allen Dingen auch schnellste Lösung. Doch unter Umständen müssen Sie vielleicht folgende Punkte berücksichtigen:

- Sie müssen am Ende rausfinden, ob und wenn ja welche Tasks abgebrochen und welche ordentlich beendet worden sind.
- Sie müssen vielleicht bei der Ausführung einer Wartemethode wie `Task.WaitAll` auch in der Lage sein, Tasks und damit das Warten auf die Tasks abzubrechen, ohne dass Sie zwingend darauf warten müssen, dass alle Tasks beendet sind: Der wartende Thread soll also dann weiterlaufen, wenn der erste Thread bereits abgebrochen wurde.

In diesem Fall müssen Sie wieder, wie bei `Parallel.For` und `Parallel.ForEach`, ein so genanntes *Cancellation Token* (in etwa: Abbrechen-Signalmarke) verwenden. Mit dessen Hilfe können Sie zur Laufzeit über seine `IsCancellationRequested`-Eigenschaft feststellen, ob ein laufender Task vorzeitig abgebrochen werden soll und, wenn dies der Fall ist, diesen tatsächlich über das Auslösen einer `OperationCanceledException` (etwa: Operationsabbruch-Ausnahme) mithilfe der Methode `ThrowIfCancellationRequested` der `CancellationToken`-Instanz beenden. Puh! Fassen wir im Telegrammstil zusammen, wie ein solcher Ablauf ausschaut:

- Sie erstellen eine `CancellationTokenSource`-Instanz, die Ihnen einerseits das `CancellationToken` liefert, und andererseits erlaubt, das Token in den »Abbruchanforderungsmodus« zu bringen. Das können Sie auf dem Token nämlich nicht direkt einstellen – Sie brauchen dazu die `CancellationTokenSource`-Instanz:

  ```
  myCtSource = New CancellationTokenSource()
  ```

- Sie ermitteln aus der `CancellationTokenSource`-Instanz das eigentliche `CancelletionToken`, und das geben Sie jedem Task bei seiner Erstellung mit:

```
myCToken = myCtSource.Token

Dim aTask As New Task(Sub()
                          'Loop
                          Do
                              myCToken.ThrowIfCancellationRequested()
                          Loop
                      End Sub, myCToken)
```

- Wenn eine Operation sehr lange dauert und damit ein Kandidat ist, abgebrochen werden zu können, rufen Sie in regelmäßigen Abständen – zum Beispiel innerhalb seiner Schleife – die `ThrowIfCancellationRequested`-Methode auf. Machen Sie das für alle lang laufenden Tasks, die parallel ablaufen und zur gleichen Zeit abgebrochen können werden sollen.
- Möchten Sie auf abbrechbare Tasks warten, zum Beispiel mit `WaitAll`, dann setzen Sie den Methodenaufruf in einen `Try … Catch`-Block und fangen Sie gezielt eine `AggregateException` ab. Auf diese Weise warten Sie, bis alle Tasks abgebrochen wurden:

```
Try
    Task.WaitAll({aTask, anotherTask, task3})
Catch ex As AggregateException
    For Each item In ex.InnerExceptions
        Debug.Print(item.Message)
    Next
End Try
```

- Oder: Falls Sie möchten, dass es mit dem Hauptthread bzw. dem Thread, der die einzelnen Tasks aufgerufen hat, direkt weitergeht, wenn schon der *erste* Task abgebrochen wurde, dann übergeben Sie `WaitAll` das `CancellationToken` und ändern Sie den abzufangenden `Exception`-Typ in eine `OperationCanceledException`:

```
Try
    Task.WaitAll({aTask, anotherTask, task3}, myCToken)
Catch ex As OperationCanceledException
    'Entsprechende Maßnahmen hier.
End Try
```

- Um die Tasks schließlich wirklich abzubrechen, bemühen Sie die `CancellationTokenSource`, etwa auf die folgende Art und Weise:

```
Private Sub AbbrechenButton_Click(ByVal sender As System.Object,
                  ByVal e As System.Windows.RoutedEventArgs) Handles AbbrechenButton.Click
    myCtSource.Cancel()
End Sub
```

- Wenn Sie nach Ablauf eines Tasks herausfinden wollen, ob ein Task durchgelaufen oder vorzeitig abgebrochen ist, überprüfen Sie einfach seine `Status`-Eigenschaft.

Das folgende Beispiel ist die Abwandlung des vorherigen Demoprogramms und demonstriert das gerade Gesagte im Kontext. In Ergänzung zum Beispiel des vorherigen Abschnitts startet dieses Beispiel gleich mehrere Tasks, um das koordinierte Warten und Abbrechen besser zu verdeutlichen. Der Benutzer kann,

während das Programm läuft, die entsprechende Schaltfläche anklicken, um beide parallel laufenden Tasks abzubrechen. Ein dritter Task nach dem Ablauf der ersten beiden Tasks läuft übrigens – auch das zeigt das Beispiel – nur dann an, wenn die *Stop!*-Schaltfläche zuvor nicht betätigt wurde. Am Ende gibt das Beispiel im Debug-Fenster aus, ob die Tasks regulär beendet oder vorzeitig abgebrochen wurden.

Im folgenden Listing sind die relevanten Stellen für das koordinierte Abbrechen der Tasks mit Fettschrift gekennzeichnet.

```
Private Sub StartWorkingButton_Click(ByVal sender As System.Object,
                            ByVal e As System.Windows.RoutedEventArgs) _
                           Handles StartWorkingButton.Click

    If StartWorkingButton.Content.ToString = "Stop!" Then
        If myCtSource IsNot Nothing Then
            myCtSource.Cancel()
            Exit Sub
        End If
    End If

    StartWorkingButton.Content = "Stop!"

    MainProgress.Maximum = MAXVALUE

    myCtSource = New CancellationTokenSource()
    myCToken = myCtSource.Token

    Dim sw = Stopwatch.StartNew

    Dim tMain As New Task(
        Sub()
            Dim t As New Task(
                Sub()
                    For z = 0 To MAXVALUE
                        Thread.SpinWait(SPINVALUE)

                        'Auf UI-Thread delegieren, um den
                        'ProgressBar zu aktualisieren.
                        Dispatcher.Invoke(Sub(insideValue As Integer)
                                              MainProgress.Value = insideValue
                                          End Sub, z)
                        If myCToken.IsCancellationRequested Then
                            Debug.Print("Throwing Cancellation Requested Exception:" &
                                        sw.ElapsedMilliseconds.ToString)
                            myCToken.ThrowIfCancellationRequested()
                        End If
                    Next
                End Sub, myCToken)

            Dim t2 As New Task(
                Sub()
                    For z = 0 To MAXVALUE \ 10
                        Thread.SpinWait(SPINVALUE * 10)

                        'Auf UI-Thread delegieren, um den
                        'ProgressBar zu aktualisieren.
                        Dispatcher.Invoke(Sub(insideValue As Integer)
                                              Label1.Content = "Task 2 ist auch aktiv:" &
                                                insideValue.ToString
                                          End Sub, z)
```

```
                    If myCToken.IsCancellationRequested Then
                       Debug.Print("Throwing Cancellation Requested Exception:" &
                                          sw.ElapsedMilliseconds.ToString)
                        myCToken.ThrowIfCancellationRequested()
                        'Throw New OperationCanceledException("Thrown manually")
                    End If
                Next
            End Sub, myCToken)

        'Worker-Tasks starten
        t.Start()
        t2.Start()

        Try
            'In dieser Version kommt die Exception beim ersten Task-Abbruch.
            Task.WaitAll({t, t2}, myCToken)
            'In dieser Version kommt die AggregateException, wenn alle Task abgebrochen wurden.
            'Task.WaitAll({t, t2})

            'Auf diese Weise wird der erste Abbruch schon abgefangen.
        Catch ex As OperationCanceledException
            Debug.Print("Caught Exception: " & sw.ElapsedMilliseconds.ToString)
            Dispatcher.Invoke(Sub()
                                  MessageBox.Show("Task abgebrochen!")
                              End Sub)
            'Version, die ohne CancellationToken wartet.
            'Catch ex As AggregateException
            '    Dispatcher.Invoke(Sub()
            '                          MessageBox.Show("Alle Tasks abgebrochen!")
            '                      End Sub)
        End Try

        'Wenn die vorherigen Tasks abgebrochen wurden, startet dieser
        'Task gar nicht erst mehr - denn auch das wird durch myCToken gesteuert.
        Dim t3 = Task.Factory.StartNew(
                            Sub()
                                Dispatcher.Invoke(Sub()
                                                      MessageBox.Show("Task 3 Start!")
                                                  End Sub)
                            End Sub, myCToken)

        Dispatcher.Invoke(Sub()
                              If myCToken.IsCancellationRequested Then
                                  Label1.Content = "Abgebrochen."
                                  StartWorkingButton.Content = "Start!"
                              Else
                                  Label1.Content = "Fertig."
                                  StartWorkingButton.Content = "Start!"
                                  MessageBox.Show("Task abgeschlossen!")
                              End If
                          End Sub)

        Debug.Print("Status Task 1:" & t.Status.ToString)
        Debug.Print("Status Task 2:" & t2.Status.ToString)
        Debug.Print("Status Task 3:" & t3.Status.ToString)

    End Sub, myCToken)

    'Übergeordneten Task starten.
    tMain.Start()
End Sub
```

Threads synchronisieren

Es gibt unzählige Situationen, bei denen Threads eben nicht so einfach nebeneinander herlaufen und sich beliebigen Zugriff auf die vorhandenen Ressourcen einer Anwendung nehmen dürfen. Denken Sie nur an den gleichzeitigen Zugriff auf Auflistungen: Wenn in dem Moment, in dem Thread A beginnt, die `Count`-Eigenschaft einer Auflistung abzufragen, Thread B gerade damit fertig war, ein neues Element hinzuzufügen, haben Sie in Thread A den Zustand erreicht, bei dem Sie unter falschen Voraussetzungen mit einer Auflistung weiterarbeiten. Solche Dinge dürfen nicht passieren; der Zugriff auf die Methoden und Eigenschaften einer Auflistung muss threadsicher und synchronisiert sein: Sie dürfen die `Count`-Eigenschaft zwar abfragen, andere Threads müssen mit dem Hinzufügen oder Löschen von Elementen allerdings warten, bis diese Abfrage erfolgt ist. Umgekehrt gilt das Gleiche: Threads, die die `Count`-Eigenschaft einer Auflistung abfragen, müssen warten, bis andere Threads damit fertig sind, Elemente zu derselben Auflistung hinzuzufügen.

Das nächste Beispielprogramm lehnt sich an das erste Beispiel dieses Kapitels an – es ist nur ein wenig »eloquenter«. Lediglich um zu zeigen, inwieweit nicht synchronisierte Vorgänge beim Threading richtig danebengehen können, modifiziert es die Ausgaberoutine der Threadroutine auf folgende Weise:

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 29\SimpleThread02 (SyncLock)
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
Imports System.Threading

Public Class frmMain

    'Feldvariable, damit die Threads durchnummeriert werden können.
    'Dient nur zur späteren Unterscheidung des laufenden Threads, wenn
    'er Ergebnisse im Ausgabefenster darstellt.
    Private myArbeitsThreadNr As Integer = 1

    'Feldvariablen, mit der demonstrativ "Mist gebaut" wird...
    Private myThreadString As String

    Private Sub btnThreadStarten_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
                                       Handles btnThreadStarten.Click

        'Neuen Thread erstellen; der Worker-Thread läuft darin als Lambda:
        Dim locThread As New Thread(
            Sub()
                Dim strTemp As String

                For c As Integer = 0 To 50
                    strTemp = Thread.CurrentThread.Name + ":: " + c.ToString
                    'Nehmen Sie die Auskommentierung von SyncLock zurück,
                    'um den "Fehler" des Programms zu beheben
```

```
                    'SyncLock Me
                    myThreadString = ""
                    For z As Integer = 0 To strTemp.Length - 1
                        'Simulieren des "Zusammenbastelns" der Zeichenfolge ...
                        myThreadString += strTemp.Substring(z, 1)
                        '... mit ein wenig mehr Arbeitsaufwand.
                        Thread.SpinWait(1000000)
                    Next
                    ThreadSafeTextWindow.TSWriteLine(myThreadString)
                    'End SyncLock
                Next
            End Sub)

        'Thread wird in jedem Fall mit Anwendung beendet:
        locThread.IsBackground = True
        'Thread hat einen Namen:
        locThread.Name = "Arbeits-Thread: " + myArbeitsThreadNr.ToString
        'Thread starten
        locThread.Start()
        'Zähler, damit die Threads durch ihren Namen unterschieden werden können
        myArbeitsThreadNr += 1

    End Sub

    Private Sub btnBeenden_Click(ByVal sender As System.Object,
                                 ByVal e As System.EventArgs) Handles btnBeenden.Click
        'Threads sind Background-Threads; ist also kein Problem.
        Me.Close()
    End Sub

End Class
```

Hier passiert Folgendes: `myThreadString` ist eine Feldvariable der Klasse, und sie wird für jedes Zeichen, das im Ausgabefenster für einen Zählungseintrag erscheinen soll, Zeichen für Zeichen zusammengesetzt. Die `TSWriteLine`-Methode gibt diesen String anschließend aus, wenn das Zusammenbasteln des Strings für einen Eintrag erledigt ist.

Wenn Sie das Programm starten und die Schaltfläche nur ein einziges Mal anklicken, sodass auch nur ein einziger Thread läuft, bleibt alles beim Alten. Doch wehe, Sie starten bereits während der erste Threads läuft, auch nur einen weiteren, dann nämlich ist Chaos angesagt, wie auch in Abbildung 29.9 zu sehen.

Abbildung 29.9 Zwei Threads im Kampf um eine Feldvariable – und das daraus resultierende und nicht wirklich zufrieden stellende Ergebnis

Das Problem ist, nicht genau voraussehen zu können, wann eine Threadprozedur zum Zug kommt – denn beim *Preemptive Multitasking* bestimmt diesen Zeitpunkt das Betriebssystem. Im hier vorliegenden Fall »meint« das Betriebssystem, dass ein anderer Thread am besten zum Zuge kommen kann, wenn der eine Thread gerade zu warten beginnt (was beide durch die *Sleep*-Methode in regelmäßigen Abständen machen).

Synchronisieren von Threads mit SyncLock

Dieses Problem können Sie durch das Blocken von Codeabschnitten in Abhängigkeit von verwendeten Objekten aus der Welt schaffen. Die `SyncLock`-Anweisung ist hier der Schlüssel zur Lösung. Wenn Sie den Code folgendermaßen umbauen, kann sich das Ergebnis wieder sehen lassen:

```
.
.
.
                For c As Integer = 0 To 50
                    strTemp = Thread.CurrentThread.Name + ":: " + c.ToString
                    'Nehmen Sie die Auskommentierung von SyncLock zurück,
                    'um den "Fehler" des Programms zu beheben
                    SyncLock Me
                        myThreadString = ""
                        For z As Integer = 0 To strTemp.Length - 1
                            'Simulieren des "Zusammenbastelns" der Zeichenfolge ...
                            myThreadString += strTemp.Substring(z, 1)
                            '... mit ein wenig mehr Arbeitsaufwand.
                            Thread.SpinWait(1000000)
                        Next
```

```
                        ThreadSafeTextWindow.TSWriteLine(myThreadString)
                    End SyncLock
                Next
            End Sub)
.
.
.
```

SyncLock verwendet ein Objekt, um den nachfolgenden Code für alle anderen Threads zu schützen, die einen Verweis auf dasselbe Objekt halten. Dieses Objekt sollte daher für die Dauer des Zugriffs nicht verändert werden, da der Schutz sonst nicht oder nicht mehr zuverlässig funktioniert.

WICHTIG Im Gegensatz zu vielen vorherrschenden Meinungen schützt SyncLock nicht das Objekt selbst vor Veränderungen, sondern den Code gegen den gleichzeitigen Zugriff von einem anderen Thread – das angegebene Objekt dient nur quasi als Aufhänger für den Schutz des entsprechenden Codes. Achten Sie auch darauf, dass es sich bei dem Objekt um eines handelt, das nicht schon an anderer Stelle zum Locken eines bestimmten Codebereichs verwendet wird. Viele Entwickler verwenden zum Beispiel das Typ-Objekt einer bestimmten Klasse als SyncLock-Argument, also beispielsweise

```
SyncLock GetType(Me)
    .
    .
    .
End SyncLock
```

Das kann legitim sein, wenn das resultierende Objekt (der Typ) wirklich an jeder Stelle der Anwendung verschiedene Codeblöcke einleitet, die alle nicht gleichzeitig ausgeführt werden dürfen. Die Gefahr ist aber groß, dass das versehentlich passiert. Falls Me nämlich an mehreren Stellen ein Formular ist, wird bei jedem SyncLock, das auf diese Weise formuliert ist, immer wieder dasselbe Objekt (nämlich Form.GetType) verwendet, und Sie kommen ganz schnell in einen Zustand, wo ein SyncLock auf das Beenden eines anderen SyncLock wartet und umgekehrt, und die Anwendung hängt komplett – hier ergibt sich also eine klassische Deadlock-Situation.

Verwenden Sie also nach Möglichkeit für SyncLock Objekte, die Sie gezielt zu diesem Zweck instanziieren, und sei es einfach nur eine Objektinstanz, die nur zum SyncLock im entsprechenden Kontext eingesetzt wird. Verwenden Sie statische Objektinstanzen nur dann, wenn Sie Threads haben, die auch im statischen Kontext geblockt werden müssen. Die Regel ist hier: Falls Sie bestimmte Member einer Klassen*instanz* gegen gleichzeitigen Zugriff schützen müssen, dann sollten Sie zum SyncLock auch Instanzobjekte verwenden; falls es dagegen statische Ressourcen sind, die gegen gleichzeitigen Zugriff geschützt werden müssen (gilt zum Beispiel bei Hardwarekomponenten), greifen Sie auf Objekte zurück, die statisch sind, also in Visual Basic mit Shared ausgezeichnet sind (Me, wie hier im Beispiel zu sehen, ist übrigens nicht immer eine gute Idee, denn das Formular könnte auch noch an anderer Stelle zum SyncLock-Vorgang herhalten, und auch das könnte in einer Deadlock-Situation resultieren).

Außerdem sollte am Rande erwähnt werden: Da SyncLock intern einen Try … Catch-Block verwendet und damit auch im Falle einer Ausnahme der Schutz des Blocks wieder aufgehoben werden kann, dürfen Sie nicht mit der Goto-Anweisung in einen SyncLock-Block springen.

Was passiert also im Detail bei der Ausführung dieser Routine? Nun, sobald der erste Thread den Block erreicht, sperrt er den Zugriff auf den Code für jeden weiteren Thread durch Zuhilfenahme des Objekts. Für dieses Objekt müssen folgende Bedingungen gelten:

- Es muss sich um ein Objekt handeln, das auf eine gültige Adresse im Managed Heap verweist – es darf also nicht `Nothing` sein
- Es muss zwingend ein Referenztyp sein (logisch, denn Wertetypen liegen nicht auf dem Managed Heap)
- Es darf durch die geschützte Routine nicht verändert werden

Im Beispiel wird das einfach durch die Verwendung der eigenen Instanz `Me` erreicht. `Me` erfüllt diese Bedingungen stets. Bei statischen Routinen, bei denen `Me` natürlich nicht anwendbar ist, sollten Sie statische Instanzen erstellen, um diese als Hilfe zum Schutz einer Routine zu verwenden, etwa:

```
Public Class SomeClass

    Private Shared S_SyncObject as New Object.
.
        SyncLock (S_SyncObject)

            'Hier steht der Code der zu schützenden Routine

        End SyncLock
.
.
End Class
```

Wenn nun ein zweiter Thread den geschützten Programmteil erreicht, dann muss er vor dem SyncLock so lange warten, bis der Thread, der den Programmteil gerade durchläuft, `End SyncLock` erreicht hat.

In unserem Beispiel kann also der Member-String ohne Sorge zu Ende aufgebaut und anschließend verarbeitet werden. Dass ein anderer Thread den Member-String an dieser Stelle in irgendeiner Form verändert, ist ausgeschlossen, weil er auf alle Fälle zu warten hat.

HINWEIS Einen Programmteil, der besonderen Synchronisationsschutz benötigt bzw. erfährt, nennt man *kritischen Abschnitt* oder neudeutsch: *Critical Section*.

Mehr Flexibilität in kritischen Abschnitten mit der Monitor-Klasse

Die `SyncLock`-Funktion hat einen großen Nachteil: Sie ist unerbittlich. Wenn ein anderer Thread bereits einen kritischen Abschnitt durchläuft, dann warten alle anderen Threads am Anfang des kritischen Abschnitts – ob sie wollen oder nicht. Sie können dagegen nichts tun. Die `Monitor`-Klasse bietet an dieser Stelle die größere Flexibilität, da sie ein paar zusätzliche Methoden im Angebot hat, mit der ein Thread auch nachschauen kann, ob er warten müsste, wenn er einen kritischen Bereich beträte. Ein weiteres Beispiel soll das verdeutlichen:

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 29\SimpleThread03 (Monitor)
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
Private Sub btnThreadStarten_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
                                   Handles btnThreadStarten.Click

    'Dieses Objekt kapselt den eigentlichen Thread
    Dim locThread As New Thread(
        Sub()

            'Dies ist der eigentliche Arbeits-Thread (Worker Thread), der das
            'Hochzählen und die Werteausgabe übernimmt

            Dim strTemp As String

            For c As Integer = 0 To 50
                strTemp = Thread.CurrentThread.Name + ":: " + c.ToString
                'Der Thread wartet maximal eine Sekunde; bekommt er in dieser
                'Zeit keinen Zugriff auf den Code, steigt er aus.
                If Not Monitor.TryEnter(Me, 1) Then
                    ThreadSafeTextWindow.TSWriteLine(
                             Thread.CurrentThread.Name + " meldet: Gerade besetzt!")
                    Thread.Sleep(50)
                Else
                    'Zugriff wurde gewährt - jetzt nimmt der Arbeits-Thread
                    'erst seine eigentliche Arbeit auf
                    myThreadString = ""
                    For z As Integer = 0 To strTemp.Length - 1
                        myThreadString += strTemp.Substring(z, 1)
                        Thread.Sleep(5)
                    Next
                    ThreadSafeTextWindow.TSWriteLine(myThreadString)
                    Monitor.Exit(Me)
                End If
            Next

        End Sub)

    'Thread-Namen bestimmen
    locThread.Name = "Arbeits-Thread: " + myArbeitsThreadNr.ToString

    'Background-Thread bestimmen:
    locThread.IsBackground = True

    'Thread starten
    locThread.Start()
    'Zähler, damit die Threads durch ihren Namen unterschieden werden können
    myArbeitsThreadNr += 1

End Sub
```

In dieser Version des Arbeitsthreads erfolgt die Synchronisation durch die `Monitor`-Klasse. Allerdings ist der Thread, der auf seinen Vorgänger warten muss, ein wenig ungeduldig. Bekommt er nicht innerhalb von einer Millisekunde Zugriff auf den kritischen Abschnitt, verarbeitet er den Code für die entsprechende Werterhöhung nicht, sondern gibt nur lapidar die Meldung *Threadname meldet: Gerade besetzt!* aus.

Abbildung 29.10 Ein Thread, der auf den kritischen Abschnitt nicht zugreifen kann, wartet hier maximal eine Millisekunde, bevor er mit einer lapidaren Meldung den Wartevorgang abbricht

Möglich wird das durch die Verwendung der `TryEnter`-Methode der `Monitor`-Klasse – die im Übrigen statisch ist; Sie brauchen die `Monitor`-Klasse also nicht zu instanziieren (Sie könnten es auch gar nicht). Diese Methode überprüft, ob der Zugriff auf einen kritischen Abschnitt, der durch ein angebbares Objekt genau wie bei `SyncLock` gesteuert wird, freigegeben ist. Trifft dies zu, liefert sie `True` als Funktionsergebnis, andernfalls `False`.

Auf diese Weise gibt es beim Ablauf des Programms Effekte, wie Sie sie auch in Abbildung 29.10 beobachten können. Doch die `Monitor`-Klasse kann noch mehr, wie das folgende Beispiel zeigt.

Monitor – Wait – Pulse

Angenommen, Sie möchten, dass im bislang gezeigten Beispielprogramm die durchgeführte Arbeitsaufgabe in Fünferportionen erfolgt. Eine Weile (5 Durchläufe) darf also der erste Thread ran, dann (weitere 5 Durchläufe) der zweite Wartende, dann der dritte usw. In diesem Fall brauchen Sie drei weitere Funktionen – oder besser – Funktionalitäten:

- Sie benötigen eine Methode, die einen Thread in einen Wartezustand versetzt, wenn er sich innerhalb eines kritischen Bereichs befindet.
- Sie benötigen eine Methode, die einen Thread, der sich in einen Wartezustand versetzt hat, wieder aufwecken kann. Gibt es mehrere wartende Threads, sollte die Funktion in der Lage sein, nicht nur alle, sondern auch nur einen (beliebigen) Thread wieder zum Leben zu erwecken.

Diese Möglichkeiten bzw. Methoden gibt es. Sie heißen `Monitor.Wait`, `Monitor.PulseAll` und `Monitor.Pulse`. Für unser Vorhaben brauchen wir darüber hinaus eine Technik, die es uns ermöglicht, zwischen einem und mehreren laufenden Threads zu unterscheiden. Denn solange nur ein einzelner Thread läuft, darf dieser sich natürlich nicht in den Schlafmodus versetzen, da es keinen anderen Thread gibt, der ihn wieder wachrütteln könnte. Doch diese Technik zu implementieren ist simpel: Eine einfache, statische Threadzählvari-

able vollführt diesen Trick. Wird ein neuer Thread gestartet, wird der Zähler zu Beginn der Threadprozedur hochgezählt; beendet er seine Arbeitsroutine, zählt er ihn wieder runter. Er versetzt sich nur dann in den Schlaf, wenn noch mindestens ein weiterer Thread läuft. Getreu dem Motto »Der Letzte macht das Licht aus« muss jeder Thread vor dem Verlassen seiner Arbeitsprozedur noch überprüfen, ob es weitere Threads gibt und den jeweils letzten prophylaktisch aus seinem Dornröschenschlaf erwecken.

Abbildung 29.11 Schlagabtausch: Wartende Threads dürfen im kritischen Bereich nur kurz verweilen und werden dann durch den nächsten wartenden Thread abgelöst, der dann wieder eine Weile werkeln kann, usw. Auf diese Weise zählt jeder Thread bis 50, nur nicht hintereinander – er muss anderen zwischendurch den Vortritt geben.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel29\SimpleThread04 (Monitor Wait Pulse)
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Wenn Sie dieses Programm starten und mindestens zwei Mal auf die Schaltfläche zum Starten des Threads klicken, sehen Sie nach einer Weile ein Ergebnis, wie es in etwa dem in Abbildung 29.11 gezeigten entspricht. Das entsprechende Listing des Arbeitsthreads sieht folgendermaßen aus:

```
    'Die brauchen wir, um festzustellen, wie viele Threads unterwegs sind.
    Private myThreadCount As Integer

    'Dies ist der eigentliche Arbeits-Thread (Worker Thread), der das
    'Hochzählen und die Werteausgabe übernimmt.
    Private Sub UmfangreicheBerechnung()

        Dim strTemp As String
        Dim stepCount As Integer
```

```
        'Neuer Thread: Zählen!
        myThreadCount += 1

        'Hier beginnt der kritische Abschnitt.
        Monitor.Enter(Me)
        For c As Integer = 0 To 50
            strTemp = Thread.CurrentThread.Name + ":: " + c.ToString
            'Der Thread wartet maximal eine Sekunde; bekommt er in dieser
            'Zeit keinen Zugriff auf den Code, steigt er aus.
            'Zugriff wurde gewährt - jetzt nimmt der Arbeits-Thread
            'erst seine eigentliche Arbeit auf.
            myThreadString = ""
            For z As Integer = 0 To strTemp.Length - 1
                myThreadString += strTemp.Substring(z, 1)
                Thread.Sleep(1)
            Next
            ThreadSafeTextWindow.TSWriteLine(myThreadString)
            'Der Thread darf sich nur dann schlafen legen, wenn mindestens
            'ein weiterer Thread unterwegs ist, der ihn wieder wecken kann.
            If myThreadCount > 1 Then
                stepCount += 1
                If stepCount = 5 Then
                    stepCount = 0
                    'Ablösung naht!
                    Monitor.Pulse(Me)
                    ThreadSafeTextWindow.TSWriteLine("Hab dich, du bist's")
                    'Abgelöster geht schlafen.
                    Monitor.Wait(Me)
                End If
            End If
        Next
        If myThreadCount > 1 Then
            'Alle anderen schlafen zu dieser Zeit.
            'Also mindestens einen wecken, bevor dieser Thread geht.
            'Passiert das nicht, schlafen die anderen bis zum nächsten Stromausfall...
            Monitor.Pulse(Me)
        End If
        'Hier ist der kritische Abschnitt wieder vorbei.
        Monitor.Exit(Me)
        'Thread-Zähler vermindern.
        myThreadCount -= 1
    End Sub
```

Synchronisieren von beschränkten Ressourcen mit Mutex

Wenn Threads Aufgaben lösen ohne dazu mit der Außenwelt interagieren zu müssen, ist es beinahe egal, wie viele Threads gleichzeitig laufen (natürlich sollten Sie dabei berücksichtigen, dass Sie grundsätzlich nur so viele Threads wie gerade nötig verwenden sollten, da das Umschalten zwischen den Threads selbst natürlich auch Rechenleistung benötigt). Doch wenn bestimmte Anwendungen Komponenten der nur begrenzt vorhandenen Hardware benötigen, dann müssen diese Komponenten zwischen den verschiedenen Threads ideal aufgeteilt werden.

An dieser Stelle kommt die `Mutex`-Klasse ins Spiel. Ihr Vorteil: Sie funktioniert zwar prinzipiell wie die `Monitor`-Klasse, doch sie ist instanziierbar. Das bedeutet: Sie können durch `Mutex`-Instanzen, deren Anzahl Sie von der Verfügbarkeit benötigter Hardwarekomponenten abhängig machen, einen Verteiler organisieren, der sich um die Zuteilung der jeweils nächsten freien Hardwarekomponente kümmert.

Das folgende Beispiel demonstriert den Umgang mit `Mutex`-Klassen. Damit dieses Beispiel die `Mutex`-Klasse auch hardwareunabhängig auf jedem Computer demonstrieren kann, ist es ein wenig anders aufgebaut als die Beispiele, die Sie bisher kennen gelernt haben: Drei Ausgabefelder innerhalb des Formulars dienen zur Emulation von drei Hardwarekomponenten; Ziel des Programms ist es, die laufenden Threads, die inhaltlich nichts anderes machen als das vorherige Beispielprogramm, so auf die drei Komponenten aufzuteilen, dass sie optimal genutzt werden.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel29\SimpleThread05 (MutexDemo)
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Wenn Sie das Programm starten, sucht es sich nach dem Klick auf die Schaltfläche *Threads starten* die erste freie Hardware-Ressource (also ein Ausgabetextfeld). Ein weiterer Klick auf diese Schaltfläche startet einen weiteren Thread, der das nächste freie Ausgabefenster verwendet. Wenn Sie den vierten Thread starten, während alle vorherigen Threads noch laufen, wartet dieser auf das nächste freie Fenster. Sobald einer der Threads beendet ist, übernimmt er dessen Ausgabefenster. Die Arbeitsthreadroutine ist so ausgelegt, dass eine einzelne Operation innerhalb des Threads unterschiedlich lange dauert. Nur so kann eine realistische Emulation einer simulierten Hardwarekomponente realisiert werden. Zu diesem Zweck gibt es ein im Konstruktor des Programms initialisiertes `Random`-Objekt, das den jeweils nächsten zufälligen Wert für die `Sleep`-Methode eines Arbeitsthreads generiert.

HINWEIS Das Programm gibt die Bildschirmmeldungen nicht direkt mit einer bestimmten Anweisung aus – so wie Sie es von den bisherigen Beispielen gewohnt waren. Stattdessen verwendet es drei `TextBox`-Steuerelemente für die Textausgabe.

Abbildung 29.12 Drei Ausgabebereiche simulieren drei Hardware-Ressourcen. Über ein Array von *Mutex*-Objekten wird erkannt, welche als Nächstes zur Verfügung steht; der nächste anstehende Thread läuft dann im nächsten freien Fenster

Das Codelisting:

```
'Feldvariable, damit die Threads durchnummeriert werden können.
'Dient nur zur späteren Unterscheidung des laufenden Threads, wenn
'er Ergebnisse im Ausgabefenster darstellt.
Private myArbeitsThreadNr As Integer = 1

'Speicher für Mutex-Objekte
Private myMutexes() As Mutex

'Speicher für TextBox-Controls
Private myTxtBoxes() As TextBox

'Zufallsgenerator für die künstlichen Wartezeiten im
'Arbeitsthread. Damit kann ein Thread unterschiedlich lange dauern.
Private myRandom As Random

Public Sub New()
    MyBase.New()

    ' Dieser Aufruf ist für den Windows Form-Designer erforderlich.
    InitializeComponent()
    'Mutexes definieren
    myMutexes = New Mutex() {New Mutex, New Mutex, New Mutex}
    'Textbox-Array zuweisen
```

```
        myTxtBoxes = New TextBox() {txtHardware1, txtHardware2, txtHardware3}
        'Zufallsgenerator initialisieren
        myRandom = New Random(DateTime.Now.Millisecond)

    End Sub

    Private Sub btnThreadStarten_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
                                       Handles btnThreadStarten.Click

        Dim locThread As New Thread(
            Sub()
                Dim locMutexIndex As Integer
                Dim locTextBox As TextBox

                'Hier beginnt der kritische Abschnitt
                'Warten, bis eine TextBox "frei" wird
                locMutexIndex = Mutex.WaitAny(myMutexes)

                'Textbox, die dem freien Mutex entspricht, finden
                locTextBox = myTxtBoxes(locMutexIndex)
                For c As Integer = 0 To 50
                    SyncLock Me
                        If Me.IsHandleCreated Then
                            'Text in der ermittelten TextBox ausgeben
                            '(wir müssen zum UI-Thread delegieren, deswegen der Umweg über Invoke).
                            Me.BeginInvoke(Sub(tb As TextBox, txt As String)
                                               tb.Text += txt
                                               tb.SelectionStart = tb.Text.Length - 1
                                               tb.ScrollToCaret()
                                           End Sub, {locTextBox,
                                           Thread.CurrentThread.Name + ":: " + c.ToString + vbNewLine})
                        End If
                    End SyncLock
                    'Eine zufällige Weile lang warten
                    Thread.Sleep(myRandom.Next(50, 400))
                Next
                'Hier ist der kritische Abschnitt wieder vorbei.
                'Verwendete TextBox (Mutex) wieder freigeben
                myMutexes(locMutexIndex).ReleaseMutex()
            End Sub)

        locThread.IsBackground = True
        locThread.Name = "Arbeits-Thread: " + myArbeitsThreadNr.ToString

        'Thread starten
        locThread.Start()
        'Zähler, damit die Threads durch ihren Namen unterschieden werden können
        myArbeitsThreadNr += 1

    End Sub
```

Ein paar zusätzliche Erklärungen zu diesem Programm: Ein `Mutex`-Array wird als Klassenmember zur Verwaltung der zur Verfügung stehenden Ressourcen im Konstruktor des Programms erstellt. Mit der statischen Funktion `WaitAny`, der ein `Mutex`-Array übergeben wird, wartet ein Thread so lange, bis eines der `Mutex`-Objekte im Array frei wird. Ist dies der Fall, liefert `WaitAny` den Index auf das jetzt freie `Mutex`-Objekt zurück, ändert dessen Zustand aber gleichzeitig in »blockiert«. Der zurückgelieferte Index wird anschließend verwendet, um die korrelierende `TextBox` zu finden, die diesem `Mutex` (nur über den Indexwert) quasi zugeordnet ist. Der Thread verwendet anschließend diese `TextBox` (wird über `locTextBox` referenziert), um die Ausgabe durchzuführen. `WaitAny` wartet übrigens nicht nur auf einen freien `Mutex`, sondern blockiert ihn auch wieder für den Thread, der ihn angefordert hat. Der Mutex bleibt so lange im »Blockiert-Zustand«, bis der Thread den Mutex mit der `ReleaseMutex`-Methode wieder freigibt.

Wie geht es weiter?

Das Thema Threading kann eigene Bücher füllen, und mit über 60 Seiten gehört dieses Kapitel bereits zu den umfangreicheren Kapiteln in diesem Buch. Aber es kann natürlich nur eine Einführung und Ihnen damit die grundlegenden Techniken an die Hand geben.

Ein guter Ausgangspunkt für weitere Quellen zur Programmierung mit der Task Parallel Library (TPL) ist sicherlich *http://Channel9.msdn.com* mit den zahlreichen Videos, auch vom TPL-Guru und –Entwickler Stephen Toub. Das TPL-Team selbst hat ebenfalls einen Blog, in dem Sie ebenfalls Hinweise auf weiterführende Informationen (*http://blogs.msdn.com/b/pfxteam/*) und viele, viele Beispielanwendungen finden (*http://code.msdn.microsoft.com/ParExtSamples*).

Stichwortverzeichnis

#Region 111
&= -Operator 56
*= -Operator 57
.cctor 354
.ctor 354, 381
.NET
  primitive Datentypen 256
.NET Framework
  Assembly 67
  Auflistungen 628
  Base Class Library 72
  BCL 72
  Class Library 72
  CLI 71
  CLR 71
  Common Language Infrastructure 71
  Common Language Runtime 71
  Common Type System 73
  CTS 73
  DateTime *siehe* Date-Datentyp
  Eigenschaften überschreiben 388
  FCL 72
  Framework, Aufbau 74
  IML 73
  Int16 *siehe* Short-Datentyp
  Int32 *siehe* Integer-Datentyp
  Int64 *siehe* Long-Datentyp
  Intermediate Language 73
  JITter 73
  Klassen erweitern 388
  Managed Heap 326
  Methoden überschreiben 388
  Multizielversionen 87
  Namespace 67
  Polymorphie *siehe* Polymorphie
  Profile 93
  Terminologien 66
  UInt16 *siehe* UShort-Datentyp
  UInt32 *siehe* UInteger-Datentyp
  UInt64 *siehe* ULong-Datentyp
  verschiedene Zielversionen in einer Projektmappe 96
  Version bei bestehendem Projekten ändern 91
  Version bei neuen Projekten auswählen 90
  was es ist 66
.NET Framework 3.5 Client Profile 93, 94
.NET Framework 4 Client Profile 93, 94
.Vstemplate-Dateien 84
/= 57
:IML
  Klassen, Umsetzung 381
+= Operator 56
<<-Operator 57
= 57
-= -Operator 56
>>-Operator 57

A

Abdocken (Fenster) 102
Abfangen von Fehlern 58
Abstrakte Klassen 409
Abstrakte Objekte (Klassen) 28
Accessor *siehe* Eigenschaften, Accessor
  unterschiedliche für Lesen/Schreiben (Eigenschaften) 367
Action(Of T) 587
Add 278
AddHandler 538
ADO.NET Entity Framework 809
AdventureWorks-Datenbanken 807
Aggregatfunktionen 784
Aggregieren und gruppieren 785
Aktionen (List (Of)) 657
Aktivierreihenfolge 146
Allgemeine Entwicklungseinstellungen 9, 77
AndNot 44
Andocken (Fenster) 102, 106
Anwendungen
  Mehrfachstart verhindern 741
Anwendungseinstellungen 736
  siehe Settings
  automatisch speichern 741
  Bereiche 738
  veränderliche 738
Anwendungsereignisse 742
  NetworkAvailabilityChanged 744
  Shutdown 743
  Startup 743
  StartupNextInstance 744
  UnhandledException 744
Anwendungsframework 740
  Aktivieren 741
  Authentifizierungsmodus für Benutzer 742
  Ereignisse *siehe* Anwendungsereignisse
  Herunterfahren, Modus 742
  Mehrfachstart verhindern 741
  Splash-Dialoge 742
  Windows XP-Stile verwenden 741
Any-Zieltyp 74

App.config
  Multizielversionen 95
Application-Objekt 204
Architektur-Explorer 112
Argumente *siehe* Parameter
ArrayList-Auflistung 628
  AddRange-Methode 628
  Arrays, umwandeln in 628
  Clear-Methode 628
  Count-Eigenschaft 628
  RemoveAt-Methode 628
  Remove-Methode 628
Arrays
  ArrayList, aus 628
  BinarySearch-Methode 612
  Dimensionen ändern 604
  durchsuchen 612
  Eigenschaften, wichtige 610
  Enumeratoren 618
  Grundsätzliches 602
  jagged 609
  konservieren, nach ReDim 606
  Konstanten 606
  Länge ermitteln 610
  Length-Methode 610
  mehrdimensional 609
  Methoden, wichtige 610
  Mutex-Objekte (Threading) 914
  Preserve-Anweisung 606
  ReDim-Anweisung 604
  Reihenfolge ändern, Elemente 611
  Reverse-Methode 611
  sortieren 611
  Sortierung, benutzerdefiniert 613
  Sort-Methoden 611
  suchen, Elemente, benutzerdefiniert 613
  verschachtelt 609
  Wertevorbelegung 606
ASCII umwandeln in Char-Datentyp 279
AscW-Methode 279
AsParallel 773
Assembly
  Erklärung 67
  in Projekt einbinden 68
Assembly (CTS) 365, 366
Assembly-Eigenschaft 700
AssemblyQualifiedName-Eigenschaft 700
Attached Properties (XAML) 213
Attribute 694
  AttributeUsage 697
  benutzerdefinierte ermitteln 709
  COMClass 697
  DllImport 697
  einfaches Beispiel 695
  Flags 558
  genereller Umgang 694
  MarshalAs 697
  Obsolete 695
  Serializable 697
  StructLayout 453
  VBFixedArray 697
  VBFixedString 697
  zur Laufzeit ermitteln 705
Attributes-Eigenschaft 700
AttributeUsage-Attribut 697
Aufbau von LINQ-Abfragen 764
Auflistungen
  ArrayList 628
  Enumeratoren 618
  For Each für benutzerdefinierte 618
  generische *siehe* Generische Auflistungen
  Grundsätzliches 622
  Hashtable 634
  IList-Schnittstelle 632
  in Relation setzen 778
  Initialisierer 626
  Items (ListBox) 393
  Key *siehe* Schlüssel, Abruf durch
  Load-Faktor 622
  Queue 648
  Schlüssel, Abruf durch 634
  SortedList 651
  Stack 650
  typsichere *siehe* Generische Auflistungen
  vorhandene 628
Auflistungen in Relation setzen 780, 782, 783
Aufzählungen 554
Ausdrücke 24
  boolesche 26
Ausdrücke, komplexe 25
Ausdruck-Lambdas 535
Ausdrucksbäume 535
Ausnahmen 58
  Behandeln mit Try/Catch/Finally 60
  Codeausführung sicherstellen 63
  globale Behandlung 744
  OutOfRangeException 263
  typabhängig behandeln 61
Aussageprüfer (List (Of)) 658
Autokorrektur für intelligentes Kompilieren 166
Automatisch implementierte Eigenschaften 341
  Standardwerte 342
Automatisches Vervollständigen *siehe* IntelliSense, Vervollständigungsliste
AutoScroll-Eigenschaft 144

B

Background Compiler 163
Base Class Library 72
BaseType-Eigenschaft 700
Basic 8
Bearbeitungshilfen *siehe* IntelliSense
Bedingte Programmausführung 39
Beendigungsmodus 115
Befehlszeilenargumente auslesen 726
Begleitdateien
  Softwarevoraussetzungen 6
Begrüßungsdialoge *siehe* Splash-Dialoge
Beispieldatenbanken (SQL Server) 807
Benutzer
  Authentifizierungsmodus 742
Benutzerdefinierte Event Handler 547
Beschränkungen 568
Bildschirm
  mehrere ~e nutzen 98, 102
  optimal nutzen 98

Binäre Suche (Arrays) 612
BinaryFormatter (Serialisierung) 669
BinarySearch 446
BinarySearch-Methode (Arrays) 612
Block-Gültigkeitsbereiche 54
Boolean-Datentyp 297
  numerische Äquivalente 297
  String, wandeln in 298
  Vergleiche 43
Boolesche Ausdrücke 26
Bottom-up-Programmierung 114
Boxing
  Details zum 467
  Schnittstellen, Besonderheiten 469
  Unboxing 467
  Wertetypen 463
Bucket (Hashtable) 647
ByRef 448
Byte-Datentyp 261
ByVal 448

C

Case-Anweisung 45
Casting 456
Catch 61
CByte 262
CDbl 267
CDec 268
Char-Datentyp 278
  Werte umwandeln in 279
CheckBox-Steuerelement
  ThreeState-Eigenschaft, Besonderheiten bei 583
ChrW-Methode 279
CInt 264, 457
CLI 71
Client Profile (.NET Framework) 93
CLng 265
Cloning *siehe* Klonen
CLR 71
  Execution Engine 72
  Garbage Collector 71
  mscoree.dll 72
CLS
  Compliance 260
Code
  als vertikale Bereiche visualisieren 112
  Ausführung bei Fehlern sicherstellen 63
  bearbeiten *siehe* Code-Editor
  Bearbeiten, Hilfe beim *siehe* IntelliSense
  Gliederungsfunktion 111
  Gültigkeitsbereiche durch Blöcke 54
  Klassencode über Dateien aufteilen 371
  mehrfache Teilstringsuche 108
  navigieren zu 107
  Refactoring 173
  Regionen 111
  Symbole suchen 107
  Umgestalten (Refactoring) 173
  Vererben von Klassencode 374
  Verweise suchen 107
  Wiederverwandung 374
  wiederverwenden 405
  Zugriff auf Settings-Variablen 182
  zwischen Methoden navigieren 108
Code Snippets
  Verwenden 176
Codeausschnitte *siehe* Code Snippets
Codedateien
  verteilte, für eine Klasse 371
Codeeditor
  abstrakte Klassen, Unterstützung für 424
  Ereignisbehandlungsroutinen, Unterstützung bei 515
  Fehler, Unterstützung beim Korrigieren von 370
  IntelliSense 358
  Interfaces, Unterstützung für 419
  Klassen, Unterstützung für abstrakte 424
  Schnittstellen, Unterstützung für 419
  Smarttags, Verwendung von 370
  Variablenzuweisung mit Konstanten 30
Code-Editor 157
  Anzeigen von 159
  Autokorrektur für intelligentes Kompilieren 166
  automatisches Codeeinrücken 162
  automatisches Vervollständigen *siehe* IntelliSense, Vervollständigungsliste
  Background Compiler 163
  Codeausschnitte *siehe* Code Snippets
  Dokumentationskommentare, benutzerdefiniert 167
  Fehlerbehebung bei Laufzeitfehlern 163
  Fehlererkennung 163
  IntelliSense *siehe* IntelliSense
  Kennen lernen am Beispiel 122
  Laufzeitfehler beheben 163
  neue Codedateien anlegen 171
  Prozeduren-Beschreibungen, benutzerdefiniert 167
  Refactoring 173
  Smarttags 165
  Umgestalten von Code (Refactoring) 173
  XML-Kommentare, benutzerdefiniert 167
CollectionBase-Auflistung
  Add-Methode (Implementierung) 631
  Elementzugriff über List 633
  IList 633
  Item-Eigenschaft (Implementierung) 631
  List 633
Collections
  Initializers 626
COMClass-Attribut 697
Common Language Infrastructure 72

Common Language Runtime 71
Common Type System 73
Comparer-Klasse 614
Conceptual Model 810
Concurrency-Checks
  LINQ to Entities 836
Constraints 568
Container für Steuerelemente 129
ContainerControl 129
ControlDesigner 130
Convenience-Patterns 498
Convert.ToByte 262
Convert.ToDecimal 268
Convert.ToDouble 267
Convert.ToInt16 263
Convert.ToInt32 264
Convert.ToInt64 265
Convert.ToSByte 262
Convert.ToSingle 266
Convert.ToUInt32 265
Convert.ToUInt64 266
Convert-Klasse 457
Covers
  Begleitdateien 125
  kennen lernen der Entwicklungsumgebung, mit 122
Covers: 123
CSByte 262
CShort 263
CSng 266
CTS 73
CType 457
  für eigene Klassen/Strukturen 506
CUInt 265
CULng 266
CultureInfo-Klasse 275
CurrentSettings-<Datum>.vssettings 78
Custom Event Handler 547

D

Date-Datentyp 298
  ParseExact-Methode 303
  Parse-Methode 303
  rechnen mit 299
  String, umwandeln in 303
Dateioperationen (My-Namespace) 733
Dateioperationen programmieren 720
Datentypen
  Berechnungen durch Prozessor 258
  Boolean *siehe* Boolean-Datentyp
  Byte *siehe* Byte-Datentyp
  Char *siehe* Char-Datentyp
  CLS-Compliance 260
  Date *siehe* Date-Datentyp
  Decimal *siehe* Decimal-Datentyp
  deklarieren und definieren 257
  Double *siehe* Double-Datentyp
  Einführung in 256
  Ermitteln des Typs 699
  generisch, benutzerdefiniert *siehe* Generische Datentypen
  Integer *siehe* Integer-Datentyp
  Long *siehe* Long-Datentyp
  Nullable 580
  null-bar (Wertetypen) 544
  numerische 256
  numerische in String wandeln 272
  numerische, Rundungsfehler 269
  numerische, Überblick über 261
  primitive 256
  Rundungsfehler 269
  SByte *siehe* SByte-Datentyp
  Short *siehe* Short-Datentyp
  Single *siehe* Single-Datentyp
  String *siehe* String-Datentyp
  TimeSpan 299
  ToString 274
  Typermittlung 699
  Typparameter 566
  UInteger *siehe* UInteger-Datentyp
  ULong *siehe* ULong-Datentyp
  umwandeln 272
  Umwandlungsversuch 277
  UShort *siehe* UShort-Datentyp
DateTime 458 *siehe* Date-Datentyp
DateTimeStyle 304
DateTimeStyles.AdjustToUniversal 304
DateTimeStyles.AllowInnerWhite 304
DateTimeStyles.allowLeadingWhite 304
DateTimeStyles.AllowTrailingWhite 304
DateTimeStyles.AllowWhiteSpaces 305
DateTimeStyles.NoCurrentDateDefault 305
DateTimeStyles.None 305
Datum
  berechnen 299
  Darstellungsstile 304
  Umwandlungsoptionen 304
Debug-Klasse 483
Debug-Konfigurationseinstellungen 186
Decimal-Datentyp 267
  Add 278
  Floor-Funktion 278
  Negate-Funktion 278
  Remainder-Funktion 278
  Round-Funktion 278
  spezielle Funktionen 278
  Truncate-Funktion 278
Deklaration, Variablen 20
Delegaten 525
  Action(Of T) 587
  an Methoden übergeben 531
  Function(Of T) 590
  Generelle Funktionsweise 525
  generische 586
  Lambdas 535
  Predicate 755
  signaturkompatibel 529
  Tuple(Of T) 590
  Variablen 528
Deserialisierung 669

Designer
ControlDesigner 130
Funktionen von Steuerelementen 130
für Formulare, Funktionsweise 130
Größe von Steuerelementen 130
Guidelines 129
Kennen lernen am Beispiel 122
Layoutfunktionen für Formulare 155
Layoutmodus für Formulare 130
Position von Steuerelementen 130
Raster in Formularen 130
Referenzsteuerelement beim Anordnen 131
Smarttags für Steuerelemente 132
Steuerelemente dynamisch anordnen 133
Steuerelemente positionieren 128
Steuerelemente, Aufgaben, häufige 132
Steuerelemente, Randabstände 129
Tastenkürzel 157
Designer Host 130
Diamant-Führungssymbol 102, 106
Dim 257
DirectoryInfo-Klasse 720
DirectX 198
Dispose 472
Dispose 484
DllImport-Attribut 697
Dokumentationskommentare, benutzerdefiniert 167
Dominierende Typen 608
DotNetCopy 714
/AutoStart-Option 719
/Silent-Option 719
Bedienung 715
Funktionsweise, Abstrakt 721
Kopierverhalten einstellen 718
Optionen 719
Double-Datentyp 267
Parse-Methode 272
vergleichen 272
Down Targeting (Multizielversionen) 95
Drill-down-Funktion 112
Duotrigesimalsystems 443
Durchschnittsberechnung 784

E

Eager Loading 823
Editieren, Code *siehe* Code-Editor
Editor *siehe auch* Code-Editor
Zeilenfortsetzungszeichen 161
EDM 798
Eigenschaften
Accessor 338, 339, 381
Accessor, unterschiedliche für Lesen/Schreiben 367
angehängte (XAML) 213
auslesen 339
automatisch implementierte 341
Default 343
definieren 337
definieren (Wertezuweisung) 338
ermitteln 339
Get-Accessor 339
Leselogik definieren 339
Let-Anweisung 343
Margin 129
MustOverride 411
Padding 129
Parameter überladen 364
Parametern, mit 342
schreiben, Werte 338
Schreiblogik definieren 338
Set-Accessor 338
Set-Anweisung 343
Standardeigenschaften 343
statische 371
überladen 364
Überschatten 432
überschreiben 386
Überschreiben (Polymorphie) *siehe* Polymorphie
Überschreibungszwang 411
Variablen, Vergleich zu öffentlichen 345
virtuelle 409
Werte auslesen 339
Werte, zuweisen an 338
Wertzuweisung 338
Eigenschaftenfenster
Selektieren von Steuerelementen 146
ElseIf-Anweisung 39
Empty (EventArgs) 523
End Select-Anweisung 45
Entitätsmodell 798
Entity Data Model
abfragen 818
Conceptual Model 810
CSDL-Datei (Conceptual Model) 810
Designer 815
Entitätscontainernamen ändern 816
erstellen 811
Funktionsweise 809
konzeptionelles Modell 810
Mapping 810
MSL-Datei (Mapping) 810
physisches Speichermodell 810
POCO 830
Praxisbeispiel 811
SSDL-Datei (Storage Model) 810
Storage Model 810
Entity Framework 809
ändern, Daten 830
Concurrency-Checks 836
Datenänderungen übermitteln 831
einfügen, Datensätze 832
kaskadiertes Löschen 834
löschen, Datensätze 834
SaveChanges 831
Schreibkonfliktprüfung 836
SQL Profiler verwenden 820
Entity SQL (eSQL) 820

Entwicklungseinstellungen
allgemeine 77
auf Standard zurücksetzen 78
auswählen 77
exportieren 78
importieren 78
Profile 77
sichern 78
Visual Basic 77
Entwicklungsumgebung
Konfigurationseinstellungen 186
Entwurfszeit-Kompatibilität 95
Enum
Praxisbeispiel 403
Zahlen-Datentyp, konvertieren aus 557
Enumeratoren 618
benutzerdefiniert 619
Enums 554
direkte Wertebestimmung 555
Dubletten 555
Elementtyp ermitteln 556
Flags 558
Flags, Abfrage von 559
Kombinationen, Abfrage von 559
Kombinieren von Werten 558
konvertieren 556
String-Datentyp, konvertieren in 557
Equals-Methode 427, 431, 644
Ereignisparameter 521
EventArgs 523
EventArgs.Empty 523
leere 523
Sender 521
Ereignisse 514
AddHandler 538
Anwendungen, für *siehe* Anwendungsereignisse
auslösen 518
auslösen, durch Überschreiben von Onxxx 519
behandeln (Handles) 515
benutzerdefinierte Event Handler 547
benutzerdefinierte, Beispiel 515
dynamische Einbinden 538
Ereignisbehandlungsroutinen 515
feuern (ugs.) 518
Handles 515
Objekte zum Konsumieren deklarieren 515
Onxxx überschreiben 519
Parameter *siehe* Ereignisparameter
RaiseEvent 519
RemoveHandler 538
verdrahten 515
WithEvents 515
Erl (VB6) 58
Erweiterungs-Manager 119
Erweiterungsmethode
kombinieren von mehreren 760
Select 756
vereinfachte Anwendung 761
Where 755
Erweiterungsmethoden 595
Erzwungene Typsicherheit 167
Event Handler 547
EventInfo-Klasse 704
Events *siehe* Ereignisse
fire (col.) *siehe* Ereignisse, auslösen
raise *siehe* Ereignisse, auslösen
Exception
OutOfRange 263
Exceptions 60
Codeausführung sicherstellen 63
typabhängig behandeln 61
Execution Engine 72
Express Edition
Visual Basic 2008 2008 6
Extended Application Markup Language *siehe* XAML 205
Extension Methods 595

F

False 297
Family (CTS) 365, 366
FamilyOrAssembly (CTS) 365, 366
Faustregeln für LINQ-Abfragen 776
Fehler 58
Abfangen mit Try/Catch 60
Behandeln 58
Codeausführung sicherstellen 63
typabhängig behandeln 61
Fehler abfangen
global 744
Feldvariablen
Speicherfolge in Strukturen 453
Fenster
abdocken 102
andocken 102, 106
Diamant-Führungssymbol 102, 106
horizontal anordnen 101
teilen 98
verschieben 105
vertikal anordnen 101
Feuern, Ereignisse (ugs.) 518
FieldInfo-Klasse 704
FIFO-Prinzip 648
FileInfo 248
FileInfo-Klasse 720
Finalize 472, 479
Finalize-Methode 431
Finally 63
flaches Klonen 674
Flags-Attribut (Enums) 558
Flags-Enum (Flags-Aufzählungen) 558
Floating Windows Style 105
FlowLayoutPanel 134
FolderBrowserDialog 242
For Each-Anweisung
Auflistungen, für benutzerdefinierte 618
For/Next-Schleifen
Fließkommazahlen 271
Format Provider 274
Formatieren, Zahlenwerte 402
Formatzeichenfolgen 515
Forms
FolderBrowserDialog 242
Formulare
AcceptButton 151
Aktivierreihenfolge von Steuerelementen 146

Formulare *(Fortsetzung)*
Anwendungsframework, Hilfe durch 740
automatisches Scrollen 143
AutoScroll-Eigenschaft 144
CancelButton 151
Eigenschaften in Settings speichern 183
gestalten *siehe* Designer
Größe von Steuerelementen 130
Hinzufügen zu Projekten 151
instanziieren, ohne 724
Klassennamen-Anpassung beim Umbenennen 175
Layoutfunktionen 155
Layoutmodus 130
Rasterung im Designer 130
Schnellzugriffstasten 151
Splash-Dialoge 742
Steuerelemente positionieren *siehe* Designer, Steuerelemente positionieren
Tabulatorreihenfolge von Steuerelementen 146
Tastenkürzel für Layout von Steuerelementen 157
Formulare: 130
Framework
Aufbau 74
Framework Class Library 72
Friend 365, 366
FullName-Eigenschaft 700
Function 356
Function(Of T) 590
Funktionen 20
optionale Parameter 359
Signatur 357
überladen 357

G

Garbage Collector 71, 474
Generationen 476
GC 474
GDI
Funktionsprinzip 195
Generate From Usage 117
Generationen 476
Generics
Typrückschluss 757
Generische Auflistungen 653
Collection(Of) 654
Dictionary(Of) 654
KeyedCollection(Of) *siehe* KeyedCollection(Of)-Auflistung
LinkedList(Of) \t 663
List(Of) 655
Queue(Of) 655
ReadOnlyCollection(Of) 654
SortedDictionary(Of) 655
SortedList(Of) 655
Stack(Of) 655
Generische Datentypen 562
beschränken auf Wertetypen 576
Beschränkungen 568
Beschränkungen kombinieren 577
Constraints 568
JITter, Umsetzung durch 566
Konstruktorbeschränkungen 576
Notwendigkeit, Beispiel 563
Schnittstellenbeschränkungen 573
Typparameter 566
Gestalten von Formularen *siehe* Designer
Get-Accessor (Eigenschaften) 339
GetCustomAttributes-Methode 700
GetEvent-Methode 701
GetEvents-Methode 701
GetField-Methode 701
GetFields-Methode 701
GetHashcode-Methode 644
GetHashCode-Methode 431
GetMember-Methode 701
GetMembers-Methode 701
GetProperties-Methode 701
GetProperty-Methode 701
GetType
Methode, als 699
GetType-Methode 431
Gleichzeitigkeit 856
Gliederungsfunktion (Code) 111
Grid 222
Group By-Methode 781
Gruppieren in LINQ-Abfragen 781
Gruppieren und aggregieren 785
Guidelines 129
Gültigkeitsbereiche
Block-Gültigkeitsbereiche 54
lokale Variable 54

H

Handles 240
Handles-Anweisung 515
Hashtable-Auflistung 634
Aufzählungen in 647
Bucket 647
DictionaryEntry 647
enumerieren 647
For Each für 647
Funktionsweise 641
Load-Faktor 641
Schlüssel-Klassen, benutzerdefiniert 644
Verarbeitungsgeschwindigkeit 637
Zugriffszeiten 640
Hintergrunderkennung von Fehlern 163

I

IComparable-Schnittstelle 614
IDE 76
Bildschirm optimal nutzen 98
Diamant-Führungssymbol 102, 106
erweitern 118
Fenster abdocken 102
Fenster andocken 102, 106
Fenster horizontal anordnen 101
Fenster teilen 98
Fenster verschieben 105
Fenster vertikal anordnen 101
floating Windows Style 105
mehrere Bildschirme nutzen 98, 102
neue Features 97
Profil auswählen 77
Speicherung des Fensterlayouts 103

IEnumerable-Schnittstelle 619
If … Then … Else-Anweisung 39
If-Operator 46
IList-Schnittstelle 632
ImageResizer 216
IML 73, 352
  Code untersuchen 352
  Codeanalyse 355
  Umsetzung in nativen Code 352
Implementierungsvorschriften *siehe* Schnittstellen
Implements 415
Implizite Konvertierung 415
Implizite Zeilenfortsetzung 161
Imports 274
Imports-Anweisung 70
Infinity 276
Infodialog (beim Starten einer Anwendung) *siehe* Splash-Dialoge
Inheritance 374
Inherits 403
INI-Dateien, Ersatz für 736
Initialisierer für Auflistungen 626
Instanziieren
  Wertetypen 449
Instanziieren von Klassen 29
Instanziieren von Klassen, Erklärung 322
Instanziieren von Objekten 347
Instanziierung
  Objektspeicher 326
  Speicher für Objekte 326
InStr-Funktion 287
InStrRev-Funktion 287
Int16 *siehe* Short-Datentyp
Int32 *siehe* Integer-Datentyp
Int64 *siehe* Long-Datentyp
Integer 457
Integer-Datentyp 264
  Enum, konvertieren in 557
  Parse-Methode 272
IntelliSense 358
  abstrakte Klassen, Unterstützung für 424
  automatisches Codeeinrücken 162
  automatisches Vervollständigen 160
  Auto-Vervollständigungsliste 160
  Beendigungsmodus 115
  Bottom-up-Programmierung 114
  Code-Editor, Fehlererkennung 163
  Codeeinrückungen 162
  Dokumentationskommentare, benutzerdefiniert 167
  Erst anwenden-Modus 115
  Erst deklarieren-Modus 115
  halbtransparent darstellen 116
  Interfaces, Unterstützung für 417, 419
  Klassen, Unterstützung für abstrakte 424
  Kurzbeschreibungen 160
  Methoden, Signaturen von 358
  Parameterinfo 160
  Parameterinfo, mehrzeilige Befehlszeilen 161
  Pascal Case 116
  Schnittstellen, Unterstützung für 417, 419
  Signaturanzeige von Methoden 358
  Teilstringfilterung 115
  testgesteuerte Entwicklung 114
  Überschreiben von Methoden, Hilfe bei 389
  Vervollständigen von Schlüsselworten 162
  Vervollständigungsliste 160
  Vervollständigungsliste filtern 160
  Vorschlagsmodus 115
Interfaces *siehe* Schnittstellen
Intermediate Language *siehe* IML
Invariant Culture 275
IsNot-Operator 427
IsNot-Operator, praktische Tipps 430
Is-Operator 427
Is-Operator, Praktische Tipps 430
Items-Auflistung (ListBox) 393

J

Jagged Arrays 609
JIT-Compiler 207
JITter 71, 73, 352
  generische Datentypen, Umsetzung von 566
Just in Time-Kompilierung 71

K

Kaskadiertes Löschen (LINQ to Entities) 834
Key (Auflistungen) *siehe* Schlüssel (Auflistungen)
KeyedCollection(Of)-Auflistung Serialisierungsprobleme 688
Klassen
  .cctor 354
  .ctor 354, 381
  abstrakt, als solche definieren 410
  abstrakte 409
  Analysieren 700
  Basisklasse erreichen 364
  CLS-Compliance 260
  Code über mehrere Codedateien aufteilen 371
  Convert 457
  DirectCast 461
  Dispose 472, 484
  Ereignisse *siehe* Ereignisse
  Ereignisse verknüpfen 515
  Ereignisse, Instanziieren zum Konsumieren von 515
  Ereignisse, Parameter für 523
  Ermitteln des Typs 699
  EventArgs 523
  explizite Konvertierung, implementieren 506
  Finalize 472
  generisch, benutzerdefiniert *siehe* Generische Datentypen
  geteilte (Partial) 371
  Hierarchie unterbrechen 433

Klassen *(Fortsetzung)*
IML, Umsetzung 381
Implementierungsvorschriften *siehe* Schnittstellen
Implements 415
implizite Konvertierung, implementieren 506
Instanzen, prüfen auf Gleichheit 427
instanziieren 29
Instanziieren 322, 324, 347
Interfaces *siehe* Schnittstellen
Kommunikation zwischen *siehe* Ereignisse
Konstruktor 347
Konstruktoren überladen 357
Konstruktoren, parametrisierte 382
Konstruktorzwang 351
Managed Heap 326
Me 364, 408
Member über Schnittstellen erreichen 634
Methoden 356
Methoden überschreiben 386
Modul 437
MustInherit 410
MyBase 364, 408
MyClass 408
New 364
Object 427
Operatoren, benutzerdefinierte *siehe* Operatorenprozeduren
Overridable 386
Overrides 386
Polymorphie 375 *siehe* Polymorphie
Prinzip als Analogie 322
private Member, indirekt zugreifen auf 634
Random 464
Schnittstellen *siehe* Schnittstellen
Schnittstellen einbinden, mehrere 426
Schnittstellen implementieren 415
serialisieren 666
Singleton 438
Speicher für Objekte 326
Speichern von Inhalten 666
Standardkonstruktor 351, 381
Sub New 347, 357
Type 699
Typermittlung 699
Vererbung 374
Wiederverwendung durch Vererbung 374
Klassendiagramme 114
Klonen 674
Kompilieren
Just in Time 71
Kompilierte Abfragen 829
Komplexe Ausdrücke 25
Konfigurationsdateien 736
Konsolenanwendungen 10, 346
Konstanten 30
Konstruktoren
.cctor 354
.ctor 354
aus Konstruktoren aufrufen 363
automatisch durch VB 381
erzwingen in Generischen Datentypen 576
Notwendigkeit 351
Parameter 349
parametrisierte 382
Standardkonstruktor 381
Strukturen, bei 449
Syntaxfehler 364
überladen 357
überladene aufrufen 363
Überschreiben (Polymorphie) *siehe* Polymorphie
Konvertieren
Arrays zu ArrayList 628
Boolean in String 298
Byte-Datentyp, in 262
Date in String 303
Datum in Zeichenkette 303
Decimal-Datentyp, in 268
DirectCast 461
Double-Datentyp, in 267
Enums 556
Erweitern von Typen 507
Fehler bei numerischen Typen 269
Integer-Datentyp, in 264
Long-Datentyp, in 265
Parse 460
ParseExact 460
SByte-Datentyp, in 262
Short-Datentyp, in 263
Single-Datentyp, in 266
String in Boolean 298
String in Date 303
UInteger-Datentyp, in 265
ULong-Datentyp, in 266
UShort-Datentyp, in 263
Verkleinern von Typen 507
versuchen 277
versuchen (TryParse) 460
Wert in Zeichenkette 272
Zeichenkette in Datum 303
Zeichenkette in Datum (aus Formatvorlagen) 303
Zeichenkette in Wert 272
Konvertierung, von Datentypen 456
Konvertierungen
explizite, implementieren benutzerdefiniert 506
implizite, implementieren benutzerdefiniert 506
Konvertierungs-Assistent 85, 86, 87
Konvertierungsprotokoll 86
Kultur, nicht bestimmt 275
Kulturabhängige Formate 274
Kurzschlussauswertung 44
Kurzschlussauswertungen 44

L

Ladenverhalten (Entity Framework) 823
Lambda-Ausdrücke in LINQ-Abfragen 754
Lambdas 534
Ausdruck-Lambdas 535
Ausdrucksbäume 535
Delegaten 535
mehrzeilige Anweisungs-Lambdas 535
Lazy Loading 823
Left-Funktion 286
Len-Funktion 286
Length-Methode (Arrays) 610
Let-Anweisung (Variablenzuweisung) 343

LIFO-Prinzip 650
Like-Operator 43
LinkedList(Of)-Auflistung 663
AddAddFirst-Methode 663
AddAfter-Methode 663
AddBefore-Methode 663
AddLast-Methode 663
FindLast-Methode 663
Find-Methode 663
First-Eigenschaft 663
Last-Eigenschaft 663
RemoveFirst -Methode 663
RemoveLast -Methode 663
Remove-Methode 663
LINQ
Abfragen gezielt ausführen 777
Abfragen kombinieren 760, 771
Abfragen parallelisieren 773
Aggregatfunktionen 784
anonyme Typen 757
Anonyme Typen 757
Aufbau einer Abfrage 764
Auflistungen gruppieren 781
Auflistungen in Relation setzen 778, 780, 782, 783
Auflistungen mit From verbinden 778
Auflistungen verbinden 778
Elemente von Abfrage trennen 777
Ergebnisse mehrere Aggregatfunktionen 785
Faustregeln für Abfragen 776
generelle Funktionsweise 751
Group By-Methode 781
Gruppieren 781
Join, Explizit 780
Join, implizit 778
Kaskadierte Abfragen 771, 776
Kombinieren von Gruppierungen und Aggregaten 785
Lambda-Ausdrücke 754
Prädikate 755
Relation von Auflistungen 778, 780, 783
Relation von gruppierten Auflistungen 782
Select-Erweiterungsmethode 756
to Objects 764
ToArray-Methode 777
ToDictionary-Methode 777
ToList-Methode 777
ToLookup-Methode 777
verzögerte Ausführung 768
Verzögerte Ausführung 771
Where-Erweiterungsmethode 755
LINQ to Entities 798
abfragen, EDM 818
ändern, Daten 830
Conceptual Model 810
Concurrency-Checks 836
Designer, EDM 815
Eager-Loading 823
EDM 798
EDM erstellen 811
einfügen, Datensätze 832
Entitätsmodell 798
Entity SQL 820
kaskadiertes Löschen 834
Kompilierte Abfragen 829
konzeptionelles Modell 810
Lazy Loading 823
löschen, Datensätze 834
Objektkontext 818
physisches Speichermodell 810
POCO 830
Praxisbeispiel 811
SaveChanges 831
Schreibkonfliktprüfung 836
SQL Profiler verwenden 820
Storage Model 810
T-SQL-Umsetzung 821
Verwalterklasse 818
LINQ to Objects
Einführung 764
List(Of) 655
List(Of)-Auflistung
Aktionen (Actions) 657
Aussageprüfer (Predicates) 658
For Each-Ersatz 657
Sortierungssteuerung-Ersatz 658
Suchsteuerung 658
Vergleicher (Comparisons) 658
ListBoxItem 430
ListBox-Steuerelement 392
ausgewählter Eintrag 430
Eintrag entfernen 430
Elemente identifizieren 430
Items-Auflistung 393
Remove 430
SelectedItem 430
selektiertes Element 430
Listen 603
Load-Faktor 622, 641
Local Type Inference 35
Lokale Typrückschlüsse 35
Lokalisieren *siehe auch* Ressourcen
Long 457
Long-Datentyp 265
Enum, konvertieren in 557
Parse-Methode 272
Look and Feel von Windows XP 741
LSet-Funktion 287
LTrim-Funktion 290

M

Main-Methode 16
Managed Heap 326
Mapping (Entity Data Model) 810
Margin-Eigenschaft 129
MarshalAs-Attribute 697
Maximalwerte 276
Maximum ermitteln 784
MaxValue 276
Me 364, 408
mehrzeilige Anweisungs-Lambdas 535
MemberInfo-Klasse 704
MemberwiseClone-Methode 431
Menu 238
Methode
mit Rückgabewerten 19
Methoden 356
Basisklassenmethoden überschreiben 386
Ersetzen 386

Methoden *(Fortsetzung)*
MustOverride 411
navigieren zwischen 108
optionale Parameter 359
Signatur 357
überladen 357
Überschatten 432
Überschreiben (Polymorphie) *siehe* Polymorphie
überschreiben in Basisklasse 386
Überschreibungszwang 411
virtuelle 409
MethodInfo-Klasse 704
Microsoft Office-Projekte
auf Visual Studio 2010 aktualisieren 86
Mid-Funktion 286
Minimalwerte 276
Minimum ermitteln 784
MinValue 276
Modale Formulare
instanziieren, ohne 724
Modifizierer *siehe* Zugriffsmodifizierer
Narrowing 507
Widening 507
WithEvents 515
Module 437
Monitor *siehe* Bildschirm
Monitor-Klasse (Threading) 906
mscoree.dll 72
MSIL-Code 207
Müllabfuhr 474
Multi-targeting *siehe* Multizielversionen
Multitasking
preemptive 904
Multizielversionen 87
App.config-Datei 95
bei bestehenden Projekten ändern 91
Down Targeting 95
Entwurfszeit-Kompatibilität 95
für neue Projekte festlegen 90
gemischte Ziele 96
Grenzen 95
Profile 93
Roundtrip 95
Steuerelemente 97
Unterschiede bei den Clientprofilen 94
MustInherit 410
MustOverride 411
Mutex
Einsatz bei Singleton-Klassen 440
Mutex-Klasse 911
Array 914
Mutual Exclusion *siehe* Mutex
My.Application 723, 726
My.Computer 723
Dateioperationen 733
FileSystem 733
My.Forms 723
My.Resources 723
My.Settings 723
Anwendungseinstellungen 736
My.User 723
My.WebServices 723
MyBase 364, 408
MyClass 408
My-Namespace
Anwendungsbeispiel 714
Anwendungseinstellungen 736
Dateioperationen 733
Formulare, ohne Instanzen 724
Hintergrund 722
Ressourcen 727
Übersetzen in andere Sprachen 730

N

Namespace
Erklärung 67
verwenden 68
Narrowing-Modifizierer 507
Navigieren zu-Feature 107
NetworkAvailabilityChanged - Ereignis (Anwendungsframework) 744
Neue Codedateien anlegen 171
Neuen Typ generieren-Feature 117
Neues Projekt erstellen 10
New 364
New, Sub
Konstruktoren aufrufen, weitere 363
MyBase 364
überladen 357
New, Sub in Klassen 347
Nichts 329
Nothing 329
als Standardwert 586
Standardwerte-Zuweisung 330
Nullable
Besonderheiten 583
Nullable(Of) 544
Nullable-Datentypen 580
Boxing, Besonderheiten 584
null-bare Wertetypen 544

O

O/R-Designer
Entity Data Model 815
Object 427
Eigenschaften 431
Equals 427, 431
Equals, praktische Tipps 430
Finalize 431
GetHashCode 431
GetType 431, 699
MemberwiseClone 431
Methoden 431
ReferenceEquals 431
Sender bei Ereignissen, Träger der Quelle 521
ToString 431
Objekte 27, 325
ableiten 28
abstrakte 28
Analysieren 700
DeepCopy 677
Deklarieren mit Ereignissen 515
entsorgen 472
Ermitteln des Typs 699
finalisieren 472
freigeben, automatisch 494
Gleichheit, prüfen auf 427
instanziieren 29, 347
klonen 674

Objekte *(Fortsetzung)*
Konstruktoren überladen 357
Managed Heap 326
Methoden von Object 427
Polymorphie *siehe* Polymorphie
serialisieren 666
Speichern von Inhalten 666
Speicherung 326
Type 699
Typermittlung 699
vergleichen 614
XML, speichern als 683
Zeiger 326
Objektkontext
LINQ to Entities 818
Obsolete-Attribut 695
Office-Projekte
auf Visual Studio 2010 aktualisieren 86
On Error GoTo, Ersatz für 59
Onxxx-Methoden *siehe* Ereignisse, Onxxx überschreiben
OOP
wichtiger Tipp zu 722
OpenFileDialog 244
Operatoren
AndAlso 44
benutzerdefinierte *siehe* Operatorenprozeduren
Bitverschiebungen 57
Boolesche 43
GetType 699
Kurzschreibweise 56
OrElse 44
Type Of 699
Überladen von Operatorenprozeduren 505
Vergleiche, für 43
Operatorenprozeduren 498
Einführung 498
Klassen vorbereiten für 499
Problembehandlung 510
Rechenoperatoren, implementieren 503
Signaturenmehrdeutigkeiten 511
Strukturen vorbereiten für 499
Typkonvertierungen, implementieren 506
überladen 505
Übersicht möglicher Operatoren 511
Vergleichsoperatoren, implementieren 506
Wahr/Falsch-Auswerter, implementieren 508
Option Strict 167, 359
Optionale Parameter 359
OrElse 44
OutOfRangeException 263
Overridable-Anweisung 386
Overrides-Anweisung 386

P

Padding-Eigenschaft 129
Page 213
Parallel LINQ 773
Parameter
Befehlszeile, auslesen 726
ByRef 448
ByVal 448
Eigenschaften, für 342
Ereignisse, für 523 *siehe auch* Ereignisparameter
EventArgs 523
Konstruktoren, in 382
Operatorenprozeduren überladen 505
überladen (Eigenschaften) 364
Pardofin 329
Parse 446, 458, 460
ParseExact 458, 460
ParseExact-Methode (Date-Datentyp) 303
Parse-Methode (Date-Datentyp) 303
Parse-Methode (numerische Werte) 272
Partial 371
Pascal Case
IntelliSense 116
navigieren zu 108
PictureBox-Steuerelement, Ausschnitte scrollen in 144
POCO 830
Polymorphie 375, 389
am Beispiel 398
Beispiel als Analogie 390
Beispiel, Praxisbeispiel 395
Beispiel, ToString 392
Einführung 389
ListBox 392
Me 408
MyBase 408
MyClass 408
ToString 392
Wiederverwenden von Code 405
PowerShell 799
Prädikate 755
Preserve-Anweisung 606
Primitive Datentypen 256
Byte 268
Decimal 269
Double 269
Integer 268
Konstanten 30
Long 268
Operatoren 56
SByte 268
Short 268
Single 268
Typliterale 30
Typsicherheit 30
UInteger 268
ULong 268
UShort 268
Private 365, 366
Profile im Sinne von Entwicklungseinstellungen *siehe* Entwicklungseinstellungen
.NET Framework-Profile 93
Programmcode bearbeiten *siehe* Code-Editor
Projekt
Assembly-Verweis 68
neues erstellen 10
starten 12
Projekte
auf neue Visual Studio-Version aktualisieren 85
Elemente hizufügen 83
Formular hinzufügen 151
Konfigurationseinstellungen 186
neu anlegen 126
neue erstellen 82

Projektmappen
Any-Zieltyp 74
gemischte .NET-Framework-Zielversionen 96
Konfigurationseinstellungen 186
Projektmappen-Explorer
neue Codedateien anlegen 171
Projektvorlagen 84
.Vstemplate-Dateien 84
ändern 84
eigene 84
organisieren 84
Property Extender 142
PropertyInfo-Klasse 704
Protected 365, 366
Protected Friend 365, 366
Public 365, 366
Pull-Methode 650
Push-Methode 650

Q

Queue-Auflistung 648

R

RaiseEvent-Anweisung 519
Random-Klasse 464
ReDim-Anweisung 604
Refactoring 173
ReferenceEquals-Methode 431
Reflection 694
Assembly-Klasse 700
AssemblyQualifiedName-Eigenschaft 700
Attribute ermitteln 705
Attribute ermitteln, benutzerdefiniert 709
Attributes-Eigenschaft 700
BaseType-Eigenschaft 700
Eigenschafteninformationen 704
Einführung 697
FullName-Eigenschaft 700
GetCustomAttributes-Methode 700
GetEvent-Methode 701
GetEvents-Methode 701
GetField-Methode 701
GetFields-Methode 701
GetMember-Methode 701
GetMembers-Methode 701
GetProperties-Methode 701
GetProperty-Methode 701
Regionen (Code) 111
Rekursion *siehe* Rekursion
Relation von Auflistungen 778, 780, 783
Relation von gruppierten Auflistungen 782
Release-Konfigurationseinstellungen 186
RemoveHandler 538
Remove-Methode (ListBox) 430
Replace-Funktion 287
Ressourcen
Abrufen 728
anlegen und verwalten 728
Internationalisieren 730
Ressourcen aufteilen (Threading) 911
Reverse-Methode 611
Right-Funktion 286
Roundtrip (Multizielversionen) 95
RSet-Funktion 287
RTrim-Funktion 290
Rundungsfehler 269

S

SByte-Datentyp 262
Schleifen 47
Continue 53
Do … Loop 51
Exit 53
For/Each 50
vorzeitig verlassen 53
vorzeitig wiederholen 53
While … End 51
Schleifenabbruchbedingungen 52
Schlüssel
benutzerdefinierter 644
Equals-Methode 644
GetHashcode-Methode 644
Unveränderlichkeit 647
Schlüssel (Auflistungen) 634
Schnellzugriffstasten 149
Schnittstellen 412
Anwendungsbeispiel 413, 414
Benennungskonventionen 414
Beschränkungen, auf (Generische Datentypen) 573
Boxing, Besonderheiten 469
explizite Member-Definition 634
Implementieren 415
implementieren in Klassen 415
Implementieren von Schnittstellen 425
private Member öffentlich machen 634
Schnittstellen, IntelliSense-Unterstützung 417
Schreibkonfliktprüfung
LINQ to SQL 836
Scope 54
Scroll-Funktionalität für Formulare 143
Select-Anweisung 45
SelectedItem-Eigenschaft (ListBox) 430
Select-Erweiterungsmethode 756
anonyme Typen 757
Sequenzdiagramme 113
Serialisierung 666
BinaryFormatter 669
Probleme bei KeyedCollection(Of) 688
SoapFormatter 669
Versionen 682
Versionsunabhängigkeit 688
XML 683
Zirkelverweise 680
Serializable-Attribute 697
Set-Accessor (Eigenschaften) 338
Set-Anweisung (Objektzuweisung) 343
Settings 180
im Code verwenden 182
Speicherort 185
Variablen einrichten 180

Settings *(Fortsetzung)*
Verknüpfen von Formularen, mit 183
Verknüpfen von Steuerelementen, mit 183
Shadows 432
Short-Datentyp 262
Enum, konvertieren in 557
Parse-Methode 272
Shutdown-Ereignis (Anwendungsframework) 743
Signaturen 357
Single-Datentyp 266
Parse-Methode 272
vergleichen 272
Singleton-Klassen 438
Smart Clients
Anwendungsframework 740
Beenden der Anwendung, Benachrichtigung 743
Ereignisse, globale *siehe* Anwendungsereignisse
Fehlerbehandlung, globale 744
Hauptthread 868
Herunterfahren, Modus 742
Internationalisieren 730
lokalisieren 727
Mehrfachstart verhindern 741
Netzwerkzustandsänderung, Benachrichtigung 744
Splash-Dialoge 742
Sprachen, andere 727
Starten der Anwendung, Benachrichtigung 743
Thread, erster 868
übersetzen in andere Sprachen 727
UI-Thread 868
Unterstützung durch Visual Basic 2010 740
Weiteres Starten der Anwendung, Benachrichtigung 744
Smarttags
Code-Editor, für 165
Codeeditor, Verwendung im 370
Steuerelemente, für 132
Snoop-Tool 98
SoapFormatter (Serialisierung) 669
Softwarevoraussetzungen 6
SortedList-Auflistung 651
Sortieren (Arrays) 611
Sort-Methode (Arrays) 611
Speicherverwaltung 326
Splash-Dialoge 742
SplitContainer 133
Split-Funktion 291
SQL Server 2008 Express
AdenventureWorks 807
Authentifizierungsmodus 804
Beispieldatenbanken 807
Dienstkonten 803
Firewall 802
PowerShell 799
Verwaltungstools 802
with Advanced Services 799
Stack-Auflistung 650
abladen (Pull-Methode) 650
aufladen (Push-Methode) 650
Standardeigenschaften 343
Standardkonstruktor 351, 381
Standardschaltflächen 151
Starten einer Visual Basic-Anwendung 12
Startseite 79
konfigurieren 80
Projekte 80
Startup-Ereignis (Anwendungsframework) 743
StartupNextInstance-Ereignis (Anwendungsframework) 744
Statement Lambdas 535
Steuerelemente
.NET Framework-Zielversionen 97
Aktivierreihenfolge 146
als Container 129
Anchor-Eigenschaft 133, 134
Anordnen in Tabellen 133
Anordnen in TableLayoutPanel 139
Anordnen in veränderbaren Bereichen 133
Anordnen mit Umbruchlogik 134
Aufgaben, häufige 132
Caption-Eigenschaft 148
Dock-Eigenschaft 133
dynamisch anordnen 133
Eigenschaften in Settings speichern 183
Erweitern vorhandener Eigenschaften 142
FlowLayoutPanel 134
Größe anpassen 130
Layoutfunktionen 155
ListBox *siehe* ListBox-Steuerelement
Name-Eigenschaft 148
Namenskonventionen im Buch 154
PictureBox 144
Positionieren, genaues 130
Property Extender 142
proportionales Vergrößern 136
Referenzsteuerelement beim Anordnen 131
Schnellzugriffstasten 149
Scrollen in Formularen/Containern 143
Seiten docken 133
selektieren, erweitertes 145
Smarttags 132
SplitContainer 133
TableLayoutPanel 133, 136
Tabulatorreihenfolge 146
Tastenkürzel für Layout 157
Text statt Caption 148
Text-Eigenschaft 148
Threading in 868
threadsicher 868
verankern 133, 134
Verankern in TableLayoutPanel 140
Steuerelemente positionieren 128
Storage Model 810
Strikte Typbindung 359
String$-Funktion 281
String.Intern-Methode 285
Stringbuilder-Klasse 293

String-Datentyp 279
Ähnlichkeit, prüfen auf 43
Boolean, aus 298
Carriage Return 283
Date, umwandeln in 303
deklarieren und definieren 280
Enum, konvertieren aus 557
Funktionen, neue in .NET 280
Geschwindigkeit beim Zusammenbauen 293
IndexOfAny-Methode 287
Intern-Methode 285
Iterieren durch Zeichen 292
Konstruktor 281
Länge ermitteln 286
Längen angleichen 287
Length-Methode 286
Like-Operator 43
numerische Werte wandeln, in 272
PadLeft-Methode 287
PadRight-Methode 287
Remove-Methode 287
Replace-Methode 287
Sonderzeichen 283
Speicherbedarf 284
Speicheroptimierung 285
Split-Methode 291
String.IndexOf-Methode 287
StringBuilder, Vergleich 293
Stringpool 285
SubString-Methode 286
Suchen und Ersetzen 287
Teile ermitteln 286
TrimEnd-Methode 290
trimmen 290
Trim-Methode 290
TrimStart-Methode 290
Unveränderlichkeit 284
vbBack 283
vbCr 283
vbCrLf 283
vbLf 283
vbNullChar 283
vbNullString 283
vbNwLine 283
vbTab 283
verketten, Performancetipps 282
Wagenrücklauf 283
Zeilenvorschub 283
zerlegen in Teile 291
Strings
konvertieren 458
parsen 458
StructLayout-Attribut 453
Structure 333, 442
Strukturen 442
CLS-Compliance 260
Ereignisse *siehe* Ereignisse
Ereignisse verknüpfen 515
Ereignisse, Instanziieren zum Konsumieren von 515
explizite Konvertierung, implementieren 506
generisch, benutzerdefiniert *siehe* Generische Datentypen
implizite Konvertierung, implementieren 506
Kommunikation zwischen *siehe* Ereignisse
Konstruktoren 449
Operatoren, benutzerdefinierte *siehe* Operatorenprozeduren
Stubs
generieren 117
Sub 356
Sub Main 16
Sub New 347
Konstruktoren aufrufen, weitere 363
MyBase 364
überladen 357
Sub- und Function-Methoden 20
Suchen
Fuzzy 107
mit dem Navigieren zu-Feature 107
Summenbildung in Abfragen 784
Symbole *siehe auch* Verweise
SyncLock-Methode 902
Syntaxfehler 364
System.Boolean (Datentyp) 297
System.Byte (Datentyp) 261
System.Char (Datentyp) 278
System.DateTime (Datentyp) 298
System.Decimal (Datentyp) 267
System.Double (Datentyp) 267
System.Globalization 274
System.Int16 (Datentyp) 262
System.Int32 (Datentyp) 264
System.Int64 (Datentyp) 265
System.SByte (Datentyp) 262
System.Single (Datentyp) 266
System.UInt16 (Datentyp) 263
System.UInt32 (Datentyp) 264
System.UInt64 (Datentyp) 265

T

TableLayoutPanel 133, 136
prozentuale und fixe Zellengrößen 138
Spalten einstellen 136
Steuerelemente anordnen, in 139
Steuerelemente verankern 140
Zeilen einstellen 136
Zellen verbinden 142
Tabulatorreihenfolge 146
Task-Manager 860
testgesteuerte Entwicklung 114
Then-Anweisung 39
Threading 856
Threads
Auslastung 860
Monitor-Klasse 906
Mutex-Klasse 911
Notwendigkeit für 861
Ressourcen aufteilen zwischen 911
Steuerelemente, in 868
SyncLock 902
UI-Thread 868
tiefes Klonen 674
TimeSpan-Struktur 299
ToString 274, 459
ListBox-Eintrag, Überschreiben für 392
Object, von 431
Polymorphie 392
ToString-Methode 402
Trace-Listener 483
Trim-Funktion 290
True 297

Try/Catch/Finally 60
TryParse 460
TryParse-Anweisung 277
Tuple(Of T) 590
Type
  Assembly-Eigenschaft 700
  AssemblyQualifiedName-Eigenschaft 700
  Attributes-Eigenschaft 700
  BaseType-Eigenschaft 700
  EventInfo-Klasse 704
  FieldInfo-Klasse 704
  FullName-Eigenschaft 700
  GetCustomAttributes-Methode 700
  GetEvent-Methode 701
  GetEvents-Methode 701
  GetField-Methode 701
  GetFields-Methode 701
  GetMember-Methode 701
  GetMembers-Methode 701
  GetProperties-Methode 701
  GetProperty-Methode 701
  MemberInfo-Klasse 704
  MethodInfo-Klasse 704
  PropertyInfo-Klasse 704
Type Of-Operator 699
Type-Klasse 699
Typen 699
  casten 456
  dominierend 608
  Ereignisse, WithEvents 515
  generisch, benutzerdefiniert *siehe* Generische Datentypen
  neue generieren 117
  null-bar 544
  Parameter für 566
  umwandeln 456
Typkonvertierung
  DirectCast 461
Typliterale 30
Typparameter 566
Typrückschluss
  generische Typparameter 757
Typsicherheit 30, 359
Typsicherheit erzwingen 167
Typumwandlungen 456
Typvarianz 591

U

Überladen 357
Überschatten von Prozeduren 432
Überschreiben 386
Überschreiben von Onxxx-Methoden *siehe* Ereignisse, Onxxx überschreiben
UInt16 *siehe* UShort-Datentyp
UInt32 *siehe* UInteger-Datentyp
UInt64 *siehe* ULong-Datentyp
UInteger-Datentyp 264
UI-Thread 868
ULong-Datentyp 265
Umgestalten von Code (Refactoring) 173
Umwandeln, von Datentypen 456
Umwandlungsversuch 277
Unboxing 467
Unendlich 276
UnhandledException-Ereignis (Anwendungsframework) 744
Unicode-Zeichen 279
UShort-Datentyp 263
Using-Anweisung 494

V

Variablen
  definieren 24
  Deklaration 20
  Eigenschaften, Vergleich zu öffentlichen 345
  globale, Initialisierungszeitpunkt 384
  Gültigkeitsbereiche 54
  Initialisierungszeitpunkt bei globalen 384
  Managed Heap 326
  öffentliche, Vergleich zu Eigenschaften 345
  Operatoren 56
  Option Strict 359
  Referenztypen 442
  Scope 54
  Speicherfolge in Strukturen 453
  strikte Typbindung 359
  Structure 442
  Typliterale 30
  typsicher 359
  Typsicherheit 30
  Verweise 326
  Werteverlust im Gültigkeitsbereich 368
  Zeiger 326
  Zuweisung durch Konstanten 30
Varianz bei Typen 591
VBFixedArray-Attribut 697
VBFixedString-Attribut 697
Vereinfachter Eigenschaftenzugriff 53
Vererbung 374
Vergleichen
  Double- oder Single-Datentyp 272
Vergleichen von Objekten 614
Vergleicher (List (Of)) 658
Vergleichsoperatoren 43
Versionsunabhängigkeit (Serialisierung) 688
Versuchte Typkonvertierung 277
Vervollständigungsliste *siehe* IntelliSense, Vervollständigungsliste
Verweis hinzufügen (Projekt) 68
Verweise
  hervorheben 108
  hervorheben, Farbe ändern 110
  suchen mit Navigieren zu-Feature 107
Verweistypen 325
Verweisvariablen 326
Verzögerte Ausführung, LINQ-Abfragen 768, 771
Vielgestaltigkeit *siehe* Polymorphie
Virtuelle Prozeduren 409
Visual Basic
  6.0-Projekte auf Version 2010 aktualisieren 87
  Anatomie einer Anwendung 14
  Anwendungsprojekt starten 12
  Main-Methode 16

Visual Basic 2010
Anwendungsframework 740
Visual Basic 6.0
Projekte auf Version 2010 aktualisieren 87
Visual Basic 6.0-Unterschiede
Eigenschaften, Default 343
Eigenschaften, Standard 343
Fehlerbehandlung 58
Kurzschlussauswertungen (AndAlso, OrElse) 44
On Error GoTo, Ersatz für 59
Operatoren 56
Visual Basic-Entwicklungseinstellungen 77
Visual Studio
aus früheren ~-Versionen auf ~ 2010 migrieren 85
der erste Start 8
erster Start 77
erweitern 118
Erweiterungs-Manager 119
IDE *siehe* Haupteintrag IDE
neue Features der IDE 97
Profil auswählen 77
Startseite *siehe* Haupteintrag Startseite
Umgebungskonfiguration 9
Versionen 6
Visual Studio Gallery 119
Visuelle XP-Stile 741
Vorlagen
siehe Projektvorlagen 76
Vorschlagsmodus 115
Vorschriften zur Klassenimplementierung *siehe* Schnittstellen
VSIX-Package-Format 120

W

Was ist .NET? 66
Werte
Zeichen umwandeln in 279
Wertetypen 332
Beschränkung, auf (Generische Datentypen) 576
Erstellen 333
klonen 334
Konstruktoren, bei 449
null-bar 544
Where-Erweiterungsmethode 755
Widening-Modifizierer 507
Window 204, 213, 219
Windows Presentation Foundation *siehe* WPF 190
Windows XP-Stile verwenden 741
With/End With 53
WithEvents-Anweisung 515
Wolfenstein 3D 198
WPF 190
Aufbau von Bildinformationen 197
Beachtenswertes 191
Beispiel für Funktionsprinzip 199
Button 206
Demo, eindrucksvolle 203
DirectX 198
Einführung 190
Ereignisse 240
Events 240
Fähigkeiten gegenüber GDI 201
Grid 222
Handles 240
ImageResizer 216
in Visual Studio-IDE 97
InitializeComponent 207
Loaded-Ereignis 209
Menu 238
neue Konzepte 191
OpenFileDialog 244
Rendering 197
Trennung von Design und Code 203
vs. GDI 192
Window 219
XAML 205
Write 483

X

XAML 205, 213
attached Properties 213
Eigenschaften des übergeordneten Objekts 213
Eigenschaften, angehängte 213
Page 213
Überblick 213
Window 213
XML 683
Abfragen mit LINQ 793
Einführung 788
Erstellen mit LINQ 791
IntelliSense-Unterstützung 794
Namespace 796
XAttribute 792
XDocument 793
XElement 792
XML-Ausdrücke
XML Literale 790
XML-Kommentare, benutzerdefiniert 167

Z

Zahlen
Zeichenketten wandeln, in 272
Zahlenkonvertierungsfehler 269
Zahlensysteme 269
Zahlenwerte
formatieren 402
Text, umwandeln in 402
Zeichen
Carriage Return 283
Datentyp für 278
Sonderzeichen 283
Unicode 279
verkettete 279
Wagenrücklauf 283
Werte umwandeln in 279
Zeilenvorschub 283
Zeichenketten *siehe* String-Datentyp
Zeigervariablen 326
Zeilenfortsetzungszeichen 161
Zeitdifferenzen 299
Zirkelverweise 680
Zufallszahlen 464
Zugriffsebenen *siehe* Zugriffsmodifizierer
Zugriffsmodifizierer 365
Ändern durch Schnittstellen 634
Assembly (CTS) 365, 366
Family (CTS) 365, 366
FamilyOrAssembly (CTS) 365, 366
Friend 365, 366
Private 365, 366
Protected 365, 366
Protected Friend 365, 366
Public 365, 366