

# 9

## Klassenvererbung und Polymorphie

---

261	<b>Wiederverwendbarkeit von Klassen durch Vererbung (Inheritance)</b>
271	<b>Überschreiben von Methoden und Eigenschaften</b>
275	<b>Das Speichern von Objekten im Arbeitsspeicher – und die daraus resultierende Vorsicht mit ihnen</b>
279	<b>Polymorphie</b>
294	<b>Abstrakte Klassen und virtuelle Prozeduren</b>
297	<b>Schnittstellen (Interfaces)</b>
311	<b>Die Methoden und Eigenschaften von Object</b>
318	<b>Shadowing (Überschatten) von Klassenprozeduren</b>
324	<b>Sonderform »Modul« in Visual Basic</b>
324	<b>Singleton-Klassen und Klassen, die sich selbst instanzieren</b>

---

### Wiederverwendbarkeit von Klassen durch Vererbung (Inheritance)

Vererbung und die Wiederverwendbarkeit und Möglichkeit zur Erweiterung von Klassen sind der zentrale Bestandteil im Framework. Ich würde sogar soweit gehen zu sagen, dass ohne die Möglichkeit, Klassen zu vererben, .NET keinen Sinn machen würde.

Visual Basic 6.0 beherrschte die Polymorphie<sup>1</sup> – die Möglichkeit, über gleiche Methodennamen verschiedene Klassen anzusprechen – nur sehr unzulänglich. In Visual Basic 6.0 konnten Sie Klassen nur durch die so genannte Delegation vererben – bei der eine Klasse eine andere Klasse als Member-Variable eingebunden und deren Eigenschaften durch neue Funktionen und Eigenschaftenprozeduren nach außen offen gelegt hat. In VB6 war die reine Polymorphie auf Schnittstellen beschränkt – mehr zu diesem Thema gibt's im ► Abschnitt »Schnittstellen (Interfaces)« ab Seite 297

---

<sup>1</sup> Etwa: »Vielgestaltigkeit«.

Bevor wir uns die Technik der Vererbung für das Beispiel zunutze machen, möchte ich Ihnen anhand einfacher Codebeispiele den Vorgang des Vererbens erklären. Unser Beispiel möchte ich dazu nicht als erstes verwenden, da es bereits einer Sonderbehandlung bei der Vererbung bedarf, die am Anfang verwirren würde und dem Verständnis für einen Moment im Wege wäre.

---

**BEGLEITDATEIEN:** Sie finden das folgende Beispielprojekt im Verzeichnis `VB 2005 - Entwicklerbuch\D - OOP\Kap09\Vererbung01`, aber ich würde Sie bitten, es zur Übung und zum besseren Verständnis von Grund auf mit nachzustellen.

---

- Legen Sie ein neues Projekt an, indem Sie *Neu* und *Projekt* aus dem Menü *Datei* wählen.
- Klicken Sie auf *Visual Basic-Projekte* in der Liste *Projekttypen*, wählen Sie *Konsolenanwendung* unter *Vorlagen* und bestimmen Sie »Vererbung« als Projektname.
- Bestimmen Sie ein Speicherverzeichnis Ihrer Wahl.
- Klicken Sie auf *OK*, um das neue Projekt zu erstellen.
- Doppelklicken Sie auf *Module1.vb* im Projektmappen-Explorer, um den Code für das Projekt anzuzeigen. Lassen Sie sich zunächst nicht durch das Schlüsselwort *Module* irritieren. Auf das Thema *Module* (das aus Framework-Sicht betrachtet in Visual Basic wieder eine Sonderrolle spielt) werde ich im Laufe dieses Kapitels noch eingehen.
- Ändern Sie den Namen von *Module1.vb* in *mblMain.vb*, indem Sie aus dem Kontextmenü den Befehl *Umbenennen* wählen und einen neuen Dateinamen eingeben. Denken Sie daran, die Dateinamenerweiterung *.vb* mit anzugeben!
- Doppelklicken Sie im Projektextplorer auf *mblMain.vb*, um das Codefenster zu öffnen.

Im Codeeditor unterhalb der Moduldefinition (also hinter *End Module*) geben Sie nun

```
Class ErsteKlasse
```

ein. Visual Basic erstellt automatisch die Zeile

```
End Class
```

darunter, um den Klassen-Codeblock abzuschließen. Diese erste Klasse soll eine Eigenschaft für die Wertezuweisung und eine Methode bekommen, mit der der Inhalt der Klasse ausgedruckt werden kann. Fügen Sie folgenden Code in die Klasse ein:

```
Class ErsteKlasse
```

```
    Protected myEinWert As Integer
```

```
    'Eigenschaft, um den Wert verändern zu können.
```

```
    Property EinWert() As Integer
```

```
        Get
```

```
            Return myEinWert
```

```
        End Get
```

```
        Set(ByVal Value As Integer)
```

```
            myEinWert = Value
```

```
        End Set
```

```

End Property
'Funktion, um den Inhalt als Zeichenkette (String) zurückzuliefern.
Function AlsZeichenkette() As String

    Return CStr(myEinWert)

```

```

End Function
End Class

```

Im Module *mdlMain* erstellen Sie nun ein kleines Rahmenprogramm, das diese neue Klasse zu Demonstrationszwecken verwendet:

```

Module mdlMain

    Sub Main()
        Dim klasse1 As New ErsteKlasse
        klasse1.EinWert = 5
        Console.WriteLine(klasse1.AlsZeichenkette())

        Console.WriteLine()
        Console.WriteLine("Zum Beenden Taste drücken")
        Console.ReadKey()
    End Sub

End Module

```

Sie können das Programm nun starten, und das Ergebnis wird dem entsprechen, was Sie sicherlich erwartet haben.

Für die nächsten Schritte stellen Sie sich einfach vor, dass Ihnen der Quelltext von *ErsteKlasse* nicht zur Verfügung steht. Unterhalb der Klassendefinition (die Sie natürlich im Geiste gar nicht sehen ...) fügen Sie nun eine weitere Klassendefinition ein:

```

Class ZweiteKlasse
    Inherits ErsteKlasse

End Class

```

Sie haben damit eine neue Klasse geschaffen, namens *ZweiteKlasse*. Diese Klasse hat keine Eigenschaften und keine Methoden, und sie hat sie doch. Wenn Sie den Cursor in die erste Prozedur der Codedatei platzieren, und unterhalb der Zeile

```
Console.WriteLine(klasse1.AlsZeichenkette())
```

die Zeile

```
Dim klasse2 As New ZweiteKlasse
```

einfügen, darunter den Instanznamen zu einfügen und den Punkt eintippen, sehen Sie, wie IntelliSense Ihnen die Elemente der neuen Klasse anbietet:



**Abbildung 9.1:** Obwohl es bislang keine Elementdefinitionen für *ZweiteKlasse* gibt, zeigt Ihnen IntelliSense dennoch eine Eigenschaft und eine Methode an

Was ist passiert?

Durch die Anweisung

`Inherits ErsteKlasse`

haben Sie bestimmt, dass *ZweiteKlasse* alle Elemente von *ErsteKlasse* erbt. Alles, was *ErsteKlasse* kann, können Sie auch mit *ZweiteKlasse* machen.

---

**WICHTIG:** Genau darum geht es bei der objektorientierten Programmierung: Durch die Vererbung schreiben Sie wieder verwendbaren Code. Wenn Sie vorhandene Klassen verändern wollen, um sie an ein bestimmtes Problem anzupassen, dann können Sie die Originalklasse in dem Zustand belassen, den sie hat. Sie vererben sie einfach in eine neue Klasse (man sagt auch: Sie »leiten sie ab«), und verändern dann die vererbte Klasse, um sie für Ihre neue Problemlösung anzupassen und zu erweitern.

---

Angenommen, Sie brauchen für ein bestimmtes Problem eine Klasse, die genau das kann, was *ErsteKlasse* kann, nur benötigen Sie zusätzlich eine weitere Eigenschaft, die den aktuellen Instanzwert um den Wert 10 erhöht ermittelt. In diesem Fall vererben Sie Klasse *ErsteKlasse*, so Sie es im Beispiel schon kennen gelernt haben, in *ZweiteKlasse* und fügen dann dort die neue Eigenschaftenprozedur hinzu:

```

Class ZweiteKlasse
    Inherits ErsteKlasse

    ReadOnly Property Um10Mehr() as Integer

        Get
            Return myEinWert + 10
        End Get
    End Property
End Class

```

Und das war es schon. Die gesamte Funktionalität, die *ErsteKlasse* hatte, hat *ZweiteKlasse* auch, und sie hat obendrein noch eine Eigenschaft mehr.

Gegenprobe: Unterhalb der Zeile

```
Dim klasse2 As New ZweiteKlasse
```

fügen Sie den Text **Console.WriteLine(klasse2.** ein; sobald Sie den Punkt tippen, zeigt Ihnen IntelliSense wieder die Elemente der Klasse an, und siehe da: Alle alten Elemente und die neue Eigenschaft sind in der Liste vorhanden!

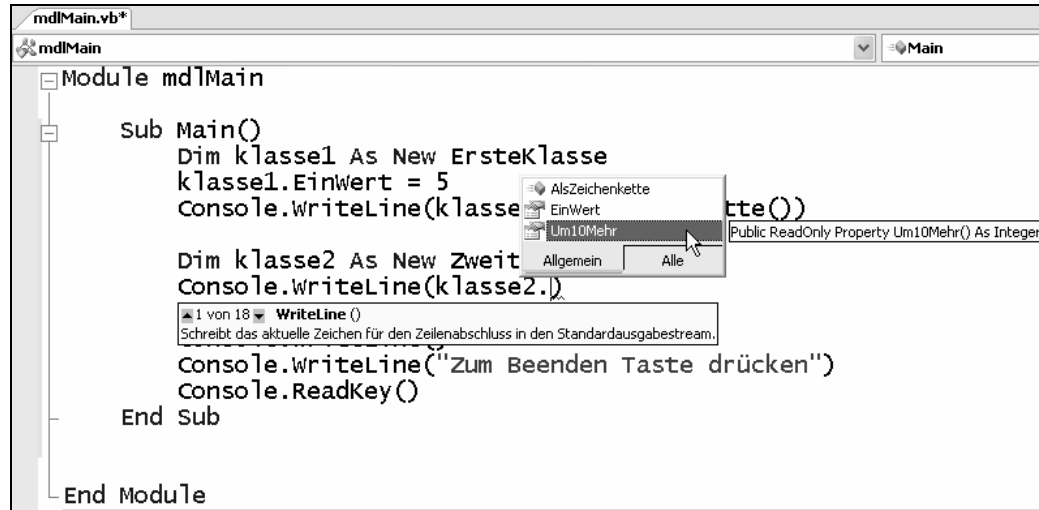


Abbildung 9.2: Die neue Eigenschaft ist jetzt zusammen mit den alten Elementen in *ZweiteKlasse* zu finden

Komplettieren Sie die Zeile,

```
Console.WriteLine(klasse2.Um10Mehr)
```

starten Sie das Programm anschließend, und schauen Sie, was passiert:

```
5
10
```

Zum Beenden Taste drücken

Haben Sie erwartet, dass 15 als zweites Ergebnis ausgegeben wird? Natürlich nicht, denn die aus *ErsteKlasse* entstandene Instanz *klasse1* ist völlig unabhängig von der Instanz *klasse2*, die aus *ZweiteKlasse* entstanden ist. Sie haben sich von *ErsteKlasse* nur den Code als Vorlage »geborgt« – die Objekte, die aus beiden Klassen entstehen können, haben natürlich beide einen unterschiedlichen Datenbereich.

---

**HINWEIS:** Wichtig für das Verständnis von Klassen ist, wie sie intern verwaltet werden. Der Code einer Klasse ist immer nur ein einziges Mal vorhanden – auch wenn Sie mehrere Instanzen einer Klasse anlegen. Deswegen ist es natürlich auch Unsinn zu glauben, dass eine Klasse die zehnmal soviel Programmcode hat wie eine Klasse »X«, bei zehnfacher Instanzierung in zehn verschiedene Objekte hundertmal so viel Speicher benötigt wie Klasse X. Sie braucht für den Programmcode immer gleich viel Speicher, egal wie viele Instanzen aus ihr entstehen. Der Programmcode ist natürlich auch nicht doppelt vorhanden, wenn Sie eine Klasse aus einer anderen ableiten. Nur der zusätzliche Programmcode durch das Hinzufügen oder Verändern vorhandener Elemente belegt zusätzlichen Speicher. Ausschließlich die Member-Variablen einer Klasse sind dafür ausschlaggebend, wie viel zusätzlichen Speicher eine Klasse beim Instanzieren in Objekte benötigt.

---

Damit das erwartete Ergebnis eintritt, müssen Sie das Programm wie folgt abändern:

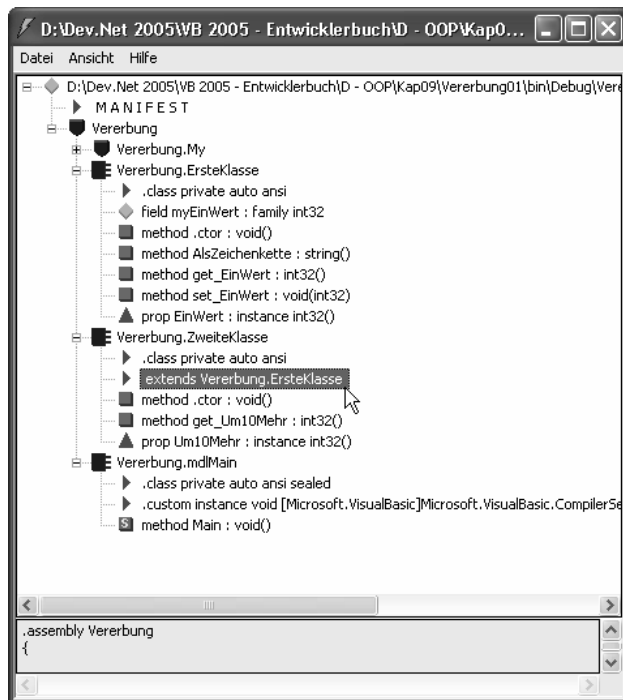
```
Module mdlMain
```

```
Sub Main()  
    Dim klasse1 As New ErsteKlasse  
    klasse1.EinWert = 5  
    Console.WriteLine(klasse1.AlsZeichenkette())  
  
    Dim klasse2 As New ZweiteKlasse  
    klasse2.EinWert = 5  
    Console.WriteLine(klasse2.Um10Mehr)  
    Console.WriteLine()  
    Console.WriteLine("Zum Beenden Taste drücken")  
    Console.ReadKey()  
End Sub
```

```
End Module
```

Jetzt betrachten wir die Klassen mit dem IML-Disassembler und schauen, ob uns Visual Basic wieder irgendetwas an Arbeit abgenommen hat.

- Starten Sie ILDASM, und öffnen Sie die Datei *Vererbung.Exe*, die Sie im Verzeichnis *\bin\Debug\* des Verzeichnisses finden, in dem Sie das Projekt angelegt hatten.
- Öffnen Sie per Mausklick auf das davor stehende Pluszeichen den Zweig *ErsteKlasse* und den Zweig *ZweiteKlasse*. Anschließend sollten Sie ein Bild vor sich sehen, etwa wie in Abbildung 9.3 zu sehen.



**Abbildung 9.3:** Die beiden Testklassen im IML-Disassembler

Genau wie in einem der vorherigen Beispiele hat Ihnen Visual Basic wieder Arbeit abgenommen und sowohl in der Basisklasse `ErsteKlasse` als auch in der abgeleiteten Klasse `ZweiteKlasse` entsprechende Konstruktorprozeduren (`.ctor`) eingefügt. In der Abbildung ebenfalls auf den ersten Blick erkennbar: Eigenschaften werden intern in Funktionen bzw. Methoden umgewandelt. Aus der `EinWert`-Eigenschaft der Basisklasse, die jeweils einen `Get`- und einen `Set`-Accessor hat, macht der Visual Basic-Compiler die beiden Funktionen `get_EinWert` und `set_EinWert`. Die Eigenschaft selbst wird darunter definiert, und der IML-Code in ihr bestimmt lediglich, welche der in der Klasse vorhandenen Funktionen die Eigenschaft auflösen (per Doppelklick auf den Eigenschaftennamen können Sie sich den folgenden Code anzeigen lassen).

```
.property instance int32 EinWert()
{
    .set instance void Vererbung.ErsteKlasse::set_EinWert(int32)
    .get instance int32 Vererbung.ErsteKlasse::get_EinWert()
} // end of property ErsteKlasse::EinWert
```

Was noch auffällt: Genau wie vermutet, sind in `ZweiteKlasse` nur die zusätzlich vorhandenen Elemente als Code vorhanden. Dass die Klasse von der Basisklasse erbt und damit auch deren Elemente »mitnutzen« kann, wird – wie in Abbildung 9.3 gezeigt – durch `extends Vererbung.ErsteKlasse` geregelt. Da die einzige Eigenschaft nur einen `Get`-Accessor hat, gibt es übrigens auch nur eine Eigenschaftsfunktion in Form von `get_Um10Mehr`.

Ebenfalls erwähnenswert: Die Member-Variable `myEinWert` habe ich vorausschauend als `Protected` deklariert. Laut Tabelle des letzten Abschnittes ist eine als `Protected` deklarierte Variable eine, »... [auf die nur] innerhalb derselben Klasse oder einer abgeleiteten Klasse zugegriffen werden kann«. Wäre

die Variable nur als Private deklariert, könnte die abgeleitete Klasse ZweiteKlasse sie nicht manipulieren; die zusätzliche Eigenschaft, die sie implementiert, würde ergo nicht funktionieren.

---

**TIPP:** Member-Variablen, die Sie in Klassen verwenden, welche später durch das Vererben wieder verwendet werden sollen, deklarieren Sie deshalb nach Möglichkeit als Protected, es sei denn, Sie wünschen ausdrücklich, dass nur die Basisklasse die Variable manipulieren darf.

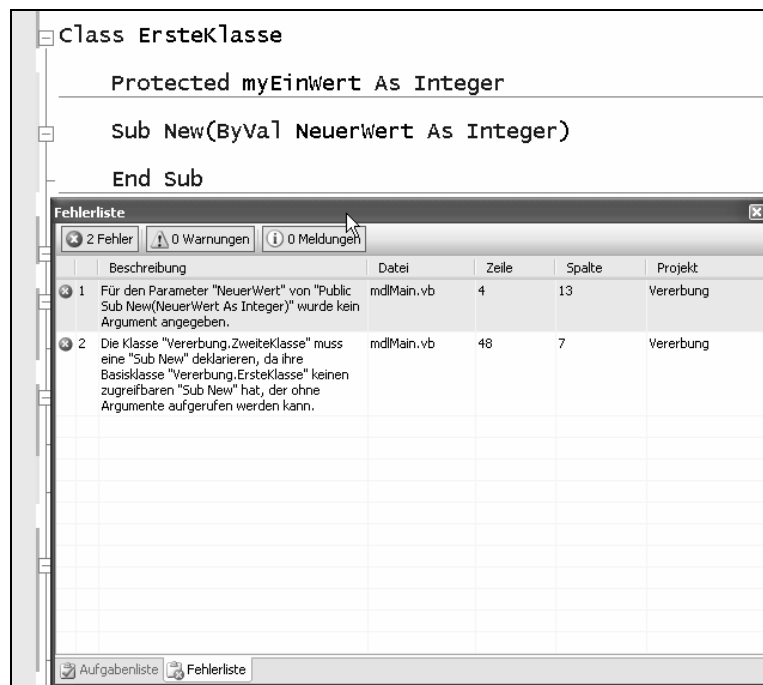
---

Nachdem nun bekannt ist, dass abgeleitete Klassen automatisch den Standardkonstruktor der Basisklasse aufrufen, wäre es interessant zu erfahren, was passiert, wenn die Basisklasse keinen Standardkonstruktor hat. Einen solchen Fall haben Sie mit der letzten Version der Klasse aus dem *RomanNumerals*-Beispiel nämlich kennen gelernt. Eine Klasse hat dann keinen automatisch generierten Standardkonstruktor, wenn sie einen parametrisierten Konstruktor hat.

Was passiert also, wenn wir der Klasse ErsteKlasse einen Konstruktor hinzufügen, der einen Parameter (zum Beispiel den Initialisierungswert für myEinWert) übernimmt? Probieren Sie es aus: Fügen Sie die Zeile

```
Sub New(ByVal NeuerWert As Integer)
```

in den Klassencode von ErsteKlasse ein. Sobald Sie die Änderungen eingefügt haben, sehen Sie diverse Fehlermarkierungen im Quelltext und die entsprechenden Beschreibungen dazu in der Fehlerliste, etwa wie in Abbildung 9.4 zu sehen.



**Abbildung 9.4:** Nach dem Einfügen eines parametrisierten Konstruktors sehen Sie gleich zwei Fehler in der Fehlerliste

Der Grund: Im Modul innerhalb von Sub Main kann klasse1 nicht mehr instanziiert werden, denn hinter

```
Dim klasse1 As New ErsteKlasse
```

steht kein Parameter. Da wir einen parametrisierten Konstruktor in die Klassendefinition eingefügt haben, gibt es keinen Standardkonstruktor mehr. Sobald Sie selbst irgendeinen Konstruktor (parametrisiert oder nicht) in eine Klasse einfügen, hört Visual Basic auf, Ihnen mit dem »Hineinkompilieren« eines Standardkonstruktors unter die Arme zu greifen. Kein Standardkonstruktor heißt: nehmen, was da ist. Und da ist nur einer, dem Parameter übergeben werden – der erste Fehler ist also erst erledigt, wenn Sie den Initialisierungswert für die Klasse mit angeben, etwa so:

```
Dim klasse1 As New ErsteKlasse(5)
Console.WriteLine(klasse1.AlsZeichenkette())
.
.
.
```

Kein Standardkonstruktor bedeutet aber auch: Die abgeleitete Klasse weiß nicht, wie sie die Basisklasse instanzieren soll (was auf jeden Fall passieren muss). In diesem Fall müssen Sie also selbst dafür Sorge tragen, dass die Basisklasse aufgerufen wird. Sie erreichen das, indem Sie den Konstruktor der Basisklasse mit dem MyBase-Schlüsselwort aufrufen.

Das folgende Listing stellt die funktionierende Version dar (Änderungen sind fett hervorgehoben; einige unwichtige Teile sind durch »...« abgekürzt):

```
Module mdlMain
```

```
    Sub Main()
        Dim klasse1 As New ErsteKlasse(5)
        Console.WriteLine(klasse1.AlsZeichenkette())

        Dim klasse2 As New ZweiteKlasse(5)
        Console.WriteLine(klasse2.Um10Mehr)

        Console.WriteLine()
        Console.WriteLine("Zum Beenden Taste drücken")
        Console.ReadLine()
    End Sub
```

```
End Module
```

```
Class ErsteKlasse
```

```
    Protected myEinWert As Integer

    Sub New(ByVal NeuerWert As Integer)
        myEinWert = NeuerWert
    End Sub

    'Eigenschaft, um den Wert verändern zu können.
    Property EinWert() As Integer
    ...
    End Property
```

```

'Funktion, um den Inhalt als Zeichenkette (String) zurückzuliefern.
Function AlsZeichenkette() As String

    Return CStr(myEinWert)

End Function
End Class

Class ZweiteKlasse
    Inherits ErsteKlasse

    Sub New(ByVal NeuerWert As Integer)
        MyBase.New(NewerWert)
    End Sub

    ReadOnly Property Um10Mehr() As Integer
        ...
    End Property

End Class

```

## Initialisierung von Member-Variablen bei Klassen ohne Standardkonstruktoren

Interessant ist zu sehen, was passiert, wenn Sie Member-Variablen schon bei ihrer Deklaration definieren, etwa wie im folgenden Beispiel:

```

Class ErsteKlasse

    Protected myEinWert As Integer = 9

    Sub New()
        'Hat keinen Sinn, füllt aber den Konstruktor mit Code.
        Console.WriteLine("Testausgabe: Kein Parameter-Konstruktor")
    End Sub

    Sub New(ByVal AuszugebenderText As String)
        'Hat keinen Sinn, füllt aber den Konstruktor mit Code.
        Console.WriteLine("Testausgabe: " & AuszugebenderText)
    End Sub.

    :
    :

```

In diesem Beispiel gibt es einen Standardkonstruktor und zusätzlich zwei parametrisierte. Da kein Code »außerhalb« einer Klasse ausgeführt werden kann, stellt sich die Frage: Wo findet die Zuweisung

```
Protected myEinWert As Integer = 9
```

denn eigentlich statt? Die Antwort offenbart wieder ein Blick in den IML-Code der Klasse. Der Compiler treibt hier den größten Aufwand, denn er muss den Code für das Initialisieren der Mem-

ber-Variablen in jedem Konstruktor einbauen (siehe Abbildung 9.4). Nur so ist gewährleistet, dass alle erforderlichen Variableninitialisierungen *in jedem Fall* durchgeführt werden.

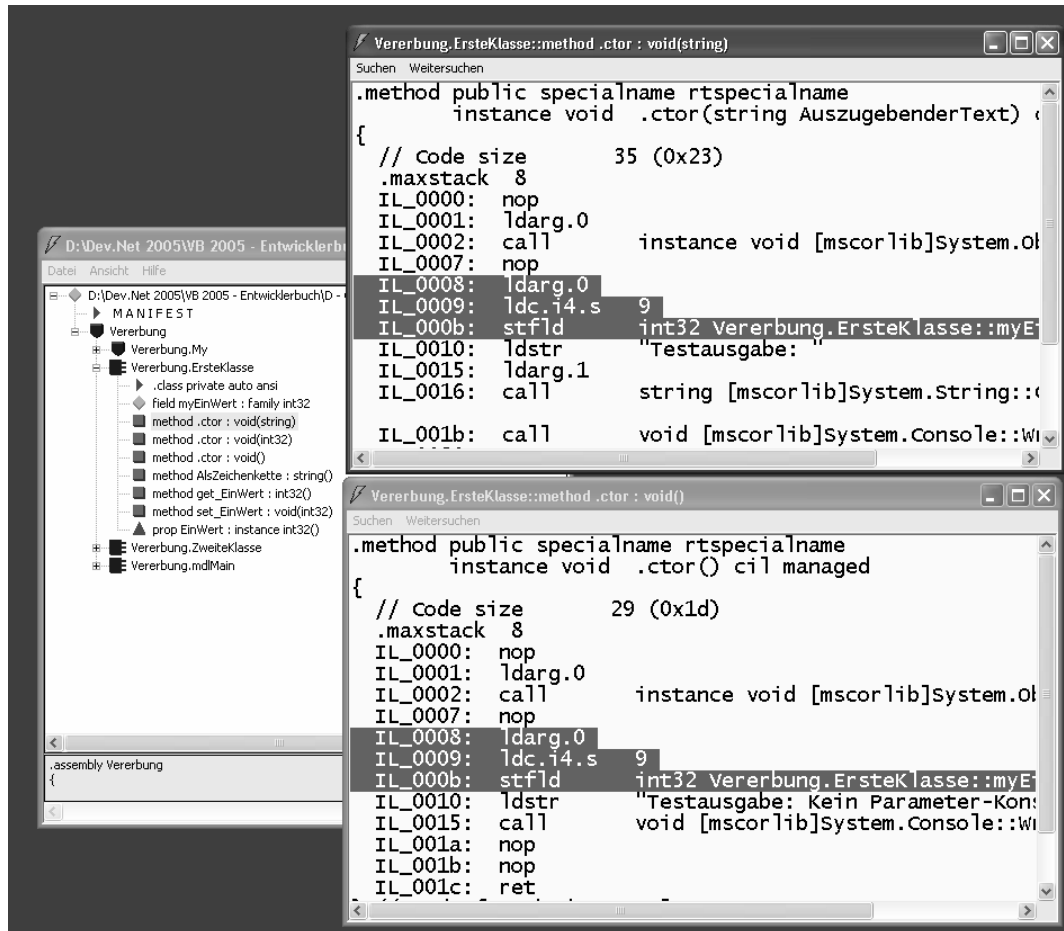


Abbildung 9.5: Initialisierungen von Member-Variablen werden im Bedarfsfall in jede Konstruktorprozedur eingebaut

## Überschreiben von Methoden und Eigenschaften

Sie denken sicherlich auch, dass das Ableiten von Klassen eine ziemlich geniale Sache ist. Es strukturiert Ihre Programme, macht sie leichter lesbar und hilft Ihnen insbesondere beim sauberen Aufbau größerer und komplexer Projekte. Doch Klassen wären nicht evolutionär, und zwar im wahrsten Sinne des Wortes, wenn es nicht die Möglichkeit gäbe, bestimmte Funktionen einer Basisklasse durch eine andere der abgeleiteten auszutauschen. Klassen, die andere Klassen einbinden, werden dadurch unglaublich flexibel.

Das Ersetzen von Methoden oder Eigenschaften durch andere in einer abgeleiteten Basisklasse erfordert allerdings, dass die Basisklasse der Klasse, die sie ableitet, auch gestattet, bestimmte Funktionen zu überschreiben. Funktionen bzw. Methoden oder Eigenschaften müssen von Ihnen also explizit gekennzeichnet werden, damit sie später überschrieben werden dürfen. In Visual Basic geschieht die Kennzeichnung einer Prozedur einer Basisklasse zum Überschreiben mit dem Schlüsselwort `Overridable` (überschreibbar). Alle Funktionen und Eigenschaften, die mindestens als `Protected` gekennzeichnet sind, können den `Overridable`-Modifizierer tragen. Die Eigenschaften bzw. Methoden, die dann anschließend eine in der Basisklasse überschreiben, müssen wiederum mit dem Schlüsselwort `Overrides` (überschreibt) gekennzeichnet sein.

---

**HINWEIS:** Es ist übrigens wichtig, dass Sie Überschreiben und Überladen nicht in einen Topf werfen, kräftig drin rumrühren, und schauen, was anschließend dabei herauskommt: Visual Basic erlaubt es nämlich, dass eine abgeleitete Klasse die Prozedur einer Basisklasse überlädt. In diesem Fall ersetzen Sie die Basisfunktion nicht, Sie ergänzen diese nur um eine weitere Überladung. Das heißt im Klartext: Wenn Sie Prozeduren einer Basisklasse wirklich überschreiben (sie also ersetzen) wollen, dann grundsätzlich nur mit der gleichen Signatur wie die der Basisklasse.

---

---

**WICHTIG:** In diesem Zusammenhang eine vielleicht nicht unwichtige Ergänzung: Während Sie beim Überladen von Funktionen innerhalb einer Klasse, die Sie komplett neu implementieren, das `Overloads`-Schlüsselwort auch weglassen und nur mit doppelten Methodennamen (aber unterschiedlichen Signaturen!) auskommen können, *müssen* Sie `Overloads` verwenden, wenn Sie eine Methode einer Basisklasse um eine weitere »Überladungsversion« in einer abgeleiteten Klasse ergänzen!

---

Ein kleines Beispiel zu überschriebenen Methoden: Angenommen, in der abgeleiteten Klasse `ZweiteKlasse` passt Ihnen die Art und Weise nicht, wie der `String` durch die Funktion `AlsZeichenkette` den Wert der Variablen als Zeichenkette zurückliefert. Sie möchten beispielsweise, dass die `AlsZeichenkette`-Funktion den Wert im Stil »der Wert lautet: xxx« ausgibt. In diesem Fall würden Sie die Klassen wie folgt verändern:

---

**BEGLEITDATEIEN:** Sie finden dieses Beispiel im Verzeichnis `.\VB 2005 - Entwicklerbuch\D - OOP\Kap09\Vererbung02`.

---

Module mdlMain

```
Sub Main()  
    Dim klasse1 As New ErsteKlasse(5)  
    Console.WriteLine("Klasse1 ergibt durch AlsZeichenkette: " & klasse1.AlsZeichenkette())  
  
    Dim klasse2 As New ZweiteKlasse(5)  
    Console.WriteLine("Klasse2 ergibt durch AlsZeichenkette: " & klasse2.AlsZeichenkette())  
  
    Console.WriteLine()  
    Console.WriteLine("Return drücken, zum Beenden")  
    Console.ReadLine()  
End Sub
```

End Module

Class ErsteKlasse

```
Protected myEinWert As Integer = 9

Sub New(ByVal NeuerWert As Integer)
    myEinWert = NeuerWert
End Sub

Sub New(ByVal NeuerWert As Integer, ByVal NurZumTesten As Integer)
    myEinWert = NeuerWert
End Sub

'Eigenschaft, um den Wert verändern zu können.
Property EinWert() As Integer
.
.
.
End Property

'Um die Funktion als String auszudrucken
Public Overridable Function AlsZeichenkette() As String

    Return CStr(myNeuerWert)

End Function
```

End Class

Class ZweiteKlasse

```
Inherits ErsteKlasse

Sub New(ByVal NeuerWert As Integer)
    MyBase.New(NewerWert)
End Sub

ReadOnly Property Um10Mehr() As Integer
.
.
.
End Property

Public Overrides Function AlsZeichenkette() As String
    Return "der Wert lautet: " & MyBase.AlsZeichenkette()
End Function
```

End Class

Wenn Sie diese veränderte Version des Programms laufen lassen, sehen Sie das folgende Ergebnis auf dem Bildschirm:

```
Klasse1 ergibt durch AlsZeichenkette: 5
Klasse2 ergibt durch AlsZeichenkette: der Wert lautet: 5
```

Zum Beenden Taste drücken

## Überschreiben vorhandener Methoden und Eigenschaften von Framework-Klassen

Nun ist der Name unserer Beispielfunktion nicht sonderlich glücklich gewählt – AlsZeichenkette ist bestenfalls ein deutsches Ausdrucksfragment, und um die Umwandlung in eine Zeichenkette beispielsweise in korrektem Englisch auszudrücken, um sich der Sprache des Frameworks selbst anzunähern, müsste es ToString heißen.

Wenn Sie allerdings die vorhandene Funktion AlsZeichenkette in der ersten Klasse in ToString ändern, passiert etwas, was nicht unbedingt vorhersagbar ist (siehe Abbildung 9.6):

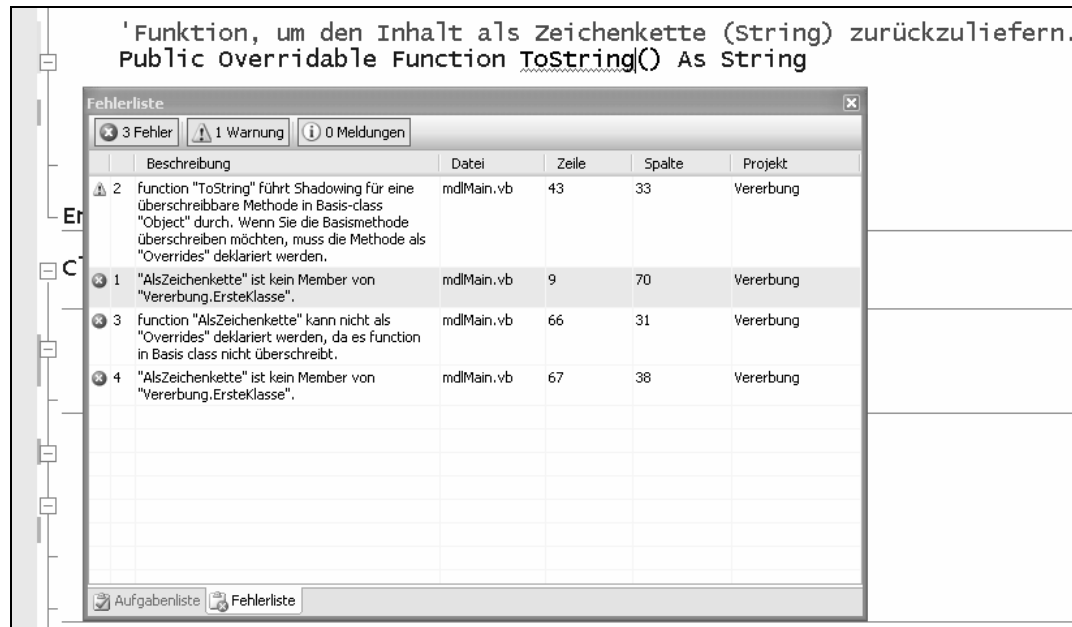


Abbildung 9.6: Versuchen Sie ToString zu implementieren, sehen Sie diese Fehlermeldung

Nur, welche Methode ist hier gemeint? IntelliSense listet die vorhandenen Member eines Objektes doch auf – wie in Abbildung 9.2 zu sehen, war eine Methode namens ToString nicht dabei. Doch wir haben uns in der verwendeten Einstellung bisher auch nicht alle Member anzeigen lassen. IntelliSense erlaubt es nämlich, die weniger wichtigen Methoden in der Vervollständigungsliste auszublenden. Erst wenn Sie in der Liste auf *Alle* umschalten, werden auch wirklich alle Member angezeigt, so auch ToString, wie in Abbildung 9.7 zu sehen.

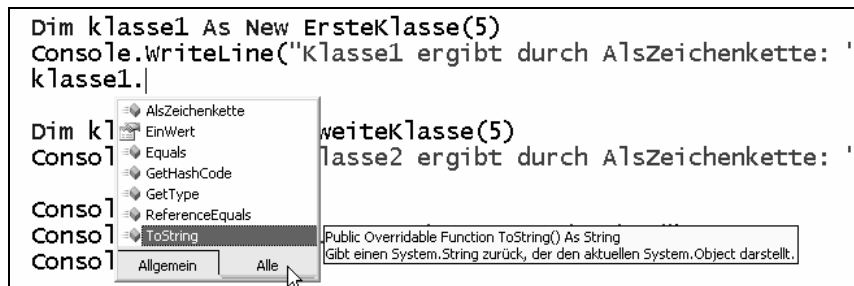


Abbildung 9.7: Jetzt sehen Sie alle Member der *ErsteKlasse*-Instanz

IntelliSense ist Ihnen an dieser Stelle von besonderem Nutzen, weil Sie die Modifizierer der ToString-Funktion in der Tooltip-Beschreibung auch gleich sehen können. Diese verraten Ihnen, dass die Funktion als *Overridable* deklariert wurde – Sie haben also selbst die Möglichkeit, die Ableitung dieser Klasse in *ErsteKlasse* mit *Overrides* zu überschreiben.

---

**BEGLEITDATEIEN:** Sie finden die so veränderte Version des Beispiels übrigens im Verzeichnis `.\VB 2005 - Entwicklerbuch\D - OOP\Kap09\Vererbung03`.

---

## Das Speichern von Objekten im Arbeitsspeicher – und die daraus resultierende Vorsicht mit ihnen

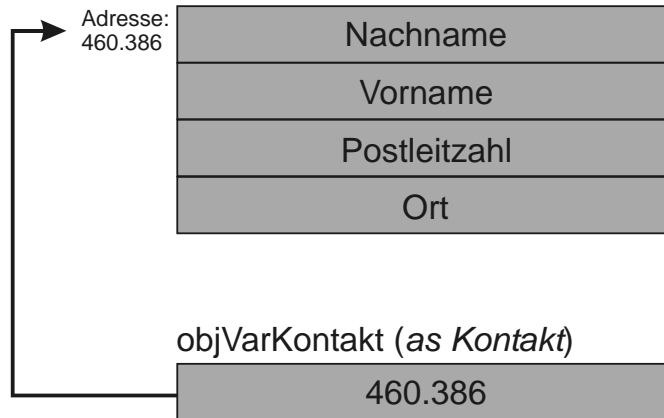
Bevor wir uns dem Allerwichtigsten dieses Kapitels nähern, wäre es zum besseren Verständnis überaus ratsam, ein paar Worte zur Speicherung von Objekten (und nahezu allen von ihnen abgeleiteten Klassen zu verlieren). Objektvariablen und Objekte sind nämlich nicht so miteinander verbunden, wie man es sich zu Anfang vielleicht vorstellt. Im Gegenteil: Der Verbund einer Objektvariablen mit den eigentlichen Daten des Objektes hält bestenfalls so gut, wie eine amerikanische Prominentenehe: Was auf den ersten Blick so innig und für immer geschaffen zu sein scheint, ist in der nächsten Sekunde auch schon wieder sauber getrennter Schnee von gestern.

Die Wahrheit ist nämlich: Eine Objektvariable speichert im Grunde genommen nur einen Zeiger auf die eigentlichen Daten im so genannten Managed Heap – einem Speicherbereich, den das .NET-Framework verwaltet, und der sich im Arbeitsspeicher Ihres Computers befindet.

Und wie wir (älteren jedenfalls) aus unseren Anfängertagen, in denen es noch 64er und Atari STs in Maschinensprache zu programmieren galt, noch alle wissen, untergliedert sich der Arbeitsspeicher eines Computers in bestimmte Speicherstellen, die alle bestimmte »Hausnummern« (die Speicheradressen) besitzen.

Wenn Sie nun ein Objekt instanzieren – dabei spielt es gar keine Rolle, ob tatsächlich *Object* oder eine aus *Object* Klasse dazu herhält – dann legt das Framework beispielsweise die Daten für diese Objektinstanz an Speicheradresse 460.386 auf dem Managed Heap ab, und die Objektvariable wird zu einer Integervariablen oder (auf 64-Bit-Systemen) zu einer Long-Variablen, die diese Adresse trägt. Bildlich gesprochen sieht das folgendermaßen aus:

## Instanz aus Klasse: Kontakt



**Abbildung 9.8:** Objektvariablen speichern im Grunde genommen nur die Speicheradressen auf die eigentlichen Daten, die das Framework im Managed Heap ablegt

Diese Tatsache hat aber entscheidende Folgen, wie das folgende Beispiel gleich zeigen wird.

---

**BEGLEITDATEIEN:** Sie finden den Quellcode dieses Beispiels im Verzeichnis `.\VB 2005 - Entwicklerbuch\D - OOP\Kap09\Vererbung04`.

---

Module mdlMain

Sub Main()

```
'Instanzieren mit New und dadurch  
'Speicher für das Kontakt-Objekt  
'auf dem Managed Heap anlegen  
Dim objVarKontakt As New Kontakt  
  
'Daten zuordnen  
With objVarKontakt  
    .Nachname = "Halek"  
    .Vorname = "Sarah"  
    .Plz = "99999"  
    .Ort = "Musterhausen"  
End With  
  
'Nur Objektvariable anlegen,  
'es wird aber kein Speicher reserviert!  
Dim objVarKontakt2 As Kontakt  
  
'objVarKontakt2 "zeigt" ab jetzt auf  
'die selbe Instanz wie objVarKontakt  
objVarKontakt2 = objVarKontakt  
  
'Und das kann man auch beweisen:  
'Das Ändern der Instanz geschieht...
```

```

objVarKontakt2.Nachname = "Löffelmann"

'durch beide Objektvariablen, die natürlich
'auch dasselbe widerspiegeln.
Console.WriteLine(objVarKontakt.Nachname)

Console.WriteLine()
Console.WriteLine("Zum Beenden Taste drücken")
Console.ReadKey()
End Sub

End Module

Class Kontakt

    Public Nachname As String
    Public Vorname As String
    Public Plz As String
    Public Ort As String

End Class

```

Wenn Sie dieses Beispiel laufen lassen, erhalten Sie

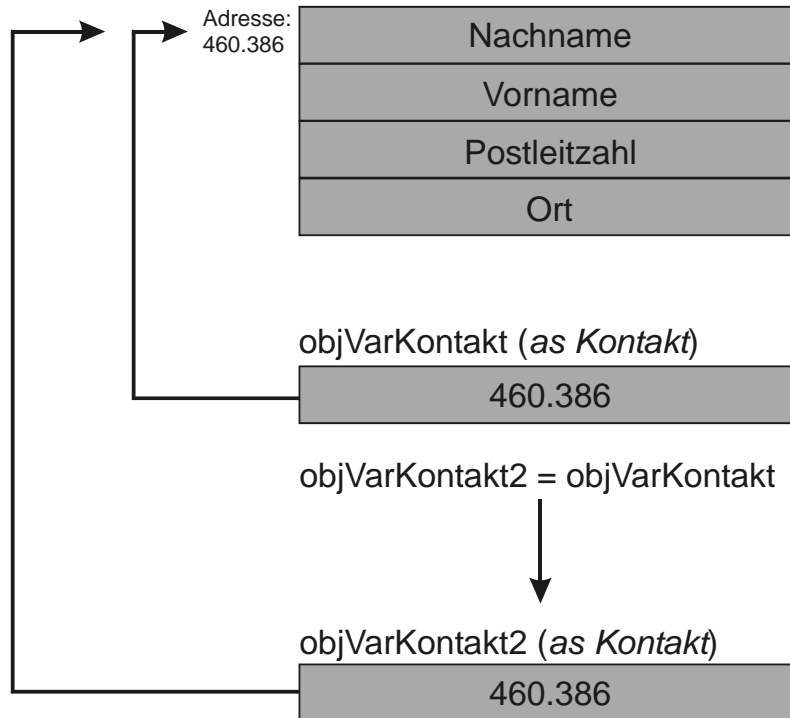
Löffelmann

Zum Beenden Taste drücken

als Ergebnis. Soweit ist das noch nichts Besonderes, doch bemerkenswert wird es dann, wenn Sie erkennen, dass es nur eine einzige Dateninstanz der Klasse Kontakt in diesem Beispiel gibt, die offensichtlich durch zwei Objektvariablen angesprochen und damit auch widerspiegelt wird: Beim Instanzieren wird die Adresse des Speichers, an dem die Daten der Instanz abgelegt werden, in der Objektvariablen objVarKontakt gespeichert. Diese Adresse wird in die Variable objVarKontakt2 übertragen, und wichtig, hierbei wird nicht etwa eine Kopie der gesamten Instanz im Managed Heap angelegt! Eine Objektvariable »zeigt« also quasi auf die Daten, die sie verwaltet. In anderen Programmiersprachen wie C++ gibt es zu diesem Zweck besondere Variablen, die auch als »Zeiger« bezeichnet werden. In Visual Basic wird eine Objektvariable, die die Instanz einer Klasse verwaltet, automatisch zu dem, was man in anderen Programmiersprachen als Zeiger bezeichnet.

In Form einer Grafik sieht das Ganze folgendermaßen aus:

## Instanz aus Klasse: Kontakt



**Abbildung 9.9:** Das Kopieren einer Objektvariablen in eine andere Objektvariable kopiert nur den Zeiger auf die Instanz – die dann durch beide Variablen manipulierbar und abrufbar wird

Dieses Verhältnis zwischen Objektvariable eigentlichem Objekt (eigentlicher Instanz) sollten Sie sich gut einprägen, da sie einerseits Quelle schwer zu findender Fehler ist – schließlich kann es auch versehentlich passieren, dass, wenn Sie nicht aufpassen, eine Objektvariable die Instanz eines Objektes verändert, die eigentlich und ausschließlich durch eine ganz andere Objektvariable angesprochen werden sollte. Andererseits kann Ihnen dieses Verhältnis auch zum Vorteil gereichen, nämlich wenn es darum geht, nicht Objekte zu kopieren, sondern nur die Zeiger auf diese – beispielsweise wenn es beim Sortieren großer Objektmengen auf Geschwindigkeit ankommt, und keine ganzen Speicherblöcke (mit den Daten der eigentlichen Instanzen) sondern nur die Zeiger auf die Objektinstanzen selbst kopiert werden müssten.

---

**HINWEIS:** Das nächste Kapitel, ► Kapitel 10, hält weitere Informationen zu diesem Thema bereit.

---

# Polymorphie

Sie haben nun viel über Vererbung und über das Überschreiben von Klassen-Membem erfahren – jetzt lautet die große Frage, wie Ihnen diese Techniken von Nutzen sein können. Damit Vererbung und Funktionsüberschreibung richtig Sinn ergeben, braucht es eine Technik, die sich »Polymorphie«<sup>2</sup> nennt. Und würden Sie mich bitten, Polymorphie in einem Satz zu erklären, dann würde ich Sie zunächst bitten, sich festzuhalten und Ihnen anschließend Folgendes zur Antwort geben: »Polymorphie in der OOP beschreibt die Möglichkeit, Methoden oder Eigenschaften in den Klassen einer Klassenerbfolge auszutauschen, und die gleichnamigen aber dennoch funktionell unterschiedlichen Methoden und Eigenschaften der veränderten Klassen der Erbfolge über *dieselbe* Objektvariable vom Typ der Basisklasse in Abhängigkeit von der tatsächlich instanziierten abgeleiteten Klasse erreichen zu können.« Das klingt sehr abstrakt, und das ist es auch.

Ich habe lange darüber nachgedacht, ein halbwegs plausibles Beispiel für Polymorphie im täglichen Leben zu finden, um zunächst einmal nur das abstrakte Konzept ein wenig zu vereinfachen. Ich gebe aber zu, dass ich zunächst auch nach stundenlangem Grübeln entnervt aufgab. Doch dann kam mir mein Kollege Jürgen mit einem Zufall zu Hilfe, den ich in meinem Leben so bislang noch nicht erlebt hatte.

Jürgen stand an der Kasse unseres örtlichen Lebensmittelhändlers und war gerade dabei, seine Einkäufe zu bezahlen, als sein Handy schellte, was die freundliche Kassiererin allerdings nicht mitbekam. Da er sein Handy zwischen Schulter und Ohr einklemmte, sah die Kassiererin auch nicht, dass er telefonierte. Und wie es der Zufall wollte, wählte Jürgen, der im Grunde genommen nun mit seiner Freundin sprach, und – eben weil er an der Kasse stand – vergleichsweise kurz angebunden war, seine Worte so, dass sie zufällig aber perfekt auf zwei Gespräche passten, die er dann (eines freiwillig mit seiner Freundin, eines unfreiwillig mit der Kassiererin) auch führte. Die Worte passten sogar so perfekt zu dem, was er zur Kassiererin hätte sagen können, dass sich zwischen ihm und der Kassiererin ein regelrechtes Kurzgespräch entwickelte. Diese Schlagabtausche wiederholten sich einige Male, bis beiden das Missverständnis auffiel, da er die Aussage der Kassiererin »das macht dann also 21,45 Euro« mit der schönen Floskel »ich dich auch« parierte.

Jürgen führte also zwei komplett unabhängige Gespräche, mit einem jeweils absolut anderen Inhalt; Jürgen verwendete aber die exakt gleiche Methode zur »Erledigung seines Anliegens«. Seine Methode, also die Wahl seine Worte, war an dieser Stelle also absolut vielgestaltig – eben polymorph –, obwohl die Worte in beiden Gesprächen dieselben waren.

Doch zurück zur Programmierung: Was würden Sie erwarten, wenn Sie im Beispiel von *Vererbung03* die Zeile des Moduls *mdlMain* von

```
Dim klasse2 As New ZweiteKlasse(5)
```

in die Zeilen

```
Dim klasse2 As ErsteKlasse  
klasse2 = New ZweiteKlasse(5)
```

abändern? Glauben Sie, dass Visual Basic einen Fehler auslöst, da Sie *klasse2* als *ErsteKlasse* deklariert, dieser Objektvariablen anschließend aber eine Instanz von zweiter Klasse zugewiesen haben?

---

<sup>2</sup> Etwa »vielgestaltig«, »in verschiedenen Formen auftretend«.

Mitnichten! Diese Vorgehensweise ist nicht nur kein Fehler, Sie haben gerade sogar das wohl mächtigste Werkzeug der objektorientierten Programmierung kennen gelernt.

Ein etwas größeres Praxisbeispiel soll zur anschaulichen Demonstration der Polymorphie dienen. Stellen Sie sich vor, Sie arbeiten in der EDV-Abteilung eines größeren Versandhauses. Ihre Aufgabe besteht darin, eine Software zu entwickeln, die Textlisten mit Artikeln verarbeitet, sie formatiert und anschließend den Summenwert ausgibt. Dabei können die verarbeiteten Artikel im Programm Datentypen mit ganz unterschiedlichen Eigenschaften sein: Video- und DVD-Artikeldatentypen speichern neben dem Titel auch die Lauflänge und den Hauptdarsteller; Bücher-Artikeldatentypen speichern stattdessen den Autor und haben obendrein einen anderen Mehrwertsteuersatz.

Ohne Polymorphie wäre die Erstellung ein vergleichsweise schwieriges Unterfangen. Sie müssten eine Art »Superset« schaffen, das alle Eigenschaften beherrscht und für die jeweils geforderten Sonderfälle programmieren. Regelrechter Spaghetti-Code wäre dabei buchstäblich vorprogrammiert.

Die Listen, die in simplen Textdateien vorliegen, sollen in diesem Beispiel ein bestimmtes Format ausweisen – Ihr Hauptprogramm muss dieses Format berücksichtigen und intern die Daten entsprechend speichern. Eine Liste, wie Sie sie von anderen Mitarbeitern Ihrer Abteilung als Textdatei bekommen, sieht typischerweise wie folgt aus:

```
;Inventarliste, die durch das Programm Inventory ausgewertet werden kann
;Format:
;Typ (1=Buch, 2=Cd oder Video), Bestellnummer, Titel, Zusatz, Brutto in Cent, Zusatz2
1;0001;Die Nachwächter;Terry Pratchett und Andreas Brandhorst;1990;
1;0002;Kristall der Träume;Barbara Wood;2490;
1;0003;Volle Deckung - Mr. Bush: Dude where is my country;Michael Moore;1290;
1;0004;Du bist nie allein;Nicholas Sparks und Ulrike Thiesmeyer;1900;
2;0005;X-Men 2 - Special Edition;128;2299;Patrik Steward
2;0006;Sex and the City: Season 5;220;2999;Sarah Jessica Parker
2;0007;Indiana Jones (Box Set 4 DVDs);359;4499;Harrison Ford
2;0008;Die Akte;135;1499;Julia Roberts
```

Sie sehen: Die Liste enthält verschiedene Artikeltypen, und das muss berücksichtigt werden. Ihr Programm muss die einzelnen Artikel der Liste verarbeiten und sollte anschließend eine neue Liste mit folgendem Format ausspucken:

```
0001  Die Nachwächter
18,60 Euro      1,30 Euro      18,60 Euro

0002  Kristall der Träume
23,27 Euro      1,63 Euro      23,27 Euro

0003  Volle Deckung - Mr. Bush: Dude where is my country
12,06 Euro      0,84 Euro      12,06 Euro

0004  Du bist nie allein
17,76 Euro      1,24 Euro      17,76 Euro

0005  X-Men 2 - Special Edition
19,82 Euro      3,17 Euro      19,82 Euro

0006  Sex and the City: Season 5
25,85 Euro      4,14 Euro      25,85 Euro
```

0007 Indiana Jones (Box Set 4 DVDs)  
38,78 Euro 6,21 Euro 38,78 Euro

0008 Die Akte  
12,92 Euro 2,07 Euro 12,92 Euro

-----  
Gesamtsumme: 189,66 Euro  
-----

Die Artikeltypen unterscheiden sich dabei in zweierlei Hinsicht: Zum einen gibt es bei Büchern einen anderen Mehrwertsteuersatz. Zum anderen sind es bei Büchern die Autoren, die in die Artikelinfo einlaufen, bei Filmen ist es die Lauflänge. Das Programm muss obendrein die Möglichkeit bieten, kurze und ausführliche Listen zu erstellen. In der ausführlichen Liste sollen dann die erweiterten Eigenschaften der einzelnen Artikeltypen zu sehen sein. Eine ausführliche Liste sieht folgendermaßen aus:

0001 Die Nachwächter  
Autor: Terry Pratchett und Andreas Brandhorst  
18,60 Euro 1,30 Euro 18,60 Euro

0002 Kristall der Träume  
Autor: Barbara Wood  
23,27 Euro 1,63 Euro 23,27 Euro

0003 Volle Deckung - Mr. Bush: Dude where is my country  
Autor: Michael Moore  
12,06 Euro 0,84 Euro 12,06 Euro

0004 Du bist nie allein  
Autor: Nicholas Sparks und Ulrike Thiesmeyer  
17,76 Euro 1,24 Euro 17,76 Euro

0005 X-Men 2 - Special Edition  
Laufzeit: 128 Min.  
Hauptdarsteller: Patrik Steward  
19,82 Euro 3,17 Euro 19,82 Euro

0006 Sex and the City: Season 5  
Laufzeit: 220 Min.  
Hauptdarsteller: Sarah Jessica Parker  
25,85 Euro 4,14 Euro 25,85 Euro

0007 Indiana Jones (Box Set 4 DVDs)  
Laufzeit: 359 Min.  
Hauptdarsteller: Harrison Ford  
38,78 Euro 6,21 Euro 38,78 Euro

0008 Die Akte  
Laufzeit: 135 Min.  
Hauptdarsteller: Julia Roberts  
12,92 Euro 2,07 Euro 12,92 Euro

Zur Programmplanung: Da Sie vorher (also während der Entwurfszeit Ihres Programms) nicht wissen, wie viele Artikel Ihnen eine Datei zur Verfügung stellt, müssen Sie den Artikelspeicher dynamisch gestalten. Dazu gibt es zwei Möglichkeiten:

- Sie lesen die Datei komplett von vorne bis hinten durch und finden, noch bevor Sie einen Artikel verarbeitet haben, heraus, wie viele Artikel Sie verarbeiten müssen. Dann dimensionieren Sie ein Array in der Größe, die der Anzahl der Artikel entspricht, die Sie ja jetzt kennen. Anschließend lesen Sie in einem zweiten Durchgang die Artikel in das Array ein.
- Oder: Sie schaffen eine Klasse zum Speichern der Artikel, die sich dynamisch vergrößert. Die Klasse selbst könnte beispielsweise ein Array vordefinieren, mit einer Initialgröße von beispielsweise 4 Elementen. Wenn diese 4 Elemente nicht mehr ausreichen, legt sie ein neues temporäres Array an, dann beispielsweise mit 8 Elementen, kopiert die vorhandenen 4 Elemente des »alten« Arrays in das neue und tauscht anschließend die beiden Arrays aus, sodass das alte Member-Array der Klasse nun Platz für 8 Elemente hat. Das Kopieren der Arrayelemente von einem zum anderen Array scheint nur auf den ersten Blick zeitintensiv –dieser Vorgang kopiert, wie wir im vorherigen Abschnitt kennen gelernt haben, aber nur die Zeiger auf die eigentlichen Artikeldaten, und das absolvieren moderne 32- und 64-Bit-Prozessoren in Lichtgeschwindigkeit.

Würden Sie sich für die erste Option entscheiden, müsste die zu importierende Datei ein zweites Mal verarbeitet werden, was gerade bei größeren Dateien natürlich viel zeitintensiver ist.

Die Artikel selbst werden ebenfalls in Klassen gespeichert. Da beide Artikel viele Gemeinsamkeiten aufweisen und sich nur marginal voneinander unterscheiden, liegt es nahe, eine so genannte Basis-Klasse zu entwerfen, die die Gemeinsamkeiten abdeckt und die Sonderfälle der jeweiligen Artikeltypen in zwei davon abgeleiteten Klassen zu implementieren. Wichtig für die Typsicherheit: Die »Listenklasse«, die die einzelnen Artikeldatentypen speichert, sollte nur Artikelklassen und von ihr abgeleitete Klassen aufnehmen.

Polymorphie ist in unserem Beispiel von unschätzbarem Wert. Die Artikelklassen, die von der Artikelbasisklasse abgeleitet sind, lassen sich nämlich über eine Objektvariable der Artikelbasisklasse steuern. Für die Entwicklung bedeutet das: Sie können ein Array verwalten, das nur den Typ »Artikelbasisklasse« aufnimmt, obwohl ganz andere (na ja, vielleicht nicht *ganz* andere, sondern nur erweiterte) Klasseninstanzen darin gespeichert werden. Der entscheidende Clou dabei ist, dass Sie im Code zwar beispielsweise eine Methode der Artikelbasisklasse für das Ausführen einer bestimmten Funktion angeben, dann aber doch die (andere) Methode einer abgeleiteten Klasse aufgerufen wird, wenn diese in der Basisklasse überschrieben wurde.

---

**BEGLEITDATEIEN:** Das mag zunächst ein wenig verwirrend klingen, wird aber klar, wenn Sie das Beispielprogramm zur Hilfe nehmen, das Sie unter dem Projektmappen-Namen *Inventory01* im Verzeichnis *.\VB 2005 - Entwicklerbuch\D - OOP\Kap09\Inventory01* finden.

---

Doch bevor wir uns mit dem Polymorphieaspekt dieses Beispiels beschäftigen, lassen Sie uns zunächst die Techniken klären, mit denen die Voraussetzungen zum Speichern der Daten geschaffen werden. Betrachten Sie dazu als erstes die Klasse *DynamicList* zur Speicherung der Artikel:

## Class DynamicList

```
Protected myStep As Integer = 4          ' Schrittweite, die das Array erhöht wird.
Protected myCurrentArraySize As Integer ' Aktuelle Array-Größe
Protected myCurrentCounter As Integer   ' Zeiger auf aktuelles Element
Protected myArray() As ShopItem         ' Array mit den Elementen.

Sub New()
    myCurrentArraySize = myStep
    ReDim myArray(myCurrentArraySize - 1)
End Sub

Sub Add(ByVal Item As ShopItem)

    'Element im Array speichern
    myArray(myCurrentCounter) = Item

    'Zeiger auf nächstes Element erhöhen
    myCurrentCounter += 1

    'Prüfen, ob aktuelle Arraygrenze erreicht wurde
    If myCurrentCounter = myCurrentArraySize - 1 Then
        'Neues Array mit mehr Speicher anlegen,
        'und Elemente hinüberkopieren. Dazu:

        'Neues Array wird größer:
        myCurrentArraySize += myStep

        'temporäres Array erstellen
        Dim locTempArray(myCurrentArraySize - 1) As ShopItem

        'Elemente kopieren; das geht mit dieser
        'statischen Methode extrem schnell, da zum Einen nur die
        'Zeiger kopiert werden, zum anderen diese Routine
        'intern nicht in Managed Code sondern nativem Assembler ausgeführt wird.
        Array.Copy(myArray, locTempArray, myArray.Length)

        'Auch hier werden nur die Zeiger auf die Elemente "verbogen".
        'Die vorherige Liste der Zeiger in myArray, die nun verwaist ist,
        'fällt dem Garbage Collector zum Opfer.
        myArray = locTempArray
    End If
End Sub

'Liefert die Anzahl der vorhandenen Elemente zurück
Public Overridable ReadOnly Property Count() As Integer
    Get
        Return myCurrentCounter
    End Get
End Property
```

```

'Erlaubt das Zuweisen
Default Public Overridable Property Item(ByVal Index As Integer) As ShopItem
    Get
        Return myArray(Index)
    End Get

    Set(ByVal Value As ShopItem)
        myArray(Index) = Value
    End Set
End Property
End Class

```

Die *Add*-Methode in dieser Klasse ist das Entscheidende. Sie überprüft, ob das Array noch ausreichend groß ist. Falls das nicht der Fall ist, führt sie den Tauschvorgang mit einer lokalen Arrayvariablen durch, die entsprechend größer definiert wurde und in die die Arrayelemente zuvor kopiert wurden. Damit »zeigt« der Klassen-Member *myArray* jetzt auf die neuen Arrayelemente – die Zeiger auf die alten Array-Elemente werden buchstäblich »vergessen«. Übrigens: Den Arbeitsspeicher, der auf diese Weise unnötigerweise belegt wird, holt sich das Framework eigenständig mithilfe der so genannten Garbage Collection (»Müllabfuhr«) wieder. Der Garbage Collector (kurz »GC«) schaut sich – vereinfacht ausgedrückt – an, ob Objekte, die Speicher belegen, noch irgendeinen Bezug zu einer verwendeten Objektvariablen haben. Falls nicht, werden sie entsorgt. In diesem Beispiel haben die Zeiger auf die ursprünglichen Elemente, die im Member-Array *myArray* gespeichert waren, nach dem Kopieren keinen Bezug mehr zu irgendeinem Objekt: Der Objektname wurde mit der Zeile

```

'temporäres Array dem Memberarray zuweisen
myArray = locTempArray

```

auf die neuen Elemente »umgebogen«. Die ursprünglichen Elemente stehen anschließend bezugslos im Speicher und werden beim nächsten GC-Durchlauf entsorgt.

---

**WICHTIG:** Behalten Sie im Hintergrund, dass die Daten der eigentlichen Objektinstanzen bei diesem Vorgang überhaupt nicht angetastet werden, und das ist auch der Grund, weswegen das Kopieren eines Arrays mit `Array.Copy` so unglaublich schnell vonstatten geht (was ebenfalls der Fall wäre, würden Sie den Vorgang selbst in die Hand nehmen, das ganze Array in einer `For`-Schleife durchlaufen, und die einzelnen Elemente dem neuen Array zuweisen).

---

Die *Add*-Methode nimmt ausschließlich Objekte eines bestimmten Typs entgegen. In diesem Beispiel habe ich sie *ShopItem* (»Ladenartikel«) genannt. Diese Klasse stellt die Basis für die Artikelspeicherung dar und sieht folgendermaßen aus:

```

Class ShopItem

    Protected myTitle As String           ' Titel
    Protected myNetPrice As Double       ' Nettopreis
    Protected myOrderNumber As String    ' Artikelnummer
    Protected myPrintTypeSetting As PrintType ' Ausgabeform

    Public Sub New()
        myPrintTypeSetting = PrintType.Detailed
    End Sub

```

```

Public Sub New(ByVal StringArray() As String)
    Title = StringArray(FieldOrder.Titel)
    'FieldOrder ist übrigens eine Enum und wird weiter unten definiert:
    GrossPrice = Double.Parse(StringArray(FieldOrder.GrossPrice)) / 100
    OrderNumber = StringArray(FieldOrder.OrderNumber)
    PrintTypeSetting = PrintType.Detailed
End Sub

Public Property Title() As String
    Get
        Return myTitle
    End Get
    Set(ByVal Value As String)
        myTitle = Value
    End Set
End Property

Public Property OrderNumber() As String
    Get
        Return myOrderNumber
    End Get
    Set(ByVal Value As String)
        myOrderNumber = Value
    End Set
End Property

Public Property NetPrice() As Double
    Get
        Return myNetPrice
    End Get

    Set(ByVal Value As Double)
        myNetPrice = Value
    End Set

End Property

Public ReadOnly Property NetPriceFormatted() As String
    Get
        Return NetPrice.ToString("#,##0.00") + " Euro"
    End Get
End Property

Public Overridable Property GrossPrice() As Double
    Get
        Return myNetPrice * 1.16
    End Get

    Set(ByVal Value As Double)
        myNetPrice = Value / 1.16
    End Set
End Property

```

```

Public ReadOnly Property GrossPriceFormatted() As String
    Get
        Return GrossPrice.ToString("#,##0.00") + " Euro"
    End Get
End Property

Public ReadOnly Property VATAmountFormatted() As String
    Get
        Return (GrossPrice - myNetPrice).ToString("#,##0.00") + " Euro"
    End Get
End Property

Public Overridable ReadOnly Property Description() As String
    Get
        Return OrderNumber & vbTab & Title
    End Get
End Property

Public Property PrintTypeSetting() As PrintType
    Get
        Return myPrintTypeSetting
    End Get

    Set(ByVal Value As PrintType)
        myPrintTypeSetting = Value
    End Set
End Property

Public Overrides Function ToString() As String

    If PrintTypeSetting = PrintType.Brief Then
        'Kurzform: Es wird in jedem Fall
        'die Description-Eigenschaft des Objektes
        'verwendet
        Return MyClass.Description & vbCr & vbLf & _
            Me.NetPriceFormatted & vbTab & Me.VATAmountFormatted & vbTab & _
            Me.GrossPriceFormatted & vbCr & vbLf
    Else
        'Langform: Die Description Eigenschaft des Objektes
        'selber wird verwendet
        Return Me.Description & vbCr & vbLf & _
            Me.NetPriceFormatted & vbTab & Me.VATAmountFormatted & vbTab & _
            Me.GrossPriceFormatted & vbCr & vbLf
    End If

    End Function

End Class

```

Es ist nicht sonderlich schwer, diese Klasse zu begreifen, denn sie dient nur zwei Aufgaben: Dem Speichern von Daten, die durch Eigenschaften zugänglich gemacht werden, und der formatierten Ausgabe einiger dieser Daten. Einige der Eigenschaften sind obendrein nur als `ReadOnly` definiert, da es keinen Sinn ergäbe, sie zu beschreiben.

## Zahlen mit ToString formatiert in Zeichenketten umwandeln

Erwähnenswert an dieser Stelle ist die Eigenschaft der ToString-Funktion primitiver Datentypen (Integer, Double, etc.), Zahlen formatiert auszugeben. In diesem Fall verwenden Sie eine Überladung von ToString, die einen String als Parameter akzeptiert. In diesem Beispiel lautet der String »#,##0.00«, der bewirkt, dass Nachkommastellen unabhängig vom Wert grundsätzlich zweistellig, Vorkommastellen im Bedarfsfall mit Tausendertrennzeichen formatiert werden. Bitte beachten Sie, dass Sie hierbei die amerikanische/englische Schreibweise verwenden, bei der das Tausendertrennzeichen ein Komma und das Nachstellentrennzeichen ein Punkt ist. Mehr zu diesem Thema finden Sie übrigens in ► Kapitel 17.

Interessant ist es, als nächstes die aus der Basisklasse abgeleiteten Klassen zu erkunden, die Sie im Folgenden abgedruckt finden:

```
Class BookItem
    Inherits ShopItem

    Protected myAuthor As String

    Public Sub New(ByVal StringArray() As String)
        MyBase.New(StringArray)
        Author = StringArray(FieldOrder.AdditionalRemarks1)
    End Sub

    Public Overridable Property Author() As String
        Get
            Return myAuthor
        End Get
        Set(ByVal Value As String)
            myAuthor = Value
        End Set
    End Property

    Public Overrides Property GrossPrice() As Double
        Get
            Return myNetPrice * 1.07
        End Get

        Set(ByVal Value As Double)
            myNetPrice = Value / 1.07
        End Set
    End Property

    Public Overrides ReadOnly Property Description() As String
        Get
            Return OrderNumber & vbTab & Title & vbCr & vbLf & "Autor: " & Author
        End Get
    End Property

End Class
```

Durch das `Inherits`-Schlüsselwort direkt nach dem `Class`-Schlüsselwort bestimmen Sie, dass die neue Klasse, die die Bücherartikel speichert, von der Basisklasse `ShopItem` abgeleitet wird. Das befähigt Objektvariablen, die als `ShopItem` definiert wurden, auch Instanzen von `BookItem` zu referenzieren, denn es gilt: Jede abgeleitete Klasse kann durch eine Objektvariable der Basisklasse »angesprochen« werden.

Dazu ein Beispiel: Wenn Sie eine Objektvariable namens `EinArtikel` auf folgende Weise deklariert

```
Dim EinArtikel as ShopItem
```

und entsprechend, etwa durch

```
EinArtikel=New ShopItem()
```

definiert haben, kann ihr Inhalt später im Programm durch die Anweisung

```
Console.WriteLine(EinArtikel.GrossPriceFormatted)
```

ausgegeben werden, und es wird, wie zu erwarten, `GrossPriceFormatted` (etwa: »Bruttopreis formatiert«) der Basisklasse `ShopItem` verwendet.

Wird hingegen – und jetzt wird es interessant – eine Instanz der abgeleiteten Klasse etwa durch die folgende Zeile

```
EinArtikel=New BookItem(...)
```

durch **dieselbe** Objektvariable `EinArtikel` angesprochen, und wird später die gleiche Funktion aufgerufen, etwa durch

```
Console.WriteLine(EinArtikel.GrossPriceFormatted)
```

so wird dieses Mal nicht die `GrossPriceFormatted`-Funktion der Basisklasse, sondern die der abgeleiteten Klasse `BookItem` verwendet. Und genau das sind die unschlagbaren Vorteile der Polymorphie: Sie haben ein Steuerprogramm, das in einer Basisklassenobjektvariablen die augenscheinlich gleiche Funktion aufruft und doch können Sie durch das Überschreiben genau dieser Funktion in einer abgeleiteten Klasse ein anderes Ergebnis erzielen.

Die Basisklasse unseres Beispiels stellt einen parametrisierten Konstruktor bereit, die ein `String-Array` übernimmt. Aus diesem `String-Array` werden die Daten für den Inhalt einer Klasseninstanz entnommen:

```
Public Sub New(ByVal StringArray() As String)
    Title = StringArray(FieldOrder.Title)
    GrossPrice = Double.Parse(StringArray(FieldOrder.GrossPrice)) / 100
    OrderNumber = StringArray(FieldOrder.OrderNumber)
    PrintTypeSetting = PrintType.Detailed
End Sub
```

Das übergebende Array enthält die einzelnen Daten immer in Elementen mit dem gleichen Index, die zum einfacheren Verständnis des Quellcodes übrigens in einer `Enum`-Aufzählung festgehalten sind (mehr zum Thema *Enums* erfahren Sie in ► Kapitel 18):

```
Enum FieldOrder ' Zuständig für die Reihenfolge der Felder
    Type
    OrderNumber
    Titel
```

```

    AdditionalRemarks1
    GrossPrice
    AdditionalRemarks2
End Enum

```

Die abgeleitete Klasse `BookItem` hat nun viel weniger Arbeit, eine Instanz zu definieren. Sie ruft einfach den Konstruktor der Basisklasse auf und ergänzt ihren eigenen Konstruktor nur noch um die Zuweisung eines bestimmten Array-Elementes des ihr übergebenen Parameters:

```

Public Sub New(ByVal StringArray() As String)
    MyBase.New(StringArray)
    Author = StringArray(FieldOrder.AdditionalRemarks1)
End Sub

```

Da Bücher im Gegensatz zu vielen anderen Artikeln mit einer anderen Mehrwertsteuer belegt sind (jedenfalls noch), muss die Funktion, die den Bruttopreis berechnet, überschrieben werden:

```

Public Overrides Property GrossPrice() As Double
    Get
        Return myNetPrice * 1.07
    End Get

    Set(ByVal Value As Double)
        myNetPrice = Value / 1.07
    End Set
End Property

```

Und schon wieder sehen Sie Polymorphie in Aktion: Die abgeleitete Klasse muss jetzt die Methode, die den formatierten Bruttopreis als `String` zurückgibt, nicht noch zusätzlich überschreiben, denn wenn Sie sich die Funktion der Basisklasse betrachten

```

Public ReadOnly Property GrossPriceFormatted() As String
    Get
        Return GrossPrice.ToString("#,##0.00") + " Euro"
    End Get
End Property

```

stellen Sie fest, dass sie nicht direkt auf eine Member-Variable zurückgreift, sondern ihrerseits eine in der Klasse implementierte Eigenschaft bemüht. Aus der Sicht der abgeleiteten Klasse `BookItem` wird hier nicht `GrossPrice` der Basisklasse, sondern der (überschriebenen) abgeleiteten Klasse aufgerufen – der formatierte Bruttopreis eines Buches berücksichtigt also korrekt 7 % Mehrwertsteuer und nicht die 16 % der Basisklasse. `GrossPriceFormatted` ruft also die überschriebene Funktion der abgeleiteten Klasse auf, obwohl diese Funktion ausschließlich in der Basisklasse zu finden ist!

Eine ähnliche Vorgehensweise legen die Funktionen `Description` (für die Zusammensetzung des Beschreibungstextes) und `VATAmountFormatted` (für den Betrag der Mehrwertsteuer) an den Tag.

Und auch die zweite Klasse zur Speicherung von Artikeln – sie nennt sich `DVDItem` – macht sich die Polymorphie zunutze:

```

Class DVDItem
    Inherits ShopItem

    Protected myRunningTime As Integer
    Protected myActor As String

```

```

Public Sub New(ByVal StringArray() As String)
    MyBase.New(StringArray)
    RunningTime = Integer.Parse(StringArray(FieldOrder.AdditionalRemarks1))
    Actor = StringArray(FieldOrder.AdditionalRemarks2)
End Sub

Public Overridable Property RunningTime() As Integer
    Get
        Return myRunningTime
    End Get
    Set(ByVal Value As Integer)
        myRunningTime = Value
    End Set
End Property

Public Overridable Property Actor() As String
    Get
        Return myActor
    End Get
    Set(ByVal Value As String)
        myActor = Value
    End Set
End Property

Public Overrides ReadOnly Property Description() As String
    Get
        Return OrderNumber & vbTab & Title & vbCrLf & vbLf & "Laufzeit: " & myRunningTime & " Min." & vbCrLf
        & vbLf & "Hauptdarsteller: " & Actor
    End Get
End Property
End Class

```

Sie sehen an diesem Beispiel wie sehr Sie die Polymorphie bei der Wiederverwendbarkeit von Code unterstützt und enorm Arbeit spart. Und das gleich in doppeltem Sinne: Wenn die Basisklasse funktioniert, funktionieren die abgeleiteten Klassen bis auf ihre zusätzliche Funktionalität genau so reibungslos. Neben dem weniger vorhandenen Tippaufwand sparen Sie obendrein viel Zeit beim Suchen von Fehlern.

Die Klassen für unser Beispiel sind damit komplett fertig gestellt. Was jetzt noch zu tun bleibt, ist die Implementierung des Hauptprogramms, das die Ursprungstextdatei einliest, die Elemente aus jeder Textzeile erstellt, sie der Liste hinzufügt und die Ergebnisse schließlich ausgibt:

```

Module mdlMain

    'Die Inventardatei muss im Programmverzeichnis stehen.
    Private Filename As String = My.Application.Info.DirectoryPath & "\Inventar.txt"

    Sub Main()

        'StreamReader zum Einlesen der Textdateien
        Dim locSr As StreamReader
        Dim locList As New DynamicList ' Die Dynamische Liste
    End Sub
End Module

```

```

Dim locElements() As String    ' Die einzelnen ShopItem-Elemente
Dim locShopItem As ShopItem    ' Ein einzelnes Shop-Element
Dim locDisplayMode As PrintType ' Der Darstellungsmodus

'Schauen, ob die Textdatei vorhanden ist:
Try
    locSr = New StreamReader(Filename, System.Text.Encoding.Default)
Catch ex As Exception
    Console.WriteLine("Fehler beim Lesen der Inventardatei!" & _
        vbNewLine & ex.Message)
    Console.WriteLine()
    Console.WriteLine("Taste drücken zum Beenden")
    Console.ReadKey()
    Exit Sub
End Try

Console.WriteLine("Wählen Sie (1) für kurze und (2) für ausführliche Darstellung")
Dim locKey As Char = Console.ReadKey.KeyChar
If locKey = "1"c Then
    locDisplayMode = PrintType.Brief
Else
    locDisplayMode = PrintType.Detailed
End If

Do
    Try
        'Zeile einlesen
        Dim locLine As String = locSr.ReadLine()

        'Nichts eingegeben, dann war's das!
        If String.IsNullOrEmpty(locLine) Then
            locSr.Close()
            Exit Do
        End If

        'Semikolon überlesen
        If Not locLine.StartsWith(";") Then

            'So braucht man kein explizites Char-Array zu deklarieren
            'um die Zeile in die durch Komma getrennten Elemente zu zerlegen
            locElements = locLine.Split(New Char() {";"c})

            If locElements(FieldOrder.Type) = "1" Then
                locShopItem = New BookItem(locElements)
            Else
                locShopItem = New DVDItem(locElements)
            End If
            locList.Add(locShopItem)
        End If

    Catch ex As Exception
        Console.WriteLine("Fehler beim Auswerten der Inventardatei!" & _
            vbNewLine & ex.Message)
    End Try
Loop

```

```

        Console.WriteLine()
        Console.WriteLine("Taste drücken zum Beenden")
        Console.ReadKey()
        locSr.Close()
        Exit Sub
    End Try

Loop

Dim locGrossAmount As Double = 0

'Alle Elemente ausgeben
For count As Integer = 0 To locList.Count - 1
    locList(count).PrintTypeSetting = locDisplayMode
    Console.WriteLine(locList(count).ToString())
    locGrossAmount += locList(count).GrossPrice
Next

Console.WriteLine()
Console.WriteLine("-----")
Console.WriteLine("Gesamtsumme: " & locGrossAmount.ToString("#,##0.00") & " Euro")
Console.WriteLine("-----")
Console.WriteLine("-----")
Console.WriteLine()
Console.WriteLine("Return drücken, zum Beenden")
Console.ReadLine()

End Sub

End Module

```

Sie sehen: Den meisten Platz beanspruchen die Print-Zeilen, die sich um die Ausgaben kümmern und die Artikel auf dem Bildschirm darstellen. Da die überwiegende Lösung des Problems bereits von den Artikelklassen bewältigt wird, beschränkt sich die eigentliche Aufgabe des Hauptprogramms darauf, die Datei zu öffnen, ein Element aus der Klasse zu lesen und zu speichern.

Auch das Hauptprogramm bedient sich der Polymorphie, nämlich dann, wenn es die Liste auf dem Bildschirm ausgibt:

```

'Alle Elemente ausgeben
For count As Integer = 0 To locList.Count - 1
    locList(count).PrintTypeSetting = locDisplayMode
    Console.WriteLine(locList(count).ToString())
    locGrossAmount += locList(count).GrossPrice
Next

```

Es iteriert in einer Zählschleife durch die einzelnen Elemente der Klasse. Da die Item-Eigenschaft der DynamicList-Klasse die Default-Eigenschaft ist, muss der Eigenschaftename Item an dieser Stelle noch nicht einmal angegeben werden – es reicht aus, das gewünschte Element durch eine direkt an den Objektnamen angefügte Klammer zu indizieren.

Mit ToString des indizierten Objektes erfolgt anschließend der Ausdruck auf dem Bildschirm. Welches ToString das Programm dabei verwendet, ist wieder abhängig von der Klasse, deren Instanz im Arrayelement gespeichert wird. Ist es ein BookItem-Objekt, wird ToString von BookItem aufgerufen; bei einem DVDItem-Objekt wird die ToString-Funktion eben dieser Klasse aufgerufen. Gleiches gilt anschließend für die Berechnung der Mehrwertsteuer und die Funktion GrossPrice.

## Polymorphie und der Gebrauch von Me, MyClass und MyBase

Die Option, dem Benutzer zur Verfügung zu stellen, entweder eine ausführliche Liste oder eine sehr kurz gehaltene Liste auszudrucken, ist eine der Anforderungen an das Programm. Nun gäbe es zwei theoretische Ansätze, die Lösung dieses Problems zu realisieren. Die erste Möglichkeit: Der Ausdruck wird komplett durch das steuernde Hauptprogramm durchgeführt. In diesem Fall müsste das Hauptprogramm dafür sorgen, dass die Sonderfälle unterschieden würden. Im Prinzip gäbe es dabei zwei verschiedene Druckroutinen, die jeweils einmal für den ausführlichen und einmal für den kompakten Ausdruck zuständig wären. Möglichkeit Nr. 2: Die Klassen selbst sorgen für die Aufbereitung der entsprechenden Texte.

Zufälligerweise gibt es eine Eigenschaft in der Basisklasse, die einen recht kompakten, beschreibenden Text für den Inhalt einer Objektinstanz zurückliefert. Diese Eigenschaft hat den Namen Description. Diese Eigenschaft wird von den abgeleiteten Klassen überschrieben; sie erweitern die Eigenschaft dahingehend, dass auch die zusätzlichen gespeicherten Informationen bei der Ausgabe berücksichtigt werden.

Viel leichter wäre es jetzt also, wenn das Programm in Abhängigkeit von der Eingabe des Anwenders entweder die Eigenschaft der Basisklasse oder die der jeweiligen abgeleiteten Klasse für die Ausgabe auf den Bildschirm verwenden würde. Die Basisklasse müsste zu diesem Zweck eine weitere Eigenschaft bereitstellen, mit der sich steuern ließe, ob die kompakte oder die ausführliche Form bei der Ausgabe der Artikelbeschreibung zu berücksichtigen ist.

Aus diesem Grund gibt es bereits in der Basisklasse eine Eigenschaft namens PrintType, die nur zwei Zustände aufweisen kann, welche in einer Enum definiert sind:

```
Enum PrintType ' Reportform
    Brief       ' kurz
    Detailed    ' ausführlich
End Enum
```

Bevor nun die eigentliche Ausgabe auf dem Bildschirm des Inhalts eines Artikelobjektes erfolgt, muss das die Ausgabe steuernde Programm dafür sorgen, dass die PrintType-Eigenschaft auf den korrespondierenden Wert gesetzt wird, den der Anwender beim Start des Programms definiert hat. Wenn diese Voraussetzung erfüllt ist, kann ein Artikelobjekt selber entscheiden, ob die Beschreibung der Basisklasse oder der eigenen Klasse zurückgeliefert werden soll. Da dieser »Entscheidungsalgorithmus« sowohl in der Basisklasse als auch in allen abgeleiteten Klassen derselbe ist, reicht es aus, ihn ausschließlich in der Basisklasse zu implementieren. Durch die Polymorphie ist es anschließend möglich, die »richtige« Description-Eigenschaft der jeweiligen Klasse abzufragen und daraus den Rückgabertext zusammenzubasteln.

Das letzte Problem ist die Realisierung der kompakten Form des Artikeldrucks. Hier müsste der ToString-Funktion die Möglichkeit gegeben sein, die Polymorphie außer Kraft zu setzen und gezielt

die Description-Eigenschaft der Basisklasse aufzurufen, egal ob es sich bei dem gespeicherten Objekt um ein abgeleitetes oder um ein Original handelt. Genau das erreichen Sie mit dem Schlüsselwort `MyClass`. `MyClass` erlaubt, gezielt auf ein Element der Klasse zuzugreifen, in der `MyClass` »steht«. Sie heben damit quasi die Überschreibung einer Methode für die Dauer des Aufrufs auf und verwenden das Original der Klasse, in der `MyClass` eingesetzt wird. Die `ToString`-Funktion der Basisklasse macht sich genau diese Eigenschaft zunutze:

```
Public Overrides Function ToString() As String

    If PrintTypeSetting = PrintType.Brief Then
        'Kurzform: Es wird in jedem Fall
        'die Description-Eigenschaft des Objektes
        'verwendet.
        Return MyClass.Description & vbCrLf & _
            Me.NetPriceFormatted & vbTab & Me.VATAmountFormatted & vbTab & _
            Me.GrossPriceFormatted & vbCrLf
    Else
        'Langform: Die Description Eigenschaft des Objektes
        'selber wird verwendet.
        Return Me.Description & vbCrLf & _
            Me.NetPriceFormatted & vbTab & Me.VATAmountFormatted & vbTab & _
            Me.GrossPriceFormatted & vbCrLf
    End If
End Function
```

Die nachstehende Tabelle fasst die verschiedenen Schlüsselworte zusammen, die den Klassenebenen-zugriff spezifizieren.

Schlüsselwort	Beschreibung
<code>Me</code>	Das Element der eigenen Klasse wird verwendet.
<code>MyClass</code>	Das Element der Klassenableitung, die das <i>MyClass</i> -Schlüsselwort enthält, wird verwendet.
<code>MyBase</code>	Das Element der Basisklasse wird verwendet.

**Tabelle 9.1:** Schlüsselworte zur Bestimmung von überschriebenen Elementen unterschiedlichen Rangs in der Vererbungshierarchie einer Klasse

## Abstrakte Klassen und virtuelle Prozeduren

Schreiben Sie Ihre Texte auch mit Microsoft Word? Kennen Sie dann auch Dokumentenvorlagen? Ja? Gut. Dann wissen Sie quasi auch, was abstrakte Klassen sind. Dokumentenvorlagen in Word dienen dazu, eine Formatierungsrichtlinie für Dokumente festzuschreiben.

Wenn ich, wie in diesem Buch, einen »Standardabsatz« schreibe, dann bestimmt die Dokumentenvorlage, dass die Schrift dafür »Minion« und die Schriftgröße auf 10 Punkt gesetzt wird. Wenn ein Dokument auf Basis einer Dokumentenvorlage erstellt ist, übernimmt es alle ihre Eigenschaften. Nur so ist gewährleistet, dass die Kapitel eines Buches später auch alle im gleichen Stil formatiert sind.

Abstrakte Klassen funktionieren nach ähnlichem Konzept. Sie stellen Prototypen von Prozeduren bereit, aber sie dienen ausschließlich als Vorlage. Um Entwickler dabei – rabiati ausgedrückt – dazu zu zwingen, bestimmte Elemente neu zu implementieren, stellen sie zusätzlich virtuelle Prozeduren bereit. Eine virtuelle Prozedur hat zwei Aufgaben:

- Sie gibt dem Entwickler die Signatur (oder die Signaturen für überladene Prozeduren) einer Prozedur vor.
- Sie »zwingt« den Entwickler, in der Ableitung der Klasse die vorhandene Prozedur zu überschreiben und sie vor allen Dingen damit zu implementieren.

Ein Praxisbeispiel (und dazu muss ich gar nicht weit ausholen): Stellen Sie sich vor, das Beispielprogramm aus dem letzten Abschnitt verrichtet seinen Dienst in einem großen Versandhaus wie Otto, Quelle oder Amazon. Hier sind natürlich wesentlich mehr Artikelgruppen zu berücksichtigen – bei Versendern wie Amazon, die weltweit operieren, sind auch noch mehr Mehrwertsteuersätze zu berücksichtigen.

Es würde ziemlich viel Sinn ergeben, die vorhandene Artikelklasse `ShopItem` als abstrakte Klasse zu formulieren. Damit Teammitglieder gezwungen sind, die mehrwertsteuerbezogenen Funktionen neu zu implementieren (sodass sie nicht vergessen werden), läge es nahe, die betroffenen Funktionen als virtuelle Funktionen auszulegen. Sie stellen damit sicher, dass jeder Entwickler *bewusst* programmiert, denn implementiert er die mehrwertsteuerbezogenen Funktionen nicht, wird seine Klasse nicht kompilierbar sein.

An dem Beispielprogramm müssen Sie dabei kaum Änderungen vornehmen. Sie können nach dem Umbau `ShopItem` auch weiterhin als Objektvariable verwenden. Sie können es nur nicht mehr direkt instanzieren – aber das haben wir im Beispielprogramm ohnehin nicht gemacht (als hätte ich es kommen sehen ...).

## Eine Klasse mit `MustInherit` als abstrakt deklarieren

Um eine abstrakte Klasse zu erstellen, verwenden Sie das Schlüsselwort `MustInherit`. Wenn Sie dieses Schlüsselwort vor das Schlüsselwort `Class` der Klasse `ShopItem` in unserem Beispiel stellen, passiert erst einmal gar nichts. Sie haben damit zunächst einmal erreicht, dass die Klasse nur noch in einer Ableitung instanziiert werden kann. Da wir `ShopItem` selbst nicht instanzieren, bleibt alles beim Alten, und es gibt auch keine Fehlermeldungen.

Was Sie allerdings nun nicht mehr können (aber vorher hätten können), ist, die Klasse irgendwo im Programm zu instanzieren. Versuchen Sie es: Fügen Sie in der Sub `Main` folgende Zeile ein,

```
Dim locShopItemVersuchsInstanz As New ShopItem
```

so erhalten Sie eine Fehlermeldung etwa wie in Abbildung 9.10 zu sehen:

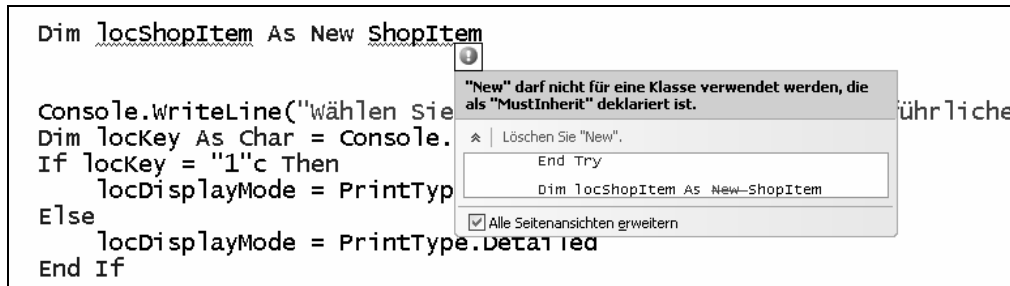


Abbildung 9.10: Eine mit *MustInherit* als abstrakt definierte Klasse kann nicht instanziiert werden

## Eine Methode oder Eigenschaft einer abstrakten Klasse mit *MustOverride* als virtuell deklarieren

Fehler wünschen Sie sich als Entwickler normalerweise eher weniger. Ist die Fehlerquelle jedoch eine virtuell definierte Methode oder Eigenschaft, sieht das allerdings anders aus: Eine Fehlermeldung tritt in der Regel deswegen auf, weil sie in einer abgeleiteten Klasse nicht überschrieben, ergo: neu implementiert wurde. Aber genau das wollen Sie mit virtuellen Methoden bzw. Eigenschaften erreichen.

In unserem Beispiel ist die Eigenschaft *GrossPrice*, die den Bruttopreis eines Artikels aus dem Nettopreis mit dem gültigen Mehrwertsteuersatz errechnet, eine willkommene Demonstration für eine virtuelle Eigenschaft, denn: Es ist wichtig, dass sich jeder Entwickler darüber im Klaren ist, dass er in einer Ableitung der Klasse *ShopItem* selbst für das korrekte Errechnen des Bruttopreises verantwortlich ist und eine Variante dieser Eigenschaft auf jeden Fall in seiner abgeleiteten Klasse implementiert.

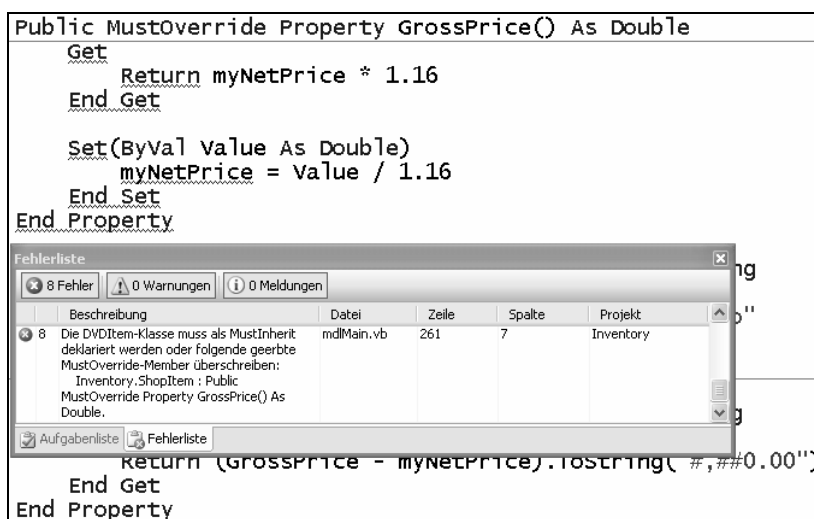


Abbildung 9.11: Mit *MustOverride* als virtuell gekennzeichnete Prozeduren dürfen keinen Code enthalten und müssen in abgeleiteten Klassen überschrieben werden

Sie deklarieren eine Prozedur mit dem Schlüsselwort *MustOverride* als virtuell. Wenn Sie die vorhandene Eigenschaft *GrossPrice* der Klasse *ShopItem* folgendermaßen abändern,

```
Public MustOverride Property GrossPrice() as Double
```

bekommen Sie ein paar – in diesem Fall – »gewünschte« Fehlermeldungen, etwa wie in Abbildung 3.19 zu sehen:

Ein Fehler liegt direkt in der Eigenschaftsprozedur, denn: Virtuelle Prozeduren selbst dürfen keinen Code enthalten. Sie dienen nur dazu, zu sagen: »Vergiss nicht, lieber Entwickler, du musst diese Prozedur in deiner abgeleiteten Klasse implementieren.« Und dass das gut funktioniert, zeigt die Fehlermeldung

Die *DVDItem*-Klasse muss als *MustInherit* deklariert werden oder folgende geerbte *MustOverride*-Member überschreiben:

```
Inventory.ShopItem : Public MustOverride Property GrossPrice() As Double.
```

an, denn es ist in der Tat richtig, dass die Klasse *DVDItem* diese Eigenschaft nicht implementiert. Stellen Sie sich vor, Videos und DVDs würden mit 15 % MwSt. besteuert – wie leicht hätte dem Entwickler »durchrutschen« können, die Eigenschaft neu zu implementieren! Die betroffenen Artikel wären in diesem Fall fälschlicherweise mit 16 % MwSt. besteuert worden.

Die Änderungen, die Sie am Programm durchführen müssen, halten sich in Grenzen:

- Sie entfernen den Code der Eigenschaftsprozedur *GrossPrice* in der Klasse *ShopItem*, sodass nur der Prototyp stehen bleibt (der Prozedurenkopf).
- Sie fügen eine Eigenschaftsprozedur *GrossPrice* in die Klasse *DVDItem* ein – eigentlich genau den Code, der sich zuvor in der Klasse *ShopItem* befand.

Danach sind alle Fehler eliminiert, und Ihr Programm ist bereit für die kommende Mehrwertsteuererhöhung bzw. »Subventionsstreichung« ;-).

---

**BEGLEITDATEIEN:** Das Programmbeispiel mit den Änderungen für abstrakte Klassen finden Sie übrigens im Verzeichnis `.\VB 2005 - Entwicklerbuch\D - OOP\Kap09\Inventory02`.

---

## Schnittstellen (Interfaces)

Das Konzept von Schnittstellen bietet Ihnen eine weitere Möglichkeit, Klassen zu standardisieren. Schnittstellen dienen dazu – ähnlich wie abstrakte Klassen – Vorschriften zu erstellen, dass innerhalb der Klassen, die von einer Schnittstelle ableiten, bestimmte Elemente zwingend erforderlich sind. Im Gegensatz zu abstrakten Klassen enthalten Schnittstellen jedoch nur diese Vorschriften; sie enthalten überhaupt keinen Code.

Da Schnittstellen keinerlei Code enthalten, können sie, genau wie abstrakte Klassen, nicht direkt instanziiert werden. Allerdings können Objektvariablen vom Typ einer Schnittstelle dazu verwendet werden, ableitende Klassen zu referenzieren. Dieses Verhalten erlaubt es, Klassenbibliotheken zu erstellen, die eine enorme Flexibilität aufweisen.

Durch die Eigenarten, die Schnittstellen aufweisen, werden sie gerade innerhalb des Frameworks zu zweierlei Zwecken verwendet:

- Sie dienen auf der einen Seite lediglich dazu, den Programmierer, der sie einbindet, an bestimmte Konventionen zu binden, ohne dass seine Klassen und die Objekte, die daraus entstehen, später durch Polymorphie vom Framework selber über diese Schnittstellen gesteuert würden. Man spricht in diesem Fall von einem so genannten »Interface Pattern« (Schnittstellenmuster).
- Sie dienen auf der anderen Seite dazu, allgemein gültige Verwalterklassen zu entwickeln, in der später alle die Objekte verwendet werden können, die die vorgegebenen Schnittstellen dafür implementieren.

Für den letzten Punkt gibt es dazu in der Regel eine Art Drei-Stufen-Konzept für Komponenten, die dem Entwickler zur Verfügung gestellt werden: Auf oberster Ebene gibt es eine Klasse, die die eigentliche Aufgabe übernimmt. Sie kann Objekte einbinden, die eine bestimmte Schnittstelle implementieren. Auf unterster Ebene stellt die Komponente dem Entwickler eben diese Schnittstelle(n) zur Verfügung. Damit der Entwickler, der die Komponente verwenden will, nicht die komplette Implementierung selber durchführen muss, sollte ihm die Komponente basierend auf den angebotenen Schnittstellen abstrakte Klassen zur Verfügung stellen, die eine gewisse Grundfunktionalität enthalten.

Ein Beispiel macht die Zusammenhänge klarer: Angenommen, Sie werden mit der Aufgabe betraut, eine Komponente zu entwickeln, die Daten tabellarisch auf dem Bildschirm darstellt. Diese Komponente – nennen wir Sie *Tabellensteuerelement* – ist prinzipiell aus zwei Unterkomponenten aufgebaut: Zum einen ist das die Komponente, die die Tabelle zeichnet, sie mit Gitternetzlinien versieht, dafür sorgt, dass der spätere Anwender einen Cursor in der Tabelle bewegen kann usw. Diese Komponente wird aber idealerweise nicht selbst dafür zuständig sein, den Inhalt einer Tabellenzelle zu malen, sondern diese Aufgabe einer weiteren Klasse überlassen und bindet sie lediglich ein. Die Aufgabe der weiteren Komponente auf der anderen Seite ist es, den Inhalt einer einzigen Tabellenzelle zu zeichnen. Diese zweite Komponente speichert also nicht nur die Daten für die einzelnen Tabellenzellen, sie sorgt auch dafür, dass diese Daten zu gegebener Zeit in Form einer Tabellenzelle auf den Bildschirm gemalt werden. Damit der spätere Entwickler, der das Tabellensteuerelement verwendet, die größtmögliche Flexibilität in seiner Verwendung hat, sollte die zweite Klasse – die Tabellenzelle – nach Möglichkeit nicht als festgeschriebene Klasse, sondern auf unterster Ebene auch als Schnittstelle zur Verfügung stellen. Auf der zweiten Ebene sollte das Tabellensteuerelement dem Entwickler auch mindestens eine auf der Schnittstelle basierende abstrakte Klasse zur Verfügung stellen, die die wichtigste Grundfunktionalität bereits enthält. Und schließlich stellt die Komponente auf oberster Ebene fix und fertige Tabellenzellenkomponenten zur Verfügung, mit denen der Entwickler direkt loslegen und Tabellen mit Daten füllen kann.

Stellt der Entwickler dann nach geraumer Zeit fest, dass die vorhandenen Zellenkomponenten nicht die Möglichkeiten bieten, die er benötigt, kann er sich entscheiden:

- Er kann entweder die abstrakte Klasse ableiten und daraus eine Tabellenzellenkomponente entwickeln, die seinen Anforderungen genügt. Der Aufwand dafür hält sich in Grenzen, weil die abstrakte Klasse einen Großteil des notwendigen Verwaltungscodes bereits zur Verfügung stellt, den er lediglich ergänzen muss.
- Falls ihn diese Vorgehensweise immer noch zu sehr einschränkt, kann er die *vollständige* Implementierung der Zellenklasse übernehmen. Er muss in diesem Fall die von der Tabellenkomponente zur Verfügung gestellte Schnittstelle einbinden, um sicherzustellen, dass alles an Funktionalität innerhalb seiner Zellenklasse vorhanden ist, was die Tabellenkomponente vorschreibt. Der Nachteil: Der Aufwand, dieses Vorhaben zu realisieren, ist logischerweise erheblich größer.

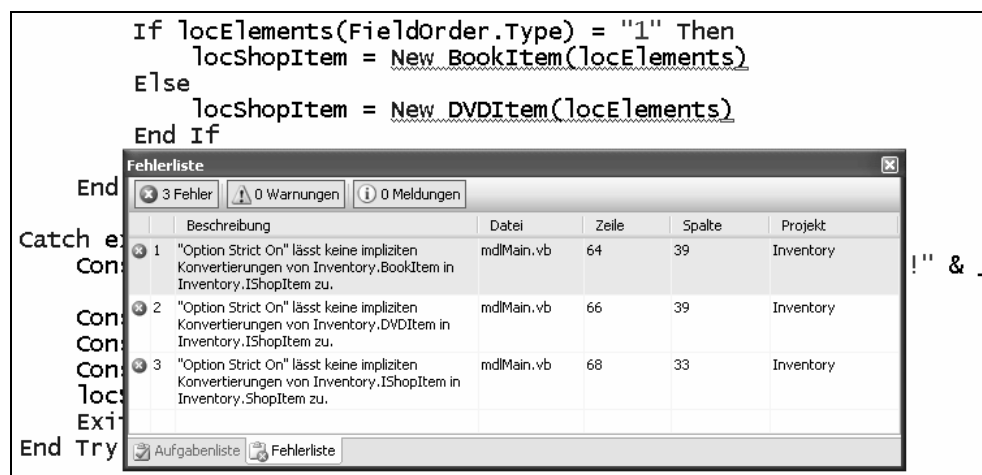
Das bislang verwendete Beispiel eignet sich ebenfalls, die Verwendung von Schnittstellen zu demonstrieren. Dazu wird eine Schnittstelle implementiert, die die Grundfunktionen vorschreibt, auf die das Hauptprogramm bislang über die Klasse *ShopItem* Zugriff hatte. Das Interface für das Beispielprogramm soll den Namen *IShopItem* bekommen (das Voranstellen des Buchstabens »I« ist eine gängige Konvention für die Benennung einer Schnittstellenklasse), und die Implementierung geschieht im Beispielprojekt noch vor dem Hauptmodul:

```
Interface IShopItem
    Property PrintTypeSetting() As PrintType
    ReadOnly Property ItemDescription() As String
    Function ToString() As String
    Property GrossPrice() As Double
End Interface
```

Als nächstes kümmern wir uns um das Hauptprogramm, das zur Steuerung jetzt nicht mehr die abstrakte Artikelbasisklasse, sondern die neue Schnittstelle verwenden soll: Dazu ist eine einzige Änderung notwendig, die Deklaration der abstrakten Artikelbasisklasse:

```
Sub Main()
    'Streamreader zum Einlesen der Textdateien
    Dim locSr As StreamReader
    Dim locList As New DynamicList ' Die Dynamische Liste
    Dim locElements() As String ' Die einzelnen ShopItem-Elemente
    Dim locShopItem As IShopItem ' Ein einzelnes Shop-Element
    Dim locDisplayMode As PrintType ' Der Darstellungsmodus
```

Da die Schnittstelle *IShopItem* alle Eigenschaften der ursprünglichen abstrakten Basisklasse *ShopItem* enthält, sind keine weiteren Änderungen erforderlich. Dennoch hat das Verändern der einen Zeile enorme Auswirkungen, und es hagelt gerade zu Fehler (siehe Abbildung 9.12):



**Abbildung 9.12:** Da zum jetzigen Zeitpunkt noch keine der abgeleiteten Klassen die Schnittstelle implementiert, ist ein Referenzieren der verwendeten Klassen noch nicht möglich

Das Hauptprogramm kann zwar nun alle Objekte verarbeiten, die die neue Schnittstelle implementiert haben. Da aber bislang keine der Klassen die Schnittstellen implementiert, ist das Programm zu diesem Zeitpunkt nicht lauffähig. Visual Basic beschwert sich zu diesem Zeitpunkt mit einem nicht so klar verständlichen Fehler: Da es keinen Zusammenhang zwischen `IocShopItem` (das als `IShopItem` deklariert ist) und den Klassen `BookItem` und `DVDItem` herstellen kann, macht es das Nächstliegende: Es versucht, die Typen implizit zu konvertieren. Da ein implizites Konvertieren von einem Typ zum anderen durch `Option Strict On` (als global gültig in den Projekteigenschaften eingestellt) unterbunden ist, liefert es eine entsprechende Fehlermeldung, die zunächst ein wenig in die Irre führen kann.

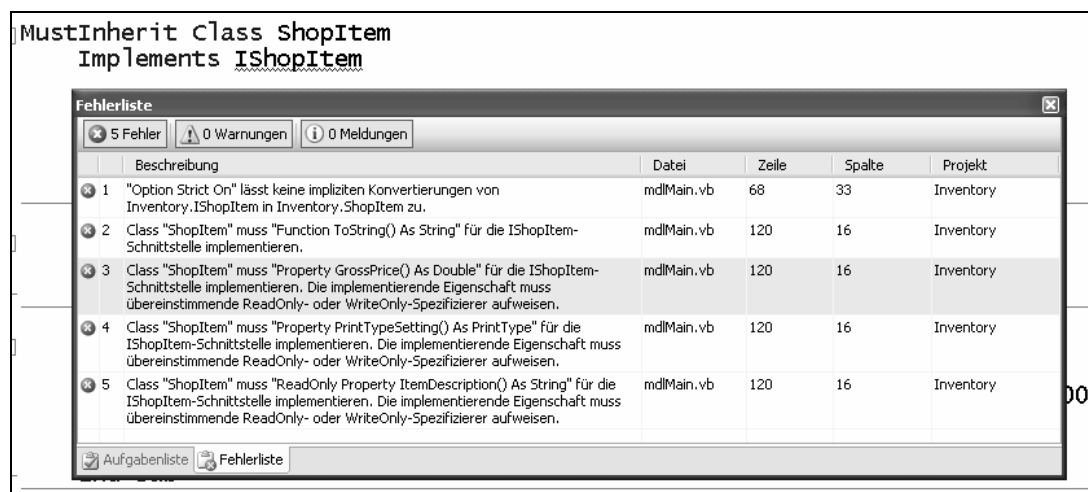
Nun leiten sich aber alle Klassen, die wir im Programm tatsächlich verwenden, von der bisherigen Klasse `ShopItem` ab. Die Implementierung der Schnittstelle `IShopItem` in der Klasse `ShopItem` bewirkt, dass anschließend auch alle von `ShopItem` abgeleiteten Klassen automatisch `IShopItem` implementieren. Es reicht also aus, wenn Sie alle geforderten Prozeduren in der abstrakten Basisklasse einbauen, damit das Programm anschließend wieder lauffähig ist.

Bei der Implementierung einer Schnittstelle in eine Klasse müssen Sie in Visual Basic auf zwei Sachen achten:

- Sie geben bei der Definierung der Klasse mit `Implements` an, welche Schnittstelle implementiert werden soll.
- Sie bestimmen für jede betroffene Prozedur der Klasse individuell, welche Schnittstellenprozedur sie implementieren soll. Auch das geschieht mit dem Schlüsselwort `Implements`.

Um die `Implements`-Anweisung hinzuzufügen, geben Sie bitte unterhalb der Zeile `MustInherit Class ShopItem` **Implements IShopItem** ein, drücken aber zunächst NICHT **Eingabe**, sondern verlassen die Zeile mit einer der Cursor-Tasten.

Sobald Sie das `Implements`-Schlüsselwort der Klassendefinition auf diese Weise hinzugefügt haben, verschwinden zwar die bisherigen Fehlermeldungen. Allerdings werden diese in der nächsten Sekunde durch andere ersetzt, wie in Abbildung 9.13 zu sehen:

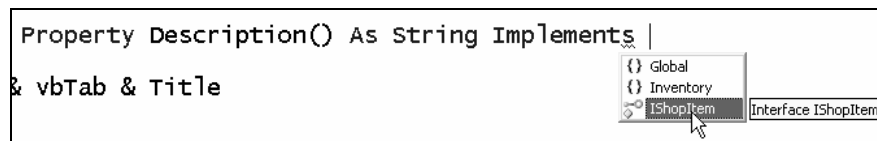


**Abbildung 9.13:** Wenn Sie eine Schnittstelle implementieren, müssen Sie auch für die Implementierung der einzelnen Prozeduren der Schnittstelle sorgen

Jetzt fragen Sie sich möglicherweise, wieso es nicht ausreicht, der Prozedur in der Klasse denselben Namen wie den der Schnittstellenprozeduren zu geben – beim Ableiten von Klassen reicht es schließlich aus.

In C# beispielsweise funktioniert das ohne Probleme, die Frage ist also berechtigt. Sie haben in C# auf der anderen Seite auch nicht wie in Visual Basic die Möglichkeit, einen ganz anderen Namen in Ihrer Klassenprozedur zu definieren, der eben nicht dem Namen der Interface-Prozedur entspricht.<sup>3</sup> Insofern bringt Ihnen dieses Feature von Visual Basic schon eine größere Flexibilität. Dazu kommt, dass Microsoft Intermediate Language, in die jede .NET-Anwendung zunächst kompiliert wird, ebenfalls so konzipiert ist, dass sie die explizite Angabe der zu implementierenden Schnittstellenprozedur erfordert.

Außerdem genießen Sie in Visual Basic dank IntelliSense einen Vorteil in Form einer Eingabehilfe: Sobald Sie am Ende einer Prozedur, der Sie ein Schnittstellenelement zuweisen wollen, das *Implements*-Schlüsselwort eingegeben haben, bietet Ihnen IntelliSense die möglichen Implementierungen zur Auswahl in einer Liste an (siehe Abbildung 9.14):



**Abbildung 9.14:** IntelliSense unterstützt Sie bei der Auswahl der richtigen Schnittstellenimplementierung

Um die Beispielanwendung wieder zum Laufen zu bewegen, müssen Sie die folgenden Prozeduren bearbeiten und ihnen die richtigen Schnittstellenprozeduren zuweisen:

```
Public MustOverride Property GrossPrice() As Double Implements IShopItem.GrossPrice
Public Property PrintTypeSetting() As PrintType Implements IShopItem.PrintTypeSetting
    Get
        Return myPrintTypeSetting
    End Get

    Set(ByVal Value As PrintType)
        myPrintTypeSetting = Value
    End Set
End Property

Public Overrides Function ToString() As String Implements IShopItem.ToString
    If PrintTypeSetting = PrintType.Brief Then
        'Kurzform: Es wird in jedem Fall
        'die Description-Eigenschaft des Objektes
        'verwendet.
    Return MyClass.Description & vbCrLf & vbLf & _
        Me.NetPriceFormatted & vbTab & Me.VATAmountFormatted & vbTab & _
        Me.GrossPriceFormatted & vbCrLf & vbLf
```

<sup>3</sup> Von einer Ausnahme abgesehen, doch dazu später in ► Kapitel 19 noch mehr.

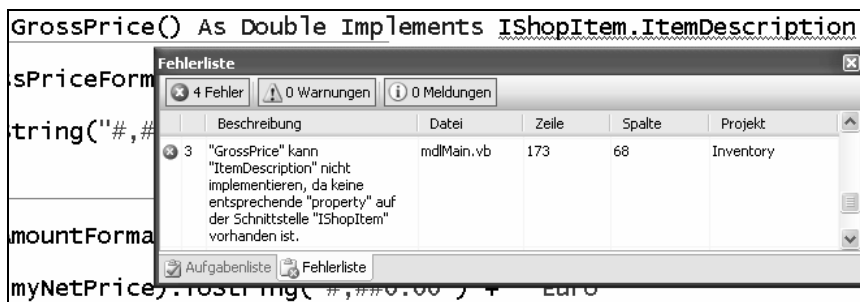
```

Else
    'Langform: Die Description Eigenschaft des Objektes
    'selber wird verwendet.
    Return Me.Description & vbCr & vbLf & _
        Me.NetPriceFormatted & vbTab & Me.VATAmountFormatted & vbTab & _
        Me.GrossPriceFormatted & vbCr & vbLf
End If
End Function

Public Overridable ReadOnly Property Description() As String Implements IShopItem.ItemDescription
    Get
        Return OrderNumber & vbTab & Title
    End Get
End Property

```

**HINWEIS:** Die letzte Eigenschaft verdient hier besonderes Interesse, denn sie bindet eine Schnittstellenprozedur ein, die einen anderen Namen hat als die Prozedur, die sie einbindet. Sie sehen also, dass Namen an dieser Stelle nur Schall und Rauch sind. Erst mit `Implements` wird festgelegt, welches Schnittstellenelement welchem Klassenelement zugeordnet werden soll. Dabei haben Sie natürlich nicht komplett freie Hand: Sie können – wie anzunehmen ist – nicht eine in einer Schnittstelle definierten Eigenschaft in einer Klassemethode sondern eben auch nur als Eigenschaft implementieren. Sie können auch keine Methode in einer Schnittstellendefinition, die als Parameter einen String entgegennimmt, einer Methode einer Klasse zuordnen, die einen Integer als Parameter erwartet. Und auch die Rückgabetypen müssen sich in der einbindenden Klasse und der Schnittstelle, die eingebunden wird, entsprechen. Fehluweisungen in dieser Form würden das Common Type System von .NET grob verletzen, und deswegen spielt auch schon der Background-Compiler nicht mit, wenn Sie einen solchen Versuch unternehmen, wie Abbildung 9.15 zeigt.



**Abbildung 9.15:** Signaturen, Elementtyp (Eigenschaft, Methode, Ereignis) und Rückgabetypen in einer Schnittstellendefinition müssen denen in der einbindenden Klasse entsprechen

Nachdem Sie diese Änderungen durchgeführt haben, werden Sie nur noch einige »Fehler« in der Klasse `DynamicList` finden. Diese Klasse akzeptierte bisher lediglich `ShopItem`-Objekte in den Methoden `Add` und `Item`. Wenn Sie die verwendeten `ShopItem`-Objekte durch solche vom Typ `IShopItem` ersetzen, ist das Programm wieder lauffähig (die geänderten Stellen sind wieder fett hervorgehoben).

## Class DynamicList

```
Protected myStepIncreaser As Integer = 4
Protected myCurrentArraySize As Integer
Protected myCurrentCounter As Integer
Protected myArray() As IShopItem

Sub New()
    myCurrentArraySize = myStepIncreaser
    ReDim myArray(myCurrentArraySize)
End Sub

Sub Add(ByVal Item As IShopItem)

    'Prüfen, ob aktuelle Arraygrenze erreicht wurde.
    If myCurrentCounter = myCurrentArraySize - 1 Then
        'Neues Array mit mehr Speicher anlegen,
        'und Elemente hinüberkopieren. Dazu:

        'Neues Array wird größer:
        myCurrentArraySize += myStepIncreaser

        'Temporäres Array erstellen.
        Dim locTempArray(myCurrentArraySize - 1) As IShopItem

        'Elemente kopieren.
        'Wichtig: Um das Kopieren müssen Sie sich,
        'anders als bei VB6, selber kümmern!
        For locCount As Integer = 0 To myCurrentCounter
            locTempArray(locCount) = myArray(locCount)
        Next

        'Temporäres Array dem Memberarray zuweisen.
        myArray = locTempArray
    End If

    'Element im Array speichern
    myArray(myCurrentCounter) = Item

    'Zeiger auf nächstes Element erhöhen.
    myCurrentCounter += 1

End Sub

'Liefert die Anzahl der vorhandenen Elemente zurück.
Public Overridable ReadOnly Property Count() As Integer
    Get
        Return myCurrentCounter
    End Get
End Property
```

```
'Erlaubt das Zuweisen.
Default Public Overridable Property Item(ByVal Index As Integer) As IShopItem
    Get
        Return myArray(Index)
    End Get
    Set(ByVal Value As IShopItem)
        myArray(Index) = Value
    End Set
End Property
End Class
```

---

**BEGLEITDATEIEN:** Das Programmbeispiel mit den Änderungen für Schnittstellen finden Sie übrigens im Verzeichnis `.\VB 2005 - Entwicklerbuch\D - OOP\Kap09\Inventory03`.

---

## Unterstützung bei abstrakten Klassen und Schnittstellen durch den Editor

Ihnen ist sicherlich aufgefallen, dass Sie die Änderung für die Schnittstellen-Implementierung durch `Implements` in der Klasse `ShopItem` nicht **Eingabe** bestätigen, sondern die Zeile mit einer der Cursor-Tasten verlassen.

Hätten Sie nämlich die Zeile mit Eingabe verlassen, dann wäre die Editor-Unterstützung für das Implementieren der Funktionsrümpfe angesprungen. In unserem Fall, in dem die Routinen aber schon komplett vorhanden waren, wäre das allerdings eher hinderlich gewesen, wie die beiden folgenden Beispiele zeigen.

Zu diesem Zweck erstellen Sie neues Visual Basic-Konsolenprojekt unter einem beliebigen Namen. Implementieren Sie anschließend eine Schnittstelle und eine Klasse nach folgender Vorlage, die Sie oberhalb der Zeile `Module Module1` in der Codedatei `Module1.vb` erfassen.

```
Interface ITest
    Property EineEigenschaft() As String
    Function EineMethode() As Integer
End Interface

Public Class BindetITestEin

End Class
```

Anschließend bewegen Sie die Einfügemarke unter die Zeile `Public Class BindetITestEin`, geben **Implements ITest** ein und drücken anschließend **Eingabe**.

```

Interface ITest
  Property EineEigenschaft() As String
  Function EineMethode() As Integer
End Interface

Public Class BindetITestEin
  Implements ITest

  Public Property EineEigenschaft() As String Implements ITest.EineEigenschaft
  Get
  End Get
  Set(ByVal value As String)
  End Set
End Property

  Public Function EineMethode() As Integer Implements ITest.EineMethode
  End Function
End Class

```

**Abbildung 9.16:** Nachdem Sie mit Implements eine Schnittstelle eingebunden haben, fügt der Editor die kompletten Coderümpfe der benötigten Elemente ein

Diese Vorgehensweise funktioniert auch ergänzend, und das heißt: Sie können eine Methode, eine Eigenschaft oder ein Ereignis (zu Ereignissen später mehr) in der Schnittstellendefinition hinzufügen, bekommen dann natürlich eine Fehlermeldung, weil dieses neue Element nicht in der implementierenden Klasse vorhanden ist. Bewegen Sie die Einfügemarke allerdings anschließend hinter den Schnittstellennamen der Implements-Anweisung unterhalb der Klassendefinition und drücken **Eingabe**, wird der Elementcoderumpf wieder an das Ende der Klasse angehängen. Probieren Sie es aus:

- Fügen Sie in der Schnittstelle ITest eine weitere Methode vom Typ String namens ZweiteMethode ein, sodass sich folgender Code ergibt:

```

Interface ITest
  Property EineEigenschaft() As String
  Function EineMethode() As Integer
  Function ZweiteMethode() As String
End Interface

```

Anschließend sehen Sie in der Fehlerliste eine Fehlermeldung mit dem Hinweis darauf, dass diese neue Methode nicht in der Klasse implementiert wurde, die die Schnittstelle einbindet.

- Um diesen Fehler zu beheben, reicht es aus, die Einfügemarke unter die Zeile Public Class BindetITestEin hinter Implements ITest zu bewegen und **Eingabe** zu betätigen.

In diesem Moment verschwindet der Fehler, da der Editor den Coderumpf für die Methode ZweiteMethode einfügt, sie mit der Schnittstelle per Implements verbindet und damit den Implementierungsvorschriften der Schnittstelle genüge tut.

### Die Tücken der automatischen Codeergänzung bei Schnittstellen oder abstrakten Klassen

Allerdings funktioniert die Unterstützung durch den Editor bisweilen nicht immer so reibungslos und kann zu – na ja, nennen wir es – »Orientierungsproblemen« führen, wenn es bereits Methoden oder Eigenschaften der Basisklasse gibt, die genau so heißen wie die zu implementierenden Methoden oder Eigenschaften, aber noch nicht mit diesen verknüpft sind.

Um das nachzustellen entfernen Sie bitte hinter der Funktion

```
Public Function ZweiteMethode() As String Implements ITest.ZweiteMethode
```

```
End Function
```

die Schnittstellenimplementierung `Implements ITest.ZweiteMethode`.

Daraufhin unterschlängelt der Editor die Zeile

```
Implements ITest
```

blau, da, wie er richtig erkennt, die Methode `ZweiteMethode` der Schnittstelle `ITest` nicht mehr korrekt implementiert ist. Doch die Editorunterstützung versagt anschließend ein wenig. Denn wenn sie nun abermals die Einfügemarke unter die Zeile `Public Class BindetITestEin` hinter `Implements ITest` bewegen und **Eingabe** betätigen, erscheint am Ende der Klasse ein Funktionsrumpf, wie Sie ihn auch in [Abbildung 9.17](#) erkennen können.

```
Public Class BindetITestEin
    Implements ITest

    Public Property EineEigenschaft() As String Implements ITest.EineEigenschaft
        Get
            End Get
        Set(Byval value As String)
            End Set
    End Property

    Public Function EineMethode() As Integer Implements ITest.EineMethode
        End Function

    Public Function ZweiteMethode() As String
        End Function

    Public Function ZweiteMethode1() As String Implements ITest.ZweiteMethode
        End Function
End Class
```

**Abbildung 9.17:** Beim automatischen Kompletieren von Coderümpfen von Elementen, die mit gleichem Namen schon vorhanden sind, wird ein »ähnlicher« Elementname erfunden

Die Methode `ZweiteMethode` gibt es nun (mehr oder weniger) zweimal – einmal mit nachgestellter »1«. Anstelle also hinter `ZweiteMethode` korrekterweise die `Implements`-Anweisung mit dem entsprechenden Schnittstellenelementnamen zu ergänzen, kreiert der Editor eine ganz neue Methode, und implementiert das Schnittstellenelement in dieser.

Falls Ihnen solche Dinge beim Entwickeln Ihrer eigenen Schnittstellenkreationen passieren, müssten Sie die `Implements`-Anweisung zur eigentlich gemeinten (und schon vorhandenen) Methode übertragen, und die vom Editor »erfundene« Methode löschen.

Noch diffuser wird es, wenn es um die Implementierung einer Methode oder Eigenschaft geht, die nur aus der Basisklasse hervorgeht.

`ToString` ist standardmäßig solch ein Kandidat. `ToString` ist in jeder neu erstellten Klasse enthalten, da er eine Methode von `Object` darstellt, und jede neue Klasse wird, wie wir ja schon wissen, implizit von `Object` abgeleitet, wenn nichts anderes gesagt wird. Damit erbt natürlich auch jede neue Klasse die `ToString`-Methode (warum die `ToString`-Methode an `Object` »hängt«, klärt übrigens der ► Abschnitt »Die Methoden und Eigenschaften von `Object`« ab Seite 311).

Nun achten Sie darauf was passiert, wenn wir der Schnittstelle exakt diese Methode hinzufügen. Erwartungsgemäß sehen wir zunächst einen Fehler im Programm, da die neue Schnittstellendefinition erwartet, dass wir eine `ToString`-Funktion implementieren. Das machen wir durch das standardmäßige Vorhandensein von `ToString` ja auch ohne weiteres Zutun; was allerdings fehlt, ist, unsere (geerbte) `ToString`-Funktion mit der Schnittstellenmethodendefinition `ToString per Implements` (á la Abbildung 9.14) zu verheiraten.

Mit `Implements` würde das auch nicht so ohne weiteres funktionieren, denn es gibt ja schließlich überhaupt keinen Methodencode, hinter den wir `Implements` für die Zuweisung des Schnittstellen-`ToString` hängen könnten.

Ergünden wir es also zunächst, wie uns der Editor »hilft«, und wie wir anschließend wirklich zum Ziel kommen:

- Fügen Sie in der Schnittstelle `ITest` eine weitere Methode vom Typ `String` namens `ToString` ein, sodass sich folgender Code ergibt:

```
Interface ITest
    Property EineEigenschaft() As String
    Function EineMethode() As Integer
    Function ZweiteMethode() As String
    Function ToString() As String
End Interface
```

Anschließend sehen Sie wieder in der Fehlerliste eine Fehlermeldung mit dem Hinweis darauf, dass diese neue Methode nicht in der Klasse implementiert wurde, die die Schnittstelle einbindet.

- Versuchen wir den Fehler auf die gleiche Weise mit der Editorunterstützung zu beheben: Platzieren Sie die Einfügemarke wieder unterhalb der Zeile `Public Class BindetITestEin` hinter `Implements ITest`, und betätigen Sie **Eingabe**.

Das Ergebnis ist wieder dasselbe wie im vorherigen Beispiel, nur leider nicht so einfach zu durchschauen, da es in der Klasse ja keinen Code für `ToString` gibt (denn dieser wurde ja aus der Basisklasse `Object` übernommen).

```

Public Class BindetITestEin
    Implements ITest
Public Property EineEigenschaft() As String Implements ITest.EineEigenschaft
    Get
        End Get
    Set(ByVal value As String)
        End Set
    End Property
Public Function EineMethode() As Integer Implements ITest.EineMethode
    End Function
Public Function ZweiteMethode() As String Implements ITest.ZweiteMethode
    End Function
Public Function ToString1() As String Implements ITest.ToString
    End Function
End Class

```

**Abbildung 9.18:** Verwirrender ist das Vorgehen des Editors bei der Codeergänzung, wenn es eine geerbte Methode oder Eigenschaft mit gleichem Namen wie in der Schnittstellendefinition gibt

Prinzipiell hat der Editor das gleiche Problem. Es gibt bereits eine Methode (ToString), nur ist sie dieses Mal nicht sichtbar, denn sie wurde aus der Basisklasse (Object in diesem Beispiel) übernommen. Also kriert der Editor wieder eine Abwandlung des Funktionsnamens, und nennt diesen genau so wie den eigentlichen nur mit einer zusätzlich hinten angestellten »1«.

Diesen Fehler können Sie nur folgendermaßen beheben:

- Sie überschreiben die Methode bzw. Eigenschaft, die es zu implementieren gilt.
- Sie rufen innerhalb der Methode oder Eigenschaft nichts weiter auf als die Basismethode (oder Eigenschaft).
- Sie implementieren die Schnittstelle an der überschriebenen Methode bzw. Eigenschaft. Der Code dafür würde in unserem Fall folgendermaßen aussehen:

```

Public Overrides Function ToString() As String Implements ITest.ToString
    Return MyBase.ToString
End Function

```

### Editorunterstützung bei abstrakten Klassen

Übrigens: Die Unterstützung, die Sie bei der Implementierung von Schnittstellen erfahren, gibt's auch bei abstrakten Klassen. Ergänzen Sie zur Demonstration das Beispiel um folgende abstrakte Klasse:

```

MustInherit Class AbstractTest
    Public MustOverride Property EineEigenschaft() As String
    Public MustOverride Function EineMethode(ByVal EinParameter As String) As String
End Class

```

Erstellen Sie nun eine Klasse, die auf AbstractTest basiert:

```
Public Class BasiertAufAbstractTest
    Inherits AbstractTest
```

```
End Class
```

In dem Moment, in dem Sie hinter Inherits AbstractTest **Eingabe** betätigen, komplettiert der Editor für Sie automatisch alle Methoden und Eigenschaften in Form von Coderümpfen, die mit MustOverride als virtuelle Methoden bzw. Eigenschaften gekennzeichnet wurden:

```
Public Class BindetITestEin
    Implements ITest

    Public Property EineEigenschaft() As String Implements ITest.EineEigenschaft
        Get

            End Get
            Set(ByVal value As String)

                End Set
        End Property

    Public Function EineMethode() As Integer Implements ITest.EineMethode

        End Function

    Public Function ZweiteMethode() As String Implements ITest.ZweiteMethode

        End Function

    Public Overrides Function ToString() As String Implements ITest.ToString
        Return MyBase.ToString
    End Function

End Class
```

## Schnittstellen, die Schnittstellen implementieren

Auch Schnittstellen können Schnittstellen implementieren. Allerdings tun sie das nicht auf die gleiche Weise, wie Klasse Schnittstellen implementieren. Stattdessen können Schnittstellen von anderen Schnittstellen erben – genau wie Klassen von anderen Klassen erben können und dabei deren komplette Funktionalität übernehmen. Das es aber bei Schnittstellen keine Funktionalitäten, sondern nur virtuelle Methoden, Eigenschaften oder Ereignisse gibt, erben Schnittstellen von anderen Schnittstellen nur deren Implementierungsvorschriften. Oder anders und einfacher ausgedrückt: Eine Schnittstelle, die von einer anderen Schnittstelle erbt, enthält sämtliche ihrer Vorschriften und die der Schnittstelle, von der sie erbt. Und darüber hinaus können Schnittstellen auch nur von Schnittstellen erben, nicht von Klassen (was aber wahrscheinlich auch ohne explizite Erwähnung klar ist, denn Schnittstellen dürfen ja anders als Klassen kein Code enthalten).

Sie können sich auch diese Zusammenhänge an einem Beispiel verdeutlichen. Ergänzen Sie das Modul aus dem vorherigen Beispiel um folgenden Code:

```
Interface IBaseInterface
    Property EineEigenschaft() As String
End Interface
```

```
Interface IMoreComplexInterface
    Inherits IBaseInterface

    Property ZweiteEigenschaft() As String
End Interface
```

Die Schnittstelle `IMoreComplexInterface` beinhaltet in diesem Fall die Vorschriften beider Schnittstellendefinitionen (der eigenen und der, von der sie erbt).

Wenn Sie diese »vereinigte« Schnittstelle in einer Klasse implementieren, ergibt sich – dank Codeergänzung durch den Editor ist das ja schnell getan – nach dem Betätigen von **Eingabe** hinter `Implements IMoreComplexInterface` im folgenden Code folgendes Ergebnis:

```
Public Class ComplexClass
    Implements IMoreComplexInterface

    Public Property EineEigenschaft() As String Implements IBaseInterface.EineEigenschaft
        Get

            End Get
        Set(ByVal value As String)

            End Set
        End Property

    Public Property ZweiteEigenschaft() As String Implements IMoreComplexInterface.ZweiteEigenschaft
        Get

            End Get
        Set(ByVal value As String)

            End Set
        End Property
End Class
```

## Einbinden mehrere Schnittstellen in eine Klasse

Mehrfachvererbung ist in Visual Basic 2005 nicht vorgesehen. Bei der Mehrfachvererbung entsteht eine neue Klasse aus mehr als einer Basisklasse und dabei übernimmt die erbende Klasse die Funktionalität von beiden Basisklassen.

Allerdings können Sie auch in Visual Basic 2005 mehr als eine Schnittstelle in einer Klasse implementieren, und auch wenn das nicht so »toll«<sup>4</sup> wie Mehrfachvererbung ist, so können Sie sich mit der

---

<sup>4</sup> Mehrfachvererbung ist nur auf den ersten Blick etwas Feines, und es wird sie in Visual Basic sehr wahrscheinlich nicht geben, da selbst einer der Erfinder von C++, der Herr Stroustrup, das inzwischen sehr skeptisch sieht. (In Java ist es ja auch *absichtlich* weggelassen worden.) Mehrfachvererbung sorgt sogar für unentscheidbare Konflikte, die der Compiler willkürlich auflösen muss. (Zwei Methoden gleichen Namens in beiden Klassen mit Implementierung, usw.) Stroustrup

Einbindung mehrerer Schnittstellen wenigstens ein wenig behelfen. Man könnte das Einbinden mehrerer Schnittstellen in einer Klasse sozusagen als »Mehrfachvererbung Light« bezeichnen.

Das Einbinden mehrerer Schnittstellen ist ein Kinderspiel. Das vorherige Beispiel der Schnittstellendefinitionen ändern wir dazu folgendermaßen ab:

```
Interface IBaseInterface
    Property EineEigenschaft() As String
End Interface

Interface IMoreComplexInterface
    'Inherits wurde entfernt, beide Schnittstellen liegen somit auf "gleicher Ebene".
    Property ZweiteEigenschaft() As String
End Interface

Public Class ComplexClass
    Implements IBaseInterface, IMoreComplexInterface

    Public Property EineEigenschaft() As String Implements IBaseInterface.EineEigenschaft
        Get

            End Get
        Set(ByVal value As String)

            End Set
        End Property

    Public Property ZweiteEigenschaft() As String Implements IMoreComplexInterface.ZweiteEigenschaft
        Get

            End Get
        Set(ByVal value As String)

            End Set
        End Property
    End Class
```

Am Code der einbindenden Klasse muss in diesem Fall nur die in Fettschrift gesetzte Zeile geändert werden. An dem restlichen Code, der die Schnittstellenelemente den Klassenelementen zuordnet, müssen keinerlei Änderungen vorgenommen werden.

## Die Methoden und Eigenschaften von Object

Object selber stellt einige Grundmethoden und -eigenschaften zur Verfügung, die damit jede neue Klasse automatisch erbt. Der Hintergrund ist, dass die Entwickler des .NET-Frameworks damit sichergestellt haben, dass jedes Objekt über eine gewisse Grundfunktionalität verfügt, auf deren Vorhandensein sich andere Klassen 100 prozentig verlassen können.

---

dazu: »[...] Mehrfachvererbung bleibt verständlich, wenn nur eine Basisklasse wirklich Member-Funktionen implementiert und die anderen nur pure virtuelle Funktionen deklariert. [...]«

## Polymorphie am Beispiel von ToString und der ListBox

Ein Beispiel dazu entführt uns für einen Moment in die Windows Forms-Entwicklung – genauer gesagt zu einer simplen Anwendung, die eine ListBox verwendet.

---

**BEGLEITDATEIEN:** Das Grundgerüst für die folgenden Experimente finden Sie unter `.\VB 2005 - Entwicklerbuch\D - OOP\Kap09\ToStringDemo`.

---

Dieses Projekt besteht aus zwei »Elementen«, dem Formular mit der Steuerung der einzigen Schaltfläche sowie einer Klasse, die die Aufgabe hat, Kontaktdaten in Form von Namen und Vornamen zu speichern:

```
'Der Formularcode
Public Class frmMain

    Private Sub btnKontaktHinzufügen_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
        Handles btnKontaktHinzufügen.Click

        'Neues Kontaktobjekt instanzieren und es mit den Inhalten
        'der Textbox füllen
        Dim locKontakt As New Kontakt(txtVorname.Text, txtNachname.Text)

        'Der Listbox hinzufügen
        lstKontakte.Items.Add(locKontakt)
    End Sub
End Class

'Die Kontakt-Klasse
Public Class Kontakt

    Private myVorname As String
    Private myNachname As String

    Sub New(ByVal Vorname As String, ByVal Nachname As String)
        myVorname = Vorname
        myNachname = Nachname
    End Sub

    Public Property Vorname() As String
        Get
            Return myVorname
        End Get
        Set(ByVal value As String)
            myVorname = value
        End Set
    End Property

    Public Property Nachname() As String
        Get
            Return myNachname
        End Get
    End Property
End Class
```

```

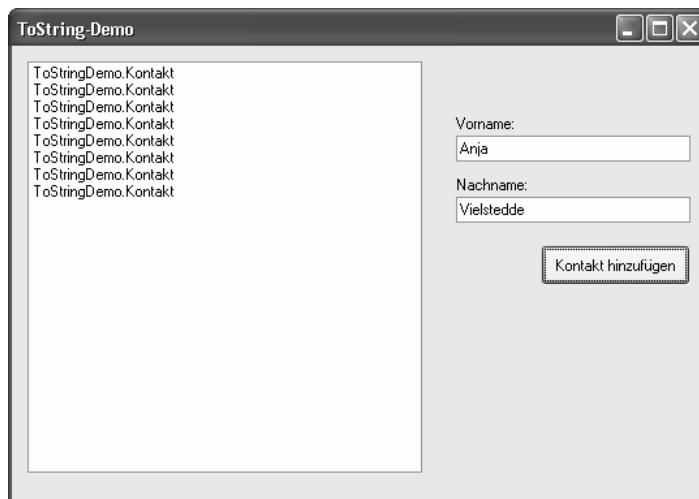
        Set(ByVal value As String)
            myNachname = value
        End Set
    End Property
End Class

```

Interessant an dieser Stelle ist die fett geschriebene Codezeile im Listing. Sobald der Benutzer Vor- und Nachnamen in die TextBox-Steuer-elemente eingegeben und die Schaltfläche betätigt hat, legt das Programm eine neue Instanz der Kontakt-Klasse an und fügt diesen Kontakt dann mithilfe der Add-Methode der Items-Auflistung der ListBox den ListBox-Elementen hinzu. Diese Vorgehensweise ist möglich, da anders, als noch in Visual Basic 6.0, die Add-Methode der ListBox nicht nur String-Elemente sondern beliebige Objekte aufnimmt.

Das Problem: Wenn die ListBox in ihrer Items-Auflistung Objekte vom Typ Object aufnimmt, kann man ihr sämtliche Typen übergeben (so auch unser Kontakt-Objekt). Das ist möglich, da, wie wir ja schon wissen, eine Objektvariable nicht auf den »eigenen Typ« sondern auch auf jeden abgeleiteten Typ verweisen kann (siehe ► Abschnitt »Polymorphie« und folgende ab Seite 279). Und da alle Klassen implizit von Object abgeleitet werden, basiert letzten Endes jede neu erfundene Klasse auch auf Object. Wann immer Ihnen also ein Parameter vom Typ Object begegnet, können Sie jeden beliebigen Typ übergeben. So weit, so gut.

Doch wer oder was sorgt nun dafür, dass ein Sinn ergebender Text in der ListBox angezeigt wird? In der jetzigen Ausbaustufe des Programms zeigt die ListBox jedenfalls noch nicht die gewünschten Ergebnisse an, was Sie schnell herausfinden können, wenn Sie das Programm laufen lassen:



**Abbildung 9.19:** Das Hinzufügen eines Objektes zur *ListBox* führt zunächst nur zur Anzeige des voll qualifizierten Objekt-namens

Das Ergebnis entspricht sicherlich nicht dem erwarteten. Aber das kann es auch gar nicht, und Sie werden sehen, warum das so ist, wenn Sie erfahren, wie die *ListBox* arbeitet, um den Text eines Objektes zu ermitteln.

Vielleicht ahnen Sie es schon: Wenn die `ListBox` ihren Anzeigebereich mit Texten füllt, ruft sie die `ToString`-Funktion eines jeden Elementes auf,<sup>5</sup> das ihrer `Items`-Auflistung hinzugefügt wurde. Das kann sie machen, ohne befürchten zu müssen, dass ein Objekt, das sie beinhaltet, kein `ToString` hat, denn `ToString` ist Bestandteil von `Object`, `Object` wiederum die Basisklassen aller Basisklassen und damit verfügt jedes andere Objekt auch über eine `ToString`-Funktion.

Die Standardimplementierung von `ToString` in `Object` sorgt allerdings lediglich dafür, dass der Typname des Objektes ausgegeben wird. Die Standardimplementierung der `ToString`-Funktion von `Object` schaut nämlich folgendermaßen aus:

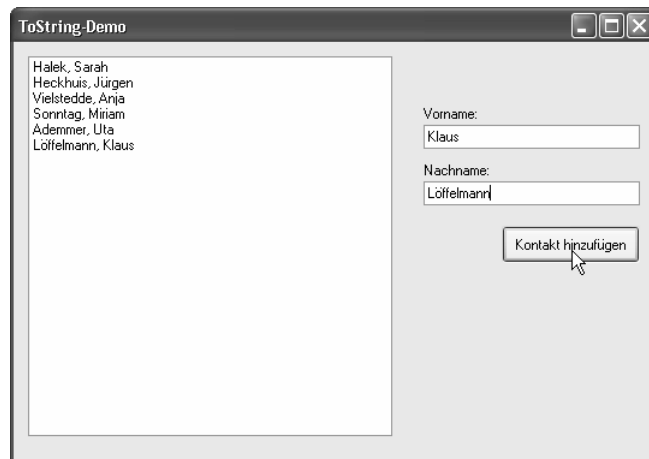
```
Public Overridable Function ToString() As String
    Return Me.GetType.ToString
End Function
```

`GetType` liefert den Typ eines Objektes zurück, der wiederum durch `ToString` in eine lesbare Zeichenfolge übersetzt wird. Und auf diese Weise zeigt `ToString` für jedes Objekt dessen Typnamen an – der Polymorphie sei Dank.

Damit `ToString` einer Klasse beispielsweise bei der Darstellung in einer `ListBox` einen Sinn ergebenden Text ausgibt, müssen Sie die `ToString`-Funktion in der entsprechenden Klasse überschreibend implementieren. Würden Sie also die Klasse `Kontakt` um folgenden Code ergänzen

```
Public Overrides Function ToString() As String
    Return Nachname & ", " & Vorname
End Function
```

wäre das Ergebnis wie erwartet, wie auch die folgende Abbildung zeigt:



**Abbildung 9.20:** Durch das Überschreiben von `ToString()` in der Klasse, deren Objektinstanz der Liste hinzugefügt wird, lässt sich der dargestellte Listentext steuern

<sup>5</sup> Um genau zu sein: Sie ruft `GetItemText` ihrer Basisklasse `ListControl` auf, und *die* ermittelt den Text durch `ToString`, falls es sich beim Objekt nicht um ein Steuerelement handelt, das an eine andere Eigenschaft gebunden werden kann. In diesem Fall würde `GetItemText` den Text des Objektes per `ToString` ermitteln, der an das Steuerelement gebunden ist (oder einen Leer-String zurückliefern, falls es keine Bindung gibt). Aus diesem Grund sehen Sie auch einen Leer-String in der Liste, falls Sie der `Items`-Auflistung die Instanz irgendeines Steuerelements übergaben (wie beispielsweise eine Schaltfläche oder das Formular selbst).

## Prüfen auf Gleichheit von Objekten mit Object.Equals oder dem Is/IsNot-Operator

Die deutsche Sprache wird immer mehr vereinfacht. Man sieht das an Wörtern wie beispielsweise »das Gleiche« und »dasselbe«. Inzwischen sind diese laut Duden dasselbe (oder das Gleiche?) – aber das war nicht immer so. Dasselbe bezeichnete einst buchstäblich ein und dasselbe. Zwei verschiedene Leute konnten nicht gleichzeitig mit demselben Auto fahren – es sei denn, sie hätten es in der Mitte durchgeschnitten (und irgendwie fahrbereit gemacht). Wohl aber mit dem gleichen: Dann wären es unterschiedliche Autos gewesen, die sich einfach nur sehr ähnlich waren.

Diesen Unterschied sollten Sie für das bessere Verständnis der folgenden Erklärung kennen.

Es gibt für jedes Objekt eine Methode namens Equals, die überprüft, ob es sich bei einer Objektinstanz, die durch eine Objektvariable referenziert wird, um dieselbe handelt, die durch eine andere Objektvariable referenziert wird. Es wird dabei NICHT überprüft, ob der Inhalt der Objekte der gleiche ist. Ein Beispiel soll das verdeutlichen, und zum besseren Verständnis sollten Sie zuvor nochmals einen Blick auf Abbildung 9.9 werfen.

---

**HINWEIS:** Statt die Equals-Methode zu verwenden, der Sie ein anderes Objekt zum Vergleichen übergeben können, haben Sie auch die Möglichkeit den Is-Operator (bzw. den IsNot-Operator) einzusetzen. Das folgende Beispiel demonstriert den Einsatz beider Möglichkeiten.

---

**BEGLEITDATEIEN:** Das folgende Beispiel finden Sie unter `.\VB 2005 - Entwicklerbuch\D - OOP\Kap09\EqualsDemo`.

---

Module mdlMain

Sub Main()

```
'Instanzieren mit New und dadurch  
'Speicher für das Kontakt-Objekt  
'auf dem Managed Heap anlegen  
Dim objVarKontakt As New Kontakt
```

```
'Daten zuordnen  
With objVarKontakt  
    .Nachname = "Halek"  
    .Vorname = "Sarah"  
    .Plz = "99999"  
    .Ort = "Musterhausen"  
End With
```

```
'objVarKontakt2 zeigt auf dasselbe Objekt;  
'die referenzierte Instanz ist dieselbe!  
Dim objVarKontakt2 As Kontakt  
objVarKontakt2 = objVarKontakt
```

```
'objVarKontakt3 zeigt auf ein gleiches Objekt;  
'die referenzierte Instanz ist nicht dieselbe, nur die gleiche!  
Dim objVarKontakt3 As New Kontakt  
'Daten zuordnen  
With objVarKontakt
```

```

        .Nachname = "Halek"
        .Vorname = "Sarah"
        .Plz = "99999"
        .Ort = "Musterhausen"
    End With

    'Der Beweis dafür:
    Console.WriteLine("Die Aussage ""objVarKontakt entspricht objVarKontakt2 ist "" " & _
        objVarKontakt.Equals(objVarKontakt2))

    Console.WriteLine("Die Aussage ""objVarKontakt entspricht objVarKontakt3 ist "" " & _
        objVarKontakt.Equals(objVarKontakt3))

    Console.WriteLine()

    'Alternativ durch das IS-Schlüsselwort:
    Console.WriteLine("Die Aussage ""objVarKontakt entspricht objVarKontakt2 ist "" " & _
        (objVarKontakt Is objVarKontakt2))

    Console.WriteLine("Die Aussage ""objVarKontakt entspricht objVarKontakt3 ist "" " & _
        (objVarKontakt Is objVarKontakt3))

    Console.WriteLine()
    Console.WriteLine("Zum Beenden Taste drücken")
    Console.ReadKey()
End Sub

```

End Module

Class Kontakt

```

    Public Nachname As String
    Public Vorname As String
    Public Plz As String
    Public Ort As String

```

End Class

Diese kleine Demo ist weitestgehend selbsterklärend. Sie definiert drei Objektvariablen auf Basis der Klasse Kontakt. Sie instanziiert daraus zwei Objekte und füttert sie mit den gleichen Daten. Die dritte Objektvariable verweist allerdings auf dieselbe Instanz wie die erste Objektvariable. Deswegen ist ein Vergleich mit Equals oder Is wahr.

---

**TIPP:** Die Regel lautet: Verweisen zwei Objektvariablen auf die gleiche Instanz, und speichern sie deswegen die gleichen Zeiger auf die eigentlichen Daten im Managed Heap, ergeben Is oder Equals als Ergebnis True, ansonsten False. Die Daten selbst (die Instanzen der Objekte) werden bei diesem Vergleich nicht verwendet.

---

Das Ausgabeergebnis dieses Programm sieht demzufolge folgendermaßen aus:  
 Die Aussage "objVarKontakt entspricht objVarKontakt2 ist " True  
 Die Aussage "objVarKontakt entspricht objVarKontakt3 ist " False

Die Aussage "objVarKontakt entspricht objVarKontakt2 ist " True  
Die Aussage "objVarKontakt entspricht objVarKontakt3 ist " False

Zum Beenden Taste drücken

## Equals, Is und IsNot im praktischen Entwicklungseinsatz

Die Frage, die sich stellt, lautet wie bei allem, was man neu lernt: Wozu brauche ich das? Was das Testen auf identische Objekte anbelangt, ist die Antwort einfach: an fast jeder Stelle.

Das geht spätestens dann los, wenn Sie ein bestimmtes Objekt in einer Auflistung wie beispielsweise der Items-Auflistung der ListBox suchen. Sie möchten beispielsweise einen bestimmten Eintrag aus der Liste einer ListBox löschen. Zu diesem Zweck würden Sie die Remove-Methode verwenden, die an der Items-Eigenschaft der ListBox »hängt«. Die Remove-Methode nimmt jedes beliebige Objekt entgegen und ist natürlich nicht in der Lage, die Inhalte der Objektinstanzen zu vergleichen, die durch ein ListBoxItem repräsentiert werden und dem Objekt entsprechen, das Sie entfernen möchten. Sie ist aber in der Lage herauszufinden, ob zwei Objekte auf die gleiche Instanz verweisen, und daran erkennt Remove, welches Objekt es zu entfernen hat.

Die folgende Demo zeigt dies im praktischen Einsatz. Mit dem schon bekannten Beispiel können Sie eine ListBox mit Kontaktdaten füllen. Doch ferner verfügen Sie über eine weitere Schaltfläche, mit der Sie einen selektierten Eintrag wieder aus der Liste entfernen können.

Den Verweis auf ein selektiertes Objekt einer ListBox können Sie mit SelectedItem der Items-Auflistung in Erfahrung bringen. Liefert SelectedItem den Wert Nothing zurück, handelt es sich um einen so genannten Null-Verweis; eine Objektvariable zeigt in diesem Fall auf keine Instanz mit Daten.

Liefert SelectedItem jedoch einen »Wert« zurück, der eben nicht Nothing ist, dann war ein Objekt der Liste selektiert. Dieses Objekt können Sie dann in einer Objektvariablen speichern und es der Remove-Methode der ListBoxItems-Auflistung übergeben:

---

**BEGLEITDATEIEN:** Das folgende Beispiel finden Sie unter `.\VB 2005 - Entwicklerbuch\D - OOP\Kap09\EqualsPraxis`.

---

Der folgende Auszug zeigt die Ereignisbehandlungsroutine der Schaltfläche, die ein Element aus der Liste löscht, sofern es selektiert war:

```
Private Sub btnKontaktLöschen_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _  
    Handles btnKontaktLöschen.Click  
    Dim locKontakt As Object  
    locKontakt = lstKontakte.SelectedItem  
    If locKontakt IsNot Nothing Then  
        lstKontakte.Items.Remove(locKontakt)  
    End If  
End Sub
```

Intern macht die Remove-Methode genau das gerade Gesagte: Sie iteriert (durchläuft) durch die Elemente und stellt fest, ob das zu löschende Element dem gerade bearbeiteten Element der Liste entspricht. Dabei führt sie die Prüfung auf Entsprechung ebenfalls mit Is bzw. der Equals-Methode durch.

# Übersicht über die Eigenschaften und Methoden von Object

Die folgende Tabelle fasst die Methoden der Klasse Object zusammen:

Member	Beschreibung
Equals	Stellt fest, ob das aktuelle Objekt, dessen <i>Equals</i> -Member verwendet wird, mit dem angegebenen <i>Object</i> identisch ist. Zwei Objekte sind dann identisch, wenn ihre Instanzen (das heißt der Datenbereich, auf den sie zeigen) übereinstimmen. <i>ValueType</i> überschreibt diese Methode, und vergleicht die einzelnen Member. Da Strukturen automatisch von <i>ValueType</i> ableiten, gilt dieses Verhalten für alle Strukturen.
GetHashCode	Produziert einen Hashcode (eine Art Identifizierungsschlüssel) auf Grund des Objekthaltens. Dieser Hashcode wird beispielsweise dann verwendet, wenn ein Objekt in einer Tabelle (einem Array) gesucht werden muss. Daher sollte <i>GetHashCode</i> nach Möglichkeit eindeutige Werte liefern, aber gleichzeitig auch vom Inhalt abhängige Werte als Grundlage der Hashcode-Berechnung mit einbeziehen. Der in <i>Object</i> implementierte Algorithmus garantiert weder Eindeutigkeit noch Konsistenz – Sie sind also angehalten, nach Möglichkeit eigene Hashcode-Algorithmen für abgeleitete Objekte zu entwickeln und diese Methode zu überschreiben, wenn Sie Ihr Objekt des Öfteren in Hash-Tabellen speichern wollen.
GetType	Ruft den aktuellen Typ der Instanz als <i>Type</i> -Objekt ab. Mehr dazu erfahren Sie im ► Kapitel 23.
(Shared) ReferenceEquals	Diese statische Methode entspricht der <i>Equals</i> -Methode, übernimmt die beiden zu vergleichenden Objekte aber als Parameter. Da diese Methode statischer Natur ist, können Sie sie nur über den Typnamen aufrufen ( <i>Object.ReferenceEquals</i> ).
ToString()	Liefert eine Zeichenkette zurück, die das aktuelle Objekt beschreibt. In der ursprünglichen Version ist das wörtlich zu nehmen; <i>ToString</i> liefert nämlich den Klassennamen zurück, wenn Sie diese Funktion nicht überschreiben. <i>ToString</i> sollte nach Möglichkeit den Inhalt des Objektes – wenigstens teilweise – als Zeichenkette zurückliefern.
Finalize	Wenn die <i>Finalize</i> -Methode eines Objektes aufgerufen wird, ist die Quelle des Aufrufs der Garbage Collector. Er signalisiert dem Objekt durch diesen Aufruf, dass es im Rahmen der »Müllabfuhr« im Begriff ist, entsorgt zu werden, und das Objekt hat in diesem Rahmen die Möglichkeit, Ressourcen freizugeben, die es nicht mehr benötigt. Mehr dazu finden Sie in ► Kapitel 12.
MemberwiseClone	Erstellt eine so genannte »flache Kopie« von dem Objekt, das die Methode beherbergt. Wenn Sie <i>MemberwiseClone</i> aufrufen, dann legt diese Methode eine Kopie aller Wertetyp-Member (dazu später mehr) an und stellt diese in einer weiteren Objektinstanz zur Verfügung. Für Referenztypen werden nur Adresskopien erstellt – sie zeigen anschließend also auf dieselben Objekte im Managed Heap, auf die auch die Referenztypen des Originals zeigen.

**Tabelle 3.2:** Die Beschreibung der Object-Member

## Shadowing (Überschatten) von Klassenprozeduren

Visual Basic kennt eine weitere Version des Ersetzens von Prozeduren in einer Basisklasse durch andere gleichen Namens einer abgeleiteten Klasse. Diesen Vorgang nennt man das so genannte »Shadowing« oder »Überschatten« von Elementen.

Wenn Sie in einer Basisklasse eine Funktion definiert haben, dann kann eine Funktion gleichen Namens in einer abgeleiteten Klasse das Original schlichtweg ausblenden, wie im folgenden Beispiel.

```
Module mdlMain

    Sub Main()
        Dim locBasisinstanz As New Basisklasse
        Console.WriteLine(locBasisinstanz.EineFunktion().ToString())

        locBasisinstanz = New AbgeleiteteKlasse
        Console.WriteLine(locBasisinstanz.EineFunktion().ToString())

        Console.WriteLine()
        Console.WriteLine("Taste drücken zum Beenden!")
        Console.ReadKey()
    End Sub

End Module
```

```
Public Class Basisklasse

    Protected test As Integer

    Sub New()
        test = 10
    End Sub

    Public Function EineFunktion() As Integer
        Return test * 2
    End Function

End Class
```

```
Public Class AbgeleiteteKlasse
    Inherits Basisklasse

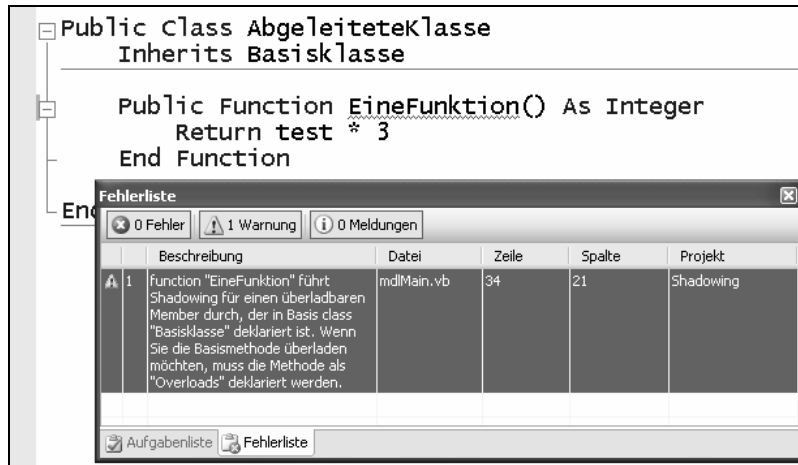
    Public Shadows Function EineFunktion() As Integer
        Return test * 3
    End Function

End Class
```

Ohne zunächst ganz genau auf die beiden Klassenimplementierungen zu achten, würden Sie wahrscheinlich annehmen, dass das Modul zunächst den Wert 20 und in der folgenden Zeile den Wert 30 ausgibt.

Doch bei diesem Beispiel handelt es sich nur augenscheinlich um die Anwendung von Polymorphie, denn die einzige Funktion der Basisklasse ist weder durch das Schlüsselwort `Overridable` als überschreibbar definiert, noch versucht die gleiche Funktion der abgeleiteten Klasse, diese Funktion mit `Overrides` zu überschreiben.

Bei genauerer Betrachtung zeigt Visual Basic für `EineFunktion` der abgeleiteten Klasse eine Warnmeldung an, etwa wie in Abbildung 9.21 zu sehen:



**Abbildung 9.21:** In diesem Beispiel blendet eine Funktion der abgeleiteten Klasse die der Basisklasse aus

Visual Basic gibt Ihnen hier eine Fehlermeldung aus, kompiliert das Programm aber dennoch. Die vermeintliche Fehlermeldung ist in diesem Fall nämlich nur eine Warnmeldung und macht Sie darauf aufmerksam, dass die Funktion von einer anderen überschattet wird.

Das hat Auswirkungen auf das Verhalten der Klasse. Denn obwohl die Objektvariable eine Instanz der abgeleiteten Instanz enthält, wird dennoch die Funktion der Basisklasse aufgerufen. Shadows unterbricht die Erbfolge der Klasse an dieser Stelle und stellt sicher, dass eine Funktion, die nicht zum Überschreiben markiert ist, auch tatsächlich nicht überschrieben werden kann.

Sie werden diese Warnmeldung übrigens los, indem Sie das Schlüsselwort Shadows vor die Funktion in der abgeleiteten Klasse setzen.

## Shadows als Unterbrecher der Klassenhierarchie

**BEGLEITDATEIEN:** Sie finden dieses Beispiel im Verzeichnis `.\VB 2005 - Entwicklerbuch\D - OOP\Kap09\Shadowing02`.

```
Option Explicit On
Option Strict On
```

```
Module mdlMain
```

```
    Sub Main()
        Dim locVierteKlasse As New VierteKlasse
        Dim locErsteKlasse As ErsteKlasse = locVierteKlasse
        Dim locZweiteKlasse As ZweiteKlasse = locVierteKlasse
        Dim locDritteKlasse As DritteKlasse = locVierteKlasse

        Console.WriteLine(locErsteKlasse.EineFunktion)
        Console.WriteLine(locZweiteKlasse.EineFunktion)
        Console.WriteLine(locDritteKlasse.EineFunktion)
        Console.WriteLine(locVierteKlasse.EineFunktion)
    End Sub
End Module
```

```

        Console.WriteLine()
        Console.WriteLine("Taste drücken zum Beenden!")
        Console.ReadKey()

    End Sub
End Module

Public Class ErsteKlasse

    Public Overridable Function EineFunktion() As String
        Return "Erste Klasse"
    End Function

End Class

Public Class ZweiteKlasse
    Inherits ErsteKlasse

    Public Overrides Function EineFunktion() As String
        Return "Zweite Klasse"
    End Function
End Class

Public Class DritteKlasse
    Inherits ZweiteKlasse

    Public Overrides Function EineFunktion() As String
        Return "Dritte Klasse"
    End Function

End Class

Public Class VierteKlasse
    Inherits DritteKlasse
    Public Overrides Function EineFunktion() As String
        Return "Vierte Klasse"
    End Function

End Class

```

Was glauben Sie, kommt heraus, wenn Sie das Beispiel laufen lassen? Die verschiedenen Objektnamen und Klassennamen mögen anfangs ein wenig verwirren, aber das Ergebnis ist klar: Das Programm gibt viermal den Text »vierte Klasse« aus. Klar, denn vierte Klasse wird ein einziges Mal instanziiert und durch jede andere Variable referenziert. Jede der anderen Objektvariablen ist über eine Klasse der Klassenerbfolge definiert und kann damit auch – wie Sie es an vielen Beispielen auf den vergangenen Seiten dieses Kapitels schon kennen gelernt haben – auf jede Instanz einer abgeleiteten Klasse verweisen.

Jetzt nehmen Sie eine kleine Veränderung an der zweiten Klasse vor,

```

Public Class DritteKlasse
    Inherits ZweiteKlasse

```

```
'"Overrides" wurde gegen "Overridable Shadows" ausgetauscht:
Public Overridable Shadows Function EineFunktion() As String
    Return "Dritte Klasse"
End Function
```

End Class

und raten Sie, was beim erneuten Start des Programms passiert. Die Ausgabe lautet:

```
Zweite Klasse
Zweite Klasse
Vierte Klasse
Vierte Klasse
```

Taste drücken zum Beenden!

Hätten Sie es gewusst? Dabei ist das Ergebnis gar nicht so schwer zu verstehen:

Im Prinzip hat `EineFunktion` der dritten und vierten Klasse überhaupt nichts mehr mit den ersten beiden Klassen zu tun. Durch `Shadows` in der dritten Klasse wird die Funktion komplett neu implementiert. Aus diesem Grund ist wie bei einer komplett anderen Funktion, die in `DritteKlasse` »dazukommt«, auch ein erneutes `Overridable` notwendig, denn anderenfalls könnte `VierteKlasse` die Funktion gar nicht mit `Overrides` überschreiben.

Verwirrend mag ein wenig sein, dass die erste Klasse den Text »Zweite Klasse« ausgibt. Doch welchen anderen Text soll sie sonst ausgeben? »Erste Klasse«, mag man auf den ersten Blick denken, doch das ist falsch, denn das würde gegen das Prinzip der Polymorphie verstoßen. »Vierte Klasse« kann nicht ausgegeben werden, denn diese Funktion ist aus Sicht von `ErsteKlasse` gesehen nicht erreichbar. In `DritteKlasse` wird diese Funktion schlicht und ergreifend durch eine komplett neue Version ersetzt. Das Framework rettet also, was zu retten ist, und versucht in der Erbfolge soweit wie möglich nach vorne zu gehen – und damit ist `ZweiteKlasse` die letzte Funktion, die durch Polymorphie erreichbar ist, bevor die Erbfolge durch `Shadows` in `DritteKlasse` unterbrochen wird.

Dieses Verhalten ist durchaus erwünscht, denn wenn Sie nicht wollen, dass ein Element einer Klasse überschrieben wird, dann lässt es sich auch nicht überschreiben. Die CLR garantiert immer, dass eine nicht überschreibbare Funktion ihre ursprünglichen Fähigkeiten behält, selbst wenn sie andere Funktionen in abgeleiteten Klassen mit gleichem Namen überschatten.

Intern gibt es zwei verschiedene Versionen von `EineFunktion`, und wenn Sie das Beispielprogramm wie folgt abändern, wird klar, was eigentlich passiert.

---

**BEGLEITDATEIEN:** Sie finden dieses Beispiel im Verzeichnis `.\VB 2005 - Entwicklerbuch\D - OOP\Kap09\Shadowing03`.

---

```
Option Explicit On
Option Strict On
```

```
Module mdlMain
    Sub Main()
        Dim locVierteKlasse As New VierteKlasse
        Dim locErsteKlasse As ErsteKlasse = locVierteKlasse
        Dim locZweiteKlasse As ZweiteKlasse = locVierteKlasse
        Dim locDritteKlasse As DritteKlasse = locVierteKlasse
    End Sub
End Module
```

```

        Console.WriteLine(locErsteKlasse.EineFunktion_a)
        Console.WriteLine(locZweiteKlasse.EineFunktion_a)
        Console.WriteLine(locDritteKlasse.EineFunktion_b)
        Console.WriteLine(locVierteKlasse.EineFunktion_b)

        Console.WriteLine()
        Console.WriteLine("Taste drücken zum Beenden!")
        Console.ReadKey()

    End Sub

End Module

Public Class ErsteKlasse

    Public Overridable Function EineFunktion_a() As String
        Return "Erste Klasse"
    End Function

End Class

Public Class ZweiteKlasse
    Inherits ErsteKlasse

    Public Overrides Function EineFunktion_a() As String
        Return "Zweite Klasse"
    End Function

End Class

Public Class DritteKlasse
    Inherits ZweiteKlasse

    Public Overridable Function EineFunktion_b() As String
        Return "Dritte Klasse"
    End Function
End Class

Public Class VierteKlasse
    Inherits DritteKlasse

    Public Overrides Function EineFunktion_b() As String
        Return "Vierte Klasse"
    End Function

End Class

```

## Sonderform »Modul« in Visual Basic

Module gibt es bei allen bislang existierenden .NET-Programmiersprachen nur in Visual Basic. Und auch hier ist ein Modul nichts weiter als eine Mogelpackung, denn das, was als Modul bezeichnet wird, ist im Grunde genommen nichts anderes als eine abstrakte Klasse mit statischen Methoden, Eigenschaften und Mitgliedern.

Halten wir fest:

- Ein Modul ist nicht instanzierbar. Eine abstrakte Klasse auch nicht.
- Ein Modul kann keine überschreibbaren Prozeduren zur Verfügung stellen. Die statischen Prozeduren einer Klasse können das auch nicht.
- Ein Modul kann nur Prozeduren zur Verfügung stellen, auf die aber nur ohne Instanzobjekt direkt zugegriffen werden kann. Das gleiche gilt für die statischen Prozeduren einer abstrakten Basisklasse.

Es gibt aber auch feine Unterschiede: So können Module beispielsweise keine Schnittstellen implementieren; das können zwar abstrakte Klassen, aber Sie können keine statischen Schnittstellenelemente definieren. Insofern ist dieser scheinbare Unterschied in Wirklichkeit gar keiner. Ein Modul kann auch nur auf oberster Ebene definiert und nicht in einem anderen geschachtelt werden.

Module setzen Sie bei der OOP vorschlagsweise so wenig wie möglich ein, denn sie widersprechen dem Anspruch von .NET, möglichst wieder verwendbaren Code zu schaffen.

Hier im Buch finden Sie aus diesem Grund Module nur, wenn es um »Quick-And-Dirty«-Projekte geht, bei denen beispielsweise eine Konsolen-Anwendung Tests durchzuführen hat oder »mal eben« etwas demonstrieren soll, genauso, wie Sie es in vergangenen Kapiteln bereits erlebt haben.

## Singleton-Klassen und Klassen, die sich selbst instanzieren

Stellen Sie sich vor, Sie möchten eine Klasse entwerfen, die beispielsweise die Funktionen eines Bildschirms oder eines Druckers steuert. Das Besondere daran ist, dass Sie eine Kontrolle über die Instanzierung dieser Klasse benötigen. Es reicht nicht aus, der Klasse selbst zu überlassen, wie oft sie sich instanziiert – einen bestimmten Drucker gibt es nur ein einziges Mal, und nach einer Instanz sollte Schluss sein.

Eine abstrakte Klasse mit statischen Prozeduren (wegen mir auch ein Modul) könnte vielleicht eine Alternative dazu sein – doch das Problem dabei ist: Weder die Funktionen eines Moduls noch die statischen Funktionen einer abstrakten Klasse können Sie in anderen Klassen überschreiben.

Die Lösung dazu sind so genannte Singleton-Klassen. Singleton-Klassen sind, anders als die Klassenvarianten, die Sie bislang kennen gelernt haben, keine »eingebauten« Klassentypen der FCL. Sie müssen sie selber entwickeln – doch das ist einfacher, als Sie denken.

---

**BEGLEITDATEIEN:** Sie finden dieses Beispiel im Verzeichnis `.\VB 2005 - Entwicklerbuch\D - OOP\Kap09\Singleton`.

---

```

Imports System.Threading

Module mdlMain

    Dim varModule As Integer = 5

    Sub New()

    End Sub

    Sub Main()
        Dim locSingleton As Singleton = Singleton.GetInstance()
        Dim locSingleton2 As Singleton = Singleton.GetInstance()

        Console.WriteLine(locSingleton Is locSingleton2)
        Console.WriteLine()
        Console.WriteLine("Taste drücken zum Beenden!")
        Console.ReadKey()
    End Sub

    Property test() As Integer
        Get

        End Get
        Set(ByVal Value As Integer)

        End Set
    End Property

End Module

Class Singleton

    Private Shared locSingleton As Singleton
    Private Shared locMutex As New Mutex

    'Konstruktor ist privat,
    'damit kann die Klasse nur von "sich selbst" instanziiert werden
    Private Sub New()
    End Sub

    'GetInstance gibt eine Singleton-Instanz zurück
    'Nur durch diese Funktion kann die Klasse instanziiert werden
    Public Shared Function GetInstance() As Singleton
        'Vorgang Thread-sicher machen
        'Wartet, bis ein anderer Thread, der diese Klasse
        'instanziiert, damit fertig ist
        locMutex.WaitOne()
        Try
            If locSingleton Is Nothing Then
                locSingleton = New Singleton
            End If
        Finally
    End Function

```

```

        'Instanzieren abgeschlossen,
        '...kann weiter gehen...
        locMutex.ReleaseMutex()
    End Try

    'Instanz zurückliefern
    Return locSingleton

End Function

End Class

```

Ihnen wird als Erstes auffallen, dass diese Klasse nicht direkt instanziiert werden kann, denn ihr Konstruktor ist privat. Wie also kommen Sie dann überhaupt an eine Instanz der Klasse?

Die Antwort auf diese Frage liegt in der statischen Funktion `GetInstance`. Sie sorgt dafür, dass die Klasse sich selbst instanziiert, wenn es erforderlich ist. Gleichzeitig achtet sie darauf, dass der Instanzierungsvorgang der Klasse thread-sicher ist: Das Framework erlaubt es nämlich, wahrhaftiges Multitasking in eigenen Programmen zu implementieren. Dieser Vorgang bezeichnet Zustände, bei denen mehrere Aufgaben innerhalb eines Programms (oder mehrerer Programme) gleichzeitig ausgeführt werden können. Damit ein weiterer Instanzierungsversuch einer Singleton-Klasse nicht ausgerechnet dann passiert, wenn ein anderer Thread fast (aber eben noch nicht ganz) mit der Instanzierung fertig ist – beide Threads sich sozusagen »mittendrin« treffen würden und es damit doch die zu verhindernden zwei Instanzen gäbe –, nutzt die Klasse eine Instanz von `Mutex`<sup>6</sup>, um dieses Auftreffen zu vermeiden.

Solange Sie keine Multithreading-Programmierung anwenden, brauchen Sie jedoch keinen Gedanken daran zu verschwenden. Mehr zu diesem Thema erfahren Sie im Threading-Kapitel später in diesem Buch (in ► Kapitel 31).

Das Hauptprogramm prüft die Funktionalität der Singleton-Klasse. Es holt sich eine Instanz mit `GetInstance` und speichert sie in einer Objektvariablen. Den gleichen Vorgang wiederholt es anschließend mit einer weiteren Objektvariablen und vergleicht die beiden »Instanzen« anschließend mit `Is`.

Und in der Tat: Das Programm gibt `True` auf dem Bildschirm aus, denn die Singleton-Klasse hat nur *eine* Instanz ihrer selbst kreiert – beide Objektvariablen zeigen also auf die gleiche Instanz.

---

<sup>6</sup> `Mutex`, abgeleitet von »**mutual exclusion**« etwa »gegenseitiger Ausschluss«.