3 Grundlagen der Sprache

Nachdem Sie in Kapitel 2 erfahren haben, wie Sie VB.NET-Programme grundsätzlich aufbauen und kompilieren, beschreibe ich in diesem Kapitel die Grundlagen der Sprache. Ich zeige, wie Sie Anweisungen schreiben, mit Datentypen umgehen, mit Schleifen und Verzweigungen arbeiten und Funktionen bzw. Prozeduren schreiben (Strukturen werden erst in Kapitel 4 behandelt). Um die Beispiele möglichst einfach zu halten, verwende ich dazu einfache Konsolenanwendungen.

3.1 Umgang mit Assemblierungen und Namensräumen

Wenn Sie die in einer Assemblierung enthaltenen Typen im Programm verwenden wollen, müssen Sie diese Assemblierung referenzieren. Die einzige Assemblierungen, die nicht referenziert werden muss, weil das der Compiler automatisch macht, ist *mscorlib.dll*. Diese Assemblierung enthält die grundlegenden Typen des .NET-Framework in verschiedenen Namensräumen. Wenn Sie mit Visual Studio kompilieren, müssen Sie zudem die Assemblierung *Microsoft.VisualBasic.dll* nicht referenzieren. *Microsoft.VisualBasic.dll* enthält spezielle Typen, die nur unter Visual Basic verwendet werden (wie beispielsweise die Funktion `IsNumeric`). Der Kommandozeilencompiler referenziert diese Assemblierung allerdings nicht automatisch.

In Visual Studio verwenden Sie zur Referenzierung anderer Assemblierungen den `Referenzen`-Eintrag im Projektmappen-Explorer, so wie ich es bereits in Kapitel 2 gezeigt habe. Wenn Sie den Kommandozeilencompiler verwenden, geben Sie die zu referenzierenden Assemblierungen im *reference*-Argument an.

Wenn Sie die in einem Namensraum enthaltenen Typen in Ihrem Programm verwenden wollen, können Sie diese unter Angabe des Namensraums voll qualifiziert angeben. Wenn Sie beispielsweise die Klasse `SqlConnection` verwenden wollen, um eine Datenbankverbindung zu einer SQL-Server-Datenbank aufzubauen, können Sie die benötigte Variable so deklarieren:

```
Dim con As System.Data.SqlClient.SqlConnection
```

Für einzelne Dateien können Sie Namensräume aber auch über eine `Imports`-Direktive, die ganz oben im Quellcode (allerdings unterhalb von eventuellen `Option`-Anweisungen) stehen muss, »importieren«:

```
Imports System.Data.SqlClient
```

Dann können Sie alle darin enthaltenen Typen ohne Angabe des Namensraums verwenden:

```
Dim con As SqlConnection
```

Falls der Compiler dann eine Kollision mit einem Typnamen eines anderen Namensraums oder einem Schlüsselwort anzeigt, können Sie diesen Typ immer noch voll qualifiziert (mit Namensraum) angeben.

Visual Studio importiert für VB.NET per Voreinstellung einige Namensräume für das gesamte Projekt. Diese Namensräume müssen Sie nicht in der `Imports`*-Anweisung angeben, damit Sie bei der Verwendung der enthaltenen Typen auf die Angabe des Namensraums verzichten können.*

Die folgenden Namensräume werden per Voreinstellung für das gesamte Projekt importiert: `Microsoft.VisualBasic`, `System`, `System.Collections`, `System.Data`, `System.Diagnostics`, `System.Drawing` und `System.Windows.Forms`. In den allgemeinen Projekteigenschaften können Sie die projektweit importierten Namensräume unter dem Eintrag IMPORTE bearbeiten. Hier können Sie natürlich auch neue Namensräume angeben.

Die `Imports`-Direktive erlaubt neben dem Importieren noch die Erstellung eines Alias für einen Namensraum oder eine Klasse:

```
Imports mb = System.Windows.Forms.MessageBox
```

Den Namensraum bzw. die Klasse können Sie dann über den Alias referenzieren:

```
mb.Show("Hello World")
```

Ich verwende diese Technik allerdings nicht, weil es dann schwer zu erkennen ist, welche Klasse bzw. welcher Namensraum tatsächlich verwendet wird.

Der Windows Class Viewer

Der Windows Class Viewer ist ein Tool, über das Sie die Namensräume des .NET-Framework nach Klassen mit einem bestimmten Namen durchsuchen können. Wenn Sie einmal nicht wissen, in welchem Namensraum eine Klasse verwaltet wird, suchen Sie einfach im Class Viewer danach. Der Class Viewer zeigt nicht nur den Namensraum, sondern auch die komplette Deklaration der Klasse an. Sie finden dieses Tool unter dem Namen *WinCv.exe* im Ordner *\Programme\Microsoft Visual Studio .NET\FrameworkSDK\Bin*.

3.2 Module und Klassen

Visual Basic.NET ist eigentlich eine vollkommen objektorientierte Programmiersprache. Sie können in VB.NET ausschließlich mit Klassen programmieren, so wie beispielsweise auch in C#. Ein Windows-Formular ist z. B. eine solche Klasse. VB.NET unterstützt aber auch klassische Module, in denen Sie einfache globale Funktionen, Prozeduren und Variablen unterbringen können. Diese einfachen Typen können Sie verwenden, ohne eine Instanz des Moduls zu erzeugen (was bei normalen Klassen notwendig wäre). VB.NET unterstützt damit die klassische strukturierte Programmierung. Ich empfehle Ihnen allerdings, möglichst auf Module zu verzichten. Die OOP besitzt gegenüber der strukturierten Programmierung einfach zu viele Vorteile. Die Programmierung von Klassen beschreibe ich in Kapitel 4. Wenn Sie in einem objektorientierten Programm einmal eine Funktion oder Variable benötigen, die global im Programm gilt, verwenden Sie statt einem Modul einfach eine Klasse mit statischen (in VB: `Shared`-)Elementen, wie ich es ebenfalls in Kapitel 4 beschreibe. Das ist wesentlich objektorientierter. Ich gehe auf Module also nicht weiter ein.

3.3 Anweisungen, Ausdrücke und Operatoren

3.3.1 Anweisungen

Elementare Anweisungen

In VB.NET werden Anweisungen nicht, wie in anderen Sprachen, abgeschlossen. Das Ende einer Anweisung ist durch das Ende der geschriebenen Zeile gekennzeichnet. Daraus folgt, dass Sie eine Anweisung nicht einfach umbrechen können und dass Sie in einer Zeile nicht direkt mehrere Anweisungen unterbringen können. Um Anweisungen in mehreren Zeilen zu umbrechen, müssen Sie einen Unterstrich an das Ende der Zeile hängen. Die Anweisung

```
brutto = netto * (1 + steuer / 100)
```

können Sie z. B. so umbrechen:

```
brutto = netto * _
   (1 + steuer / 100)
```

Wollen Sie eine Zeile mitten in einer Textkonstante umbrechen, können Sie *nicht* einfach nur den Unterstrich verwenden:

```
MessageBox.Show("Das ist ein nicht funktionierender _
   Versuch mit dem Umbruch einer Textkonstante")
```

Bei Textkonstanten müssen Sie dem Compiler mitteilen, wo die Textkonstante endet und wieder beginnt. Addieren Sie dazu einzelne Textkonstanten mit dem Textadditions-Operator (&):

```
MessageBox.Show("Das ist ein funktionierender " & _
   "Versuch mit dem Umbruch einer Textkonstante")
```

Wollen Sie mehrere Anweisungen in einer Zeile unterbringen, hängen Sie einen Doppelpunkt an die Zeile an:

```
txtBrutto.Text = "": txtNetto.SetFocus
```

Mehrere Anweisungen in einer Zeile machen den Quellcode jedoch unübersichtlich. Schreiben Sie also grundsätzlich (mit wenigen Ausnahmen) nur eine Anweisung pro Zeile.

Ansonsten bauen sich Anweisungen auf, wie in anderen Programmiersprachen prinzipiell auch. Zuweisungen verwenden den Zuwei-

sungsoperator = (der in VB.NET identisch ist mit dem Vergleichsoperator).

Elementare Anweisungen sind, wie Sie ja wahrscheinlich bereits wissen, entweder Zuweisungen von arithmetischen oder logischen Ausdrücken an Variablen:

```
i = 1 + 1
```

oder Aufrufe von Prozeduren, Funktionen und Methoden:

```
Console.WriteLine("Hello World")
```

Arithmetische und logische Ausdrücke arbeiten mit den ab Seite 79 beschriebenen Operatoren. Das Beispiel verwendet die Operatoren = (Zuweisung) und + (Addition). Den Aufruf von Prozeduren/Funktionen/Methoden beschreibe ich ab Seite 76.

Daneben existieren noch die Struktur-Anweisungen (Schleifen und Verzweigungen), die ich ab Seite 132 beschreibe.

Bei Anweisungen unterscheidet VB.NET Groß- und Kleinschreibung nicht (wie es z. B. in C++, Java und C# der Fall ist). Sie können die einzelnen Elemente einer Anweisung also beliebig groß- oder kleinschreiben. Visual Studio formatiert Anweisungen allerdings immer automatisch so um, dass die Elemente der Anweisung deren Deklaration entsprechen.

3.3.2 Kommentare

Kommentare leiten Sie mit einem einfachen Apostroph ein:

```
MessageBox.Show("Hello World") ' Das ist ein Kommentar
```

Alles, was dem Kommentarzeichen folgt, wird vom Compiler nicht als Programmcode gewertet. Mehrzeilige Kommentare sind prinzipiell nicht möglich. Sie müssen dazu einzelne Kommentarzeilen schreiben:

```
' Das ist ein
' mehrzeiliger
' Kommentar
```

Statt des Apostrophs könnten Sie zur Einleitung eines Kommentars theoretisch auch das Schlüsselwort `Rem` verwenden, aber das hat in

dieser Welt noch nie jemand gemacht, weswegen ich lieber auch darauf verzichte.

Eine besondere Art des Kommentars ist der Aufgabenkommentar. Wie ich bereits in Kapitel 2 beschrieben habe, können Sie den Text eines Kommentars automatisch in die Aufgabenliste übernehmen, wenn Sie den Kommentar mit einem Aufgabentoken (per Voreinstellung: `HACK`, `TODO`, `UNDONE`, `UnresolvedMergeConflict`) beginnen:

```
Sub Main()
   ' TODO: Den Rest programmieren :-)
End Sub
```

Sie sollten diese speziellen Aufgabenkommentare immer dann nutzen, wenn irgendwo im Quellcode noch irgendetwas zu tun ist. Da diese Kommentare immer automatisch in der Aufgabenliste erscheinen und Sie diese Liste auch gut filtern können (etwa nur die Kommentare anzeigen lassen), haben Sie eine gute Übersicht über die noch zu erledigenden Aufgaben.

3.3.3 Der Aufruf von Prozeduren, Funktionen und Methoden

Prozeduren, Funktionen und Methoden enthalten, wie Sie ja bereits wahrscheinlich wissen, vorgefertigte Programme, die Sie über den Namen der Prozedur, Funktion bzw. Methode aufrufen können.

Aufruf

Prozeduren, Funktionen und Methoden werden immer mit deren Namen aufgerufen, gefolgt von Klammern, in denen Sie der Prozedur/Funktion/Methode meist Argumente übergeben, die die Ausführung steuern. Prozeduren und Funktionen sind in Modulen, Methoden in Klassen organisiert. `IsNumeric` ist z. B. eine globale Funktion, die im Modul `Information` gespeichert ist. Die `WriteLine`-Methode gehört dagegen zu der Klasse `Console`.

Wenn Sie eine Prozedur oder Funktion aufrufen, müssen Sie das Modul nicht mit angeben (können es aber):

```
' Aufruf ohne Modul
If IsNumeric(txtCentimeter.Text) Then
   ...
```

```
' Aufruf mit Modul
If Information.IsNumeric(txtCentimeter.Text) Then
   ...
```

Methoden von Strukturen oder Klassen können normal oder statisch deklariert sein. Normale Methoden können Sie nur über eine Instanz der Struktur oder Klasse (ein Objekt) aufrufen. Dazu geben Sie den Objektnamen gefolgt vom Verweisoperator (dem Punkt) und dem Methodennamen an. Das folgende Beispiel erzeugt ein `FileInfo`-Objekt um über die Methode `LastAccessTime` Informationen zu einer Datei auszulesen:

```
' FileInfo-Objekt erzeugen
Dim fi As System.IO.FileInfo
fi = New System.IO.FileInfo("C:\Boot.ini")
' Datum des letzten Zugriffs auf die Datei auslesen
Dim dt As Date
dt = fi.LastAccessTime()
' und ausgeben
Console.WriteLine(dt.ToShortDateString())
```

Statische Methode können dagegen ohne Instanz, über den Namen der Struktur oder Klasse, aufgerufen werden. Wie Sie ja bereits wissen, sind Strukturen und Klassen in Namensräumen organisiert. Wenn Sie den Namensraum nicht »importiert« haben, müssen Sie dessen kompletten Namen beim Aufruf einer Methode angeben. Das folgende Beispiel rundet eine Zahl über die Methode `Round` der `Math`-Klasse, die zum Namensraum `System` gehört:

```
Dim d As Double = 1.2345
d = System.Math.Round(d, 2)
```

Ist der Namensraum importiert, müssen Sie nur noch die Klasse beim Aufruf angeben:

```
d = Math.Round(d, 2)
```

Argumente übergeben

Die Deklaration einer Prozedur, Funktion oder Methode legt fest, wie viele Argumente welchen Typs übergeben werden müssen. In Visual Studio zeigt IntelliSense beim Schreiben einer Anweisung die Dekla-

ration(en) der Prozedur/Funktion/Methode und damit die zu übergebenden Argumente an (Abbildung 3.1).

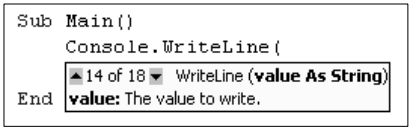

Bild 3.1: Die Syntaxhilfe beim Schreiben von Anweisungen mit Methoden

Viele Methoden liegen gleich in mehreren, so genannten »überladenen« Varianten vor. Das Überladen erläutere ich noch näher für eigene Prozeduren und Funktionen ab Seite 132 (das Überladen von Prozeduren und Funktionen ist identisch mit dem Überladen von Methoden), fürs Erste reicht es aus, dass Sie wissen, dass eine Methode auch in mehreren Varianten vorkommen kann. Die `WriteLine`-Methode der `Console`-Klasse kommt z. B. (zurzeit) in 18 Varianten vor. Die einzelnen Varianten können Sie in der Syntaxhilfe über die Cursortasten oder einen Klick auf die Pfeile in der Syntaxhilfe anzeigen lassen. Die Variante 14 erwartet z. B. nur einen einfachen String (eine Zeichenkette) als Argument (Abbildung 3.1).

Die »alten« Visual Basic-Funktionen (die im Namensraum `Microsoft.VisualBasic` organisiert sind), arbeiten zudem häufig mit optionalen Argumenten, die Sie in der Syntaxbeschreibung an den eckigen Klammern erkennen. Die Syntax der `MsgBox` ist ein gutes Beispiel dafür:

```
MsgBox(Prompt[, Buttons] [, Title]) As MsgBoxResult
```

Die optionalen Argumente können Sie einfach weglassen. Im einfachsten Fall schreiben Sie dann aber das Komma, wenn noch ein Argument folgt:

```
MsgBox("Hello World", , "Meldung")
```

Beim Aufruf einer Prozedur, Funktion oder Methode können Sie die Reihenfolge der übergebenen Argumente durch explizites Benennen auch ändern. Wichtiger ist das explizite Benennen aber für optionale Argumente, hier können Sie dann auf die zusätzlichen Kommata verzichten. Dazu schreiben Sie für jedes Argument den Namen des Arguments, gefolgt von dem Token := und dem Wert:

```
Name:=Wert
```

Die `MsgBox`-Funktion kann z. B. auch so aufgerufen werden:

```
MsgBox(Prompt:="Hello World", Title:="Meldung")
```

Gibt eine Funktion oder Methode einen Wert zurück, können Sie diesen Wert in Zuweisungen, in arithmetischen oder logischen Ausdrücken oder als Argument einer anderen Prozedur, Funktion oder Methode verwenden. Stellen Sie sich einfach vor, dass der Compiler Funktionen bzw. Methoden immer zuerst aufruft, bevor er weitere Teile der Anweisung auswertet und das Ergebnis der Funktion/Methode dann als Wert in der Anweisung weiterverarbeitet. So sind einfache Zuweisungen möglich:

```
sinY = Math.Sin(y)
```

oder Ausdrücke:

```
result = Math.Min(x1, x2) / Math.Max(y1, y2)
```

oder geschachtelte Aufrufe:

```
result = Math.Min(y1, Math.Max(y2, y3))
```

Überall da, wo ein bestimmter Datentyp erwartet wird, können Sie neben Konstanten und Variablen immer auch eine Funktion oder Methode einsetzen, die diesen Datentyp zurückgibt.

3.4 Datentypen

Wie Sie ja sicherlich bereits wissen, kommen Datentypen in einem Programm sehr häufig vor. Wenn Sie einen Ausdruck schreiben, verwendet dieser einige Datentypen und besitzt im Ergebnis auch einen Datentyp. Variablen besitzen einen Datentyp, einfache Wertangaben ebenfalls, genau wie Argumente und Rückgabewerte von Methoden.

VB.NET unterstützt die üblichen Datentypen wie z. B. `Integer` und `Double`. Alle VB.NET-Typen sind nur Aliasnamen für Datentypen, die in der CLR definiert sind. `Integer` steht z. B. für `System.Int32`. Prinzipiell können Sie immer die CLR-Typen verwenden, z. B. um eine Variable zu deklarieren:

```
Dim i As System.Int32
```

Verwenden Sie aber lieber die VB.NET-Typen, damit Ihr Quellcode besser lesbar wird.

3.4.1 Typsicherheit, der Datentyp Object, Wert- und Referenztypen

Typsicherheit

VB.NET ist eine typsichere Sprache, wenn Sie die Option `Strict` einschalten. Wenn dann irgendwo ein bestimmter Typ erwartet wird, können Sie nur einen Typ einsetzen, der zum erwarteten passt. Die Zuweisung eines Strings an eine `Integer`-Variable ist z. B. nicht möglich:

```
Dim s As String = "10"
Dim i As Integer = s ' Fehler wenn Option Strict
                     ' eingeschaltet ist
```

Das Beispiel generiert den Compilerfehler »Option Strict erlaubt keine impliziten Konvertierungen von 'String' in 'Integer'«. VB.NET sichert damit ab, dass Sie nicht versehentlich einen falschen Datentyp verwenden. Und das ist in größeren Programmen ein wesentlicher Vorteil. Es hat schließlich seinen Grund, dass eigentlich alle modernen Sprachen typsicher sind. Allein Visual Basic erlaubt (aus historischen Gründen) noch die unsichere Konvertierung von Typen (indem die Option `Strict` ausgeschaltet wird).

Wollen Sie bei eingeschalteter Option `Strict` trotzdem einen anderen Datentyp einsetzen als der Compiler erwartet, müssen Sie diesen explizit konvertieren. Wie das geht, zeige ich ab Seite 102. Hier nur ein Beispiel:

```
Dim i As Integer = Convert.ToInt32(s)
```

Ist der zugewiesene Datentyp vom Typ her passend und kleiner oder gleich groß, konvertiert VB.NET diesen meist automatisch. Die Zuweisung eines `Byte`-Typs an einen `Integer`-Typ ist z. B. problemlos möglich, weil `Byte` immer in einen `Integer`-Typ hineinpasst:

```
Dim b As Byte = 10
Dim i As Integer = b
```

Die Option `Strict` können Sie für das gesamte Projekt in den Projekteigenschaften einschalten, wie ich es bereits in Kapitel 2 beschrieben

habe. Alternativ können Sie diese Option in einzelnen Dateien über die Anweisung

```
Option Strict {On | Off}
```

ein- oder ausschalten. Diese Anweisung muss ganz oben in der Datei stehen.

Alles ist ein Objekt

In VB.NET sind alle Datentypen Klassen, die zumindest von der Klasse `Object` abgeleitet sind. Datentypen sind also immer Objekte und besitzen zumindest die von `Object` geerbten Instanzmethoden. Instanzmethoden sind Methoden, die nur über eine Instanz der Klasse aufgerufen werden können. Tabelle 3.1 listet die Instanzmethoden der `Object`-Klasse auf.

Methode	Beschreibung
`Equals(` *obj* `As Object)`	Vergleicht die Daten eines Objekts mit den Daten eines anderen Objekts auf Gleichheit (d. h., dass die in den Objekten gespeicherten Werte gleich sind).
`GetHashCode()`	Ermittelt einen Hash-Code für das Objekt. Ein Hash-Code ist ein aus den Daten des Objekts ermittelter Code, der das Objekt in verkürzter Form identifiziert.
`GetType()`	Ermittelt den Typ des Objekts. Gibt ein Objekt der Klasse `Type` zurück. Aus diesem Objekt können Sie sehr viele Informationen auslesen. Die Eigenschaft `Name` liefert z. B. den Namen des Typs, die Eigenschaft `Namespace` gibt den Bezeichner des Namensraums zurück, in dem der Typ verwaltet wird.
`ToString()`	Diese Methode gibt die Daten des Objekts als Zeichenkette zurück.

Tabelle 3.1: Die Instanzmethoden des Basistyps Object

In den von `Object` abgeleiteten Typen sind diese Methoden normalerweise mit einer neuen Implementierung überschrieben. Die `ToString`-Methode gibt z. B. bei einem `Integer`-Datentyp die gespeicherte Zahl als Zeichenkette zurück:

```
Dim i As Integer = 10
Console.WriteLine(i.ToString())
```

Bei einem `Double`-Typ wird die Zeichenkette unter Berücksichtigung einer Ländereinstellung zurückgegeben. `ToString` können Sie ohne Argument aufrufen, dann wird die Systemeinstellung verwendet:

```
Dim x As Double = 1.234
Console.WriteLine(x.ToString())
```

Auf deutschen Systemen kommt dabei der String »1,234« heraus. Alternativ können Sie diese Methode bei einem `double`-Wert auch mit einem Format-String aufrufen:

```
Console.WriteLine(x.ToString("0.00"))
```

Die Zahl wird nun auf zwei Stellen hinter dem Komma formatiert ausgegeben. Den Umgang mit Zeichenketten und deren Formatierung beschreibe ich in Kapitel 5.

Die Verwaltung aller Datentypen als Objekt ist sogar so konsequent, dass Sie auch für einfache Konstanten die Methoden des zugrunde liegenden Typs aufrufen können:

```
Dim s As String = 10.ToString()
```

Die einzelnen Typen besitzen meist noch zusätzliche Methoden und Eigenschaften. Der Typ `String` besitzt z. B. eine Vielzahl an Methoden zur Arbeit mit Zeichenketten, wie die `Replace`-Methode, über die Sie einen Teilstring durch einen anderen String ersetzen können.

Einige Methoden davon sind statisch. Diese Methoden können Sie aufrufen, ohne eine Instanz des Datentyps zu besitzen. Die `Format`-Methode der `String`-Klasse ist z. B. eine solche statische Methode:

```
Console.WriteLine(String.Format("{0:0.00}", 1.234))
```

Ich beschreibe die wichtigsten dieser Methoden in Kapitel 5.

Wert- und Referenztypen

VB.NET unterscheidet Wert- und Referenztypen. Werttypen speichern ihren Wert direkt im Stack, der Wert von Referenztypen wird auf dem Heap gespeichert. Ein Referenztyp referenziert seinen Wert. Zu den Werttypen gehören alle Standard-Datentypen (außer `String`),

Strukturen und Aufzählungen. Alle anderen Typen, wie »richtige« Objekte und der Typ `String`, sind Referenztypen.

Der Stack ist ein spezieller Speicherbereich, den der Compiler für jede aufgerufene Methode neu reserviert. Alle lokalen Daten einer Methode werden, sofern es sich um Werttypen handelt, immer auf dem Stack abgelegt. Der Stack wird auch verwendet, um Argumente an eine Methode zu übergeben. Der Aufrufer legt die Argumente auf dem Stack ab, die Methode liest die Argumente aus dem Stack aus.

Der Heap ist ein anderer Speicherbereich, der allerdings global für das gesamte Programm gilt und so lange besteht, wie das Programm läuft. Auf dem Heap werden üblicherweise programmglobale Daten, aber eben auch Referenztypen abgelegt.

Wenn eine Variable einen Werttyp besitzt, speichert diese Variable den Wert (auf dem Stack). Eine Variable, die einen Referenztyp besitzt, speichert nicht den Wert, sondern nur die Adresse des Speicherbereichs, der den Wert verwaltet. Die Adresse wird auf dem Stack verwaltet, der Wert auf dem Heap.

Weil Werttypen direkt im Stack gespeichert sind, kann der Compiler den Wert dieser Typen sehr schnell bearbeiten. Um auf einen Referenztyp zuzugreifen, benötigt der Compiler zwei Lesevorgänge: Einmal muss die Adresse der Daten aus dem Stack ausgelesen werden, dann kann der Compiler mit dieser Adresse die Daten selbst referenzieren. Referenztypen werden also prinzipiell etwas langsamer bearbeitet als Werttypen. Referenztypen besitzen aber auch Vorteile.

Wenn Sie einen Werttyp an eine Methode übergeben, erzeugt der Compiler normalerweise eine Kopie der gespeicherten Daten. Die Übergabe von Werttypen, die große Datenmengen speichern, kann somit recht viel Zeit in Anspruch nehmen. Übergeben Sie einen Referenztyp, wird nur die Referenz auf die gespeicherten Daten übergeben, was nur sehr wenig Zeit benötigt.

Auf Referenztypen können mehrere beliebige Referenzen zeigen. Das ergibt sich schon daraus, dass nur die Referenzen bei Zuweisungen

oder bei der Übergabe eines Referenztyps an eine Methode kopiert werden. Diese Technik wird bei der objektorientierten Programmierung sehr häufig verwendet. Mit Werttypen ist das prinzipiell nicht möglich, weil der Compiler immer die Werte kopiert[1].

Wenn Sie selbst Typen erzeugen, können Sie eigentlich nur bei den strukturierten Typen entscheiden, ob Sie einen Wert- oder einen Referenztyp verwenden: Entweder Sie erzeugen eine Struktur (Werttyp) oder eine Klasse (Referenztyp). Wie das geht, zeige ich in Kapitel 4.

Erzeugen von Werttypen

Werttypen werden implizit erzeugt, wenn Sie diese deklarieren. Das folgende Beispiel deklariert eine `Integer`-Variable:

```
Dim i As Integer
```

Der Compiler reserviert sofort einen Speicherbereich, der mit einem Leerwert initialisiert wird (0 bei Zahltypen, ein Leerstring bei Strings).

Erzeugen von Referenztypen

Wenn Sie einen Referenztyp deklarieren, verweist dieser noch nicht auf eine Instanz des Typs:

```
' Klasse zur Demonstration von Referenztypen
Public Class Point
   Public x As Integer
   Public y As Integer
End Class

Sub Main()
  ' Variable für den Referenztyp
  Dim p As Point
  ...
End Sub
```

Wenn Sie einen Referenztypen nicht initialisieren, speichert dieser den Wert `Nothing`. Beim Versuch, mit einem solchen Typ zu arbeiten, generiert das Programm eine Ausnahme. Sie können allerdings ab-

1. Eine Ausnahme ist die Übergabe von Argumenten an eine Prozedur, Funktion oder Methode mit der Übergabeart »By Reference« (siehe Seite 146).

fragen, ob ein Referenztyp auf eine Instanz zeigt, indem Sie über den `Is`-Operator mit `Nothing` vergleichen:

```
Dim p As Point
If Not (p Is Nothing) Then
   p.x = 10
Else
   Console.WriteLine("p zeigt nicht auf eine Instanz")
End If
```

Möglicherweise werden Sie sich fragen, was das Ganze soll. In der Praxis kommt es aber sehr häufig vor, dass Referenztypen nicht auf eine Instanz zeigen. Gut, wenn Sie dann damit umgehen können.

Wenn Sie Referenztypen verwenden wollen, müssen Sie diese über das Schlüsselwort `New` erzeugen:

```
Dim p As Point
p = New Point()
```

Wie Variablen für Werttypen können Sie auch Variablen für Referenztypen direkt bei der Deklaration initialisieren:

```
Dim p As Point = New Point()
```

Die Kurzform dieser Anweisung ist:

```
Dim p As New Point()
```

In den Klammern können Sie Argumente übergeben, die den Typen direkt bei der Erzeugung initialisieren. Welche Werte übergeben werden können, legen die Konstruktoren dieser Typen fest. Konstruktoren werden in Kapitel 4 behandelt.

Zerstören von Typen

Wie alles in der Welt müssen auch gespeicherte Daten irgendwann einmal sterben. Spätestens dann, wenn ein Programm beendet ist, werden alle Speicherbereiche freigegeben, die das Programm reserviert hatte.

Typen, die Sie in Methoden verwenden, werden automatisch freigegeben, wenn die Methode beendet ist und die Variablen bzw. Referenzen ungültig werden. Werttypen werden, da sie auf dem Stack angelegt werden, immer sofort nach der Ausführung der Methode freigegeben, da der Compiler dann den gesamten Stack freigibt.

Referenztypen, die ja auf dem Heap angelegt werden, werden vom Garbage Collector automatisch zerstört. Sie brauchen also nichts weiter zu tun, als die Referenzen auf ein Objekt freizugeben bzw. sich darauf zu verlassen, dass diese automatisch freigegeben werden, wenn die Objektvariable ihren Gültigkeitsbereich verlässt. Wenn Sie das Objekt selbst freigeben wollen setzen Sie die Objektvariable auf `Nothing`:

```
p = Nothing
```

Dieses explizite Freigeben wird allerdings nur in sehr seltenen Fällen sinnvoll sein. Wenn Sie z. B. in einer aufwändigen Funktion Referenztypen einsetzen und nach deren Verwendung Arbeitsspeicher sparen wollten, während die Funktion noch weiter ausgeführt wird, könnten Sie auf die Idee kommen, die Objektreferenzen von Hand freizugeben. Dummerweise werden diese aber erst aus dem Arbeitsspeicher entfernt, wenn der Garbage Collector Zeit dafür hat. Und das tritt mit ziemlicher Sicherheit nicht mitten in der Ausführung einer Funktion ein.

Zuweisung von Typen

Wenn Sie einen Werttyp einem anderen zuweisen, werden die Werte kopiert:

```
Dim i, j As Integer
i = 10
j = i
```

Am Ende dieses Beispiels besitzt *j* den Wert 10, ist aber immer noch ein eigener Speicherbereich. Wenn Sie *j* danach verändern, wird *i* nicht davon beeinflusst:

```
j = 11 ' i ist immer noch 10
```

Wenn Sie einen Referenztyp einem anderen Referenztyp zuweisen, werden nicht die Werte, sondern die Referenzen kopiert:

```
' Demoklasse
Public Class Point
   Public x As Integer
   Public y As Integer
End Class
```

```
Sub Main()
   ' Referenztypen erzeugen
   Dim p1 As Point = New Point()
   Dim p2 As Point = New Point()
   p1.x = 10
   ' Die Referenz von p2 auf die
   ' Referenz von p1 setzen
   p2 = p1
   ' Eine Änderung von p2 betrifft nun auch p1
   p2.x = 11
   Console.WriteLine("p1.x = " & p1.x & _
         ", p2.x = " & p2.x)
End Sub
```

In der letzten Anweisung dieses Beispiels besitzt *p1.x* denselben Wert wie *p2.x*, beide Referenzen zeigen auf denselben Speicherbereich im Heap. Der Speicherbereich, auf den *p2* anfangs gezeigt hat, wurde vom Garbage Collector zwischenzeitlich entsorgt, da er nicht mehr referenziert wird.

Der Typ String als Ausnahme

`String` ist eigentlich auch ein Referenztyp. Der Compiler erlaubt für diesen Typ aber auch eine Erzeugung ohne `New`:

```
Dim s1 As String = "Das ist ein String"
```

Alternativ können Sie auch `New` erzeugen. Dem Konstruktor können Sie dazu verschiedene Argumente übergeben. U. a. können Sie den String mit einer bestimmten Anzahl von Zeichen initialisieren:

```
Dim s2 As String = New String("*"c, 1024)
```

Bei Zuweisungen eines Strings auf einen anderen verhält sich ein String ebenfalls etwas anders als ein normaler Referenztyp: Wenn Sie zwei Stringvariablen oder Eigenschaften einander zuweisen, kopiert der Compiler zunächst die Referenz.

```
Dim s1, s2 As String
s1 = "Zaphod"
s2 = s1
```

Nach der Ausführung dieser Anweisungen referenziert *s2* dieselbe Zeichenkette wie *s1*. Soweit stimmen Strings noch mit normalen Referenztypen überein.

Wenn Sie einen String über die zweite Variable bearbeiten, erzeugt der Compiler einen neuen String:

```
s2 = "Ford"
```

Nun referenziert *s1* einen anderen String als *s2*. Eine Änderung des Strings in *s2* bewirkt keine Änderung des Strings in *s1*. VB.NET sorgt so dafür, dass Strings ähnlich wie Werttypen behandelt werden können, obwohl es sich um Referenztypen handelt.

Informationen über einen Typen ermitteln

Über die `GetType`-Funktion können Sie Informationen zu einem Typ ermitteln:

```
GetType(Typ)
```

`GetType` ergibt ein Objekt der Klasse `Type`. Dasselbe Objekt wird auch von der `GetType`-Methode zurückgegeben, die jede Instanz eines Typs besitzt. Die `GetType`-Funktion wird aber nicht auf Instanzen, sondern auf die Typen direkt angewendet.

Die `Type`-Klasse besitzt eine große Anzahl an Eigenschaften und Methoden, über die Sie Informationen zum Typ ermitteln können. Die `Name`-Eigenschaft gibt z. B. den Namen des Typs zurück. Wenn Sie beispielsweise eine Klasse besitzen, die die Daten einer Person speichert:

```
Class Person
   Public FirstName As String
   Public LastName As String
End Class
```

können Sie mit `GetType` Informationen zu dieser Klasse ermitteln:

```
Dim t As Type = GetType(Person)
Console.WriteLine("Name: " & t.Name)
Console.WriteLine("Assemblierung: " & _
   t.Assembly.FullName)
Console.WriteLine("Basistyp: " & t.BaseType.Name)
```

```
Console.WriteLine("Voller Name: " & t.FullName)
Console.WriteLine("Typ ist " & _
   (IIf(t.IsClass, "eine", "keine")) & " Klasse")
Console.WriteLine("Typ ist " & _
   (IIf(t.IsArray, "ein", "kein")) & " Array")
Console.WriteLine("Typ ist " & _
   (IIf(t.IsEnum, "eine", "keine")) & " Aufzählung")
Console.WriteLine("Typ ist " & _
   (IIf(t.IsInterface, "eine", "keine")) & _
   " Schnittstelle")
```

Dieser Vorgang wird übrigens als Reflektion (Reflection) bezeichnet: Sie können unter .NET jederzeit Informationen zu allen Typen erhalten. Das funktioniert sogar dann, wenn diese Typen in externen Assemblierungen gespeichert sind, zu denen Sie keinen Quellcode besitzen. .NET stellt Ihnen dazu Klassen zur Verfügung, die Sie im Namensraum `System.Reflection` finden.

3.4.2 Übersicht über die Standarddatentypen

Die vordefinierten VB.NET-Datentypen listet Tabelle 3.2 auf. Die VB.NET-Bezeichner dieser Datentypen sind nur Aliase für den entsprechenden CLR-Datentyp. `Integer` steht z. B. für den Typ `System.Int32`. Die CLR-Datentypen sind im Namensraum `System` gespeichert.

VB.NET-Datentyp	CLR-Datentyp	Größe	Wertebereich
`Byte`	`Byte`	8 Bit	0 bis 255
`Char`	`Char`	16 Bit	ein beliebiges Unicode-Zeichen
`Short`	`Int16`	16 Bit	-32.768 bis 32.767
`Integer`	`Int32`	32 Bit	-2.147.483.648 bis 2.147.483.647
`Long`	`Int64`	64 Bit	-9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807
`Single`	`Single`	32 Bit	$-3{,}4028235 * 10^{38}$ bis $-1{,}401298 * 10^{-45}$ für negative Zahlen; $1{,}401298 * 10^{-45}$ bis $3{,}4028235 * 10^{38}$ für positive Zahlen; maximal 7 Dezimalstellen

VB.NET-Datentyp	CLR-Datentyp	Größe	Wertebereich
`Double`	`Double`	64 Bit	-1,79769313486231570 * 10^{308} bis -4,94065645841246544 * 10^{-324} für negative Zahlen; 4,94065645841246544 * 10^{-324} bis 1,79769313486231570 * 10^{308} für positive Zahlen; maximal 16 Dezimalstellen
`Decimal`	`Decimal`	128 Bit	0 bis +/- 79.228.162.514.264.337.593.543.950.335 ohne Dezimalpunkt; 0 bis +/- 7,9228162514264337593543950335 mit 28 Dezimalstellen; die kleinste Zahl ungleich 0 ist +/- 0,0000000000000000000000000001
`Date`	`DateTime`	16 Bit	1.1.2001 bis 31.12.9999
`Boolean`	`Boolean`	16 Bit	`True` oder `False`
`String`	`String`	variabel	beliebige Unicode-Zeichenketten

Tabelle 3.2: Die VB.NET-Standarddatentypen

.NET kennt zudem noch einige Integer-Datentypen mir Vorzeichen, die in VB.NET nicht repräsentiert werden (`SByte`*,* `UInt16`*,* `UInt32` *und* `UInt64`*). Diese Typen sind nicht CLS*[2]*-kompatibel, sollten also in .NET-Assemblierungen nicht verwendet werden. Trotzdem kann es sein, dass eine externe Assemblierung (die z. B. in C# entwickelt wurde) diese Typen verwendet. Wenn Sie eine solche Assemblierung referenzieren und verwenden, können Sie diese speziellen Typen über Methoden der* `Convert`*-Klasse in VB.NET-Typen konvertieren.*

2. Die Common Language Specification (CLS) definiert Regeln, die dafür sorgen, dass .NET-Assemblierungen zu allen .NET-Sprachen kompatibel sind.

Instanzmethoden der Datentypen

Die Standardtypen besitzen wie alle Typen die Instanzmethoden, die sie von `Object` geerbt haben (Seite 80). Der `ToString`-Methode können Sie bei nummerischen Typen allerdings auch einen Format-String oder einen Format-Provider übergeben, um die Zahl zu formatieren. Formatierungen werden in Kapitel 5 behandelt. Ich zeige hier nur kurz, wie Sie z. B. einen `Double`-Wert auf zwei Stellen hinter dem Komma formatieren können:

```
Dim d As Double = 1.2345
Console.WriteLine(d.ToString("0.00"))
```

Neben den geerbten Methoden besitzen die Standardtypen noch die in Tabelle 3.3 dargestellten zusätzlichen Instanzmethoden.

Methode	Beschreibung

`CompareTo(`
`  value As Object)`
`  As Integer`
	vergleicht einen Typ mit einem anderen. Rückgabe: < 0: der Typ ist kleiner als der andere, 0: beide Typen sind gleich, > 0: der Typ ist größer als der andere.

`GetTypeCode()` `  As TypeCode`	ermittelt den Datentyp des Typs. Gibt einen Wert des Typs `TypeCode` zurück. `TypeCode` ist eine Aufzählung von Konstanten, die so benannt sind wie die Typen. `TypeCode.Byte` definiert z. B. einen `Byte`-Typ.

Tabelle 3.3: Die zusätzlichen Methoden der Standarddatentypen

Die `CompareTo`-Methode wird hauptsächlich implizit verwendet, wenn ein Typ in einer Auflistung gespeichert ist, die sortierbar ist. Explizit brauchen Sie diese Methode eigentlich nie aufzurufen, da Sie für Vergleiche die Vergleichsoperatoren (Seite 130) verwenden können.

Der Datentyp `String` besitzt noch eine Vielzahl weiterer Methoden. Ich beschreibe die wichtigsten davon separat in Kapitel 5.

Klassenmethoden und -eigenschaften der Datentypen

Alle Datentypen besitzen neben den Instanzmethoden auch noch Klassenmethoden und -eigenschaften, die Sie ohne Instanz der Klasse verwenden können. Die `MaxValue`-Eigenschaft der nummerischen Typen gibt z. B. den größten speicherbaren Wert zurück:

```
Console.WriteLine("Double kann maximal " & _
   Double.MaxValue & " speichern.")
```

Über die `Parse`-Methode können Sie einen String in den entsprechenden Typ umwandeln:

```
Dim d As Double
d = Double.Parse("1.234")
```

Tabelle 3.4 zeigt die wichtigsten dieser Eigenschaften und Methoden.

Eigenschaft/Methode	Beschreibung
`MinValue`	liefert den kleinsten speicherbaren Wert.
`MaxValue`	liefert den größten speicherbaren Wert.
`Parse(s As String [...]) As Typ`	Über diese Methode können Sie einen String in den Typ umwandeln, auf dem Sie die Methode anwenden. `Parse` erlaubt bei numerischen Typen zusätzlich die Angabe der zu verwendenden Kultur. Die Kultur wird in Kapitel 5 behandelt.
`TryParse(s As string, style As NumberStyles, provider As IFormatProvider, ByRef result As Double) As Boolean`	Über diese komplexe Methode können Sie überprüfen, ob eine Umwandlung eines Strings in einen speziellen Typ möglich ist.

Tabelle 3.4: Die wichtigsten Klasseneigenschaften und -methoden der Standardtypen

Die `TryParse`-Methode will ich hier nicht näher beschreiben, weil Sie dazu wissen müssen, wie Sie mit Schnittstellen umgehen (`IFormatProvider`) und was Kultur-Informationen sind. Schnittstellen werden in Kapitel 4 behandelt, Kultur-Informationen (Globalisierung) in Kapitel 5. Das folgende Beispiel überprüft einen String darauf, ob der mit den aktuellen Ländereinstellungen in einen `Double`-Wert konvertierbar ist:

```
Dim s As String = "abc", d As Double
Dim culture As System.Globalization.CultureInfo
culture = System.Globalization.CultureInfo.CurrentCulture()
If Not Double.TryParse(s, _
   System.Globalization.NumberStyles.Float, _
   culture.NumberFormat, d) Then
   Console.WriteLine(s + " kann nicht in einen " & _
      "double-Wert umgewandelt werden.")
End If
```

3.4.3 Integerdatentypen

VB.NET unterscheidet einige Datentypen für die Speicherung von Integerwerten (Werte ohne Dezimalanteil): `Byte`, `Short`, `Integer` und `Long`. Bei der Auswahl dieser Datentypen müssen Sie eigentlich nur beachten, dass der Datentyp groß genug ist für Ihre Anwendung und ob Sie ein Vorzeichen benötigen (das `Byte` nicht unterstützt). Verwenden Sie im Zweifelsfall aber lieber einen zu großen Datentyp als einen zu kleinen. Speicher besitzen heutige Computer wohl mehr als ausreichend.

Die gängigen Literale für Integerwerte sind Zahlen:

```
Dim i As Integer = 1234
```

Alternativ können Sie auch die hexadezimale Schreibweise verwenden. Stellen Sie dazu `&H` vor den Wert:

```
Dim i As Byte = &HFF ' 255
```

Wenn Sie eine Zahl ohne Dezimalstelle schreiben, erkennt der Compiler diese als `Integer` oder `Long`, je nachdem, ob der Wert in den Typ passt:

```
Dim i As Long
i = 10 ' die Zahl 10 ist vom Typ Integer
i = 2147483648 ' die Zahl 2147483648 ist vom Typ Long
```

Ausprobieren können Sie dies, indem Sie den Typ eines Literals ermitteln:

```
Console.WriteLine("Das Literal 2147483648 besitzt " & _
   "den Typ '" + 2147483648.GetType().Name + "'")
```

Sie können einen solchen Wert aber auch einem kleineren oder größeren Datentyp zuweisen:

```
Dim i As Byte
i = 10
Dim j As Integer
j = b
```

Wenn Sie versuchen, einem Datentyp ein zu großes oder nicht konvertierbares Literal zuzuweisen, meldet der Compiler einen Fehler:

```
Dim b As Byte, i As Integer
i = 1024 ' Compilerfehler "Der Konstantenausdruck
         ' ergibt einen Wert, der in Typ "Byte"
         ' nicht repräsentierbar ist"
i = 10.5 ' Compilerfehler "Option Strict erlaubt keine
         ' impliziten Konvertierungen von Double in
         ' Integer"
```

Der zweite Fehler wird nur dann gemeldet, wenn die Option `Strict` eingeschaltet ist. Ohne diese Option würde der Compiler den `Double`-Wert einfach in einen `Integer`-Wert runden.

Konvertierungen werden ab Seite 102 behandelt.

Sie können einem Literal ein Suffix anhängen um damit den Datentyp zu bestimmen.

Suffix	Datentyp
S	Short
I	Integer
L	Long

Tabelle 3.5: Die Suffixe für Integer-Literale

Die Groß- und Kleinschreibung spielt dabei keine Rolle.

3.4.4 Fließkommatypen

Fließkommatypen (`Single`, `Double und Decimal`) speichern Dezimalzahlen mit einer festgelegten maximalen Anzahl an Ziffern. Die mögliche Anzahl der Dezimalstellen hängt davon ab, wie viele Ziffern vor dem

Komma gespeichert sind. `Single` kann z. B. nur zwei Dezimalstellen speichern, wenn fünf Ziffern vor dem Komma gespeichert sind. `Single` ist deswegen normalerweise auch nicht besonders geeignet. Verwenden Sie lieber `Double`, denn dieser Datentyp kann bei fünf Vorkomma-Ziffern bereits neun Dezimalstellen speichern. Bei einer Ziffer vor dem Komma kann `Double` etwa fünfzehn, `Single` nur etwa sieben Dezimalstellen speichern.

`Decimal` ist ein spezieller Datentyp mit einer festgelegten Anzahl von Ziffern (29). Damit können Sie sich also schnell ausrechnen, wie viele Dezimalstellen dieser Datentyp speichern kann, wenn eine bestimmte Anzahl Ziffern vor dem Komma verwaltet wird. Diesen Datentyp können Sie verwenden, wenn Sie hochgenaue Berechnungen ausführen wollen.

Übersteigt die Anzahl der Dezimalstellen eines Ausdrucks die speicherbaren Stellen, rundet der Compiler den Wert auf die speicherbaren Dezimalstellen. Bei der folgenden Zuweisung

```
Dim number As Single = 99999.125F
```

wird der Wert auf 99999.13 gerundet. Das F steht übrigens dafür, dass es sich bei dem Literal um einen `Single`-Wert handelt (der in anderen Sprachen auch als `Float` bezeichnet wird). Beim Runden verwendet der Compiler ein Verfahren, das dem mathematischen Verfahren ähnlich ist. Ist die Ziffer rechts von der letzten darstellbaren Ziffer eine 5, rundet der Compiler die Ziffer davor nur dann auf,

- wenn diese Ziffer < 5 ist und es sich um eine gerade Ziffer handelt oder
- wenn diese Ziffer >= 5 ist und es sich um eine ungerade Ziffer handelt.

Bei

```
Dim number As Single = 99999.135F
```

wird der Wert also auf 99999.13 abgerundet, bei

```
Dim number As Single = 99999.155F
```

wird allerdings auf 99999.16 aufgerundet.

Dieses spezielle Rundungsverfahren stellt sicher, dass beim Runden möglichst keine Ungleichheiten entstehen. Bei dem in Deutschland üblichen kaufmännischen Runden, bei dem bei einer 5 hinter der letzten im Ergebnis darzustellenden Ziffer immer aufgerundet wird, werden mehr Zahlen aufgerundet als abgerundet. Die 5 steht eben eigentlich genau in der Mitte des Zahlenstrahls.

Weitere Ziffern werden allerdings nicht berücksichtigt. Bei

```
Dim number As Single = 99999.13599999F
```

wird der Wert ebenfalls auf 1.3 gerundet.

Als Literal für einen Fließkommadatentyp können Sie eine Zahl in der englischen Schreibweise verwenden. Der Compiler wertet Zahlen mit Dezimalstellen grundsätzlich als `Double` aus:

```
Dim number As Double = 1.234
```

Einem Fließkomma-Typ können Sie ohne Konvertierung (bei eingeschalteter Option `Strict`) fast alle anderen Zahldatentypen zuweisen. Lediglich die Zuweisung eines `Double`-Werts an einen `Decimal`-Typ ist nicht möglich.

Einem `Single`-Typ können Sie auch bei eingeschalteter Option `Strict` einen `Double`-Wert zuweisen. Ist der Wert zu groß, resultiert der spezielle Wert +∞ (positiv unendlich). Der Compiler erzeugt keinen Kompilier- und keinen Laufzeitfehler bei solchen Zuweisungen. Mit dem Wert +∞ können Sie zwar weiterrechnen, das Ergebnis wird aber immer ein ähnlicher Wert sein. Diese speziellen Werte werden auf Seite 98 beschrieben.

Wenn Sie einem `Decimal`-Typ ein Literal zuweisen wollen, das einen Nachkommaanteil besitzt, müssen Sie dieses explizit als `Decimal` kennzeichnen. Hängen Sie dazu das Suffix *D* an:

```
Dim d As Decimal = 1.234D
```

Integerwerte können Sie allerdings ohne Konvertierung zuweisen:

```
Dim d As Decimal = 1234
```

Für Dezimalzahlen können Sie auch die wissenschaftliche Schreibweise verwenden:

```
Dim x As Double = 5E-2 ' 0.05
```

5E-2 steht im Beispiel für: 5 * 10-2.

Für die Festlegung des Datentyps können Sie eines der in Tabelle 3.6 dargestellten Suffixe verwenden.

Suffix	Datentyp
F	Single
R	Double
D	Decimal

Tabelle 3.6: Suffixe für die Festlegung des Datentyps eines Fließkomma-Literals

3.4.5 Über- und Unterläufe und spezielle Werte

Über- und Unterläufe

Wenn Sie einem Typ einen zu großen Wert zuweisen, resultiert daraus ein Überlauf:

```
Dim value1 As Byte = 255
Dim value2 As Byte = 2
value1 += value2 ' Überlauf, Ergebnis: 1
```

Der Wert in der Variablen *value1* ist durch die Addition mit 2 übergelaufen.

Ein Unterlauf wird erzeugt, wenn Sie einen zu kleinen Wert zuweisen:

```
Dim value1 As Byte = 0
Dim value2 As Byte = 2
value1 -= value2 ' Unterlauf, Ergebnis: 254
```

Wenn Sie unter Visual Studio oder mit dem Kommandozeilencompiler die Voreinstellung für die Behandlung von Unter- und Überläufen übernehmen, resultiert daraus eine Ausnahme vom Typ `System.OverflowException`, und das ist auch gut so. Damit bekommen Sie mit, dass ein Überlauf aufgetreten ist und können Ihr fehlerhaftes Programm reparieren.

Sie haben aber alternativ die Möglichkeit, die Prüfung auf Unter- und Überläufe abzuschalten. In Visual Studio schalten Sie dazu die entsprechende Option in den Konfigurationseigenschaften (unter OPTIMIERUNGEN) des Projekts ein. Sinnvoll ist die Überprüfung auf jeden Fall für die `Debug`-Konfiguration. Wenn Sie Ihr Programm in der Entwicklungsphase nur mit dieser Konfiguration kompilieren und fleißig testen (lassen), werden Über- und Unterläufe auf jeden Fall gemeldet, sodass Sie darauf reagieren und den Fehler beseitigen können.

Wenn Sie die Überprüfung für das Release abschalten, wird Ihr Programm bei Berechnungen und Zahl-Zuweisungen minimal schneller, aber dafür auch wesentlich unsicherer. Der Überlauf in dem obigen Beispiel würde dann z. B. den Wert 1 ergeben ($11111111_2 + 00000001_2 = 00000000_2 + 00000001_2 = 00000001_2$). Wenn Sie nicht darauf achten, Ihre Datentypen ausreichend groß zu dimensionieren, kann ein Überlauf zu enormen logischen Fehlern führen, nach denen Sie u. U. tagelang (oder wie ich meist nächtelang) suchen.

+0, -0, +∞, -∞, NaN

Die Typen `Single` und `Double` (nicht `Decimal`!) können spezielle Werte speichern. Wenn Sie z. B. einen `Double`-Typ, der eine positive Zahl speichert, durch 0 teilen, resultiert der Wert +∞ (positiv unendlich):

```
Dim d1 As Double = 10, d2 As Double = 0, d3 As Double
d3 = d1 / d2
Console.WriteLine(d3) ' Ausgabe: +unendlich
```

Teilen Sie einen negativen Wert durch 0, resultiert der Wert -∞ (negativ unendlich). Wenn Sie 0 durch 0 teilen, ergibt das den Wert `NaN` (Not a Number). In einigen speziellen Fällen resultiert auch der Wert –0 oder +0. Die Regeln dazu sind ziemlich kompliziert und in einem Standard für Fließkommaoperationen beschrieben (IEEE 754). Sie finden diesen Standard im Internet unter der Adresse *grouper.ieee.org/groups/754/*.

Über einige statische Methoden der `Double`- und der `Single`-Klasse können Sie herausfinden, ob ein Typ einen dieser Werte speichert:

```
If Double.IsInfinity(d3) Then
   Console.WriteLine("d3 ist unendlich.")
End If
```

```
If Double.IsNegativeInfinity(d3) Then
   Console.WriteLine("d3 ist negativ unendlich.")
End If
If Double.IsPositiveInfinity(d3) Then
   Console.WriteLine("d3 ist positiv unendlich.")
End If
If Double.IsNaN(d3) Then
   Console.WriteLine("d3 ist NaN.")
End If
```

Die Bedeutung dieser Werte ist für die Praxis auch wohl eher gering. Vermeiden Sie einfach das Teilen durch 0, dann kann eigentlich nichts passieren.

3.4.6 Datumswerte

Datumswerte werden in einem `Date`-Typ gespeichert. Wenn die Option `Strict` nicht eingeschaltet ist, können Sie einem solchen Typ direkt einen String zuweisen, der ein Datum (inkl. Zeit) in dem Format speichert, das für das System eingestellt ist. Ist die Option `Strict` eingeschaltet, können Sie eine der Varianten der `ToDateTime`-Methode der `Convert`-Klasse verwenden, um ein Datum zu erzeugen:

```
Dim d As Date
d = Convert.ToDateTime("31.12.2005")
```

Alternativ können Sie eine neue Instanz des `Date`-Typs über den `New`-Operator erzeugen und dieser Instanz im Konstruktor der `Date`-Klasse verschiedene Werte übergeben, mit denen Sie das Datum spezifizieren können:

```
Dim d As Date = New Date(2005, 12, 31) ' 31.12.2005
```

Alternativ können Sie auch eine der Klasseneigenschaften der `Date`-Klasse verwenden, um das aktuelle oder ein bestimmtes Datum einzustellen:

```
d = Date.Now
d = Date.Today
d = Date.Parse("31.12.2005")
```

`Now` liefert dabei das bis auf die Millisekunde genaue aktuelle Datum, `Today` das aktuelle Datum ohne Zeitangabe.

Um ein Datum auszugeben, können Sie eine der To-Methoden verwenden:

```
Console.WriteLine(d.ToLongDateString())
Console.WriteLine(d.ToShortDateString())
Console.WriteLine(d.ToLongTimeString())
Console.WriteLine(d.ToShortTimeString())
```

Ein Datumswert wird übrigens immer mit einer genauen Zeitangabe gespeichert.

Den Umgang mit Datumswerten beschreibe ich in Kapitel 5.

3.4.7 Zeichen und Zeichenketten

Für die Speicherung einzelner Zeichen verwenden Sie den Typ `Char`. Zeichenketten werden im Typ `String` gespeichert. `Char` ist ein Werttyp, `String` ein Referenztyp (der aber ähnlich einem Werttyp ausgewertet wird, siehe Seite 87). Zeichen werden immer im Unicode-Format gespeichert. In diesem Format wird ein Zeichen in einem zwei Byte großen Speicherbereich verwaltet. Damit sind Zeichensätze möglich, die 65535 verschiedene Zeichen speichern. Der Zeichensatz »Western Latin-1« wird in westlichen Ländern eingesetzt und beinhaltet alle Zeichen, die in diesen Ländern verwendet werden. Für japanische und chinesische Länder existieren andere Zeichensätze.

`Char`-Literale werden wie Strings in Anführungszeichen eingeschlossen, müssen aber mit dem Literalzeichen C gekennzeichnet werden, wenn die Option `Strict` eingeschaltet ist (im anderen Fall verwendet der Compiler einfach das erste Zeichen eines zugewiesenen Strings):

```
Dim c As Char = "A"c
```

`String`-Literale schließen Sie ebenfalls in Anführungszeichen ein, fügen allerdings kein Literalzeichen an

```
Dim s As String = "Das ist ein String"
```

Strings werden automatisch dynamisch verwaltet. Die Größe des Strings ist lediglich durch den verfügbaren Speicher begrenzt. Wenn Sie einen String neu zuweisen oder verändern, erzeugt das Programm immer eine neue Instanz des String:

```
Dim s As String = "Das ist ein"
s = s & "Test" ' erzeugt eine neue Instanz
```

Wenn Sie viel mit einem String arbeiten, ist das nicht sehr effizient. Dann können Sie alternativ eine Instanz der `StringBuilder`-Klasse verwenden, die das Umkopieren vermeidet (siehe Kapitel 5).

In Stringausdrücken ruft der Compiler implizit die `ToString`-Methode anderer Typen auf, wenn Sie das nicht machen. Deshalb können Sie beispielsweise auch bei eingeschalteter Option `Strict` einen `Integer`-Typ mit einem String ohne Konvertierung addieren:

```
Dim i As Integer = 42
Dim s As String = i & " ist eine gute Zahl"
```

Die `String`-Klasse besitzt eine große Anzahl an Methoden zur Arbeit mit Zeichenketten. In Kapitel 5 beschreibe ich, wie Sie diese nutzen.

3.4.8 Der Typ Object

Der Typ `Object` ist ein besonderer Typ. Diesem Typ können Sie jeden anderen Typ zuweisen:

```
Dim o As Object
o = 10
o = "Hallo"
```

Viele Methoden besitzen Argumente vom Typ `Object`. Das ist auch der Haupt-Anwendungsbereich dieses Typs. `Object`-Argumenten können Sie beliebige Werte übergeben. Die `WriteLine`-Methode der `Console`-Klasse ist ein Beispiel dafür. Methoden, die den übergebenen Wert als String auswerten, nutzen dabei die `ToString`-Methode, die jeder Typ von `Object` geerbt hat.

Wenn Sie einen `Object`-Typ in einem nummerischen Ausdruck verwenden, müssen Sie diesen in den entsprechenden Typ konvertieren, wenn die Option `Strict` eingeschaltet ist:

```
Dim o As Object = 10
Dim i As Integer = CType(o, Integer) * 10
```

Die `CType`-Funktion wandelt den im ersten Argument übergebenen Wert in den Typ um, der im zweiten Argument angegeben wird.

In Stringausdrücken können Sie allerdings auch die `ToString`-Methode des `Object`-Typen verwenden:

```
Dim o As Object = "42"
Dim s As String
s = o.ToString() + " ist eine gute Zahl"
```

`Object` merkt sich den gespeicherten Typ. Über die `GetType`-Methode können Sie den Typ ermitteln:

```
Console.WriteLine(o.GetType().Name)
```

Beachten Sie, dass `GetType` ein Objekt der Klasse `Type` zurückgibt, über das Sie noch wesentlich mehr Informationen über den Typ erhalten können, als nur dessen Name. Die Eigenschaft `IsArray` ist z. B. `True`, wenn der Typ ein Array ist.

Verwenden Sie den Typ `Object` *idealerweise nur in Ausnahmefällen. Die Typsicherheit von VB.NET (bei eingeschalteter Option* `Strict`*) geht mit diesem Typ verloren. Wenn Sie* `Object`*-Variablen verwenden, wissen Sie in größeren Programmen nie genau, welchen Datentyp diese tatsächlich speichern. Logische Programmfehler und Ausnahmefehler sind damit vorprogrammiert.*

Boxing und Unboxing

Der Datentyp `Object` ist ein Referenztyp. Boxing und Unboxing sind Techniken, die im Hintergrund verwendet werden und die ermöglichen, dass der Typ `Object` nicht nur Referenztypen, sondern auch Werttypen verwalten kann. Wenn ein Werttyp an einen `Object`-Typ zugewiesen wird, wird implizit Boxing verwendet:

```
Dim i As Integer = 10
Dim o As Object = i ' i wird geboxt
```

Boxing ist eine implizite Konvertierung eines Werttypen in den Typ `Object`. Da `Object` ein Referenztyp ist, werden die verwalteten Werte folglich auf dem Heap gespeichert. Wenn Sie einen Werttypen an eine `Object`-Variable zuweisen oder an ein `Object`-Argument übergeben, erzeugt der Compiler eine »Box« auf dem Heap und kopiert den Wert des Werttypen in diese Box. Die Box simuliert einen Referenztypen und kann deshalb über eine `Object`-Variable verwendet werden.

Unboxing bedeutet, dass ein `Object`-Typ in einen Werttyp konvertiert wird (was immer explizit geschehen muss):

```
Dim o As Object = 10
Dim i As Integer = CType(o, Integer) ' der Wert von o
   ' wird in den Werttyp »entboxt«
```

3.4.9 Konvertierungen

Wie Sie ja bereits wissen, ist VB.NET eine typsichere Sprache wenn `Option Strict` eingeschaltet ist. Dann können Sie einem Typ nur einen passenden anderen Typ zuweisen, dessen Maximal- bzw. Minimalwert ohne Kürzungen in den anderen Typ passt. Einer `Integer`-Variable können Sie z. B. ohne Probleme einen `Byte`-Wert zuweisen:

```
Dim b As Byte = 255
Dim i As Integer = b
```

Der Compiler konvertiert den Datentyp dann implizit. Wenn der Grundtyp aber nicht passt oder der Wert nicht in den zugewiesenen Typ passen würde, wird ein Fehler gemeldet:

```
Dim s As String = "255"
Dim i As Integer = s ' Fehler
Dim f As Single = 1.234
i = f ' Fehler
```

Literale können Sie über die Datentypszeichen auf einen bestimmten Datentyp festlegen. Andere Typen müssen Sie über die Funktion `CType` oder mit Hilfe der Methoden der `Convert`-Klasse explizit konvertieren.

Explizite und implizite Konvertierungen

Jeder Typ kann spezielle Operatoren für implizite und explizite Konvertierungen enthalten. Solch ein Operator ist so etwas wie eine Methode, die allerdings automatisch aufgerufen wird, wenn der Typ konvertiert werden soll. Konvertierungs-Operatoren können für die verschiedensten Typen definiert sein. Wenn Sie in C# eine eigene Klasse oder Struktur erzeugen, können Sie beliebig viele Konvertierungs-Operatoren erzeugen, die dann die implizite oder explizite Konvertierung der verschiedenen Typen erlauben. VB.NET lässt die Imple-

mentierung von Operatoren allerdings leider nicht zu und ermöglicht auch keine Konvertierung von Typen, die in anderen Sprachen entwickelt wurden und die Konvertier-Operatoren implementieren.

Einfache Typen können in VB.NET aber konvertiert werden. Ein `Integer`-Typ lässt z. B. (u. a.) die implizite Konvertierung eines `Byte`- und eines `Short`-Typs und die explizite Konvertierung eines `Single`- und eines `Double`-Typs zu:

```
Dim i As Integer = 0, b AS Byte = 10
Dim f As Single = 1.2345
i = b ' Implizite Konvertierung
i = CType(f, Integer) ' Explizite Konvertierung.
   ' Die Dezimalstellen gehen verloren.
```

Bei impliziten Konvertierungen geht nichts verloren, bei expliziten Konvertierungen kann es sein, dass Informationen verloren gehen. Explizite Konvertierungen nehmen Sie über die `CType`-Funktion vor.

Konvertierung über CType

Die `CType`-Funktion ermöglicht die Konvertierung der verschiedensten Datentypen ineinander. Die Syntax dieser Funktion ist:

```
CType(expression As Object, typename As Object) As Typ
```

Im ersten Argument übergeben Sie den zu konvertierenden Wert, im zweiten einen Typbezeichner. `CType` ist eigentlich keine Funktion. Der Compiler setzt einen Aufruf von `CType` in eine direkte Konvertierung um. In der Laufzeit erfolgt also kein Funktionsaufruf. Dabei können allerdings nur die Standarddatentypen konvertiert werden. Die Konvertierung von Instanzen eigener (C#-)Klassen, die spezielle Konvertierungsoperatoren implementieren, ist leider nicht möglich.

So können Sie beispielsweise einen `Double`-Typ in einen `Integer`-Typ konvertieren:

```
Dim d As Double = 1.55
Dim i As Integer = CType(d, Integer)
```

Beim expliziten Konvertieren kann es sein, dass Informationen verloren gehen oder verändert werden. Die `Integer`-Variable des Beispiels speichert nach der Ausführung z. B. den Wert 2. Wie in VB.NET üblich, wird dabei annähernd mathematisch gerundet (siehe ab Seite 94).

Schlägt die Konvertierung fehl, weil Sie beispielsweise einen nicht nummerischen String in einen Integer-Wert konvertieren, erzeugt das Programm eine Ausnahme vom Typ `System.InvalidCastException`. Diese Ausnahme können Sie abfangen, wie ich noch in Kapitel 6 zeige.

Konvertierungen über die Convert-Klasse

Alternativ zu `CType` können Sie auch die Methoden der `Convert`-Klasse zur Konvertierung verwenden. Einen String konvertieren Sie z. B. über die `ToInt32`-Methode in einen Integer:

```
Dim s As String = "10"
Dim i As Integer = Convert.ToInt32(s)
```

Schlägt die Konvertierung fehl, erzeugt die Methode eine Ausnahme vom Typ `System.FormatException`:

```
s = "10a"
i = Convert.ToInt32(s) ' Ausnahme
```

Tabelle 3.7 zeigt die wichtigsten Klassenmethoden der `Convert`-Klasse. Diese Methoden liegen in mehreren Varianten vor, denen Sie unterschiedliche Datentypen übergeben können. Ich beschreibe nicht alle diese Varianten und erläutere deswegen die Grundlagen: Die Varianten, die keinen `String` übergeben bekommen, besitzen nur ein Argument. Der `ToInt32`-Methode können Sie z. B. im ersten Argument (u. a.) einen `Single`-, `Short`- oder `Double`-Wert übergeben. Die Variante, die einen String übergeben bekommt, kann zusätzlich noch mit einem zweiten Argument aufgerufen werden, in dem Sie einen Format-Provider übergeben, der die länderspezifische Formatierung des Strings festlegt. Übergeben Sie diesen nicht, wird die Systemeinstellung berücksichtigt. Einige der Methoden der `Convert`-Klasse sind für Visual Basic .NET irrelevant, weil diese in Datentypen konvertieren, die VB.NET nicht kennt. Dazu gehört beispielsweise die Methode `ToUInt32`, die in einen 32-Bit-Integerwert ohne Vorzeichen konvertiert. Tabelle 3.7 listet diese Methoden nicht auf.

Methode	konvertiert in
`ToBoolean(...)`	`Boolean`
`ToByte(...)`	`Byte`

Methode	konvertiert in
ToChar(...)	Char
ToDateTime(...)	Date
ToDecimal(...)	Decimal
ToDouble(...)	Double
ToInt16(...)	Short
ToInt32(...)	Integer
ToInt64(...)	Long
ToSingle(...)	Single
ToString(...)	String

Tabelle 3.7: Die wichtigsten Klassenmethoden der Klasse Convert

Für Konvertierungen in Strings können Sie übrigens auch die `ToString`-Methode verwenden, die alle Typen besitzen.

3.4.10 Aufzählungen (Enums)

Aufzählungen sind Datentypen, die aus mehreren benannten Zahlkonstanten bestehen. Eine Aufzählungen wird nach dem folgenden Schema deklariert:

```
[Attribute] [Modifizierer] Enum Name [As Typ]
   Konstantenliste
End Enum
```

In der Liste geben Sie einen oder mehrere Bezeichner an:

```
Private Enum Direction
   North
   South
   East
   West
End Enum
```

Wenn Sie keinen Datentyp angeben, besitzt die Auflistung den Typ `Integer`. Sie können den Datentyp aber auch definieren:

```
Private Enum Direction As Long
   North
   South
   East
   West
End Enum
```

Verwenden können Sie die Integer-Typen (`Byte`, `Short`, `Integer` und `Long`). Die Aufzählung können Sie nun überall da einsetzen, wo sie gültig ist. Eine Prozedur, Funktion oder Methode kann z. B. ein Argument dieses Typs besitzen:

```
Private Sub Move(ByVal direction As Direction)
   If direction = direction.North Then
      ' ...
   End If
End Sub
```

Beim Aufruf der Methode muss nun ein passender Typ übergeben werden:

```
Move(Direction.North)
```

Wenn Sie bei der Deklaration der Auflistung keinen Wert für die einzelnen Konstanten angeben, erhalten diese einen bei o beginnenden Wert. Sie können den Wert allerdings auch festlegen:

```
Private Enum Direction As Integer
   North = 1
   South = 2
   East = 4
   West = 8
End Enum
```

Wenn Sie die Werte wie im Beispiel so definieren, dass diese die einzelnen Bits der Typs repräsentieren, können Sie die Auflistung bitweise einsetzen

```
Move(Direction.North Or Direction.West)
```

und natürlich auswerten:

```
Private Sub Move(ByVal direction As Direction)
   If (direction And direction.North) > 0 Then
      Console.WriteLine("Going North")
   End If
   If (direction And direction.South) > 0 Then
      Console.WriteLine("Going South")
   End If
   If (direction And direction.West) > 0 Then
      Console.WriteLine("Going West")
   End If
   If (direction And direction.East) > 0 Then
      Console.WriteLine("Going East")
   End If
End Sub
```

Auflistungen können natürlich konvertiert werden, was aber normalerweise explizit erfolgen muss:

```
Dim d As Direction = Direction.North
Dim b As Byte = CType(d, Byte)
```


Tipps

Häufig ist es notwendig den Namen eines oder mehrerer Felder einer Auflistung zu ermitteln, wenn Sie nur den Wert des Feldes kennen. Sie können dann einfach die Methode `System.Enum.GetName(`*`enumType`* `As Type, `*`value`* `As Object)` verwenden (achten Sie darauf, dass Sie den `System`-Namensraum angeben, da der Compiler ansonsten das `Enum`-Schlüsselwort annimmt statt der `Enum`-Klasse). Das folgende Beispiel ermittelt den Namen der Konstante mit dem Wert 2:

```
Console.WriteLine( System.Enum.GetName(GetType(Direction), 2))
```

Über `GetNames` können Sie auch alle Konstanten durchgehen:

```
Dim name As String
For Each name In System.Enum.GetNames(GetType(Direction))
   Console.WriteLine(name)
Next
```

3.5 Variablen, Konstanten und Arrays

In Variablen und Konstanten können Sie Werte speichern und Objekte referenzieren. Eine Variable oder eine Konstante besitzt wie alle Typen einen Datentyp. Variablen können im Programm verändert werden, der Wert einer Konstante ist unveränderlich. VB.NET unterscheidet die üblichen Bereiche für die Lebensdauer und die Gültigkeit. Variablen und Konstanten können Sie innerhalb einer Prozedur, Funktion oder Methode deklarieren. Dann gilt die Variable nur innerhalb der Prozedur, Funktion bzw. Methode und lebt nur so lange, wie die Prozedur/Funktion/Methode ausgeführt wird. Eine andere Möglichkeit der Deklaration ist innerhalb eines Moduls oder einer Klasse. Variablen innerhalb eines Moduls werden zu globalen Variablen, Variablen innerhalb von Klassen werden zu Eigenschaften. Eigenschaften werden in Kapitel 4 behandelt.

3.5.1 Deklaration von Variablen und Konstanten

Implizite Deklaration

Visual Basic.NET unterstützt (leider) immer noch die veraltete implizite Variablendeklaration. Wenn Sie die Option `Explicit` in den Konfigurationseigenschaften des Projekts oder über die Anweisung

```
Option Explicit {ON | Off}
```

ausschalten, werden Variablen, die nicht explizit deklariert werden, von Visual Basic.NET bei der ersten Verwendung im Quelltext implizit als lokale Variable mit dem Typ `Object` deklariert.

Die implizite Deklaration sollten Sie grundsätzlich vermeiden. Sie erzeugen damit zwangsläufig schwer lokalisierbare Fehler. Schreiben Sie nämlich einen Variablennamen aus Versehen falsch, deklariert VB.NET einfach eine neue, leere Variable und Ihre Anweisung verwendet nicht den korrekten Wert. Außerdem sind implizit erzeugte Variablen typunsicher und langsam, weil diese den Typ `Object` *besitzen.*

Die Option `Explicit` sollte es in einer modernen Programmiersprache wie VB.NET gar nicht mehr geben. Andere Programmiersprachen kennen einen solchen Unsinn[3] wie implizite Variablendeklaration erst gar nicht.

Explizite Deklaration

Ist die Option `Explicit` eingeschaltet, müssen Sie Variablen immer explizit deklarieren. Bei der expliziten Deklaration werden Variablen folgendermaßen deklariert:

```
[Attribute] {Dim | Static | Private | Public} _
  Name1 [As Datentyp] [ = Wert] _
  [, Name2 [As Datentyp] [ = Wert]] _
  [...]
```

Die Regeln für den Bezeichner sind die üblichen: Verwenden Sie nur Buchstaben, Zahlen und den Unterstrich und beginnen Sie den Bezeichner mit einem Buchstaben oder dem Unterstrich.

Bei der Deklaration können Sie der Variablen direkt einen Wert zuweisen:

```
Dim i As Integer = 42
```

Wenn Sie keinen Wert zuweisen, wird die Variable mit einem Leerwert initialisiert. Bei Zahltypen ist das die Zahl 0, bei Zeichenketten ein Leerstring (`""`).

Wenn Sie mehrere Variablen in einer Zeile deklarieren, trennen Sie die Deklarationen durch Kommata:

```
Dim i As Integer, d As Double
```

Besitzen die einzelnen Variablen denselben Datentyp, können Sie nur einmal, am Ende angeben:

```
Dim i, j, k As Integer
```

3. Nicht, dass Sie diesen Ausbruch falsch verstehen: VB.NET ist eine sehr gute Programmiersprache, aber Microsoft will sich scheinbar von so einigen, total veralteten »Features« einfach nicht trennen (teilweise auch auf Druck von lernunwilligen Visual-Basic-6-Programmierern).

Ist für eine Variablen kein Datentyp angegeben, besitzt diese den Datentyp der nächsten Deklaration in der Liste. Ist kein Datentyp angegeben (was aber nur dann möglich ist, wenn die Option `Strict` ausgeschaltet ist), verwendet VB.NET den Typ `Object`:

```
Option Strict Off
...
Dim o ' o besitzt den Typ Object
```

Die verschiedenen Deklarations-Varianten unterscheiden sich im Gültigkeitsbereich und der Lebensdauer. Tabelle 3.8 beschreibt die verschiedenen Deklarationen. Die Deklaration innerhalb von Klassen wird in dieser Tabelle übrigens nicht beschrieben, weil Kapitel 4 näher darauf eingeht.

Deklaration	Wo?	Gültigkeitsbereich	Lebensdauer
`Dim`	In einer Prozedur oder Funktion	Automatisch lokal	Nur solange die Prozedur bzw. Funktion „lebt“
`Static`	In einer Prozedur oder Funktion	Statisch lokal	Solange das Programm läuft
`Private`	Im Deklarationsbereich eines Moduls	Modulglobal	Solange das Programm läuft
`Public`	Im Deklarationsbereich eines Moduls	Programmglobal	Solange das Programm läuft

Tabelle 3.8: Die Gültigkeitsbereiche und Lebensdauern von Variablen

Automatisch lokale Variablen

Automatisch lokale Variablen gelten nur innerhalb der Prozedur, Funktion oder Methode, in der sie deklariert sind. Bei jedem Aufruf der Prozedur/Funktion/Methode wird eine solche Variable neu erstellt und mit einem Leerwert initialisiert. Bei nummerischen Variablen ist dies die Zahl 0, bei Strings ein Leerstring (`""`).

In VB.NET gelten lokale Variablen blockweise. Wenn Sie eine Variable in einem Block deklarieren, können Sie außerhalb des Blocks nicht auf die Variable zugreifen:

```
Dim i As Integer = 0
If i = 0 Then
   ' Deklaration innerhalb einer Abfrage
   Dim s As String = "i ist 0"
End If
' Kein Zugriff außerhalb des Blocks
Console.WriteLine(s) ' dieser Zugriff ist nicht möglich
```

Daneben können Sie innerhalb einer Prozedur/Funktion/Methode keine Variable erneut deklarieren, die in einem untergeordneten Block bereits deklariert wurde:

```
Dim i As Integer = 0
If i = 0 Then
   ' Deklaration innerhalb einer Abfrage
   Dim s As String = "i ist 0"
End If
Dim s As String ' diese Deklaration ist nicht
   ' möglich, weil sie die untergeordnete
   ' Variable verdecken würde
```


Statisch lokale Variablen

Statisch lokale Variablen sind zwar wie automatisch lokale Variablen ebenfalls nur innerhalb der Prozedur, Funktion bzw. Methode gültig, sie werden jedoch, solange das Programm läuft, nicht aus dem Speicher entfernt. Damit besitzen diese Variablen beim nächsten Aufruf der Prozedur/Funktion/Methode denselben Wert wie beim letzten Verlassen derselben. So können Sie sich in einer Prozedur, Funktion oder Methode Werte dauerhaft merken, ohne die Variable modulglobal deklarieren zu müssen.

Ein einfaches Programm soll beispielsweise eine Variable bei der Betätigung eines Schalters (*cmdCount*) hochzählen und das Ergebnis jeweils ausgeben:

```
Private Sub btnCount_Click(ByVal sender As _
  System.Object, ByVal e As System.EventArgs) _
  Handles btnCount.Click
   Static count As Long
   count += 1
```

```
    MessageBox.Show(count.ToString())
End Sub
```

Modulglobale Variablen

Modulglobale (mit `Private` deklarierte) Variablen gelten innerhalb des Moduls, in dem sie deklariert werden, in allen Prozeduren und Funktionen und behalten ihren Wert, solange das Programm läuft. Modulglobale Variablen verwenden Sie immer dann, wenn Sie Daten zwischen mehreren Prozeduren und Funktionen innerhalb des Moduls austauschen müssen und der Austausch der Daten nicht über Argumente möglich ist. Im Besonderen ist dies bei Ereignisprozeduren der Fall, bei denen Sie die Argumentliste nicht erweitern können. Der Zähler aus dem letzten Beispiel soll z. B. über einen Schalter zählen und über einen andern Schalter den Zähler-Wert ausgeben:

```
Private count As Integer

Private Sub btnCount_Click(ByVal sender As _
  System.Object, ByVal e As System.EventArgs) _
  Handles btnCount.Click
   count += 1
End Sub

Private Sub btnShow_Click(ByVal sender As _
  System.Object, ByVal e As System.EventArgs) _
  Handles btnShow.Click
   MessageBox.Show(count.ToString())
End Sub
```

Programmglobale Variablen

Modulglobale Variablen gelten nur innerhalb des Moduls, in dem sie deklariert sind. Müssen Sie den Wert einer Variablen über mehrere Module austauschen, verwenden Sie programmglobale Variablen, die Sie in einem Standardmodul mit `Public` deklarieren. Programmglobale Variablen gelten im gesamten Programm. Ein gutes Beispiel für eine programmglobale Variable ist z. B. der Name einer Datenbankdatei, die Sie in Ihrem Projekt in mehreren Formularen öffnen müssen. Beim Start den Anwendung schreiben Sie diesen Dateina-

men in eine programmglobale Variable (z. B. indem Sie den Anwender den Namen in einem Dialog suchen lassen). Diese verwenden Sie immer dann, wenn Sie die Datenbank öffnen müssen. Das Zähler-Beispiel könnte man so erweitern, dass das Ausgeben in einem Extra-Formular geschieht, dann brauchen Sie auch dort eine programmglobale Variable.

Deklaration von Konstanten

Konstanten werden ähnlich wie Variablen deklariert:

```
[Attribute] [{Public | Private}] Const _
   Name [As Typ]= Ausdruck
```

Ist die Option `Strict` eingeschaltet, müssen Sie bei der Deklaration einen Datentyp angeben. Unabhängig davon müssen Sie immer direkt einen Wert zuweisen. Auf der Modul- bzw. Klassenebene können Sie die Sichtbarkeit der Konstante über die Modifizierer `Private` und `Public` definieren. Innerhalb einer Prozedur, Funktion oder Methode verwenden Sie keinen Modifizierer:

```
Const LEFT_MARGIN As Integer = 10
```

Setzen Sie überall da, wo Sie mit konstanten Werten arbeiten, die sich eventuell einmal ändern können, grundsätzlich besser Konstanten ein. Wenn ein solcher Wert später geändert werden muss, können Sie einfach die Konstante anpassen. Auch die Umsetzung einer Konstanten in eine Variable oder Eigenschaft ist einfach. Dann können Sie den Wert z. B. aus einer Konfigurationsdatei auslesen um dem Anwender die Konfiguration des Programms zu ermöglichen.

3.5.2 Arrays

Arrays erlauben das zusammenhängende Speichern mehrerer gleichartiger Informationen. Arrays verhalten sich wie eine Liste einzelner Variablen, auf die Sie über einen Namen und einen Index zugreifen können.

Das .NET-Framework stellt neben Arrays noch viele weitere Klassen zur Verfügung, über die Sie Daten zusammenhängend speichern können. Diese Klassen, die eine wesentlich flexiblere Arbeit mit den Daten ermöglichen, stelle ich in Kapitel 5 vor.

VB.NET-Arrays sind komplexe Objekte, die einige Operationen auf den gespeicherten Daten erlauben. So können Sie ein Array beispielsweise über die (statische) `Sort`-Methode sortieren und über die `IndexOf`- oder `BinarySearch`-Methode durchsuchen.

Arrays erzeugen

In VB.NET sind Arrays Objekte, die Sie ähnlich einfachen Variablen deklarieren können. Dazu geben Sie nach dem Bezeichner in Klammern den Endindex der einzelnen Dimensionen an. Ein eindimensionales Array deklarieren Sie beispielsweise so:

```
Dim intArray(2) As Integer
```

Der Datentyp der im Array gespeicherten Elemente kann ein beliebiger Typ sein.

Alternativ können Sie das Array dynamisch mit `New` erzeugen:

```
Dim intArray() As Integer
intArray = New Integer(2) {}
```

Beachten Sie, dass Sie geschweifte Klammern (die normalerweise für Initialisierungen verwendet werden) anfügen müssen, auch wenn das Array nicht initialisiert werden soll.

Diese Variante besitzt den Vorteil, dass Sie den Endindex des Array auch dynamisch zur Laufzeit festlegen können. Zudem können Sie darüber einer bereits existierenden Arrayvariablen im Programm ein vollkommen neues Array zuweisen. Die alten Arrayinhalte gehen dabei natürlich verloren.

Der Index der einzelnen Dimensionen beginnt immer bei 0. Leider hat Microsoft für VB.NET (auf Druck von traditionellen Visual-Basic-6-Programmierern) festgelegt, dass bei der Deklaration der Endindex angegeben wird und nicht, wie in anderen Sprachen, die Anzahl der Elemente. Das im obigen Beispiel deklarierte Array besitzt also drei Elemente (mit dem Index 0, 1 und 2)!

Zugriff auf die Elemente erhalten Sie, indem Sie den Index in Klammern angeben:

```
intArray(0) = 10
intArray(1) = 11
intArray(2) = 12
```

Wenn Sie auf alle Elemente eines Arrays sequenziell zugreifen wollen, können Sie dies in einer `For`-Schleife machen. Über die Eigenschaft `Length` erhalten Sie die Anzahl der im Array gespeicherten Elemente:

```
Dim i As Integer
For i = 0 To intArray.Length - 1
   Console.Write(intArray(i) & " ")
Next
```

Beachten Sie, dass `Length` *die Gesamtanzahl der gespeicherten Elemente zurückgibt. Handelt es sich um ein mehrdimensionales Array, können Sie* `Length` *nicht verwenden. Setzen Sie dann die* `GetLength`*-Methode ein, der Sie die Dimension übergeben können, deren Elementzahl Sie ermitteln wollen.*

Alternativ können Sie die `For Each`-Schleife verwenden:

```
Dim i As Integer
For Each i In intArray
   Console.Write(i & " ")
Next
```

Innerhalb dieser Schleife haben Sie über die Variable, die Sie in den Klammern angeben, Zugriff auf die einzelnen Elemente. Diese Variable ist allerdings schreibgeschützt, Sie können also nichts hineinschreiben. Handelt es sich bei den gespeicherten Elementen um Referenztypen, können Sie diese aber natürlich auch über die Variable bearbeiten.

Mehrdimensionale Arrays

Mehrdimensionale Arrays erzeugen Sie, indem Sie die Dimensionen bei der Deklaration durch Kommata angeben und bei der Erzeugung des Arrays für jede Dimensionen deren Obergrenze festlegen. Ein

zweidimensionales `String`-Array mit zwei Elementen in der ersten und drei Elementen in der zweiten Dimension erzeugen Sie z. B. so:

```
Dim persons(1, 2) As String
```

Der Zugriff auf ein solches mehrdimensionale Array erfolgt wie bei einem eindimensionalen, nur dass Sie eben die Indizes der einzelnen Dimensionen angeben müssen. Das folgenden Listing speichert die Daten von Personen im erzeugten Array:

```
persons(0, 0) = "Fred-Bogus"
persons(0, 1) = "Trumper"
persons(0, 2) = "New York"
persons(1, 0) = "Merril"
persons(1, 1) = "Overturf"
persons(1, 2) = "New York"
```

Zweidimensionale Arrays werden heute eigentlich kaum noch benötigt. Für die Speicherung von strukturierten Daten sind Instanzen von Strukturen oder Klassen viel besser geeignet. So können Sie ein eindimensionales Array verwenden, um zusammenhängende, strukturierte Daten in Objekten zu speichern. In Kapitel 4 erfahren Sie, wie Sie Klassen erzeugen, in Kapitel 5, wie Sie Daten idealerweise im Programm verwalten.

Arrays initialisieren

Speichert ein Array Werttypen, werden diese bei der Erzeugung des Arrays direkt mit einem Leerwert initialisiert. Die `Array`-Klasse ruft dazu den Standardkonstruktor des Typs auf, der den Wert in der Regel auf einen Leerwert setzt. Sie können diese Initialisierung jederzeit wiederholen, indem Sie die `Initialize`-Methode aufrufen.

Sie können ein Arrays aber auch direkt bei der Erzeugung mit definierten Werten initialisieren, indem Sie die zu speichernden Werte in geschweiften Klammern, getrennt durch Kommata angeben. Dann dürfen Sie allerdings keine Indexgrenzen angeben:

```
Dim bohemians As String() = {"Löwenherz", "Myoko", _
   "Ragnarök", "Prediger", "ZZ Top"}
```

Wollen Sie ein mehrdimensionales Array initialisieren, geben Sie die Werte der einzelnen Dimensionen wieder in geschweiften Klammern an:

```
Dim persons2(,) As String = _
   {{"Fred-Bogus", "Trumper", "New York"}, _
   {"Merril", "Overturf", "New York"}}
```

Zuweisungen von Arrays

Beim Zuweisen von Arrays müssen Sie beachten, dass Arrays Referenztypen sind. Wenn Sie ein Array einem anderen zuweisen, wird lediglich die Referenz kopiert:

```
Dim intArray1() As Integer = {1, 2, 3}
Dim intArray2() As Integer = {10, 20, 30}
intArray1 = intArray2
```

In diesem Beispiel zeigen nun beide Variablen auf dasselbe Array. Das Array das die Variable *intArray1* referenziert hat, steht nicht mehr zur Verfügung, und wird vom Garbage Collector entsorgt. Wenn Sie das Array nun über eine der Variablen verändern, ist die Veränderung natürlich auch über die andere Variable sichtbar.

Wenn Sie Arrays wirklich kopieren wollen, müssen Sie die `Clone`-Methode verwenden. Diese Methode gibt allerdings einen `Object`-Typ zurück, den Sie in den korrekten Array-Typ konvertieren müssen:

```
intArray1 = New Integer() {1, 2, 3}
intArray2 = New Integer() {10, 20, 30}
intArray2 = CType(intArray1.Clone(), Integer())
```

Da nun beide Variablen auf verschiedene Arrays verweisen, betrifft die Veränderung eines Arrays nicht mehr das andere.

Dynamische Arrays

Alle Arrays können in der Laufzeit dynamisch in der Größe verändert werden. Dazu verwenden Sie die `Redim`-Anweisung und geben den neuen Endindex des Array an:

```
Dim numbers(1) As Integer
numbers(0) = 10
numbers(1) = 20
```

```
' Redimensionieren
ReDim numbers(2)
numbers(0) = 10
numbers(1) = 20
numbers(2) = 20
```

Wenn Sie ein Array so redimensionieren, wie im Beispiel, gehen die alten Inhalte verloren. Alternativ können Sie über den Zusatz `Preserve` erreichen, dass die alten Inhalte bestehen bleiben. Damit können Sie in der Laufzeit des Programms Daten dynamisch einlesen, deren Anzahl Sie nicht zuvor ermitteln können. Das folgende Beispiel speichert einfach einige Zahlen dynamisch in einem Array, um das Prinzip zu verdeutlichen:

```
For i = 1 To 100
   ReDim Preserve numbers(i)
   numbers(i) = i
Next
```

Beachten Sie, dass das dynamische Einlesen von Daten wesentlich schneller ausgeführt wird, wenn Sie das Array dynamisch erzeugen (Seite 115). Dazu müssen Sie allerdings bei der Erzeugung wissen, wie viele Elemente zu speichern sind.

Mehrdimensionale Arrays redimensionieren

Bei mehrdimensionalen Arrays können Sie nur die letzte Dimension redimensionieren.

Arrays leeren

Die Elemente eines Array können Sie über

`Erase` *`Arrayvariable`*

oder

`Arrayvariable` `= Nothing`

löschen. Das Objekt wird dann freigegeben (aber erst später durch den Garbage Collector entsorgt). Die Arrayvariable zeigt danach nicht mehr auf ein gültiges Objekt und muss dynamisch neu erzeugt oder redimensioniert werden, um wieder verwendet werden zu können.

Methoden und Eigenschaften eines Arrays

Alle VB.NET-Arrays sind von der Klasse `Array` abgeleitet und besitzen deswegen die Eigenschaften und Methoden dieser Klasse. Wie viele andere .NET-Klassen besitzt auch die `Array`-Klasse Instanz- und Klassenelemente. Die wichtigsten Instanzeigenschaften und -methoden zeigt Tabelle 3.9.

Eigenschaft / Methode	Beschreibung
`Clone() As Array`	erzeugt eine Kopie eines Arrays.
`CopyTo(` *`array`* `As Array,` *`index`* `As Integer)`	kopiert den Inhalt eines Arrays in ein anderes Array. In *`index`* geben Sie den Index des Zielarrays an, ab dem das Kopieren beginnen soll.
`GetLength(` *`dimension`* `As Integer) As Integer`	ermittelt die Anzahl der Elemente in einer gegebenen Dimension.
`Initialize()`	Über diese Methode können Sie die einzelnen Elemente eines Arrays auf einen Standardwert (in der Regel ein Leerwert) initialisieren. `Initialize` ruft dazu einfach den Standardkonstruktor der gespeicherten Typen auf.
`Length`	Diese Eigenschaft gibt die Gesamtanzahl der gespeicherten Elemente zurück .
`Rank`	Über diese Eigenschaft können Sie die Gesamtanzahl der Dimensionen ermitteln.

Tabelle 3.9: Die wichtigsten Instanzeigenschaften und -methoden der Array-Klasse

Wie viele anderer .NET-Klassen, besitzt auch die `Array`*-Klasse Klassenmethoden. Die wichtigsten beschreibt Tabelle 3.10.*

Eigenschaft / Methode	Beschreibung

`BinarySearch(`
`  array As Array,`
`  value As Object`
`  [, comparer As`
`  IComparer]) As`
`  Integer`
`BinarySearch(`
`  array As Array,`
`  index As Integer,`
`  length As Integer,`
`  value As Object`
`  [, comparer As`
`  IComparer]) As`
`  Integer`
	Über diese Methode können Sie im Array einen Wert über die schnelle binäre Suche ermitteln. Das Array muss dazu sortiert sein. Wenn Sie im dritten Argument einen Index angeben, durchsucht die Methode ab diesem Index die in *length* angegebene Anzahl Elemente. Geben Sie keinen Index an, wird das ganze Array durchsucht. Im letzten Argument können Sie ein Objekt übergeben, das die Schnittstelle `IComparer` implementiert, über deren `Compare`-Methode der Vergleich der Elemente erfolgt. So können Sie benutzerdefinierte Suchen programmieren. Die Standarddatentypen implementieren diese Schnittstelle bereits, weswegen Sie diese auch ohne `IComparer`-Objekt durchsuchen können. Wie Sie eigene Klassen programmieren, die die binäre Suche ermöglichen, zeige ich in Kapitel 4.

`IndexOf(`
`  array As Array,`
`  value As Object`
`  [, startIndex As`
`  Integer]`
`  [, count As Integer])`
`  As Integer`
	Über `IndexOf` können Sie ein Array sequenziell (und damit langsam) durchsuchen.

`LastIndexOf(`
`  array As Array,`
`  value As Object`
`  [, startIndex As`
`  Integer]`
`  [, count As Integer])`
`  As Integer`
	`LastIndexOf` durchsucht ein Array sequenziell, ermittelt aber den Index des letzten gespeicherten Werts, der zum Suchwert identisch ist.

Eigenschaft / Methode	Beschreibung

`Sort(`*`array`* `As Array` `[,`*`index`* `As Integer,` *`length`* `As Integer]` `[,`*`comparer`* `As IComparer]) As Array`
`Sort(`*`array`* `As Array,` *`comparer`* `As IComparerf` `As Array`
`Sort(` *`keys`* `As Array,` *`items`* `As Array` `[,` *`index`* `As Integer,` *`length`* `As Integer]` `[,` *`comparer`* `As IComparer])` `As Array`
`Sort(` *`keys`* `As Array,` *`items`* `As Array,` *`comparer`* `As IComparer)` `As Array`
	Über diese Methode können Sie ein Array ganz oder teilweise sortieren. Die gespeicherten Typen müssen dazu die `IComparer`-Schnittstelle implementieren, was bei den meisten Typen bereits der Fall ist. Für eine benutzerdefinierte Sortierung können Sie ein separates Objekt übergeben, dass diese Schnittstelle implementiert. In Kapitel 4 zeige ich, wie Sie eigene Klassen erzeugen, die sortierbar sind.

Tabelle 3.10: Die wichtigsten Klassenmethoden der Array-Klasse

3.5.3 Namensrichtlinien

Bei der Programmierung macht es immer Sinn, sich an gewisse Richtlinien bei der Vergabe von Bezeichnern zu halten. Damit machen Sie es anderen Programmierern leichter, Ihren Quelltext zu verstehen. In verschiedenen Programmiersprachen existieren unterschiedliche Notationen für die Benennung. C++-Programmierer verwenden z. B. meist die so genannte »Ungarische Notation«, Visual-Basic-6-Programmierer wenden die »Reddik-Konvention« an. Für C# beschreibt Microsoft in der C#-Dokumentation eine Richtlinie für die Benennung von Bezeichnern, die recht übersichtlich ist. Diskussionen in ver-

schiedenen Newsgroups zeigen, dass sich diese Richtlinien durchsetzen. Für VB.NET steht zur Diskussion, ob die bisher verwendete Reddik-Namenskonvention weiter verwendet wird. Bei dieser Konvention wird dem eigentlichen Variablennamen zumindest ein klein geschriebenes Präfix vorangestellt, das den Datentyp kennzeichnet. Eine Stringvariable, die einen Vornamen speichern soll, wird beispielsweise so deklariert:

```
Dim strVorname As String
```

Zusätzlich dazu wird meist noch ein Zeichen vorangestellt, das den Gültigkeitsbereich kennzeichnet. »g« steht z. B. für programmglobal, »m« für modulglobal.

Ich denke, dass diese Konvention mittlerweile überholt ist:

- Das .NET-Framework enthält so viele Typen, dass Sie gar nicht genügend Präfixe finden, die diese Typen kennzeichnen.
- Mit der Option `Strict` ist VB.NET typsicher. Es ist dann also unmöglich, einem Typen einen unpassenden Typ zuzuweisen. Die Typpräfixe verlieren damit an Bedeutung.
- In VB.NET sollten Sie, auch wenn die veralteten Module noch möglich sind, grundsätzlich (modern) objektorientiert programmieren. Die OOP kennt keine globalen Variablen, deren Kennzeichnung in strukturierten Programmen noch sinnvoll wäre.

Da VB.NET prinzipiell sehr viel Ähnlichkeit mit C# besitzt, verwende ich auch in VB.NET die Konventionen, die Microsoft vorschlägt.

Die Grundlage der Benennung von Bezeichnern ist dabei das so genannte PascalCasing und das camelCasing. Beim PascalCasing beginnt der Bezeichner mit einem Großbuchstaben und wird dann klein weitergeschrieben. Besteht der Bezeichner aus mehreren Worten, wird jedes Wort wieder mit einem Großbuchstaben begonnen:

```
Public LeftMargin As Integer
```

Das camelCasing ist ähnlich, nur dass der Bezeichner mit einem Kleinbuchstaben begonnen wird:

```
Dim leftMargin As Integer
```

Das PascalCasing wird hauptsächlich bei öffentlichen Elementen verwendet, das camelCasing bei privaten oder lokalen Elementen. Tabelle 3.11 fasst die Richtlinien zusammen.

Element	Namensrichtlinie
Klasse	▪ PascalCasing
Schnittstelle	▪ PascalCasing ▪ »I« als Präfix
Aufzählungen (Enums)	▪ PascalCasing (für den Namen der Aufzählung und die Werte)
Eigenschaften	▪ Bennennung mit Substantiven (Hauptwörtern) oder Substantiv-Phrasen (z. B. *Color*, *FirstName*) ▪ PascalCasing für öffentliche Eigenschaften ▪ camelCasing für private und geschützte Eigenschaften
Methoden	▪ Bennennung mit Verben oder Verb-Phrasen (z. B. *Remove*, *RemoveAll*) ▪ PascalCasing
Ereignisse	▪ Benennen Sie Ereignisbehandlungsmethoden mit dem »EventHandler«-Suffix (z. B. *MouseEventHandler*) ▪ Verwenden Sie die zwei Argumente *sender* und *e* ▪ Bennennen Sie Ereignisargument-Klassen mit dem Suffix »EventArgs« (z. B. *MouseEventArgs*) ▪ PascalCasing
Argumente	▪ Verwenden Sie aussagekräftige Namen ▪ camelCasing

Tabelle 3.11: Namensrichtlinien für C#-Programme

3.6 Ausdrücke und Operatoren

3.6.1 Arithmetische Ausdrücke und Operatoren

Arithmetische Ausdrücke ergeben einen Wert (eine Zahl, ein Datum, eine Zeichenkette), der in weiteren Ausdrücken oder in Zuweisungen verwendet werden kann. Der Ausdruck

```
1 + 1
```

ergibt z. B. den Wert 2 (wenn ich richtig gerechnet habe ...).

Arithmetische Ausdrücke verwenden die in Tabelle 3.12 beschriebenen Operatoren.

Operator	Bedeutung
()	Klammern; Verschieben der Rechenpriorität
+	Addition
&	Addition von Strings
-	Subtraktion
*	Multiplikation
/	Division
\	Integer-(Ganzzahl-)Division
`Mod`	Modulo-(Restwert-)Division
^	Potenzierung

Tabelle 3.12: Die arithmetischen Operatoren

Die bekannten Operatoren beschreibe ich hier nicht. Operatoren wie \ und `Mod` müssen allerdings u. U. etwas näher erklärt werden.

Der \-Operator führt eine Ganzzahl-Division zweier Zahlen durch. Das Ergebnis ist immer eine ganze Zahl, bei der eventuelle Nachkommastellen abgeschnitten sind. Die Operation 5 \ 2 ergibt z. B. 2. Sind die Operanden Dezimalzahlen, werden diese zuvor gerundet. Die Ganzzahl-Division 5.7 \ 2 führt demnach zu dem Ergebnis 3 (6 \ 2).

Der `Mod`-Operator ergibt dagegen den Restwert des linken Operators, der sich nicht mehr durch den rechten Operator teilen ließ. Das Ergebnis von 5 Mod 2 ist also 1. Dezimalzahlen werden genau wie bei \ gerundet: 5.7 Mod 2 ergibt 0.

Über den Operator + können Sie auch Strings verketten:

```
Dim firstName As String = "Donald"
Dim lastName As String = "Duck"
Console.WriteLine(firstName + " " + lastName)
```

VB.NET kennt aber auch den speziellen Operator & für die Verkettung von Strings. Dieser Operator besitzt eine Bedeutung, wenn die Option `Strict` ausgeschaltet ist und Sie Zahltypen zu einem String zusammenfassen, ohne diese explizit zu konvertieren. Die Verwendung des +-Operators würde eine normale Addition bewirken:

```
Option Strict Off
...
Dim i1 As Integer = 4
Dim i2 As Integer = 2
Dim antwort As String
antwort = i1 + i2
Console.WriteLine(antwort) ' Ergebnis: "6"
```

Wenn Sie dann mit & addieren, addiert der Compiler korrekt die aus den Zahlwerten resultierenden Strings:

```
antwort = i1 & i2 ' Ergebnis: "42"
```

Der &-Operator wird allerdings unwichtig, wenn Sie die Option `Strict` einschalten, da Sie Zahltypen dann explizit konvertieren müssen:

```
antwort = i1.ToString() + i2.ToString()
```


3.6.2 Logische und Bitoperationen

Die meisten Operatoren für logische und bitweise Operationen sind in VB.NET identisch. VB.NET erkennt meist aus dem Kontext, ob eine logische oder eine bitweise Operation verwendet werden soll. Wenn diese automatische Erkennung jedoch fehlschlägt, müssen Sie mit Klammern nachhelfen.

Operator	Bedeutung für Ausdrücke	Bedeutung für nummerische Werte
`Not`	Negiert einen Ausdruck	Kippt alle Bits um (aus 00001111 wird z. B. 11110000)
`And`	*Ausdruck1* `And` *Ausdruck2* ergibt `True`, wenn beide Ausdrücke `True` ergeben. Der Compiler wertet immer alle Teilausdrücke aus.	*Num1* `And` *Num2*: Im Ergebnis ist Bit *n* gesetzt, wenn Bit *n* in beiden nummerischen Werten ebenfalls 1 ist (0101 `And` 0100 ergibt z. B. 0100).

Operator	Bedeutung für Ausdrücke	Bedeutung für nummerische Werte
`AndAlso`	arbeitet wie `And`. Der Compiler wertet aber rechte Teilausdrücke nicht weiter aus, wenn linke Teilausdrücke dazu führen, dass das Ergebnis `False` wird.	keine Bedeutung
`Or`	*Ausdruck1* `Or` *Ausdruck2* ergibt `True`, wenn einer der beiden Ausdrücke `True` ergibt. Sind beide Ausdrücke `False`, ist das Ergebnis ebenfalls `False`. Der Compiler wertet immer alle Teilausdrücke aus.	*Num1* `Or` *Num2*: Im Ergebnis ist Bit *n* gesetzt, wenn Bit *n* in einem der beiden nummerischen Werte ebenfalls 1 ist (0101 `Or` 1001 ergibt z. B. 1101).
`OrElse`	arbeitet wie `Or`. Der Compiler wertet aber rechte Teilausdrücke nicht weiter aus, wenn linke Teilausdrücke dazu führen, dass das Ergebnis `False` wird.	keine Bedeutung
`Xor` (Exklusiv Oder)	*Ausdruck1* `Xor` *Ausdruck2* ergibt `True`, wenn einer der beiden Ausdrücke `True` ist und der andere `False`.	*Num1* `Xor` *Num2*: Im Ergebnis ist Bit *n* gesetzt, wenn Bit *n* in einem Wert 1 und im anderen 0 ist (0101 `Xor` 1001 ergibt z. B. 1100).

Tabelle 3.13: Die logischen und bitweisen Operatoren von VB.NET

Die wichtigsten Operatoren hier sind `Not`, `And`, `AndAlso`, `Or` und `OrElse`.

Der folgende Quellcode überprüft, ob jetzt gerade Samstag oder Sonntag nach 12 Uhr ist:

```
Dim dt As Date = Date.Now
If ((dt.DayOfWeek = DayOfWeek.Saturday) Or _
  (dt.DayOfWeek = DayOfWeek.Sunday)) And _
  (dt.TimeOfDay.Hours > 12) Then
   Console.WriteLine("Jetzt ist Samstag oder " & _
```

```
    "Sonntag nach 12 Uhr")
Else
   Console.WriteLine("Jetzt ist nicht Samstag " & _
    "oder Sonntag nach 12 Uhr")
End If
```

Wenn Sie die logische Bedeutung dieser Operatoren verwenden, passen Sie auf die Priorität der Operatoren auf, wenn Sie mehrere Operatoren in einem Ausdruck verwenden: `Not` wird vor `And` / `AndAlso` ausgewertet und `And` / `AndAlso` vor `Or` / `OrElse`. Zur Sicherheit sollten Sie kombinierte logische Operationen immer klammern. Dies gilt besonders, wenn Sie `Not` verwenden. Aufpassen müssen Sie auch, wenn der Compiler den Operator auch als bitweisen Operator auswerten kann. Klammern hilft hier in der Regel.

Bei logischen Ausdrücken wertet VB.NET immer alle Teilausdrücke aus, wenn Sie `And` oder `Or` verwenden. Das kann manchmal zum Problem werden, beispielsweise dann, wenn Sie einen String daraufhin überprüfen, ob die gespeicherte Zeichenkette nummerisch ist und einen bestimmten Wert besitzt:

```
Dim s As String = "abc"
If IsNumeric(s) And Convert.ToDouble(s) > 0 Then
  ...
```

Dieses Beispiel ergibt eine Ausnahme vom Typ `FormatException`, da die Konvertierung in einen `Double`-Wert nicht möglich ist. Um dieses Problem zu vermeiden, bietet Ihnen VB.NET die Operatoren `AndAlso` und `OrElse`. Wenn Sie diese Operatoren einsetzen, wertet VB.NET rechte Teilausdrücke nicht weiter aus, wenn ein linker Teilausdruck dazu führt, dass das Ergebnis `False` werden würde:

```
Dim s As String = "abc"
If IsNumeric(s) AndAlso Convert.ToDouble(s) > 0 Then
  ...
```

Dieses Beispiel ergibt keine Ausnahme, weil der linke Teilausdruck `False` ergibt und der rechte damit nicht mehr ausgewertet wird.

Verwenden Sie für logische Ausdrücke möglichst immer `AndAlso` *und* `OrElse`*. Wenn Sie dann noch auf die Reihenfolge der Ausdrücke achten, vermeiden Sie Probleme, die ansonsten mit* `And` *bzw.* `Or` *entstehen können. Zudem verhält sich VB.NET mit diesen Operatoren wie andere Sprachen, wie beispielsweise C++ und C#.*

3.6.3 Zuweisungen

Für Zuweisungen stellt VB.NET nicht nur den Operator =, sondern auch noch einige spezielle Operatoren zur Verfügung. Eine einfache Zuweisung sieht z. B. so aus:

```
i = 1
```

Die erweiterten Operatoren erlauben die Zuweisung eines Ausdrucks bzw. Werts, den Sie gleich noch über eine arithmetische Operation mit den Operanden berechnen. Die Anweisung

```
i += 1
```

addiert beispielsweise den Wert 1 auf die Variable i und bedeutet soviel wie

```
i = i + 1
```

Operator	Bedeutung
=	einfache Zuweisung
+=	Additionszuweisung. `i += 1` entspricht `i = i + 1`.
&=	Additionszuweisung für Strings
-=	Subtraktionszuweisung. `i -= 1` entspricht `i = i - 1`.
*=	Multiplikationszuweisung. `i *= 1` entspricht `i = i * 1`.
/=	Divisionszuweisung. `i /= 1` entspricht `i = i / 1`.
\=	Ganzzahldivisionszuweisung. `i \= 1` entspricht `i = i \ 1`.
^=	Potenzierungszuweisung. `i ^= 2` entspricht `i = i ^ 2`.

Tabelle 3.14: Die Zuweisungsoperatoren

3.6.4 Vergleiche

Für Vergleiche bietet VB.NET die üblichen Operatoren. Ein Vergleichsausdruck ergibt immer den booleschen Wert `True` (wenn der Vergleich wahr ist) oder `False` (wenn der Vergleich falsch ist). Der Vergleich auf Gleichheit verwendet den Operator =.

Operator	Operation
=, <, <=, >, >=, <>	Arithmetischer Vergleich. Strings werden von links nach rechts entsprechend ihrem ANSI-Code verglichen.
`Like`	Vergleicht zwei String-Ausdrücke. Für den Vergleich können Wildcards (*, ?, #, [*Zeichenliste*]) verwendet werden.
`Is`	Ermittelt, ob ein Objekt auf `Nothing` zeigt oder ob zwei Objektvariablen dasselbe Objekt referenzieren.

Tabelle 3.15: Die Vergleichsoperatoren

Die normalen Vergleichsoperatoren können Sie nur auf die Standard-Werttypen (nicht auf Strukturen) und auf den Referenztyp `String` anwenden. Bei allen Referenztypen außer String lässt der Compiler die Verwendung dieser Operatoren nicht zu. Das ist leider auch dann der Fall, wenn eine unter C# (oder C++) entwickelte Klasse diese Operatoren implementiert.

Strings werden immer Zeichen für Zeichen nach dem Unicode der Zeichen verglichen. Die linken Zeichen besitzen dabei eine höhere Priorität als die rechten. Ist der Unicode eines Zeichens größer oder kleiner als der Unicode des korrespondierenden Zeichens, wird der Vergleich damit beendet. Der String "10" ist demnach also kleiner als der String "2". Beim Vergleich verwendet der Compiler zudem die Option `Compare`. Ist diese auf `Binary` eingestellt, wird Groß- und Kleinschreibung unterschieden. Setzen Sie diese Option auf `Text`, wenn Sie diese Unterscheidung nicht vornehmen wollen. Wie bei den anderen Optionen können Sie die Option `Compare` in den Projekteigenschaften oder über die Anweisung

```
Option Compare {Text | Binary}
```

ganz oben in der Datei einstellen.

Vergleiche von Referenztypen können Sie mit `Is` vornehmen. Damit können Sie überprüfen, ob zwei Referenztypen auf dieselbe Instanz zeigen:

```
Class Demo
   Public Value As Integer
End Class

Sub Main()
   ' Neue Instanz der Demo-Klasse erzeugen
   Dim d1 As Demo = New Demo()
   d1.Value = 10
   ' Eine zweite Variable auf diese Instanz verweisen
   Dim d2 As Demo = d1
   ' Überprüfen, ob beide Variablen auf dieselbe
   ' Instanz zeigen
   If d1 Is d2 Then
      Console.WriteLine("d1 zeigt auf dieselbe " & _
         "Instanz wie d2")
   End If
End Sub
```

Mit `Is` können Sie auch überprüfen, ob ein Referenztyp überhaupt eine Instanz verwaltet, indem Sie mit `Nothing` vergleichen:

```
Dim d3 As Demo
If d3 Is Nothing Then
   Console.WriteLine("d3 zeigt nicht auf eine Instanz")
End If
```

Wenn Sie zwei Referenztypen daraufhin überprüfen wollen, ob diese dieselben Werte speichern, müssen Sie die von `Object` geerbte `Equals`-Methode verwenden, die in den Referenztypen des .NET-Framework (und hoffentlich auch in den Referenztypen anderer Hersteller) überschrieben ist. Das folgende Beispiel setzt zur Demonstration zwei Instanzen der Klasse `System.Drawing.Point` ein:

```
Dim p1 As System.Drawing.Point
p1 = New System.Drawing.Point(10, 50)
Dim p2 As System.Drawing.Point
p2 = New System.Drawing.Point(10, 50)
```

```
If p1.Equals(p2) Then
   Console.WriteLine("p1 ist gleich p2")
Else
   Console.WriteLine("p1 ist ungleich p2")
End If
```

Einige Typen wie `String` *besitzen zusätzlich eine* `CompareTo`*-Methode, über die Sie herausfinden können, ob der Typ kleiner, gleich oder größer ist als ein anderer. Die Rückgabe dieser Methode ist <0 (der Typ ist kleiner), 0 (beide Typen sind gleich) oder >0 (der Typ ist größer).*

3.7 Verzweigungen und Schleifen

VB.NET kennt natürlich auch die gängigen Schleifen und Verzweigungen. Bevor ich diese beschreibe, soll ein Tipp eventuelle Probleme vermeiden: Wenn Sie in Visual Studio programmieren, versehentlich eine Endlosschleife produzieren (eine Schleife, die nie beendet wird) und das Programm ausführen, scheint Ihr Programm nicht mehr zu reagieren, weil der Prozessor in diesem Fall sehr stark ausgelastet ist. Statt das Programm über den Task-Manager »abzuschießen«, betätigen Sie lieber in Visual Studio einfach Strg + Pause um das Programm zu unterbrechen.

3.7.1 Die If-Verzweigung

Die `If`-Verzweigung ist in VB.NET gegenüber anderen Programmiersprachen um optionale `ElseIf`-Blöcke erweitert:

```
If Bedingung1 Then
   [Anweisungsblock 1]]
[ElseIf Bedingung2 > Then
   [Anweisungsblock 2]]
[...]
[Else
   [Anweisungsblock n]]
End If
```

Der optionale `ElseIf`-Block kann beliebig oft in die `If`-Verzweigung eingebaut werden. Die Bedingung eines `ElseIf`-Blocks wird nur dann

geprüft, wenn keine der Bedingung vorstehender Blöcke erfüllt war. Dasselbe gilt für den `Else`-Block, nur dass hier keine Bedingung mehr überprüft wird. Eine einfache `If`-Verzweigung, die überprüft, ob heute Wochenende oder ein Arbeitstag ist, sieht so aus:

```
Dim wd As Integer
wd = Weekday(Now)
If (wd = 7) Or (wd = 1) Then
   ' Samstag oder Sonntag
   Console.WriteLine("Es ist Wochenende, " & _
      "nur Buchautoren müssen arbeiten")
Else
   Console.WriteLine("Es ist kein Wochenende, " & _
      "alle müssen arbeiten")
End If
```

Mit `ElseIf`-Blöcken können Sie in einer Verzweigung mehrere unterschiedliche Bedingungen überprüfen. Die Möglichkeit *unterschiedliche* Bedingungen zu überprüfen, unterscheidet die `If`-Verzweigung auch von der `Select Case`-Verzweigung (bei der Sie nur eine Bedingung überprüfen können). Das folgende (einfache) Beispiel überprüft eine Variable darauf, ob diese einen Wert größer 100, größer 10 oder größer 1 speichert:

```
Dim i As Integer = 11
If i > 100 Then
   Console.WriteLine("i ist größer 100")
ElseIf i > 10 Then
   Console.WriteLine("i ist größer 10")
ElseIf i > 1 Then
   Console.WriteLine("i ist größer 1")
Else
   Console.WriteLine("i ist kleiner oder gleich 1")
End If
```

Kurzform der If-Verzweigung in einer Zeile

Sie können die `If`-Verzweigung auch einzeilig schreiben, dann allerdings ohne `ElseIf`-Blöcke:

```
If Bedingung Then Anweisung [Else Anweisung]
```

Mit einem Trick können Sie auch mehrere Anweisungen in einem Block dieser Verzweigung verwenden: Trennen Sie die einzelnen Anweisungen einfach durch einen Doppelpunkt.

Wenn Sie eine `If`*-Verzweigung mit mehreren* `ElseIf`*-Blöcken benötigen, weil die Bedingungen unterschiedlich sind, speichern Sie das Ergebnis von Funktionsaufrufen in Variablen, wenn dieses Ergebnis in mehreren Bedingungen überprüft wird. Die Beispiele zur* `If`*-Verzweigung verwenden diese Technik.*

3.7.2 Select Case

Die `Select Case`-Verzweigung überprüft einen Ausdruck auf mehrere mögliche Ergebnisse:

```
Select Case Testausdruck
   [Case Ausdrucksliste 1
      [Anweisungsblock 1]]
   [Case Ausdrucksliste 2
      [Anweisungsblock 2]]
   [...]
   [Case Else
      [Anweisungsblock n]]
End Select
```

Auch die `Select Case`-Verzweigung ist gegenüber anderen Programmiersprachen erheblich erweitert. So können Sie im Testausdruck einen beliebigen arithmetischen oder logischen Ausdruck angeben (andere Programmiersprachen erlauben hier nur Ganzzahl-Datentypen).

Die einzelnen `Case`-Blöcke überprüfen, ob der Ergebniswert des Testausdrucks mit einem der Werte in der Ausdrucksliste übereinstimmt. Die Ausdrucksliste kann, durch Kommata getrennt, einzelne Werte und Bereiche (*Wert1* `To` *Wert2*) enthalten. Im Gegensatz zu manchen anderen Programmiersprachen können Sie in VB.NET komplexe Ausdrücke und auch Strings in der Ausdrucksliste verwenden. In der Praxis werden jedoch normalerweise lediglich einfache Werte und Listen von Werten eingesetzt.

Ist einer der Werte in der Ausdrucksliste mit dem Wert des Testausdrucks identisch, wird der zugehörige Anweisungsblock ausgeführt. Danach ist die `Select Case`-Verzweigung, anders als z. B. in C++ oder C#, beendet. Der optionale `Case Else`-Block wird ausgeführt, wenn keiner der vorherigen Blöcke ausgeführt wurde.

Das folgende Beispiel lässt den Anwender eine Zahl eingeben und wertet diese dann aus:

```
Console.Write("Ihre Lieblingszahl: ")
i = Convert.ToInt32(Console.ReadLine())
Select Case i
   Case 7
      Console.WriteLine("7 ist eine gute Zahl.")
   Case 42
      Console.WriteLine("Cool. Trinken Sie einen " & _
        "pangalaktischen Donnergurgler.")
   Case 3
      Console.WriteLine("3 ist OK.")
   Case Else
      Console.WriteLine("Denken Sie noch einmal " & _
        "darüber nach.")
End Select
```

Tipp

Verwenden Sie, wo immer es geht, `Select Case`, wenn Sie mehrere Fälle überprüfen wollen. `Select Case` wird wesentlich schneller ausgeführt als eine äquivalente `If`-Verzweigung mit mehreren `ElseIf`-Blöcken. Beim `Select Case` wird der Testausdruck nur einmal ausgewertet, `ElseIf`-Bedingungen werden jedoch der Reihe nach einzeln ausgewertet.

3.7.3 Die kopfgesteuerten Schleifen

VB.NET kennt drei kopfgesteuerte Schleifen (obwohl eine eigentlich ausreichen würde).

Die `Do While`-Schleife läuft, solange eine Bedingung erfüllt ist:

```
Do While Bedingung
   Anweisungen
Loop
```

Da die Bedingung im Schleifenkopf erfolgt, wird die Schleife nur dann ausgeführt, wenn die Bedingung beim ersten Eintritt in die Schleife wahr ist.

Das folgende Beispiel schleift solange der Inhalt einer Variablen kleiner 4 ist:

```
Dim i As Integer = 1
i = 1
Do While i < 4
   Console.WriteLine(i)
   i += 1
Loop
```

Die `Do While`-Schleife ist funktional identisch mit der `While`-Schleife:

```
While Bedingung
   Anweisungen
End While
```

Die `Do While`-Schleife kann aber im Gegensatz zur `While`-Schleife mit `Exit Do` explizit verlassen werden.

Die `Do Until`-Schleife unterscheidet sich auch nicht allzu sehr von der `Do While`-Schleife. Der Unterschied ist lediglich der, dass die `Do Until`-Schleife läuft, *bis* die Bedingung erfüllt ist:

```
Do Until Bedingung
   Anweisungen
Loop
```

Wollen Sie eine `Do Until`-Schleife in eine `Do While`-Schleife umwandeln, müssen Sie lediglich die Bedingung umformulieren. Die Beispiel-Schleife oben kann z. B. auch so formuliert werden:

```
Dim i As Integer = 1
Do Until i > 3
   Console.WriteLine(i)
   i += 1
Loop
```

3.7.4 Die fußgesteuerten Schleifen

Die `Do`-Schleifen existieren auch in einer Variante, die die Bedingung am Schleifenfuß prüft:

```
Do
   Anweisungen
Loop While Bedingung

Do
   Anweisungen
Loop Until Bedingung
```

Die Bedingungsprüfung am Schleifenfuß bewirkt, dass die Schleife mindestens einmal durchlaufen wird. Wenn der Programmcode in der Schleife also auf jeden Fall einmal ausgeführt werden soll, verwenden Sie eine solche Schleife.

3.7.5 Do-Schleife ohne Bedingung

Die `Do`-Schleife existiert noch in einer Variante ohne Bedingungsprüfung:

```
Do
   Anweisungen
Loop
```

Diese »Endlosschleife« können Sie verwenden, wenn die Bedingung zu komplex ist, um im Kopf oder Fuß der Schleife geprüft zu werden, oder wenn beim Eintreten der Bedingung zusätzliche Anweisungen ausgeführt werden sollen. Prüfen Sie die Bedingung einfach innerhalb der Schleife und verlassen Sie die Schleife dann mit `Exit Do`.

3.7.6 Die For Next-Schleife

Die `For Next`-Schleife ist eine einfache Zählschleife, die eine Zahl-Variable automatisch hochzählt, bis diese einen Wert besitzt, der größer ist als der in der Schleife angegebene Endwert:

```
For Zähler = Startwert To Endwert [Step Schrittweite]
   Anweisungen
Next [Zähler]
```

Die Schrittweite ist standardmäßig 1. Mit dem optionalen `Step` können Sie die Schrittweite aber auch auf größere oder negative Werte oder auch auf Dezimalwerte setzen. So können Sie z. B. von 3 nach 1 rückwärts zählen:

```
Dim i As Long
For i = 3 To 1 Step -1
   Console.WriteLine(i)
Next
```

Die `For Next`-Schleife können Sie mit `Exit For` explizit verlassen.

3.7.7 Die For Each-Schleife

Mit der `For Each`-Schleife können Sie alle Elemente eines Arrays oder einer Auflistung (Collection) ohne Kenntnis der Anzahl der gespeicherten Elemente durchgehen.

```
For Each Element In {Array | Auflistung}
   [Anweisungen]
Next [Element]
```

Element ist eine Variable vom Typ der gespeicherten Daten (oder vom Typ `Object`). Das folgende Beispiel erzeugt ein Array und geht dieses mit `For Each` durch:

```
Dim value As Integer
Dim numbers() As Integer = {11, 12, 13, 14, 15}
For Each value In numbers
   Console.WriteLine(value)
Next
```

Eine `For Each`-Schleife können Sie, wie eine `For Next`-Schleife, mit `Exit For` vor dem impliziten Beenden explizit beenden.

`For Each` besitzt einen Vorteil gegenüber der Iteration mit einem Integer-Index: Wenn sich während der Iteration die Anzahl der Elemente ändert, weil Sie z. B. einzelne Elemente löschen, führt `For Each` *nicht* zu einem Fehler.

3.7.8 Die With-Anweisung

Über die `With`-Anweisung können Sie die Elemente eines Objekts in Anweisungen verwenden, ohne das Objekt jedes Mal neu angeben zu müssen.

```
With Objekt
   [Anweisungen]
End With
```

Methoden und Eigenschaften des Objekts geben Sie innerhalb des `With`-Blocks nur mit einem Punkt als Präfix an. Damit erleichtern Sie sich die Schreibarbeit, aber dem Compiler auch einiges an Programmaufwand. Sie können die `With`-Anweisung auch verschachteln. Das folgende Beispiel arbeitet mit zwei Klassen, die Klasse `Point` definiert einen Punkt, die Klasse `Rect` ein Rechteck. `Rect` verwendet eine Instanz von `Point` für die Festlegung der Position des Rechtecks:

```
Class Point
   Public x As Integer
   Public y As Integer
End Class

Class Rect
   Public Position As Point = New Point()
   Public Height As Integer
   Public Width As Integer
End Class
```

Im Programm wird eine Instanz von `Rect` erzeugt und über `With` auf deren Eigenschaften zugegriffen:

```
Sub Main()
   Dim r As Rect = New Rect()
   With r
      With .Position
         .x = 10
         .y = 20
      End With
      .Height = 50
      .Width = 100
   End With
End Sub
```

3.8 Verzweigungs-Funktionen

VB.NET kennt einige Funktionen, die abhängig von einer Bedingung oder einem Index unterschiedliche Werte zurückgeben. Besonders die `IIf`- Funktion ist sehr hilfreich, da Sie damit in vielen Situationen die komplexere `If`-Verzweigung vermeiden können.

3.8.1 IIf

Die `IIf`-Funktion (Inline If) wertet einen logischen Ausdruck aus und gibt je nach Ergebnis des Ausdrucks einen von zwei übergebenen Werten zurück:

```
IIf(Expression As Boolean, TruePart As Object, _
  FalsePart As Object) As Object
```

Sie können diese Funktion sehr gut einsetzen, wenn abhängig von einer Bedingung der eine oder der andere Wert verwendet werden soll. Das folgende Beispiel ermittelt, ob jetzt gerade Vor- oder Nachmittag ist:

```
Console.WriteLine("Jetzt ist " & _
  IIf(Date.Now.Hour < 12, "Vormittag", "Nachmittag"))
```

3.8.2 Switch

Die `Switch`-Funktion arbeitet ähnlich `IIf`, nur dass Sie einzelne Paare von Bedingung und zurückzugebendem Wert übergeben.

```
Switch(ParamArray VarExpr() As Object) As Object
```

Wird eine Bedingung wahr, gibt `Switch` den Wert zurück, der zu dieser Bedingung gehört. Die einzelnen Bedingungen werden von links nach rechts geprüft. Ergibt eine Bedingung `False`, wird die nächste geprüft etc.

Der Namensraum `System.Diagnostics` enthält eine Klasse mit dem Namen `Switch`. Wenn Sie diesen Namensraum importiert haben (was per Voreinstellung der Fall ist), müssen Sie `Switch` voll qualifiziert verwenden. Das folgende Beispiel überprüft, ob heute Samstag, Sonntag oder ein Arbeitstag ist:

```
Console.WriteLine(Microsoft.VisualBasic.Switch( _
  Date.Now.DayOfWeek = 6, "Samstag", _
  Date.Now.DayOfWeek = 0, "Sonntag", _
  True, "Arbeitstag"))
```

Als kleiner Trick wurde im letzten Argument-Paar als Bedingung `True` übergeben. Der Wert dieses Paars wird also zurückgegeben, wenn keine der vorstehenden Bedingungen wahr wird.

3.8.3 Choose

Der `Choose`-Funktion übergeben Sie einen Zahlwert als Index gefolgt von beliebig vielen `Object`-Argumenten:

```
Choose(Index As Double, ParamArray Choice() _
  As Object) As Object
```

`Choose` gibt den Ausdruck zurück, der an der Index'ten Stelle der Parameterliste steht:

```
Console.WriteLine(Choose(1, "a", "b", "c")) ' -> "a"
Console.WriteLine(Choose(3, "a", "b", "c")) ' -> "c"
```

3.9 Prozeduren und Funktionen

Prozeduren und Funktionen sind bei der (veralteten) strukturierten Programmierung die Basis der Wiederverwendung von Programmcode. Immer dann, wenn Sie merken, dass Sie Programmcode in identischer oder ähnlicher Form an mehreren Stellen eines Programms verwenden, können Sie diesen in eine Prozedur oder Funktion packen und bei Bedarf aufrufen. VB.NET unterscheidet Prozeduren und Funktionen vom Begriff her und bei der Deklaration. Im Prinzip sind Funktionen aber nur Prozeduren, die einen Wert zurückgeben, der in Ausdrücken oder bei Zuweisungen weiterverwendet werden kann.

Wenn Sie eine Prozedur oder Funktion in einem Modul unterbringen, gilt diese global und kann überall im Programm einfach über deren Namen aufgerufen werden. Ist die Prozedur bzw. Funktion mit `Public` deklariert, können Sie diese nicht nur im Modul, sondern im gesamten Projekt aufrufen. Dann programmieren Sie klassisch strukturiert.

Implementieren Sie eine Prozedur / Funktion in einer Klasse, wird diese zu einer Methode der Klasse und Sie programmieren (modern) objektorientiert. In Kapitel 4 beschreibe ich, wie das geht.

Die Deklaration einer Prozedur / Funktion unterscheidet sich nicht von der einer Methode.

3.9.1 Die Deklaration von Prozeduren und Funktionen

Prozeduren werden mit dem Schlüsselwort `Sub`, Funktionen mit dem Schlüsselwort `Function` deklariert. Prozeduren deklarieren Sie nach der folgenden Syntax:

```
[Attribute] [{Private | Public}] Sub Name ([Argumentliste])
   ' Definition des Codes
   ' ...
End Sub
```

Funktionen werden etwas anders deklariert:

```
[Attribute] [{Private | Public}] Function Name _
  ([Argumentliste]) [As Datentyp]
   ' Definition des Codes
   ' ...
   ' Rückgabe des Funktionswertes
   Return Ausdruck
End Function
```

Die optionale Argumentliste ist eine kommabegrenzte Liste mit einzelnen Argumenten. Deklarieren Sie die einzelnen Argumente nach der folgenden Syntax:

```
[[{ByVal | ByRef}] Argument [As Datentyp]
```

Als Datentyp können Sie alle die Datentypen verwenden, die auch für Variablen möglich sind. Eine Prozedur, die ein Datum übergeben bekommt und den Wochentag an der Konsole ausgibt, wird z. B. folgendermaßen deklariert:

```
Private Sub OutputWeekdayName(ByVal dt As Date)
   Select Case dt.DayOfWeek()
      Case 0
         Console.WriteLine("Heute ist Sonntag")
```

```
      Case 1
         Console.WriteLine("Heute ist Montag")
      Case 2
         Console.WriteLine("Heute ist Dienstag")
      Case 3
         Console.WriteLine("Heute ist Mittwoch")
      Case 4
         Console.WriteLine("Heute ist Donnerstag")
      Case 5
         Console.WriteLine("Heute ist Freitag")
      Case 6
         Console.WriteLine("Heute ist Samstag")
   End Select
End Sub
```

Beim Aufruf müssen Sie einen passenden Datentyp übergeben:

```
OutputWeekdayName(Date.Now)
```

Der Rückgabewert von Funktionen

In einer Funktion können Sie den Rückgabewert über die `Return`-Anweisung zurückgeben:

```
Return Ausdruck
```

`Return` beendet gleichzeitig auch die Ausführung der Funktion.

Alternativ (und Visual-Basic-6-kompatibel) können Sie den Rückgabewert auch in den Funktionsnamen schreiben. Der Funktionsname steht innerhalb der Funktion für eine lokale Variable. Der Wert dieser Variablen wird am Ende der Funktion vom Compiler automatisch an den Aufrufer zurückgegeben.

Das folgende Beispiel deklariert eine Funktion, die das Gehalt eines Mitarbeiters einer Firma berechnet:

```
Public Function CalcSalary(ByVal hours As Double, _
  ByVal bonus As Double) As Double

   ' Berechnung des Gehalts
   Dim salary As Double
   If (hours <= 40) Then
```

```
      salary = hours * 50
   Else
      salary = (40 * 50) + ((hours - 40) * 75)
   End If
   salary += salary * bonus

   ' Rückgabe des Ergebnisses
   Return salary
End Function
```

Der Aufruf dieser Funktion erfolgt dann wie bei jeder anderen Funktion auch:

```
Dim salary As Double
salary = CalcSalary(125, 0.1)
Console.WriteLine("Der Mitarbeiter verdient " & salary)
```

Private Prozeduren und Funktionen

Mit dem Schlüsselwort `Private` deklarieren Sie eine private Prozedur bzw. Funktion. Wie eine private Variable gilt diese Funktion bzw. Prozedur nur in der Datei, in der sie deklariert ist und kann von anderen Modulen bzw. Klassen des Projekts aus nicht aufgerufen werden. Private Prozeduren können denselben Namen tragen wie andere Prozeduren in anderen Modulen. VB.NET erlaubt die Verwendung eines bereits deklarierten Prozedurnamens, sofern die neue Prozedur privat deklariert ist.

Öffentliche Prozeduren und Funktionen

Mit dem Schlüsselwort `Public` deklarieren Sie eine öffentliche Prozedur bzw. Funktion. Öffentliche Prozeduren bzw. Funktionen können auch von außen aufgerufen werden. Eine `Public`-Funktion bzw. -Prozedur in einem Modul kann von jeder Stelle des Projekts aus über deren Namen aufgerufen werden. Eine `Public`-Funktion oder Prozedur in einer Klasse wird allerdings zu einer neuen Methode der Klasse (siehe Kapitel 4).

3.9.2 Überladene Funktionen

Wenn Sie Funktionen oder Prozeduren überladen, deklarieren Sie mehrere Varianten derselben. Die einzelnen Varianten müssen sich in den Typen und/oder der Reihenfolge der Argumente (in der so ge-

nannten *Signatur*) unterscheiden. Der Rückgabetyp kann unterschiedlich sein, wird aber nicht als Unterscheidungsmerkmal gewertet. Zwei Prozeduren bzw. Funktionen mit derselben Signatur, aber unterschiedlichen Rückgabetypen, sind nicht möglich. Der Compiler muss einfach die Chance haben, die einzelnen Varianten der Methode über die Datentypen der beim Aufruf übergebenen Argumente zu identifizieren.

Das folgende Beispiel verdeutlicht dies anhand der Funktion zur Berechnung eines Mitarbeitergehalts. Die zweite Variante berechnet das Gehalt ohne Bonus:

```
Public Function CalcSalary(ByVal hours As Double) _
  As Double
   ' Berechnung des Gehalts
   Dim salary As Double
   If (hours <= 40) Then
      salary = hours * 50
   Else
      salary = (40 * 50) + ((hours - 40) * 75)
   End If

   ' Rückgabe des Ergebnisses
   Return salary
End Function
```

Bei der Anwendung der Funktion bzw. Prozedur können Sie nun die eine oder die andere Methode zur Berechnung des Gehalts verwenden:

```
' Aufruf der ersten Variante der Funktion
        Dim salary As Double
        salary = CalcSalary(125, 0.1)

' Aufruf der zweiten Variante der Funktion
salary = CalcSalary(125)
```

Je nach dem Typ der übergebenen Argumente verwendet der Compiler die eine oder die andere Variante der Funktion. Überladene Prozeduren und Funktionen werden hauptsächlich in der OOP verwendet. Die Klassen des .NET-Framework enthalten sehr viele Methoden, die in mehreren Varianten implementiert sind, wie Sie ja bereits gesehen haben.

3.9.3 Call by Value und Call by Reference

Wenn ein Argument ohne Übergabeart deklariert ist, verwendet der Compiler die Übergabeart »By Value«. Visual Studio stellt solchen Argumenten allerdings automatisch das Schlüsselwort `ByVal` vor, um den Quelltext deutlicher zu machen.

Sie können die Übergabeart über die Schlüsselwörter `ByVal` und `ByRef` explizit einstellen. Bei der Übergabeart »By Value« erzeugt der Compiler grundsätzlich Programmcode, der bewirkt, dass der Wert dessen, was an diesem Argument an die Prozedur bzw. Funktion übergeben wird, in eine lokale Variable kopiert wird. Diese lokale Variable erzeugt der Compiler natürlich im Hintergrund. Über den Namen des Arguments können Sie in der Prozedur bzw. Funktion auf diese Variable zugreifen. Wenn Sie den Wert des Arguments in der Prozedur verändern, bewirkt dies keine Änderung des Wertes einer eventuell beim Aufruf übergebenen Variablen.

Bei der Übergabeart »By Reference« erzeugt der Compiler allerdings keine lokale Variable, wenn beim Aufruf selbst eine Variable übergeben wird. Stattdessen übergibt der Compiler einfach die Adresse der übergebenen Variable. Wird das Argument in der Prozedur geändert, bewirkt dies immer auch eine Änderung der übergebenen Variablen. Call By Reference benötigen Sie immer dann, wenn eine Funktion mehr als einen Wert zurückgeben soll und die Funktion kein Objekt bzw. keine Struktur als Funktionsrückgabewert besitzen soll. Wenn Sie Objekte an Prozeduren, Funktionen oder Methoden übergeben, werden diese in VB.NET übrigens – anders als in anderen Sprachen – idealerweise By Value übergeben. Der Compiler übergibt dann die Adresse des Objekts (die ja als Wert in der Objektreferenz gespeichert ist). Auf Seite 147 finden Sie nähere Informationen dazu. Das folgende Beispiel für Call By Reference deklariert eine Prozedur, die eine Variable inkrementiert:

```
Private Sub Inc(ByRef number As Integer, ByVal add As Integer)
   number += add
End Sub
```

Beim Aufruf müssen Sie eine Variable vom korrekten Typ übergeben, um den Wert zurückzuerhalten:

```
Dim i As Integer = 1
Inc(i, 2)
```

Da die Übergabe »By Reference« zu schwer lokalisierbaren logischen Fehlern führen kann, nämlich dann, wenn Sie innerhalb einer Prozedur bzw. Funktion ein Argument modifizieren und damit eine an dieser Stelle übergebene Variable nach außen hin verändern, sollten Sie grundsätzlich »By Value« übergeben. Nur, wenn Sie Referenzargumente explizit benötigen, übergeben Sie »By Reference«.

In manchen Fällen kann die Übergabe »By Reference« allerdings Performancevorteile bringen, da Visual Basic beim Aufruf der Funktion bzw. Prozedur lediglich die Adresse der übergebenen Variablen auf dem Stack ablegen muss und nicht den eventuell im Speicher größeren Wert. Dieser Vorteil würde nur beim häufigen Aufrufen einer Prozedur bzw. Funktion erkennbar werden. Testen Sie gegebenenfalls, ob die Übergabe »By Reference« Geschwindigkeitsvorteile bringt. Sie gehen dann allerdings das Risiko ein, schwer lokalisierbare logische Fehler zu erzeugen.

3.9.4 Übergeben von Referenztypen

In VB.NET sollten Sie Referenztypen immer By Value übergeben. Das unterscheidet VB.NET von vielen anderen Sprachen, bei denen diese Typen By Reference übergeben werden müssen. In VB.NET kopiert der Compiler aber den Wert der Referenz in die lokale Argument-Variable. Der Wert der Referenz ist die Adresse des Objekts im Speicher. Sie müssen natürlich beachten, dass Sie das übergebene Objekt nach außen verändern, wenn Sie innerhalb der Methode Eigenschaften des Objekts beschreiben.

Sie können Referenztypen auch By Reference übergeben. Dann verlangsamen Sie die Bearbeitung des Objekts innerhalb der Methode allerdings, weil der Compiler zwei Referenzen auflösen muss: die auf die Objekt-Variable und die auf das Objekt.

Das folgende Beispiel deklariert eine Klasse und übergibt eine Instanz dieser Klasse an eine Prozedur. Die Prozedur verändert die Werte der Instanz:

```
Class Point
    Public x As Integer
    Public y As Integer
End Class

Module Start
   Sub Main()
      ' Referenztyp erzeugen
      Dim p As Point = New Point()
      ' und einer Prozedur übergeben
      SetPoint(p)
      ' Werte ausgeben
      Console.WriteLine(p.x & "," & p.y)
   End Sub
End Module
```


3.9.5 Funktionen, die Referenztypen zurückgeben

Funktionen können neben Werttypen auch Referenztypen zurückgeben. Deklarieren Sie den Rückgabetyp der Funktion mit dem entsprechenden Typ. Das folgende Beispiel erzeugt in einer Funktion eine Auflistung (Collection) aus mehreren Länderbezeichnungen:

```
Private Function CreateCountryCollection() As Collection
   ' Referenztyp erzeugen
   Dim c As Collection = New Collection()
   ' und definieren
   c.Add("Germany")
   c.Add("USA")
   c.Add("Great Britain")
   ' Referenztyp zurückgeben
   Return c
End Function
```

Beim Aufruf weisen Sie die Rückgabe einer Variablem vom entsprechenden Typ zu:

```
Dim c As Collection
c = CreateCountryCollection()
```

3.9.6 Funktionen, die Arrays zurückgeben

Funktionen können auch Arrays zurückgeben. Deklarieren Sie den Rückgabetyp dazu mit einer leeren Klammer. In der Funktion erzeugen Sie ein lokales Array, das Sie füllen und schließlich zurückgeben. Das folgende Beispiel erzeugt in einer Funktion ein Array aus Länder-Bezeichnungen:

```
Private Function CreateCountryArray() As String()
   Dim countries(2) As String
   countries(0) = "Germany"
   countries(1) = "USA"
   countries(2) = "Great Britain"
   ' Rückgabe des Arrays
   Return countries
End Function
```

Beim Aufruf deklarieren Sie ein leeres Array und weisen die Rückgabe der Funktion darauf zu:

```
Dim ca() As String
ca = CreateCountryArray()
```

3.9.7 Die Übergabe von Arrays

Wollen Sie ein Array an eine Prozedur oder Funktion übergeben, deklarieren Sie dieses als Argument mit Klammern, aber ohne Angabe der Elemente und Dimensionen. In der Funktion können Sie das Array mit `For Each` oder mit `For Next` durchgehen (indem Sie den oberen Index über `Length` herausfinden, wenn es sich um ein eindimensionales Array handelt). Das folgende Beispiel deklariert eine Funktion, die alle in einem Array übergebenen Werte addiert und das Ergebnis zurückgibt:

```
Public Function Sum(ByVal numbers() As Integer) As Long
   Dim i As Integer, result As Long
```

```
   result = 0
   For i = 0 To numbers.Length - 1
      result += numbers(i)
   Next
   ' Ergebnis zurückgeben
   Return result
End Function
```

Beim Aufruf der Funktion wird das Array wie eine normale Variable übergeben:

```
Dim numbers() As Integer = {11, 12, 13}
Dim result As Long
result = Sum(numbers)
```

Alternativ können Sie natürlich auch ein neu erzeugtes Array übergeben:

```
result = Sum(New Integer() {10, 11, 12, 13})
```

3.9.8 Variable Argumente mit Parameter-Arrays

Über das `ParamArray`-Schlüsselwort können Sie Argumente deklarieren, an denen der Aufrufer eine beliebige Anzahl Werte übergeben kann. `ParamArray` wird dazu immer mit einer Array-Deklaration verwendet und muss als letztes Argument der Methode eingesetzt werden. Der Compiler erlaubt nur ein `ParamArray`-Argument pro Prozedur bzw. Funktion. Innerhalb der Prozedur/Funktion erhalten Sie ein normales Array. So können Sie z. B. eine Funktion erzeugen, die eine beliebige Anzahl `Integer`-Werte summiert und die Summe zurückgibt:

```
Private Function Sum(ByVal ParamArray numbers() As Integer) As Long
   Dim i As Integer, result As Long
   result = 0
   For i = 0 To numbers.Length - 1
      result += numbers(i)
   Next
   ' Ergebnis zurückgeben
   Return result
End Function
```

Beim Aufruf übergeben Sie die einzelnen Werte kommabegrenzt:

```
result = Sum(10, 11, 12)
```

3.9.9 Optionale Argumente

Neben dem Überladen von Funktionen/Prozeduren steht Ihnen auch die Möglichkeit zur Verfügung, optionale Argumente zu deklarieren, um verschiedene Aufrufvarianten zu realisieren. Viele der speziellen Visual-Basic-Funktionen arbeiten mit solchen Argumenten (die .NET-Methoden allerdings mit überladenen Varianten). Die `MsgBox`-Funktion besitzt beispielsweise nur ein Pflicht-Argument. Alle weiteren Argumente sind optional. Solche Funktionen können Sie natürlich auch schreiben. Optionale Argumente werden mit dem Schlüsselwort `Optional` gekennzeichnet. Einem optionalen Argument müssen Sie immer einen Defaultwert zuweisen. Das folgende Beispiel demonstriert dies anhand einer Prozedur zur Inkrementierung einer `Integer`-Variablen:

```
Private Sub Inc(ByRef number As Integer, _
  Optional ByVal add As Integer = 1)
   number += add
End Sub
```

Der Aufruf erfolgt, wie ich dies bereits im Abschnitt »Argumente übergeben« ab Seite 77 beschrieben habe:

```
i = 1
Inc(i) ' Aufruf ohne das optionale Argument
Inc(i, 10) ' Aufruf mit dem optionalen Argument
```

Die Implementierung optionaler Argumente besitzt den Vorteil, dass Sie nur eine Version der Prozedur/Funktion/Methode implementieren müssen. Überladene Prozeduren, Funktionen oder Methoden besitzen den Vorteil, dass die verschiedenen Varianten vollkommen unterschiedliche Argumente besitzen können. Das Überladen ist zudem moderner als optionale Argumente.

3.10 Bedingte Kompilierung

Sie können über eine Verzweigung ähnlich der `If`-Verzweigung Programmcode bedingungsabhängig kompilieren:

```
#If Bedingung1 Then
   Anweisungen
[#ElseIf Bedingung2 Then
   [Anweisungen]]
[#Else
   [Anweisungen]]
#End If
```

Die Arbeitsweise entspricht der der `If`-Verzweigung, mit dem Unterschied, dass Code in Blöcken, deren Bedingung nicht zutrifft, vom Compiler nicht berücksichtigt wird. Dieser Code kann dann auch für die aktuelle Umgebung nicht benutzbare Anweisungen enthalten.

Der Bedingungsausdruck darf lediglich Literale, Operatoren und Konstanten für bedingte Kompilierung enthalten. Diese speziellen Konstanten können Sie in der Datei über

```
#Const Name = Wert
```

einrichten oder über die Konfigurationseigenschaften des Projekts im Bereich BUILD definieren. Diese Einstellung gilt dann für das gesamte Projekt.

Interessant ist die bedingte Kompilierung zur Ausgabe von Debug-Informationen. Die vordefinierte Konstante `DEBUG` ist dann `True`, wenn Sie die Debug-Konfiguration kompilieren. In der `Release`-Konfiguration ist diese Konstante `False`. Debuginformationen, die Sie in einem bedingten Kompilierblock ausgeben, werden also nicht in das Release übertragen. Daneben eignet sich die bedingte Kompilierung noch hervorragend zur Erzeugung von eingeschränkten Shareware-Versionen einer Anwendung:

```
#Const SHAREWARE = True

Public Sub Speichern()
 ' In der Debugkonfiguration Debuginfos ausgeben
 #If Debug Then
```

```
    ' Debugausgabe
    Console.WriteLine(Date.Now.ToString() & _
       ": Anwender speichert.")
 #End If

 ' Sharewareversion einschränken
 #If SHAREWARE Then
    Console.WriteLine("Sorry, aber Speichern " & _
      "funktioniert in der Shareware-Version nicht")
 #Else
    ' Programmcode zum Speichern
    ' ...
 #End If
End Sub
```

3.11 Reflektion und Attribute

Reflektion

Reflektion erlaubt, dass ein Programm Informationen zu den Typen einer Assemblierung auslesen kann. Das .NET-Framework stellt dazu Klassen zur Verfügung, die im Namensraum `System.Reflection` verwaltet werden. Ein Teilbereich der Reflektion erlaubt sogar, Assemblierungen und deren Typen in der Laufzeit dynamisch zu erzeugen. Entwicklungsumgebungen wie Visual Studio und SharpDevelop nutzen Reflektion, um Informationen zu den Typen zu ermitteln und auszugeben, die Sie im Programm verwenden.

Reflektion ist ein mächtiger Mechanismus, den ich hier nur grundlegend beschreiben kann. Der `Reflection`-Namensraum enthält ca. 40 Klassen und Schnittstellen, mit denen Sie arbeiten können.

Einen Teil der Reflektion nutzen Sie bereits, wenn Sie über die `GetType`-Funktion oder die `GetType`-Methode eines Typs Informationen zum Typ auslesen. Sie können aber auch die Typen anderer Assemblierungen auslesen. Der folgende Quellcode zeigt, wie Sie dies prinzipiell machen:

```
Dim a As System.Reflection.Assembly

' Ermitteln der Assemblierung, in der der Typ
' String gespeichert ist
a = System.Reflection.Assembly.GetAssembly( GetType(String))

' Alternativ können Sie die Assemblierung auch
' aus einer Datei laden
Dim filename As String
filename = "C:\WINNT\Microsoft.NET\" & _
  "Framework\v1.0.3617\System.dll"
a = System.Reflection.Assembly.LoadFrom(filename)

' Alle Typen auslesen
Dim t As Type
For Each t In a.GetTypes()
   ' Informationen zum Typ ausgeben
   Console.Write(t.Name + ": ")
   Console.Write(IIf(t.IsPublic, "Public ", "Private "))
   If (t.IsArray) Then Console.WriteLine("Array")
   If (t.IsClass) Then
      Console.WriteLine("Class")
      ' Methoden auslesen
      Console.WriteLine("   Methoden")
      Dim m As System.Reflection.MethodInfo
      For Each m In t.GetMethods()
         Console.WriteLine("   " + m.Name)
      Next
   End If
   If (t.IsEnum) Then Console.WriteLine("Enum")
   If (t.IsInterface) Then
      Console.WriteLine("Interface")
   End If
   Console.WriteLine()
Next
```

Attribute

Über Attribute können Sie Typen oder Elemente dieser Typen (Eigenschaften, Methoden) beschreiben. Über Reflektion können diese Attribute ausgelesen werden. Einige Attribute werden aber auch vom Compiler verwendet. Das Attribut `WebMethod` definiert z. B. eine Methode als Web-Methode (die in einem Webdienst verwendet wird).

Attribute sind in Attributklassen definiert. Das .NET-Framework definiert einige Attribute vor. Einige der wichtigsten stellt Tabelle 3.16 dar. Wenn Sie eines davon verwenden wollen, setzen Sie den Attributnamen in eckigen Klammern vor die Deklaration und übergeben in Klammern eventuelle Argumente.

Attribut	Beschreibung
`Conditional(` *`conditionString`* `As String)`	markiert ein Element als nur zu kompilieren, wenn das als String angegebene (mit `#Const` definierte) Präprozessor-Symbol den Wert `True` besitzt. Wenn Sie dieses Element im Quellcode verwenden und das Präprozessor-Symbol ist `False` oder nicht vorhanden, erzeugt die CLR keine Ausnahme, sondern führt die entsprechende Anweisung einfach nicht aus. Damit können Sie z. B. ganz einfach Debug-Ausgaben im Programm ermöglichen, deren Anweisungen Sie nicht unbedingt im Release der Anwendung entfernen müssen. Beachten Sie, dass dieses Attribut im `System.Diagnostics`-Namensraum verwaltet wird.
`Obsolete(` *`message`* `As String,` `[,` *`isError`* `As Boolean])`	markiert einen Typ oder ein Element als überholt. Bei der Verwendung generiert der Compiler eine Warnung, wenn Sie *isError* nicht angeben. Geben Sie `True` in *isError* an, erzeugt der Compiler einen Fehler.

Tabelle 3.16: Die wichtigsten Attribute

Das folgende Beispiel verwendet das Attribut `Conditional` für Debugausgaben und `Obsolete`, um eine Klasse als veraltet zu kennzeichnen:

```
' Klasse mit Attribut, das einen Typ als veraltet
' kennzeichnet
<Obsolete("Verwenden Sie die Klasse XPrinter")> _
Public Class Printer
    ' ...
End Class

Public Class XPrinter
    ' ...
End Class

Module Start
    ' Funktion, die über ein Attribut als nur zu
    ' kompilieren definiert wird, wenn die angegebene
    ' Konstante für bedingte Kompilierung True ist
    <System.Diagnostics.Conditional("DEBUG")> _
    Public Sub TraceOutput(ByVal message As String)
        Console.WriteLine(message)
    End Sub

    Sub Main()
        ' Eine Instanz der Klasse Printer erzeugen.
        ' Der Compiler generiert eine Warnung mit
        ' dem im Attribut angegebenen Text
        Dim p As Printer = New Printer()

        ' Methode verwenden, die nur kompiliert wird,
        ' wenn das Präprozessor-Symbol
        ' DEBUG True ist
        TraceOutput("Irgendeine Debug-Info ...")
    End Sub

End Module
```

Das `Obsolete`-Attribut ist in globalen Klassenbibliotheken sinnvoll, die bereits von Anwendungen verwendet werden und die Sie nach-

träglich um neue Klassen, Methoden oder Eigenschaften erweitern. Da alte Anwendungen die alten Elemente verwenden, müssen Sie diese in der Klassenbibliothek belassen. Kennzeichnen Sie diese dann für die Entwicklung neuer Anwendungen als veraltet.

Sie können auch eigene Attributklassen deklarieren um selbst definierte Attribute verwenden zu können. Diese müssen Sie von der Klasse `System.Attribute` ableiten. Die Beschreibung dieser Technik würde den Rahmen dieses Buchs sprengen.