8 Auf zum Klassentreffen!

216 Und ab in den Sandkasten
216 Konsolenanwendung in VB.NET
218 Das Klassenprinzip am einfachsten Beispiel
220 Das Klassenprinzip am eigenen Beispiel
221 Statische und nicht-statische Methoden und Variablen
223 Nicht verwirren lassen: Static und Shared in VB
225 Smarttags im Editor von Visual Basic
227 Kleiner Exkurs – womit startet ein Programm?
227 Mit Sub New bestimmen, was beim Instanzieren passiert – der Klassenkonstruktor
230 Überflüssige Funktionen mit dem Obsolete-Attribut markieren
231 Überladen von Funktionen und Konstruktoren
238 Zusätzliche Werkzeuge für .NET
241 Statische Konstruktoren und Variablen
246 Eigenschaften
256 Zugriffsmodifizierer von Klassen, Prozeduren, Eigenschaften und Variablen

Stellen Sie sich vor, Sie könnten dem Dilemma des Eingangsbeispiels aus dem letzten Kapitel entgehen, indem Sie eine Einheit verwenden könnten, die in sich völlig geschlossen ist, aber dennoch völlig erweiterbar bleibt. Und zu verstehen, wie und wieso das mit Klassen funktionieren kann, lassen Sie mich versuchen, das Konzept von Klassen mithilfe einer Analogie zu beschreiben.

Und ab in den Sandkasten

Versetzen Sie sich in Ihre Kindheit zurück. Und zwar so weit zurück, dass Sie im Sandkasten sitzen und mit Förmchen und im Matsch spielen. Sie werden es nicht glauben, aber genau zu diesem Zeitpunkt haben Sie bereits das Klassenkonzept angewendet. Ein Förmchen ist im Grunde genommen nämlich nichts anderes als eine Klasse. Das Förmchen gibt vor, wie Objekte ausschauen sollen, die aus ihm entstehen werden, aber es selbst ist noch kein Objekt, sondern nur eine Vorlage. Wenn Sie aus einer Klasse (einem Förmchen) einen Sandkuchen (ein Objekt) machen wollen, dann müssen Sie die Klasse in ein Objekt instanzieren. Um bei der Analogie zu bleiben: Sie instanzieren einen Sandkuchen aus einem Förmchen, indem Sie nassen Sand in die Form hineingeben, das ganze Ding umdrehen und die Form abziehen.

Klassen in der objektorientierten Programmierung beinhalten Daten *und* Programmcode. Was bei dieser Analogie sind aber die Daten und was der Programmcode? Die Daten sind das, was Sie in das Förmchen hinein geben. Das kann nasser Sand, nasse Muttererde, Lehm, Ton oder Ähnliches sein. Je nachdem, wie Sie die Daten (den Inhalt, das Substrat) verändern, variieren ihre Ergebnisse in Form der instanzierten Objekte. Ihr instanziertes Objekt wird heller oder dunkler, wird fest, wenn es von der Sonne getrocknet wird, oder es zerbröselt nach ein paar Minuten. Doch alle Objekte, die Sie aus Ihrem Förmchen generieren, halten sich an bestimmte Rahmenbedingungen. Diese Rahmenbedingungen könnten Sie mit dem Programmcode einer Klasse vergleichen. Die Silhouette des Förmchens gibt buchstäblich die Rahmenbedingungen vor. Wenn das Förmchen in Sternform »programmiert« wurde, ergeben sich aus ihm auch sternförmige Sandkuchen, ist es in Form eines Schmetterlings »programmiert«, bekommen Sie eben Sandkuchenobjekte in Zitronenfalterform heraus.

Der Programmcode innerhalb einer Klasse regelt, welche Daten die Objekte, die aus ihr hervorgehen, später speichern können und reglementiert auch den Zugriff auf die Daten.

Mit diesem Wissen können Sie Programmkonzepte von Grund auf ändern und mit .NET Klassen schaffen, die auf der einen Seite die Daten für Problemlösungen speichern und auf der anderen Seite Code zur Verfügung stellt, um die Daten zu verwalten und den Zugriff auf diese zu reglementieren.

HINWEIS: Bei den Beispielprogrammen der folgenden Abschnitte handelt es sich nicht um Windows-, sondern um so genannte Konsolenanwendungen – um Anwendungen also, die sich ausschließlich unter der Windows-Eingabeaufforderung verwenden lassen.

Konsolenanwendung in VB.NET

Im Gegensatz zu Visual Basic 6.0 können Sie in Visual Basic .NET auch ohne größeren Aufwand Konsolenanwendungen entwerfen. Das sind Programme, die über keine grafische Oberfläche verfügen, sondern nur unter der Windows-Eingabeaufforderung laufen und mit dem Anwender über reine Textein- und -ausgabe kommunizieren. Konsolenanwendungen sind für Programme sehr gut geeignet, die bei der reinen Stapelverarbeitung eingesetzt werden sollen und wenig oder überhaupt keine Kommunikation mit dem Anwender erfordern. Aber gerade auch beim Debuggen – zum Beispiel, um ohne großen Aufwand neue Typen (Klassen, Strukturen) zu testen – leisten sie sehr gute Dienste.

Möchten Sie selber eine neue Kommandozeilenanwendung erstellen, wählen Sie in der Visual Studio-IDE aus dem Menü *Datei* den Menüpunkt *Neu/Projekt* und im Dialog, den Visual Studio anschließend zeigt, aus dem Zweig *Visual Basic* und dem Bereich *Windows* die Vorlage *Konsolenanwendung*.

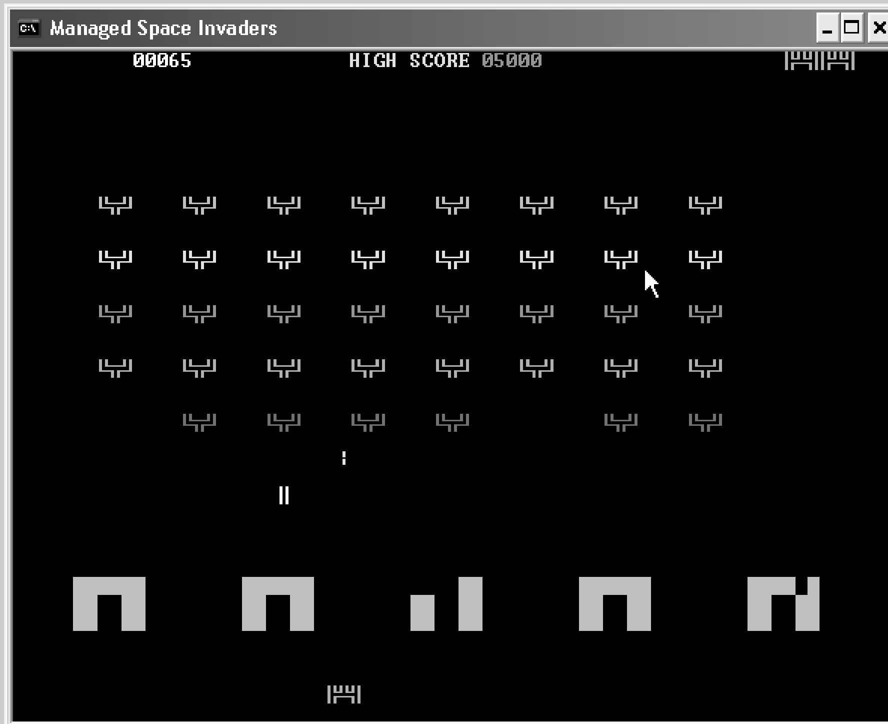

Abbildung 8.1: Kid George vom BCL-Team demonstriert die Möglichkeiten von Konsolenanwendungen eindrucksvoll mit seiner Implementierung von Space Invaders![1]

Seit Visual Studio 2005, bzw. dem Framework 2.0 gibt es übrigens einige Erweiterungen in der Kommandozeilenunterstützung aus .NET heraus. So haben Sie direkt über die `Console`-Klasse Einfluss auf die Farbgebung, auf die direkte Positionierung der jeweils nächsten Ausgabe und Sie können sogar ganze Bildschirmbereiche mit einfachen Befehlen verschieben, und damit Scrolling in jede Richtung oder sogar einfache bewegte Bildschirmgrafiken realisieren. Ein Blick in die Hilfe zum `Console`-Objekt lässt Sie hier Interessantes erfahren, und auch die Space Invaders-Demo, die Sie über den IntelliLink *D0802* herunterladen können, ist schön anzuschauen – auch wenn der Quellcode in C# und nicht in Visual Basic vorliegt.

---

[1] Wenn Sie den Quellcode dieser Anwendungen herunterladen, müssen Sie – Stand 3.1.2006 – leider erst drei kleine Fehler beseitigen, da die Anwendung noch mit einer Beta-Version von Visual Studio entwickelt wurde. Die Eigenschaft `BackSpace` korrigieren Sie in `Backspace` und `SpaceBar` an zwei Stellen in `Spacebar`. Die Fehlerliste wird Ihnen beim Finden der Fehler helfen. Bedenken Sie, dass C# die Groß-/Kleinschreibung unterscheidet!

Das Klassenprinzip am einfachsten Beispiel

Bevor wir uns an die Entwicklung einer eigenen Klasse machen, betrachten wir zunächst den Umgang mit zwei Wertetypen in .NET genauer, die Sie in eigenen Programmen vielleicht schon selbst verwendet haben: Die Datentypen `Decimal` und `Date`.

Der `Decimal`-Datentyp wird verwendet, um gebrochene Werte mit höchster Präzision zu speichern. Er findet Einsatz gerade in finanzmathematischen Anwendungen, weil er durch besondere Rechenverfahren zahlensystembedingte Rundungsfehler ausschließt.

Mit dem `Date`-Datentyp speichern Sie Datumswerte. Sie können mit seiner Hilfe aber auch spezielle Funktionen verwenden, die beispielsweise herausfinden, welche Wochentagsnummer ein bestimmtes Datum hat (1 für Montag, 2 für Dienstag, etc.).

Diese beiden Datentypen, die zu den so genannten *primitiven Datentypen* (da fest im CTS verankert) zählen, bieten sich für die Demonstration einfachster Klassen[2] an, weil man an ihnen am einfachsten erkennt und versteht, was mit der Aussage »Klassen kapseln Daten *und* Programmcode« gemeint ist.

Zu diesem Zweck schauen wir uns ein einfaches Beispielprogramm an.

BEGLEITDATEIEN: Sie finden die Codedateien zu diesem Beispiel im Verzeichnis *.\VB 2005 - Entwicklerbuch\D - OOP\Kap08\PrimDataTypes*. Öffnen Sie dort die entsprechende Projektmappen-Datei (*.sln*).

Schauen wir uns das Programmchen, bevor wir es starten, zunächst einmal an:

```
Module Module1

    Sub Main()

        '*****************************************************
        'Klassen bzw. Strukturen speichern Daten...
        '*****************************************************

        'Einen Decimal-Datentyp deklarieren und ihm den Wert 11.100,34 zuweisen
        Dim locDecimal As New Decimal(11100.34)
        Console.WriteLine("locDecimal enthält den Wert:  " & locDecimal)

        'Einen Date-Datentyp deklarieren und ihm den Wert "10.8.2005" zuweisen
        Dim locDate As New Date(2005, 8, 10)

        '*****************************************************
        '...stellen aber auch Programmcode zur Verfügung
        '*****************************************************
```

[2] Obwohl es sich bei ihnen streng genommen nicht um Klassen sondern um Strukturen handelt. Doch für die ersten Gehversuche ignorieren wir diesen Unterschied zunächst, da er für das Verständnis, um das es zunächst geht, nicht relevant ist, und Klassen und Strukturen diesbezüglich auf gleichen Prinzipien basieren; mehr über die Unterschiede zwischen den beiden Entitätstypen *Klassen* und *Strukturen* erfahren Sie später in diesem Kapitel.

```
        Dim locAusgabe As String
        'ToString wandelt unter Angabe des Formats den Wert
        'von locDecimal in eine Zeichenkette um
        locAusgabe = locDecimal.ToString("#,##0.0000 $")
        Console.WriteLine("Formatierte Zahlenausgabe:  " & locAusgabe)

        'DayOfWeek bestimmt den Wochentag des angegebenen Datums.
        Dim locWochentag As Integer
        locWochentag = locDate.DayOfWeek
        Console.WriteLine("Der " & locDate & " war der " & locWochentag & ". Tag der Woche!")

        'Auf Tastendruck warten
        Console.ReadKey()
    End Sub

End Module
```

Wie Sie im Listing erkennen können, besteht dieses Programm aus zwei kleinen Blöcken. Der erste Block deklariert wie angekündigt die zwei Datentypen, weist ihnen Werte zu und gibt diese schließlich als Beweis für ordnungsgemäßes Funktionieren auf dem Bildschirm aus.

Der zweite Block verwendet die beiden Datentypen ein zweites Mal – doch dieses Mal werden Funktionen verwendet, die direkt irgendwelche Operationen mit den Daten durchführen, die die beiden Typen speichern (im oben stehenden Listing fett gedruckt): Die `ToString`-Funktion des `Decimal`-Wertes wandelt den Wert, den die `Decimal`-Variable speichert, in eine druckbare Zeichenkette um und formatiert die Zahlendarstellung so, dass Tausenderpunkte gezeigt und genau vier Nachkommastellen mit ausgegeben werden – dazu dient die Formatzeichenfolge »#,##0.0000 $«. Jeder »Lattenzaun« dient dabei für eine mögliche auszugebende Zahl, die berücksichtigt wird, wenn die Zahl ausreichend groß ist. Jede »0« dient für eine Ziffer, die in jedem Fall ausgegeben wird. Ist die Zahl nicht groß bzw. nicht klein genug, um die entsprechenden Ziffern »zu produzieren«, werden an den entsprechenden Stellen Nullen ausgegeben.

Im zweiten Beispiel sorgt die `DayOfWeek`-Funktion des `Date`-Wertes, dass die Wochentagsnummer des Datums errechnet wird, das gegenwärtig in der `Date`-Variablen `locDate` gespeichert wird.

Die Ausgabe sieht dementsprechend folgendermaßen aus:

```
locDecimal enthält den Wert:  11100,34
locDate enthält den Wert:  10.08.2005

Formatierte Zahlenausgabe:  11.100,3400 $
Der 10.08.2005 war der 3. Tag der Woche!
```

Was dieses simple Beispiel verdeutlicht: Selbst einfache Datentypen wie `Decimal`, `Date` aber auch `Integer`, `Double` und andere dienen nicht nur zur Speicherung von Daten. Sie enthalten auch Programmcode. Und das Kapseln dieses Programmcodes im Datentyp (der Struktur oder der Klasse) ist das eigentliche Geniale daran.

Um Daten zu manipulieren, brauchen Sie nicht irgendwelche Funktionen aufzurufen, die sich sonst irgendwo in Ihrem Programm befinden. Sie bedienen sich einfach der Funktionen, die Sie direkt mit in dem »Bereich« Ihres Programms speichern, der auch zur Speicherung der eigentlichen Daten dient. Und dazu dienen eben die Klassen (bzw. Strukturen – doch die sind unter Berücksichtigung unseres momentanen Wissenstandes noch dasselbe).

Bis hierher haben Sie gesehen, was es heißt, in der CLR vorhandene Klassen bzw. Strukturen zu verwenden, die es ermöglichen, Daten und Programmfunktionalität unter einem Dach zu kapseln. Sie konnten nachvollziehen, dass gerade Code umfangreicher Projekte auf diese Weise sehr viel einfacher wieder zu verwenden ist, da er – etwa wie ein Kontextmenü des Elements einer Windows-Anwendung – quasi an dem Objekt »hängt«, das er verarbeiten bzw. manipulieren soll. Sie können sich auch sicherlich vorstellen, dass dieses Prinzip angewendet in Ihren eigenen Programmen sehr dazu beiträgt, dass diese aufgeräumter, klarer strukturiert und auch sehr viel leichter lesbar werden.

Das Klassenprinzip am eigenen Beispiel

Wie es nun aussieht, eigene Datentypen in Form von eigenen Klassen (und später auch in Form von eigenen Strukturen zu entwerfen), soll das Beispiel in diesem Abschnitt demonstrieren.

Ziel unseres ersten Klassengehversuchs soll es sein, die Problemstellung aus dem letzten Kapitel (Sie erinnern sich an die antiquarischen Bücher) ein wenig OOP-näher zu verpacken. Auch dazu finden Sie in den Begleitdateien ein vorbereitetes Programmbeispiel.

BEGLEITDATEIEN: Sie finden die Codedateien zu diesem Beispiel im Verzeichnis *.\VB 2005 - Entwicklerbuch\D - OOP\Kap08\RomNum_Net01*. Öffnen Sie dort die entsprechende Projektmappen-Datei (*.sln*).

Die Klasse, mit der wir die Problemlösung beginnen, soll zunächst nur die Aufgabe haben, einen Wert vom Typ `RomanNumerals` zu speichern. Dabei ist es natürlich klar, dass wir das Framework nicht dazu bringen werden, römische Numeralia in nativer Form zu speichern. Vielmehr kann unsere Klasse den eigentlichen Wert nur durch einen der bereits vorhandenen primitiven Datentypen speichern und on top Konvertierungsfunktionen zur Verfügung stellen, die sich um die entsprechenden Umwandlungen von römischen Numeralia, die in Zeichenkettenform vorliegen, in numerische Werte und umgekehrt kümmern. Oder anders gesagt: Der Plan ist es, einen eigenen Datentypen zu schaffen, der in etwa wie ein schon vorhandener Datentyp funktioniert – beispielsweise wie ein `Integer` (der ganze Zahlen in einem bestimmten Zahlenbereich), ein `String` (der Zeichenketten) oder `Double` (der gebrochene Werte speichert). Unser neuer Typ speichert ebenfalls einen Wert, etwa wie der Datentyp `Integer`. Doch gleichzeitig stellt dieser neue Typ auch Funktionen bereit, um römischen Numeralia in numerische Werte, oder gespeicherte Werte wieder zurück in römische Numeralia umzuwandeln.

Nicht weniger als fünf Funktionen braucht unsere Klasse dazu. Wieso fünf? Aus folgendem Grund: Wir werden zwei Funktionen haben, die unabhängig vom Wert verwendbar sind, den die Klasse speichert. Sie stellen die eigentlichen »Malocher« in der Klasse dar, und die eine wandelt römische Numeralia, die als String vorliegen, in Integer-Werte um; die andere macht das Umgekehrte. Und dann haben wir zwei Funktionen, die das Gleiche mit dem Wert machen, den die Klasse selbst speichert.

Schauen wir uns die Beschreibung der Funktionen an:

- `RomanNumeralFromValue` wandelt die angegebene arabische Zahl (ganz normaler `Integer`) in einen String um, der dann das entsprechende römische Numerale enthält. Das war in der prozeduralen Beispielversion des letzten Kapitels schon so.
- `ValueFromRomanNumeral` wandelt den angegebenen String, der das römische Numerale enthält, wieder zurück in einen Integerwert – die arabische Zahl. Auch das war schon so in der prozeduralen Version.

- `FromRomanNumeral` weist den angegebenen String, der ein römisches Numerale enthält, dem »Klassenelement« zu. Das ist neu, denn in der an den VB6-Stil angelehnten Version dieses Beispiels gab es nichts, wohinein man das römische Numerale hätte speichern können.
- `ToRomanNumeral` hat gar keinen Parameter. Es wandelt das »Klassenelement« zurück in einen String, der das römische Numerale darstellt.

Es gibt im Übrigen noch eine fünfte Funktion in dieser Klasse, nämlich:

- `FromInt` verhält sich prinzipiell wie `FromRomanNumeral`, nimmt allerdings kein römisches Numerale sondern einen normalen Integer-Wert als Parameter entgegen, um die Klasseninstanz mit einem Wert zu füllen.

Sie können also wahlweise nur die »festen« Funktionen der Klasse verwenden, um Elemente hin- und herzukonvertieren, ohne irgendetwas zu speichern. Sie können aus der Klasse aber auch ein »Klassenelement machen« – korrekt muss es heißen: Sie instanzieren die Klasse und erhalten ein Objekt – um dann im `RomanNumeral`-Objekt selbst den Wert für die römische Darstellung zu speichern. Sie brauchen in diesem Fall keine spezielle Variable mehr (wie im vorherigen Beispiel), die den »unterliegenden« Wert speichert.

Der folgende Abschnitt arbeitet die Unterschiede zwischen den beiden Funktionstypen heraus.

Statische und nicht-statische Methoden und Variablen

Im Prinzip unterscheidet sich der neue Datentyp `RomanNumerals` nicht von »normalen«, so genannten primitiven Datentypen: Ein `Integer` speichert beispielsweise ganze Zahlen in einem bestimmten Wertebereich. Wenn Sie den Inhalt einer Integervariablen mit `Console.WriteLine` ausgeben, wird er in arabischen Ziffern dargestellt. Der Datentyp des Beispielprogramms speichert ganze Werte im Bereich von 1-3999; er liefert die Zahlen im Bedarfsfall aber in »römischer Schreibweise« aus.

Schauen Sie sich als nächstes den Inhalt der Klasse `RomanNumerals` an (doppelklicken Sie dazu im Projektmappen-Explorer auf *RomanNumerals.vb*):

```
Public Class RomanNumerals

    Private myUnderlyingValue As Integer = 1000

    Public Function ToRomanNumeral() As String

        'Statische Funktion aufrufen
        Return RomanNumeralFromValue(myUnderlyingValue)

    End Function

    Public Sub FromNumeral(ByVal RomanNumeral As String)

        'Statische Funktion aufrufen
        myUnderlyingValue = ValueFromRomanNumeral(RomanNumeral)

    End Sub
```

```
    Public Sub FromInt(ByVal ArabicInt As Integer)

        'einfach der Instanzvariable zuweisem
        myUnderlyingValue = ArabicInt

    End Sub

    Public Shared Function RomanNumeralFromValue(ByVal ArabicNumber As Integer) As String

    .
    .
    .

    End Function

    Public Shared Function ValueFromRomanNumeral(ByVal RomanNumeral As String) As Integer

    .
    .
    .

    End Function

End Class
```

Den Inhalt der beiden Funktionen habe ich aus Platzgründen nicht gezeigt, da er in diesem Zusammenhang weniger interessiert und vom eigentlichen Thema nur ablenkt. Später innerhalb dieses Kapitels werde ich für die Demonstration anderer Konzepte nochmals auf die Beschreibung dieser Funktionen zurückkommen.

Betrachten Sie zunächst die Variable `myUnderlyingValue` (im Listing fett markiert). Auf diese Variable kann (fast) vom gesamten Klassencode aus zugegriffen werden, da sie direkt unter dem Klassenkopf definiert wurde. Variablen, die so deklariert werden, dass sie in der ganzen Klasse gültig sind, werden als Klassenvariablen oder als *Member-Variablen* bezeichnet. Man sagt auch: Der Gültigkeitsbereich der Member-Variablen umfasst die gesamte Klasse.

Die drei nächsten Funktionen sind Instanzmethoden der Klasse. Sie lassen sich nur verwenden, wenn die Klasse zuvor *instanziert* wurde. Mit `FromNumeral` können Sie eine Instanz dieser Klasse mit einem Wert füllen und damit die Klasse z. B. folgendermaßen verwenden:

```
Dim Numerale as New RomanNumerals
Numerale.FromNumeral("IV")
```

Um noch mal zurück zur Sandkastenanalogie zu kommen: Sie nehmen das Förmchen `RomanNumerals` und füllen es mit *neuem* (*New*) Sand. Den geformten Sandklotz, der dabei herauskommt, nennen Sie `Numerale`. Sie können jetzt ein Stöckchen nehmen, und »IV« in den Sandklotz kratzen. (Sie könnten im Übrigen aber niemals die »IV« in das Förmchen – in die Vorlage `RomandNumerals` kratzen – deswegen müssen Sie eine Klasse in ein Objekt instanzieren, bevor Sie seinen Wert verändern können.)

Mit dieser Vorgehensweise haben Sie den Wert 4 der Klasseninstanz von `RomanNumerals` zugewiesen. Wenn Sie den Wert abrufen wollen, verwenden Sie die Funktion `ToRomanNumeral`,

```
Dim strVar as String=Numerale.ToRomanNumeral()
```

die den Wert der Klasseninstanz als `String` zurückliefert.

Wenn Sie von den so genannten *statischen* (den »festen«) Funktionen Gebrauch machen möchten, rufen Sie sie mit dem Klassennamen selbst als Referenz auf. Etwa:

```
Dim strVar as String=RomanNumerals.RomanNumeralFromValue ("IX")
```

Sie haben jetzt im Gegensatz zum vorherigen Beispiel den Inhalt der Klasseninstanz nicht berührt. Statische Methoden (Funktionen) werden als solche bezeichnet, da sie *immer* zur Verfügung stehen – sie müssen die Klasse nicht in ein Objekt instanzieren, um sie verwenden zu können. Natürlich brauchen auch statische Funktionen Daten (Sandkuchen), die sie verarbeiten können – aber in unserem Beispiel liefern Sie der Funktion den Sandkuchen, in den sie die »IV« kratzen soll, gleich mit.

WICHTIG: In Visual Basic werden statische Funktionen oder Variablen mit dem Schlüsselwort `Shared` definiert – was leider ein wenig verwirrend ist, denn das Schlüsselwort `Static` gibt es ebenfalls.

Nicht verwirren lassen: Static und Shared in VB

`Static` im Gegensatz zu `Shared` bewirkt, dass eine Variable, die nur innerhalb einer Prozedur (`Sub` oder `Function`) verwendet wird, ihren Inhalt auch nach Verlassen der Unterroutine nicht verliert. Dennoch können Sie auf diese Variable nur innerhalb der Unterroutine zugreifen, in der sie definiert wird. Diese Verwendung von `Static` gibt es übrigens bei keiner anderen der mit Visual Studio ausgelieferten .NET-Programmiersprachen. Im Prinzip ist eine mit `Static` definierte Variable eine, die als `Shared`-Member für die Klasse definiert und mit einem internen Attribut versehen wurde, das die Verwendung auf den Gültigkeitsbereich reglementiert, in dem sie deklariert wurde.

Vorhanden sind `Static`-Variablen übrigens nur aus Gründen der Kompatibilität zum alten Visual Basic 6.0 (und vorherigen Versionen).

Um Beispiele für statische Methoden im Framework zu finden, brauchen Sie gar nicht lange zu suchen – Sie finden sie schon bei den primitiven Typen. Ein Beispiel:

Mit

```
Dim intTemp as Integer=5
Dim strTemp as String=intTemp.ToString()
```

wandeln Sie den Wert 5 in eine Zeichenkette um und weisen sie der String-Variablen *strTemp* zu. Sie haben hier eine nicht-statische Methode der Klasseninstanz (des aus der Klasse hervorgegangenen Objektes, in diesem Fall `intTemp`) verwendet.

Mit

```
Dim intTemp As Integer
intTemp = Integer.Parse("1234")
```

hingegen verwenden Sie die statische Funktion der »Klasse«[3] `Integer`, die versucht, eine Zeichenkette in eine `Integer`-Variable umzuwandeln. Inhalt von möglicherweise existierenden Instanzen (also wieder von `intTemp`, das aus `Integer` entstand) wird dabei nicht verändert oder auch nur in irgendeiner Form verwendet. Sie nutzen `Parse` nur als eigenständige Funktion (und geben ihr einen eigenen Sandkuchen zur Verarbeitung). Der Sandkuchen, der durch den Wert von `intTemp` selbst dargestellt wird, trocknet weiterhin unangetastet in der Sonne.

Nicht-statische Member-Variablen sind logischerweise nicht aus statischen Methoden zu erreichen. Platzieren Sie beispielsweise die Zeile

```
myUnderlyingValue=5
```

in der statischen Funktion `RomandNumeralFromValue`, dann zeigt Ihnen Visual Basic sofort einen Fehler an, etwa wie in Abbildung 8.2 zu sehen.

```
Public Shared Function RomanNumeralFromValue(ByVal Arab

    Dim locCount As Integer = 0
    Dim locDigitValue As Integer = 0
    Dim locRoman As String = ""
    Dim locDigits As String = ""

    myUnderlyingValue = 5
    Auf einen Instanzmember einer Klasse kann nicht ohne explizite Instanz einer Klasse von einer/m freigegebenen
    Methode/Member aus verwiesen werden.
    locDigits = "IVXLCDM"
```

Abbildung 8.2: Was dieser Fehlerhinweis wirklich meint: In statischen Funktionen können Sie nicht auf Member-Variablen einer Instanz zugreifen

Ganz klar: Die statische Funktion `RomanNumeralsFromValue` kann den Klassen-Member auch gar nicht verwenden, denn wie sollte sich das auswirken? Sie können die Klasse `RomanNumerals` in 100 verschiedene Objekte instanziert haben, und jedes Objekt hat einen anderen Wert für `myUnderlyingValue`. Da `RomanNumeralFromValue` eine statische Klassenfunktion ist, könnte man, falls ein Verändern von `myUnderlyingValue` erlaubt wäre, vielleicht noch annehmen, dass sich dabei der Wert *aller* zu diesem Zeitpunkt instanzierten Klassen ändert, aber das ergibt letzten Endes überhaupt keinen Sinn. Es wäre so, als hätten Sie 100 Sandkuchen aus einem Förmchen erzeugt, und würden erwarten, dass sich die schon zum Trocknen ausgelegten Sandkuchen veränderten, wenn Sie das Förmchen selbst vielleicht mithilfe eines Feuerzeugs heiß machten und umformten.

[3] In Anführungszeichen deswegen geschrieben, da es sich bei `Integer` nicht wirklich um eine Klasse, sondern um eine Struktur handelt. Doch dazu später mehr.

Eine kleine Anmerkung zu einer Tatsache, die leicht verwirrt: Um die statischen Funktionen einer Klasse oder Struktur aufzurufen, können Sie übrigens nicht nur den Weg über den Klassen- bzw. Strukturnamen, sondern auch den über die Instanz der Klasse nehmen. Die Zeilen

```
Dim intTemp As Integer
intTemp = intTemp.Parse("1234")
```

bewirken exakt dasselbe, wie der Aufruf von `Integer.Parse`, also wie der Weg über den Klassennamen selbst – allerdings ist das kein toller Stil, der auch dem Compiler seit Visual Studio 2005 Anlass zum Meckern gibt (siehe Abbildung 8.3).

TIPP: Wie in Abbildung 8.3 zu sehen, gibt es – vielleicht kennen Sie das schon von Office 2003 – im Visual Studio-2005-Editor nach einer Bearbeitung eines Objektes bzw. Objektnamens so genannte Smarttags; so Sie sich ▶ Kapitel 3 zu Gemüte geführt haben, wissen Sie darüber längst Bescheid. Der folgende graue Kasten fasst den Umgang mit Smarttags nochmals kurz zusammen.

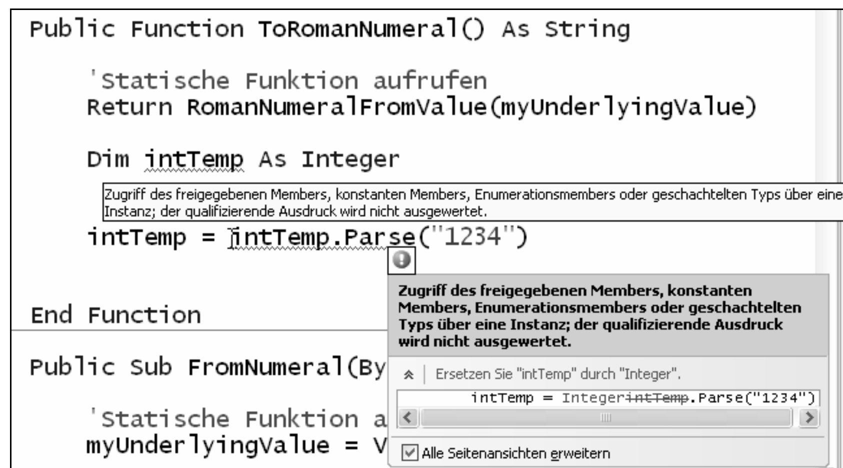

Abbildung 8.3: Ist zwar gestattet, aber nicht schön: Aufrufen von statischen Methoden über eine Objektvariable. Verbesserungsvorschläge gibt es durch den Smarttag-Klick (auf's Rufzeichen)

Smarttags im Editor von Visual Basic

Einen Smarttag erkennen Sie zunächst als kleinen, hellen Unterstrich in einer Codezeile im Visual Basic-Editor, an der es seiner Meinung nach irgendetwas zu verbessern oder anzumerken gibt.

Fahren Sie mit dem Mauszeiger auf diesen Unterstrich, verwandelt er sich in ein kleines Symbol mit der Form eines Ausrufungszeichen, das Ihnen verschiedene Arten von Unterstützung anbietet (welche kommt ganz auf den Zusammenhang an). In vielen Fällen verbirgt sich hinter dem Smart Tag ein Dialog, der Ihnen einen Korrekturvorschlag für den erkannten »Fehler« unterbreitet.

Die folgende Abbildung verdeutlicht den Zusammenhang. ▶

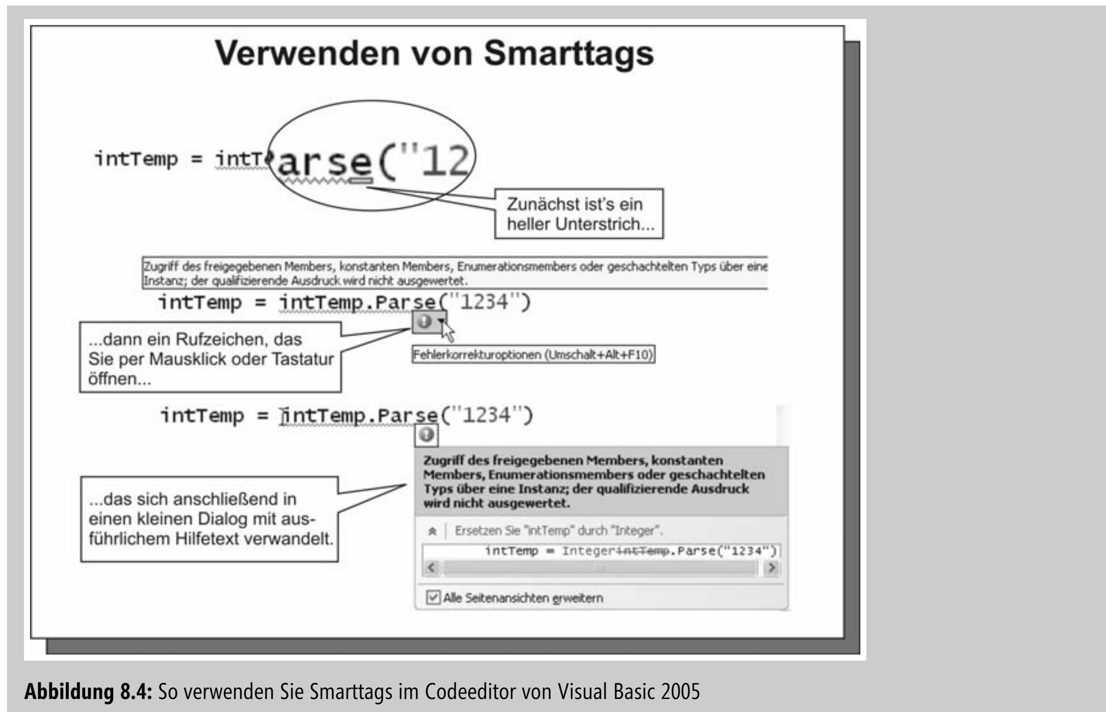

Abbildung 8.4: So verwenden Sie Smarttags im Codeeditor von Visual Basic 2005

Schauen Sie sich zum Abschluss das Hauptprogramm und dessen Verhalten in Aktion an. Wenn Sie das Programm starten, werden Sie aufgefordert, eine Zahl einzugeben. Das Programm nimmt die Zahl entgegen und zeigt Ihnen das entsprechende römische Numerale auf dem Bildschirm an. Die Klasse, die das Hauptprogramm enthält, befindet sich übrigens in der Klassendatei *Main.vb*, die Sie per Doppelklick im Projekt-Explorer zum Bearbeiten öffnen können.

```
Public Class Main
    Shared Sub Main()
        Dim locRomanNumeral As New RomanNumerals

        'Text ausgeben; Anwender zur Eingabe auffordern.
        Console.Write("Geben Sie eine Zahl zwischen 1 und 3.999 ein: ")

        'Zahl als Text einlesen, mit der statischen Methode Parse in Integer
        'umwandeln und das Ergebnis der Klasseninstanz zuweisen.
        locRomanNumeral.FromInt(Integer.Parse(Console.ReadLine()))

        'Das römische Literale ausgeben, das in der Klasseninstanz gespeichert ist...
        Console.WriteLine("Entspricht dem römischen Numerale " & locRomanNumeral.ToRomanNumeral)

        'Dies nur noch, damit nicht alles sofort wieder verschwindet...
        Console.WriteLine()
        Console.WriteLine("Return drücken zum Beenden...")
        Console.ReadLine()
    End Sub
End Class
```

Dieses Programm veranschaulicht noch mal das gerade Besprochene. Es legt mit `New` eine neue Instanz der Klasse `RomanNumerals` mit dem Objekt `locRomanNumeral` an. Es benutzt eine statische Funktion des `Integer`-Typs, um die von der Tastatur durch `ReadLine` eingelesene Zeichenkette in einen `Integer`-Wert umzuwandeln. Und es verwendet eine Instanzfunktion von `locRomanNumeral` (also der Klasse `RomanNumerals`), um den Wert des `RomanNumeral`-Objektes als römisches Numerale auszugeben.

Kleiner Exkurs – womit startet ein Programm?

Wie Sie sicherlich unschwer bemerkt haben, startet das Programm mit der Prozedur *Sub Main*. Wenn Sie hingegen eine Windows-Applikation entwickeln, startet sie standardmäßig mit der Darstellung des als erstes dem Projekt hinzugefügten Formulars. Natürlich können Sie bestimmen, wie Ihr Programm starten soll. Und auch wenn Sie eine Windows-Applikation entwickeln, muss diese nicht notwendigerweise mit einem Formular »beginnen«.

Sie legen das Startobjekt der Anwendung fest, indem Sie den Eigenschaftendialog des Projektes anzeigen lassen. Diesen bringen Sie auf den Bildschirm, indem Sie mit der rechten Maustaste auf den Projektnamen im Projekt-Explorer klicken und im Menü, das sich jetzt öffnet, *Eigenschaften* auswählen. Im aufklappbaren Listenfeld *Startobjekt* wählen Sie aus, wie Ihr Programm starten soll.

Darüber hinaus haben Sie für Windows-Anwendungen durch das so genannte Anwendungsframework auch die Möglichkeit, in den Startprozess einzugreifen. Bevor in einer Windows-Anwendung also das erste Formular angezeigt wird, können Sie bereits eine Ereignisbehandlungsroutine implementieren, in der Sie bestimmte notwendige Vorbereitungen erledigen. Mehr zum Anwendungsframework für Windows Forms-Anwendungen erfahren Sie in ▶ Kapitel 26.

Mit Sub New bestimmen, was beim Instanzieren passiert – der Klassenkonstruktor

Sicherlich haben Sie bemerkt, dass die Klasse aus dem ersten Beispiel ein wenig unhandlich war. Sie musste erst instanziert werden, und anschließend konnten Sie dem aus ihr hervorgehenden Objekt durch die Benutzung einer Methode einen Wert zuweisen. Von Klassen, die Sie aus dem Framework möglicherweise bereits verwendet haben, wissen Sie, dass die Instanzierung einer Klasse mit `New` und der Angabe eines Parameters die Initialisierung der Klasseninstanz einfach und unkompliziert macht. Die bisher verwendete Beispielklasse ist hinsichtlich dessen ein wenig armselig, doch das soll sich jetzt ändern.

Bisher instanzieren Sie die Klasse mit einer Codezeile und weisen der Klasseninstanz einen Wert mit einer weiteren Codezeile zu:

```
Dim locRomanNumeral As New RomanNumerals
locRomanNumeral.FromInt(Integer.Parse(Console.ReadLine()))
```

Einfacher wäre es, wenn Sie schon beim `New` in der ersten Zeile bestimmen könnten, welchen Wert die Objektinstanz annehmen soll, etwa mit:

```
Dim locRomanNumeral As New RomanNumerals(Integer.Parse(Console.ReadLine))
```

Ab Visual Basic .NET ist das kein Problem. Im Gegensatz zu Visual Basic 6.0 kennt Visual Basic .NET die `Sub New` für Klassen. Eine solche Funktion nennt man den *Klassenkonstruktor*. Sie formulieren einen Klassenkonstruktor genau wie jede andere Prozedur, und damit ist seine Implementierung denkbar einfach.

BEGLEITDATEIEN: Sie finden die Codedateien zum folgenden Beispiel im Verzeichnis *.\VB 2005 - Entwicklerbuch\D - OOP\Kap08\RomNum_Net02*. Öffnen Sie dort die entsprechende Projektmappen-Datei (*.sln*).

Lassen Sie uns die veränderte Version der Klasse betrachten:

```
Public Class RomanNumerals

    Private myUnderlyingValue As Integer

    Public Function ToRomanNumeral() As String

        'Statische Funktion aufrufen.
        Return Me.RomanNumeralFromValue(myUnderlyingValue)

    End Function

    Public Sub New(ByVal RomanNumeral As String)

        'Statische Funktion aufrufen.
        myUnderlyingValue = ValueFromRomanNumeral(RomanNumeral)

    End Sub

    Public Sub New(ByVal ArabicInt As Integer)

        'Einfach der Instanzvariablen zuweisen.
        myUnderlyingValue = ArabicInt

    End Sub

    Public Shared Function RomanNumeralFromValue(ByVal ArabicNumber As Integer) As String
    ...'Zuviel Programmcode; dieser Part interessiert momentan nicht.
    .
    .
    .
    End Function

    Public Shared Function ValueFromRomanNumeral(ByVal RomanNumeral As String) As Integer
    .
    .
    .
    End Function

End Class
```

Whoops, was ist das? Nicht nur, dass in dieser Version zwei Funktionen fehlen – `Sub New` ist hier wie angekündigt vorhanden und das sogar doppelt!

Warum? Die beiden veralteten Funktionen FromNumeral und FromInt dienten bislang lediglich der Initialisierung einer Klasseninstanz. Mit der überarbeiteten Version können Sie eine Klasseninstanz schon im Konstruktor definieren, und damit sind die beiden Funktionen hinfällig.

Wenn Sie möchten, dass Code im Konstruktor einer Ihrer Klassen ausgeführt wird, in dem Sie dann beispielsweise Member-Variablen initialisieren können, dann implementieren Sie eine Sub New. Der einzige Unterschied zu normalen Funktionen besteht darin, dass der Prozedurenname aus einem Schlüsselwort besteht, welches Visual Studio-Editor und -Compiler als ein solches erkennen und blau markieren. Konstruktoren können übrigens nur aus Subs und nicht aus Functions bestehen – was auch keinen Sinn ergäbe, da das Instanzieren einer Klasse mit New ja schon das instanzierte Objekt sozusagen als »Funktionsergebnis« von New zurückliefert.[4]

Zurück zum Beispielprogramm: Um die Klasseninstanz zu deklarieren und ihr einen Wert zuzuweisen, benötigen Sie mit dieser Version des Beispielprogramms nur noch eine einzige Zeile. Sub Main des Beispielprogramms dieser Version sieht folgendermaßen aus:

```
Public Class Main

    Shared Sub Main()

        'Text ausgeben; Anwender zur Eingabe auffordern.
        Console.Write("Geben Sie eine Zahl zwischen 1 und 3.999 ein: ")

        'Instanz der Klasse RomanNumerals bilden UND
        'Zahl als Text einlesen, mit der statischen Methode Parse in Integer
        'umwandeln und das Ergebnis der Klasseninstanz zuweisen.
        Dim locRomanNumeral As New RomanNumerals(Integer.Parse(Console.ReadLine))

        'Das römische Literale ausgeben, das in der Klasseninstanz gespeichert ist.
        Console.WriteLine("Entspricht dem römischen Numerale " & locRomanNumeral.ToRomanNumeral)

        'Dies nur noch, damit nicht alles sofort wieder verschwindet.
        Console.WriteLine()
        Console.WriteLine("Return drücken zum Beenden...")
        Console.ReadLine()

    End Sub

End Class
```

Was dieses Beispiel anbelangt, stellt sich nur noch die Frage, worin der Sinn von zwei Konstruktoren in der Beispielklasse besteht. Doch lassen Sie mich aus gegebenem Anlass zuvor einen kleinen Ausritt in die Abwärtskompatibilität von Code Libraries nehmen, bevor wir zum eigentlichen Thema zurückkommen:

[4] Wobei diese Erklärung rein technisch natürlich falsch ist, und lediglich als Eselsbrücke dienen soll.

Überflüssige Funktionen mit dem Obsolete-Attribut markieren

Machen Sie das hier im Beispiel gezeigte, bitte AUF KEINEN FALL in Ihren eigenen Klassen. Löschen Sie keine Funktionen oder sonstige Elemente bei Assemblies, die Sie schon im Einsatz haben. Sie riskieren dadurch, dass Anwendungen, die bereits mit einer älteren Version einer solchen Klasse arbeiten, nicht mehr funktionieren.

Stattdessen können Sie eine Technik des Frameworks verwenden, die es erlaubt, Elemente einer Klasse oder die Klasse selbst zu markieren – so genannte *Attribute*. Attribute selbst basieren auf der `Attribute`-Klasse des Frameworks, die allerdings selbst, mit Ausnahme bestimmter Informationen die sie speichern, über keinerlei Funktionen verfügen.

Sie dienen nur dazu, das markierte Element näher zu beschreiben, und damit verhalten sie sich, wie beispielsweise verschiedene Schriften in einem Text. Zeichnen Sie in Ihren Texten bestimmte Passagen mit verschiedenen *Attributen* aus, dann machen Sie den Leser auf gewisse Dinge **aufmerksam** (so wie hier mit dem Attribut Fett- oder Kursivschrift). Dem gleichen Zweck dienen Attribute zur Markierung von Klassen und Klassenelementen.

Stattdessen können Sie ein Element, das nicht mehr benutzt werden soll, mit dem `Obsolete`-Attribut kennzeichnen. Das `Obsolete`-Attribut vor einem Funktionsnamen bewirkt, dass der Compiler, wenn er die Verwendung eines derart gekennzeichneten Elements entdeckt, eine Warnung oder einen Fehler ausgibt.

O.k., zugegeben, ich hab das in diesem Beispiel auch nicht gemacht. Die beiden Funktionen sind noch vorhanden – sie waren nur durch eine `#Region`-Direktive ausgeblendet – Sie konnten das natürlich nur dann bemerken, wenn Sie sich den Quellcode nicht nur hier im Buch, sondern auch in der Visual Studio-IDE angeschaut haben.

Hier sind die beiden versteckten Funktionen – die folgenden Zeilen demonstrieren Ihnen dabei gleich die korrekte Verwendung des `Obsolete`-Attributs:

```
#Region "Obsolete Funktionen"
    <Obsolete("Verwenden Sie statt dieser Funktion den Klassenkonstruktor")> _
    Public Sub FromNumeral(ByVal RomanNumeral As String)

        'Statische Funktion aufrufen
        myUnderlyingValue = ValueFromRomanNumeral(RomanNumeral)

    End Sub

    <Obsolete("Verwenden Sie statt dieser Funktion den Klassenkonstruktor")> _
    Public Sub FromInt(ByVal ArabicInt As Integer)

        'einfach der Instanzvariable zuweisem
        myUnderlyingValue = ArabicInt

    End Sub
#End Region
```

Wenn Sie eine der Funktionen nun spaßeshalber im Beispielprogramm verwenden, macht Sie die IDE in der Aufgabenliste darauf aufmerksam.

```
locRomanNumeral.FromNumeral("XI")
"Public Sub FromNumeral(RomanNumeral As String)" ist veraltet: "Verwenden Sie statt dieser Funktion den Klassenkonstruktor"
```

Abbildung 8.5: Verwenden Sie eine als obsolet gekennzeichnete Funktion, werden Sie vom Compiler darauf hingewiesen, dies besser nicht zu tun

Falls Sie möchten, dass die Verwendung einer Funktion, die als obsolet gekennzeichnet ist, den Kompilierungsvorgang sogar fehlschlagen lässt, geben Sie als zweiten Parameter des `Obsolete`-Attributs einfach `True` an. Das Programm lässt sich dann solange nicht mehr starten, bis der Entwickler die entsprechenden Maßnahmen ergriffen und seine Anwendung ohne die Verwendung der »alten« Funktion umgeschrieben hat.

Überladen von Funktionen und Konstruktoren

Falls Sie zu den alten VB6-Hasen gehören, dann kennen Sie sicherlich den Vorteil von optionalen Parametern. Das Überladen von Funktionen in .NET ist ein nur auf den ersten Blick ähnliches, aber letzten Endes dennoch völlig anderes Konzept. Gemeinsam haben beide Konzepte, dass sie eine Liberalisierung von Parameterübergaben an Funktionen ermöglichen. Das war es aber dann auch schon mit den Gemeinsamkeiten.

Mit dem Überladen von Funktionen geben Sie Ihren Klassen eine enorme Flexibilität und Anpassungsgabe. Überladen von Funktionen bedeutet: Sie erstellen verschiedene Funktionen mit gleichen Namen, die sich nur durch den Typ, die Typreihenfolge oder die Anzahl der übergebenden Parameter unterscheiden. Als Beispiel schauen Sie sich bitte den folgenden Codeausschnitt an:

```
Sub EineProzedur()
    'Tu was.
End Sub

Sub EineProzedur(ByVal ein_Parameter As Integer)
    'Tu was anderes.
End Sub

Sub EineProzedur(ByVal ein_anderer_Parameter As String)
    'Tu was anderes.
End Sub

Sub EineProzedur(ByVal ein_Parameter As Integer, ByVal ein_anderer_Parameter As String)
    'Tu was anderes.
End Sub

Sub EineProzedur(ByVal ein_ganz_anderer_Parameter As Integer)
    'Fehler: ein Integer als Parameter gab's schon mal.
    'Die Methode 'EineProzedur' wurde mehrfach mit identischen Signaturen definiert.
End Sub
```

Es ist, als würde die Signatur – so nennt man die Mischung aus Parametertypen und Parameterreihenfolge beim Aufrufen einer Funktion – Bestandteil des Namens werden, und daran wird dann erkennbar, welche der vorhandenen `EineProzedur` aufgerufen werden soll. Der Variablenname hat damit übrigens überhaupt nichts zu tun – nur der übergebene Typ ist für die Identifizierung der Signatur entscheidend.

Aus diesem Grund bereitet die letzte `Sub` des Beispiels auch Probleme. Ihr wird genau wie der ersten eine Variable vom Typ `Integer` übergeben. Zwar ist der Variablenname ein anderer, aber darauf kommt es überhaupt nicht an – Namen sind hier tatsächlich nicht mehr als Schall und Rauch.

Wozu eignet sich die Funktionsüberladung in der Praxis? Nun, die Anwendung von Klassen wird dadurch ungleich flexibler. Schon bei unserem Beispiel kommen Sie spätestens beim Anwenden der Klasse `RomanNumerals` in den Genuss des Komforts von überladenen Funktionen. Sie müssen nicht wissen, welche Funktion beispielsweise für welche Teilaufgabe zuständig ist; sie können einfach die (eine) Funktion verwenden, und IntelliSense[5] unterstützt Sie bei der Auswahl der richtigen Signatur sogar. Wenn Sie die Zeile in Ihr Programm eingeben, die die Klasse in ein Objekt instanziert, dann zeigt Ihnen IntelliSense, nachdem Sie die geöffnete Klammer hinter `New` eingegeben haben, Ihre Optionen an, etwa wie in der folgenden Grafik zu sehen:

```
Shared Sub Main()

    'Text ausgeben; Anwender zur Eingabe auffordern
    Console.Write("Geben Sie eine Zahl zwischen 1 und 3.999 ein: ")

    'Zahl als Text einlesen, mit der statischen Methode Parse in Integer
    'umwandeln und das Ergebnis der Klasseninstanz zuweisen
    Dim locRomanNumeral As New RomanNumerals(Integer.Parse(Console.ReadLine))
                                  1 von 2  New (RomanNumeral As String)
```

Abbildung 8.6: IntelliSense hilft Ihnen bei der Auswahl der richtigen Signatur bei überladenen Funktionen – hier ist die Reihenfolge der beiden Programmzeilen übrigens noch nicht stimmig!

Im Gegensatz zu optionalen Parametern können Sie also Methoden implementieren, deren Versionen völlig unterschiedliche Dinge tun. Natürlich haben beide Methodenversionen des Beispiels thematisch miteinander zu tun (sollten sie auch, anderenfalls sollten Sie ihnen komplett andere Namen geben). Wichtig ist: Die Funktionsweise der beiden überladenen Prozeduren kann im Gegensatz zu *einer* Prozedur mit optionalen Parametern nicht nur komplett anders implementiert werden (die erste initialisiert die Klasse mit einem Integer, die andere durch die Angabe eines römischen Numerales), die beiden Methoden sind auch visuell sauber voneinander getrennt.

HINWEIS: Das Prinzip der Funktionsüberladung funktioniert für `Subs` genauso wie für `Functions`. Allerdings ist Folgendes bei der Verwendung von Funktionen wichtig zu wissen: Der Rückgabetyp kann nicht als Differenzierungskriterium von Signaturen verwendet werden. Das heißt im Klartext:

[5] Zur Wiederholung: So nennt sich die Eingabehilfe des Codeeditors, mit deren Hilfe sich Elementnamen vervollständigen, alle Elemente eines Objektes in Listen anzeigen oder Funktions- und Eigenschaftenüberladungen sich schon bei der Codeeingabe sichtbar machen lassen. Abbildung 8.6 zeigt IntelliSense in Aktion.

```
Function EineFunktion() As Integer
    'Tu was.
End Function

Function EineFunktion(ByVal AndereSignatur As Integer) As Integer
    'Das funktioniert.
End Function

Function EineFunktion() As String
    '"Public Function EineFunktion() As Integer" und "Public Function EineFunktion() As String" können
    ' sich nicht gegenseitig überladen, da sie sich nur durch Rückgabetypen unterscheiden.
End Function
```

Die ersten beiden Funktionen sind o.k., da sich die Signaturen von einander unterscheiden. Die letzte Funktion unterscheidet sich von der ersten allerdings nur durch den Rückgabetyp, und aus diesem Grund meldet der Visual Basic-Compiler schon zur Entwurfszeit einen Fehler.

TIPP: Sie können – zur besseren Lesbarkeit – das *Overloads*-Schlüsselwort verwenden, um die Überladung einer Methode deutlich zu machen:

```
Overloads Function EineFunktion() As Integer
    'Wenn Overloads verwenden,...
End Function

Overloads Function EineFunktion(ByVal AndereSignatur As Integer) As Integer
    '...dann bei bei den Funktionen
End Function
```

Dabei sollten Sie berücksichtigen: Wenn Sie sich für das *Overloads*-Schlüsselwort entscheiden, müssen Sie es bei *allen* Methodenvariationen mit Überladungen verwenden.

Methodenüberladung und optionale Parameter

Auch in der .NET-Version bietet Ihnen Visual Basic noch das Hilfsmittel der optionalen Parameter an. Optionale Parameter haben gegenüber überladenen Methoden entscheidende Nachteile:

- Sie werden von vielen anderen .NET-Programmiersprachen nicht unterstützt. Wenn Sie in Ihren Klassen optionale Parameter verwenden, haben ausschließlich Visual Basic-Entwickler etwas davon. Weder C# noch J# noch managed C++[6] unterstützen optionale Parameter.
- Die Verwendung von optionalen Parametern macht Ihren Code schwerer lesbar und kaum wiederverwendbar. Wenn Sie optionale Parameter verwenden, müssen Sie Fallunterscheidungen durchführen, indem Sie durch die Abfrage von Standardwerten herausfinden, welche Parameter vom Aufrufer übergeben wurden und welche nicht. Eine Methode, die nur aufgrund ihrer Parameter vielleicht zwei völlig verschiedene Dinge macht, trägt dann quasi den gequetschten »Doppelcode« in einem Funktionsrumpf. Mit überladenen Funktionen hingegen haben Sie zwei Problemlösungen auch optisch sauber voneinander getrennt.

[6] Managed C++ nennt sich die .NET-Version von C++, die wie alle anderen .NET-Programmiersprachen verwalteten (managed) Code erzeugt.

- Sollten Sie ohne strikte Typbindung arbeiten (*Option Strict Off*) und gleichzeitig überladene Funktionen und optionale Parameter für die gleichen Methoden verwenden, bedeutet das einen unglaublichen Leistungsverzicht, da die Laufzeitbibliothek von Visual Basic unter Umständen erst herausfinden muss, welche der Routinen am ehesten zum angegebenen Parameter passt. Bei sehr flexiblen Parameterübergaben können das obendrein potenzielle Fehlerquellen werden, die ich – ganz ehrlich gesagt – bei Fehlverhalten niemals debuggen möchte.

Aus diesen Gründen gehe ich auf die Eigenschaft, Parameter optional an Funktionen zu übergeben, auch nicht näher ein. Meine Empfehlung: Falls Sie optionale Parameter in früheren Visual Basic-Versionen kennen gelernt haben, versuchen Sie sie am besten nicht mehr zu verwenden. Falls Sie das Konzept der optionalen Parameterübergabe gar nicht kennen: umso besser!

Gegenseitiges Aufrufen von überladenen Methoden

Das Überladen von Funktionen begegnet Ihnen im Framework an jeder Ecke. In der Regel wird die Funktionsüberladung vom Framework verwendet, um dem Anwender die Handhabung so angenehm wie möglich zu machen. So werden ihm Signaturen für bestimmte Funktionen mit nur sehr wenigen Parametern angeboten, um Tipparbeit sparen, und gleichzeitig andere Versionen derselben Funktion mit sehr viel mehr Parametern für die größtmögliche Flexibilität.

Die *WriteLine*-Methode ist hier ein sehr anschauliches Beispiel, da sie mit nicht weniger als 18 Überladungen daherkommt. Wenn Sie im Codeeditor von Visual Basic die Anweisung *Console.Writeline* schreiben und anschließend die geöffnete Klammer eintippen, zeigt Ihnen IntelliSense die Signaturen der 18 verschiedenen Überladungen an.

```
'Das römische Literale ausgeben, das in der Klassen
Console.WriteLine("Entspricht dem römischen Numeral
 6 von 18  WriteLine (value As String)
 value: Der zu schreibende Wert.       t nicht alles sofort wieder ver
Console.WriteLine()
Console.WriteLine("Taste zum Beenden drücken...")
Console.ReadKey()
```

Abbildung 8.7: Die *WriteLine*-Methode hat nicht weniger als 18 Überladungen vorzuweisen!

Natürlich wäre das Überladen von Methoden keine wirkliche Arbeitserleichterung, wenn Entwickler die eigentliche Funktionalität für alle überladenen Methoden immer wieder implementieren müssten. Überladene Methoden können sich deswegen gegenseitig aufrufen – vom Sonderfall *Sub New* einmal abgesehen – ohne Einschränkungen.

Der übliche Weg, den Anwendern Ihrer Klassen (und damit meistens sich selbst) große Flexibilität in die Hand zu geben ist, eine universale Methode zu entwickeln, die alles kann und sie anschließend durch Überladungen »nach unten abzuspecken«.

Ein Beispiel: Angenommen, Sie haben eine Klasse entwickelt, die eine Methode zur Verfügung stellt, mit der man Kreise auf den Bildschirm malen kann. Sie nennen diese Methode *Circle*, und diese stellt in ihrer Universalversion folgende Fähigkeiten zur Verfügung:

```
Public Sub Circle(ByVal Xpos As Integer, ByVal YPos As Integer, ByVal XRadius As Integer, _
                 ByVal YRadius As Integer, ByVal StartAngle As Integer, ByVal EndAngle As Integer)
        'Hier steht der Code für Circle.
End Sub
```

Nun benötigen Sie die komplette Flexibilität dieser Methode aber nur in den seltensten Fällen. Sie müssen nicht jedes Mal X- *und* Y-Radius der Figur und noch seltener den Start- und Endwinkel des Kreises mit angeben. Eine abgespeckte Version könnte daher wie folgt aussehen:

```
Public Sub Circle(ByVal Xpos As Integer, ByVal YPos As Integer, ByVal Radius As Integer)
    Circle(Xpos, YPos, Radius, Radius, 0, 359)
End Sub
```

Auf den ersten Blick sieht es so aus, als würde sich die Funktion selber aufrufen. Macht sie aber nicht. Da die Signatur der verwendeten *Circle*-Methode (2. Zeile) nicht der eigenen Signatur (1. Zeile) entspricht, schaut der VB-Compiler, welche *Circle*-Methode in Frage kommt und verwendet in diesem Beispiel die zuerst verwendete.

TIPP: Aus Geschwindigkeitsgründen sollten Sie davon absehen, dass überladene Methoden quasi treppchenweise die jeweils nächst flexiblere Methode aufrufen, wenn Sie im Vorfeld wissen, dass solche Methoden beispielsweise in Schleifen hunderte von Malen im Laufe eines Programmlebens aufgerufen werden. Implementieren Sie eine universale Methode, die alles kann, und rufen Sie sie von jeder weiteren Version der Methode direkt auf – das spart Ausführungszeit in solchen Fällen.

Schlechtes Beispiel:

```
Public Sub Circle(ByVal Xpos As Integer, ByVal YPos As Integer, ByVal Radius As Integer)
    'Nicht so gut, wir "stolpern" quasi zum Ziel.
    Circle(Xpos, YPos, Radius, Radius)
End Sub

Public Sub Circle(ByVal Xpos As Integer, ByVal YPos As Integer, ByVal XRadius As Integer, _
                  ByVal YRadius As Integer)
    Circle(Xpos, YPos, XRadius, YRadius, 0, 359)
End Sub

Public Sub Circle(ByVal Xpos As Integer, ByVal YPos As Integer, ByVal XRadius As Integer, _
                 ByVal YRadius As Integer, ByVal StartAngle As Integer, ByVal EndAngle As Integer)
    'Hier steht der Code für Circle.
End Sub
```

Gutes Beispiel:

```
Public Sub Circle(ByVal Xpos As Integer, ByVal YPos As Integer, ByVal Radius As Integer)
    'Besser: direkter Sprung zur eigentlichen Methode.
    Circle(Xpos, YPos, Radius, Radius, 0, 359)
End Sub

Public Sub Circle(ByVal Xpos As Integer, ByVal YPos As Integer, ByVal XRadius As Integer, _
                  ByVal YRadius As Integer)
    Circle(Xpos, YPos, XRadius, YRadius, 0, 359)
End Sub
```

```
Public Sub Circle(ByVal Xpos As Integer, ByVal YPos As Integer, ByVal XRadius As Integer, _
              ByVal YRadius As Integer, ByVal StartAngle As Integer, ByVal EndAngle As Integer)
    'Hier steht der Code für Circle.
End Sub
```

Gegenseitiges Aufrufen von überladenen Konstruktoren

Problematischer wird es, wenn sich überladene Konstruktoren gegenseitig aufrufen sollen – betriebsbedingt gibt es dabei nämlich Einschränkungen. Mit dem Wissen des vorherigen Abschnittes starten Sie vielleicht rein aus dem Gefühl heraus den Versuch, das Beispiel auch bei den Konstruktoren folgendermaßen umzugestalten:

```
Public Sub New(ByVal RomanNumeral As String)

    'Statische Funktion aufrufen.
    Dim temp As Integer = ValueFromRomanNumeral(RomanNumeral)
    New(temp)

End Sub

Public Sub New(ByVal ArabicInt As Integer)

    'Einfach der Instanzvariablen zuweisen.
    myUnderlyingValue = ArabicInt

End Sub
```

Visual Basic quittiert diesen Aufruf mit einem simplen Syntaxfehler[7] und lässt den Aufruf ganz einfach nicht zu. Mit einem kleinen Trick lässt sich der Fehler zwar nicht abstellen, uns aber dem Ziel ein klein wenig näher kommen. Denn tauschen Sie die zweite Zeile in der ersten `Sub New` gegen die folgende

```
Me.New(temp)
```

aus, dann lautet die Fehlermeldung in der Fehlerliste:

```
Ein Aufruf an einen Konstruktor ist nur als erste Anweisung in einem Instanzenkonstruktor gültig.
```

Aha. Schauen wir einmal, was passiert, wenn wir die Zeilen zu einer zusammenfassen, etwa mit

```
Me.New(ValueFromRomanNumeral(RomanNumeral))
```

sodass sie dadurch in der ersten Zeile steht. Sie werden sehen, jetzt funktioniert es. Natürlich haben wir in diesem Fall nichts gewonnen – weder Programmierarbeit gespart noch die Ausführungsgeschwindigkeit erhöht. In anderen Fällen, bei komplexeren Klassen, kann das aber ganz anderes ausschauen.

Wichtig ist zu wissen, dass Sie genau spezifizieren, welches `New` Sie aufrufen. Prinzipiell gibt es bei jeder Klasse nämlich zwei Versionen von `New`, die aber nicht durch Überladen entstanden sind: Das `New` in Ihrer Klasse und das `New` der Klasse, von der Sie Ihre Klasse abgeleitet haben. Da Sie hier im

[7] Kleine nostalgische Randbemerkung: Das ist eine Fehlermeldung, die es schon beim Commodore 64 gab – der verfügte auch über ein Microsoft-Basic – und die sich bis heute gehalten hat!

Beispiel nicht explizit bestimmt haben, aus welcher Klasse Sie ableiten, hat Ihre Klasse automatisch von `Object` geerbt. Natürlich hat auch `Object` einen Konstruktor, und Sie können sowohl Methoden der Basisklasse als auch Methoden Ihrer Klasse aufrufen. Wenn Sie `Me` angeben, spezifizieren Sie Ihre Klasse; wenn Sie `MyBase` angäben, würden Sie damit die Basisklasse spezifizieren – in diesem Fall die Klasse `Object`. Mehr über das Vererben und über Aufrufe von Funktionen von Basisklassen erfahren Sie im nächsten Kapitel.

Hat jede Klasse einen Konstruktor?

Oh ja. Zwar gab es im ersten Beispiel dieses Kapitels eine Version, die nicht einmal über einen Standardkonstruktor[8] im Quellcode verfügte, aber für einen solchen Fall verlangt es das CTS,[9] dass der Compiler automatisch den Code generiert, um einen leeren Standardkonstruktor der Klasse hinzuzufügen. In diesem Standardkonstruktor wird auch der Code platziert, der bei der Initialisierung von Member-Variablen ausgeführt wird. Ein Beispiel: Lassen Sie uns noch einmal zu der ersten Version des Beispielprogramms zurückkehren, in der es noch keine Konstruktoren gab. Um die einzige Member-Variable mit einem Standardwert vorzuinitialisieren, könnten Sie die Zeile wie folgt abändern:

```
Public Class RomanNumerals

    Private myUnderlyingValue As Integer = 1000
    .
    .
    .
```

Die Frage: Wann wird diese Initialisierung eigentlich ausgeführt? Zur Beantwortung dieser Frage muss ich ein komplexes Themengebiet teilweise vorwegnehmen, denn Sie müssen wissen, wie Programme unter .NET letzten Endes ablaufen.

Wie im letzten Kapitel schon kurz angerissen, übersetzt der Compiler Ihr Programm nicht direkt in nativen Maschinencode, sondern in eine »Zwischensprache« namens *Intermediate Language* oder kurz *IL*. Die Rahmendaten Ihres Programms, beispielsweise Konstanten, definierte Attribute, aber auch Informationen über die Version werden in einem speziellen Datenbereich in der gleichen Datei abgelegt. Diese Daten nennt man in .NET *Metadaten*. Wenn Sie nun ein Programm starten, dann gibt es zu diesem Zeitpunkt natürlich noch nichts, was der Prozessor ausführen kann, denn er versteht ja – was Intel-Plattformen anbelangt – nur Pentium- bzw. x86-Code. Es muss also einen Mechanismus geben, der zwischen den Zeitpunkten von Programmstart und Programmausführung Ihr Programm in Maschinencode übersetzt – und dieses Werkzeug ist Bestandteil der CLR und nennt sich *JITter*.[10] Den »vorläufigen« IL-Code können Sie sich mit einem speziellen Werkzeug aus der .NET-Werkzeugsammlung anschauen – er offenbart alle Wahrheiten, auch solche, die der Compiler ohne Ihr Zutun hinzugefügt hat.

---

[8] Als Standardkonstruktor bezeichnet man einen parameterlosen Konstruktor, für den Fall VB also eine `Sub New`, der keine Parameter übergeben werden.

[9] Zur Wiederholung: Das **C**ommon **T**ype **S**ystem ist ein System, das strikte Typbindung und Codekonsistenz erfordert und damit die Common Language Runtime (CLR) zur Coderobustheit zwingt.

[10] Zur Auffrischung: JIT ist die Abkürzung von »Just in time«, auf deutsch etwa »genau rechtzeitig«.

Mit diesem Wissen bewaffnet, können Sie sich jedes kompilierte Visual Basic-Programm in seinem »IL-Zustand« anschauen und herausfinden, was der VB-Compiler daraus gemacht hat. Dabei sind gar nicht so sehr die einzelnen Befehle entscheidend, sondern die Metadaten, die auch Auskunft darüber geben, aus welchen Komponenten, Typen, Signaturen, etc. sich Ihr Programm zusammensetzt.

Zusätzliche Werkzeuge für .NET

Das Framework-SDK (*Software Development Kit*, etwa: *Software-Entwicklungs-Baukasten*), das selbstverständlich Bestandteil von Visual Studio .NET ist, beinhaltet ein paar zusätzliche Tools, die Sie nicht in der Programmgruppe von Visual Studio finden. Einige dieser zusätzlichen Werkzeuge lassen sich zudem nur als Konsolenanwendung verwenden.

Eine Übersicht über die zusätzlichen Werkzeuge (und darüber, wo sie sich oft mit Erfolg vor Ihnen verstecken) finden Sie, indem Sie aus dem Startmenü *Alle Programme* auswählen und in der Programmgruppe *Microsoft .NET Framework SDK 2.0* im Unterpunkt *Tools* das HTML-Dokument *Tools* anklicken.

In der vorliegenden Framework .NET-SDK-Version 2.0 finden Sie die Tools im Verzeichnis *%windir%\Microsoft.NET\Framework\v2.0.xxxx* - die letzten »X« geben dabei die Build-Nummer an[11]; »%windir%« bezeichnet das Basisverzeichnis von Windows.

Andere Werkzeuge, die Bestandteil von Visual Studio .NET sind, wie beispielsweise der Intermediate Language Disassembler (*ILDASM*), den wir als nächstes verwenden werden, sind ebenfalls nicht direkt von der Visual Studio-IDE aus zu erreichen (warum eigentlich nicht?). Sie finden ihn für Visual Studio 2005 im Verzeichnis *C:\Programme\Microsoft Visual Studio 8\SDK\v2.0\Bin* unter dem Namen *ILDASM.Exe* (natürlich nur dann, wenn Sie den vorgeschlagenen Pfad bei der VS-Installation übernommen haben). Mein Vorschlag: Legen Sie eine Verknüpfung zu diesem Programm auf dem Desktop Ihres Computers an, weil Sie ihn später noch des Öfteren gebrauchen werden.

Wichtig dabei ist zu wissen: Wenn Sie die Befehlszeile *Visual Studio 2005 .NET Command Prompt* verwenden, die Sie in der Programmgruppe Microsoft *Visual Studio 2005* des Startmenüs und dort in der Untergruppe *Visual Studio Tools* finden, müssen Sie sich um das Finden der richtigen Pfade keine Gedanken machen, da die *Path*-Variable dieser Befehlszeile entsprechend eingerichtet ist. Das alleinige Eingeben der Programmnamen führt dann in den meisten Fällen zum Ziel.

Nach diesem kurzen Exkurs in die Verzeichnistiefen des SDKs lassen Sie uns zurück zu unserem Vorhaben kommen. Es ist wichtig für das Nachvollziehen des folgenden Szenarios, dass Sie das erste Beispielprogramm (*RomNum_Net01*) nochmals geladen und erstellt haben, damit die Binärdateien (die eigentlich ausführbaren Programmdateien, die den IL-Code enthalten) vorhanden sind.

- Falls Sie sich nicht sicher sind, laden Sie das Projekt, und wählen Sie *Projektmappe neu erstellen* aus dem Menü *Erstellen*.
- Starten Sie anschließend den Intermediate-Language-Disassembler *ILDASM* (siehe vorheriger grauer Kasten).

[11] Die letzte Build-Version des RTMs (*Release to Manufacturing* – das ist die Version, die dann ausgeliefert wurde und in die Presswerke kam) trug die Versionsnummer 50727. Nach einem Service-Pack kann sich diese Version anpassen.

■ Wählen Sie *Öffnen* aus dem Menü *Datei* und im Dialog, der jetzt erscheint, die .EXE-Datei des Beispielprogramms. Sie finden die Programmdatei *RomNum_Net01.exe* im Projektverzeichnis und dort im Unterverzeichnis *\bin\Debug*.

HINWEIS: Sie werden im Debug-Verzeichnis zwei .EXE-Dateien finden – eine davon endet mit *vshost.exe*. Diese verwenden Sie bitte *nicht*. Bei ihr handelt es sich um ein kleines Hilfsprogramm, dass von der Visual Studio IDE benötigt wird, um die Erstellzeiten Ihrer Projekte zu beschleunigen, und um das vielfach gewünschte *Edit & Continue* zu ermöglichen, das es Ihnen gestattet, Änderungen an Ihrem Programm vorzunehmen, *während* Sie es debuggen. *Edit & Continue* steht Ihnen in dieser Version von Visual Studio übrigens nicht zur Verfügung, wenn dieses unter einem 64-Bit-Windows-Betriebssystem läuft.

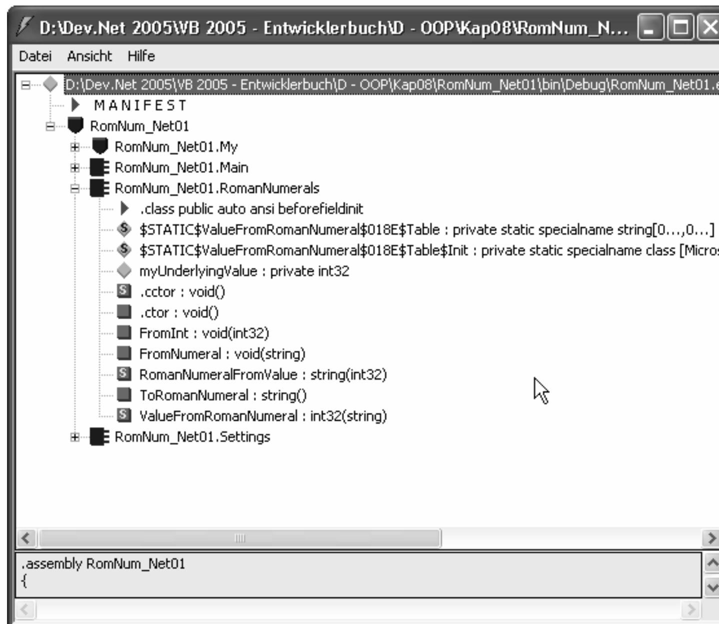

Abbildung 8.8: Die Metadaten der EXE-Datei erlauben die Ansicht der Programmstruktur im IL-Disassembler

■ Im Fenster, das anschließend erscheint, öffnen Sie den Zweig *RomNum_Net01* und den sich darunter befindlichen Zweig *RomNum_Net01.RomanNumerals* ebenfalls.

■ Aus dem Menü *Ansicht* wählen Sie die Option *Quellcodezeilen anzeigen*. Sie sollten anschließend ein Bild vor Augen haben, das etwa dem in Abbildung 8.8 entspricht.

In dieser Abbildung sehen Sie die Struktur des Programms. Sie finden Elemente wieder, die Sie selber bestimmt haben – beispielsweise die Funktionsnamen. Sie erkennen an den vorangestellten Symbolen, welche Strukturen statisch sind und welche nicht-statisch. Und Sie erkennen ebenfalls zwei Methoden (jeweils eine statische und eine nicht-statische), für die Sie nicht der unmittelbare Urheber waren – die Methoden `.ctor` und `.cctor`.

HINWEIS: An dieser Stelle können Sie übrigens sehr schön erkennen, was für ein Aufwand intern betrieben werden muss, um statische Variablen im VB6-Sinne zu ermöglichen. Zuständig dafür sind die beiden Einträge »$STATIC$ ...«, die das Array als klassenglobal-statisch (im CTS-Sinne) deklarieren, die Verwendung aber auf `ValueFromRomanNumeral` limitieren.

Fürs erste beschäftigen wir uns mit dem Standardkonstruktor der Klasse, mit `.ctor`. Obwohl Sie in der Klasse keinen VB-Code für eine `Sub New` ohne Parameter platziert hatten, ist er dennoch vorhanden. Ein Doppelklick offenbart, wozu er in unserem Beispiel dient (siehe Abbildung 8.9).

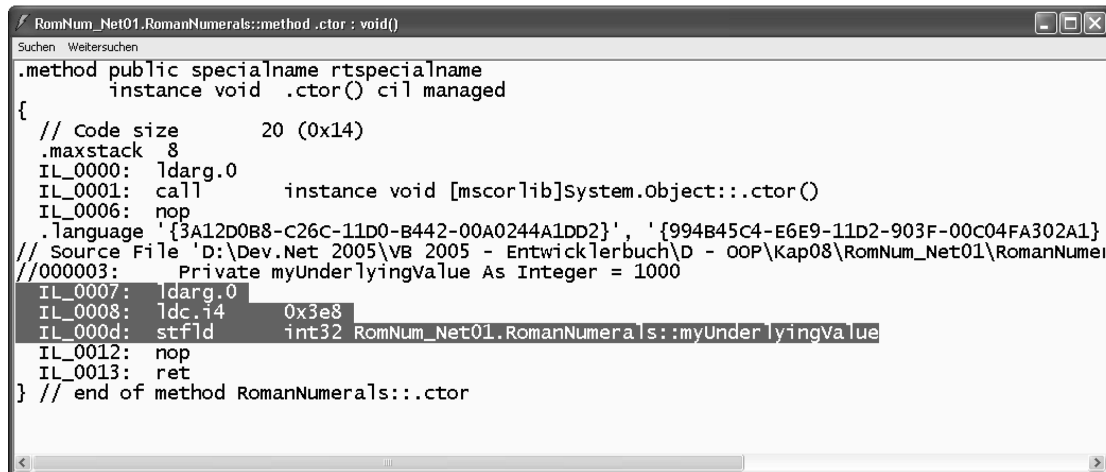

Abbildung 8.9: Der disassemblierte IML-Code des Standardkonstruktors

Auch ohne genau zu wissen, welche IL-Anweisung für welche Aufgabe zuständig ist, wird eines doch sofort deutlich: Im ersten Teil der Methode wird der Standardkonstruktor der Basisklasse aufgerufen. Anschließend wird die Variable `myUnderlyingValue` im zweiten Part der Methode mit dem Wert *1000* (hexadezimal *3E8*) initialisiert.

Dieses Beispiel zeigt, dass Ihnen der Visual Basic-Compiler eine ganze Menge an Arbeit abgenommen hat. Er hat dafür gesorgt,

- dass es einen Standardkonstruktor mit der Methode *.ctor* überhaupt gibt,
- dass innerhalb des Standardkonstruktors die Basisklasse *Object* (`call instance void [mscorlib]System.Object::.ctor()`) aufgerufen wird und
- dass alle Member-Variablen, falls erforderlich, innerhalb dieses Standardkonstruktors definiert – also mit den Werten »gefüllt« werden, so Sie dies bei deren Deklarierung angegeben haben.

Schauen wir uns zum Vergleich den Konstruktor des zweiten Beispiels an, denn hier haben Sie mit *Sub New* eine Konstruktorlogik selbst implementiert.

Wählen Sie dazu im IL-Disassembler *Öffnen* aus dem Menü *Datei* und im Dialog, der jetzt erscheint, die .EXE-Datei des zweiten Beispielprogramms (auch die sollte neu erstellt sein, damit nicht nur alle Änderungen, sondern die erforderlichen Unterverzeichnisse überhaupt vorhanden sind). Sie finden die Programmdatei *RomNum_Net02.exe* im entsprechenden Projektverzeichnis und dort wieder im Unterverzeichnis *\bin\Debug*.

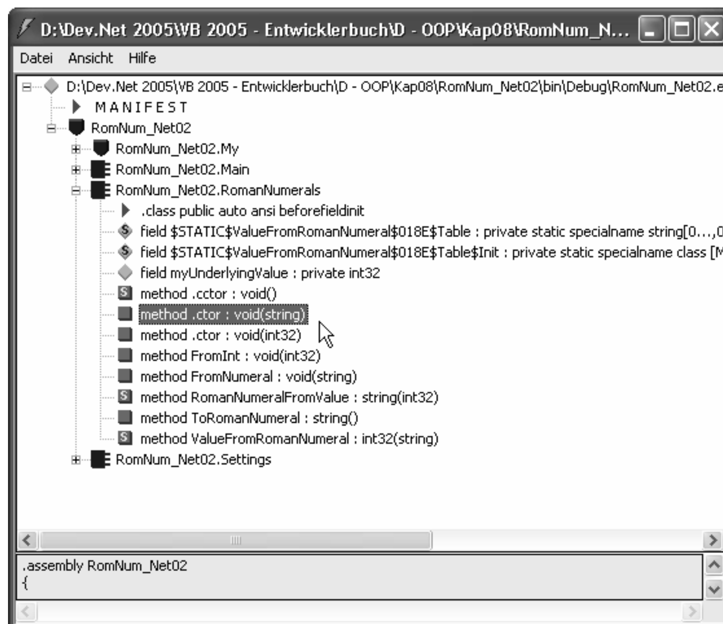

Abbildung 8.10: Die Strukturansicht der zweiten Version des Beispielprogramms

In Abbildung 8.10 können Sie erkennen, dass es keinen Standardkonstruktor, sondern nur die beiden Konstruktoren gibt, die als `Sub New` auch in der Klasse definiert waren.

Behalten Sie diese Tatsache bitte im Hinterkopf, denn sie wird uns zum Zeitpunkt der Auseinandersetzung mit der Klassenvererbung noch ein wenig beschäftigen.

HINWEIS: Schließen Sie Ihre Dateien, nachdem Sie sie im IML-Disassembler betrachtet haben; wenn Sie das Schließen der Dateien vergessen und anschließend eine neue Version kompilieren möchten, gibt Ihnen der Compiler eine Fehlermeldung aus, da die Dateien vom Disassembler gesperrt wurden und darum nicht durch neue Versionen ersetzt werden können.

Statische Konstruktoren und Variablen

Ich finde, es ist nun an der Zeit, sich ein wenig mit den Prozeduren zu beschäftigen, die den eigentlichen Job in unserem Beispielprogramm erledigen. Insbesondere die Funktion `ValueFromRomanNumeral` bedarf einer genaueren Betrachtung – hier gibt es nämlich etwas, das unnötig Rechenzeit verschleudert.

```
Public Shared Function ValueFromRomanNumeral(ByVal RomanNumeral As String) As Integer

    Static Table(6, 1) As String
    Dim locCount As Integer
    Dim locChar As Char
    Dim retValue As Integer
    Dim z1 As Integer, z2 As Integer
```

```
If RomanNumeral = "" Then
    Return 0
End If

'Tabelle zum Nachschlagen
'das kann man auch eleganter machen,
'aber für diese Version soll es reichen.
Table(0, 0) = "I" : Table(0, 1) = "1"
Table(1, 0) = "V" : Table(1, 1) = "5"
Table(2, 0) = "X" : Table(2, 1) = "10"
Table(3, 0) = "L" : Table(3, 1) = "50"
Table(4, 0) = "C" : Table(4, 1) = "100"
Table(5, 0) = "D" : Table(5, 1) = "500"
Table(6, 0) = "M" : Table(6, 1) = "1000"

locCount = 0

Do While locCount < Len(RomanNumeral)

    locChar = RomanNumeral.Chars(locCount)

    'VB6-Stil: If locCount < Len(RomanNumeral) - 1 Then
    If locCount < RomanNumeral.Length - 1 Then
        For z1 = 0 To 6
            If Table(z1, 0) = locChar Then Exit For
        Next z1
        For z2 = 0 To 6
            'VB6-Stil If Table(z2, 0) = Mid$(RomanNumeral, locCount + 1, 1) Then
            If Table(z2, 0) = RomanNumeral.Substring(locCount + 1, 1) Then
                Exit For
            End If
        Next z2
        If CInt(Table(z1, 1)) < CInt(Table(z2, 1)) Then

            'Stringfragment entfernen
            'VB6-Stil: RomanNumeral = Left$(RomanNumeral, locCount - 1) + Mid$(RomanNumeral,
            'locCount + 2)
            RomanNumeral = RomanNumeral.Substring(0, locCount - 1) + _
                        RomanNumeral.Substring(locCount + 2)
            retValue = retValue + (CInt(Table(z2, 1)) - CInt(Table(z1, 1)))
        Else
            For z2 = 0 To 6
                If Table(z2, 0) = locChar Then Exit For
            Next z2
            'VB6-Stil: retValue = retValue + Convert.ToInt32(Val(Table(z2, 1)))
            retValue += CInt(Table(z2, 1))
            locCount += 1
        End If
    Else
        For z2 = 0 To 6
            If Table(z2, 0) = locChar Then Exit For
        Next z2
        retValue += CInt(Table(z2, 1))
```

```
            locCount += 1
        End If
    Loop

    'VB6-Stil: ValueFromRomanNumeral = retValue : Exit Function
    Return retValue

End Function
```

Wie Sie aus dem Listing erkennen können (fett hervorgehoben), bedient sich die Routine einer Tabelle, um die Zuordnung von römischem Numerale und eigentlichem Wert vorzunehmen. Genau diese Tabelle ist aber Stein des Anstoßes, denn: Sie wird jedes Mal neu aufgebaut, wenn die Funktion aufgerufen wird. Da sie ohnehin nicht viel Speicher benötigt, wäre es ungleich besser, sie ein einziges Mal anzulegen und sie dann bis zum Ende des Programms im Speicher zu belassen.

Dazu bietet sich ein statisches (also ein als `Shared` deklariertes) Array an, das idealerweise im statischen Konstruktor der Klasse definiert wird. Der Umbau erfordert nur wenige Handgriffe.

BEGLEITDATEIEN: Sie finden die Codedateien zu diesem Beispiel im Verzeichnis *.\VB 2005 - Entwicklerbuch\D - OOP\Kap08\RomNum_Net03*. Öffnen Sie dort die entsprechende Projektmappen-Datei (*.sln*).

```
Public Class RomanNumerals

    Private myUnderlyingValue As Integer = 1000
    Private Shared Table(6, 1) As String

    Shared Sub New()

        'Tabelle zum Nachschlagen
        'Diesmal statisch (Shared) deklariert - das spart schon einmal Zeit.
        Table(0, 0) = "I" : Table(0, 1) = "1"
        Table(1, 0) = "V" : Table(1, 1) = "5"
        Table(2, 0) = "X" : Table(2, 1) = "10"
        Table(3, 0) = "L" : Table(3, 1) = "50"
        Table(4, 0) = "C" : Table(4, 1) = "100"
        Table(5, 0) = "D" : Table(5, 1) = "500"
        Table(6, 0) = "M" : Table(6, 1) = "1000"
    End Sub
```

Neben der Member-Variablen wird direkt unter der Klassendefinition auch das Array deklariert – und zwar als statisches (statisch auch im Sinne des CTS). Zusätzlich zu den schon vorhandenen Konstruktoren gibt es jetzt auch noch einen mit dem Zugriffsmodifizierer `Shared`, der bewirkt, dass aus dem Konstruktor ein statischer Konstruktor wird.

Die Frage, die sich aus dieser Modifizierung ergibt, lautet: *Was* ruft *wann* den statischen Konstruktor der Klasse auf? Der Zeitpunkt des Instanzierens der Klasse reicht ja bei weitem nicht aus, weil die statischen Funktionen `RomanNumeralFromValue` und `ValueFromRomanNumeral` nicht funktionierten, wenn nicht zuvor mindestens eine Klasseninstanz erstellt würde.

Um das Verhalten zu verdeutlichen, müssen wir die Debug-Fähigkeiten von Visual Studio bemühen. Visual Studio erlaubt das Setzen von Haltepunkten innerhalb des Programmcodes. Trifft das Programm bei seinem Ablauf auf einen solchen Haltepunkt, wird die Programmausführung unterbrochen, und die Visual Studio-IDE wechselt in den Debug-Modus, in dem Codezeilen Schritt für Schritt ausgeführt werden können. Genau das ist der Plan für die nächsten Schritte.

In der Klasse `Main` fügen wir dazu eine weitere Instanzierungsanweisung ein, um die Unterschiede beim Verhalten der Konstruktoren im Vergleich zueinander beobachten zu können (beide Stellen sind im Listing fett markiert). Außerdem entzerren wir die Codezeile, die die Zeichenkette von der Tastatur einliest, damit wir es beim Debuggen einfacher haben. Es ergibt sich folgendes neues Listing für das Hauptprogramm.

```
Public Class Main

    Shared Sub Main()

        'Text ausgeben; Anwender zur Eingabe auffordern
        Console.Write("Geben Sie eine Zahl zwischen 1 und 3.999 ein: ")

        'Zahl als Text einlesen, mit der statischen Methode Parse
        'in Integer umwandeln...
        Dim locInt As Integer = Integer.Parse(Console.ReadLine)

        'und das Ergebnis der Klasseninstanz zuweisen.
        'An dieser Stelle befindet sich der Haltepunkt!
        Dim locRomanNumeral As New RomanNumerals(locInt)

        'Das römische Literale ausgeben, das in der Klasseninstanz gespeichert ist
        Console.WriteLine("Entspricht dem römischen Numerale " & locRomanNumeral.ToRomanNumeral)

        'Nur zum Testen.
        Dim locOtherRomanNumeral As New RomanNumerals("XXXIV")  ' < < < <

        'Dies nur noch, damit nicht alles sofort wieder verschwindet
        Console.WriteLine()
        Console.WriteLine("Return drücken zum Beenden...")
        Console.ReadLine()

    End Sub

End Class
```

Um die benötigten Haltepunkt zu setzen (einen im Hauptprogramm, einen weiteren im statischen Konstruktor der Klasse `RomanNumerals`), fahren Sie zunächst mit dem Cursor auf folgende Zeile (fett markiert) der Moduls *Main.vb*.

```
        'und das Ergebnis der Klasseninstanz zuweisen.
        'An dieser Stelle befindet sich der Haltepunkt!
        Dim locRomanNumeral As New RomanNumerals(locInt)
```

und drücken die Taste **F9**. Vor der Zeile erscheint ein kleiner, roter Ball, der den Haltepunkt kennzeichnet. Gleichzeitig wird die Zeile rot hervorgehoben. Öffnen Sie nun im Editor die Codedatei *RomanNumerals.vb*, und platzieren Sie einen weiteren Haltepunkt auf der folgenden Codezeile:

```
Public Class RomanNumerals

    Private myUnderlyingValue As Integer = 1000
    Private Shared Table(6, 1) As String

    Shared Sub New()

        'Tabelle zum Nachschlagen
        'Diesmal statisch (Shared) deklariert - das spart schon einmal Zeit.
        'Hier liegt der Haltepunkt:
        Table(0, 0) = "I" : Table(0, 1) = "1"
        Table(1, 0) = "V" : Table(1, 1) = "5"
        Table(2, 0) = "X" : Table(2, 1) = "10"
        Table(3, 0) = "L" : Table(3, 1) = "50"
        Table(4, 0) = "C" : Table(4, 1) = "100"
        Table(5, 0) = "D" : Table(5, 1) = "500"
        Table(6, 0) = "M" : Table(6, 1) = "1000"

    End Sub
```

Starten Sie nun das Programm, indem Sie *Starten* aus dem Menü *Debuggen* anklicken. Sie sehen kurze Zeit später ein Bild, etwa wie in Abbildung 8.11 zu sehen.

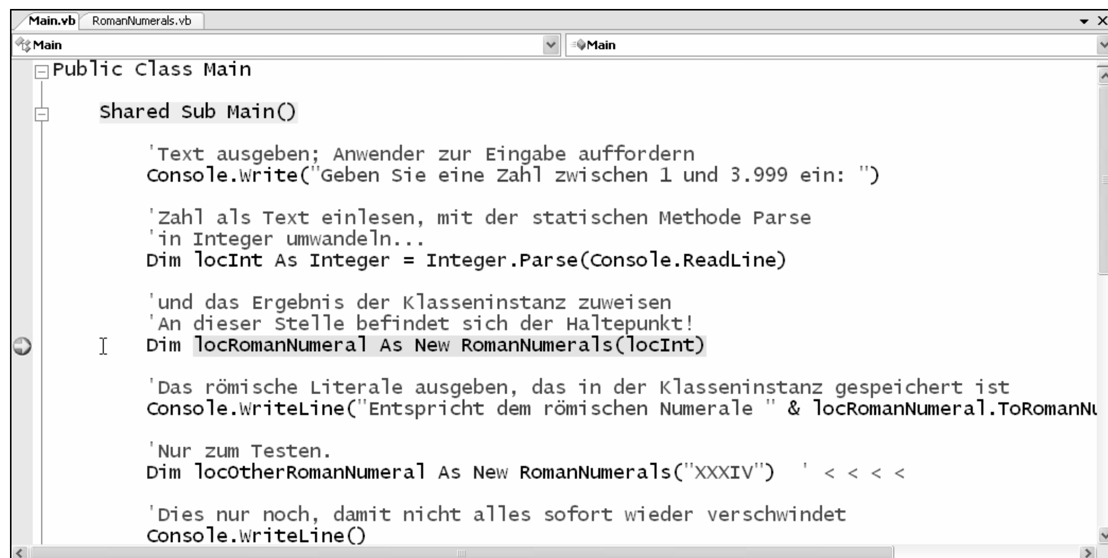

Abbildung 8.11: Ein Haltepunkt bringt das Programm in den Einzelschrittmodus

Ihr Projekt befindet sich jetzt im so genannten Einzelschrittmodus, mit dem Sie die Codezeilen Schritt für Schritt ausführen und sofort nachvollziehen können, welchen »Weg« das Programm

nimmt und was mit jedem Schritt passiert. Drücken Sie im Folgenden für jeden Programmschritt die Taste **F10**.[12] Sie werden feststellen, dass das Programm, nachdem es die Zeile

```
Dim locRomanNumeral As New RomanNumerals(locInt)
```

erreicht hat, zunächst in den statischen Konstruktor springt, dort die Arrayelemente initialisiert, bevor es anschließend erst in den (nicht-statischen) Konstruktor

```
Public Sub New(ByVal ArabicInt As Integer)
```

gelangt.

Wenn Sie nun weiter durch das Programm »steppen«, gelangen Sie irgendwann zur zweiten Klasseninstanzierung,

```
'Nur zum Testen.
Dim locOtherRomanNumeral As New RomanNumerals("XXXIV")  ' < < < <
```

und Sie werden feststellen: Der statische Konstruktor der Klasse *RomanNumerals* wird dieses Mal nicht aufgerufen.

Der statische Konstruktor wird also nur ein einziges Mal aufgerufen, und zwar zu dem Zeitpunkt, zu dem die Klasse das erste Mal in Ihrer Applikation verwendet wird. Damit nutzen Sie statische Konstruktoren immer dann, wenn Sie grundlegende Dinge für eine Klasse vorzubereiten haben, die nur ein einziges Mal erledigt werden müssen.

HINWEIS: Der statische Konstruktor wird auch aufgerufen, bevor Sie das erste Mal eine *statische* Funktion der Klasse verwenden.

Eigenschaften

Das Beispielprogramm ist durch die Konstruktoren ungleich flexibler und einfacher handhabbar geworden. Durch das Auflösen der vormals zwei Funktionen in Konstruktorroutinen gibt es allerdings jetzt einen Nachteil: In der aktuellen Version gibt es keine Möglichkeit, den Inhalt einer Klasseninstanz im Nachhinein zu verändern.

Es gäbe natürlich nun die Möglichkeit, die Funktionen wieder einzubauen. Funktionen sind allerdings auf eine Richtung des Datenflusses limitiert. Sie können entweder nur dazu herhalten, den Instanzwert zu verändern *oder* ihn abzufragen (OK, eine Funktion könnte ihn auch gleichzeitig verändern *und* abfragen, aber das wäre natürlich totaler Unfug, denn die Eingabe entspräche immer der Ausgabe).

Sie könnten sich in solchen Fällen dadurch helfen, dass Sie eine Funktion beispielsweise namens `GetValue` und eine weitere Funktion namens `SetValue` definieren. Moderne Programmiersprachen wie Visual Basic oder C# kennen dazu einen viel eleganteren Weg über so genannte Eigenschaftenprozeduren (`Property`-Prozeduren).

[12] Mit **F11** führen Sie Codezeilen schrittweise aus und springen bei einem Methodenaufruf in die entsprechende Prozedur. Mit **F10** führen Sie Codezeilen ebenfalls schrittweise aus; Prozeduren werden jedoch als Ganzes abgearbeitet, also wie eine einzige Codezeile behandelt. Da wir einen weiteren Haltepunkt in *Sub Shared…* gesetzt haben, unterbricht die Ausführung dort trotz **F10** (Methodenaufruf »als Ganzes«).

Wenn Sie schon längere Zeit mit Visual Basic (egal, ob mit .NET oder 6.0) programmiert haben, dann haben Sie Eigenschaften natürlich längst kennen gelernt. Mit Hilfe von Eigenschaften können Sie in der Regel bestimmte Zustände von Objekten abfragen *und* verändern. Möchten Sie beispielsweise wissen, ob die Schaltfläche eines Formulars anwählbar ist, verwenden Sie die Eigenschaft in Abfrageform, etwa wie hier:

```
If Schaltfläche.Enabled Then TuWas
```

Oder Sie legen die Eigenschaft eines Objektes fest, etwa wie mit der folgenden Zeile:

```
Schaltfläche.Enabled = false    ' Abschließen, kommt keiner mehr 'ran
```

Soweit die Funktionsweise aus der Sicht desjenigen, der das Objekt später verwendet. Die viel interessantere Frage lautet: Wie statten Sie Ihre eigenen Klassen mit Eigenschaften aus?

Visual Basic stellt Ihnen zu diesem Zweck, wie schon erwähnt, Eigenschaftenprozeduren zur Verfügung. Eine Eigenschaft wird in Visual Basic folgendermaßen definiert:

```
Property EineEigenschaft() As Datentyp

    Get
        Return New Datentyp
    End Get

    Set(ByVal Value As Datentyp)
        mach_irgendwas_mit = Value
    End Set

End Property
```

Wenn Sie diese Eigenschaft in einer Klasse implementieren, können Sie sie bei instanzierten Objekten dieser Klasse auf folgende Weise verwenden:

Zuweisen von Eigenschaften

Mit der Anweisung

```
Object.EineEigenschaft = Irgendetwas
```

weisen Sie der Eigenschaft `EineEigenschaft` des Objektes einen Wert zu. Sie können im *Set*-Accessor[13] (`Set(ByVal Value as Datentyp)`) der Eigenschaftenprozedur mit `Value` auf das Objekt zugreifen, das sich in `Irgendetwas` befindet. Nur der *Set*-Teil der Eigenschaftenprozedur wird in diesem Fall ausgeführt.

Ermitteln von Eigenschaften

Umgekehrt können Sie mit der Anweisung

```
Irgendetwas=Object.EineEigenschaft
```

den Inhalt der Eigenschaft wieder auslesen. In diesem Fall wird nur der *Get*-Accessor der Eigenschaftenprozedur ausgeführt, die das Ergebnis mit `Return` zurückliefert.

[13] Etwa: »Zugreifer«.

Damit haben Sie jetzt das erforderliche Rüstzeug, um eine Eigenschaftenprozedur in die nächste Version des Beispiels einzubauen.

BEGLEITDATEIEN: Sie finden die Codedateien zu diesem Beispiel im Verzeichnis *.\VB 2005 - Entwicklerbuch\D - OOP\Kap08\RomNum_Net04*. Öffnen Sie dort die entsprechende Projektmappen-Datei (*.sln*).

Sie finden folgende Codezeilen im Klassencode von *RomanNumerals*:

```
Property UnderlyingValue() As Integer

    Get
        Return myUnderlyingValue
    End Get

    Set(ByVal Value As Integer)
        myUnderlyingValue = Value
    End Set

End Property
```

TIPP: Die Visual Basic-Editor unterstützt Sie beim Erstellen von Eigenschaftenprozeduren. Wenn Sie die Eigenschaftendefinition der Eigenschaft in den Editor tippen und am Ende der Zeile die Eingabetaste drücken, fügt Visual Studio den vollständigen Codeblock für das Grundgerüst der Eigenschaftenprozedur für Sie ein.

In der `Main`-Klasse können Sie die neue Eigenschaft verwenden, etwa wie mit dem folgenden modifizierten Code von *Class Main*:

```
Public Class Main

    Shared Sub Main()

        'Text ausgeben; Anwender zur Eingabe auffordern
        Console.Write("Geben Sie eine Zahl zwischen 1 und 3.999 ein: ")

        'Zahl als Text einlesen, mit der statischen Methode Parse
        'in Integer umwandeln...
        Dim locInt As Integer = Integer.Parse(Console.ReadLine)

        'und das Ergebnis der Klasseninstanz zuweisen
        'An dieser Stelle befindet sich der Haltepunkt!
        Dim locRomanNumeral As New RomanNumerals(locInt)

        'Das römische Literale ausgeben, das in der Klasseninstanz gespeichert ist
        Console.WriteLine("Entspricht dem römischen Numerale " & locRomanNumeral.ToRomanNumeral)

        'Nur für den Abstand.
        Console.WriteLine()

        'Wert mit der neuen Eigenschaft verändern.
        locRomanNumeral.UnderlyingValue = 200
```

```
        'und neues Ergebnis ausgeben
        Console.WriteLine("und 200 entspricht dem römischen Numerale " & locRomanNumeral.ToRomanNumeral)

        'Dies nur noch, damit nicht alles sofort wieder verschwindet.
        Console.WriteLine()
        Console.WriteLine("Return drücken zum Beenden...")
        Console.ReadLine()

    End Sub

End Class
```

Natürlich erkennt auch IntelliSense in der Visual Studio-IDE sofort, dass Sie eine Eigenschaft in der Klasse `RomanNumerals` eingebaut haben. Sobald Sie nach der Eingabe des Objektnamens `locRomanNumerals` den Punkt auf der Tastatur drücken, können Sie die neue Eigenschaft in der Liste erkennen (siehe Abbildung 8.12).

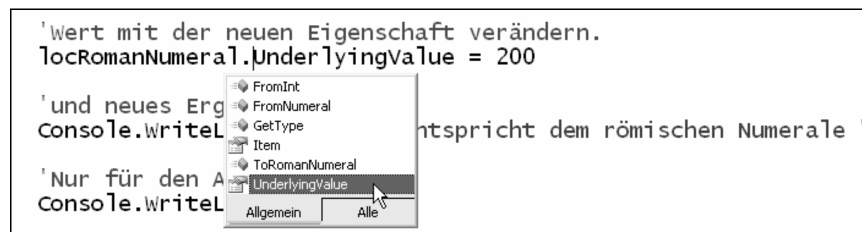

Abbildung 8.12: IntelliSense »lernt« neue Eigenschaften sofort und bietet sie Ihnen im Bedarfsfall direkt an

Nur-Lesen und Nur-Schreiben-Eigenschaften

Sie haben die Möglichkeit, Eigenschaften auf den Datenfluss in nur eine Richtung zu begrenzen.

Falls Sie möchten, dass eine bestimmte Eigenschaft nur gelesen werden kann, ist Folgendes zu tun:

- Sie verwenden den Modifizierer `ReadOnly` vor der `Property`-Definition.
- Und Sie sorgen dafür, dass nur ein *Get*-Accessor in der Eigenschaftenprozedur implementiert wird.

Beispiel für eine Nur-Lesen-Eigenschaft:

```
ReadOnly Property NurLesen() As Integer

    Get
        Return 5
    End Get

End Property
```

TIPP: Beginnen Sie direkt mit dem `ReadOnly`-Schlüsselwort, wenn Sie die Eigenschaftendefinition der Eigenschaft in den Editor tippen. Sobald Sie am Ende der Zeile die Eingabetaste drücken, fügt Visual Studio den vollständigen Codeblock für das Grundgerüst der Nur-Lesen-Eigenschaft ein und verzichtet auf den *Set*-Accessor.

Falls Sie möchten, dass eine bestimmte Eigenschaft nur geschrieben werden kann, ist Folgendes zu tun:

- Sie verwenden den Modifizierer `WriteOnly` vor der Property-Definition.
- Sie sorgen dafür, dass nur ein *Set*-Accessor in der Eigenschaftenprozedur implementiert wird.

Beispiel für eine Nur-Schreiben-Eigenschaft:

```
WriteOnly Property NurSchreiben() As Integer

    Set(ByVal Value As Integer)
        MachEtwasMit(Value)
    End Set

End Property
```

TIPP: Beginnen Sie direkt mit dem `WriteOnly`-Schlüsselwort, wenn Sie die Eigenschaftendefinition der Eigenschaft in den Editor tippen. Sobald Sie am Ende der Zeile die Eingabetaste drücken, fügt Visual Studio den vollständigen Codeblock für das Grundgerüst der Nur-Schreiben-Eigenschaft ein und verzichtet auf den *Get*-Accessor.

Eigenschaften mit Parametern

In Visual Basic können Eigenschaften, wie Funktionen, beliebig viele Parameter übernehmen. Die Parameter werden in der Eigenschaftenprozedur genauso wie bei Funktionen eingebunden, und auf die Parameter lässt sich dann ebenfalls genauso Zugriff darauf nehmen.

Ein Beispiel. Angenommen Sie haben eine Eigenschaftenprozedur etwa wie die folgende definiert,

```
Property EigenschaftMitParametern(ByVal Par1 As Integer, ByVal Par2 As String) As Integer

    Get
        If Par1 = 0 Then
            Return 10
        Else
            Return 20
        End If
    End Get

    Set(ByVal Value As Integer)
        If Par2 = "Klaus" Then
            'MachIrgendEtwas
        ElseIf Par1 = 20 Then
            'MachIrgendetwasAnderes
        End If
    End Set

    End Property
```

dann können Sie sie mit beispielsweise folgenden Anweisungen zuweisen bzw. abfragen:

```
'Eigenschaft mit Parametern setzen.
locRomanNumeral.EigenschaftMitParametern(10, "Klaus") = 5

'Eigenschaft mit Parametern abfragen.
If locRomanNumeral.EigenschaftMitParametern(20, "Test") = 20 Then
    'Mach Irgendetwas
End If
```

HINWEIS: Im Gegensatz zu Visual Basic 6.0 werden Wertetypen-Parameter als Wert und nicht als Referenz übergeben. Änderten Sie einen Wert innerhalb einer Eigenschaftenprozedur in Visual Basic 6.0, so veränderte sich auch der Wert der Variablen im aufrufenden Code (es sei denn, er war explizit durch Einklammern vor dem Überschreiben geschützt). In Visual Basic .NET übergeben Sie standardmäßig den Wert an eine Eigenschaft – Sie arbeiten also in jedem Fall mit einer Kopie der übergebenden Variablen.

Eine Differenzierung mit `Set` und `Let` wie in Visual Basic 6.0 gibt es übrigens in Visual Basic .NET nicht mehr. Da alles von `Object` abgeleitet ist und dadurch im Grunde genommen ausschließlich Objekte manipuliert werden, ist auch das `Let` überflüssig geworden, und alles müsste mit `Set` manipuliert werden – und dann kann man `Set` auch direkt weglassen.

WICHTIG: Eigenschaften mit Parametern sollten Sie nur in Ausnahmefällen verwenden, da sie eine besondere Eigenart von Visual Basic sind. Im Gegensatz zu Visual Basic kennt C# zwar auch Eigenschaften, kann aber nur auf parameterlose Eigenschaften oder Default-Eigenschaften zugreifen (mehr dazu im ▶ Abschnitt »Default-Eigenschaften« auf Seite 252). Wenn Sie also Klassen im Team oder für die breite Öffentlichkeit entwickeln, sollten Sie auf Eigenschaften mit Parametern nach Möglichkeit ganz verzichten.

Überladen von Eigenschaften

Eigenschaften lassen sich wie Funktionen überladen. Zum Einsatz kommt dieses Prinzip fast ausschließlich bei Default-Eigenschaften (siehe nächster Abschnitt). Wie bei »normalen« Funktionen kann nur die Eigenschaftensignatur (die Reihenfolge, die Typen und die Anzahl der übergebenden Parameter) nicht aber der Rückgabetyp zur Unterscheidung herhalten. Daher ist die Überladung von Eigenschaften auch nur dann möglich, wenn mindestens eine Eigenschaftenvariation Parameter entgegennimmt.

Ein Beispiel für das Überladen von Eigenschaften:

```
Property Überladung() As Integer
    Get
       'Hier der Code für die Ermittlung der Eigenschaft.
    End Get
    Set(ByVal Value As Integer)
        'Hier der Code für die Zuweisung.
    End Set
End Property
```

```
Property Überladung(ByVal Par1 As Integer) As Integer
    Get
        'Hier der Code für die Ermittlung der Eigenschaft.
    End Get
    Set(ByVal Value As Integer)
        'Hier der Code für die Zuweisung.
    End Set
End Property
Property Überladung() As String
    Get
        'Geht nicht, da sich diese Eigenschaft...
    End Get
    Set(ByVal Value As String)
        'nur durch den Rückgabetyp von der ersten unterscheidet.
    End Set
End Property
```

Statische Eigenschaften

Visual Basic sieht auch statische Eigenschaften vor. Genau wie bei statischen Funktionen sind statische Eigenschaften direkt über die Klasse und nicht über die Klasseninstanz definiert. Das bedeutet, dass statische Eigenschaften Funktionalitäten bereitstellen sollten, die für alle Objekte dieser Klasse gelten und nicht für eine bestimmte Objektinstanz.

Das beste Beispiel für eine statische Eigenschaft ist die `Now`-Eigenschaft von `DateTime`. Sie liefert die aktuelle Uhrzeit zurück und ist natürlich von den Eigenschaften einzelner `DateTime`-Instanzen völlig unabhängig. Statische Eigenschaften werden, ebenfalls wie statische Funktionen, mit dem `Shared`-Schlüsselwort deklariert. Ein Beispiel für eine statische Eigenschaft finden Sie im folgenden Abschnitt.

Default-Eigenschaften (Standardeigenschaften)

Default-Eigenschaften (Standardeigenschaften) haben in Visual Basic 6.0 eine erleichternde Funktion für schreibfaule Entwickler gehabt. Mit Hilfe einer Default-Eigenschaft konnten sie bestimmen, welche Eigenschaft verwendet wird, wenn Sie beim Zugriff auf ein Objekt gar keine Eigenschaft verwendet haben. Das CTS erlaubt derartige Typunsicherheiten nicht – denn bei der Zuweisung beispielsweise von

```
Dim EineTextBox as TextBox
Dim einObjekt as Object
.
.
.
einObject = EineTextBox
```

ist natürlich nicht sichergestellt, ob Sie die Textbox selbst oder das Ergebnis der Default-Eigenschaft der Textbox an `einObjekt` zuweisen wollen. Ausnahmen bilden dabei parametrisierte Eigenschaften, da durch die Signatur der Eigenschaft deutlich wird, dass Sie nicht das Objekt selbst, sondern das

Resultat einer parametrisierten Eigenschaft zurückliefern wollen.[14] In diesem Fall ergibt das auch Sinn, denn:

Stellen Sie sich vor, Sie entwickeln eine Array-Klasse, die verschiedene Elemente verwaltet. Gäbe es keine Default-Eigenschaft, müssten Sie ein Element auf folgende Weise abfragen:

```
'Index setzen.
ArrayKlasse.Index = 5
'Element abfragen.
MachEtwasMit = ArrayKlasse.Item
```

Das wäre natürlich äußerst umständlich. Einfacher wird es so, wie es auch tatsächlich funktioniert, nämlich durch die Spezifizierung des Indexes und die Abfrage in einer Zeile. In Visual Basic ist das kein Problem durch eine Eigenschaft mit einem Parameter, etwa wie folgt:

```
MachEtwasMit = ArrayKlasse.Item(5)
```

Noch einfacher wird es, wenn die Eigenschaft `Item` in diesem Beispiel zur Default-Eigenschaft erklärt wird. Dann brauchen Sie den Eigenschaftennamen nämlich gar nicht mehr angeben, und die folgende Zeile wäre ausreichend:

```
MachEtwasMit = ArrayKlasse(5)
```

Die entsprechende Definition für die `Item`-Eigenschaft sähe in diesem Fall folgendermaßen aus:

```
Default Public Property Item(ByVal Index As Integer) As Integer

    Get
        Return InternesArray(Index)
    End Get

    Set(ByVal Value As Integer)
        InternesArray(Index) = Value
    End Set

End Property
```

WICHTIG: Im Gegensatz zu Visual Basic 6 gibt es in Visual Basic .NET keine parameterlosen Default-Eigenschaften. Ebenfalls gut zu wissen: Default-Eigenschaften können nicht als statisch deklariert werden. Das hindert Sie aber natürlich nicht daran, eine Eigenschaft zu implementieren, die sich statisch verhält. Die folgende Beispielimplementierung macht das deutlich.

Mit diesem Wissen können wir das Beispielprogramm um eine weitere Eigenschaft ergänzen – ich nenne sie `BaseValues`. Mit der `BaseValues`-Eigenschaft können Sie die Elemente der Konvertierungstabelle abfragen. Sie übergeben ihr den Index als Wert zwischen 0 und 6, und die Eigenschaft liefert das entsprechende römische Numerale für eine Basiszahl (z.B. »I« oder »X«) zurück. Über den Sinn dieser Prozedur möchte ich nicht lamentieren – ich gebe zu, dass sie programmtechnisch keinen denkbaren Nutzen hat. Vielleicht benötigt aber einer Ihrer Team-Mitarbeiter diese Tabelle für das

[14] Ähnlich wie bei Überladungen, bei denen auch nur durch die Signaturen unterschieden wird, welche der mehreren vorhandenen Funktionen gemeint ist.

Implementieren einer eigenen Funktionalität, und auf diese Weise kann er ohne Probleme auf die benötigten Daten zurückgreifen.

Die entsprechenden Änderungen finden sich ebenfalls schon in der *RomanNET04*-Version, und die statische Default-Eigenschaft sieht folgendermaßen aus:

```
Default ReadOnly Property Item(ByVal Index As Integer) As String

    Get
        Return Table(Index,0)
    End Get

End Property
```

Diese kleine Eigenschaft verfügt nun über alle besprochenen Komponenten. Sie kann »nur lesen«, verfügt demzufolge auch nur über einen *Get*-Accessor. Sie greift auf eine statische Variable zu, obwohl sie selbst nicht als statisch definiert ist. Und sie ist die Default-Eigenschaft der Klasse, was bedeutet, dass Sie sie direkt mit dem Instanznamen abfragen können. Der einzige Unterschied zu einer echten statischen Eigenschaft: Sie sind nicht in der Lage, auf die Elemente mit dem Klassennamen, sondern nur mit dem Instanznamen zuzugreifen – aber was soll's!

Die Verwendung dieser Eigenschaft im Hauptprogramm demonstriert ihren Umgang:

```
Public Class Main
    Shared Sub Main()
        'Text ausgeben; Anwender zur Eingabe auffordern.
        Console.Write("Geben Sie eine Zahl zwischen 1 und 3.999 ein: ")

        'Instanz der Klasse RomanNumerals bilden UND
        'Zahl als Text einlesen, mit der statischen Methode Parse in Integer
        'umwandeln und das Ergebnis der Klasseninstanz zuweisen.
        Dim locRomanNumeral As New RomanNumerals(Integer.Parse(Console.ReadLine))

        'Das römische Literale ausgeben, das in der Klasseninstanz gespeichert ist.
        Console.WriteLine("Entspricht dem römischen Numerale " & locRomanNumeral.ToRomanNumeral)

        'Nur für den Abstand.
        Console.WriteLine()

        'Wert mit der neuen Eigenschaft verändern
        locRomanNumeral.UnderlyingValue = 200

        'und neues Ergebnis ausgeben.
        Console.WriteLine("und 200 entspricht dem römischen Numerale " & locRomanNumeral.ToRomanNumeral)

        'Nur für den Abstand.
        Console.WriteLine()

        'Hier wird auf die Default-Eigenschaft zugegriffen.
        For locCount As Integer = 0 To 6
            Console.WriteLine("Element Nr. {0} entspricht römischen Numeral {1}", locCount, _
                              locRomanNumeral(locCount))
        Next
```

```
        'Dies nur noch, damit nicht alles sofort wieder verschwindet.
        Console.WriteLine()
        Console.WriteLine("Return drücken zum Beenden...")
        Console.ReadLine()
    End Sub
End Class
```

Wenn Sie das Programm starten, verhält es sich wie folgt:

```
Geben Sie eine Zahl zwischen 1 und 3.999 ein: 2557
Entspricht dem römischen Numerale MMDLVII

und 200 entspricht dem römischen Numerale CC

Element Nr. 0 entspricht römischen Numeral I
Element Nr. 1 entspricht römischen Numeral V
Element Nr. 2 entspricht römischen Numeral X
Element Nr. 3 entspricht römischen Numeral L
Element Nr. 4 entspricht römischen Numeral C
Element Nr. 5 entspricht römischen Numeral D
Element Nr. 6 entspricht römischen Numeral M

Return drücken zum Beenden...
```

Öffentliche Variablen oder Eigenschaften – eine Glaubensfrage?

Jetzt haben Sie schon so viel über Eigenschaften erfahren – vielleicht fragen Sie sich, wieso man sie anstelle von einfachen öffentlichen Member-Variablen einsetzen sollte. Solange, wie Sie Eigenschaften in einer Klasse nur benötigen, um irgendwelche Werte zu speichern, aber beim Abfragen oder Setzen dieser Werte nichts Weiteres passieren muss, wären öffentliche Variablen eigentlich ausreichend.

Die Klasse

```
Class EineEigenschaft
    Public DieEigenschaft As Integer
End Class
```

erfüllt zunächst nämlich den gleichen Zweck wie die folgende Klasse,

```
Class EineWeitereEigenschaft

    Private myDieEigenschaft As Integer

    Public Property DieEigenschaft() As Integer
        Get
            Return myDieEigenschaft
        End Get
        Set(ByVal Value As Integer)
            myDieEigenschaft = Value
        End Set
    End Property
End Class
```

die natürlich sehr viel mehr Schreibarbeit erfordert. Prinzipiell ist das richtig. Und dennoch ist die zweite Methode der ersten Methode vorzuziehen, denn es geht bei der objektorientierten Programmierung um Datenkapselung. `myDieEigenschaft` ist der eigentliche Datenträger dieser Klasse, und es gilt das Innehaben seiner Verwaltung bis zum Äußersten zu verteidigen. Bei einfachen Sachen mag die erste Alternative noch die bessere, da schnellere sein. Aber wenn Ihre Programme komplexer werden, wird sich der zusätzliche Aufwand für die zweite Methode schnell bezahlt machen. Ihre Datenstruktur bleibt in jedem Fall unberührt und unabhängig, und diese Vorgehensweise garantiert, dass Sie die jederzeit die volle Kontrolle über Ihre Daten behalten. Denken Sie nämlich daran, dass Sie im Set-Accessor auch die Möglichkeit haben, auf eine Wertezuweisung mit Programmcode Einfluss zu nehmen. Sie könnten beispielsweise für die möglichen Wertzuweisungen eine Bereichsüberprüfung durchführen, und bei Überschreitung entweder die Maximal- oder Minimalwerte erzwingen:

```
Class EineWeitereEigenschaft

    Private myDieEigenschaft As Integer

    Public Property DieEigenschaft() As Integer
        Get
            Return myDieEigenschaft
        End Get
        Set(ByVal Value As Integer)
            If Value < 0 Then Value = 0
            If Value > 100 Then Value = 100
            myDieEigenschaft = Value
        End Set
    End Property
End Class
```

Ein weiteres wichtigeres Argument ist das Ersetzen von Eigenschaften durch die so genannte Polymorphie beim Vererben von Klassen, das ich im nächsten Kapitel beschreiben werde. Eine einmal als öffentlich deklarierte Variable bleibt für alle Zeiten öffentlich. Sie können in vererbten Klassen keine zusätzliche Steuerung hinzufügen. Haben Sie hingegen Ihre Daten nur durch Eigenschaftenprozeduren nach außen offen gelegt, können Sie zu einem späteren Zeitpunkt noch zusätzliche Regeln (Bereichsabfragen, Fehler abfangen) hinzufügen. Sie brauchen dazu die ursprüngliche Klasse kein bisschen zu verändern.

Zugriffsmodifizierer von Klassen, Prozeduren, Eigenschaften und Variablen

Bevor wir uns der Vererbung widmen, möchte ich an dieser Stelle kurz ein paar Worte über die Zugriffsmodifizierer verlieren, mit denen Sie in Visual Basic .NET bestimmen können, von wo aus der Zugriff auf ein Element gestattet ist und von wo aus nicht. Die Zugriffsmodifizierer `Private` und `Public` haben Sie schon kennen gelernt. Sie bestimmen, ob auf ein Element nur innerhalb eines bestimmten Gültigkeitsbereiches zugegriffen werden kann (`Private`) oder von überall aus (`Public`). Welche weiteren es für Objekte, Klassen und Funktionen/Eigenschaften gibt, zeigen die folgenden Tabellen:

Zugriffsmodifizierer bei Klassen

HINWEIS: Wenn nichts anderes gesagt wird, werden Klassen standardmäßig als `Friend` deklariert.

Zugriffsmodifizierer	CTS-Bezeichnung	Beschreibung

Private	Private	Als Privat können Klassen nur dann definiert werden, wenn sie geschachtelt in einer anderen Klasse definiert sind. Beispiel:
		Public Class A
		  Private Class B
		  End Class
		End Class
		
		Public Class C
		  'Zugriff verweigert, Class B ist Private!
		  Dim b as A.B
		End Class

Public	Public	Sie können auf die Klasse uneingeschränkt von außen zugreifen, auch aus anderen Assemblies heraus.
Friend	Assembly	Sie können innerhalb der Assembly auf die Klasse zugreifen, aber nicht aus einer anderen Assembly heraus.
Protected	Family	Es gilt das für Private Gesagte. Zusätzlich gilt: Auch aus der Klasse abgeleitete Klassen können auf die mit *Protected* gekennzeichneten und geschachtelten »inneren« Klassen zugreifen.
Protected Friend	FamilyOrAssembly	Der Zugriff auf die geschachtelte Klasse ist in abgeleiteten und von Klassen der gleichen Assembly aus möglich.

Tabelle 8.1: Mögliche Zugriffsmodifizierer für Klassen in Visual Basic .NET

Zugriffsmodifizierer bei Prozeduren (Subs, Functions, Properties)

HINWEIS: Wenn nichts anderes gesagt wird, werden `Subs`, `Functions` und `Properties` standardmäßig als `Public` deklariert. *Sie sollten gerade bei diesen Elementen aber auf jeden Fall einen Zugriffsmodifizierer verwenden, damit beim Blättern durch den Quellcode schnell deutlich wird, welchen Zugriffsmodus ein Element innehat.*

.

Zugriffsmodifizierer	CTS-Bezeichnung	Beschreibung
Private	Private	Nur innerhalb einer Klasse kann auf die Prozedur zugegriffen werden.
Public	Public	Sie können auf die Prozedur uneingeschränkt von außen zugreifen, auch aus anderen Assemblies heraus.
Friend	Assembly	Sie können innerhalb der Assembly auf die Prozedur zugreifen, aber nicht aus einer anderen Assembly heraus.
Protected	Family	Nur innerhalb der Klasse oder einer abgeleiteten Klasse kann auf die Prozedur zugegriffen werden. ▶

Zugriffsmodifizierer	CTS-Bezeichnung	Beschreibung
Protected Friend	FamilyOrAssembly	Nur innerhalb der Klasse, einer abgeleiteten Klasse oder innerhalb der Assembly kann auf die Prozedur zugegriffen werden.

Tabelle 8.2: Mögliche Zugriffsmodifizierer für Prozeduren in Visual Basic .NET

Zugriffsmodifizierer bei Variablen

Variablen, die auf Klassenebene nur mit `Dim` deklariert werden, gelten als `Private`, also nur von der Klasse aus zugreifbar. Variablen, die innerhalb eines Codeblocks oder auf Prozedurebene deklariert werden, gelten nur für den entsprechenden Codeblock. Innerhalb eines Codeblocks können Sie nur die `Dim`-Anweisung und keine anderen Zugriffsmodifizierer verwenden. Auf Prozedurenebene können Sie eine Variable zusätzlich als `Static` deklarieren. Mehr über den `Static`-Zugriffsmodifizierer erfahren Sie im ▶ Abschnitt »Statische und nicht-statische Methoden und Variablen« auf Seite 221.

Zugriffsmodifizierer	CTS-Bezeichnung	Beschreibung
Private	Private	Nur innerhalb einer Klasse kann auf die Variable zugegriffen werden. Variablen innerhalb von Prozeduren oder noch kleineren Gültigkeitsbereichen können nicht explizit als *Private* definiert werden, sind es aber standardmäßig.
Public	Public	Von außen kann auf die Klassenvariable uneingeschränkt zugegriffen werden. Sie sollten Variablen aber bestenfalls als *Protected* deklarieren und sie nur durch Eigenschaften nach außen offen legen. Mehr zu diesem Thema erfahren Sie im ▶ Abschnitt »Öffentliche Variablen oder Eigenschaften – eine Glaubensfrage?« auf Seite 255.
Friend	Assembly	Sie können innerhalb der Assembly auf die Klassenvariable zugreifen, aber nicht aus einer anderen Assembly heraus. Es gilt das für *Public* Gesagte.
Protected	Family	Nur innerhalb derselben oder einer abgeleiteten Klasse kann auf die Variable zugegriffen werden. Variablen sollten in Klassen, bei denen Sie davon ausgehen, dass sie später des Öfteren vererbt werden, als *Protected* definiert werden, damit abgeleitete Klassen ebenfalls darauf zugreifen können.
Protected Friend	FamilyOrAssembly	Nur innerhalb der Klasse, einer abgeleiteten Klasse oder innerhalb der Assembly kann auf die Klassenvariable zugegriffen werden. Von dieser Kombination sollten Sie absehen.
Static	- - -	Sonderfall in Visual Basic. Lesen Sie dazu bitte die Ausführungen im ▶ Abschnitt »Statische und nicht-statische Methoden und Variablen« auf Seite 221.

Tabelle 8.3: Mögliche Zugriffsmodifizierer für Prozeduren in Visual Basic .NET

Diese Tabellen sollen Ihnen kompakt und auf einen Blick die Zugriffsmodifizierer von Variablen verdeutlichen. Die CTS-Bezeichnungen der Zugriffsmodifizierer benötigen Sie, wenn Sie sich den IML-Code einer Klasse anschauen, um zu erkennen, welchen Zugriffsmodus beispielsweise eine Methode hat.

Unterschiedliche Zugriffsmodifizierer für Eigenschaften-Accesors

Seit Visual Studio 2005 ist es möglich, dass die *Get*- und *Set*-Accessors unterschiedliche Zugriffsmodifizierer aufweisen. So haben Sie beispielsweise die Möglichkeit, zu bestimmen, dass zwar eine bestimmte Eigenschaft von jedem Ort aus gelesen werden kann (`Public`) aber Eigenschaften nur von derselben Klasse aus geschrieben werden dürfen (`Private`). Im Code würde eine solche Eigenschaft folgendermaßen ausschauen:

```
Module Module1

    Sub Main()
        Dim locEigenschaftenTest As New EigenschaftenTest("Text für Eigenschaft")

        'Das Auslesen der Eigenschaft ist problemlos möglich
        Console.WriteLine("Eigenschaft enthält: " & locEigenschaftenTest.EineEigenschaft)

        'Der Set-Zugriffsmodifizierer verbietet aber das Schreiben,
        'weil er 'private' ist.
        locEigenschaftenTest.EineEigenschaft = "Neuer Text"
    End Sub

End Module

Public Class EigenschaftenTest

    Private myEineEigenschaft As String

    Sub New(ByVal textFürEigenschaft As String)
        'Ist erlaubt - die Klasse darf die
        'Eigenschaft beschreiben!
        EineEigenschaft = textFürEigenschaft
    End Sub

    Public Property EineEigenschaft() As String
        Get
            Return myEineEigenschaft
        End Get
        Private Set(ByVal value As String)
            myEineEigenschaft = value
        End Set
    End Property

End Class
```

BEGLEITDATEIEN: Sie finden die Codedateien zu diesem Beispiel im Verzeichnis *.\VB 2005 - Entwicklerbuch\D - OOP\Kap08\PropertiesDemo*. Öffnen Sie dort die entsprechende Projektmappen-Datei (*.sln*).

Dieses kleine Beispiel besteht aus zwei Einheiten – einem Modul und einer Klasse. Die Klasse `EigenschaftenTest` enthält einen parametrisierten Konstruktor sowie eine Eigenschaft, die aber Accessoren mit unterschiedlichen Zugriffsmodifizierern hat.

Innerhalb der Konstruktors (`Sub New`) ist der Schreibzugriff auf die Eigenschaft problemlos möglich, da aus der Klasse selbst heraus auch auf Elemente zugegriffen werden kann, deren Zugriff mit `Private` eingeschränkt wurde. Innerhalb des Moduls funktioniert der Zugriff allerdings nicht mehr, da sich das Programm zum Zeitpunkt des Zugriffs außerhalb der Klasse befindet, und wegen `Private` ist von diesem Punkt aus kein Herankommen an die Eigenschaft möglich.

Doch wozu brauchen Sie unterschiedliche Zugriffsmodifizierer in Eigenschaften während der Entwicklung Ihrer Software? Denken Sie beispielsweise an das Zurverfügungstellen von Assemblies (Klassenbibliotheken) für anderer Entwickler: Sie möchten beispielsweise in der Lage sein, bestimmte Eigenschaften Ihrer Klassen von jedem Punkt Ihrer Assembly aus zu manipulieren; Sie möchten aber gleichzeitig verhindern, dass ein Entwickler, der Ihre Assembly verwendet, dazu auch in der Lage ist. In diesem Fall definieren Sie den *Get*-Accessor als `Public` – das Lesen der Eigenschaft stellt schließlich kein Risiko dar und kann von überall aus erfolgen – aber den *Set*-Accessor als `Friend`. Vom gesamten Programmcode, der sich in Ihrer Klassenbibliothek befindet, können Sie dann die Eigenschaft der betreffenden Klasse manipulieren; Entwickler, die die Assembly einbinden, also von außerhalb Ihrer Assembly zugreifen, können sie aber nicht mehr direkt manipulieren. Solcherlei Eigenschaften sind dann wichtig, wenn es sich bei ihnen um Quasi-Konstanten handeln soll. Ihre Assembly definiert die Eigenschaft wann und wie sie will auf Grund bestimmter Zustände. Die Assemblies, die sie konsumieren, müssen aber mit dieser Einstellung leben. Klassen, die beispielsweise Einstellungen aus der Windows-Registry widerspiegeln, können davon Gebrauch machen. Denkbar wäre auch, eine Verbindungszeichenfolge zu einem SQL Server auf diese Weise freizulegen – Sie hätten zwar aus Ihrer Assembly heraus Manipulationsspielraum für die Verbindungszeichenfolge; eine Assembly könnte aber immer nur die Verbindungszeichenfolge mit der Eigenschaft auslesen, die Sie innerhalb Ihrer Assembly vorgeben.