Visual Basic 2008

Einfach programmieren lernen

GÜNTER BORN BENJAMIN BORN

Markt+Technik

→leicht →klar →sofort

Das können Sie schon

Projekte anlegen 40
Datentypen und Variablen verwenden 53
Felder deklarieren und einsetzen 63
Mit Zuweisungen und Operatoren arbeiten 68

Das lernen Sie neu

Steueranweisungen im Programm 86
Arbeiten mit Schleifen 90
Enumerationen und Sonderfälle bei Schleifen 94
Funktionen und Prozeduren 100
Debuggen von Anwendungen 109

Kapitel 3

Programmlogik und Debugging

In diesem Kapitel lernen Sie, wie sich Programmabläufe in Visual Basic 2008 steuern lassen. Neben Anweisungen zur Verzweigung des Programmablaufs werden auch Schleifen besprochen, mit denen sich Codeabschnitte mehrfach ausführen lassen. Funktionen oder Prozeduren ermöglichen Ihnen, bestimmte Anweisungsfolgen mehrfach zu verwenden. Am Kapitelende erfahren Sie noch, was es im Hinblick auf das Testen (Debuggen) von Programmen zu wissen gibt.

Steueranweisungen im Programm

Steueranweisungen ermöglichen Ihnen, den Ablauf innerhalb eines Programms in Abhängigkeit von bestimmten Bedingungen zu beeinflussen. Visual Basic 2008 stellt Ihnen verschiedene Konstruktionen zur Kontrolle des Programmablaufs zur Verfügung.

If ... Then-Anweisung

Die einfachste Anweisung für eine Verzweigung wird durch das Schlüsselwort *If* eingeleitet und mit dem Schlüsselwort *Then* abgeschlossen.

```
If x < 0 Then x = 0
```

Die *If*-Anweisung ermöglicht das Ausführen von Anweisungen oder Programmteilen in Abhängigkeit von einer vorgegebenen Bedingung. In obigem Beispiel wird der Wert der Variablen *x* in einer Bedingung überprüft. Liefert die Bedingung den Wert *True* (d. h. ist *x* kleiner Ø), ist die hinter dem Schlüsselwort *Then* folgende Anweisung auszuführen. Im konkreten Beispiel wird der Wert der Variablen *x* auf Ø gesetzt.

Um mehr als einen Befehl in Abhängigkeit von einer Bedingung auszuführen, muss eine modifizierte Variante der *If*-Anweisung verwendet werden. Der Block mit den auszuführenden Befehlen wird dabei mit den Schlüsselwörtern *End If* abgeschlossen. Die Struktur einer solchen Anweisung sieht folgendermaßen aus:

```
If x < 0 Then
  x = -1 * x
  y = 1
End If
```

Als Bedingung wird in der *If*-Anweisung wieder der Wert der Variablen *x* ausgewertet. Ist der Wert kleiner Ø, sind die Anweisungen im Codeblock bis zur abschließenden *End If*-Anweisung auszuführen. Im konkreten Fall wird der Betrag der Variablen *x* durch Multiplikation mit –1 bestimmt und die Variable *y* auf 1 gesetzt.

If ... Then ... Else

Die gerade besprochene *If*-Anweisung ermöglicht nur die Abarbeitung eines Codeblocks, wenn die abgefragte Bedingung zutrifft. Häufig wird jedoch ein Mechanismus benötigt, der zusätzlich die Ausführung eines zweiten Codeblocks erfordert, falls die Bedingung nicht erfüllt sein sollte. Dieser Codeblock

wird mit dem Schlüsselwort *Else* eingeleitet und mit dem Schlüsselwort *End If* abgeschlossen. Die *If*-Anweisung besitzt dann folgende Struktur:

```
If x < 0 Then
  x = 0
  y = 1
Else
  y = 0
End If
```

Die obige Codefolge prüft den Wert der Variablen *x* in der *If*-Anweisung ab. Ist der Wert von *x* negativ, soll der erste Codeblock zwischen den Schlüsselwörtern *Then* und *Else* ausgeführt werden. Trifft die Bedingung nicht zu, d. h., ist der Wert von *x* größer gleich o, wird der Codeblock zwischen *Else* und *End If* ausgeführt.

Dieses Wissen soll jetzt an einem kleinen Beispielprogramm praktisch erprobt werden. Den Quellcode sehen Sie hier:

```
Imports System.Console   ' wg. ReadLine etc.
Module Module1
  Sub Main()
   ' Demonstration einer If-Abfrage
   Dim x As Integer
   Dim y As Integer

   Write("Bitte eine Zahl von -3 bis 9 eingeben: ")
   x = CInt(ReadLine())    ' Lese Wert
   WriteLine("")           ' neue Zeile
   If x < 0 Then
    WriteLine("Wert kleiner 0: " & x.ToString())
    x = 0
    y = 1
   Else
    WriteLine("Wert größer gleich 0: " & x.ToString())
    y = 0
   End If
   ReadLine()    ' Warte auf Benutzereingabe
  End Sub
End Module
```

Eine Besonderheit, auf die wir kurz eingehen möchten, findet sich gleich in der ersten Zeile. Dort ist die Anweisung *Imports System.Console* aufgeführt. Was ist der Sinn dieser Anweisung? In den vorherigen Beispielen wurden die Methoden *ReadLine()* und *WriteLine()* mit Anweisungen der Art:

```
System.Console.ReadLine()
System.Console.WriteLine()
```

eingetragen. Mit dieser Information weiß der Visual Basic-Übersetzer genau, in welchem Namensraum und in welcher Klasse er die betreffenden Methoden findet. Da in der Vorlage für Konsolenanwendungen bereits ein Verweis auf den Namensraum *System* enthalten ist, hätte auch die kürzere Schreibweise

```
Console.ReadLine()
Console.WriteLine()
```

funktioniert. Der Nachteil dieser Ansätze besteht in der erforderlichen Schreibarbeit. Diese lässt sich reduzieren, indem die *Imports*-Anweisung am Programmanfang vorgibt, aus welchem Namensraum und welcher Klasse eventuell Methoden zu verwenden sind (siehe auch *Kapitel 4* im Abschnitt »Wozu sind Namensräume erforderlich?«).

Die eigentlichen Programmanweisungen sind recht einfach zu verstehen. Nach der Deklaration der Variablen und einer Benutzerausgabe mit der Aufforderung zur Eingabe eines Werts muss diese eingelesen werden. Hierzu wird die im Code fett ausgezeichnete *ReadLine()*-Methode verwendet. Die als Text anfallende Benutzereingabe wird dann mit der *CInt()*-Methode in einen Integerwert konvertiert und der Variablen *x* zugewiesen.

Im Beispielprogramm wird der vom Benutzer eingelesene Wert der Variablen *x* im nächsten Schritt in einer *If*-Anweisung ausgewertet. Abhängig vom Wert erfolgt dann eine Benutzerausgabe, wie es in den hier sichtbaren Fenstern der Eingabeaufforderung gezeigt wird.

Hinweis

Der Text zur Benutzerinformation suggeriert zwar, dass nur Zahlen zwischen –3 und 9 einzugeben sind. Die Anweisungen im oben gezeigten Programmcode prüfen aber lediglich, dass der Wert kleiner Ø ist. Sie können ja ggf. zur Übung das Beispiel erweitern, sodass die Auswertung nur erfolgt, wenn die Eingabe im Bereich zwischen –3 und 9 liegt. Die Anweisung könnte dann folgendermaßen aussehen:

```
If (x >= -3) AND (x <= 10) Then
…
End if
```

Interessant ist hier der Umstand, dass gleich zwei Abfragen kombiniert wurden. Die beiden Abfragen sind in Klammern eingefasst und der *AND*-Operator verknüpft die Ergebnisse. Nur wenn die erste und die zweite Bedingung wahr sind, wird der *If*-Zweig ausgeführt. Ein *OR*-Operator würde dagegen zwei Bedingungen über »oder« verknüpfen, d. h., der *If*-Zweig wird ausgeführt, sobald eine Bedingung wahr ist. Die in obigem Listing gezeigte zweite *If*-Abfrage samt den Ausgabeanweisungen ließe sich dann innerhalb der hier gezeigten *If*-Abfrage unterbringen. Man spricht dann auch von geschachtelten *If*-Anweisungen, mit denen sich beliebige logische Abfragen durchführen lassen.

If … Then … ElseIf

Reicht Ihnen eine *If … Then … Else*-Anweisungsfolge nicht? Benötigen Sie zusätzliche Abfragen? Über das Schlüsselwort *ElseIf* ist es möglich, mehrere *If*-Blöcke zu schachteln. Hier ein Beispiel für diese Konstruktion:

```
If x < 0 Then
  x = -1 * x
  y = 1
ElseIf x = 100 Then
  x = 0
  y = 10
ElseIf x = 50 Then
  x = x - 10
  y = 5
End If
```

Hier wird die Variable *x* auf verschiedene Bedingungen überprüft. Sollte eine der Bedingungen erfüllt sein, wird der Codeblock innerhalb der Bedingung ausgeführt.

Select Case

Mehrfach geschachtelte *If*-Abfragen werden leicht unübersichtlich. Um eine Variable auf mehrere Werte abzuprüfen und verschiedene Codeblöcke in Abhängigkeit vom Wert der Variablen auszuführen, lässt sich die *Select Case*-Anweisung verwenden. Die folgende Codesequenz demonstriert den Einsatz der *Select Case*-Anweisung:

```
Select Case a
  Case 1
    b = 100
    c = 20
  Case 2
    b = 200
    c = 40
  Case 3, 4, 5
    b = 300
    c = 60
  Case Else
    b = 0
    c = 0
End Select
```

Hinter den Schlüsselwörtern *Select Case* ist die zu überprüfende Variable anzugeben. Die zu überprüfende Bedingung für den Wert der Variablen *a* ist in den nachfolgenden *Case*-Zweigen anzugeben. Zur Laufzeit werden die einzelnen *Case*-Anweisungen nacheinander überprüft. Ist die Bedingung erfüllt, wird der zugehörige Block mit allen Anweisungen bis zur nächsten *Case*-Anweisung ausgeführt. Im konkreten Beispiel wird geprüft, ob die Variable *a* den Wert 1, dann den Wert 2 und schließlich die Werte 3, 4 oder 5 aufweist. Trifft dies zu, wird der jeweilige Codeblock ausgeführt. Sollte keine der angegebenen Bedingungen zutreffen, wird der Code im optionalen *Case Else*-Zweig ausgeführt.

Arbeiten mit Schleifen

Mit Schleifen ist es möglich, verschiedene Codeblöcke mehrfach auszuführen. Neben *For* ... *Next*-Schleifen lassen sich auch Codeblöcke in die beiden Schlüsselwörter *Do* ... *Loop* einschließen und zu einer Schleife kombinieren. In diesem Fall muss die Schleife mit einer Abbruchbedingung versehen werden, die am Schleifenkopf hinter dem Schlüsselwort *Do* stehen, aber auch am Schleifenende hinter dem Schlüsselwort *Loop* folgen kann. Nachfolgend lernen Sie die entsprechenden Schleifenvarianten kennen.

Do While ... Loop-Schleife

Die *Do While*-Anweisung erzeugt eine Schleife, die in Abhängigkeit einer am Schleifenkopf definierten Bedingung ausgeführt wird. Ergibt die Bedingung den Wert *True*, ist die Schleife auszuführen. Andernfalls wird der Code der Schleife übersprungen, weshalb man auch von einer abweisenden Schleife spricht. Hier sehen Sie ein Beispiel für eine solche Schleife:

```
x = 1
Do While x < 10
  x = x + 1
Loop
```

In dieser Schleife wird beim Eintritt die Bedingung $x < 10$ überprüft. Sollte die Bedingung erfüllt sein, tritt der Programmablauf in die Schleife ein. Dabei wird der Code innerhalb der Schleife (zwischen *Do* und *Loop*) so lange ausgeführt, wie die Bedingung erfüllt ist. Trifft die Bedingung beim Eintritt in die Schleife dagegen nicht zu, wird das Programm mit den nach dem Schlüsselwort *Loop* folgenden Anweisungen fortgesetzt.

> **Hinweis**
>
> Achten Sie bei Verwendung einer solchen Schleife darauf, dass die Schleifenbedingung beim Eintritt immer definiert ist. Zudem muss die Bedingung die Werte *False* oder *True* liefern, d. h., es ist sicherzustellen, dass das Abbruchkriterium irgendwann erfüllt wird, damit die Schleife verlassen werden kann.

Do Until ... Loop-Schleife

Die *Do Until*-Anweisung bewirkt ebenfalls eine Überprüfung der Bedingung beim Eintritt in die Schleife. Allerdings wird der Code innerhalb der Schleife nur dann ausgeführt, wenn die Bedingung den Wert *False* liefert. Hier sehen Sie ein Beispiel für eine solche Schleife:

```
x = 1
Do Until x > 10
  x = x + 1
Loop
```

Im konkreten Fall wird die Schleife verlassen, sobald die Variable *x* den Wert 11 erreicht. Dann wird die auf das Schlüsselwort *Loop* folgende Anweisung ausgeführt.

Do ... Loop While-Schleife

Bei der *Do ... Loop While*-Schleife erfolgt die Prüfung der Abbruchbedingung am Schleifenende. Liefert die Prüfung der Bedingung den Wert *True*, wird die Schleife weiter ausgeführt. Der nachfolgende Codeausschnitt zeigt den Einsatz dieser Anweisung:

```
a = 1
Do
  a = a + 1
Loop While a < 10
```

Die Schleife wird also mindestens einmal durchlaufen. Ist am Ende der Schleife der Wert der Variablen *a* kleiner als 10, wird die Schleife und damit die Anweisungen zwischen *Do* und *Loop* ausgeführt. Sobald die Bedingung nicht mehr zutrifft, endet die Schleife und der auf die *Loop*-Anweisung folgende Befehl wird ausgeführt.

Do ... Loop Until-Schleife

Ähnlich wie die *Do ... Loop While*-Schleife ist der Code innerhalb der Schleife mindestens einmal auszuführen, da die Bedingung bei der *Do ... Loop Until*-Schleife erst am Schleifenende überprüft wird. Die Schleife ist dabei so lange auszuführen, wie die Schleifenbedingung den Wert *False* liefert. Sobald das Abbruchkriterium am Schleifenende den Wert *True* liefert, wird das Programm mit der auf *Loop* folgenden Anweisungszeile ausgeführt. Hier sehen Sie ein Beispiel für eine solche Schleife:

```
a = 1
Do
  a = a + 1
Loop Until a >= 10
```

In dem obigen Codeausschnitt wird die Variable so lange in den Schleifendurchläufen erhöht, bis sie den Wert *a = 10* erreicht. Dann terminiert die Schleife.

Tipp

In obiger Schleife würde auch die Bedingung *a = 10* als Abbruchkriterium reichen, da der Wert von 1 bis n hochgezählt wird. Bei komplexeren Programmen ist es aber häufig nicht immer ersichtlich, in welchen Wertebereichen eine zu überprüfende Variable liegt. Aus diesem Grund empfiehlt es sich, die Bedingung immer so zu formulieren, dass das Abbruchkriterium immer sicher erfüllt wird. Mit *a = 10* würde die Schleife z. B. weitergeführt, wenn *a* irgendwann mit dem Wert 11 starten würde. Die Bedingung *a >= 10* bewirkt dagegen einen Abbruch, wenn der Wert *a* entgegen der ursprünglichen Annahme größer oder gleich 10 ist.

For ... Next-Schleife

Bei dieser Schleifenvariante werden die Anweisungen zwischen die Schlüsselwörter *For* ... *Next* als Block eingefügt. Charakteristisch für diese Art der Schleife ist, dass sie eine vordefinierbare Anzahl an Durchläufen besitzt. Der nachfolgende Codeabschnitt zeigt den Aufbau dieser Schleife:

```
For i = 1 To 10
  a = a + 1
Next i
```

Die im Schleifenkopf auf das Schlüsselwort *For* folgende Variable *i* wird als Schleifenindex bezeichnet. Es muss sich um eine Variable handeln, die üblicherweise vom Datentyp *Integer* ist. Es können aber auch andere numerische Datentypen verwendet werden. Vor dem Schlüsselwort *To* steht der Startwert für den Schleifenindex. Der Endwert des Schleifenindex ist nach dem Schlüsselwort *To* anzugeben. Dieser Schleifenindex beginnt in der obigen Sequenz mit dem Wert 1 und wird bei jedem Durchlauf um 1 erhöht. Die Schleife wird abgearbeitet, bis die Variable *i* den Wert 10 erreicht hat, und endet dann.

Möchten Sie eine von 1 abweichende Schrittweite für den Schleifenindex verwenden, lässt sich dies über das optionale Schlüsselwort *Step* realisieren.

```
For i = 1 To 100 Step 10
  a = a + 1
Next i
```

Handelt es sich beim Schleifenindex um eine Gleitkommazahl, können Sie auch die Schrittweite als Gleitkommazahl angeben. Um einen absteigenden Schleifenindex zu erzwingen, muss der Startwert größer sein als der Endwert, und über das Schlüsselwort *Step* lässt sich die Schrittweite (z. B. –1) vorgeben.

Schleifen eignen sich hervorragend, um auf die Elemente von Feldern (Arrays) zuzugreifen. Der folgende kurze Codeabschnitt deklariert den Schleifenindex sowie ein Feld mit zehn Temperaturwerten:

```
Dim i As Integer
Dim Temperatur (0 To 9) As Double
For i = 0 To 9
 Temperatur(i) = 10.0
Next i
```

In der Schleife werden dann alle Feldelemente auf den Wert 10.0 gesetzt.

Enumerationen und Sonderfälle bei Schleifen

Die obigen Informationen zu Schleifen mittels *Do ... Loop* und *For ... Next* stellen quasi das Standardhandwerkszeug für den Visual Basic-Programmierer dar. Es gibt aber einige Spezialfälle, wo sich besondere Konstruktionen von Schleifen eignen. Nachfolgend möchten wir zumindest kurz Aufzählungen (Enumerationen) vorstellen und zeigen, wie sich diese mit Schleifen bearbeiten lassen.

Enumerationen, das steckt dahinter

Enumerationen stellen eine elegante Möglichkeit dar, um benannte Konstanten zu vereinbaren und diese anschließend über ihre Namen anzusprechen. Eine Enumeration umfasst dabei immer eine Gruppe benannter Konstanten, die als ganzzahlige Werte (vom Typ *Byte*, *Short*, *Integer*, *Long*) zu vereinbaren sind. Diese benannten Konstanten der Enumerationsliste stellen die sogenannten Member (Mitglieder) der Enumeration dar.

> **Hinweis**
>
> Der Vorteil einer Enumeration ist, dass sich diese als Parameter an Methoden übergeben und mittels *For*-Schleifen bearbeiten lässt. Wichtig ist, dass Enumerationen nur auf Modul-, Klassen-, Struktur- oder Namespace-Ebene definierbar sind. Sie können die Enumeration also nicht auf der Ebene einer Prozedur anlegen.

Die Anweisung zur Definition einer Enumeration besteht aus den Schlüsselwörtern *Enum* und *End Enum*. Dazwischen werden die Member der Enumeration als Konstanten definiert. Dies wird an einem Beispiel deutlich, wo die

Enumeration *Farbe* mit den Membern *Rot*, *Gruen* und *Blau* definiert werden soll.

```
Enum Farbe
  Rot
  Gruen
  Blau
End Enum
```

Da sowohl der Typ für die Enumeration als auch die Initialisierungswerte fehlen, verwendet Visual Basic den Datentyp *System.Int32* (also *Integer*) und weist den Membern der Enumeration fortlaufend die Wert 0, 1, 2 zu. Um die Member einer Enumeration mit eigenen Werten und einem eigenen Typ zu versehen, ist die folgende Deklaration zu verwenden:

```
Enum Farbe As Integer
  Rot = &HFF0000
  Gruen = &HFF00
  Blau = &HFF
End Enum
```

Die Member der Enumeration sind vom Datentyp *Integer* und weisen einen benutzerdefinierten Wert auf.

> **Hinweis**
>
> Statt der Dezimalschreibweise wurden die Literale im obigen Beispiel in der Hexadezimalschreibweise angegeben (erkennbar an der vorangestellten Zeichenfolge *&H*). Bei Farbangaben hat die Hexadezimalschreibweise den Vorteil, dass die RGB-Farbanteile sehr leicht festlegbar sind (je zwei Hexadezimalziffern für die Farben Rot, Grün und Blau). Der Enumerationsmember Blau erhält dann den Hexadezimalwert &HFFoooo zugewiesen.

Nach der obigen Definition können Sie über den Namen der Enumeration auf die Member zugreifen.

```
Dim Farbe1 As Integer
Dim Farbe2 As Integer
Dim Farbe3 As Integer
Farbe1 = Farbe.Rot
Farbe2 = Farbe.Gruen
Farbe3 = Farbe.Blau
```

Hier wurden drei *Farbe*-Variablen definiert, denen dann die Werte der Enumeration zugewiesen werden. Die Werte der Enumeration *Farbe* lassen sich mit folgender Anweisung auf der Konsole ausgeben:

```
Console.WriteLine("Enumeration Farbe {0}, {1}, {2}", _
  Farbe.Rot, Farbe.Gruen, Farbe.Blau)
```

Mit der folgenden Anweisung werden der Name des Members sowie dessen Wert als Hexadezimalzahl in der Konsole angezeigt:

```
Console.WriteLine("Farbe {0} = &H{1}", Farbe.Rot, _
  Hex(Farbe.Rot))
```

Die Angabe *Farbe.Rot* bewirkt innerhalb der *WriteLine()*-Methode, dass Visual Basic den Member im Platzhalter {0} an die Methode übergibt. Die Ausgabe der *WriteLine*-Anweisung ist in folgender Abbildung als erste Ergebniszeile zu sehen. Offenbar zeigt die Methode *WriteLine()* den Namen des Members an.

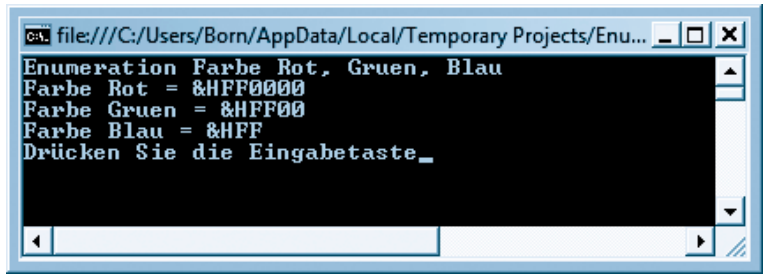
Tipp

Um den Wert des betreffenden Members als Hexadezimalzahl zu erhalten, wird die Visual Basic-Funktion *Hex()* verwendet. Um den Wert ggf. in eine Integerzahl zu wandeln, könnten Sie die Funktion *CInt()* verwenden.

Der nachfolgend gezeigte Quellcode eines einfachen Beispiels verwendet die obigen Kenntnisse, um eine Enumeration *Farbe* zu definieren. Anschließend werden die Werte im Fenster der Eingabeaufforderung ausgegeben.

```
Module Module1
    ' Projekt: Enumeration1
    ' Beispiel für die Verwendung einer Enumeration
    Enum Farbe As Integer
        Rot = &HFF0000
        Gruen = &HFF00
        Blau = &HFF
    End Enum

    Sub Main()
        Dim Farbe1 As Integer
        Dim Farbe2 As Integer
        Dim Farbe3 As Integer
        Farbe1 = Farbe.Rot
        Farbe2 = Farbe.Gruen
        Farbe3 = Farbe.Blau
```

```
        Console.WriteLine( _
            "Enumeration Farbe {0}, {1}, {2}", _
            Farbe.Rot, Farbe.Gruen, Farbe.Blau)
        Console.WriteLine( _
            "Farbe {0} = &H{1}", Farbe.Rot, _
            Hex(Farbe.Rot))
        Console.WriteLine( _
            "Farbe {0} = &H{1}", Farbe.Gruen, _
            Hex(Farbe.Gruen))
        Console.WriteLine( _
            "Farbe {0} = &H{1}", Farbe.Blau, _
             Hex(Farbe.Blau))
        Console.Write("Drücken Sie die Eingabetaste")
        Console.ReadLine()   ' Warte auf Eingabetaste
    End Sub
End Module
```



Die For Each-Schleife

Die auf den vorhergehenden Seiten angesprochene *For ... Next*-Schleife besitzt den Nachteil, dass die Zahl der Iterationsschritte vorab bekannt sein muss. Bei Enumerationen, bei Auflistungen von Objekten (auch als Collections bezeichnet) oder bei Feldern wäre es äußerst hilfreich, wenn es eine Schleife gäbe, die einfach alle Elemente durchläuft. Genau dieses leistet die *For Each*-Schleife. Die Schleife führt eine Iteration über jeden Wert der in der Schleife angegebenen Variablen durch. Enthält die Variable eine Enumeration, werden deren Member in der Schleife abgearbeitet. Bei Feldern durchläuft die Schleife alle Feldelemente und bei Auflistungen werden deren Member bearbeitet. Eine *For Each*-Schleife lässt sich folgendermaßen definieren:

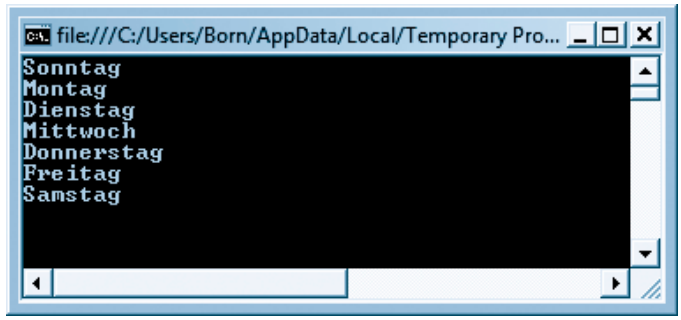
```
For Each tag In wochentage
  Console.WriteLine(tag)
Next
```

Hinter den Schlüsselwörtern *For Each* folgt die Schleifenvariable (hier *tag*), die bei jedem Durchlauf einen Wert aus dem hinter dem Schlüsselwort *In* angegebenen Element (hier *wochentage*) aufnimmt. Wichtig ist daher, dass die Schleifenvariable vom gleichen Datentyp wie das hinter *In* angegebene Element ist.

Nachfolgend finden Sie den kompletten Quellcode eines Beispiels mit einer *For Each*-Schleife. Im Feld *wochentage* werden die Namen der Wochentage abgelegt. Anschließend listet die *WriteLine()*-Anweisung innerhalb der *For Each*-Schleife die in der Schleifenvariable *tag* enthaltenen Werte auf.

```
Module Module1
  Sub Main()
  ' Projekt: Wochentage
  ' Demonstration einer For Each ... Next-Schleife
   Dim tag As String  ' Schleifenindex
   ' Feld mit den Wochentagen
   Dim wochentage As String() = { _
       Sonntag", "Montag", _
       "Dienstag", "Mittwoch", "Donnerstag", _
       "Freitag", "Samstag"}

   ' Schleife über alle Feldelemente
   For Each tag In wochentage
    Console.WriteLine(tag)    ' Wochentag ausgeben
   Next
   Console.ReadLine()
  End Sub
End Module
```

Das hier gezeigte Fenster der Eingabeaufforderung zeigt die Ausgaben des Beispiels mit den Werten des Arrays *wochentage*.



Hinweis

Der Vorteil der *For Each*-Schleife besteht darin, dass diese automatisch die im angegebenen Element enthaltenen Member in der Schleifenvariablen liefert.

Ganz nützlich: Exit und Continue

Bei Schleifen und den nachfolgenden Prozeduren gibt es Fälle, in denen die Bearbeitung unterbrochen werden muss. Visual Basic 2008 bietet für diesen Zweck die *Exit*-Anweisung an.

- Für eine *Do ... Loop*-Schleife ist die Anweisung *Exit Do* innerhalb einer Schleife für die Unterbrechung vorgesehen.
- Bei *For*-Schleifen verwenden Sie die Anweisung *Exit For*, um den Ablauf zu unterbrechen.
- Bei den auf den nachfolgenden Seiten vorgestellten Prozeduren wird für Unterbrechungen die Anweisung *Exit Sub* verwendet.

Das Schlüsselwort *Exit* bewirkt, dass die Abarbeitung der Schleife sofort beendet und der Programmablauf am Ende des Codeblocks (hinter dem Schlüsselwort *Loop* oder *Next* oder *End Sub*) fortgesetzt wird. Die folgenden Anweisungen demonstrieren die Verwendung des *Exit*-Schlüsselworts:

```
Do
  If wert < 0 Then
    Exit Do
  Else
    summe = summe + 1
  End If
  i = i + 2
Loop While i <= 10
```

Die hier gezeigte *Do ... Loop*-Schleife endet standardmäßig erst, sobald die Variable *i* den Wert 12 annimmt. Weist die Variable *wert* aber einen negativen Wert auf, erzwingt die im *If*-Zweig enthaltene *Exit Do*-Anweisung das Beenden der Schleife. Dann wird die auf *Loop While* folgende Anweisung ausgeführt.

Möchten Sie die Schleife beim Eintreten einer bestimmten Bedingung nicht beenden, sondern nur die Ausführung eines Codeblocks einmalig überspringen? Dann lässt sich der *Continue*-Befehl verwenden. Hier sehen Sie einen Codeabschnitt, der diesen *Continue*-Befehl in einer *For*-Schleife benutzt:

```
Dim i As Integer
```

```
Dim b As Integer = 1
For i = 1 To 100 Step 10
  b = b + i
  If b > 30 Then Continue For
  Console.WriteLine(i & ":" & b)
Next i
```

Die obige Schleife wird in Zehnerschritten durchlaufen, wobei der Wert des Schleifenindex sowie der Inhalt der Variablen *b* im Fenster der Eingabeaufforderung anzuzeigen sind. Sobald jedoch der Wert der Variablen *b* größer als 30 wird, soll die Anzeige der Ergebnisse in der Eingabeaufforderung unterbleiben. Die im *If*-Block enthaltene *Continue*-Anweisung löst diese Aufgabe. Sobald die Bedingung der *If*-Anweisung erfüllt ist, wird die *Continue For*-Anweisung ausgeführt. Diese *Continue*-Anweisung bewirkt, dass der Programmablauf an der *Next*-Anweisung fortgesetzt wird. Es werden also alle auf *Continue* bis zur *Next*-Anweisung folgenden Befehle übersprungen. Die Schleife wird aber trotzdem die vorgesehene Anzahl an Durchläufen erreichen.

Funktionen und Prozeduren

Bisher haben Sie nur Beispiele kennen gelernt, welche lediglich linearen Programmcode enthalten. Um Codeteile häufiger zu verwenden, bietet sich die Verlagerung in Funktionen oder Prozeduren an. Eine Funktion oder Prozedur kann beliebig oft aus dem Hauptprogramm aufgerufen werden. Dann wird der in der Funktion oder Prozedur enthaltene Code ausgeführt und anschließend die Programmausführung im aufrufenden Modul fortgesetzt. Der Unterschied zwischen Prozeduren und Funktionen besteht drin, dass eine Funktion einen Wert an den aufrufenden Befehl zurückliefern kann.

Arbeiten mit Funktionen

Verwenden Sie Funktionen, falls ein Wert als Ergebnis an das aufrufende Programm zurückgegeben und direkt in einer Zuweisung verwendet werden soll. Eine Funktion wird mit den folgenden Anweisungen definiert:

```
Function name (Parameter) As Typ
  …
End Function
```

Das Schlüsselwort *Function* leitet die Definition ein und muss von einem Funktionsnamen gefolgt werden. Dieser Funktionsname dient später im aufrufenden Programm zur Identifikation der gewünschten Funktion (z. B. *Addiere()*) und sollte daher aussagekräftig sein. Die der Funktion zu übergebenden Para-

meter werden in Klammern hinter dem Funktionsnamen aufgeführt. Anschließend können noch das Schlüsselwort *As* sowie die Angabe für den Datentyp der Funktion folgen. Der Codeblock einer Funktion ist mit den Schlüsselwörtern *End Function* zu beenden. Die innerhalb des mit *Function* ... *End Function* begrenzten Codeblocks stehenden Anweisungen werden beim Aufruf der Funktion abgearbeitet. Eine Funktion lässt sich direkt in einer Zuweisung in der Art

```
x = Funktionsname (a, b)
```

verwenden. Das von der Funktion zurückgelieferte Ergebnis wird einer Variablen zugewiesen.

Hinweis

Fehlt eine Typangabe für den Rückgabewert einer Funktion, wird standardmäßig der Typ *Object* verwendet. Vor dem Schlüsselwort *Function* kann (ähnlich wie bei Variablen) noch ein Schlüsselwort wie *Public*, *Private* stehen. Mit dem Schlüsselwort *Private* erreichen Sie, dass nur innerhalb des umgebenden Moduls auf diese Funktion zugegriffen werden kann. Das Schlüsselwort *Public* bewirkt dagegen, dass die Funktion auch von außerhalb des Codeblocks aufgerufen werden darf. Diese Mechanismen werden bei der in *Kapitel 4* besprochenen objektorientierten Programmierung verwendet.

Falls Sie statt mit Modulen mit Klassen arbeiten (siehe das nächste Kapitel), muss der Funktionsdeklaration das Schlüsselwort *Shared* vorangestellt werden. Nur dann ist die Funktion ohne Instanziierung abrufbar.

Schauen wir uns in einem einfachen Beispiel den Aufbau einer Funktion sowie deren Verwendung in einem Programm an. Im kaufmännischen Bereich ist häufiger der Bruttobetrag aus dem Nettobetrag und der Mehrwertsteuer zu ermitteln. Diese Rechenvorschrift ließe sich als Funktion mit dem Namen *Brutto* realisieren. Dann braucht im Programm nur noch die Funktion *Brutto* mit dem Nettowert und der Mehrwertsteuer als Parameter aufgerufen zu werden. Die Funktion gibt den berechneten Bruttowert zurück. Der Quellcode zur Definition der Funktion könnte zum Beispiel so aussehen:

```
Function Brutto(Netto As Decimal, _
  MwSt As Decimal) As Decimal
  Return Netto * (1.0 + MwSt/100.0)
End Function
```

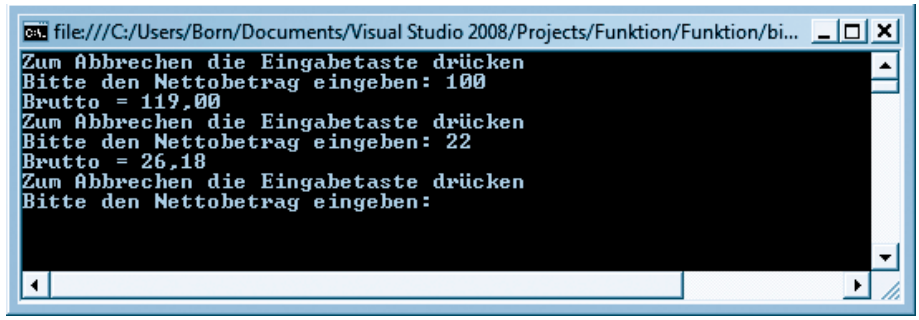
Der Parameter *Netto* ist vom Datentyp *Decimal* und enthält den Nettowert. Der Mehrwertsteuersatz in Prozent wird ebenfalls als Datentyp *Decimal* übergeben. Die Funktion gibt selbst ebenfalls einen Wert vom Typ *Decimal* zurück. Innerhalb der Funktion findet sich nur eine Anweisung, die den Bruttobetrag ermittelt und das Ergebnis über das Schlüsselwort *Return* zurückgibt.

Kommen wir jetzt zu dem Programmteil, in dem die so definierte Funktion verwendet wird. Im aktuellen Beispiel wird vom Benutzer die Eingabe einer Zahl erwartet. Diese Eingabe wird zuerst auf Plausibilität überprüft. Dann ist der Bruttobetrag aus der Eingabe und dem Mehrwertsteuersatz zu errechnen. Abschließend soll das Ergebnis im Fenster der Eingabeaufforderung angezeigt werden. Das Programm liest dabei so lange Benutzereingaben ein, bis die ↵-Taste ohne Eingabe eines Nettowerts gedrückt wird. Wir haben mit dem Beispiel also quasi einen sehr einfachen Mehrwertsteuerrechner programmiert. Nachfolgend sehen Sie den Quellcode des gesamten Beispiels:

```
Module Module1
 ' Projekt: Funktion
 ' Demonstriert den Einsatz einer Funktion
 ' zur Mehrwertsteuerberechnung

  Function Brutto( _
   Netto As Decimal, _
   MwSt As Decimal) As Decimal
      Return CDec(Netto * (1.0 + MwSt / 100.0))
  End Function

  Sub Main()
   Const UmsSt As Double = 19.0  ' Mehrwertsteuersatz
   Dim tmp As String             ' Benutzereingabe
   Dim Netto As Decimal
   Dim Preis As Decimal

   Do     ' Endlosschleife
    Console.WriteLine( _
     "Zum Abbrechen die Eingabetaste drücken")
    Console.Write( _
     "Bitte den Nettobetrag eingeben: ")
    tmp = Console.ReadLine() ' Benutzereingabe
    If tmp = "" Then Exit Do ' Eingabeschleife beenden
    If IsNumeric(tmp) Then   ' eine Zahl?
     Netto = CDec(tmp)       ' String in Dezimalzahl
     Preis = Brutto(Netto, UmsSt) ' Bruttopreis
     Console.WriteLine("Brutto = {0:###0.00}", _
         Preis) ' Preis ausgeben
```

```
      Else          ' Fehlermeldung
       Console.WriteLine(" *** Fehleingabe ***")
      End If
       Loop Until False
  End Sub
End Module
```

Nachfolgend sehen Sie die Ausgaben des Programms im Fenster der Eingabeaufforderung.



Hier noch einige Anmerkungen, wie das Beispiel implementiert wurde. Um mehrere Benutzereingaben zu verarbeiten, wurde eine Endlosschleife mit folgenden Anweisungen verwendet.

```
Do
 WriteLine("Zum Abbrechen die Eingabetaste drücken")
 Write ("Bitte den Nettobetrag eingeben: ")
 tmp = ReadLine()
 If tmp = "" Then Exit Do    ' Eingabeschleife beenden
...
Loop Until False
```

Die Schleife endet nie, da am Schleifenende die *Until*-Bedingung fest auf den Wert *False* gesetzt ist. Über einen Aufruf der *Write()*-Methode wird der Benutzer zur Eingabe eines Werts aufgefordert. Die *ReadLine()*-Methode liest dann die mit ↵ abgeschlossene Benutzereingabe ein. Diese Eingabe wird der Hilfsvariablen *tmp* zugewiesen, da die Methode nur Zeichenketten liefert. Zudem lässt sich über die Stringvariable *tmp* prüfen, ob der Benutzer die ↵-Taste ohne weitere Eingabe gedrückt hat. In diesem Fall wird eine leere Zeichenkette "" zurückgegeben. Die *If*-Anweisung prüft, ob die Variable *tmp* eine leere Zeichenkette enthält. Trifft dies zu, bewirkt die *Exit Do*-Anweisung ein Verlassen der Schleife (dies ist das Abbruchkriterium).

Die eingelesenen Zeichen müssen anschließend in einen Wert vom Typ *Decimal* konvertiert werden. Dies kann über die Visual Basic-Funktion *CDec()* erfolgen. Allerdings sollte der Fall abgefangen werden, dass der Benutzer keine Zahl eintippt, sondern eine beliebige Zeichenkette (z. B. 123Hallo).

> **Hinweis**
>
> Eine Prüfung auf gültige numerische Werte ist mittels der Funktion *IsNumeric()* aus der Visual Basic-Kompatibilitätsklasse möglich. Diese liefert den Wert *True* zurück, falls die eingegebenen Zeichen eine gültige Zahl ergeben.

Die folgenden Zeilen verifizieren daher die Eingabe und wandeln die Eingabezeichen in eine Zahl um:

```
If IsNumeric(tmp) Then       ' eine Zahl?
  Netto = CDec(tmp)          ' String in Dezimalzahl
 ...
Else
 ...
End If
```

Der Aufruf der Funktion zur Berechnung des Bruttobetrags reduziert sich anschließend auf eine Zeile:

```
Preis = Brutto (Netto, UmsSt)
```

Die Anweisung enthält auf der linken Seite die Variable *Preis* zur Aufnahme des Ergebnisses. Auf der rechten Seite steht der Funktionsname *Brutto*. Die zu übergebenden Parameter werden in Klammern hinter dem Namen angegeben. Nach Ausführung dieser Anweisung liegt der Bruttowert in der Variablen *Preis* vor und kann mittels der *WriteLine()*-Methode ausgegeben werden.

```
Console.WriteLine ("Brutto = {0:####.00}", Preis)
```

Auch hier sollte verhindert werden, dass die Methode bei der Ausgabe nicht signifikante Dezimalziffern abschneidet. Daher wurde erneut eine Formatschablone im Platzhalter {0} verwendet.

Hinweis

Die Formatschablone besteht aus den Zeichen hinter dem Doppelpunkt und kann verschiedene Zeichen enthalten. Ein Punkt steht für den Dezimalpunkt im Ausgabewert. Eine Ø signalisiert eine Ziffer, die immer ausgegeben werden muss. Das Zeichen # steht für eine Ziffer, die optional ausgegeben werden kann.

So verwenden Sie Prozeduren

Eine andere Möglichkeit zum Wiederverwenden von Programmcode besteht darin, diesen in eine Prozedur auszulagern. Der Unterschied zwischen Prozeduren und Funktionen besteht darin, dass Prozeduren keinen Rückgabewert besitzen und folgendermaßen zu definieren sind:

```
Sub Name(Parameter)
  …
End Sub
```

Die Definition der Prozedur wird durch das Schlüsselwort *Sub* eingeleitet und mit dem Schlüsselwort *End Sub* beendet. Diese beiden Begriffe klammern den Code der Prozedur. An das Schlüsselwort *Sub* müssen sich der Prozedurname sowie die in Klammern stehende Parameterliste anschließen.

Hinweis

Sie können Prozeduren nur innerhalb eines Moduls oder einer Klasse definieren. Bei Klassen ist dabei das Schlüsselwort *Shared* am Zeilenanfang anzugeben, um die Prozedur ohne Instanziierung aufrufen zu können. Weiterhin können Sie Schlüsselwörter wie *Public* verwenden, um die Gültigkeit der Prozedur über die Klassengrenze festzulegen.

Nun kann das vorherige Beispiel so abgeändert werden, dass die Berechnung der Bruttowerte und die Ausgabe des Ergebnisses in einer Prozedur erfolgen. Die Prozedur mit dem Namen *Anzeige* erwartet als Parameter einen Textwert und den Mehrwertsteuersatz als Dezimalwert. Hier sehen Sie den Code für die Beispielprozedur.

```
Sub Anzeige (txt As String, MwSt As Decimal)
  Dim Netto, Preis As Decimal

  If IsNumeric(txt) Then      ' eine Zahl?
```

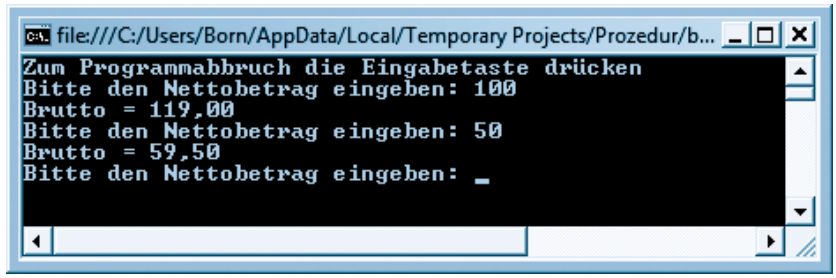
```
    Netto = CDec(txt)          ' String in Dezimalzahl
    Preis = Brutto (Netto, MwSt)
    Console.WriteLine ("Brutto = {0:###0.00}", Preis)
  Else
    Console.WriteLine (" *** Fehleingabe ***")
  End If
End Sub
```

Die Prozedur kann nun aus dem Programm folgendermaßen aufgerufen werden:

```
Anzeige(tmp, MwSt)
```

Der Aufruf einer Prozedur unterscheidet sich also nicht wesentlich vom Aufruf einer Funktion. Prozeduren liefern lediglich keinen Wert für eine Zuweisung zurück. Hier sehen Sie den Quellcode des gesamten Beispiels sowie die Ausgabe der Ergebnisse im Fenster der Eingabeaufforderung:

```
Module Module1
  Sub Anzeige(txt As String, MwSt As Decimal)
  ' Hilfsprozedur für die Anzeige der Ergebnisse
   Dim Netto, Preis As Decimal

   If IsNumeric(txt) Then     ' eine Zahl?
       Netto = CDec(txt)      ' String in Dezimalzahl
       Preis = Netto * (1.0 + MwSt / 100.0)
       Console.WriteLine("Brutto = {0:###0.00}", _
              Preis)
   Else
       Console.WriteLine(" *** Fehleingabe ***")
   End If
  End Sub

  Sub Main()
   ' Demonstration der Verwendung einer Prozedur
   Const MwSt As Decimal = 19.0   ' Mehrwertsteuer
   Dim tmp As String              ' Hilfsvariable

   Console.WriteLine( _
   "Zum Programmabbruch die Eingabetaste drücken")
   Do                  ' Wiederholen bis Leereingabe
    Console.Write("Bitte den Nettobetrag eingeben: ")
    tmp = Console.ReadLine()
    If tmp = "" Then Exit Do ' Eingabeschleife beenden
     Anzeige(tmp, MwSt)       ' Prozeduraufruf
   Loop Until False
  End Sub
End Module
```



Parameterübergabe als Referenzen

Beim Aufruf von Funktionen und Prozeduren können Sie einen oder mehrere Parameter übergeben. Die Parameter sind durch Kommata zu trennen und in Klammern einzufassen. Aus Sicht der Prozedur oder Funktion stellen die übergebenen Parameter so etwas wie Variablen dar. Sie können also innerhalb der Funktion bzw. Prozedur auf diese Variablen zugreifen. Dabei ist aber relevant, was passiert, wenn der Inhalt dieser Parameter innerhalb der Funktion oder Prozedur geändert wird. Hier lassen sich zwei Varianten unterscheiden:

- Parameter lassen sich als Call-by-Value übergeben. Der Funktion oder Prozedur wird als Parameter eine Kopie der ursprünglichen Variablen übergeben. Änderungen am Wert eines Parameters wirken sich dann nur lokal innerhalb der Funktion bzw. Prozedur aus, das aufrufende Programm bemerkt von der Änderung nichts.
- Die Alternative besteht darin, die Parameter über Call-by-Reference zu übergeben. Dabei wird der Funktion oder Prozedur ein Verweis (Reference) auf die Variable des aufrufenden Programms übergeben. Wird der Wert eines Parameters innerhalb der Prozedur oder Funktion geändert, wirkt sich dies über den Verweis auf die Variable im aufrufenden Programm aus.

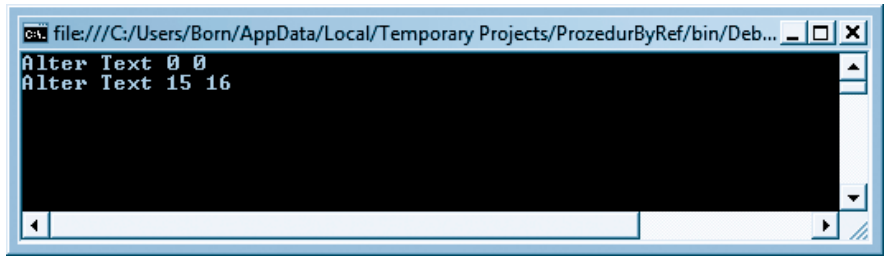
Ob ein Parameter als Call-by-Value oder als Call-by-Reference übergeben wird, lässt sich bei der Definition der Übergabeparameter vorgeben. Bei Call-by-Value muss der zu übergebende Parameter mit dem Schlüsselwort *ByVal* gekennzeichnet werden. Das Schlüsselwort *ByRef* gibt an, dass die Variable als Reference übergeben wird. Ist nichts spezifiziert, verwendet Visual Basic Call-by-Value.

Der Quellcode des folgenden Beispiels demonstriert diesen Ansatz. Der Prozedur *Test* werden ein Text als Call-by-Value und zwei Dezimalwerte als Call-by-Reference übergeben. Die Werte werden im Hauptprogramm mit Vorgabewerten initialisiert und anschließend in der Prozedur verändert. Vor und nach dem

Aufruf der Prozedur listet das Hauptprogramm die Werte der Variablen im Fenster der Eingabeaufforderung auf.

```
Module Module1
 Sub Test(ByVal txt As String, _
    ByRef Netto As Decimal, _
    ByRef Preis As Decimal)
  ' Ändere die Werte

    txt = "Neuer Text"
    Netto = 15.0
    Preis = 16.0
 End Sub

 Sub Main()
  Dim Text As String = "Alter Text"
  Dim Wert1 As Decimal = 0.0
  Dim Wert2 As Decimal = 0.0

  Console.WriteLine(Text & " " & _
            Wert1.ToString() & " " & _
            Wert2.ToString())

  Test(Text, Wert1, Wert2)

  Console.WriteLine(Text & " " & _
            Wert1.ToString() & " " & _
            Wert2.ToString())
  Console.ReadLine()
 End Sub
End Module
```

Die nachfolgende Abbildung zeigt die Ausgabe der Werte vor und nach dem Aufruf der Prozedur *Test()* im Fenster der Eingabeaufforderung. Während der mit Call-by-Value übergebene Text »Alter Text« im Hauptprogramm nicht verändert wird, wirken sich die Wertzuweisungen in der Prozedur auch auf das aufrufende Programm aus.



Optionale Parameter übergeben

Bei der Deklaration von Funktionen und Prozeduren legen Sie auch die Zahl der Argumente fest. Ist die Zahl der Argumente variabel, d. h. sind einige Argumente optional, müssen Sie dies bei der Deklaration berücksichtigen. Optionale Parameter werden durch das Schlüsselwort *Optional* bei der Deklaration der Funktion oder Prozedur gekennzeichnet. Zusätzlich zum Schlüsselwort *Optional* müssen Sie noch einen Standardwert für den optionalen Parameter angeben. Dieser stellt sicher, dass der Parameter definiert ist. Der nachfolgend gezeigten Funktion *Brutto* ist als optionaler Parameter der Mehrwertsteuersatz zu übergeben. Die Deklaration der Funktion sieht dann folgendermaßen aus:

```
Shared Function Brutto(ByVal Netto As Decimal, _
  ByVal Optional MwSt As Decimal = 19) As Decimal
```

Die Angabe *Optional MwSt As Decimal = 19* signalisiert dem Visual Basic-Compiler, dass *MwSt* ein optionaler Parameter und mit dem Standardwert 19 zu initialisieren ist.

> **Achtung**
>
> Optionale Parameter für Funktionen bzw. Prozeduren dürfen immer nur am Ende der Parameterliste deklariert werden. Eine weitere Einschränkung besteht darin, dass nur *ByVal*-Parameter als »optional« gekennzeichnet werden können. Das Schlüsselwort *Shared* ist lediglich erforderlich, wenn die Funktion in einer Klasse verwendet wird.

Die Funktion *Brutto* kann nun in folgenden Varianten aufgerufen werden:

```
Brutto(netto)
Brutto(netto, 19)
```

Beim ersten Aufruf wird der optionale Parameter weggelassen und der Compiler setzt den Standardwert 19 zum Berechnen ein.

Debuggen von Anwendungen

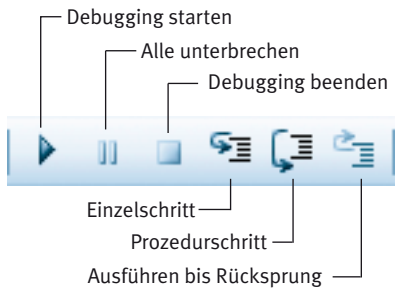
Ein besonderer Punkt bei der Programmentwicklung ist die Fehlersuche und -behebung. Selten wird ein Programm auf Anhieb fehlerfrei arbeiten. Der Übersetzer (Compiler) zeigt Ihnen zwar syntaktische Fehler wie falsch geschriebene Schlüsselwörter an. Sind Optionen wie *Option Explicit On* oder *Option Strict On* gesetzt, werden auch nicht deklarierte Variablen oder nicht passende

Datentypen bei Zuweisungen bemängelt. Aber logische Fehler innerhalb eines Programms (z. B. eine Schleife wird nicht 9-mal wie beabsichtigt, sondern 10-mal ausgeführt) findet der Compiler nicht. Solche Fehler lassen sich nur aufdecken, indem Sie das Programm testen. Dieser auch als Debuggen (kommt von »entwanzen«) bezeichnete Vorgang wird direkt in der Entwicklungsumgebung unterstützt. Mit den nachfolgend erläuterten Funktionen können Sie den Programmablauf direkt im Quellcode verfolgen und auch den Inhalt von Variablen ansehen oder sogar ändern.

Arbeiten mit dem Debugger

Visual Basic 2008 Express Edition enthält (wie auch Visual Studio) einen integrierten Debugger, über den Sie den Programmablauf verfolgen und beeinflussen können. Die Entwicklungsumgebung stellt über Schaltflächen der Symbolleiste, über das Menü *Debuggen* oder über Funktionstasten verschiedene Möglichkeiten bereit, um das Programm unter Kontrolle des Debuggers auszuführen.

> **Tipp**
>
> Zeigen Sie auf Schaltflächen der Symbolleiste, blendet die Entwicklungsumgebung eine QuickInfo mit dem Namen des betreffenden Befehls ein.



- Klicken Sie in der *Standard*-Symbolleiste der Entwicklungsumgebung die Schaltfläche *Debugging starten* an, wird das Programm übersetzt und ausgeführt. Alternativ können Sie im Menü *Debuggen* den Befehl *Debugging starten* wählen oder die Funktionstaste F5 drücken, um das Programm unter der Kontrolle des Debuggers auszuführen.

- Ist der Programmablauf unterbrochen, wird die Schaltfläche bzw. der Menübefehl *Debugging starten* in *Weiter* geändert. Sie können dann den Programmablauf über diese Schaltfläche bzw. den Befehl sowie durch Drücken der Funktionstaste F5 fortsetzen.
- Den Programmablauf können Sie unterbrechen, indem Sie in der Symbolleiste der Entwicklungsumgebung die Schaltfläche *Alle unterbrechen* anklicken. Zudem finden Sie den gleichnamigen Befehl im Menü *Debuggen* oder Sie drücken die Tastenkombination Strg+Unterbr.
- Über die Schaltfläche *Einzelschritt* der Symbolleiste, über den gleichnamigen Befehl im Menü *Debuggen* sowie über die Funktionstaste F8 veranlassen Sie, dass die nächste Programmanweisung unter der Kontrolle des Debuggers ausgeführt wird. Anschließend unterbricht der Debugger den Programmablauf wieder.
- Möchten Sie die Anweisungen einer Prozedur oder einer Funktion komplett abarbeiten lassen, verwenden Sie den Befehl *Prozedurschritt* des Menüs *Debuggen*. Oder Sie klicken auf die gleichnamige Schaltfläche der Symbolleiste bzw. drücken die Tastenkombination ⇧+F8. Dann wird die jeweils nächste Anweisung wie beim *Einzelschritt* ausgeführt. Handelt es sich bei der Anweisung um einen Funktions- oder Prozeduraufruf, wird der Programmcode der aufgerufenen Funktion/Prozedur komplett abgearbeitet. Der Debugger stoppt den Ablauf erst nach dem Abarbeiten der Funktion/Prozedur an der auf den Aufruf folgenden Anweisung.
- Wurde der Programmablauf innerhalb einer Funktion oder Prozedur angehalten, können Sie den Befehl *Ausführen bis Rücksprung* im Menü *Debuggen* oder die gleichnamige Schaltfläche der Symbolleiste wählen. Dann wird der Programmcode der Funktion bzw. Prozedur bis zum Ende durchlaufen. Der Debugger stoppt, sobald die Funktion/Prozedur beendet und die nächste Anweisung im aufrufenden Programm erreicht wird. Die Tastenkombination Strg+⇧+F8 besitzt die gleiche Wirkung.

Achtung

Falls Sie den Debugger mit Tastenkombinationen steuern möchten, achten Sie darauf, dass das Fenster der Entwicklungsumgebung den Fokus besitzt, bevor Sie die Tasten drücken. Andernfalls leitet Windows die Tastencodes an das Programmfenster der getesteten Anwendung weiter. Daher ist es in vielen Fällen einfacher, die Schaltflächen der Symbolleiste zur Debugger-Steuerung zu verwenden.

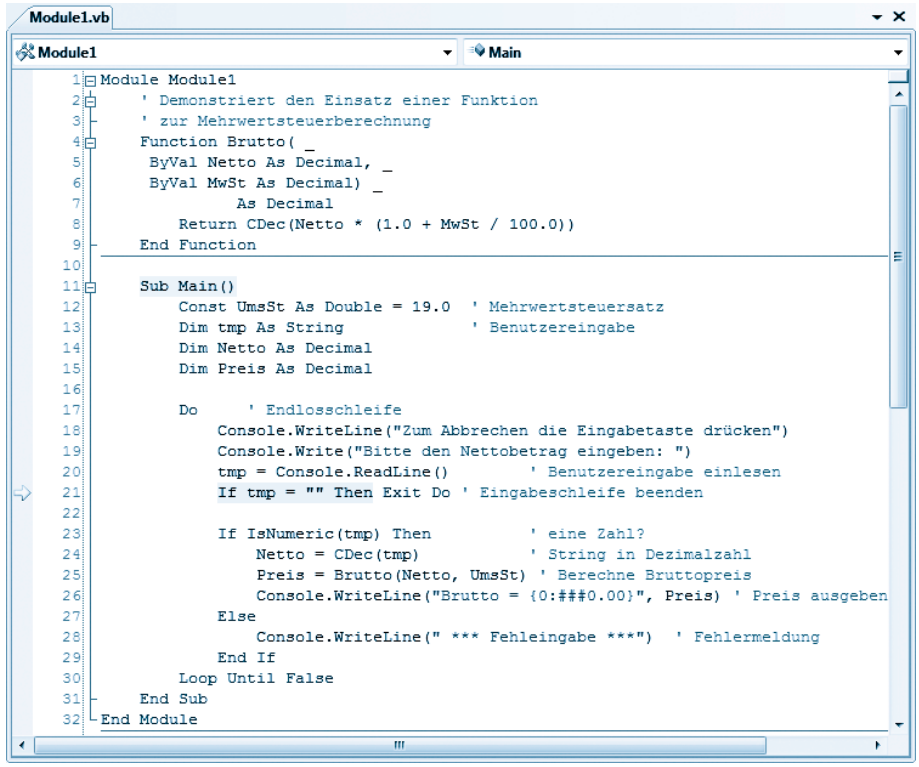
Der Debugger kennzeichnet die nächste auszuführende Anweisung im Codefenster durch eine gelbe Hinterlegung. Gleichzeitig markiert ein gelber Pfeil am linken Rand des Codefensters (der graue Rand wird auch als Kennzeichenleiste bezeichnet) die betreffende Anweisung. Wird der Programmablauf mit der Funktion *Alle unterbrechen* gestoppt, kennzeichnet die Entwicklungsumgebung die aktuell auszuführende Programmanweisung im Quellcode mit einem grünen Hintergrund.

Tipp

Sie können den gelben Pfeil, der die nächste auszuführende Anweisung markiert, per Maus vertikal in der Kennzeichenleiste des Quellcodefensters nach oben oder unten ziehen. Auf diese Weise lässt sich eine andere Anweisungszeile als nächste auszuführende Anweisung markieren. Bei der Fortsetzung des Programms beginnt der Programmablauf bei dieser Anweisung.

Die gleiche Wirkung erreichen Sie, indem Sie die gewünschte Anweisungszeile mit der rechten Maustaste anklicken und den Kontextmenübefehl *Nächste Anweisung festlegen* wählen. Sie sollten aber beachten, dass durch dieses Überspringen von Befehlen unerwartete Ergebnisse auftreten können (z. B. Variablen weisen ggf. falsche Werte auf).

Weiterhin können Sie eine Anweisungszeile mit der rechten Maustaste anklicken und dann im Kontextmenü den Befehl *Ausführen bis Cursor* wählen. Dann setzt der Debugger die Programmausführung so lange fort, bis die betreffende Zeile erreicht wird.



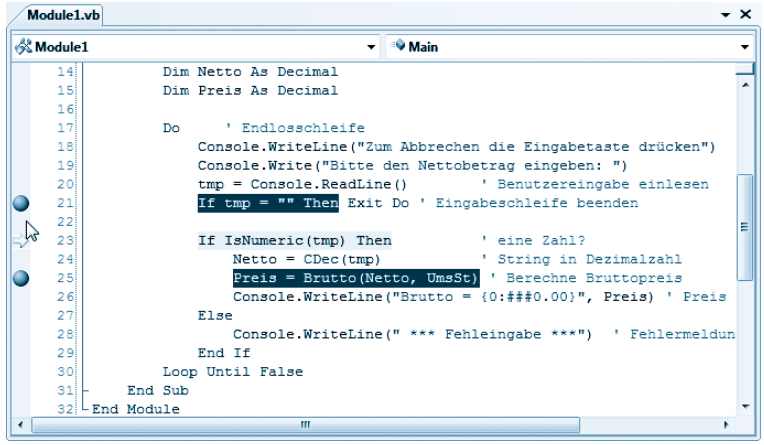
Das Debuggen wird beendet, sobald Sie das zu debuggende Programm schließen. Weiterhin können Sie das Programm zwangsweise beenden, indem Sie in der Entwicklungsumgebung die Schaltfläche *Debuggen beenden* der Symbolleiste oder den gleichnamigen Befehl des Menüs *Debuggen* anwählen. Die Tastenkombination Strg+Alt+Break besitzt die gleiche Wirkung. Nach dem Beenden des Debug-Modus können Sie den Quellcode erneut im Fenster der Entwicklungsumgebung bearbeiten.

Hinweis

Tritt beim Ausführen einer Anwendung ein Laufzeitfehler auf, kann der Benutzer den Debugger der Entwicklungsumgebung starten (sofern die Entwicklungsumgebung auf dem Zielsystem installiert ist).

Arbeiten mit Haltepunkten

Die schrittweise Kontrolle des Programmablaufs ist bei umfangreicheren Programmen etwas aufwändig. Der Programmablauf lässt sich aber über sogenannte Haltepunkte gezielt an bestimmten Programmstellen unterbrechen. Um Haltepunkte an den betreffenden Anweisungszeilen im Codefenster des Debuggers zu setzen, reicht es, die betreffenden Stellen in der Kennzeichenleiste anzuklicken.



Eine mit einem Haltepunkt versehene Zeile wird braun unterlegt und in der Kennzeichenleiste erscheint ein roter Punkt. Sie können mehrere Haltepunkte pro Modul bzw. Klasse setzen. Der Debugger unterbricht den Programmablauf, sobald ein Haltepunkt erreicht wird. Zum Löschen eines Haltepunkts klicken Sie einfach auf dessen roten Punkt in der Kennzeichenleiste.

Hinweis

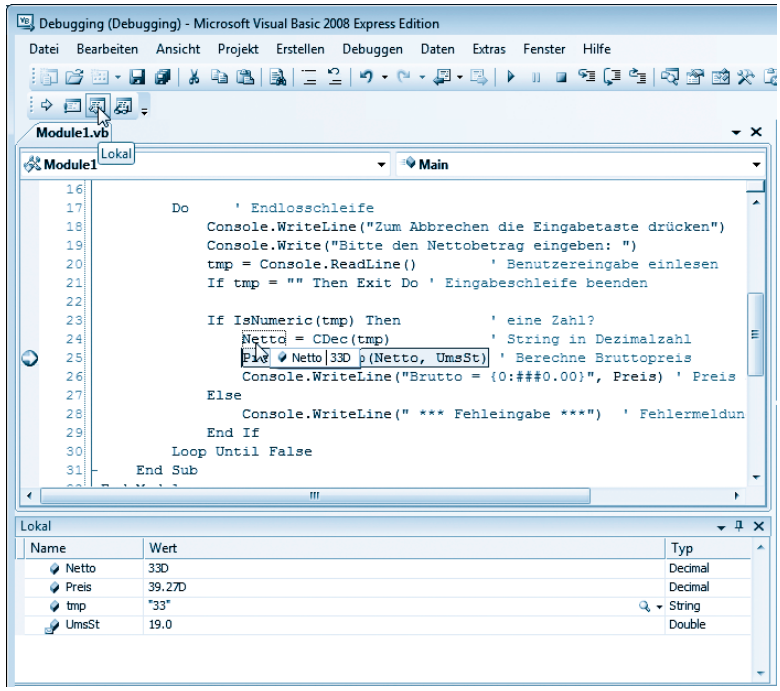
Alternativ können Sie die Befehle im Menü *Debuggen* oder im Kontextmenü verwenden, um einzelne Haltepunkte oder alle Haltepunkte zu löschen. Zudem gibt es im Kontextmenü einen Befehl, um Haltepunkte lediglich zu deaktivieren.

In den nachfolgenden Kapiteln lernen Sie Ereignisprozeduren kennen. Beachten Sie beim Testen des Programmcodes solcher Ereignisprozeduren, dass Haltepunkte nur dann einen Abbruch bewirken, wenn das Ereignis auch auftritt (z. B. Anklicken einer Schaltfläche).

Werte anzeigen und überwachen

Läuft ein Programm unter der Kontrolle des Debuggers ab, können Sie die Werte von Variablen abfragen und automatisch anzeigen lassen. Stellen Sie zuerst sicher, dass der Programmablauf im Debugger angehalten wurde.

- Zeigen Sie im Codefenster mit der Maus auf den Namen einer Variablen oder Eigenschaft, wird deren aktueller Wert als QuickInfo eingeblendet. Dies funktioniert aber nur, falls die Variable bereits mit einem Wert initialisiert wurde.



- Über die Schaltfläche *Lokal* der *Debuggen*-Symbolleiste können Sie das Teilfenster *Lokal* am unteren Rand der Entwicklungsumgebung einblenden. Dort listet der Debugger die Namen, die Werte und die Typen der lokalen Variablen und Konstanten auf.

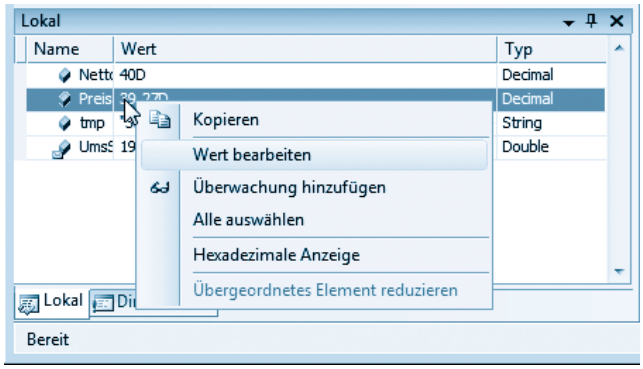
Auf diese Weise können Sie die Werte von Konstanten und Variablen während des Debuggens sehr einfach im Auge behalten.

Werte ändern und überwachen

Stellen Sie beim Debuggen fest, dass eine Variable einen falschen Wert aufweist, und möchten Sie diesen während des Debuggens versuchsweise ändern?

1 Stellen Sie sicher, dass der Programmablauf angehalten wurde, und klicken Sie im *Lokal*-Fenster die gewünschte Variable mit der rechten Maustaste an.

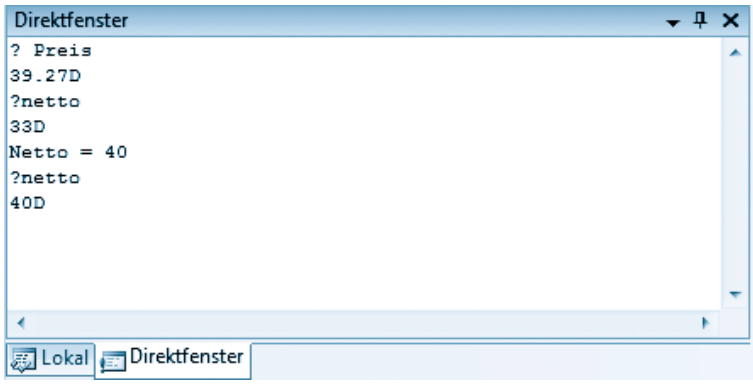
2 Wählen Sie den Kontextmenübefehl *Wert bearbeiten* und überschreiben Sie anschließend den in der Spalte *Wert* hervorgehobenen Wert.



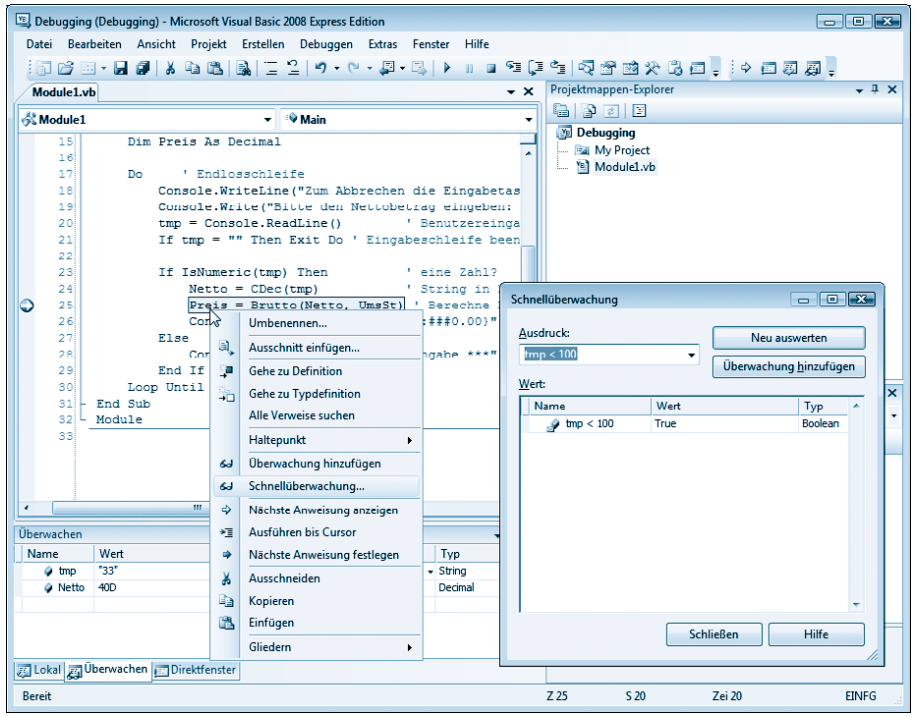
Die Änderung wird wirksam, sobald Sie sie durch Drücken der ↵-Taste übernehmen.

Die zweite Möglichkeit zum Eingreifen stellt das Direktfenster bereit. Sie müssen das Fenster lediglich über die Schaltfläche *Direkt* der Symbolleiste (siehe *Kapitel 1*) bzw. über den Menübefehl *Debuggen/Fenster/Direkt* oder durch Drücken der Tastenkombination Strg+G im Fenster der Entwicklungsumgebung einblenden. Das Direktfenster stellt so etwas wie die Konsole für die Entwicklungsumgebung dar. Sie können Befehle eintippen und durch Drücken der ↵-Taste ausführen lassen.

- Tippen Sie ein Fragezeichen, gefolgt vom Namen einer Variablen oder Konstante, in das Fenster ein und drücken die ↵-Taste, gibt der Debugger den Wert des betreffenden Felds im Direktfenster aus. Ein an den Wert angehängter Buchstabe »D« signalisiert, dass das Ergebnis eine Dezimalzahl darstellt.
- Tippen Sie den Namen einer Variablen, gefolgt von einem Gleichheitszeichen, ein, können Sie anschließend einen Wert angeben. Sobald Sie die ↵-Taste drücken, wird dieser Wert der Variablen zugewiesen.



Auf die gleiche Weise können Sie sogar Methoden aufrufen oder den Wert von Eigenschaften abfragen. Zudem steht Ihnen noch ein Überwachungsfenster zur Verfügung, in dem Sie zu überwachende Variablen aufnehmen können.



- Um eine Variable lediglich zu überwachen, klicken Sie diese im Codefenster mit der rechten Maustaste an und wählen im Kontextmenü den Befehl *Überwachung hinzufügen*. Dann wird der Variablenname sofort in die Überwachungsliste aufgenommen.
- Der Kontextmenübefehl *Schnellüberwachung* öffnet das gleichnamige Dialogfeld, in dem Sie in das Feld *Ausdruck* auch einen Ausdruck der Art »tmp < 100« eintippen und mittels der Schaltfläche *Überwachung hinzufügen* im Fenster *Überwachen* aufnehmen können. Die Schaltfläche *Neu berechnen* zeigt den aktuellen Wert des Ausdrucks im Dialogfeld an. Schließen Sie danach das Dialogfeld über die am unteren Rand eingeblendete Schaltfläche *Schließen*.

Der Debugger blendet das Fenster *Überwachen* ein, sobald Sie einen Eintrag definieren. Alternativ können Sie das Fenster über den Menübefehl *Debuggen/Fenster/Überwachen* anzeigen lassen. Der Befehl ist aber nur verfügbar, wenn ein Programm im Debug-Modus ausgeführt wird. Beim Ablauf des Programms unter der Kontrolle des Debuggers zeigt das Direktfenster die Werte der über-

wachten Variablen bzw. Ausdrücke an. Geänderte Werte werden mit roter Schrift hervorgehoben.

> **Tipp**
>
> Zum Ändern eines Ausdrucks wählen Sie diesen per Doppelklick im Überwachungsfenster an. Danach lässt sich der Ausdruck direkt korrigieren. Löschen können Sie einen Überwachungsausdruck, indem Sie diesen im Überwachungsfenster mit der rechten Maustaste anklicken und im Kontextmenü den Befehl *Überwachung löschen* wählen. Der Kontextmenübefehl *Alle löschen* entfernt dagegen alle Einträge aus dem Überwachungsfenster.
>
> Sie können auch markierte Ausdrücke vom Codefenster direkt in das Überwachungsfenster ziehen, um deren Werte aufzunehmen. Es muss sich aber um vollständige Ausdrücke handeln.

Aufrufliste ansehen

Beim Testen von Programmen, die Funktionen, Prozeduren oder Methoden aufrufen, ist es unter Umständen wichtig, die Hierarchie der Aufrufe zu verfolgen. Auch dies ist im Debugger möglich, indem Sie das Fenster *Aufrufliste* über den Menübefehl *Debuggen/Fenster/Aufrufliste* oder durch Drücken der Tastenkombination Strg+L einblenden. Der Befehl *Aufrufliste* ist aber nur verfügbar, wenn ein Programm im Debug-Modus ausgeführt wird.

Im Fenster werden die aufgerufenen Methoden, Funktionen und Prozeduren aufgeführt. Ein gelber Pfeil kennzeichnet das aktuell ausgeführte Element.

Hinweis

Jeder Eintrag der Liste besteht aus dem Namen der Programmdatei, gefolgt vom Namen der Klasse und der aufgerufenen Prozedur bzw. Methode. In der Spalte *Sprache* wird zudem die Sprache des Quellcodes angegeben. Ob Zeilennummern etc. mit in der Liste erscheinen, können Sie über das Kontextmenü festlegen. Klicken Sie die Zeile im Fenster *Aufrufliste* mit der rechten Maustaste an und wählen Sie im Kontextmenü die aufgelisteten Befehle an. Ein markierter Befehl (z. B. *Modulnamen anzeigen*) bewirkt, dass die betreffende Information im Fenster eingeblendet wird.

Ist das Fenster *Aufrufliste* verdeckt, können Sie es über den gleichnamigen Registerreiter in den Vordergrund holen. Fehlt der Registerreiter (z. B. weil das Fenster vom Benutzer ausgeblendet wurde), können Sie das Fenster über den Menübefehl *Debuggen/Fenster/Aufrufliste* erneut einblenden.

Kleine Erfolgskontrolle

Auf den vorherigen Seiten haben Sie Schleifen, Verzweigungen und Prozeduren kennengelernt. Zudem wissen Sie, wie sich Programme mit dem Debugger testen lassen. Zur Überprüfung Ihrer bisherigen Kenntnisse sollten Sie die folgenden Fragen beantworten.

- **Welchen Zweck hat die If-Anweisung?**

  (Über diese Anweisung können Sie Bedingungen auswerten. Der zwischen *If* … *Then* und *End If* stehende Programmcode wird nur ausgeführt, wenn die abgefragte Bedingung zutrifft.)

- **Was versteckt sich hinter einer *For*-Schleife?**

  (Die *For*-Schleife ermöglicht die wiederholte Ausführung eines Codeblocks, wobei der Start- und Endwert für den Schleifenindex *i* die Zahl der Iterationen bestimmen.)

- **Welche *Do … Loop*-Schleifenvarianten gibt es?**

  (Do While … Loop, Do Until … Loop, Do … Loop While, Do … Loop Until.)

- **Wozu wird eine *For Each*-Schleife gebraucht?**

  (Diese Schleifenvariante hat Vorteile, wenn alle Elemente eines Arrays, einer Enumeration oder einer Auflistung abgearbeitet werden.)

- **Was ist der Unterschied zwischen Funktionen und Prozeduren?**

  (Funktionen geben einen Ergebniswert zurück, während dies bei Prozeduren nicht der Fall ist. Sollen sich Änderungen der Werte in einer Prozedur auf das aufrufende Programm auswirken, müssen diese als Call-by-Reference-Parameter übergeben werden.)

- **Was ist der Unterschied zwischen Call-by-Reference und Call-by-Value?**

  (Mit Call-by-Value übergebene Parameter können in der Prozedur geändert werden, ohne dass sich dies im aufrufenden Programm auswirkt. Eine Call-by-Reference-Parameterübergabe reicht dagegen die Adresse der Variablen aus dem aufrufenden Programm an die Prozedur weiter. Dadurch erhält das aufrufende Programm Zugriff auf die Ergebniswerte der Prozedur.)

- **Wie können Sie Fehler in einem Programm aufspüren?**

  (Syntaktische Fehler werden bei der Übersetzung in der Fehlerliste gemeldet. Logische Fehler lassen sich finden, indem das Programm unter der Kontrolle des Debuggers ausgeführt und das Verhalten der Programmanweisungen analysiert bzw. beobachtet wird.)