PROGRAMMER'S CHOICE

Christian Wenz
Tobias Hauser
Jürgen Kotz
Karsten Samaschke

ASP.NET 4.0 mit Visual Basic 2010

Leistungsfähige Webapplikationen programmieren

ADDISON-WESLEY

DVD

3

Spracheinführung Visual Basic 10

Zur optimalen Nutzung der Möglichkeiten, die Ihnen mit ASP.NET 4.0 zur Verfügung stehen, sollten Sie nicht nur die reinen ASP.NET-Programmierelemente verwenden. Verknüpfen Sie doch ASP.NET mit einer Programmiersprache. Diese wird Ihnen bei der Entwicklung wertvolle Hilfe leisten können.

Die wesentlichen Standardsprachen sind C# (sprich C sharp), eine Weiterentwicklung von Microsoft, welche die Vorteile von C++ und Visual Basic vereinen soll, und Visual Basic 10. Sie haben aber auch die Möglichkeit, andere Sprachen wie C++ (eine objektorientierte Weiterentwicklung der Programmiersprache C, die ihre Wurzeln schon in den 70er-Jahren hatte) zu verwenden.

Wir werden uns in diesem Buch auf Visual Basic 10 (im weiteren Verlauf werde ich jedoch zur Vereinfachung an vielen Stellen nur noch Visual Basic oder VB verwenden) als Programmiersprache beschränken, da hiermit die wesentlichen Konzepte einfach und umfassend abgedeckt werden können.

3.1 Zur Einführung: Die Geschichte von Visual Basic

Wie bereits in den Einführungssätzen angedeutet, hat Visual Basic bereits eine lange Entwicklungsgeschichte hinter sich.

Visual Basic wurde 1991 vorgestellt. Mit Visual Basic 1 wurden basierend auf den bereits schon länger existierenden Basic-Dialekten neue Features wie beispielsweise die Erzeugung eines Benutzer-Frontends, ohne eigenen Code schreiben zu müssen, basierend auf einem ereignisorientierten Programmiermodell eingeführt. Die erste Version von Visual Basic war hinsichtlich ihrer Verbreitung (und damit auch in kommerzieller Hinsicht) noch nicht besonders erfolgreich.

Im Jahr 1992 wurde mit Version 2.0 Datenbankanbindung via ODBC unterstützt. Nach und nach wurde, auch mit der Einführung von Access und der Verknüpfung der Programmiersprache mit der Datenbank, Visual Basic auch zu einem kommerziellen Erfolg.

1993 kam dann bereits die Version 3 von Visual Basic auf den Markt. Ein weiterer großer Schritt war die Einführung von objektorientierten Elementen im Jahr 1995 mit der Version 4. Schon 1997 wurde die Version 5 herausgebracht, welche die Sprache weiter erweiterte, dieses Mal unter anderem um die Möglichkeit, eigene ActiveX-Komponenten zu schreiben und den Programmcode nicht mehr nur interpretieren zu lassen, sondern vor der Ausführung zu kompilieren (in maschinennahen optimierten Code zu übersetzen) und damit die Performance erheblich zu verbessern.

Mit Visual Basic 6 wurde der letzte Schritt der weiteren Entwicklung von Visual Basic mit jeweils abwärtskompatiblen Programmversionen gemacht. Diese Version kam 1998 auf den Markt.

Mit der Einführung von .NET im Jahr 2002 wurde Visual Basic komplett überarbeitet. Neben einigen Inkonsistenzen, die sich mit der kontinuierlichen Weiterentwicklung eingeschlichen haben (die Behandlung von Variablen und Objektdaten war beispielsweise unterschiedlich), wurde die Sprache modernisiert und grundlegend neu konzipiert. Dies hatte leider zur Folge, dass Visual Basic in den .NET-Versionen nicht abwärtskompatibel zu Visual Basic 6-Code ist.

Auch hier ist die Entwicklung nicht stehen geblieben, sondern schon 2003 wurde ein weiterer Entwicklungsschritt getan (mit Visual Basic .NET), der mit der Version von Visual Basic 2005 bereits kurze Zeit später seinen Nachfolger fand. 2008 erschien dann mit dem .NET Framework 3.5 Visual Basic 9, ehe nun, zwei Jahre später, die aktuelle Version Visual Basic 10 die bislang letzte, aber sicherlich nicht finale Version veröffentlicht wurde.

Wir werden in einem späteren Abschnitt auf die Unterschiede zwischen Visual Basic 6 und Visual Basic 10 eingehen. Sie werden erfahren, was alles zu beachten ist, wenn man Visual Basic 6-Programme in Visual Basic 10 umschreibt.

Wie bereits gesagt, liegt nun der nächste Evolutionsschritt der Sprache in der Version Visual Basic 10 vor. Die Möglichkeiten dieser Sprache werden wir Ihnen nun näher bringen.

3.2 Programmierung mit dem Visual Web Developer

Bevor wir uns mit den eigentlichen Programmierelementen beschäftigen, welche die Programmiersprache Visual Basic ausmachen, wollen wir Ihnen kurz die besonderen Features vorstellen, die Ihnen zur Verfügung stehen, wenn Sie zur Programmerstellung den Visual Web Developer verwenden.

Im vorhergehenden Kapitel haben Sie die Installation aller ASP.NET-Komponenten kennengelernt. Wir haben uns auf die Visual Web Developer 2010 Express Edition konzentriert, da dieser als Freeware zur Verfügung steht. Sie liegt diesem Buch außerdem auf der CD bei.

Mit dem Visual Web Developer erhalten Sie nicht nur alle für die ASP.NET-Entwicklung wesentlichen Werkzeuge an die Hand, nebenbei werden auch für Ihre Visual Basic-Programmierung wesentliche unterstützende Funktionalitäten bereitgestellt.

Sie erhalten Vorlagen, in denen bereits ein Grundgerüst für Ihr Projekt vorgegeben wird, sodass Sie Ihren eigenen Programmcode nur noch ergänzen müssen. Dies wird Ihnen sehr viel Tipparbeit sparen.

Weiterhin wird der Programmcode automatisch formatiert, und Schlüsselwörter werden hervorgehoben. Die notwendigen Einrückungen für Schleifen oder Bedingungen werden automatisch vorgenommen, und im Hintergrund läuft ein Parser, der automatisch die Syntax prüft und auf eventuelle Fehler hinweist, bevor die Seite ausgeführt wird.

Abbildung 3.1: Visual Web Developer – Startseite

Als weiteres Tool wird zur Unterstützung der Entwicklung ein lokaler Webserver mitgeliefert, der für die Darstellung und Ausführung der kompilierten serverseitigen Programme auf Ihrem lokalen Rechner sorgen kann.

Nachfolgend wollen wir Ihnen die ersten Schritte kurz vorstellen, die Sie durchführen müssen um in Ihre ASP.NET-Seiten Visual Basic-Code einzubinden. Weiterhin werden wir auch auf die Trennung von ASP.NET-Code und reinen Visual Basic-Abschnitten eingehen.

In einem späteren Abschnitt, wenn Sie erste Schritte mit Visual Basic gemacht haben, werden wir Ihnen dann weitere Features des Visual Web Developers im Detail vorstellen, die Ihnen die Programmierung erleichtern werden.

3.2.1 Erzeugung einer Website

Zunächst machen wir uns einmal die Möglichkeiten des Visual Web Developers zunutze und erzeugen eine einfache ASP.NET-Website. Dies kann über zwei Wege erfolgen:

Dazu können Sie einfach den Link Neue Website aufrufen. Zum selben Ergebnis kommen Sie wenn Sie unter dem Menüpunkt Datei und den darunter liegenden Menüpunkten NeuE Website... einen Dialog öffnen, den wir hier einmal abgebildet haben.

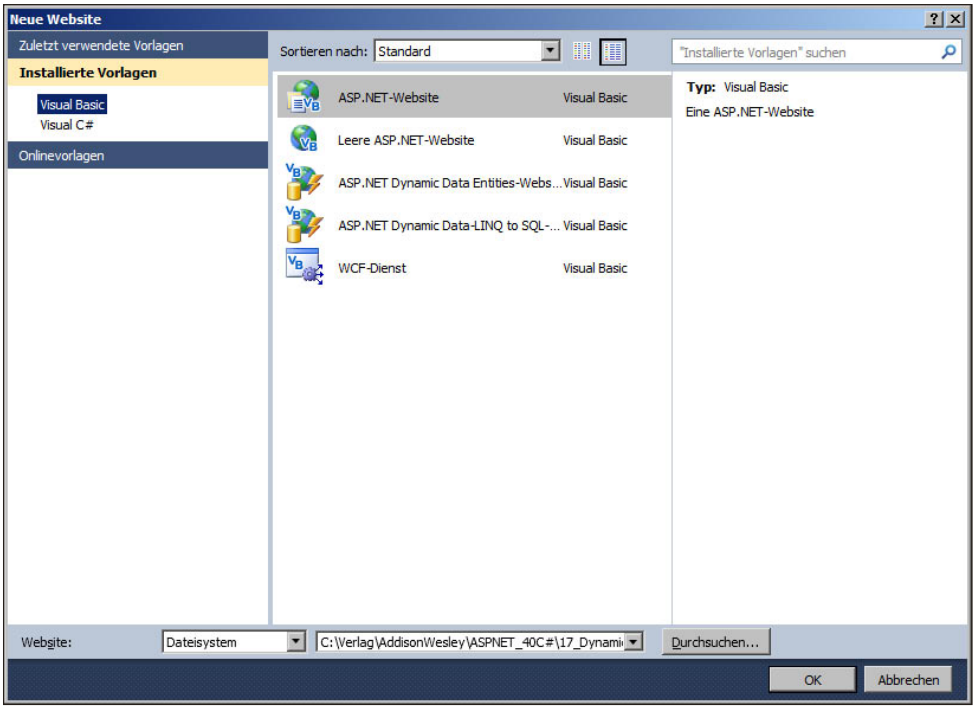

Abbildung 3.2: Erstellen einer neuen Website, Schritt 1

Unter Installierte Vorlagen wird hier die Standardsprache für die Webseite festgelegt. Als Standardwert ist hier bereits der Wert Visual Basic vorgegeben. Sie können den Speicherort der Webseite sowohl über das Dateisystem oder über eine URL (mit dem Feld http) oder die Angabe einer FTP-Adresse festlegen. Wenn Sie eine URL festlegen, können Sie entweder auf einen lokal installierten IIS referenzieren oder aber auch auf einen Remote-Webserver, solange er ebenfalls IIS verwendet. Auch über FTP können Sie eine Remote-Website definieren und an diese Dateien weitergeben.

Welche Art der Speicherung Sie verwenden, hängt vom Gesamtumfeld ab, in dem Sie sich bewegen. Wenn Sie in einer Gruppe entwickeln und auf einem gemeinsamen Server testen wollen, bietet sich eine Remote-Website an. Wenn Sie keinen IIS installieren wollen und nur lokale Tests auf Ihrem Rechner machen wollen sowie keine IIS-Features nutzen, ist die Speicherung im Dateisystem eventuell besser geeignet.

Wir verwenden hier einen Ort im Dateisystem.

Sie haben unter Visual Studio eine Reihe von Möglichkeiten, wählen Sie hier zunächst einmal die ASP.NET-Website aus, und klicken Sie dann den OK-Button.

Ihr Frontend hat nun mehr oder weniger das nachfolgende Aussehen:

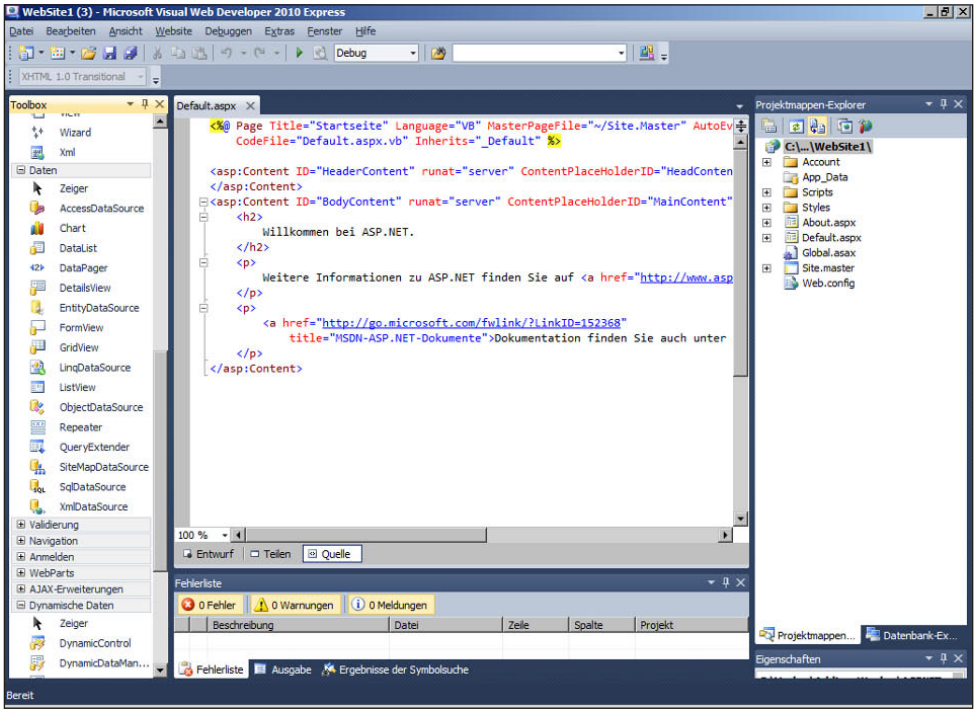

Abbildung 3.3: Frontend nach Erstellung der neuen Webseite (Schritt 2)

Neben der bereits geöffneten Seite *Default.aspx* wurde noch eine Reihe weiterer Dateien und Verzeichnisse erstellt. Die Dateistruktur sieht in etwa wie folgt aus:

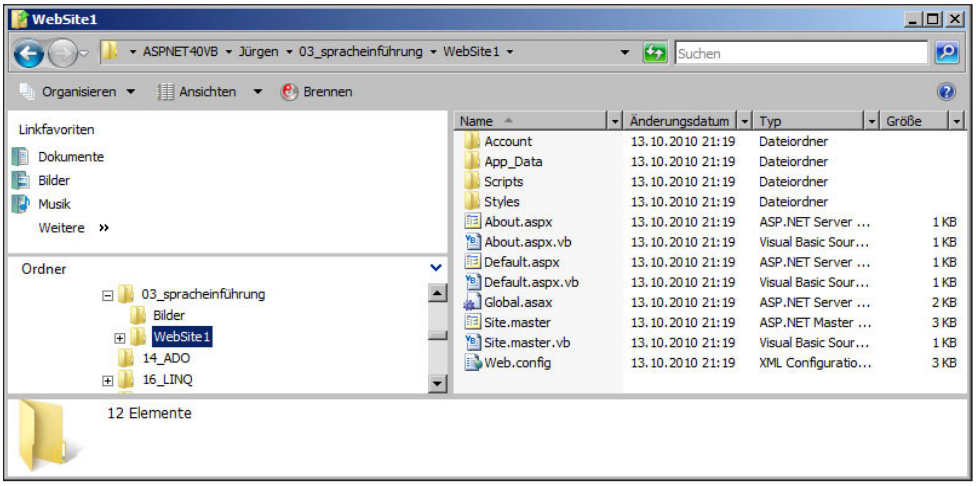

Abbildung 3.4: Angelegte Dateistruktur nach Erstellung einer neuen Website

Uns interessiert nachfolgend die Seite *Default.aspx*. Sie können den Visual Basic-Code auch vollständig von den ASP.NET-Elementen abtrennen und diesen in der korrespondierenden Webseite *Default.aspx.vb* ablegen. Wir beschränken uns in diesem Kapitel auf die Verwendung von Visual Basic-Code innerhalb einer ASP-Seite.

Mit diesen **.aspx*-Seiten können Sie Ihre ersten Schritte mit der Programmiersprache Visual Basic durchführen.

3.2.2 Das obligatorische »Hello World«

Bevor wir uns auf die wesentlichen Programmierelemente von Visual Basic konzentrieren, sollten Sie die Programmierumgebung und die einfache selbstständige Erstellung von Webseiten besser kennenlernen. Als Teil des Visual Web Developers wird auch ein lokaler Webserver mit ausgeliefert, der auf Ihrem Rechner ausgeführt wird. Dadurch können Sie die Ergebnisse Ihrer Arbeit ohne großen Aufwand quasi im Entstehen betrachten und kontrollieren.

Das Programm

Um Ihr erstes Programm in Visual Basic zu schreiben, löschen Sie als Erstes den gesamten vorbereiteten Code und ersetzen ihn durch die nachfolgenden Zeilen:

```
<%@ Page Language="VB" %>

<script runat="server">
  Sub Page_Load()
      Response.Write("Hello World")
    End Sub
</script>
```

Listing 3.1: Das erste Visual Basic und ASP.NET-Programm (Hello_World.aspx)

Speichern Sie diesen Programmtext nun unter dem Namen Hello_World.aspx ab.

Sie finden diesen Quelltext übrigens wie alle anderen Programme auch auf der beiliegenden CD.

Dies stellt den ersten Quelltext dar, den Sie erstellt haben. Nachfolgend wollen wir Ihnen noch kurz erläutern, was dieser Programmcode bedeutet.

Die erste und die letzte Zeile bedeuten, dass der von ihnen eingeschlossene Teil ein Skript darstellt (standardmäßig in der Sprache Visual Basic). Als Programmtext wird eine Prozedur mit dem Namen `Page_Load` erstellt. Diese Prozedur wird mit dem Schlüsselwort `Sub` begonnen und mit den Schlüsselwörtern `End Sub` abgeschlossen.

Der Inhalt dieser Prozedur besteht aus einer einzigen Zeile: Die Ausgabe im Browser wird durch die ASP.NET-Methode `Response.Write()` ausgeführt, die den in Anführungszeichen gesetzten Text erzeugt.

Die Ausgabe

Wie lässt sich nun eine Ausgabe erzeugen bzw. das Ergebnis betrachten? Hierzu haben Sie ein weiteres Tool im Visual Web Developer, das Ihnen die Arbeit erleichtert: Sie finden es unter DEBUGGEN, wie im nachfolgenden Bild beschrieben.

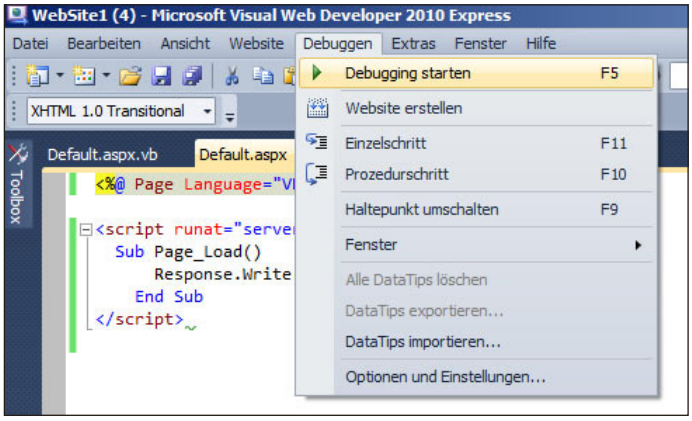

Abbildung 3.5: Start der Ausgabe des »Hello World«-Programms

Wenn Sie dies durchführen, wird ein Browser gestartet, auf dem Sie die Ausgabe betrachten können. Dies wird in Abbildung 3.6 dargestellt.

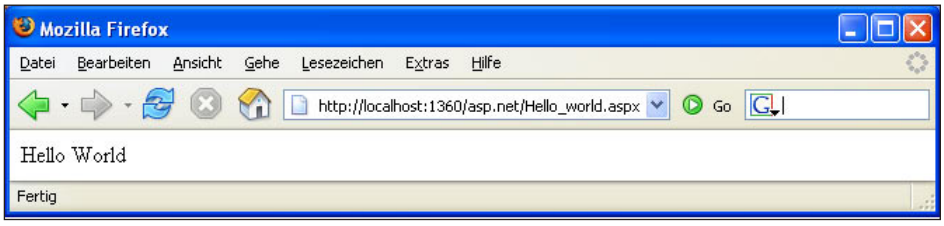

Abbildung 3.6: Ausgabe von Hello World

Dasselbe Ergebnis hätten Sie übrigens auch erhalten, wenn Sie F5 gedrückt hätten.

Üblicherweise wird für die Anzeige der übersetzten Programme der Internet Explorer als Browser gestartet. Die im Browser angezeigte URL können Sie aber auch einfach, wie wir es hier vorgeführt haben, durch einen anderen Browser darstellen lassen. Die erzeugten Programme sind also grundsätzlich unabhängig vom eingesetzten Browser.

Was ist sonst noch passiert?

Mit dem Drücken auf den Debuggen starten F5-Knopf ist nicht nur der Webbrowser mit der anzuzeigenden Seite geladen worden. Bevor dies geschehen ist, ist im Hintergrund der ASP.NET Development Server gestartet worden, den Sie mit dem Visual Web Developer gemeinsam installiert haben und der in den Standardeinstellungen als Webserver zur Verfügung steht. Dass dieser Service gestartet wurde, können Sie in der Taskleiste erkennen.

Sie können sich die Details mit einem Rechtsklick ansehen:

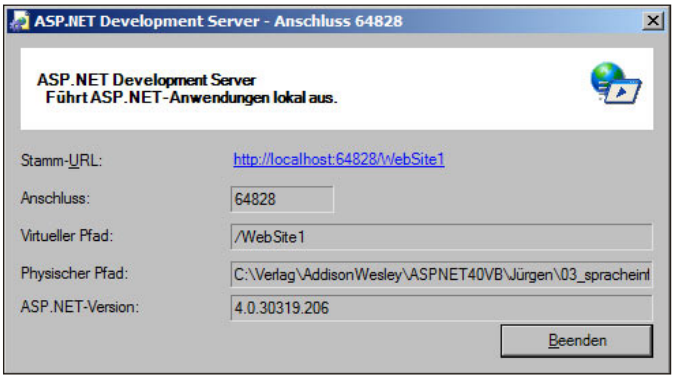

Abbildung 3.7: Anzeige des ASP.NET Development Servers

Dieser Service steht Ihnen allerdings nur lokal auf Ihrem Entwicklungsrechner zur Verfügung.

Weiterhin ist Ihr Programm im Hintergrund übersetzt worden. Es ist ausführbarer Code erzeugt worden, und es hat eine Überprüfung Ihres Codes stattgefunden.

Sie können sich diese Codeprüfung selbst ansehen, wenn Sie unter Debuggen den Menüpunkt Fenster - Ausgabe anwählen. Schalten Sie dafür auf Ausgabe anzeigen von Erstellen. Ihr Ergebnis müsste in etwa so aussehen, wie es das nachfolgende Bild auch zeigt.

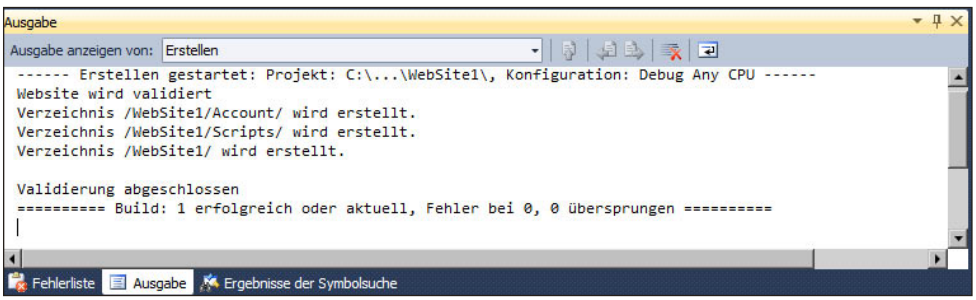

Abbildung 3.8: Ergebnis der Ausgabe des Erstellungsverlaufs von »Hello World.aspx«

Nun kennen Sie das Handwerkszeug, um Ihre ersten ASP.NET-Programme mit Visual Basic selbst schreiben und überprüfen zu können.

3.3 Grundbegriffe von Datentypen bis zu Schleifen

Im nachfolgenden Abschnitt werden wir Ihnen die wesentlichen Schlüsselwörter und Konzepte der Programmierung vorstellen sowie auf die Besonderheiten eingehen, die Visual Basic von anderen Programmiersprachen unterscheidet.

Die grundlegenden Konzepte einer Programmiersprache setzen wir hierbei voraus, sodass, falls Sie ein absoluter Programmieranfänger sind, Ihnen die Darstellung eventuell zu kompakt sein könnte. Sie werden genug ausführliche Spracheinführungen in der einschlägigen Fachliteratur finden können.

3.3.1 Standarddatentypen

Um Daten im Computer vorzuhalten und manipulieren zu können, werden üblicherweise Variablen definiert, die in unterschiedlichen Datentypen als Standard bereitgestellt werden. Sie haben auch die Möglichkeit, eigene Datentypen selbst zu definieren.

Grundsätzlich müssen Sie Variablen vor der ersten Verwendung deklarieren, das heißt, Sie müssen festlegen, welchen Namen die von Ihnen festzulegende Variable hat und welcher Datentyp dieser Variablen zugeordnet ist. Hierfür stehen Ihnen in Visual Basic die Schlüsselwörter `Dim` und `As` zur Verfügung.

Eine Deklaration von Variablen hat also die nachfolgende Struktur:

```
Dim variablenname As Datentyp
```

Nach der Deklaration von Variablennamen und zugehörigem Datentyp können Sie (auch als Teil der Deklaration) per Zuweisung dieser Variable einfach Werte zuordnen.

Daher bietet es sich an, einfach strukturierte Variablen bei ihrer Deklaration gleich mit einem Standardwert oder, falls bereits bekannt, dem benötigten Wert zu versehen.

```
Dim variablenname As Datentyp = wert
```

ACHTUNG

Beachten Sie, dass der Wert einer Variablen ohne eine vorher durchgeführte Zuweisung automatisch vorinitialisiert wird (im Gegensatz zu C#). Das bedeutet, dass Sie auf eine Definition eines initialen Wertes im Notfall verzichten können.

Seit Visual Basic 9 ist auch eine *implizite Typisierung* von Variablen möglich. Das bedeutet, Sie können bei der Definition der Variablen auf den Datentyp verzichten, allerdings funktioniert das nur, wenn der Variablen sofort ein Wert zugewiesen wird und wenn die Variable nur einen lokalen Gültigkeitsbereich besitzt.

```
Dim variablenname = wert
```

Der Compiler erkennt aufgrund des zugewiesenen Wertes automatisch den Datentyp.

ACHTUNG

Verwechseln Sie bitte diese Definition nicht mit dem aus den alten VB-Tagen noch bekannten Datentyp `Variant`. Bei der impliziten Typisierung ist ein nachträgliches Ändern des Datentyps nicht mehr möglich.

Auch IntelliSense unterstützt diese Art der Variablendefinition, wie Sie in Abbildung 3.9 sehen. In dem gezeigten Fall werden sämtliche Methoden und Eigenschaften des Datentyps `String` vorgeschlagen.

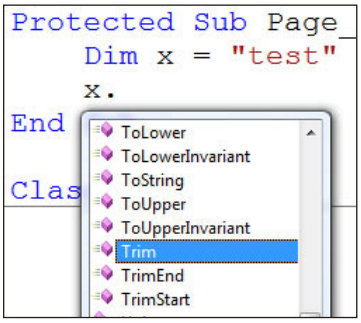

Abbildung 3.9: IntelliSense bei impliziter Typisierung

ACHTUNG

Verwenden Sie jedoch implizite Typisierung nicht grundsätzlich, denn dies kann Ihren Programmcode sehr schnell unleserlich machen. Dieses Sprachmerkmal wurde im Hinblick auf LINQ (mehr dazu in Kapitel 16) eingefügt und sollte auch nur für diese Zwecke in Kombination mit den dafür anderen neuen Sprachmitteln eingesetzt werden.

Die grundlegenden möglichen Datentypen werden wir Ihnen nachfolgend vorstellen.

Unterschiedliche Datentypen für Variablen wurden unter anderem deswegen eingeführt, da es sich als erheblich effizienter herausgestellt hat, unterschiedliche Manipulierungswerkzeuge nur für bestimmte Datentypen zuzulassen; sowohl die Effizienz bei der Speicherung wie auch bei der Verarbeitung wird über diese Standarddatentypen deutlich gesteigert.

Es macht beispielsweise keinen Sinn, eine Multiplikationsoperation für Zeichenketten zu definieren. Andererseits können Integer-Rechenoperationen und Fließkommaberechnungen prozessoroptimiert durchgeführt werden.

Zahlentypen

Für die Speicherung von Zahlen und Operationen mit diesen gibt es eine breite Anzahl von unterschiedlichen Datentypen. Es gibt beispielsweise Datentypen für die Speicherung von Bytes (dies sind ganze Zahlen von 0 bis 255). Für eine solche Zahl ist auch nur ein Byte Speicherplatz zur Speicherung notwendig. Der Wertebereich von Bytes ist somit auch nur auf 256 Werte beschränkt.

Wenn Sie ganze Zahlen mit einem größeren Wertebereich als dem Wertebereich eines Bytes speichern wollen, stehen Ihnen dafür mehrere weitere Integer-Datentypen wie `Short` (2 Byte), `Integer` (4 Byte) oder `Long` (8 Byte) zur Verfügung.

In .NET 4.0 wurde für sehr große Ganzzahlen ein neuer Datentyp `BigInteger` eingeführt, um die bislang bestehenden Limitationen zu überbrücken. Dieser neue Datentyp befindet sich dabei im

Namespace `System.Numerics` der jedoch erst nach dem Hinzufügen eines Verweises auf die gleichnamige Bibliothek zur Verfügung steht.

Wollen Sie Zahlen mit Nachkommastellen (sogenannte Gleitkommazahlen) verwenden, so gibt es auch hierfür Datentypen, die eine Darstellung und optimierte Rechenoperationen mit diesen Zahlentypen ermöglichen.

Aus Speicherplatzgründen stehen Ihnen unterschiedliche Datentypen mit verschiedener Genauigkeit oder auch einem verschieden großen Wertebereich zur Verfügung. Als der häufigste sei hier der Datentyp `Double` genannt.

In der nachfolgenden Tabelle haben wir für Sie alle standardmäßig in Visual Basic vordefinierten Standarddatentypen für Zahlen aufgelistet.

Bezeichnung	Wertebereich von	Wertebereich bis	Größe in Bit
`Byte`	0	255	8
`Short`	–32768	32767	16
`Integer`	–2.147.483.648	2.147.483.647	32
`Long`	–9.223.372.036. 854.775.808	9.223.372.036. 854.775.807	64
`Single`	–3,402823*10^38	3,402823*10^38	32
`Double`	–1,79769313486232 *10^308	1,79769313486232 *10^308	64
`Decimal`	–79.228.162.514. 264.337.593.543. 950.335	79.228.162.514.264. 337.593.543.950.335	96

Tabelle 3.1: Standard-Zahlendatentypen

Der Datentyp `Decimal` stellt bereits eine Besonderheit dar, da Sie festlegen können, wie viele Nachkommastellen Ihr Zahlenwert haben soll. Nach dieser Festlegung wird ohne Rundungen bis auf diese Nachkommastellen gerechnet. Aus diesem Grund bietet sich dieser Datentyp für die Werte an, die zwar Nachkommastellen besitzen, aber eine geringe Anzahl von Nachkommastellen haben (wie beispielsweise bei Geldbeträgen). In alten Visual Basic-Versionen gab es stattdessen den Datentyp `Currency`.

Sie haben die Möglichkeit, weitere Datentypen zu definieren oder bereits im .NET Framework definierte Zahlentypen zu verwenden. Diese stellen allerdings keine Standardzahlentypen dar. So gibt es Integer-Zahlentypen, deren Wertebereiche grundsätzlich positiv sind, sogenannte Unsigned Integers (`UInteger`, `ULong`, `UShort`). Außerdem steht Ihnen noch ein Datentyp der Länge Byte zur Verfügung, der mit einem Vorzeichen versehen ist (`SByte`).

Zeichentypen und Datentypen für Zeichen und Zeichenketten

Für die Verwendung von Zeichen (die auch in gewisser Weise Repräsentanten von ganzen Zahlen sind) und Zeichenketten (also Aneinanderreihungen von Zeichen) stehen ebenfalls Standarddatentypen zur Verfügung.

Der Bezeichner für den Datentyp für Zeichen lautet `Char`, und der Datentyp für Zeichenketten lautet `String`.

Der Wertebereich für `Char` ist dabei 16 Bit und bezieht sich auf die Unicode-Zeichentabelle des Systems (also 2 Byte pro Zeichen).

Der Datentyp `String` stellt einfach eine sequenzielle Liste von miteinander verknüpften einzelnen Zeichen vom Datentyp `Char` dar. Auf dieser Basis gibt es viele Möglichkeiten des Vergleichs und der Manipulation, die wir Ihnen in einem späteren Abschnitt noch näher erklären werden.

Wahrheitswerte

Ein weiteres Format soll die Lesbarkeit von sogenannten logischen Ausdrücken erleichtern. Eigentlich stellt dieses Format ein Byte zur Verfügung, in das nur zwei Werte gespeichert werden können. Einer der Werte steht für »Wahr« und der andere für »Falsch«.

Der Bezeichner für diesen Datentyp ist `Boolean`.

Repräsentation von Datentypen im Speicher

Nachdem wir nun eine Reihe von Datentypen kennengelernt haben, hier ein kleiner Ausflug dahin, wie die Repräsentation der Datentypen im Speicher erfolgen kann.

Es gibt hierbei zwei grundlegende Wege: Der eine Weg ist die direkte Abbildung von Daten im Speicher, wie es bei allen Datentypen, die Zahlen repräsentieren, wie `Integer` und `Single`, aber auch bei `Boolean` der Fall ist. Auch der Datumstyp `Date` und ein einzelnes Zeichen `Char` werden direkt im Speicher abgebildet. Man spricht hierbei von sogenannten Wertetypen.

Die zweite Art, Datentypen zu repräsentieren, ist über Referenzen auf die eigentlichen Daten. Dies ist bei Datentypen, bei denen die tatsächliche Größe variabel oder unbestimmt ist, der Fall. Zu diesen Datentypen gehören `Strings`, aber auch `Arrays` und die benutzerdefinierten Datentypen. Diese werden Referenztypen genannt.

Konvertierung von Datentypen

In der Praxis werden Sie häufig Variablen von einem Datentyp in einen anderen umwandeln müssen. Das .NET Framework stellt hier eine Klasse zur Verfügung, die Ihnen eben diese Konvertierungen ermöglicht. Diese Methoden sind Teil der Klasse `System.Convert`.

Visual Basic stellt außerdem eigene Datentypkonvertierungsfunktionen zur Verfügung. Sie stellen im Wesentlichen Konvertierungsroutinen dar, die durch Methoden aus der `System.Convert`-Basisklasse ebenfalls abgebildet werden können. Die Funktionen in Visual Basic lassen sich einfach herleiten: Sie verwenden den Namen des Zieldatentyps und stellen diesem ein C voran. Beispielsweise wandelt `erg = CBool (wert)` den Wert der Variablen `wert` in den Datentyp `Boolean` um und weist das Ergebnis `erg` zu. Diese Konvertierung ist übrigens eine explizite Konvertierung, das heißt, sie ist sprachspezifisch für Visual Basic.

Nachfolgend wollen wir uns aber genauer mit den Methoden der Klasse `System.Convert` beschäftigen, da diese sprachunabhängig sind und diese Klasse Teil der .NET Basisklassenbibliothek ist.

Sie können mit diesen Methoden die in der nachfolgenden Tabelle aufgelisteten Datentypen ineinander umwandeln.

Falls der umzuwandelnde Wert außerhalb des Wertebereichs des Zieldatentyps liegt, tritt ein Laufzeitfehler (eine Ausnahme vom Typ `OverflowExeption)` auf, der per Ausnahmebehandlung abgefangen werden kann. Falls eine Umwandlung in den Zieldatentyp unmöglich ist, tritt eine Ausnahme vom Typ `InvalidCastException` oder `FormatException` auf. Auf Ausnahmen und ihre Behandlung gehen wir später noch detaillierter ein.

Zieldatentyp	Methode	Kommentar
`Boolean`	`erg= Convert.ToBoolean (wert)`	Erlaubter Wertebereich ist der des Zieldatentyps. Bei Strings sind dies die Werte »True« und »False ». Einige Datentypen wie Variablen vom Typ `DateTime` lassen sich nicht in `Boolean` umwandeln und erzeugen eine `InvalidCastException`.
`Char`	`erg= Convert.ToChar (wert)`	Erlaubter Wertebereich ist der des Zieldatentyps. Bei der Konvertierung von einem String in ein Zeichen wird nur das erste Zeichen des Strings umgewandelt.
`SByte`	`erg= Convert.ToSByte (wert)`	Zahlenwerte können sich zwischen –128 und 127 bewegen. Falls der Wert Nachkommastellen besitzt, werden diese gerundet.
`Byte`	`erg= Convert.ToByte (wert)`	Zahlenwerte können sich zwischen 0 und 255 bewegen. Falls der Wert Nachkommastellen besitzt, werden diese gerundet.
`Short`	`erg= Convert.ToInt16 (wert)`	Erlaubter Wertebereich ist der des Zieldatentyps. Falls der Wert Nachkommastellen besitzt, werden diese gerundet.
`Integer`	`erg= Convert.ToInt32 (wert)`	Erlaubter Wertebereich ist der des Zieldatentyps. Falls der Wert Nachkommastellen besitzt, werden diese gerundet.
`Long`	`erg= Convert.ToInt64 (wert)`	Erlaubter Wertebereich ist der des Zieldatentyps. Falls der Wert Nachkommastellen besitzt, werden diese gerundet.
`UShort`	`erg= Convert.ToUInt16 (wert)`	Erlaubter Wertebereich ist der des Zieldatentyps (nur positive Zahlen). Falls der Wert Nachkommastellen besitzt, werden diese gerundet.
`UInteger`	`erg= Convert.ToUInt32 (wert)`	Erlaubter Wertebereich ist der des Zieldatentyps (nur positive Zahlen). Falls der Wert Nachkommastellen besitzt, werden diese gerundet.
`ULong`	`erg= Convert.ToUInt64 (wert)`	Erlaubter Wertebereich ist der des Zieldatentyps (nur positive Zahlen). Falls der Wert Nachkommastellen besitzt, werden diese gerundet.
`Single`	`erg= Convert.ToSingle (wert)`	Erlaubter Wertebereich ist der des Zieldatentyps.
`Double`	`erg= Convert.ToDouble (wert)`	Erlaubter Wertebereich ist der des Zieldatentyps.

Zieldatentyp	Methode	Kommentar
`Decimal`	`erg= Convert.ToDecimal (wert)`	Erlaubter Wertebereich ist der des Zieldatentyps.
`DateTime`	`erg= Convert.ToDateTime (wert)`	Erlaubter Wertebereich ist der des Zieldatentyps.
`String`	`erg= Convert.ToString (wert)`	Je nach umzuwandelndem Datentyp wird ein String zurückgeliefert, der den booleschen Wert, ein nach lokalen Einstellungen des Servers generiertes Datum, oder einen eine Zahl repräsentierenden String enthält.

Tabelle 3.2: Datentypkonvertierung

Um also einen Wert von `Boolean` in `Integer` umzuwandeln, können Sie also beispielsweise die nachfolgenden Zeilen schreiben (wir geben außerdem die ermittelten Werte noch zusätzlich aus).

```
<%@ Page Language="VB" %>
<script runat="server">
  Sub Page_Load()
    Dim varInt As Integer
    Dim varBool As Boolean
    varBool = False
    varInt = Convert.ToInt32(varBool)
    Response.Write("Wert von VarBool: ")
    Response.Write(varBool)
    Response.Write("<br>Ausgabe von VvrInt: ")
    Response.Write(varInt)
    varBool = True
    varInt = Convert.ToInt32(varBool)
    Response.Write("<br>Wert von VarBool: ")
    Response.Write(varBool)
    Response.Write("<br>Ausgabe von VarInt: ")
    Response.Write(varInt)
  End Sub
</script>
```

Listing 3.2: Typkonvertierung von Datentypen (TypKonv.aspx)

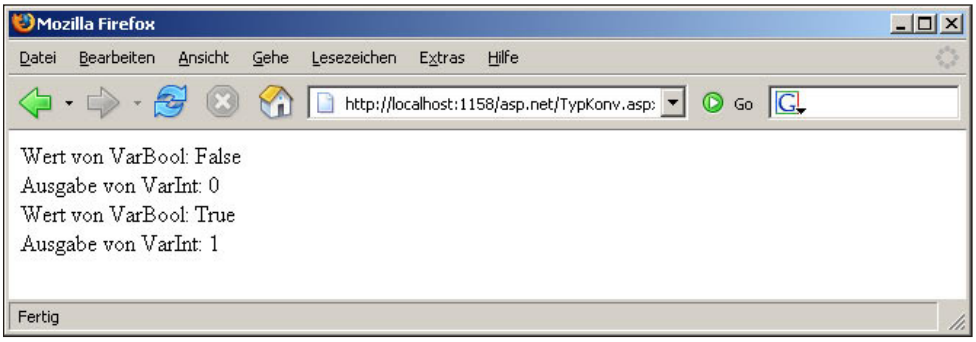

Abbildung 3.10: Datentypkonvertierung

HINWEIS

Bei der Konvertierung von Datentypen sollten Sie beachten, dass es Situationen geben kann, in denen Informationen bei der Konvertierung verloren gehen könnten.

Relativ unproblematische Konvertierungen sind diejenigen, die von einem kleineren zu einem umfangreicheren Datentyp durchgeführt werden. Eine Konvertierung von `Short` zu `Integer` ist ein solches Beispiel.

Wandeln Sie Datentypen von einem Datentyp mit einem großen Wertebereich in einen Datentyp mit einem kleineren Wertebereich um, so ist dies möglich, es besteht aber die Gefahr, dass die Daten nicht verarbeitet werden können.

Natürlich ist es unproblematisch, den Wert einer `Integer`-Variablen in einen `Short`-Wert umzuwandeln, solange der Wert der Variablen sich im erlaubten Wertebereich für `Short` bewegt. Um solche Umwandlungen durchführen zu können, müssen Sie also zum Zeitpunkt der Umwandlung bereits eine Reihe von Informationen über den Inhalt der Variablen haben.

Kommen wir nun zur Problembehandlung bei der Datentypkonvertierung: Die Datenkonvertierung in Visual Basic wird durch drei Ausnahmeklassen unterstützt, die Sie auch im Nachhinein abfangen können.

Mit der Ausnahme `OverflowException` fangen Sie ab, dass Sie zu große Werte in Werte umwandeln, die vom Zieldatentyp nicht dargestellt werden können.

Die Ausnahme `InvalidCastException` fängt ab, wenn Sie bei Umwandlungen von `Single` oder `Double` nach `Decimal` Probleme haben. Die ursprüngliche Zahl lässt sich nicht als Dezimalzahl darstellen, der Ursprungswert ist unendlich, oder er kann durch `Decimal` nicht dargestellt werden.

Ein anderer Fall, in dem die Ausnahme `FormatException` auftritt, ist, wenn eine explizite Konvertierung nicht durchgeführt werden kann, weil für diese Konvertierung kein Datentyp definiert ist. Beispielsweise wenn Sie einen `String` nach `Boolean` umwandeln wollen und der String keinen booleschen Wert beschreibt.

Im nachfolgenden Beispiel werden Ihnen Typkonvertierungen und die Ausnahmebehandlung bei diesen Konvertierungen demonstriert.

Wir haben hier die Form der strukturierten Ausnahmebehandlung mittels der Kontrollstruktur von `Try ... Catch ... End Try` ausgewählt. Diese Kontrollstruktur und weitere werden wir später noch genauer betrachten.

```
<%@ Page Language="VB" %>
<script runat="server">
  Sub Page_Load()
    Dim var1 As Integer
    Dim Var2 As Byte
    Var1 = 1233
    Try
      Var2 = Convert.ToByte(Var1)
    Catch ex As OverflowException
       Response.Write("Der Wert von Var1 ist groesser als Byte")
       Response.Write("<br>daher wird er nicht zugewiesen")
    End Try
    Response.Write("<br>Var1: ")
    Response.Write(Var1)
    Response.Write("<br>Var2: ")
    Response.Write(Var2)
  End Sub
</script>
```

Listing 3.3: Typkonvertierung mit strukturierter Ausnahmebehandlung (TypkonvError.aspx)

Wir machen in diesem Programm eine Zuweisung zu einem Wert, der eigentlich kein Byte-Datentyp sein kann, da der zulässige Wertebereich überschritten wäre. Der Versuch der Zuweisung erzeugt eine Ausnahme, die wir entsprechend abgefangen haben. Statt der Zuweisung geben wir einen Fehlertext aus. Um zu zeigen, dass die Zuweisung tatsächlich nicht stattgefunden hat, geben wir die Werte der beiden Variablen im Nachhinein aus.

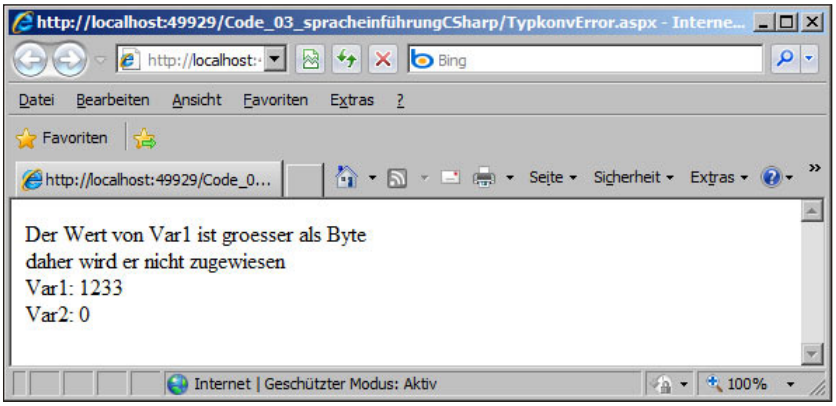

Abbildung 3.11: Datentypkonvertierung mit einfacher Ausnahmebehandlung

Wir werden uns des Themas Fehler- und Ausnahmebehandlung im Absatz 3.9 noch einmal ausführlich annehmen.

3.3.2 Operatoren

Operatoren stellen grundlegende Werkzeuge zur Manipulation von Dateninhalten dar. Diese lassen sich auf die vorgestellten Standarddatentypen anwenden, genauso wie auf die später vorgestellten Datentypen, die im .NET Framework zur Verfügung stehen.

Operatoren zur Zahlenmanipulation

Zahlen lassen sich durch die vier Grundrechenarten sowie die Potenzierung manipulieren. Bei der Division stehen Ihnen neben einem einfachen Divisionsoperator, der den Bruchteil einer Zahl ermittelt, Operatoren zur Verfügung, die Ihnen eine Integer-Division mit Restermittlung ermöglichen. Die nachfolgende Tabelle listet die wesentlichen Operatoren zur Zahlenmanipulation auf.

Für die Verwendung dieser Operatoren gibt es noch eine weitere Abwandlung, die (ähnlich wie es in C üblich ist) das Ergebnis gleich dem links vom Operator stehenden Operanden zuweist.

Für die Ermittlung eines Rests bei der Integer-Division ist kein eigenständiger Zuweisungsoperator vorgesehen.

Diese Zuweisungsoperatoren sind in der zweiten Spalte der Tabelle aufgelistet.

Operator	Zuweisungsoperator	Funktion
+	+=	Addition
-	-=	Subtraktion oder Vorzeichen
*	*=	Multiplikation
/	/=	Division
\	\=	Integer-Division (Division ohne Rest)
Mod		Modulo: Rest einer Division (4 Mod 3 = 1)
^	^=	Potenzierung

Tabelle 3.3: Operatoren zur Manipulation von Zahlen

Vergleichsoperatoren

Vergleichsoperatoren liefern als Ergebnis einen booleschen Wert, der als Kriterium für Verzweigungen oder für das Verlassen von Schleifenabläufen verwendet werden kann.

Nachfolgend sind die booleschen Operatoren, die in Visual Basic Verwendung finden, kurz aufgelistet und ihre Funktion beschrieben.

Operator	Funktion
=	Gleich
<>	Ungleich
<=	Kleiner oder gleich
>=	Größer oder gleich
<	Kleiner als
>	Größer als
IsNot	Prüfung, dass zwei Objektreferenzierungen nicht auf das gleiche Objekt verweisen
Is	Prüfung, ob zwei Objektreferenzierungen auf das gleiche Objekt verweisen
Like	Prüfung, ob eine Zeichenkette einem angegebenen Muster genügt

Tabelle 3.4: Operatoren zum Vergleich

Auch hinter diesen Vergleichsoperatoren verbirgt sich nichts zusätzliches Besonderes, was Sie nicht aus anderen Programmiersprachen bereits kennen würden.

Logische Operatoren

Logische Operatoren stellen auf Binärebene eine wichtige Möglichkeit zur Datenmanipulation dar. Auch für Verzweigungen, die auf der Basis von mehreren verknüpften Bedingungen entstehen, sind logische Operatoren von großer Wichtigkeit.

Sie werden Und- und Oder-Verknüpfungen kennen (eine Und-Verknüpfung ist beispielsweise nur dann Wahr, wenn beide Teilbedingungen wahr sind).

Exotischer sind Exklusives-Oder-Verknüpfungen oder etwa die Prüfung, ob eine Operation den logischen Wert wahr oder falsch ergeben hat. Die `Not`-Verknüpfung besitzt lediglich einen Operand und invertiert seinen Wert.

Grundsätzlich stecken hinter diesen logischen Operatoren aber keine besonderen Geheimnisse, Sie müssen einfach die Ergebnisse der jeweiligen Operation kennen und entsprechend anwenden.

In der nachfolgenden Tabelle haben wir für Sie die wichtigsten logischen Operatoren beschrieben, um welche Art von Operator es sich handelt, und die zugehörigen Wahrheitswerte aufgelistet.

Operator	Beschreibung	Operand 1	Operand 2	Ergebnis
`And`	Und-Verknüpfung	Falsch	Wahr	Falsch
		Falsch	Falsch	Falsch
		Wahr	Falsch	Falsch
		Wahr	Wahr	Wahr
`Or`	Oder-Verknüpfung	Falsch	Falsch	Falsch
		Falsch	Wahr	Wahr
		Wahr	Falsch	Wahr
		Wahr	Wahr	Wahr
`Not`	Nicht	Wahr		Falsch
		Falsch		Wahr
`Xor`	Exklusives Oder	Falsch	Falsch	Falsch
		Falsch	Wahr	Wahr
		Wahr	Falsch	Wahr
		Wahr	Wahr	Falsch
`=`	Äquivalent die Umkehrung von Xor	Falsch	Falsch	Wahr
		Falsch	Wahr	Falsch
		Wahr	Falsch	Falsch
		Wahr	Wahr	Wahr
`AndAlso`	Prüfung: Falls der erste Wert nicht falsch ist wie `And`	Falsch	*Nicht bewertet*	Falsch
		Wahr	Wahr	Wahr
		Wahr	Falsch	Falsch
`OrElse`	Prüfung: Falls der erste Wert nicht wahr ist wie `Or`	Wahr	*Nicht bewertet*	Wahr
		Falsch	Wahr	Wahr
		Falsch	Falsch	Falsch

Tabelle 3.5: Logische Operatoren und die ihnen zugeordneten Wahrheitswerte

Für die Operatoren `And` und `Or` gibt es noch jeweils ein weiteres Pendant, in dem der zweite Ausdruck nicht ausgewertet wird, wenn auf Basis der Auswertung des ersten Ausdrucks schon klar ist, welches Ergebnis erreicht wird. Diese haben wir in der Tabelle mit aufgeführt. Beim Operator `AndAlso` wird, falls der erste Ausdruck `false` ist, nicht weiter ausgewertet, beim Operator `OrElse` ist dies der Fall, falls der erste Ausdruck bereits `true` ist.

```
<%@ Page Language="VB" %>

<script runat="server">
  Sub Page_Load()
     Dim Var1 As String = "a"
    Dim Var3 As String = "1"
     Response.Write("Logik mit und ohne AndAlso:<br>")
     Response.Write("Var1 = a und Var3 = 1 ergibt:<br>")
     If (Isnumeric(Var1) AndAlso Var1 <> 1) Then
       Response.Write("<br>AndAlso:True: Var1 ist Zahl AND <>1 ")
     Else
       Response.Write("<br>AndAlso:False: Var1 ist keine Zahl AND <>1 ")
     End If
     Try
       If (Isnumeric(Var1) And Var1 <> 1) Then
         Response.Write("<br>And:True: Var1 ist Zahl AND <>1")
       Else
         Response.Write("<br>And:False: Var1 ist keine Zahl AND <>1 ")
       End If
     Catch e As Exception
        Response.Write("<br>Var1 ist keine Zahl und nicht umwandelbar ")
        Response.Write("daher Cast-Fehler bei AND<br>")
     End Try
     If (Isnumeric(Var1) AndAlso Var1 = 1) Then
       Response.Write("<br>AndAlso:True: Var1 ist Zahl AND = 1")
     Else
       Response.Write("<br>AndAlso:False: Var1 ist keine Zahl AND = 1 ")
     End If
     Try
       If (Isnumeric(Var1) And Var1 = 1) Then
         Response.Write("<br>And:True: Var1 ist Zahl AND = 1")
       Else
         Response.Write("<br>And:False: Var1 ist keine Zahl AND = 1")
       End If
     Catch e As Exception
        Response.Write("<br>Var1 ist keine Zahl und nicht umwandelbar ")
        Response.Write("daher Cast-Fehler bei AND<br>")
     End Try
     If (Isnumeric(Var3) AndAlso Var3 <> 1) Then
       Response.Write("<br>AndAlso:True: Var3 ist Zahl AND <> 1")
     Else
       Response.Write("<br>AndAlso:False: Var3 ist keine Zahl AND <>1 ")
     End If
     Try
       If (Isnumeric(Var3) And Var3 <> 1) Then
         Response.Write("<br>And:True: Var3 ist Zahl AND <> 1")
       Else
         Response.Write("<br>And:False: Var3 ist keine Zahl AND <> 1")
       End If
     Catch e As Exception
        Response.Write("<br>Var3 ist keine Zahl und nicht umwandelbar")
```

```
        Response.Write("<br>daher Cast-Fehler bei AND")
    End Try
    If (Isnumeric(Var3) AndAlso Var3 = 1) Then
        Response.Write("<br>AndAlso:True: Var3 ist Zahl AND = 1")
    Else
        Response.Write("<br>AndAlso:False: Var3 ist keine Zahl AND = 1")
    End If
    Try
      If (Isnumeric(Var3) And Var3 = 1) Then
        Response.Write("<br>And:True: Var3 ist Zahl AND = 1")
      Else
        Response.Write("<br>And:False: Var3 ist keine Zahl AND = 1")
      End If
    Catch e As Exception
        Response.Write("<br>Var3 ist keine Zahl und nicht umwandelbar")
        Response.Write("<br>daher Cast-Fehler bei AND")
    End Try
  End Sub
</script>
```

Listing 3.4: Logik mit And und AndAlso (Logik.aspx)

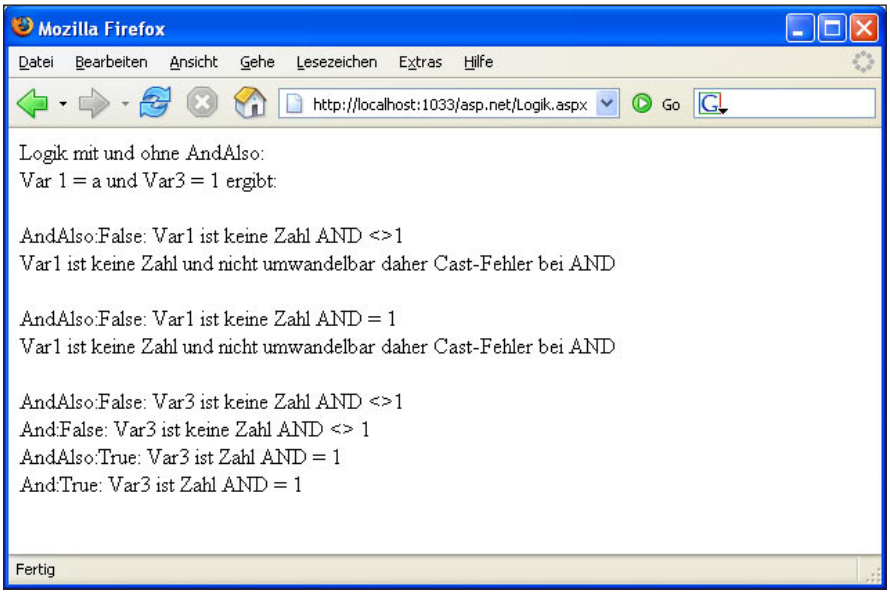

Abbildung 3.12: Logik mit And und AndAlso

Ternärer Operator

Mit dem `IIf`-Operator steht in Visual Basic ein ternärer Operator zur Verfügung. Bei einer `IIf`-Abfrage wird einer von zwei möglichen Werten zurückgegeben. Im nachfolgenden Beispiel wird überprüft, ob eine Variable `i` einen Wert größer als 5 besitzt. Ist das der Fall, wird das zweite Argument zurückgegeben, ansonsten das dritte Argument.

```
Dim s As String = IIf(i>5, "groß", "klein").ToString()
```

NEU

Die Funktionalität des `IIf`-Operators ist mit der Einführung von Visual Basic 9 verändert worden. Erst ab dieser Version handelt es sich tatsächlich um einen ternären Operator. Im Gegensatz zu früheren Versionen werden nicht mehr beide Argumente ausgewertet, sondern nur noch das passende.

Die Anweisung

```
Dim i as Integer = Convert.ToInt32(IIf (x Is Nothing, 0, x.Test))
```

hätte, falls x tatsächlich `Nothing` wäre, in den Vorgängerversionen einen Laufzeitfehler vom Typ `NullReferenceException` geworfen, weil die Eigenschaft Test nicht ausgewertet werden konnte. Jetzt wird einfach 0 zurückgegeben.

Sonstige Operatoren

Neben den oben beschriebenen mathematischen und logischen Operatoren gibt es noch eine Reihe weiterer Operatoren, die eine gewisse Bedeutung haben. Diese sind nachfolgend im Einzelnen aufgelistet und kurz beschrieben.

Zunächst haben wir da zwei Operatoren, die nah an Assembler angelehnt sind, die sogenannten Bitshift-Operatoren, mit denen in einer Variablen die Inhalte bitweise nach links oder rechts verschoben werden. Bei einem Integerwert entspricht die Verschiebung um eine Stelle einer Multiplikation mit zwei (bei der Verschiebung nach links) oder einer Division durch zwei (bei der Verschiebung nach rechts), unter der Voraussetzung, dass das Ergebnis noch im gültigen Bereich liegt. Ansonsten ist das Ergebnis negativ.

In der nachfolgenden Grafik haben wir anhand eines Byte-Wertes die Wirkungsweise der Bitshift-Verschiebung illustriert.

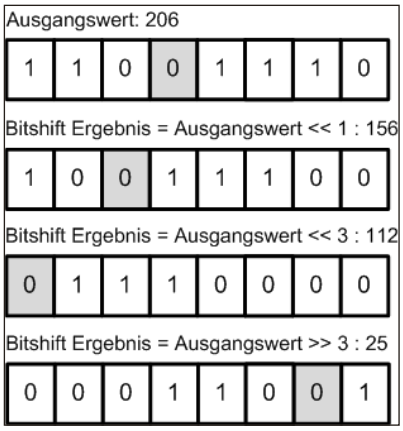

Abbildung 3.13: Illustration von Bitshift-Operationen

Diese Operatoren sind nur auf Integer-Variablen anwendbar, der zweite Operand legt fest, um wie viele Bits die Verschiebung zu erfolgen hat.

Einen weiteren einfachen, aber sehr nützlichen Operator stellt derjenige dar, mit dem Zeichenketten miteinander verknüpft werden. Dieser wird durch `&` repräsentiert.

Operator	Funktion	Beispiel
<<	Bitshift nach links	`Erg = Wert << 3`
>>	Bitshift nach rechts	`Erg = Wert >> 2`
&	Verknüpfung von Zeichenketten	`A = B & C`

Tabelle 3.6: Sonstige Operatoren

Wir sollten zum Abschluss noch ein paar Worte über die Hierarchien von Operationen und Klammerung verlieren.

Grundsätzlich gilt in der Hierarchie der Ausführung, dass eine Operation höherer Ordnung vor einer Operation niederer Ordnung ausgeführt wird. Punktrechnung geht also vor Strichrechnung. Durch die Verwendung von Klammern, die Sie beliebig schachteln können, können Sie diese Hierarchie, wie in der Schulmathematik gelernt, aufheben.

Die Verwendung von Klammern empfiehlt sich auch, wenn Sie Rechenoperatoren mit logischen Operatoren mischen wollen, da es die Lesbarkeit einer Formel erheblich erhöhen kann.

3.3.3 Strukturierte Datentypen

Wie mit dem Standarddatentyp `String` schon angedeutet, werden Sie häufig grundlegende Datentypen miteinander verknüpfen wollen, um die benötigten Daten in einfachen Strukturen vorzuhalten. Somit erschaffen Sie neuartige Klassen von Datentypen.

Nachfolgend werden wir Ihnen einige grundlegende strukturierte Datentypen vorstellen, die auch in den alten Versionen von Visual Basic vergleichbare Verwendung gefunden haben.

Sie basieren auf den Möglichkeiten des .NET Frameworks und werden in gleichartiger Form damit auch von anderen Sprachen, die sich dieses Frameworks bedienen, verwendet. Durch die Definition dieser Klassen werden gleich die Möglichkeiten zur Manipulation der Datentypen mitgeliefert.

Arrays

Arrays sind eine Möglichkeit, beispielsweise die Standarddatentypen, die wir bereits kennengelernt haben, zu Wertegruppen zusammenzufassen. Hierbei werden die Daten in einer Art Liste zusammengefasst und können quasi über einen Index ausgewählt, manipuliert und angezeigt werden.

Die Deklaration und die Initialisierung eines Arrays erfolgen, wie Sie es auch schon bei einfachen Variablentypen gelernt haben, durch das vor dem Arraynamen gestellte Schlüsselwort `Dim` (oder zur Reinitialisierung `ReDim`), die Zuordnung des Grunddatentyps über das Schlüsselwort `As` und die zusätzliche Angabe der Anzahl der Felder, die vom Standarddatentyp bereitgestellt werden sollen:

```
Dim Arg1(6) As String
```

Wir haben in diesem Beispiel also ein Array mit sieben Elementen vom Datentyp `String` deklariert. Das erste Element ist bereits `Arg1(0)`. Einzelne Elemente innerhalb des Arrays werden nun direkt zugewiesen:

```
Arg1(2) = "Dritter Wert"
```

Nun können Sie entweder auf einzelne Elemente des Arrays zugreifen, diese Elemente verändern, austauschen oder löschen oder das Array um zusätzliche Elemente ergänzen.

Wie bei normalen Variablendefinitionen können Sie auch bei Arrays bereits bei der Definition die Variable initialisieren.

```
Dim Arg1() As String = {"A", "B", "C", "D"}
```

Auf eine Größenangabe des Arrays wird hier verzichtet, da die Größe aufgrund der Anzahl der Initialwerte ermittelt wird. Sie dürfen an dieser Stelle die Arraygröße gar nicht angeben.

Eine weitere Besonderheit bei Arrays stellt die Möglichkeit dar, mehrdimensionale Datenfelder zu erzeugen. Die Deklaration eines solchen Datentyps geschieht durch eine kommagetrennte Angabe der Größe der jeweiligen Dimension. Hier ein kleines Beispiel zur Veranschaulichung:

```
Dim Arg1(6, 2, 6) As String
```

Mit dieser Deklaration haben wir ein dreidimensionales Objekt erzeugt, in dem Zeichenketten abgelegt sind. Die maximale »Breite«, »Höhe« und die »Tiefe« sind angegeben. Sie können in diesem Datenarray 147 Elemente (7*3*7 Elemente) ablegen.

Dieser Datentyp stellt bereits einen Typ dar, der als Teil des .NET Frameworks bereitgestellt wird.

Es gibt einige weitere Sprachelemente, welche die Handhabung eines Arrays erleichtern. Mittels der Methode `LBound()` können Sie für die jeweils angegebene Dimension die mögliche Untergrenze ermitteln. Mittels `UBound()` ermitteln Sie die Obergrenze der jeweiligen Dimension.

Falls Sie den von einem Array belegten Speicherplatz wieder freigeben wollen, steht Ihnen dafür die Methode `Erase()` zur Verfügung.

Zur Prüfung, ob eine Variable ein Array ist, können Sie `IsArray()` verwenden.

Nachfolgend noch einmal tabellarisch die einzelnen Funktionen, die Sie für die einfachere Verwendung von Arrays zur Verfügung haben.

Funktion	Beschreibung	Beispiel
`Dim Array(`Dimension1,...`) As` Datentyp	Initialisieren eines Arrays	`Dim Feld (1,3,10) As String`
`LBound(`Arrayname, Dimension`)`	Ermitteln der Untergrenze einer Arraydimension	`Wert = LBound(Feld,3)`
`UBound(`Arrayname, Dimension`)`	Ermitteln der Obergrenze einer Arraydimension	`Wert = UBound(Feld,3)`
`Erase` Arrayname	Freigabe des Arrays	`Erase Feld`
`IsArray(`Arrayname`)`	Prüfung, ob Variable ein Array ist	`Erg = IsArray(Feld)`

Tabelle 3.7: Funktionen zur Behandlung von Arrays

Datumstypen

Datum und Uhrzeit stellen einen strukturierten Datentyp dar. Die Elemente dieses strukturierten Datentyps lassen sich auf unterschiedliche Weise ausgeben und anzeigen. Zum einen kann die Anzeige (ähnlich wie auch schon bei Gleitkommazahlen, wo in Deutschland das Dezimalzeichen ein Komma ist, in angelsächsischen Ländern aber ein Punkt) länderspezifisch sein, zum anderen können Sie Daten auf unterschiedliche Weise darstellen (Sie können Wochenendtage mit angeben oder weglassen, Monatsbezeichnungen in verschiedener Weise ausgeben oder Ähnliches).

Eine Datumsanzeige kann in langer Form mit ausgeschriebenem Monatsnamen und Wochentag erfolgen oder auch in kurzer Form, als einfache Aneinanderreihung von Zahlen. Sie können die Informationen genauer halten oder auch Details weglassen (wie die Uhrzeit oder die Sekunden einer Uhrzeit).

In der nachfolgenden Tabelle ist einmal beispielhaft eine Reihe unterschiedlicher möglicher Datumsformate ein und desselben Datums aufgelistet:

Datum	Uhrzeit
25.2.2006	1:23
25. Februar 2006	1:23:45
25/02/06	01:23
25. Feb. 06	1 h 23 min 45 sec

Tabelle 3.8: Verschiedene Darstellungen von Datums- und Uhrzeitformaten

In Visual Basic wird als Standarddatentyp für ein Datum und eine Uhrzeit der Datentyp `Date` zur Verfügung gestellt. Wenn Sie einer Variablen vom Datentyp `Date` einen Wert direkt zuweisen wollen, so ist der entsprechende Wert zwischen Doppelkreuze zu schreiben:

```
Dim Geburtsdatum as Date = #12/24/1967#
```

oder alternativ können Sie eine neue Instanz einer `Date`-Struktur erstellen:

```
Dim Geburtsdatum as Date = New Date(1967,12,24)
```

In der Darstellung und der Auswertung des Datentyps `Date` gibt es eine Reihe von Funktionen, mit denen Sie beispielsweise das aktuelle Systemdatum und die aktuelle Systemzeit ermitteln können. Sie können auch einzelne Elemente dieses Datentyps mit verschiedenen Funktionen manipulieren oder auswerten.

Das .NET Framework stellt dabei die `DateTime`-Struktur zur Verfügung, `Date` ist der Alias in Visual Basic für diese Struktur. Mit dieser Struktur erhalten Sie eine ganze Reihe von Methoden, um Daten oder Teile von Daten zu setzen, auszuwerten und in vielen möglichen unterschiedlichen Formen auch anzuzeigen.

In der nachfolgenden Tabelle sind einige wesentliche Eigenschaften zur Ermittlung von Elementen der `DateTime`-Struktur aufgelistet.

Eigenschaft	Beschreibung
`DateTime.Now`	Liefert die aktuelle Systemzeit und das Systemdatum.
`DateTime.Today`	Liefert das Systemdatum.
`DateTime.TimeOfDay`	Liefert die aktuelle Systemzeit.
`DateTime.Ticks`	Liefert die Anzahl an 100 Nanosekunden-Intervallen, die seit dem 1.1.0001 vergangen sind. Zu dieser Eigenschaft gibt es keine korrespondierende Visual Basic-Funktion.
`DateTime.Second`	Liefert die Sekunde eines vorgegebenen Datums als Zahl.
`DateTime.Minute`	Liefert die Minute eines vorgegebenen Datums als Zahl.
`DateTime.Hour`	Liefert die Stunde eines vorgegebenen Datums als Zahl.
`DateTime.Weekday`	Liefert den Wochentag eines vorgegebenen Datums als Zahl.
`DateTime.Month`	Liefert den Monat eines vorgegebenen Datums als Zahl.
`DateTime.Year`	Liefert das Jahr eines vorgegebenen Datums als Zahl.
`DateTime.DayOfWeek`	Liefert den Namen des Wochentags eines Datums.
`MonthName`	Liefert den Namen des Monats einer Zahl (in Deutschland ergibt `MonthName(1)` z.B. Januar).

Tabelle 3.9: Ermitteln von Datum, Uhrzeit und Elementen davon

Sie haben neben den Möglichkeiten, verschiedene Eigenschaften auszulesen, auch die Möglichkeit, an `DateTime`-Strukturen mit verschiedenen Methoden Manipulationen vorzunehmen. In der nachfolgenden Tabelle haben wir einige davon aufgelistet und erläutert.

Methode	Beschreibung
`DateTime.Parse()`	Mittels dieser Methode können Sie eine Zeichenkette in eine Struktur vom Typ `DateTime` umwandeln.

`DateTime.Add()`
`DateTime.AddTicks()`
`DateTime.AddSeconds()`
`DateTime.AddMinutes()`
`DateTime.AddHours()`
`DateTime.AddDays()`
`DateTime.AddMonths()`
`DateTime.AddYears()`
`DateTime.Subtract()`
	Diese Gruppe von Methoden können Sie dazu verwenden, um Daten innerhalb der `DateTime`-Struktur zu verändern, indem Sie Zeiteinheiten hinzufügen oder (durch Hinzufügen negativer Werte oder Verwendung von `DateTime.Subtract`) vermindern. Falls Sie außerhalb des zugelassenen Wertebereiches der `DateTime` gelangen sollten, wird eine `ArgumentOutOfRangeException` erzeugt.

`DateTime.ToString()`
`DateTime.ToLongDateString()`
`DateTime.ToLongTimeString()`
`DateTime.ToShortDateString()`
`DateTime.ToShortTimeString()`
	Mittels dieser Methode wandeln Sie den Wert einer `DateTime` Struktur in einen String um. Der String wird je nach gewählter Methode formatiert.

`DateTime.Compare()`	Mit dieser Methode vergleichen Sie zwei `DateTime`-Strukturen miteinander. Das Ergebnis ist ein Integerwert, der wie folgt zu interpretieren ist: Wert < 0: der erste `DateTime`-Wert ist kleiner als der zweite `DateTime`-Wert. Wert = 0: Beide `DateTime`-Werte sind gleich. Wert > 0: der erste `DateTime`-Wert ist größer als der zweite `DateTime`-Wert.
`DateTime.IsLeapYear()`	Ermittlung, ob das Jahr der DateTime-Klasse als Schaltjahr definiert ist.

Tabelle 3.10: Manipulation und Vergleich von Datum, Uhrzeit und Elementen davon

Weiterhin sind mathematische Operatoren für Addition und Subtraktion für die `DateTime`-Struktur definiert. Ebenso können Sie auch die Vergleichsoperatoren verwenden.

Wenn Sie Additionen und Subtraktionen durchführen, ist das Ergebnis ein Wert in der Struktur `TimeSpan`. Diese Struktur ist grundsätzlich ähnlich aufgebaut wie die Struktur `DateTime`. Ein wesentlicher Unterschied liegt darin, dass die Struktur `DateTime` einen Zeitpunkt repräsentiert, die Struktur `TimeSpan` aber einen Zeitraum.

```
<%@ Page Language="VB" %>
<script runat="server">
  Sub Page_Load()
    Dim Arg1 As New System.DateTime()
    Dim Arg2 As New System.DateTime()
    Dim Arg3 As New System.TimeSpan()
    Arg1 = System.DateTime.Parse("24.01.1969 05:32:12")
    Arg2 = System.DateTime.Now
    Response.Write("Einige Datumsausgaben:")
    Response.Write("<br>Datum: ")
    Response.Write(Arg1.ToLongDateString())
    Response.Write("<br>Jetzt: ")
    Response.Write(Arg2.ToLongDateString())
    Response.Write("<br>Stunde: ")
    Response.Write(Arg1.Hour.ToString())
    Response.Write("<br>Minute: ")
    Response.Write(Arg1.Minute.ToString())
    Response.Write("<br>Sekunde: ")
    Response.Write(Arg1.Second.ToString())
    Response.Write("<br>Jahr: ")
    Response.Write(Arg1.Year.ToString())
    Response.Write("<br>Monat: ")
    Response.Write(Arg1.Month.ToString())
    Response.Write("<br>Tag: ")
    Response.Write(Arg1.Day.ToString())
    Response.Write("<br>Datum: ")
    Response.Write(Arg1.ToString())
    Response.Write("<br>Zeitdifferenz zwischen Jetzt und Datum (als TimeSpan):")
    Arg3 = Arg2 - Arg1
    Response.Write(Arg3.ToString)
  End Sub
</script>
```

Listing 3.5: Anzeige von verschiedenen Datumsformaten (DateTime.aspx)

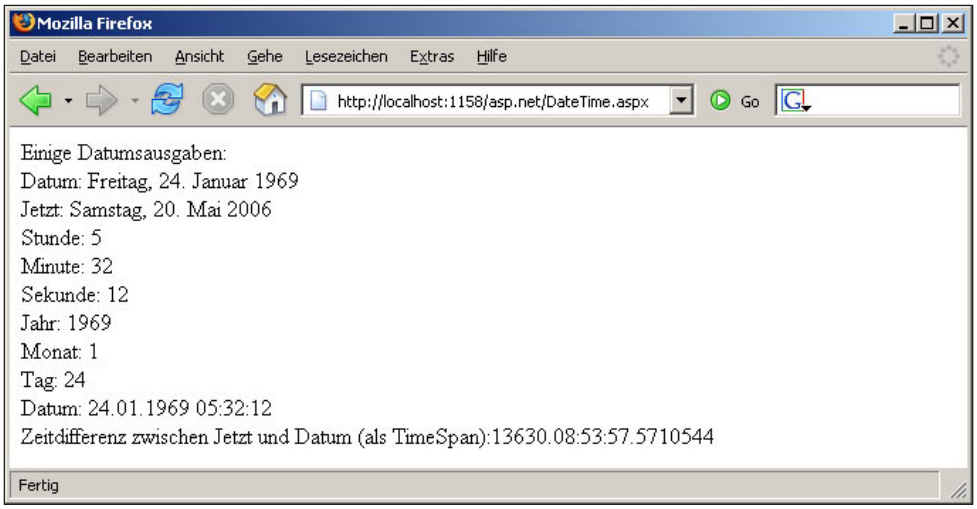

Abbildung 3.14: Ausgabe verschiedener Datumsformate und Elemente

3.3.4 Wertetyp- und Referenztypsemantik

Wie bereits erwähnt, unterscheiden sich Variablen in zwei Gruppen, den Wertetypen und den Referenztypen. Bei Wertetypen handelt es sich dabei um die folgenden Datentypen:

- Byte
- Short
- Integer
- Long
- BigInteger
- UShort
- UInteger
- ULong
- Single
- Double
- Decimal
- Date
- Char
- Boolean
- und sonstige Strukturen

Bei den Wertetypen wird der Wert der Variablen direkt an dieser Speicherstelle im Stack gespeichert.

Zu den Referenztypen gehören folgende Typen:

» String

» und sonstige andere als Klassen implementierte komplexe Objekte

Bei den Referenztypen wird an der Speicherstelle im Stack auf eine Adresse im Heap referenziert. Referenztypen werden somit wie Zeiger auf einen anderen Speicherbereich implementiert.

Doch Wertetypen und Referenztypen unterscheiden sich nicht nur in der Art der Speicherung, sondern auch in der Art und Weise, wie Daten zugewiesen werden.

Wertetypsemantik

Bei der Zuweisung eines Wertetyps wird eine Kopie der entsprechenden Variablen gemacht. Wird die zugewiesene Variable manipuliert, hat das keine Auswirkungen auf die Originalvariable.

```
<%@ Page Language="VB" %>
<script runat="server">
  Sub Page_Load()
    Dim Arg1 As Integer = 7
    Dim Arg2 As Integer = Arg1
    Arg2 = 9
    Response.Write(Arg1.ToString())
    Response.Write(Arg2.ToString())
  End Sub
</script>
```

Listing 3.6: Beispiel für Wertetypsemantik

Im Beispiel in Listing 3.6 wird eine Variable `Arg1` vom Typ `Integer` definiert und dabei der Wert 7 zugewiesen. Im nächsten Schritt wird eine zweite Integervariable `Arg2` definiert, und ihr wird der Wert von `Arg1` zugewiesen. Der Wert von `Arg1` wird aufgrund der Wertetypsemantik in die Speicherzelle `Arg2` kopiert. Anschließend wird der Wert von `Arg2` auf 9 gesetzt. Wenn beide Variablen danach ausgegeben werden, dann haben sie auch unterschiedliche Werte; einmal den Wert 7 für `Arg1` und 9 für `Arg2`. Sie sehen, dass bei der Zuweisung `Arg2 = Arg1` eine Kopie des Wertes durchgeführt wurde.

Referenztypsemantik

Bei der Zuweisung eines Referenztyps hingegen wird keine Kopie der entsprechenden Variablen gemacht. Vielmehr wird die Adresse im Heap, die auf das tatsächliche Objekt zeigt, kopiert, und somit zeigen beide Variablen auf denselben Speicherbereich. Wird die zugewiesene Variable manipuliert, hat das Auswirkungen auf die Originalvariable.

```
<%@ Page Language="VB" %>
<script runat="server">
  Sub Page_Load()
    Dim Arg1 As New Person()
    Arg1.Name = "Christian"
    Dim Arg2 As Person = Arg1
    Arg2.Name = "Tobias"
    Response.Write(Arg1.Name.ToString())
```

```
    Response.Write(Arg2.Name.ToString())
  End Sub
</script>
```

Listing 3.7: Beispiel für Referenztypsemantik

Im Beispiel in Listing 3.7 wird eine Variable `Arg1` vom Typ `Person` (ohne der Objektorientierung vorgreifen zu wollen, sehen Sie die Definition dieses Typs in Listing 3.8) definiert und instanziiert. Dann wird der Eigenschaft `Name` der Wert »Christian« zugewiesen. Im nächsten Schritt wird eine zweite Personenvariable `Arg2` definiert, und ihr wird `Arg1` zugewiesen. Dabei wird die Adresse von `Arg1` aufgrund der Referenztypsemantik in die Speicherzelle `Arg2` kopiert. Anschließend wird die Name-Eigenschaft von `Arg2` auf »Tobias« gesetzt. Wenn von beiden Variablen danach die Name-Eigenschaft ausgegeben wird, dann haben sie identische Werte, nämlich »Tobias«. Sie sehen, dass bei der Zuweisung `Arg2 = Arg1` eine Kopie der Adresse und nicht des Wertes durchgeführt wurde.

```
Public Class Person
    Private _name As String
    Public Property Name() As String
        Get
            Return _name
        End Get
        Set(ByVal value As String)
            _name = value
        End Set
    End Property
End Class
```

Listing 3.8: Definition einer Klasse Person

ACHTUNG

Obwohl es sich beim Datentyp `String` um einen Referenztyp handelt, verhält er sich wie ein Wertetyp. `String` ist somit ein Referenztyp mit Wertetypsemantik. Die Ausnahme wurde von den Framework-Entwicklern gemacht, um die Verarbeitung dieses doch wesentlichen Datentyps intuitiver zu gewährleisten.

Nullable Typen

Wertetypen werden bei der Definition mit einem Standardwert versehen, sofern Sie nicht direkt einen Initialwert zuweisen. Für sämtliche Zahlenwerte ist das der Wert 0, für den Datentyp `Boolean` der Wert `false` und für den Datentyp `Date` der 01.01.0001.

Das bedeutet, dass Wertetypen niemals den Wert `Nothing` annehmen können. Gerade bei Datenbankanwendungen ist es jedoch sinnvoll, Spalten, die den Wert `null` (nicht belegt) besitzen dürfen, nicht mit einem Initialwert zu versorgen. Stellen Sie sich vor, in einer beliebigen Tabelle gibt es ein Feld Geburtsdatum. Wenn der Wert nicht bekannt ist, wird eben nichts eingetragen. Der Wert in der Datenbank ist `null`, und somit weiß jeder, dass diese Information nicht bekannt ist. Wird jedoch der Initialwert eingetragen, hätten diese Personen stattdessen am 01.01.0001 Geburtstag. Peinlich für denjenigen, der in seiner Anwendung eine Serienbrieffunktionalität besitzt, die den Personen dann zum 2008. Geburtstag gratuliert.

Deswegen wurden in der Version 2.0 des Frameworks sogenannte *Nullable Types* eingeführt. Das bedeutet, dass Wertetypen jeweils ein Nullable-Äquivalent besitzen, das auch den Wert `Nothing` annehmen kann.

Einen **Nullable Integer** definieren Sie dabei wie folgt:

```
Dim Arg1 As Nullable(Of Integer)
```

NEU

Mit der Einführung von Visual Basic 9 können Sie die Definition auch wie folgt angeben:

```
Dim Arg1 as Integer?
```

3.3.5 Kontrollstrukturen und Schleifen

Kontrollstrukturen und Schleifen stellen zwei grundlegende Elemente einer Programmiersprache dar.

Bei einer Kontrollstruktur wird auf der Basis von vorher ermittelten Daten eine Entscheidung für den weiteren Programmablauf getroffen.

Durch Schleifen wird sichergestellt, dass bestimmte Abschnitte mehrfach zu durchlaufen sind, bis ein bestimmter Status erreicht ist.

Weiterhin gibt es noch den Sprungbefehl als primitive Variante einer Schleife. Was ist ein Sprungbefehl? Ein Programm ist üblicherweise ein Block, der aus mehreren Programmzeilen besteht. Ein Sprungbefehl veranlasst das Programm, zu einer bestimmten Stelle zu springen und die Ausführung von dieser Stelle aus weiter fortzuführen.

Diese wichtigen Elemente der Programmierung wollen wir Ihnen in den nächsten Abschnitten näher bringen.

Im Wesentlichen werden von Visual Basic zwei grundlegende Arten von Kontrollstrukturen zur Verfügung gestellt. Es gibt sie in unterschiedlichen Varianten, aber diese Abwandlungen dienen dem Zweck, die Lesbarkeit Ihres Programmcodes zu erhöhen, denn Sie können die durch die Variante erzielte Besonderheit auch auf einem anderen Weg mit einer anderen der bereitstehenden Kontrollstrukturen erreichen.

Die Kontrollstruktur If ... Then ... Else

Die Abfrage eines Zustandes – sei es der Inhalt einer Variablen, das Abfragen eines bestimmten Systemstatus oder der Vergleich zweier Datenstrukturen – und die anschließende Weiterverarbeitung aufgrund des Ergebnisses dieser Prüfung stellen ein wesentliches Element für die Erstellung von Programmen dar.

Die Syntax für diese Statusabfragen lautet in Visual Basic:

```
If Bedingung Then
  Anweisung(en)
End If
```

Sie können Ihre Anweisungen um einen alternativen Anweisungsblock erweitern. Es wird dann entweder der eine oder der andere Block ausgeführt und anschließend mit dem normalen Programmcode fortgefahren. Hierfür ergänzen Sie Ihre `If`-Bedingung um einen `Else`-Block.

Die Syntax für diesen Bedingungsblock sieht dann wie folgt aus.

```
If  Bedingung Then
  Anweisung(en)
Else
  Anweisung(en)
End If
```

Beachten Sie, dass Sie durch die Angabe der Schlüsselwörter `Else` und `End If` den Abschnitt der Folgeanweisungen begrenzen. Verwenden Sie den Visual Web Developer als Editor, geschieht ein Einrücken zum optischen Absetzen des vom Bedingungsblock begrenzten Programmcodes automatisch. Wenn Sie einen Editor verwenden, mit dem dies nicht möglich ist, sollten Sie aus Gründen der Übersichtlichkeit eine solche Strukturierung manuell vornehmen. Dies gilt auch für die weiteren Elemente, die Ihren Programmcode strukturieren.

Sollte der Visual Web Developer diese automatische Formatierung einmal nicht durchführen, dies geschieht zumeist wenn ein Compilerfehler vorliegt, dann können Sie die automatische Formatierung mit der Tastenkombination STRG+K, STRG+D durchführen lassen (Voraussetzung ist natürlich, dass keine Compilerfehler im Code vorhanden sind).

Noch ein kleiner Tipp am Rande, wenn Sie nach einem VB Schlüsselwort die Taste TAB zweimal drücken, dann wird im Visual Web Developer sofort der gesamte Block für dieses Schlüsselwort automatisch angelegt.

```
<%@ Page Language="VB" %>
<script runat="server">
  Sub Page_Load()
    Dim ZufallsObj As New Random()
    Dim Zufall1 As Double
    Dim Zufall2 As Double
    Randomize()
    Zufall1 = ZufallsObj.Next()
    Zufall2 = ZufallsObj.Next()
    Response.Write("If...Then...Else")
    Response.Write(" <br> ")
    Response.Write("Zufallszahl1: ")
    Response.Write(Zufall1.ToString)
    Response.Write(" <br> ")
    Response.Write("Zufallszahl2: ")
    Response.Write(Zufall2.ToString)
    Response.Write(" <br> ")
    If Zufall1 > Zufall2 Then
      Response.Write("Erste Zufallszahl grösser")
      Response.Write(" <br> ")
    Else
      Response.Write("Zweite Zufallszahl grösser")
      Response.Write(" <br> ")
    End If
  End Sub
</script>
```

Listing 3.9: Eine Bedingungsabfrage mit If ... Then ... Else und generierten Zufallszahlen (IfThenElse.aspx)

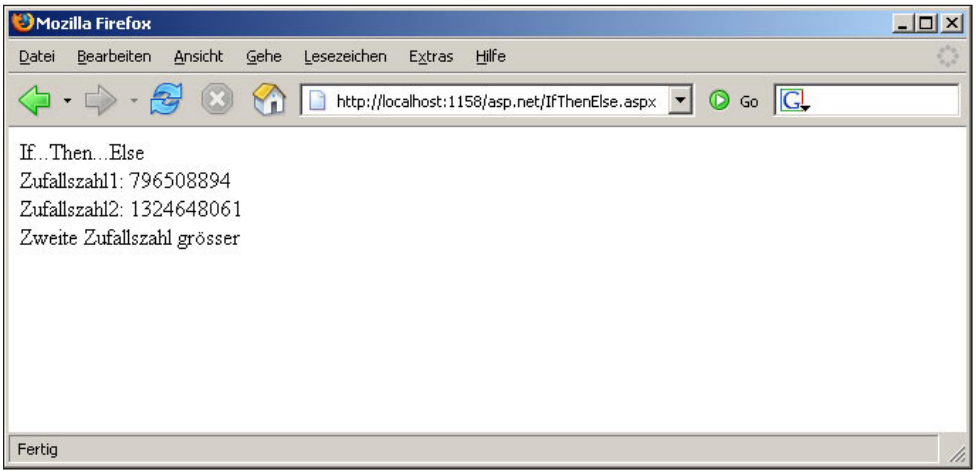

Abbildung 3.15: Eine Bedingungsabfrage mit If ... Then ... Else und generierten Zufallszahlen

Noch eine kleine ergänzende Anmerkung: In diesem Beispiel haben wir die `Random`-Klasse des .NET Frameworks verwendet, um zwei Zufallszahlen zu erzeugen, die verglichen wurden. Wir haben zunächst ein `Random`-Objekt angelegt und mittels der Methode `Random.Next()` dann die Zufallszahlenerzeugung durchgeführt.

Sie können zwischen dem `If`- und dem `Else`-Block optional noch weitere `ElseIf`-Blöcke einbauen.

```
<%@ Page Language="VB" %>
<script runat="server">
  Sub Page_Load()
    Dim ZufallsObj As New Random()
    Dim Zufall1 As Integer
    Randomize()
    Zufall1 = ZufallsObj.Next()
    Response.Write("If...Then...ElseIf...Then...Else")
    Response.Write(" <br> ")
    Response.Write("Zufallszahl1: ")
    Response.Write(Zufall1.ToString)
    Response.Write(" <br> ")
    If Zufall1 < 30 Then
      Response.Write("kleine Zufallszahl")
      Response.Write(" <br> ")
    ElseIf Zufall1 < 60 Then
      Response.Write("mittlere Zufallszahl")
      Response.Write(" <br> ")
    Else
      Response.Write("große Zufallszahl")
      Response.Write(" <br> ")
    End If
  End Sub
</script>
```

Listing 3.10: Eine Bedingungsabfrage mit If-ElseIf-Else (IfThenElseIfThenElse.aspx)

Die Verwendung von mehreren `ElseIf`-Blöcken macht Ihren Code jedoch sehr schnell übersichtlich. Die bessere Alternative dazu sehen Sie im folgenden Abschnitt.

Die Kontrollstruktur Select ... Case

Eine weitere Kontrollstruktur, mit der Sie einen ganzen Satz von Zuständen prüfen und bearbeiten können, bilden die nachfolgenden Bedingungsblöcke.

Mit der Kontrollstruktur `Select ... Case` haben Sie die Möglichkeit, quasi eine Reihe von `If ... Then ... ElseIf ... Then ... Else`-Abfragen gebündelt ausführen zu lassen. Die Syntax für diese Kontrollstruktur ist wie folgt:

```
Select Wert
Case Ergebnis1
  Anweisungen
Case Ergebnis2
  Anweisungen
Case Ergebnis3
  Anweisungen
...
Case Else
  Anweisungen
End Select
```

Der `Case Else`-Zweig ist auch in diesem Fall optional.

Abgeschlossen wird diese Kontrollstruktur durch `End Select`.

Wenn Sie also den Inhalt einer Variablen auf verschiedene Werte überprüfen und auf Basis dieser unterschiedlichen Werte Anweisungsketten ausführen wollen, können Sie dies mit der `Select ... Case`-Kontrollstruktur entsprechend durchführen.

```
<%@ Page Language="VB" %>
<script runat="server">
  Sub Page_Load()
    Dim Arg1 As Integer
    Arg1 = Now.DayOfWeek
    Response.Write("Der Wochentag ist: ")
    Response.Write("<br> ")
    Select Case Arg1
      Case 1
        Response.Write("Montag")
        Response.Write("<br> ")
      Case 2
        Response.Write("Dienstag")
        Response.Write("<br> ")
      Case 3
        Response.Write("Mittwoch")
        Response.Write("<br> ")
      Case 4
        Response.Write("Donnerstag")
        Response.Write("<br> ")
      Case 5
        Response.Write("Freitag")
        Response.Write("<br> ")
      Case 6
        Response.Write("Samstag - Wochenende!")
        Response.Write("<br> ")
      Case 0
        Response.Write("Sonntag - Wochenende!")
```

```
        Response.Write("<br> ")
    End Select
  End Sub
</script>
```

Listing 3.11: Die Kontrollstruktur Select Case ermittelt den aktuellen Wochentag (SelectCase.aspx).

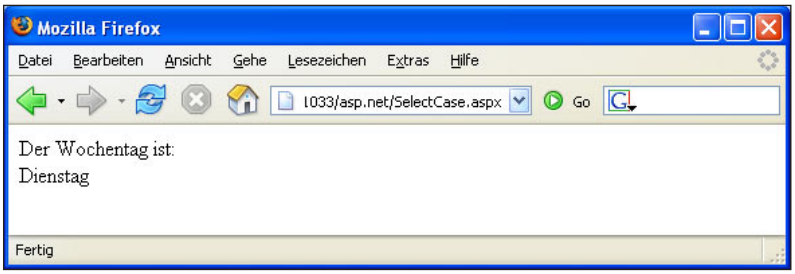

Abbildung 3.16: Die Kontrollstruktur Select Case ermittelt den aktuellen Wochentag (SelectCase.aspx).

Schleifen und unbedingte Sprungbefehle

In der Programmierung haben Sie immer wieder Abschnitte, die mehrfach durchlaufen werden müssen, oder Abschnitte, die strukturell gleichartig sind und sich nur von den Inhalten der Variablen unterscheiden. Als prozedurale Programmierelemente stellt Ihnen Visual Basic hierfür verschiedene Schleifenstrukturen und Sprungbefehle zur Verfügung.

Der Goto-Befehl als unbedingter Sprung

Der `Goto`-Befehl ist ein sehr mächtiger, aber auch primitiver Befehl. Im Prinzip wird zur Verwendung dieses Sprungbefehls eine Stelle im Programmcode markiert. Zu dieser Markierung springen Sie dann, indem Sie den Befehl `Goto` Markierung aufrufen. Alternativ können Sie auch die Zeilennummer, zu der zu springen ist, angeben.

Die Gefahr bei der Verwendung des `Goto`-Sprungbefehls liegt darin, dass sie Programme sehr unübersichtlich und schwer lesbar machen können. Der von Ihnen erstellte Code wird schon bald nach Fertigstellung vermutlich nicht einmal für Sie nachvollziehbar sein. In frühen Basic-Varianten musste man, mangels Alternativen, unbedingte Sprungbefehle exzessiv verwenden. Dies führte häufig zu extrem verschachteltem Programmcode, dem sogenannten Spaghetticode.

Ihnen steht unter Visual Basic eine ganze Reihe von alternativen Programmierelementen zur Verfügung, welche die Verwendung des `Goto`-Befehls fast unnötig machen.

Die For ... Next-Schleife

Ein besseres Mittel für die Programmierung von Schleifen ist beispielsweise die `For ... Next`-Schleife, die einen Zähler mitführt, der sicherstellen kann, dass die Schleife wieder verlassen werden kann, wenn dieser Zähler einen bestimmten Wert erreicht hat.

Die Syntax einer `For ... Next`-Schleife ist die nachfolgende:

```
For Bedingung = Wert1 to Wert2 Step Schrittfolge
  Anweisungen
  ...
Next
```

Über den Parameter Schrittfolge, der entweder eine ganze Zahl oder auch eine Kommazahl sein kann, (es spielt dabei keine Rolle, ob die Zahl positiv oder negativ ist), können Sie auch steuern, wie oft eine Schleife durchlaufen wird. Die Schleife können Sie so auch leicht in eine Endlosschleife verwandeln. Hierfür setzen Sie einfach die Schrittweite für den Zähler auf 0.

HINWEIS

Die `For ... Next`-Schleife können Sie übrigens vorzeitig verlassen, indem Sie im Anweisungsblock die Anweisung `Exit For` einbauen.

Schleifen können Sie beliebig ineinander schachteln. Ein Überkreuzen von zwei Schleifen ist – logisch nachvollziehbar – aber nicht möglich.

```
<%@ Page Language="VB" %>
<script runat="server">
  Sub Page_Load()
    Dim Arg2 As Integer
    Arg2 = 0
    Response.Write("Schleife mit For...Next")
    Response.Write(" <br> ")
    For Arg1 As Double = 2 To 40 Step 3.4
      Arg2 = Arg2 + 1
      Response.Write("Argument in Durchlauf ")
      Response.Write(Arg2.ToString)
      Response.Write(": ")
      Response.Write(Arg1.ToString)
      Response.Write(" <br> ")
      If Arg2 > 10 Then Exit For
    Next
    Response.Write("Programm Fertig ")
  End Sub
</script>
```

Listing 3.12: Eine For ... Next-Schleife (ForNext.aspx)

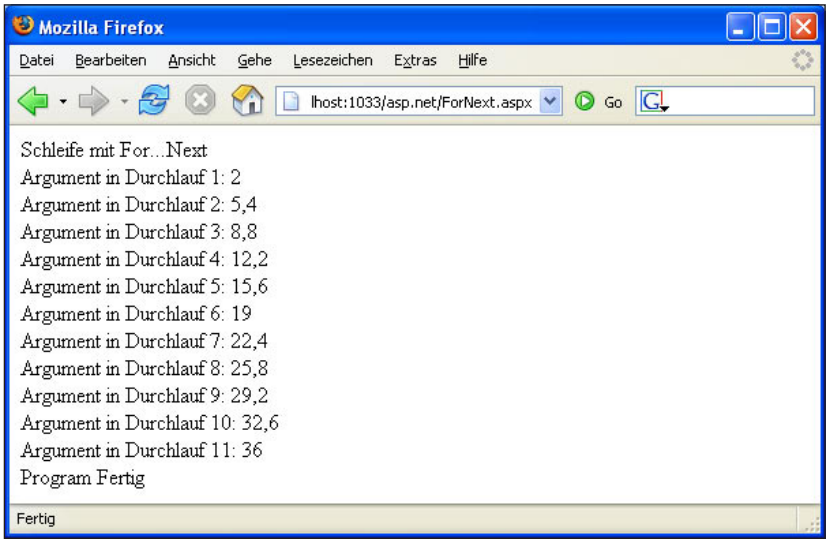

Abbildung 3.17: Eine For ... Next-Schleife (ForNext.aspx)

Die For Each-Schleife

Sehr ähnlich zur `For … Next`-Schleife verhält sich die `For Each`-Schleife. Bei dieser Art von Schleife durchlaufen Sie aufzählbare Elemente wie zum Beispiel Arrays oder Collections (Auflistungsobjekte). Deswegen brauchen Sie auch keine Schleifenzähler mitzugeben, da diese aufzählbaren Elemente vom ersten bis zum letzten Eintrag automatisch durchlaufen werden.

Die Syntax einer `For Each`-Schleife ist die nachfolgende:

```
For Each Element as Datentyp in Liste
  Anweisungen
  ...
Next
```

HINWEIS

Die `For Each`-Schleife können Sie übrigens genauso wie die `For … Next`-Schleife mit `Exit For` vorzeitig verlassen.

Auch `For Each`-Schleifen können Sie beliebig ineinander schachteln, jedoch auch nicht überkreuzen.

```
<%@ Page Language="VB" %>
<script runat="server">
  Sub Page_Load()
    Dim Arg1() As Integer = {1 , 2, 3, 4, 5}
    Response.Write("Schleife mit For Each")
    Response.Write(" <br> ")
    For Each Arg2 As Integer In Arg1
      Response.Write("Argument in Durchlauf ")
      Response.Write(Arg2.ToString)
    Next
    Response.Write("Program Fertig ")
  End Sub
</script>
```

Listing 3.13: Eine For Each-Schleife (ForEach.aspx)

Die Do ... Loop-Schleife

Mit der `Do ... Loop`-Schleife haben Sie eine weitere Möglichkeit, eine Schleifenstruktur aufzubauen. Hier können Sie bei jedem Schleifendurchlauf eine Bedingung auf ihren Wahrheitswert überprüfen.

Die Syntax einer `Do ... Loop`-Schleife ist untenstehend beschrieben.

```
Do While Bedingung
  Anweisungen
Loop
```

In diesem Falle wird der Wahrheitswert der Bedingung am Anfang der Schleife (kopfgesteuerte Schleife) überprüft.

Alternativ haben Sie die Möglichkeit, die Überprüfung der Bedingung an das Ende der Schleife zu stellen (fußgesteuerte Schleife). Dann stellt sich die Syntax der `Do ... Loop`-Schleife wie folgt dar:

```
Do
  Anweisungen
Loop While Bedingung
```

Diese Schleife wird jetzt so lange durchlaufen, wie die `Bedingung` den logischen Wert wahr besitzt, mindestens aber einmal. Wenn Sie eine Schleife so lange durchlaufen wollen, wie eine Bedingung den Status falsch hat, dann haben Sie noch folgende alternative Syntax zur Verfügung (auch hier kann die Bedingung natürlich auch am Ende der Schleife überprüft werden).

```
Do  Until Bedingung
  Anweisungen
Loop
```

Auch bei der `Do ... Loop`-Schleife haben Sie die Möglichkeit, diese mit einem »Ausstiegsbefehl« vorzeitig zu verlassen. In diesem Fall ist das zugehörige Schlüsselwort `Exit Do`. Eine typische Schleifenstruktur hätte also die folgende Form:

```
Do Until Bedingung
  Anweisungen
  If Bedingung2 Then
    Exit Do
  End If
  Anweisungen
Loop
```

Nachfolgend auch hier noch einmal ein Listing mit der zugehörigen Bildschirmausgabe.

```
<%@ Page Language="VB" %>
<script runat="server">
  Sub Page_Load()
    Dim Ket As String = "Test"
    Dim CatKet As String = ""
    Dim KetErg As String = "TestTestTestT"
    Dim Zaehler As Integer = 0
    Response.Write("Eine Schleife mit Do...Loop <br>")
    Response.Write("KetErg:")
    Response.Write(KetErg)
    Do While CatKet <> KetErg
      Response.Write("<br>CatKet sieht jetzt so aus:")
      Response.Write(CatKet)
      CatKet = CatKet & Ket
      If Len(CatKet) > Len(KetErg) Then
        Response.Write("<br>Konnte KetErg nicht bauen <br>")
        Exit Do
      End If
    Loop
    Response.Write("Abschlusswert von CatKet:")
    Response.Write(CatKet)
    Response.Write("<br>Teil 1 Fertig <br>")
    CatKet = ""
    Do
      Response.Write("<br>CatKet sieht jetzt so aus:")
      Response.Write(CatKet)
      CatKet = CatKet & Ket
      If Len(CatKet) > Len(KetErg) Then
        Response.Write("<br>Konnte KetErg nicht bauen <br>")
        Exit Do
      End If
    Loop Until CatKet = KetErg
    Response.Write("Abschlusswert von CatKet:")
```

```
    Response.Write(CatKet)
    Response.Write("<br>Teil 2 Fertig <br>")
  End Sub
</script>
```

Listing 3.14: Die Do ... Loop-Schleife (DoLoop.aspx)

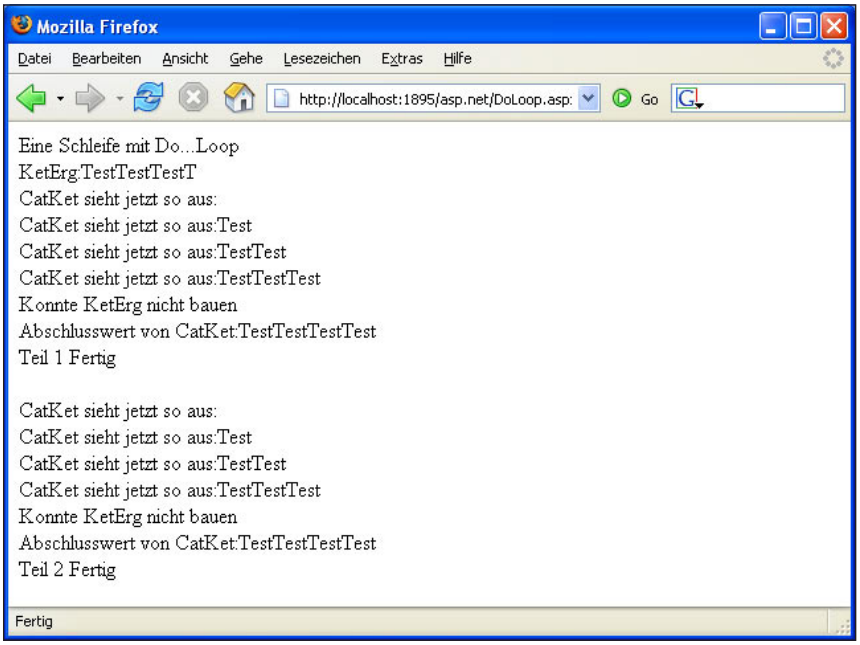

Abbildung 3.18: Die Do ... Loop-Schleife (DoLoop.aspx)

Die While ... End While-Schleife

Als letzte Möglichkeit für die Programmierung von Schleifen unter Visual Basic existiert eine Schleife, die in den Möglichkeiten der `Do ... Loop`-Schleife auch enthalten ist.

Sie heißt `While ... End While`-Schleife.

Ihre Syntax gestaltet sich wie folgt:

```
While Bedingung
  Anweisungen
End While
```

Auch bei dieser Schleife können Sie über die `Exit While`-Hintertür die Schleife vorzeitig verlassen.

```
While Bedingung
  Anweisungen
  If Bedingung2 Then
    Exit While
  End If
  Anweisungen
End While
```

Nachfolgend das obligatorische kleine Listing mit Ausgabe.

```
<%@ Page Language="VB" %>
<script runat="server">
  Sub Page_Load()
    Dim ZufallsObj As New Random()
    Dim Arg As Single = 0
    Dim Count As Integer = 0
    Response.Write("Schleife mit ")
    Response.Write(" While...End While:<br> ")
    Response.Write("Wie oft ist meine Zufallszahl kleiner 0.8?<br> ")
    While Arg < 0.8
      Count = Count + 1
      Arg = ZufallsObj.NextDouble()
      Response.Write("Durchlauf ")
      Response.Write(Count.ToString)
      Response.Write(" <br>  Zufallswert:")
      Response.Write(Arg.ToString)
      Response.Write(" <br> ")
      If Count = 10 Then
        Response.Write(" <br>10 Versuche erreicht <br>")
        Exit While
      End If
    End While
    Response.Write("Fertig <br>")
  End Sub
</script>
```

Listing 3.15: Eine Schleife mit While (WhileEndWhile.aspx)

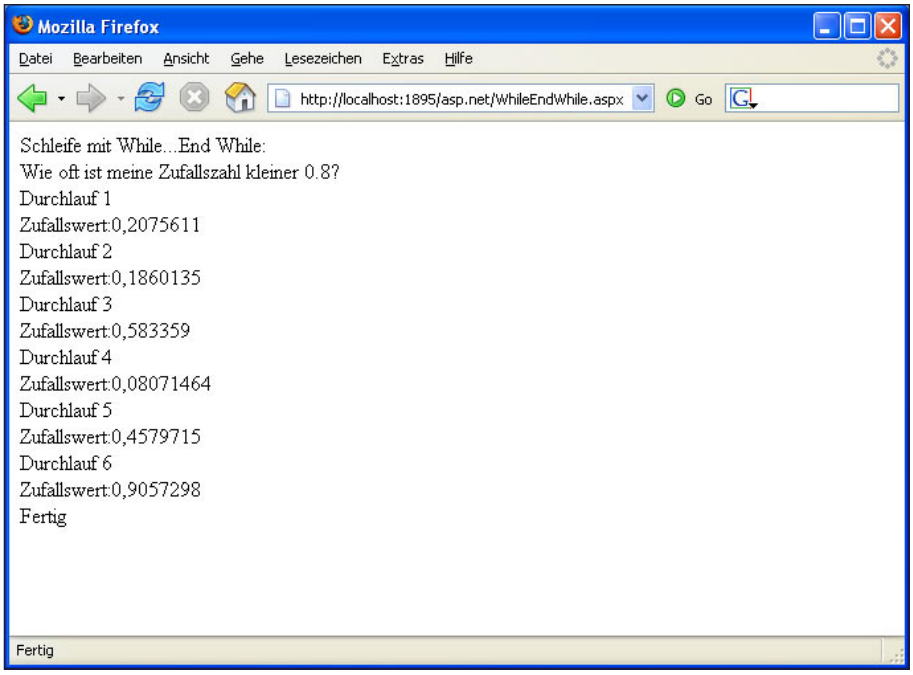

Abbildung 3.19: Eine Schleife mit While (WhileEndWhile.aspx)

Auch in diesem Programm haben wir das `Random`-Objekt erzeugt. Und dann mittels der Methode `Random.NextDouble()` eine Zufallszahl zwischen 0 und 1 erzeugt.

3.4 Programmelemente und Programmebenen

Nachdem Sie nun verschiedene grundlegende Programmelemente für die Visual Basic-Programmierung kennengelernt haben, ist es an der Zeit, Ihnen einen Überblick über die Strukturierung von Programmen zu geben.

Sie haben die Möglichkeit, diverse Programmelemente zusammenzufassen und in Dateien abzuspeichern.

Grundsätzlich können Sie diese Dateien über sogenannte Assemblies in Ihren zu übersetzenden Quellcode importieren lassen und auf diese Weise verschiedene Codeelemente zusammenstellen. Um die Programmelemente möglichst einfach zusammenstellen zu können, bietet Visual Basic ein ganzes Reich von Programmelementen an, die Ihren Code weiter modular strukturieren.

Hier sind neben prozeduralen Programmelementen auch die objektorientierten Strukturen zu nennen. Zunächst möchten wir Ihnen Funktionen und Prozeduren als prozedurale Programmelemente vorstellen. In einem zweiten Schritt werden wir dann die erweiterten Elemente der objektorientierten Programmierung ergänzen, die auch auf Funktionen und Prozeduren anwendbar sind.

3.4.1 Funktionen und Prozeduren

Mit Funktionen und Prozeduren erhalten Sie die Möglichkeit, Programmabschnitte zu gliedern und diese einzelnen Gliederungselemente in andere Programme zu übernehmen, einfach indem Sie die Funktion oder die Prozedur übernehmen. Sie müssen bei der Erstellung von Prozeduren oder Funktionen, die Sie in andere Programme übernehmen wollen, einige Dinge berücksichtigen, auf die wir nachfolgend eingehen werden.

Mit der objektorientierten Programmierung haben Methoden wesentliche Aufgaben von Funktionen und Prozeduren übernommen, in ihrer Programmierung sind beide Elemente sehr ähnlich, sodass sich die Funktionen und Prozeduren einfach in Methoden umwandeln lassen, mit denen eine noch bessere Wiederverwendbarkeit ermöglicht wird.

Programmierung von Prozeduren

Prozeduren sind kleine Unterprogramme, die über den von Ihnen vergebenen Prozedurnamen aufgerufen werden. Sie haben die Möglichkeit, eine Liste von Parametern zu definieren, die an diese Prozedur übergeben werden, mit denen die Prozedur dann arbeiten kann.

In einer Prozedur haben Sie die Möglichkeit, lokale Variablen einzusetzen. Dies sind Variablen, die im globalen Programmumfeld (außerhalb der Prozedur) unbekannt sind und nicht verwendet werden. Sie sollten es vermeiden, den Gültigkeitsbereich von Variablen zu groß zu wählen, und ihn im Wesentlichen auf diese lokalen Variablen innerhalb von Prozeduren und Funktionen beschränken. Ansonsten wäre die Wiederverwendbarkeit des Codes stark eingeschränkt.

ACHTUNG

Eine lokale Variable, die innerhalb eines Blocks (Schleife, Kontrollstruktur etc.) definiert wird, ist auch nur innerhalb dieses Blocks gültig, man spricht dann von sogenannten blockweiten Variablen. Außerhalb des Blocks sind diese Variablen nicht zugänglich, eine Verwendung führt zu einem Compilerfehler.

In der Konsequenz heißt das: Verwenden Sie möglichst lokal deklarierte Variablen. Verzichten Sie so weit wie möglich auf Variablen, die übergreifend über verschiedene Programmabschnitte bekannt sein müssen.

Wenn Sie von einer aufrufenden Ebene Daten an eine Prozedur oder Funktion weitergeben wollen, bieten sich hierfür die in einer Prozedur und Funktion deklarierbaren Übergabeparameter an. Hierbei erfolgt eine Wertübergabe an die Übergabeparameter, die innerhalb der Prozedur wie lokale Variablen zu verwenden sind. Außerhalb der Prozedur sind diese deklarierten Übergabeparameter wie auch die sonstigen dort deklarierten lokalen Variablen nicht gültig.

Sie haben die Möglichkeit, für jeden Parameter festzulegen, ob er durch die Prozedur verändert werden kann oder ob eben dies nicht möglich ist. Dies erfolgt durch das Hinzufügen der Schlüsselwörter `ByVal` (Übergabe einer Kopie des Wertes, keine Manipulation möglich) oder `ByRef` (Übergabe der Referenz auf die Variable, Manipulation möglich). `ByVal` ist der Standardwert, dies ist ein wesentlicher Unterschied zu den Visual Basic-Versionen 1–6. Bei diesen alten Versionen wurden Parameter standardmäßig `ByRef` übergeben.

Eine Prozedur in Visual Basic wird durch die nachfolgende Syntax aufgerufen:

```
Sub Prozedurname([ByVal|ByRef] Parameter As Datentyp, [ByVal|ByRef] Parameter2 As Daten-
typ, ...)
  Anweisungen
End Sub
```

Sie können beliebig viele (auch keine) Parameter deklarieren und übergeben.

```
<%@ Page Language="VB" %>
<script runat="server">
  Sub Page_Load()
    Dim Arg1 As String = "Testtext"
    Dim Arg2 As Integer = 0
    Dim Arg3 As String = "Hallo"
    Response.Write("Aufruf einer Prozedur <br>")
    Response.Write("Einige Variablen vor Prozeduraufruf:")
    Response.Write("<br>Arg1:")
    Response.Write(Arg1)
    Response.Write("<br>Arg2:")
    Response.Write(Arg2.ToString)
    Response.Write("<br>Arg3:")
    Response.Write(Arg3)
    Response.Write(" <br> ")
    Prozedur(Arg1, Arg3)
    Response.Write("<br>Variablen nach Prozeduraufruf:")
    Response.Write("<br>Arg1:")
    Response.Write(Arg1)
    Response.Write("<br>Arg2:")
    Response.Write(Arg2.ToString)
    Response.Write("<br>Arg3:")
    Response.Write(Arg3)
    Response.Write(" <br> ")
```

```
    Response.Write("Program Fertig ")
  End Sub

  Sub Prozedur(ByVal Arg2 As String, ByRef Arg3 As String)
    Response.Write("<br>An Prozedur übergeben:")
    Response.Write(" <br>Arg2:")
    Response.Write(Arg2)
    Response.Write(" <br>Arg3:")
    Response.Write(Arg3)
    Arg2 = "Hallo2"
    Arg3 = "Hallo3"
    Response.Write("<br>Neuzuweisung der Variablen <br>")
    Response.Write("Arg2:")
    Response.Write(Arg2)
    Response.Write("<br>Arg3:")
    Response.Write(Arg3)
    Response.Write("<br>Prozedur Ende<br>")
  End Sub
</script>
```

Listing 3.16: Eine Prozedur wird aufgerufen (Procedure.aspx).

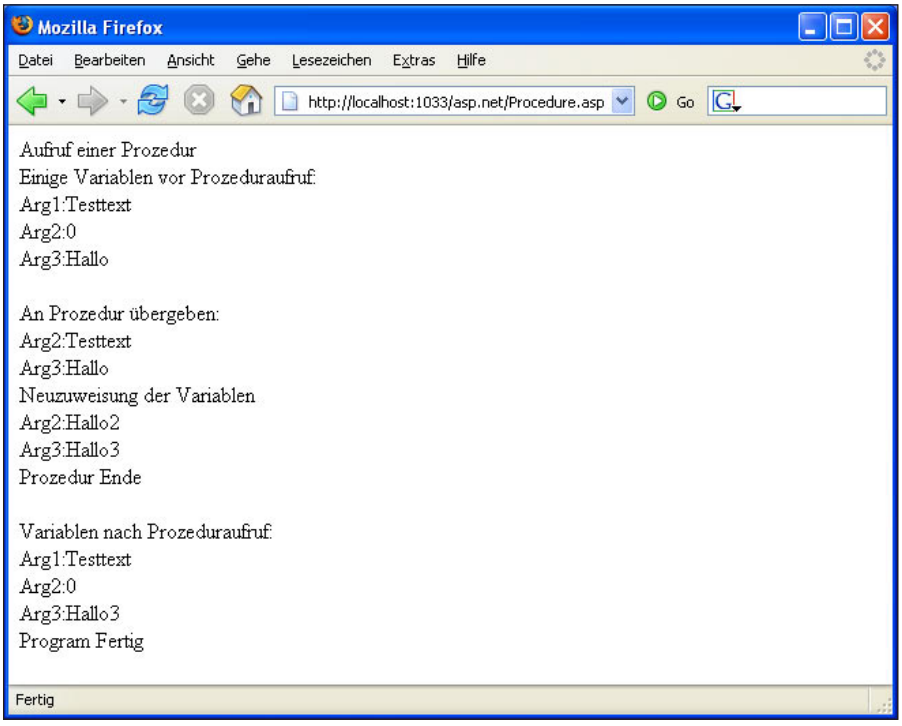

Abbildung 3.20: Eine Prozedur wird aufgerufen (Procedure.aspx).

Weiterhin haben Sie die Möglichkeit, eine Prozedur auch vorzeitig über `Exit Sub` oder `Return` zu verlassen. Dies ist beispielsweise sinnvoll, wenn eine bestimmte Bedingung bereits vor Ende der Prozedur eingetreten ist.

HINWEIS

Sie können vor einer Prozedur noch per Schlüsselwort definieren, in welchem Kontext die Zugriffe auf die Prozeduren ermöglicht werden können. Die entsprechenden Schlüsselwörter und Bedeutungen sind im nachfolgenden Abschnitt zur Objektorientierung erklärt. In Tabelle 3.11 finden Sie diese Informationen zusammengefasst.

Syntax und Entwicklung von Funktionen

Funktionen sind in Visual Basic sehr ähnlich zu Prozeduren aufgebaut. Der einzige – aber wesentliche – Unterschied besteht darin, dass Sie mit Funktionen einen Rückgabewert definieren, der an den aufrufenden Programmabschnitt zurückgeliefert wird.

Da der Rückgabewert auch ein strukturierter Datentyp sein kann, haben Sie die Möglichkeit, innerhalb von Funktionen komplexe Datenstrukturen aufzubauen und an den aufrufenden Programmabschnitt zurückzuliefern.

Eine Funktion in Visual Basic kann also beispielsweise die folgende Syntax haben:

```
Function Funktionsname(Parameterliste As Datentyp,…) As Rückgabedatentyp
  Anweisungen
  Return Rückgabewert
End Function
```

Die Verwendung und Deklaration der Übergabeparameter erfolgt analog zur Verwendung bei Prozeduren. Sie müssen nur zusätzlich noch den Datentyp, den die Funktion besitzt, mit deklarieren. Damit legen Sie fest, von welchem Datentyp der Rückgabewert der Funktion sein soll.

In dem obigen Beispiel wird explizit über das Schlüsselwort `Return` und dem angehängten Wert der Rückgabewert angegeben und unmittelbar die Funktion verlassen.

Als weiteres mögliches Szenario einer Zuweisung des Rückgabeparameters können Sie auch folgenden Weg wählen:

```
Function Funktionsname(Parameterliste As Datentyp,…) As Rückgabedatentyp
  Anweisungen
  Funktionsname = Rückgabewert
  Anweisungen
End Function
```

Hier wird die Funktion weiter ausgeführt, und die Rückgabe an das aufrufende Programm erfolgt erst bei Erreichen von `End Function`.

Die Funktion wird im Hauptprogramm aufgerufen:

```
Ergebnis = Funktionsname(Parameter1,…)
```

Nun noch ein vollständiges Programm:

```
<%@ Page Language="VB" %>
<script runat="server">
  Sub Page_Load()
    Dim Arg1 As Integer = 0
    Dim Erg As Integer = 0
    Response.Write("Aufruf einer Funktion")
    Response.Write(" die eine ganzzahlige Zufallszahl berechnet.<br> ")
    Response.Write("Die Obergrenze (9) wird als Parameter mitgegeben.<br>")
```

```
    Erg = Zufallsinteger(9)
    Response.Write(" Folgende Zahl wurde ermittelt: ")
    Response.Write(Erg)
  End Sub

  Function Zufallsinteger(ByVal Grenze As Integer) As Integer
    Dim Zufall As Single = 0
    Dim Erg As Single
    Zufall = Rnd()
    Erg = Grenze * Zufall
    Zufallsinteger = Convert.ToInt32(Erg)
  End Function
</script>
```

Listing 3.17: Die Verwendung einer Funktion (Function.aspx)

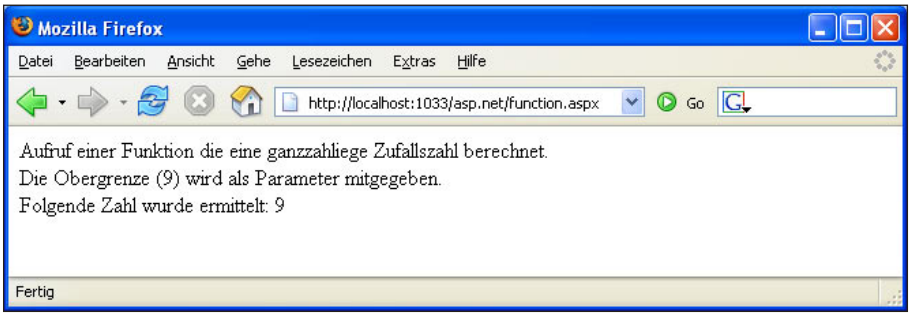

Abbildung 3.21: Die Verwendung einer Funktion

Grundsätzlich ist es sinnvoll, sowohl in Funktionen als auch in Prozeduren ausschließlich mit lokalen Variablen zu arbeiten und Datenübergaben nur über Parameterlisten und übergebene Datenstrukturen durchzuführen.

Diese Vorgehensweise ermöglicht Ihnen eine erheblich einfachere Wiederverwendung von Funktionen in anderen Programmen.

HINWEIS

Auch bei Funktionen können Sie durch vorangestellte Schlüsselwörter wie `Public` oder `Private` die Zugriffsberechtigungen steuern. Im nachfolgenden Abschnitt gehen wir genauer darauf ein. Die Schlüsselwörter und ihre Bedeutung sind in Tabelle 3.11 zusammengefasst.

3.4.2 Objektorientierung

Die Unterstützung der Objektorientierung stellt eine der wesentlichen Erweiterungen in den .NET-Versionen von Visual Basic dar, die gegenüber den alten Visual Basic-Versionen erheblich erweitert und modernisiert wurde. Mit der Objektorientierung werden bestimmte Programmelemente gekapselt und mit Eigenschaften und Logiken (Methoden) versehen. Diese können in anderen Programmabschnitten weiter verwendet werden.

In gewisser Weise stellen sie damit einen weiteren Schritt nach Funktionen und Prozeduren dar, um die Wiederverwendbarkeit von Code zu verbessern.

Klassen

Die Unterstützung von Klassen und ihren Instanzen, den daraus abgeleiteten Objekten, wird ja bereits seit Langem in Visual Basic bereitgestellt. Nachfolgend wollen wir auf die Elemente, die in der aktuellen Version von Visual Basic zur Verfügung stehen, eingehen.

Klassen stellen die Baupläne der daraus erstellten Objekte dar. In einer Klasse werden also alle wesentlichen Elemente festgelegt, die das daraus hervorgehende Objekt beschreiben. Das Objekt kann dabei ein konkretes Programm sein, aber auch der Repräsentant einer Datenstruktur, die definiert wird.

Nachfolgend stellen wir Ihnen die wesentlichen Elemente, die es bei der Definition von Klassen zu beachten gibt, vor.

Die Zugriffsrechte auf und die Sichtbarkeit von Programmcode ist ein wesentliches Konzept der objektorientierten Programmierung. Dieses Konzept wird mit dem Schlagwort Kapselung von Code beschrieben.

Grundsätzlich stellt die Kapselung von Programmabschnitten auch außerhalb der objektorientierten Programmierung ein wesentliches Strukturmittel dar, das die Lesbarkeit und die Wiederverwendbarkeit Ihrer Programmblöcke vereinfachen oder gar erst ermöglichen kann.

Sie fassen mittels Kapselung verschiedene Elemente von Methoden und Eigenschaften zu einer Gruppe zusammen. Teil dieser Gruppe können durchaus auch eine oder mehrere andere Klassen sein. Diese Elemente befinden sich damit in einer definierten Klasse und bilden nach außen eine Einheit. Sie können beispielsweise auch steuern, ob Informationen und Abläufe dieser Einheit von außen frei zugänglich sind oder Einschränkungen unterliegen oder ob Sie gar jeden Zugriff von außen verwehren wollen.

In der nachfolgenden Tabelle sind die unterschiedlichen Zugriffsebenen kurz aufgelistet und erklärt.

Zugriffsbezeichner	Bedeutung
`Public`	Dieses Klassenelement ist öffentlich zugänglich, es gibt keinerlei Zugriffsbeschränkungen nach außen.
`Private`	Dieses Klassenelement kann nur innerhalb des Kontextes, in dem es deklariert wurde, verwendet werden. Dies bedeutet, dass Sie innerhalb einzelner Module, in denen die Deklaration erfolgt ist, freien Zugriff haben, aber nicht außerhalb.
`Protected`	Die Elemente dieser Klasse können nur aus der Klasse, innerhalb derer sie deklariert wurde, verwendet werden oder in einer daraus abgeleiteten (vererbten) Klasse.
`Friend`	Die Elemente der Klasse sind nur in derselben Assembly oder demselben Modul zugänglich. Dieses stellt die Standarddeklaration dar, wenn kein Bezeichner angegeben wurde.
`Protected Friend`	Die Elemente der Klasse sind nur aus der Klasse, in der sie deklariert wurden, zugänglich (oder aus daraus abgeleiteten Klassen) oder aus demselben Modul. Es stellt die Vereinigungsmenge der `Friend`- und der `Protected`- Zugriffsberechtigungen dar.

Tabelle 3.11: Zugriffsberechtigungsebenen von Klassen, Methoden, Eigenschaften und Ähnlichem

HINWEIS

Diese Zugriffsberechtigungsebenen können Sie übrigens auch bei Funktionen und Prozeduren verwenden, die damit innerhalb einer Klasse mit den entsprechenden Berechtigungsstrukturen ausgestattet werden. Dies ist eine Erweiterung des klassischen prozeduralen Gedankens. Auch Funktionen und Prozeduren sind damit objektorientierte Elemente. Sie sind die Methoden des objektorientierten Universums.

Ein weiteres Konzept stellt die Vererbung von Code, Eigenschaften oder Aktionen dar. Die Definition einer Klasse kann auf der Definition einer bereits vorhandenen Klasse aufbauen. Die neue Klasse übernimmt hierbei die Merkmale der Basisklasse. Die Übernahme der Merkmale der Basisklasse bezeichnet man als Vererbung.

Die neue Klasse ist aus der Basisklasse abgeleitet worden. Sie kann zusätzliche Elemente enthalten oder auch ein bereits bestehendes Element durch ein anderes Element ersetzen. Dieses Verhalten nennt man Überschreiben.

Wir werden diese Elemente in den nachfolgenden Abschnitten zu Methoden und Eigenschaften noch kennenlernen.

Die Nutzung der Vererbung bietet sich an, wenn es Klassen gibt, die konzeptionell eine Spezialisierung einer Basisklasse darstellen.

Schlüsselwort	Bedeutung
`Inherits`	Mit dieser Anweisung wird die Basisklasse angegeben, aus der die aktuelle Klasse abgeleitet wird.
`MustInherit`	Diese Klasse ist abstrakt, und die Elemente der Klasse müssen durch vererbte (abgeleitete) Klassen implementiert sein. Die damit gekennzeichnete Klasse ist also zwingend als Basisklasse gekennzeichnet, von der andere Klassen abzuleiten sind.
`NotInheritable`	Diese Klasse darf nicht vererbt (abgeleitet) werden. Diese Klasse ist damit also nicht als Basisklasse verwendbar.

Tabelle 3.12: Schlüsselwörter, die im Zusammenhang mit Vererbung stehen

Kommen wir noch einmal auf das Überschreiben von Elementen einer Klasse zurück. Es existiert eine Reihe von Schlüsselwörtern, die klassifizieren, ob die Elemente überschreibbar sind oder nicht. Diese haben wir in der nachfolgenden Tabelle aufgeführt.

Außerdem haben wir noch die Überladung mit aufgelistet. Überladung ist die Technik, eine Programmeinheit (z.B. eine Prozedur) mehrfach mit demselben Namen innerhalb einer Klasse zu erstellen. Die Prozeduren sollten sich nicht in der bereitgestellten Funktionalität unterscheiden, sondern diese Funktionalität auf Parameter mit unterschiedlichen Datentypen anwenden, die korrekte Prozedur wird anhand der übergebenen Datentypen ausgewählt. Sie können außerdem auch die Anzahl der Parameter zusätzlich variieren. Dies vereinfacht die Verwendung von vielen Eigenschaften erheblich.

Bevor wir nun endgültig zur angekündigten Tabelle kommen, noch ein paar Worte zum sechsten Schlüsselwort, das sich dort wieder findet. Das Shadowing stellt ebenfalls Modifikationen abgeleiteter Klassen aus einer Basisklasse dar.

Schlüsselwort	Bedeutung
`Overloads`	Gruppe von gleichnamigen Prozeduren, die sich in Anzahl und Typ der verwendeten Parameter unterscheiden können. Mit diesem Schlüsselwort kennzeichnen Sie also eine Überladung der Prozeduren.
`Overrides`	Die deklarierte Methode oder Eigenschaft überschreibt eine gleichnamige Methode oder Eigenschaft aus einer Basisklasse. Die Parameter der neu deklarierten Methode oder Eigenschaft müssen in ihren Datentypen und der Anzahl der Parameter mit denen der überschriebenen Methode oder Eigenschaft übereinstimmen.
`Overridable`	Diese Methode oder Eigenschaft kann durch die Methode oder Eigenschaft einer abgeleiteten Klasse überschrieben werden.
`MustOverride`	Diese Methode oder Eigenschaft muss von einer Methode oder Eigenschaft in einer abgeleiteten Klasse überschrieben werden, damit diese Klasse erzeugt werden kann. Dies ist nur bei einer abstrakten Basisklasse zulässig. Damit wird sichergestellt, dass bei nicht implementierten Klassen – also Klassen ohne Code – der notwendige Code in der abgeleiteten Klasse existiert.
`NotOverridable`	Diese Methode oder Eigenschaft darf nicht von einer abgeleiteten Methode oder Eigenschaft überschrieben werden. Dieses ist die Standardbelegung. Wenn eine Overridable-Methode oder Eigenschaft nicht erneut überschrieben werden können soll, dann verwenden Sie dieses Schlüsselwort. In der Basisklasse ist es nicht notwendig.
`Shadows`	Neudefinition von Elementen der Basisklasse. Hierbei wird Polymorphie ausgeschaltet.

Tabelle 3.13: Überladung, Überschreibung und Shadowing

Die Deklaration einer Klasse erfolgt mit dem `Class`-Schlüsselwort. In einer Klasse werden, wie schon erwähnt, bestimmte Programmabschnitte (oder Funktionen) deklariert, es können Datentypen deklariert, gruppiert und bereitgestellt werden.

Kurz: Die Struktur eines Codeblocks wird hier festgelegt. Formal müssen Sie einige weitere Punkte bei der Deklaration einer Klasse und seiner Elemente in Hinblick auf deren Methoden und Eigenschaften beachten.

Eine Klasse wird wie nachfolgend beschrieben deklariert:

```
[Bezeichner] Class Klassenname
  [Inherits Basisklasse]
  [Implements Interfacename]
  [Deklarationen und Unterroutinen]
End Class
```

Innerhalb einer Klasse können Sie nun eine Reihe von Unterroutinen bauen.

Mit diesen Rahmenparametern können Sie nun Klassen und ganze Bibliotheken von Klassen deklarieren.

Strukturierung von Klassen

Wenn Sie eine Reihe von Klassen erstellt haben, stellt sich schnell die Frage nach weiteren Strukturierungsmöglichkeiten dieser Klassen. In diesem Zusammenhang steht Ihnen als Strukturierungselement der `Namespace` zur Verfügung, in dem Sie mehrere Klassen zusammenfassen können.

Auch einen `Namespace` können Sie weiter hierarchisch gliedern, sodass Sie auch mehrere Namespaces (und Klassen) zu einem neuen `Namespace` zusammenfassen können. Es entstehen dabei Strukturen, die in etwa wie folgt dargestellt aussehen:

```
Namespace Ebene1
  Namespace Ebene2
    Class Klassenname
    End Class
  End Namespace
  Class Klassenname
  End Class
End Namespace

Namespace Ebene1
  Class Klassenname
  End Class
End Namespace
```

Auf diese Weise ist auch das .NET Framework organisiert.

Erzeugung von Objekten

Die objektorientierte Programmierung ermöglicht es Ihnen, dynamisch Objekte zu erzeugen, die nach der Verwendung dynamisch auch wieder freigegeben und verworfen werden. Die Erzeugung erfolgt auf der Basis der in der Klassendeklaration festgelegten Struktur. Die Deklarationen in einer Klasse stellen ja den Bauplan des Hauses dar, das durch die konkreten Elemente (z.B. eine gemauerte Treppe, die im Bauplan durch die Skizze einer Treppe symbolisch angelegt ist), also durch Objekte, in der Realität des Programmablaufs erzeugt wird.

Das Erzeugen von Objekten erfolgt dabei durch den Aufruf eines Konstruktor-Schlüsselwortes. In der aktuellen Version von Visual Basic wird die Instanziierung im Speicher und die Erzeugung gemeinsam über das Schlüsselwort `New` durchgeführt.

Sie deklarieren also ein Objekt und geben ihm die Struktur der zu verwendenden Klasse und erzeugen anschließend dieses Objekt.

```
Dim Beispiel As Beispielklasse
Beispiel = New Beispielklasse()
```

Schon haben Sie das Objekt `Beispiel`, das alle Prozeduren und Eigenschaften der `Beispielklasse` beinhaltet.

Der Konstruktor kann dabei auch überladen werden. Wenn Sie eine Anforderung haben, dass bestimmte Eigenschaften bereits bei der Objekterzeugung initialisiert werden müssen oder sollen, dann können Sie dies sehr einfach mit einem Konstruktor machen.

Schauen Sie sich die Definition der Klasse `Person` im Listing 3.8 noch einmal an. Diese Klasse besteht aus einer Eigenschaft `Name`. Wenn jetzt die Anforderung besteht, dass der Name bereits bei der Objekterzeugung mit einem Wert versehen wird, dann können Sie einen Konstruktor definieren, an den der Wert für den Namen übergeben wird.

```
Sub New(ByVal personenName As String)
    _name = personenName
End Sub
```

Die Instanziierung der Personenklasse könnte dann wie folgt aussehen:

```
Dim p As New Person("Christian")
```

Hier wird der Wert »Christian« an den Konstruktor übergeben und damit die `Name`-Eigenschaft mit diesem Wert initialisiert.

Auf diese Art und Weise können Sie sehr einfach Ihr Objekt mit Werten vorinitialisieren.

Wenn Sie in Ihrer Klasse keinen Konstruktor selbst anlegen, wird vom Compiler standardmäßig ein Standardkonstruktor angelegt.

Seit Visual Basic 9 steht eine neue zusätzliche Möglichkeit zur Verfügung, Werte bequem mit Werten zu initialisieren. Wenn Sie eine Klasse mit sehr vielen Eigenschaften haben und Sie wollen eine Möglichkeit haben, diese Werte in jeglicher Kombination vorzuinitialisieren, dann müssten Sie für jede Kombination einen entsprechenden Konstruktor definieren. Der Aufwand hierfür ist enorm. Deswegen bietet Visual Basic nun eine vereinfachte Objektinitialisierung.

Ohne entsprechende Konstruktoren anzulegen, können Sie die Initialwerte der Eigenschaften bei der Objekterzeugung in geschweiften Klammern kombiniert mit dem Schlüsselwort `With` angeben.

```
Dim p As New Person With {.Name = "Christian"}
```

Über diese Syntax können Sie kommagetrennt auch mehrere Eigenschaften in beliebiger Kombination initialisieren.

Anonyme Typen

Im Hinblick auf LINQ und in Kombination mit impliziter Typisierung gibt es seit Visual Basic 9 nun auch die Möglichkeit, anonyme Typen zu definieren.

Auf der Basis von anonymen Typen können komplexe Typen ohne Klassendefinitionen instanziiert und genutzt werden.

Um einen anonymen Typen zu instanziieren, wird der `New`-Operator mit der vereinfachten Objektinitialisierung kombiniert. Alle in geschweiften Klammern angegebenen Werte werden vom Compiler dabei als Eigenschaften interpretiert. Aufgrund der impliziten Typisierung sind anonyme Typen nur lokal definierbar.

```
Dim Buch = New With {.Titel = "ASP.NET Programmer's Choice", .Verlag = "Addison-Wesley"}
Respone.Write(Buch.Titel & Buch.Verlag)
```

Dieser kleine Beispielcode definiert einen komplexen Typ mit zwei Eigenschaften Nummer und Titel.

Für anonyme Typen steht auch wiederum IntelliSense zur Verfügung. Der Compiler erzeugt automatisch Typnamen, die jedoch im Programmcode nicht ansprechbar sind.

Methoden

Methoden stellen die Algorithmen innerhalb einer Klasse dar. Sie sind also die Berechnungsprozeduren und Funktionen einer Klasse, oder einfacher ausgedrückt: Mithilfe von Methoden ermöglichen Sie Ihren Klassen, die Ausführung von Aktionen zu definieren.

Die Deklaration von Methoden ist also die Deklaration von Prozeduren oder Funktionen innerhalb einer Klasse.

Damit können Sie sich an den bei Prozeduren und Funktionen bereits erläuterten Elementen einfach orientieren.

Felder und Eigenschaften

Felder einer Klasse sind die innerhalb dieser Klasse deklarierten Variablen. Über die Zugriffssteuerung, die Sie auch auf die Deklaration von Variablen innerhalb einer Klasse anwenden können (Sie erinnern sich: Tabelle 3.11), können Sie den Zugriff auf die Variablen bestimmen.

Mit Eigenschaften werden Informationen, die Klassen zurückliefern sollen, bezeichnet. Um eine Eigenschaft zu erstellen, müssen Sie diese deklarieren. Sie müssen einen Namen für die Eigenschaft festlegen und einen Datentyp zuweisen, der von der Eigenschaft zurückgegeben werden soll.

```
[Bezeichner] Property Eigenschaftsname As Datentyp
  [Get
    Anweisungen
  End Get]
  [Set
    Anweisungen
  End Set]
End Property
```

Im Prinzip deklarieren Sie also eine neue Funktion mit dem Schlüsselwort `Property`. Sie können Parameterlisten mit übergeben, Sie können die Zugriffssteuerung festlegen, alles, wie Sie es von Prozeduren und Funktionen bereits kennen.

Nun ist der Rahmen der Eigenschaft festgelegt. Hierfür gibt es zwei Anweisungsblöcke in der Eigenschaft. Innerhalb des einen Anweisungsblocks werden Eigenschaftswerte ermittelt:

```
Get
  Anweisungen
End Get
```

Das Setzen der Eigenschaften wird durch den Abschnitt

```
Set
  Anweisungen
End Set
```

vorgenommen. Das Ermitteln von Eigenschaften und das Setzen von Eigenschaften wird hierbei im Code konsequent aufgetrennt. Auf den `Get`-Block wird zugegriffen, wenn Sie innerhalb eines Ausdrucks auf die Eigenschaft zugreifen. Der `Set`-Block wird ausgeführt, wenn Sie der Eigenschaft einen Wert zuweisen wollen.

Falls Sie Eigenschaften so anlegen wollen, dass sie entweder nur gelesen oder nur geschrieben werden können, dann stehen Ihnen hierfür die Schlüsselwörter `ReadOnly` und `WriteOnly` zur Verfügung.

Standardmäßig können Sie allerdings sowohl das Lesen als auch das Schreiben von Werten durchführen.

NEU

Um das Anlegen einer Property möglichst komfortabel zu gestalten, müssen Sie im Codeeditor nur noch das Schlüsselwort `Property` angeben und danach zweimal die TAB-Taste drücken. Es wird Ihnen automatisch das komplette Codegerüst angelegt, und Sie brauchen nur noch die farblich hervorgehobenen Stellen Ihren Bedürfnissen anzupassen.

Mit Visual Studio 10 wurden sogenannte automatisch implementierte Eigenschaften eingeführt.

Die Syntax für eine Property kann nun auch stark verkürzt wie folgt geschrieben werden:

```
Public Property Name As String
```

Wie Sie sehen, müssen Sie nicht mehr die private Variable und den `Get`- und `Set`-Zweig implementieren. Das macht der Compiler für Sie, Sie müssen lediglich darauf achten dass Sie nicht zusätzlich eine Variable `_Name` definieren, denn das ist der Variablenname den bereits der Compiler automatisch anlegt.

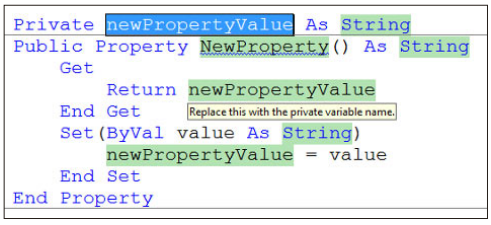

Abbildung 3.22: Eine auf der Basis von Code-Snippets angelegte Property

Entwicklung von Klassen

Die Klasse, die wir in diesem Abschnitt deklarieren, hat eine Methode und einige Eigenschaften, um Ihnen die Entwicklung einer Klasse noch einmal konkret näher zu bringen.

```
<%@ Page Language="VB" %>
<script runat="server">
  Sub Page_Load()
    Dim Erg As Integer
    Dim Zahl As LottoZahl
    Dim Zahl2 As SuperZahl
    Dim Zahl3 As Spiel77
    Zahl = New LottoZahl()
    Zahl2 = New SuperZahl()
    Zahl3 = New Spiel77()
    Erg = Zahl.ZufallsWert()
    Response.Write("Aufruf der Klasse LottoZahl<br>")
    Response.Write("Erg hat den Wert: ")
    Response.Write(Erg)
```

```
    Erg = Zahl2.ZufallsWert()
    Response.Write("<br>Aufruf der Klasse SuperZahl <br>")
    Response.Write("Erg hat den Wert: ")
    Response.Write(Erg)
    Response.Write(" <br> ")
    Erg = Zahl2.ZufallsInt(0, 999999)
    Response.Write("Aufruf der Klasse SuperZahl mit BasisMethode")
    Response.Write("<br>Erg hat den Wert: ")
    Response.Write(Erg)
    Erg = Zahl3.ZufallsWert()
    Response.Write("<br>Aufruf der Klasse Spiel77 <br>")
    Response.Write("Erg hat den Wert: ")
    Response.Write(Erg)
  End Sub

  Public Class ZufallsZahl
    Public Function ZufallsInt(ByVal Unten As Integer, ByVal Oben As Integer) As Integer
      Dim Zufall As Single = 0
      Dim Erg As Single
      Dim Intervall As Integer
      Intervall = Oben - Unten
      Zufall = Rnd()
      Erg = Intervall * Zufall
      ZufallsInt = Convert.ToInt32(Erg) + Unten
    End Function
  End Class

  Public Class LottoZahl
    Inherits ZufallsZahl
    ReadOnly Property ZufallsWert() As Integer
      Get
        Return ZufallsInt()
      End Get
    End Property
    Overloads Function ZufallsInt() As Integer
      Dim Zufall As Single = 0
      Dim Erg As Single
      Zufall = Rnd()
      Erg = 48 * Zufall
      ZufallsInt = Convert.ToInt32(Erg) + 1
    End Function
  End Class

  Public Class SuperZahl
    Inherits ZufallsZahl
    ReadOnly Property ZufallsWert() As Integer
      Get
        Return ZufallsInt()
      End Get
    End Property
    Overloads Function ZufallsInt() As Integer
      Dim Zufall As Single = 0
```

```
      Dim Erg As Single
      Zufall = Rnd()
      Erg = 9 * Zufall
      ZufallsInt = Convert.ToInt32(Erg)
    End Function
  End Class

  Public Class Spiel77
    Inherits ZufallsZahl
    ReadOnly Property ZufallsWert() As Integer
      Get
        Return ZufallsInt(0, 9999999)
      End Get
    End Property
  End Class
</script>
```

Listing 3.18: Eine Klasse mit einer Eigenschaft wird deklariert (Class.aspx).

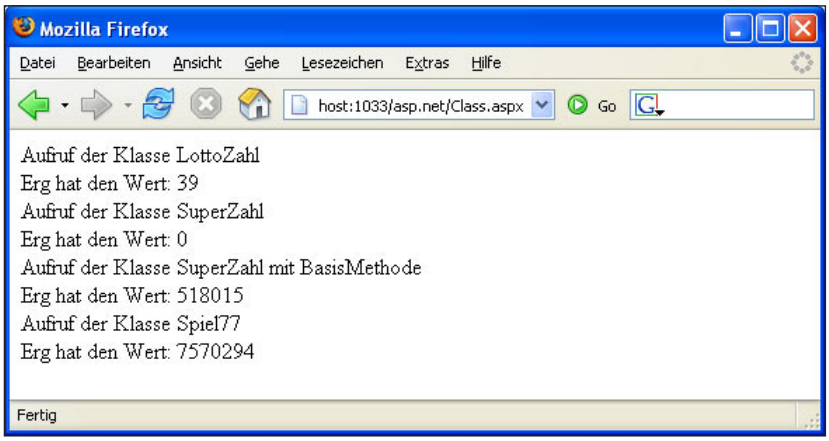

Abbildung 3.23: Eine Klasse mit einer Eigenschaft wird deklariert.

In diesem kleinen Programm ist eine Basisklasse deklariert worden, die eine Methode enthält. Weiterhin haben wir drei Klassen definiert, die aus dieser Basisklasse entstanden sind.

Wir haben in einer Klasse die Basismethode durch Überladung verändert. Wir haben sie in der zweiten Klasse durch Überladen um eine gleichartige Methode mit weniger Parametern ergänzt. In der dritten Klasse haben wir nur eine Eigenschaft hinzugefügt, die ein ganz spezieller Methodenaufruf der Basisklasse ist.

Die Funktionsweise Überladung ist durch den doppelten Aufruf der Methode einmal in der `ReadOnly`-Eigenschaft und zum anderen direkt im erzeugten Objekt demonstriert worden.

3.4.3 Zusammenstellung von Bibliotheken, Einbindung von Namespaces und externen Objekten

Sie haben in Visual Basic die Möglichkeit, Ihre Objekte und Klassenbibliotheken in einzelne Dateien abzuspeichern und auf diese Art für die verschiedenen Projekte nur einzelne Elemente zu verwenden.

Bei der Neuerstellung eines Codeblocks werden automatisch bereits erste Standardelemente für ein Programm vorgegeben.

In einem anderen Programmblock können Sie dann diese Bibliotheken importieren und damit die dort deklarierten Elemente zur Verfügung stellen.

Den Import einer Bibliothek führen Sie über die Verwendung der Compiler-Direktiven durch. Neben der direkten Verwendung im Kommandozeilenmodus können Sie diese Direktiven auch direkt im Visual Web Developer angeben. Auf die Verwendung des Kommandozeilenmodus gehen wir in einem späteren Abschnitt noch einmal gesondert ein.

Eine Bibliothek können Sie außerdem unter dem Menüpunkt WEBSITE und dem Unterpunkt VERWEIS HINZUFÜGEN ergänzen. Sie erhalten ein Untermenü, in dem unter anderem alle .NET-Komponenten aufgelistet sind. Das nachfolgende Bild zeigt hiervon einen kleinen Ausschnitt.

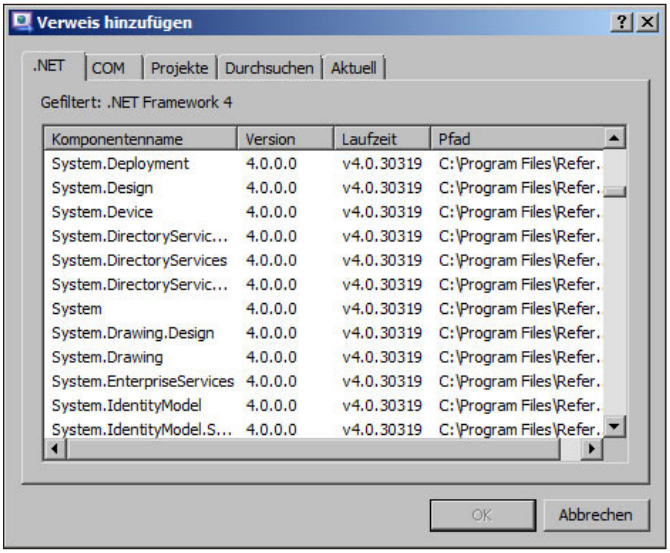

Abbildung 3.24: Ein kleiner Ausschnitt von Verweisen

Jetzt geht es noch darum, wie Sie Klassen und deren Funktionalitäten von zusätzlich referenzierten Bibliotheken nutzen können.

In den Quelltext importieren Sie dann den Namespace mit dem Schlüsselwort `Imports`.

```
Imports Microsoft.Build.Conversion
```

Wenn Sie einen Namespace importiert haben, brauchen Sie nicht mehr zur Verwendung der Klasse den Namespace explizit voranzustellen.

Diese Zeile importiert also genau diesen beschriebenen Namespace zur Verwendung im Quelltext. Danach können Sie im Quelltext die darin deklarierten Klassen mit allen Methoden und Eigenschaften verwenden.

3.5 Basisfunktionen des .NET Frameworks

Es gibt eine ganze Reihe von Elementen von Visual Basic, die mit Einführung von .NET aus der Programmiersprache ausgegliedert wurden und als Teil des .NET Framework bereitgestellt werden. Diesen Elementen wollen wir uns nun nachfolgend eingehender widmen.

Durch die weitere Bereitstellung der verschiedenen Funktionen als Methoden des .NET Frameworks in der wohlbekannten Syntax ist sichergestellt, dass eine Migration alter Programme so einfach wie möglich gehalten wird.

Einige Änderungen haben sich ergeben, auf die wir im dritten Teil dieser Spracheinführung hinweisen wollen. Weiterhin sind viele Funktionen erheblich erweitert worden, sodass Ihnen noch effektivere Werkzeuge zur Verfügung stehen werden.

Die Funktionalitäten, auf die wir nachfolgend eingehen, sind Teil der Klassenbibliothek des .NET Frameworks. Wir haben einige der Module, wo es geeignet erschien, inhaltlich in einem Abschnitt zusammengefasst.

3.5.1 Standardfunktionen und Methoden zur Stringmanipulation

Das Anzeigen, Umwandeln, Ergänzen und Manipulieren von Zeichenketten stellt einen Bereich der Programmierung dar, den Sie bei Ihren Programmen sicher häufig brauchen werden.

Als Erstes möchten wir kurz die Möglichkeiten der Stringmanipulation erläutern. Sie haben die Möglichkeit, Funktionen, die in der Visual Basic-Historie ihren Ursprung haben, zu verwenden, oder Sie können auf die Methoden des .NET Frameworks zurückgreifen.

Wir fassen die vorgestellten Eigenschaften und Methoden abschließend jeweils in einer kleinen Tabelle noch einmal stichpunktartig zusammen. Die folgende Zeichenkette dient als Basis für die jeweiligen Manipulationsziele (der String wird mit je drei Leerzeichen begonnen und abgeschlossen).

```
Dim Bstring As String = "   abc def GHI jkl   "
```

Wenn Sie bestimmte Abschnitte aus einem String herausschneiden wollen, so können Sie die Methode `String.Remove()` verwenden. Als Ergebnis wird jeweils der entsprechend manipulierte String zurückgeliefert. Ein Parameter ist die Anzahl der betroffenen Zeichen. Alternativ geben Sie vor diesem Parameter einen zweiten Parameter an, der das Startzeichen, ab dem die Manipulation beginnen soll, kennzeichnet.

Beispiele:

» `Erg= Bstring.Remove(0, 8)`
liefert als Ergebnis: `"ef GHI jkl   "`

» `Erg= Bstring.Remove(8)`
liefert als Ergebnis: `"   abc d"`

» `Erg= Bstring.Remove(5, 8)`
liefert als Ergebnis: `"   abI jkl"`

Auch das Beschneiden von Strings um Leerzeichen am Anfang und am Ende der Zeichenkette ist sehr hilfreich und wird durch die Methoden `String.Trim()` (schneidet an beiden Enden der Zeichenkette), `String.TrimEnd()` (schneidet am rechten Ende der Zeichenkette) und `String.TrimStart()` (schneidet am linken Ende der Zeichenkette) unterstützt.

Sie entsprechen also `LTrim()`, `RTrim()` und `Trim()` in Visual Basic.

Beispiele:

» `Erg = Bstring.TrimStart()`
liefert als Ergebnis: `"abc def GHI jkl   "`

» `Erg = Bstring.TrimEnd()`
liefert: `"   abc def GHI jkl"`

» `Erg = Bstring.Trim()`
liefert als Ergebnis: `"abc def GHI jkl"`

Wenn Sie Informationen über eine Zeichenkette erhalten wollen, so können Sie beispielsweise mit der Eigenschaft `String.Length` die Länge einer Zeichenkette ermitteln. Die Frage, ob in einer Zeichenkette ein bestimmter String vorkommt, können Sie mit der Methode `String.Contains()` klären (hierbei geben Sie als Parameter den zu suchenden Substring an). Wenn Sie die Positionen der von Ihnen gesuchten Zeichenkette ermitteln wollen, können Sie dies mit der Methode `String.IndexOf()` und, falls Sie die Position rückwärts suchen wollen, mit `String.LastIndexOf()` ermitteln. Hierbei gibt es verschiedene überladene Methoden, mittels derer Sie den Suchbereich (Startposition der Suche, Anzahl der zu durchsuchenden Zeichen im String) festlegen können. Wollen Sie eine Zeichenkette in einem String ersetzen, so steht Ihnen `String.Replace()` zur Verfügung.

Sie kennen nun also auch Alternativen zu `Len`, `InStr()`, `InStrRev()` und `Replace()`.

Beispiele:

» `Dim Erg As Integer = Bstring.Len`
liefert als Ergebnis: `21`

» `Dim Erg As Integer = Bstring.IndexOf(" ")`
liefert als Ergebnis: `1`

» `Dim Erg As Integer = Bstring.LastIndexOf("")`
liefert als Ergebnis: `21`

» `Dim Erg As String = Bstring.Replace _`
`("c def GHI ", " ")`
liefert als Ergebnis den String: `"   ab jkl   "`

Abschließend wollen wir auch noch auf die Möglichkeiten von Umwandlungen in Groß- oder Kleinbuchstaben in einem String oder der Aufspaltung von einem String in mehrere Zeichenketten (bzw. des Zusammenfügens von mehreren Zeichenketten in einen String) mit oder ohne trennendes Kennzeichen eingehen.

Die Buchstaben in einer Zeichenkette lassen sich einfach über `String.ToLower()` in Kleinbuchstaben und `String.ToUpper()` in Großbuchstaben umwandeln. Sonderzeichen und Ziffern sind hiervon übrigens nicht berührt. Sehr umfassende Möglichkeiten zur Stringformatierung stehen Ihnen mit der Methode `String.Format()` zur Verfügung.

Das Auftrennen und das Zusammenfügen von Zeichenketten erfolgt über die Methoden `String.Split()` und `String.Join()`.

Mit den eben beschriebenen Methoden können Sie die Visual Basic-Funktionen `LCase()`, `UCase()`, `Format()`, `Split()` und `Join()` sprachunabhängig verwenden.

```
<%@ Page Language="VB" %>
<script runat="server">
  Sub Page_Load()
    Dim Arg1 As String
    Dim Arg3 As String
    Dim Arg4 As Integer
    Arg1 = "   abc def GHI jkl   "
    Response.Write("Einige Stringmanipulationen<br>")
    Arg3 = Arg1.Trim
    Response.Write("Trim: ")
    Response.Write(Arg1)
    Response.Write("<br>")
    Response.Write(Arg3)
    Arg3 = Arg1.Remove(0, 10)
    Response.Write("<br>Remove(0, 10): ")
    Response.Write(Arg1)
    Response.Write("<br>")
    Response.Write(Arg3)
    Arg3 = Arg1.Remove(10)
    Response.Write("<br>Remove(10): ")
    Response.Write(Arg1)
    Response.Write("<br>")
    Response.Write(Arg3)
    Arg3 = Arg1.Remove(5, 5)
    Response.Write("<br>Remove(5,5): ")
    Response.Write(Arg1)
    Response.Write("<br>")
    Response.Write(Arg3)
    Arg3 = Arg1.ToLower
    Response.Write("<br>ToLower: ")
    Response.Write(Arg1)
    Response.Write("<br>")
    Response.Write(Arg3)
    Arg3 = Arg1.ToUpper
    Response.Write("<br>ToUpper: ")
    Response.Write(Arg1)
    Response.Write("<br>")
    Response.Write(Arg3)
    Arg4 = Arg1.Length
```

```
    Response.Write("<br>Length: ")
    Response.Write(Arg1)
    Response.Write("<br>")
    Response.Write(Arg4)
    Response.Write("<br>")
  End Sub
</script>
```

Listing 3.19: Strings manipulieren (StringManipulation.aspx)

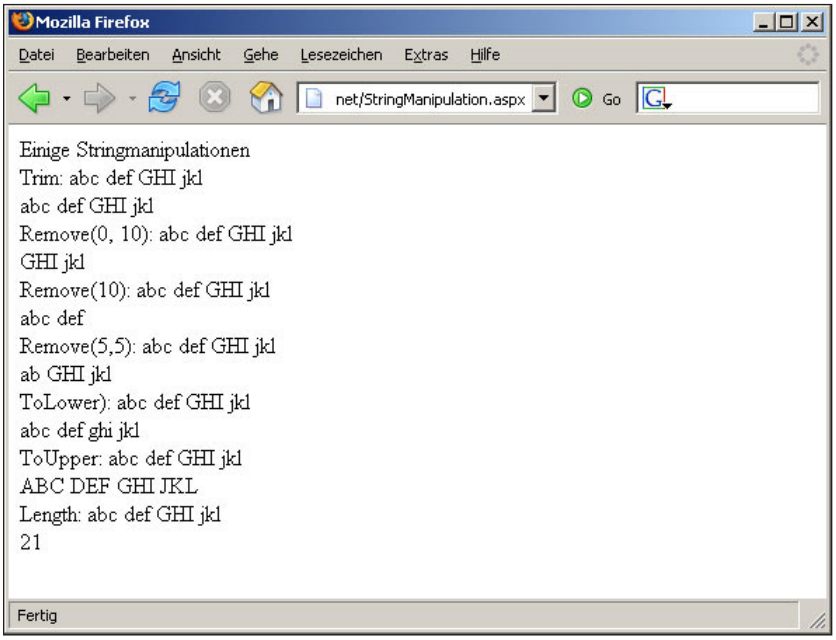

Abbildung 3.25: Strings manipulieren

In anderen Szenarien haben Sie die Möglichkeit, durch die Verwendung zusätzlicher Parameter diese Methoden weitaus umfangreicher nutzen zu können.

3.5.2 Andere nützliche Methoden und Funktionen

Es existiert eine ganze Reihe weiterer nützlicher Methoden in Visual Basic. Diese Methoden sind im Wesentlichen durch die flexibleren Methoden aus dem .NET Namespace ersetzt oder um sie ergänzt worden, sodass Ihnen auch hier wie bereits bei den Stringmanipulationsmethoden beschrieben mehr Möglichkeiten offen stehen.

Generierung von Zufallszahlen

Mit den Funktionen `Randomize()` und `Rnd()` werden Zufallszahlen erzeugt. Zunächst sollten Sie den Zufallszahlengenerator mit `Randomize(`*Zahl*`)` initialisieren. Zufallszahlen werden im Anschluss daran mit `Rnd` erzeugt.

Falls Sie eine identische Reihe von Zufallszahlen ein zweites Mal erzeugen wollen, müssen Sie vor dem Neuinitialisieren des Zufallszahlengenerators eine Zufallszahl mit einem negativen Argument erzeugen und im Anschluss daran für die Initialisierung erneut denselben Startwert für `Randomize()` angeben.

Als sprachunabhängiges Element der Zufallszahlengenerierung ist die Verwendung der `Random`-Klasse die bessere Wahl. Wir haben diese bereits in den Beispielen für die `If ... Then ... Else`-Kontrollstruktur und die `While ... End While`-Schleife verwendet.

Nachfolgend noch ein paar kurze weitere Erläuterungen zu den als Teil der `Random`-Klasse zur Verfügung stehenden Methoden:

Methode	Beschreibung
`Random.Next()`	Es wird eine ganzzahlige positive Zufallszahl zurückgeliefert. Der mögliche Wertebereich ist der Wertebereich eines Int32-Wertes. Sie können einen oder zwei Parameter angeben, ein einzelner Parameter definiert eine obere Grenze und zwei Parameter eine untere und eine obere Grenze für den zulässigen Wertebereich. `Random.Next(4,8)` würde also Zufallszahlen zwischen 4 und 8 zurückliefern.
`Random.NextBytes()`	Mit dieser Methode lassen sich alle Elemente eines Byte-Arrays mit verschiedenen Zufallszahlen belegen.
`Random.NextDouble()`	Mit dieser Methode wird eine Zufallszahl erzeugt, die zwischen 0 und 1 liegt und den Datentyp `Double` hat.

Tabelle 3.14: Methoden zur Erzeugung von Zufallszahlen

Für Beispiele verweisen wir auf die bereits oben erwähnten Programmbeispiele.

Mathematische Methoden

Mathematische Funktionen, die Sie eventuell noch aus alten Visual Basic-Versionen kennen, sind mit der Einführung von .NET nicht mehr existent. Sie wurden durch Methoden aus der statischen Klasse `Math` ersetzt.

Nachfolgend werden wir Ihnen einen kurzen Überblick über diese Klasse geben. Wir werden auf diese Methoden und Eigenschaften auch noch einmal eingehen, wenn wir Ihnen das Handwerkszeug zur Codekonvertierung von alten Visual Basic-Versionen auf die aktuelle .NET-Version geben.

HINWEIS

Wenn Sie die Methoden nicht explizit mit Angabe des Namens der Klasse aufrufen wollen, so müssen Sie zu Beginn des Programmcodes mittels

```
Imports System.Math
```

die statische Klasse Ihrem Programmabschnitt zur Verfügung stellen. Sie können in diesem Fall auf das ansonsten voranzustellende `Math.` verzichten. Wir werden hier einen Mittelweg gehen und in den einführenden Beschreibungen und Erläuterungen das qualifizierende `Math.` verwenden. In Listings werden wir die verwendeten Namespaces durch Importe einfügen und dann auf die vorangestellten Namespace-Bezeichnungen verzichten.

Als grobe Einteilung lassen sich die mathematischen Methoden von den Funktionalitäten her in trigonometrische Methoden und andere (nicht trigonometrische) Methoden einteilen.

Trigonometrische Methoden und allgemeine mathematische Funktionen

Für Operationen der Trigonometrie stellt Visual Basic alle klassischen Funktionen in Form von Methoden bereit. Die Berechnung von Sinus, Cosinus, Tangens und Arcustanges ist mit diesen Methoden kein Problem.

Mit der Methode `Math.Exp()` lässt sich die Exponentialfunktion abbilden, `Math.Log()` ermittelt den natürlichen Logarithmus, `Math.Log10()` den Logarithmus auf der Basis 10 und `Math.Sqrt()` beispielsweise die Quadratwurzel.

Die Funktionalitäten und die Verwendung der Trigonometrie-Methoden und der Methoden für Exponentialfunktion und die anderen allgemeinen Funktionen entnehmen Sie dem nachfolgenden Beispielprogramm.

```
<%@ Page Language="VB" %>
<script runat="server">
  Sub Page_Load()
    Dim Arg As Double
    Dim Erg As Double
    Dim Erg2 As Single
    Dim pi As Double = 3.14159263
    Arg = 1
    Response.Write("Einige trigonometrische ")
    Response.Write("Funktionen: <br> ")

    Erg = Math.Sin(Arg)
    Response.Write("Sinus von ")
    Response.Write(Arg.ToString)
    Response.Write(" ist: ")
    Response.Write(Erg.ToString)
    Response.Write("<br>")

    Erg = Math.Cos(Arg)
    Response.Write("Cosinus von ")
    Response.Write(Arg.ToString)
    Response.Write(" ist: ")
    Response.Write(Erg.ToString)
    Response.Write("<br>")

    Erg = Math.Tan(Arg)
    Response.Write("Tangens von ")
    Response.Write(Arg.ToString)
    Response.Write(" ist: ")
    Response.Write(Erg.ToString)
    Response.Write("<br>")

    Erg = Math.Atan(Arg)
    Response.Write("Arcustangens von ")
    Response.Write(Arg.ToString)
    Response.Write(" ist: ")
    Response.Write(Erg.ToString)
    Response.Write("<br>")
```

```
    Erg = Math.Exp(Arg)
    Response.Write("Exponentialfunktion von ")
    Response.Write(Arg.ToString)
    Response.Write(" ist: ")
    Response.Write(Erg.ToString)
    Response.Write("<br>")

    Arg = 2
    Erg = Math.Sqrt(Arg)
    Erg2 = Math.Sqrt(Arg)
    Response.Write("Quadratwurzel von ")
    Response.Write(Arg.ToString)
    Response.Write(" ist: ")
    Response.Write(Erg.ToString)
    Response.Write(" (Double);")
    Response.Write(Erg2.ToString)
    Response.Write(" (Single)<br>")
  End Sub
</script>
```

Listing 3.20: Trigonometrische und mathematische Funktionen

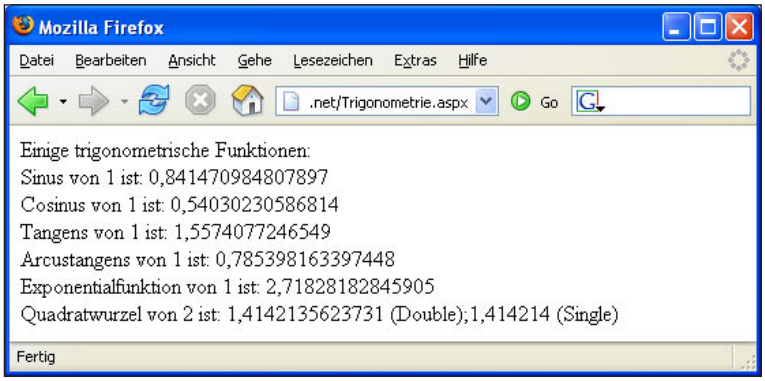

Abbildung 3.26: Trigonometrische und mathematische Funktionen

Anpassungen von Zahlen

Die Methoden `Math.Abs()`, `Math.Sign()` und `Math.Round()` lassen sich am besten damit beschreiben, dass mit ihnen Anpassungen an Zahlen vorgenommen werden können.

Die erste Methode `Math.Abs()` ermittelt den Absolutwert einer Zahl – das bedeutet negative Zahlen werden eliminiert. `Math.Sign()` ermittelt das Vorzeichen einer Zahl. Die dritte Methode `Math.Round()` führt eine Rundung einer reellen Zahl durch. Auch wenn der zurückgegebene Datentyp derselbe ist wie der der ursprünglichen Zahl, werden doch die Nachkommastellen abgeschnitten.

Auch hier werden die Funktionalitäten dieser drei Methoden an einem Beispiel kurz demonstriert.

```
<%@ Page Language="VB" %>
<script runat="server">
  Sub Page_Load()
```

```
    Dim Arg As Double
    Dim Arg2 As Double
    Dim Erg As Double
    Arg = -1.2345
    Arg2 = 1.2345
    Response.Write("Einige mathematische ")
    Response.Write("Funktionen: <br> ")
    Erg = Math.Abs(Arg)
    Response.Write("Abs von ")
    Response.Write(Arg.ToString())
    Response.Write(" ist: ")
    Response.Write(Erg.ToString())
    Response.Write("<br>")
    Erg = Math.Sign(Arg)
    Response.Write("Signum von ")
    Response.Write(Arg.ToString())
    Response.Write(" ist: ")
    Response.Write(Erg.ToString())
    Response.Write("<br>")
    Erg = Math.Sign(Arg2)
    Response.Write("Signum von ")
    Response.Write(Arg2.ToString())
    Response.Write(" ist: ")
    Response.Write(Erg.ToString())
    Response.Write("<br>")
    Erg = Math.Round(Arg)
    Response.Write("Round von ")
    Response.Write(Arg.ToString())
    Response.Write(" ist: ")
    Response.Write(Erg.ToString())
    Response.Write("<br>")
  End Sub
</script>
```

Listing 3.21: Mathematische Methoden (Math.aspx)

Hier die Ausgabe im Browser:

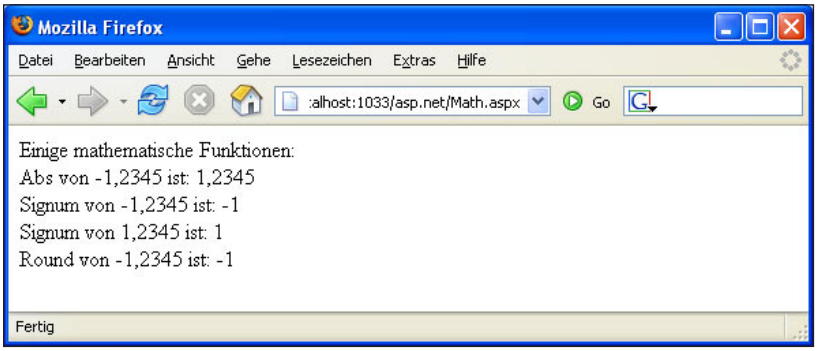

Abbildung 3.27: Mathematische Methoden

Mit der aktuellen Frameworkversion 4.0 gibt es jetzt auch eine Unterstützung von komplexen Zahlen. Genauso wie der neue Datentyp `BigInteger` befindet sich die Klasse `Complex` in der Bibliothek `System.Numerics`.

Um eine komplexe Zahl zu definieren und zu instanzieren gibt es eine eigene Konstruktorüberladung bei der sowohl der reelle und imaginäre Teil der Zahl angegeben wird.

```
Dim c As New Complex(5,2)
```

3.6 Unterschiede zwischen Visual Basic 6 und Visual Basic 10

Im nachfolgenden Abschnitt wollen wir noch einmal explizit auf die Unterschiede zwischen der Version 6 und der aktuellen Version von Visual Basic eingehen, um Ihnen das Umschreiben von Programmcode zu erleichtern. Mit der Einführung von den Visual Basic .NET-Versionen hat Microsoft – wie bereits erwähnt – zum ersten Mal die Abwärtskompatibilität der Programmiersprache nicht weiter verfolgt.

Wenn Sie Ihre Programme auf .NET-Code umschreiben wollen, so können Sie eine ganze Reihe unterschiedlicher Strategien verfolgen. Sie können Ihre Programme dahingehend modifizieren, dass eine Kommunikation zwischen Visual Basic 6 und .NET-Komponenten über den COM Layer gesteuert werden. Auf diese Weise können Sie Teile fürs Erste so belassen, wie sie von Ihnen ursprünglich erstellt wurden, und sich zunächst um die Komponenten kümmern, bei denen Sie von den zusätzlichen Möglichkeiten von .NET am meisten profitieren. Der Nachteil ist, dass Sie sich mit Registrierung und Versionierung von COM-Komponenten herumschlagen müssen.

Wenn Sie ein Upgrade Ihres Programmcodes vornehmen, beachten Sie, dass Sie nicht nur einfach Code verändern werden, sondern dass Sie unter Umständen neue Technologien einführen müssen, die Ihre Arbeiten umfangreicher machen werden, als Sie ursprünglich gedacht haben. Berücksichtigen Sie immer, dass unter Umständen ein vollständiges Neuschreiben Ihrer Programme und Programmabschnitte eine durchaus überlegenswerte Option ist.

Wir werden in den nachfolgenden Abschnitten daher nur auf die Änderungen bei Schlüsselwörtern und wesentliche Umschreibung des Codes eingehen.

Wir verzichten dabei bewusst auf eine Gegenüberstellung des Umgangs mit ActiveX-Elementen im Gegensatz zu .NET-Elementen. Auch Ersatz der COM-Komponenten durch korrespondierende Elemente des .NET Frameworks werden wir in diesem Abschnitt nicht weiter thematisieren.

3.6.1 Das ist neu eingeführt worden

In den .NET-Versionen von Visual Basic ist eine ganze Reihe von neuen Schlüsselwörtern eingeführt worden, die in den älteren Versionen von Visual Basic noch nicht zur Verfügung standen. Diese Schlüsselwörter legen ihren wesentlichen Schwerpunkt auf Themen um die objektorientierte Programmierung. Es werden die unterschiedlichen Schlüsselwörter zur Klassifizierung von Klassenelementen eingeführt wie `Protected`, `Overrides` oder auch `Shadows`.

Der Datentyp `Currency` wurde durch den Datentyp `Decimal` ersetzt.

ACHTUNG

Beachten Sie, dass der Wertebereich für `Integer`-Variablen erheblich erweitert wurde. Dies kann eventuell zu anderem Verhalten bei verschiedenen Programmabschnitten führen, bei denen Sie »alte« `Integer`-Variablen aus Visual Basic 6 verwenden. Das VB6-Integer war ein 16-Bit-Integer, unter .NET handelt es sich um ein 32-Bit-Integer.

Als weitere Schlüsselwörter sind diejenigen zur Fehler- und Ausnahmebehandlung eingeführt worden (`Try`, `Catch`, `Finally`, `Throw`).

3.6.2 Das hat sich verändert

Es haben sich einige Verwendungen von Programmelementen verändert. Auf den anderen Wertebereich von Integer sind wir ja bereits eingegangen.

Im Wesentlichen ist die Auslegung der Sprachelemente strenger, an vielen Stellen jedoch einheitlicher geworden. Funktionen und Methoden müssen nun in jedem Fall die Parameterübergabe mit in Klammern gesetzten Werten durchführen.

Nicht mehr erlaubt ist:

```
Response.Write "Hallo"
```

Diese Zeile muss nun lauten:

```
Response.Write("Hallo")
```

Bei den Datentypen hat es insbesondere bei Arrays Veränderungen gegeben. Wenn Sie ein Array festlegen, dann muss in Visual Basic unter .NET die untere Dimensionsgrenze immer 0 sein. In Visual Basic 6 waren auch noch negative oder auch positive Dimensionsuntergrenzen erlaubt.

Der Datentyp `Variant` wird nicht mehr verwendet.

Auch bei den mathematischen Funktionen (die nun Teil des .NET Frameworks sind) haben sich einige Schreibweisen verändert, die Anpassungen von Code notwendig machen (diese Änderungen werden aber bei der Übersetzung der Programme auch als fehlerhaft gekennzeichnet und sind damit einfach zu identifizieren und zu beseitigen. Hier nun einige dieser Änderungen: Quadratwurzeln werden nicht mehr mit `SQR`, sondern `Sqrt()` geschrieben. Vorzeichen werden nicht mehr mit `Sgn`, sondern mit `Sign()` ermittelt. Der Arcustangens wird nicht mehr mit `Atn`, sondern mit `Atan()` berechnet.

Abschließend noch eine weitere wichtige Änderung, die mit Visual Basic .NET durchgeführt wurden: Bei Parameterübergaben werden die Übergabeparameter nun mit `ByVal` und `ByRef` klassifiziert. Standardmäßig (wenn Sie nicht selber explizit deklarieren) würde ein Parameter mit `ByVal` übergeben, in Visual Basic 6 war es noch `ByRef`.

Die sogenannten benutzerdefinierten Typen aus den alten VB-Versionen werden nun als Strukturen definiert.

Das Schlüsselwort `Set`, das, bei VB6 zur Zuweisung von Objekten verwendet wurde, wurde in den neuen .NET-Versionen abgeschafft. Visual Basic sieht jetzt keinen Syntaxunterschied mehr vor bei der Zuweisung von Referenztypen und Wertetypen.