Kapitel 10

Code und Klassen

Sie wissen von Funktionen und Sub-Prozeduren. Sie haben Eigenschaften-Prozeduren und Methoden kennengelernt. Sie sind mit den Techniken vertraut, Klassen anzulegen. Sie haben einen Kurs besucht oder das Handbuch gelesen. Sie sind also kein Anfänger mehr.

Ich werde Sie nicht mit all dem Zeug belästigen, das schon fast jeder Einsteiger beherrscht – jedenfalls nicht über kurze Einführungen hier und da hinaus. Statt dessen werden Sie hier Kommentare, Veranschaulichungen und hin und wieder meine Meinung vorfinden. In diesem Kapitel geht es daher um einige ausgewählte Punkte in Sachen Codierung und Anlegen von Klassen.

10.1 Methoden und Eigenschaften

Betrachten wir zunächst ein paar Dinge, die mit den Methoden und Eigenschaften (die Ihnen als Funktionen und Sub-Prozeduren vertraut sind) einer Klasse zu tun haben.

Den Beispiel-Code zu diesem Abschnitt finden Sie im Beispiel `Misc1.vbp` im Ordner zu Kapitel 10 auf der Buch-CD.

10.1.1 Get, Set, Let – auf geht's!

Ein Klassen-Modul besteht aus Funktionen, Sub-Prozeduren und Eigenschaften.

Eine Funktion und eine Sub-Prozedur sind eigentlich ein und dasselbe, abgesehen davon, daß eine Funktion einen Wert zurückgibt, eine Sub-Prozedur dagegen nicht – kein sehr schwerwiegender Unterschied. Daher macht man auch keinen weiteren Unterschied, wenn man bei Klassen in beiden Fällen von »Methoden« spricht.

Sie erinnern sich auch sicher daran, daß eine Eigenschaft intern über zwei Methoden implementiert wird – eine zum Setzen des Eigenschaftswertes, eine zweite zum Auslesen desselben.

Genau genommen sind Funktionen, Sub-Prozeduren und Eigenschaften alle dasselbe. Sie unterscheiden sich lediglich in der Syntax der Aufrufe und der Implementierung im Code.

Sie sind sich sogar so ähnlich, daß mitunter die Entscheidung nicht leicht fällt, ob eine bestimmte Aufgabe besser über eine Methode oder über eine Eigenschaft zu implementieren ist. Sogar die Syntax ist in allen Fällen sehr ähnlich:

```
Public Sub MeineSubProzedur(ParameterListe)
Public Function MeineFunktion(ParameterListe) As RückgabeWert
Public Property Get Eigenschaft(ParameterListe) As Variant
Public Property Let Eigenschaft(ParameterListe, NeuerWert _
 As Variant)
Public Property Set Eigenschaft(ParameterListe, NeuerWert _
 As Variant)
```

Sie werden sich vielleicht wundern, welchen Sinn die Parameterliste bei den Eigenschaften-Prozeduren haben soll. In Visual Basic können Sie parametrisierte Eigenschaften definieren, auch wenn diese Möglichkeit nur selten genutzt wird. Wir werden gleich noch darauf zurückkommen, aber zunächst ...

10.1.2 Mehr zu Eigenschaften

Eine Eigenschaft kann auf zweierlei Weise implementiert werden: als einfache, öffentlich deklarierte Variable eines Formulars bzw. einer Klasse oder als Satz von Eigenschaften-Prozeduren. Eine `Property Get`-Prozedur gibt den Wert der Eigenschaft zurück. Mit einer `Property Let`-Prozedur wird der Wert der Eigenschaft gesetzt. Über eine `Property Set`-Prozedur weisen Sie einer Eigenschaft eine Objekt-Referenz zu.

Sie sollten generell Eigenschaften-Prozeduren vorziehen. In Eigenschaften-Prozeduren können Sie die übergebenen Werte auf ihre Gültigkeit hin prüfen, eine Fehlerbehandlung einbauen und dem Zugriff auf eine Eigenschaft zusätzliche Funktionalität verleihen. Auch wenn Sie anfangs glauben sollten, diese zusätzliche Funktionalität nie zu brauchen, empfiehlt sich die Verwendung von Eigenschaften-Prozeduren – es wird Ihnen leichter fallen, sich mit dem Umgang mit Objekten und mit deren Möglichkeiten vertraut zu machen. Schließlich wandelt Visual Basic sowieso die öffentlich deklarierten Variablen intern in einen Satz von Eigenschaften-Prozeduren um, dagegen können Sie gar nichts unternehmen.

Sie können eine Eigenschaft als schreibgeschützt gestalten, indem Sie die `Property Let`- bzw. `Property Set`-Prozedur entfallen lassen. Umgekehrt können Sie eine Eigenschaft nur schreibbar machen, indem Sie die `Property Get`-Prozedur weglassen. Derartige Einschränkungen beispielsweise können Sie bei über öffentliche Variablen definierten Eigenschaften nicht vornehmen.

10.1.3 Eigenschaft oder Methode?

Hier nun einige Argumente, die Ihnen bei der Entscheidung zwischen der Implementierung einer Operation als Methode oder als Eigenschaft helfen können. Die ersten vier davon stammen aus der Microsoft-Dokumentation.

Argument: Daten oder Aktion

Eigenschaften sollten zum Zugriff auf die Daten eines Objekts verwendet werden, Methoden dagegen zur Ausführung einer Operation. Zum Beispiel: Die Merkmale `Color` oder `Caption` eines Objekts sind Eigenschaften – sie beziehen sich direkt auf den Dateninhalt eines Objekts. Die Operationen `Move` oder `Show` sind Methoden – sie tun etwas.

Argument: Syntax

Die Syntax der Zuweisung eines Funktions-Rückgabewerts ist identisch mit der Zuweisung eines Eigenschaftswertes. Betrachten Sie einmal die folgenden beiden Implementierungen für eine Eigenschaft bzw. Methode namens `ErrorResult`.

```
Public Function ErrorResult() As Long
      ErrorResult = 5
End Function

Property Get ErrorResult() As Long
      ErrorResult = 5
End Function
```

In beiden Fällen erfolgt die Zuweisung so:

```
MeinErgebnis = EinObjekt.ErrorResult
```

Umgekehrt sieht es allerdings anders aus. Das Setzen einer Eigenschaft `ErrorResult` sähe wie folgt aus:

```
Property Let ErrorResult(vNewValue As Long)
      ' ...verschiedene Interne Operationen
End Function
```

Und nun die Zuweisung:

```
EinObjekt.ErrorResult = IrgendeineZahl
```

Dies können Sie so natürlich nicht mit einer Methode machen. Sie müßten dazu wohl eine weitere Methode einführen, wie etwa `SetErrorResult`, der »IrgendeineZahl« als Parameter übergeben würde.

Argument: Eigenschaften-Fenster

Dieses Argument betrifft ausschließlich ActiveX-Controls. Es kann sich jedoch als hilfreicher Fingerzeig bei der Entscheidung zwischen der Implementierung als Methode oder als Eigenschaft erweisen. Würden Sie wollen, daß ein Merkmal als Eigenschaft im Eigenschaften-Fenster (bzw. in einem Eigenschaften-Dialog) erscheinen soll? Falls ja, dann ist die Implementierung als Eigenschaft eindeutig sinnvoll. Falls nicht, kann es trotzdem Fälle geben, in denen die Implementierung als Eigenschaft vorzuziehen wäre. Dieses Argument kann also in einigen Fällen eine klare Antwort geben, in anderen jedoch nicht.

Argument: Fehlersensibilität

Wenn Sie versuchen, einer schreibgeschützten Eigenschaft einen Wert zuzuweisen, werden Sie mit der Fehlermeldung »Objekt erforderlich« konfrontiert.

Versuchen Sie dagegen, einer Funktion einen Wert zuzuweisen (vorausgesetzt, die Funktion gibt kein Objekt zurück, das als Ziel der Zuweisung wirken könnte),

wird der Fehler »Funktionsaufruf auf der der linken Seite der Zuweisung muß den Typ `Variant` oder `Object` zurückgeben.« ausgelöst.

Die Entscheidung anhand der Art der Fehlermeldung zu treffen, klingt fast nach einem Akt der Verzweiflung – das nächste Argument setzt dem allerdings noch einen drauf...

Argument: Rettungsanker
Werfen Sie eine Münze!

Ehrlich gesagt, alle vorangegangenen Argumente sind gut. Es gibt viele Fälle, in denen die Entscheidung zwischen einer Implementierung als Methode oder als Eigenschaft schwer fällt. Erlauben Sie mir daher, Ihnen einige weitere hilfreiche Regeln mitzugeben.

Argument: VBX-Konsistenz
Bei der alten VBX-Technologie konnte man als Control-Entwickler noch keine eigenen Methoden implementieren. Daher wurden Eigenschaften auf so manch äußerst kreative Weise zweckentfremdet. Sie sehen das beispielsweise am damaligen Common-Dialog-Control, das eine `Action`-Eigenschaft hatte. Die Zuweisung eines Werts an diese Eigenschaft öffnete den gewünschten Dialog.

In Anbetracht des Arguments »Daten oder Aktion« hätte Action eigentlich als Methode implementiert werden müssen. Besser wären weiterhin jeweils separate Methoden zum Aufruf der verschiedenen Dialoge.

Wenn Sie nun jedoch dieses Control in Visual Basic neu schreiben wollten, würde ich Ihnen die Beibehaltung der `Action`-Eigenschaft anraten. Die Leute sind schließlich daran gewöhnt und in vorhandenen Projekten würden diesbezüglich keine Änderungen notwendig werden. Sie könnten natürlich noch die jeweiligen Dialog-Aufrufe als zusätzliche Methoden implementieren.

Argument: Parameterlisten
Wie Sie bereits gesehen haben, können Eigenschaften tatsächlich mit mehreren Parametern neben der reinen Wertzuweisung versehen werden. Bis auf wenige Ausnahmen, in denen ein Array-Effekt beabsichtigt ist, sollten Sie eine Methode anstelle einer Eigenschaft implementieren, wenn Sie Parameter bei einer Eigenschaft verwenden wollen.

Kommen wir nun zu den parametrisierten Eigenschaften ...

10.1.4 Parametrisierte Eigenschaften

Angenommen, Ihr Objekt soll eine Eigenschaft als Array offenlegen. Dies könnten Sie etwa so implementieren:

```
Public T(8) As Variant
```

Hier würde Ihnen Visual Basic jedoch einen Strich durch die Rechnung machen – öffentliche Arrays werden nämlich nicht unterstützt. Zum Glück können Sie diesen Effekt jedoch über eine parametrisierte Eigenschaft erhalten:

```
Private m_t(8) As Variant

Public Property Get T(Index As Integer) As Variant
   T = m_t(Index)
End Property

Public Property Let T(Index As Integer, ByVal NeuerWert As Variant)
   m_t(Index) = NeuerWert
End Property
```

Der folgende Code bestätigt dies:

```
Dim tx As New clsIrgendwas
tx.T(1) = 5
tx.T(2) = 6
Debug.Print tx.T(1)
Debug.Print tx.T(2)
```

Die Werte 5 und 6 werden tatsächlich im Direktfenster ausgegeben.

Sie können auch zweidimensionale Arrays als Eigenschaft implementieren, indem Sie zwei Parameter als Index verwenden, usw. Dies funktioniert, solange die Parameterlisten der `Get`- und `Let`-Prozeduren identisch sind. Im konkreten Fall sollten Sie jedoch eine Fehlerbehandlung einfügen, um die Übergabe gültiger Index-Werte sicherzustellen.

Da Sie nun über parametrisierte Eigenschaften Bescheid wissen, vergessen Sie sie am besten gleich wieder. In den meisten Fällen, in denen Parameter benötigt werden, sollten Sie eine Funktion oder eine Sub-Prozedur verwenden. Mehr dazu erfahren Sie später.

10.1.5 Objekte als Eigenschaften

Weiter vorne in diesem Buch sahen Sie, daß sich die `Set`-Anweisung von einer gewöhnlichen Zuweisung unterscheidet, da eine Objekt-Referenz anstelle einer einfachen Variablen zugewiesen wird. Dies betrifft genauso Eigenschaften, die ein Objekt darstellen.

Das Beispiel-Projekt `Misc1.vbp` enthält die Klasse `clsMisc2` mit folgender Eigenschaft:

```
Public Property Get ClassName() As String
   ClassName = "clsMisc2"
End Property
```

Dies ist die Standard-Eigenschaft der Klasse (Voreinstellung). Wenn Sie etwa auf ein Objekt des Typs `clsMisc2` die Operation

```
Debug.Print obj
```

ausführen, wird »`clsMisc2`« im Direktfenster ausgegeben.

Eine Objekt-Eigenschaft in einer Klasse `clsMisc1`, die ein Objekt des Typs `clsMisc2` aufnehmen soll, könnte so definiert werden:

```
Private m_Object As clsMisc2
' Beispiel einer Objekt-Eigenschaft
Public Property Get O1() As clsMisc2
   Set O1 = m_Object
End Property

Public Property Set O1(ByVal vNewValue As clsMisc2)
   Set m_Object = vNewValue
End Property
```

Der Zugriff darauf würde beispielsweise so erfolgen:

```
   Dim tx As New clsMisc1
   Dim tobj As New clsMisc2
   Set tx.O1 = tobj
   Debug.Print tx.O1
```

Damit wird wieder »`clsMisc2`« im Direktfenster ausgegeben. Beachten Sie, daß die Zuweisung an die Eigenschaft `tx.O1` über die `Set`-Anweisung erfolgen muß.

10.1.6 Überladen von Eigenschaft und Funktionen

Sie kennen ja Variants, diesen »supertollen, fantastischen, magischen« Variablen-Typ, der Daten nahezu jeden Typs aufnehmen kann. Ich habe Ihnen gleich noch einiges mehr zu Variants zu sagen. Aber ich denke, ich sollte Ihnen auch einmal sagen, wozu sie gut sein können, bevor ich Ihnen erkläre, warum sie so fürchterlich sind.

In C++ und anderen objektorientierten Sprachen gibt es ein Konzept, das Visual Basic betreffend nur selten zur Sprache kommt. Es geht dabei um die Möglichkeit, daß ein Objekt gleichzeitig über mehrere Funktionen mit dem gleichen Namen verfügen kann, die anhand des Datentyps voneinander unterschieden werden.

So könnte ein Objekt etwa zwei `Print`-Funktionen enthalten, wobei die eine einen String, die andere eine Bitmap als Parameter akzeptiert. Wird nun beispielsweise `Print ("MeinString")` aufgerufen, wird die erstere Funktion aufgerufen und der String ausgegeben. Wird dagegen `Print (BitmapObjektReferenz)` aufgerufen, wird die zweite Funktion aufgerufen und statt dessen die Bitmap ausgegeben.

Ein C++-Programmierer wird dafür in der Tat zwei separate Funktionen implementieren. Der Kompiler legt dann anhand des übergebenen Datentyps fest, welche der `Print`-Funktionen tatsächlich aufgerufen werden soll.

Visual Basic unterstützt dieses Konzept nicht direkt. Dennoch kann es durch die Verwendung von Variants mit leichten Einschränkungen aufgegriffen werden. Da ein Variant nahezu jeden Datentyp aufnehmen kann, läßt sich zur Laufzeit per Code entscheiden, welchen Datentyp der übergebene `Variant`-Parameter enthält.

Könnte man so eine `Variant`-Eigenschaft anlegen, die sowohl ein Objekt als auch andere Datentypen aufnehmen könnte, wie etwa im folgenden Code – würde das funktionieren? Nein.

```
Public Property Get V1() As Variant
   V1 = m_Variant
End Property

Public Property Let V1(ByVal vNewValue As Variant)
   m_Variant = vNewValue
End Property
```

Warum nicht? Während dies bei den meisten Datentypen einwandfrei funktioniert, würde es bei Objekt-Typen fehlschlagen, die eine Zuweisung über eine `Set`-Anweisung erfordern. Statt dessen müssen Sie die Operation an den Datentyp im `Variant`-Parameter anpassen. Enthält der `Variant`-Parameter ein Objekt, müssen Sie die `Set`-Anweisung verwenden. Der folgende Code funktioniert:

```
' Beispiel einer Variant-Eigenschaft
Public Property Get V1() As Variant
   If VarType(m_Variant) = vbObject Then
      Set V1 = m_Variant
   Else
      V1 = m_Variant
   End If
End Property

Public Property Let V1(ByVal vNewValue As Variant)
   If VarType(vNewValue) = vbObject Then
      Set m_Variant = vNewValue
   Else
      m_Variant = vNewValue
   End If
End Property
```

Zum Testen dieser Eigenschaft legen Sie eine neue Klasse `clsMisc3` an, die mit der Klasse `clsMisc2` übereinstimmt, außer daß die Eigenschaft `ClassName` nicht als Standard-Eigenschaft der Klasse festgelegt ist. Dann lassen Sie einmal den folgenden Code laufen:

```
Private Sub cmdVariant1_Click()
   Dim tv As New clsMisc1
   Dim obj2 As New clsMisc2
   Dim obj3 As New clsMisc3
   tv.V1 = 5
   Debug.Print tv.V1
   tv.V1 = obj3
   Debug.Print tv.V1.ClassName
   Debug.Print VarType(tv)
   tv.V1 = obj2
   Debug.Print tv.V1
   Debug.Print VarType(tv.V1)
End Sub
```

Folgendes wird im Direktfenster ausgegeben:

```
5
clsMisc3
 9
clsMisc2
 8
```

In diesen Ausgaben steckt eine interessante Feinheit. Wenn die Zuweisung `tv.V1 = obj3` ausgeführt wird, stellt die Eigenschaft fest, daß ein Objekt übergeben worden ist, und führt erwartungsgemäß die `Set`-Anweisung aus. Dies wird auch durch die Rückgabe des Wertes 9 (`vbObject`) durch die `VarType`-Funktion bestätigt. Doch was passierte hier mit `clsMisc2`? Erinnern Sie sich daran, daß bei `clsMisc2` die Eigenschaft `ClassName` als Standard-Eigenschaft festgelegt ist, und daß diese eine `String`-Eigenschaft ist. Bei der Zuweisung `tv.V1 = obj2` enthält der `Variant`-Parameter die `String`-Rückgabe der Standard-Eigenschaft anstelle des Objekts selbst. Dies ist eines der vielen Beispiele einer Situation, in der die Verwendung einer `Variant`-Variablen die Chancen verringert, daß Sie selbst oder irgend jemand anders durch Ihren Code durchblickt.

Dies verdeutlicht ebenfalls mein Widerstreben gegen die Verwendung von Standard-Eigenschaften. Auf den ersten Blick scheinen sie eine praktische Reduzierung des Tippaufwands beim Programmieren zu sein. Sie verringern jedoch die Lesbarkeit und die Wartbarkeit einer Anwendung.

Das hier gezeigte Prinzip für den Umgang mit Objekten oder anderen Datentypen kann auf alle `Variant`-Datentypen ausgedehnt werden. So kann beispielsweise die `Remove`-Methode einer Collection sowohl einen String als auch einen numerischen Wert als Parameter annehmen. Wird ein String übergeben, wird der Parameter als Schlüssel interpretiert. Wird eine Ganzzahl übergeben, wird der Parameter als Index interpretiert. Auch wenn ich Microsofts Quellcode des `Collection`-Objekts nicht zu Gesicht bekommen habe, gehe ich jede Wette darauf ein, daß man dabei im Prinzip die hier beschriebene Technik angewendet hat.

10.1.7 Eigenschaften überladen auf etwas verquere Weise

Die Klasse `clsMisc1` enthält eine zweite Eigenschaft, `V2`, die einen anderen Weg des Überladens einer Eigenschaft demonstriert. In diesem Fall jedoch dient dieser Weg nur dazu, zwischen einer Variablen- und einer Objekt-Zuweisung zu unterscheiden. Es stellt sich die Frage, was passiert, wenn eine Eigenschaft sowohl über eine `Property Let`- als auch über eine `Property Set`-Methode verfügt.

```
Public Property Get V2() As Variant
   '...
End Property

Public Property Let V2(ByVal vNewValue As Variant)
   MsgBox "I'm in Let V2"
End Property

Public Property Set V2(ByVal vNewValue As Variant)
   MsgBox "I'm in Set V2"
End Property
```

Probieren Sie folgenden Test-Code aus:

```
Dim tv As New clsMisc1
Dim obj As New clsMisc2
tv.V2 = 5
tv.V2 = obj
Set tv.V2 = obj
```

Die ersten beiden Anweisungen sind identisch mit denen des vorigen Beispiels. In beiden Fällen wird der Wert (`5` oder `obj`) in einen `Variant` umgewandelt und an die `Property Let`-Prozedur übergeben. In der dritten Anweisung wird `obj` gleichfalls wie zuvor in einen `Variant` umgewandelt, nun jedoch an die `Property Set`-Prozedur übergeben.

Es wäre doch gelacht, wenn es nicht einen Weg gäbe, die Umwandlung in einen `Variant` zu umgehen. Nun, es gibt einen Weg:

```
Public Property Set V2(vNewValue As Object)
   MsgBox "I'm in Set V2"
End Property
```

Dies ist definitiv effizienter als der `Variant`-Ansatz. Tatsächlich kann `vNewValue` als jeder beliebiger Objekt-Typ deklariert werden (wie etwa hier als `clsMisc2`) – es wird bestens funktionieren.

Aber verletzt dies nicht die Regel, daß der von einer Eigenschaft zurückgegebene Datentyp dem Datentyp beim Setzen der Eigenschaft entsprechen muß? Es zeigt sich, daß für die `Property Set`-Funktion eigene Regeln gelten. Der Datentyp, der einer `Property Set`-Funktion übergeben wird, kann jeder beliebige Objekt-Typ sein, ganz und gar unabhängig vom eigentlichen Datentyp der Eigenschaft

(dem Datentyp, der von `Property Get` zurückgegeben wird). So wird beispielsweise folgendes Muster einer Definition für eine Eigenschaft möglich:

```
Public Property Get T() As Integer
End Property

Public Property Let T(ByVal vNewValue As Integer)
End Property

Public Property Set T(x As Object)
End Property
```

Ja, Sie lesen richtig. `T` ist eine Integer-Eigenschaft, der auch ein Objekt zugewiesen werden kann! Angenommen, die Klasse `MeinObjekt` enthält so eine Eigenschaft, sind beide Fälle gültig:

```
obj.T = 5
Set obj.T = obj  ' oder jedes andere Objekt
```

Nun, da Sie wissen, daß es geht, lassen Sie besser die Finger davon. Soweit ich das sehe, gibt es keine praktische Verwendung dafür. Sie verwirren damit nur jeden, der mit solchem Code arbeiten soll.

Etwas anderes sollte Ihnen in den obenstehenden kleinen Test-Prográmmchen aufgefallen sein: Die Zuweisung eines Objekts zu einer `Variant`-Eigenschaft erfordert keine ausdrückliche `Set`-Anweisung. Visual Basic wandelt das Objekt bei der Übergabe als Parameter selbständig in einen `Variant` um. Manchmal übertreibt es Visual Basic jedoch mit seiner Fürsorge.

10.1.8 Boshafter Typ-Zwang

Was denken Sie – was wird durch folgenden Code ausgegeben?

```
Dim s$
s$ = "1"
s$ = s$ + 5
Debug.Print s$
Debug.Print s$ + 8
Debug.Print s$ + "8"
```

Vielleicht 15, 158, 158?

Ließen Sie das unter Visual Basic 3.0 laufen, erhielten Sie eine Fehlermeldung, die einen falschen Datentyp reklamieren würde. Richtig, Visual Basic 3 läßt Sie keinen String und eine Zahl addieren – solch eine Operation sei sinnlos. Sie müßten die Zahl zuerst ausdrücklich in einen String konvertieren, etwa so: `s$ = s$ + Str$(5)`.

Seit Visual Basic 4.0 würde der Code jedoch dies ausgeben: 6, 14, 68. Das bedeutet, daß Visual Basic zwangsweise Variablen verschiedener Datentypen konvertiert, damit eine Operation durchführbar wird.

Darin steckt jedoch ein gewisser Widerspruch. Ich habe zum Beispiel in der Schule gelernt, daß eine konsequente Typprüfung eine gute Sache ist. Es kann zwar durchaus möglich sein, daß meine Anweisung `s$ = s$ + 5` volle Absicht ist, doch ist es eher wahrscheinlich, daß ich damit versehentlich einen Bug fabriziert habe. Sicher, Microsoft empfiehlt, den Verkettungs-Operator »&« zu verwenden, um Strings aneinanderzureihen. Doch das trifft nicht den Kern der Sache – es erhöht lediglich die Wahrscheinlichkeit eines Bugs. Bei anderen Datentypen konvertiert Visual Basic nämlich weiterhin fröhlich drauf los.

Als professioneller Programmierer erwarte ich eigentlich von einem Compiler, so viele Bugs wie möglich aufzudecken. Schauen Sie sich einmal folgenden Code an:

```
Dim I As Integer
I = 55.5
Debug.Print I
```

Ausgegeben wird 56. Aus der einen Perspektive (aus Microsofts) ist das so in Ordnung. Offensichtlich weiß doch ein Programmierer, was er tut, wenn er eine Fließkommazahl einem Integer zuweist, und er wird sich darüber freuen, daß der Kompiler die notwendige Konvertierung automatisch vornimmt.

Ich hätte jedoch nichts dagegen, in so einem Fall ausdrücklich eine Konvertierung, hier etwa über die Anweisung `CInt`, vornehmen zu müssen. Wenn Sie in einer meiner Anwendungen derartigen Code gefunden hätten, wären Sie damit höchstwahrscheinlich auf einen Programmiererfehler gestoßen. Denn es hätte ja auch sein können, daß ich eigentlich eine Fließkommavariable hätte verwenden wollen. Wenn der Kompiler auf eine derartige Typ-Diskrepanz stoßen würde, könnte ich mich um das Problem kümmern und entweder meine eigene Konvertierung einfügen oder das Problem anderweitig in den Griff bekommen. So jedoch riskiere ich, daß der Bug unentdeckt bleibt, bis ich vielleicht auf Ungenauigkeiten in den Berechnungen meiner Anwendung aufmerksam (gemacht) werde – unter Umständen lange Zeit nach der Auslieferung meiner Anwendung.

Dies weist auf einen fundamentalen Unterschied der Programmierphilosophie hin. Eine ganze Reihe von Programmierern, einschließlich ich selbst, haben Microsoft darum gebeten, die derzeitige Handhabung nicht zu ändern, jedoch eine Option in die Entwicklungsumgebung einzufügen, die eine striktere Form der Typprüfung ermöglicht (etwa in der Art, wie nach Visual Basic 1.0 die Anweisung `Option Explicit` eingeführt wurde).

Wenn Sie mit dieser Ansicht übereinstimmen, möchte ich Sie gerne dazu einladen, dazu Microsoft direkt zu kontaktieren und eine Nachricht etwa folgenden Inhalts zuzusenden: »Ich unterstütze eine Option für strikte Datentypprüfung in

kommenden Visual-Basic-Versionen.« Sie können sich dabei gerne auf dieses Buch berufen. Senden Sie die Nachricht (natürlich in Englisch) per E-Mail an vbwish@microsoft.com.

Ich weiß nicht, ob das was bewirken wird – ob sich überhaupt jemand um solche Wünsche kümmert. Ich weiß nur, daß ich eine solche Aktion schon in der ersten Ausgabe dieses Buches zu Visual Basic 5.0 vorgeschlagen habe, jedoch das Problem in Visual Basic 6.0 weiterhin existiert. Daher ermuntere ich Sie, Ihren Wunsch noch einmal an Microsoft abzusetzen – vielleicht kommt es ja dann in Visual Basic 7...

Falls Sie jedoch nicht mit meiner Ansicht übereinstimmen sollten, bitte ich Sie natürlich darum, diese für sich zu behalten.

Eine Bemerkung am Rande: Ich kann mir vorstellen, daß einige unter Ihnen nun denken werden, daß ein derartiger Kommentar ein wenig überzogen sein dürfte. Wie komme ich denn dazu, Visual Basic auf diese Weise zu kritisieren, und dies dazu in einem Buch, das im Sinn hat, Sie von der Großartigkeit von Visual Basic zu überzeugen? Wie komme ich dazu, eine derart offensichtlich persönliche Meinung zu propagieren, in einem seriösen technischen Buch, das doch naturgemäß objektiv sein sollte? Nun, wenn Sie so denken sollten, dann tut es mir leid, ändert aber nichts daran. Ich habe Sie in der Einführung vorgewarnt, daß dieses Buch auch eine Art Kommentar zur Dokumentation sein soll, und kein Ersatz derselben. Dies ist ein technisches Buch, und es ist ein seriöses (mehr oder weniger), aber ich habe keineswegs Objektivität versprochen, ausgenommen, es kommen harte Fakten auf den Tisch.

10.1.9 Option Explicit

Es dreht sich um die gleiche Angelegenheit: Als erstes nach der Installation von Visual Basic sollten Sie in den Optionen der Entwicklungsumgebung (Menü EXTRAS/OPTIONEN) im Register EDITOR die Option VARIABLENDEKLARATION ERFORDERLICH setzen. Damit stellen Sie sicher, daß bei einem falsch geschriebenen Variablennamen der Kompiler Sie warnt, anstelle automatisch eine neue Variable unter dem falsch geschriebenen Namen anzulegen.

10.1.10 Mehr zu Variants

Sie haben gerade gesehen, daß Variants beim Überladen von Methoden und Eigenschaften recht nützlich sein können. Sie sind auch dann nützlich, wenn sie einen Wert aufnehmen sollen, dessen Datentyp zur Design-Zeit noch nicht feststeht, oder wenn eine Liste von Variablen verschiedene Datentypen aufnehmen können soll. Es gibt sicher noch einige weitere Situationen, in denen Variants nützlich sein können.

Hier nun das Wichtigste, was Sie über Variants wissen sollten. Abgesehen von den erwähnten speziellen Fällen, in denen es ohne die Verwendung eines Variants nicht geht, sollten Sie sie vermeiden. Warum?

- Sie sind langsam.
- Sie sind langsam und ineffizient.
- Sie sind langsam und verschwenden Speicher.
- Sie führen zu zusätzlichen Datentyp-Konvertierungen, von denen sich viele Ihrer Kontrolle entziehen und Ihre Anwendung bremsen.
- Sie zwingen zu besonderer Sorgfalt beim Testen bei der Verwendung als Funktions-Parameter (die Funktion muß sowohl jeden Datentyp als auch jeden möglichen Wert verarbeiten können).

In der Visual-Basic-Dokumentation wird auf diese Punkte nur ungenügend eingegangen, insbesondere kaum auf die folgenden:

- Wenn Sie eine Variable oder einen Parameter ohne Typangabe deklarieren, verwendet Visual Basic automatisch Variants.
- Vergessen Sie die Deklaration des Rückgabewerts einer Funktion, verwendet Visual Basic automatisch Variants.
- Fügen Sie eine neue Eigenschaften-Prozedur über den PROZEDUR-EINFÜGEN-Dialog ein, verwendet Visual Basic automatisch Variants.
- Wenn Sie mehrere Variablen mit einer `Dim`-Anweisung deklarieren (das ist eine Angewohnheit meist von C-Programmierern), kann es leicht passieren, daß Sie unbeabsichtigt Variants deklarieren. In der folgenden Anwendung sind beispielsweise die beiden Variablen `i` und `j` Variants:

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

Bemühen Sie sich um Code-Effizienz! Denken Sie daran, für alle Variablen und Funktions-Parameter Datentypen anzugeben. Ändern Sie die Datentypen aller Eigenschaften entsprechend den Erfordernissen. Zumindest könnten Sie eine `Def`-Anweisung zu Beginn jedes Moduls einfügen, um als Standard-Datentyp einen anderen als Variant festzulegen.

Insbesondere im Hinblick auf die Möglichkeit, Native Code zu kompilieren, ergibt dies Sinn, da in Native Code einige Datentypen (ganz besonders `Integer` und `Long`) höchst effizient verarbeitet werden.

Das Projekt `VarTest.vbp` im Ordner zu Kapitel 10 auf der Buch-CD demonstriert die Unterschiede in der Performance von Variants und nativen Datentypen. Es wird die Zeit gemessen, die zur mehrfachen Ausführung der folgenden beiden Prozeduren benötigt wird:

```
Public Sub LongTest()
   Dim l As Long
   Dim ctr As Long
   l = 100
```

```
    For ctr = 1 To 1000
       l = l + 1
    Next ctr
End Sub

Public Sub VariantTest()
    Dim v As Variant
    Dim ctr As Variant
    v = 100
    For ctr = 1 To 1000
       v = v + 1
    Next ctr
End Sub
```

Ist dies ein fairer Test? Natürlich nicht. Wer weiß schon, welcher Datentyp in der Schleife der Prozedur `VariantTest` tatsächlich verwendet wird – es könnte ja auch ein Fließkommatyp sein.

Die Ergebnisse sprechen jedoch Bände. Bei meinen Tests stellte ich fest, daß `VariantTest` innerhalb der Visual Basic-Entwicklungsumgebung etwa 2,2mal langsamer als die `LongTest`-Prozedur war. In einer als P-Code kompilierten ausführbaren Datei erhöhte sich die Diskrepanz auf das 2,7fache. Bei einem Native-Code-Kompilat schließlich benötigte die `VariantTest`-Prozedur 45mal so lange wie die `LongTest`-Prozedur.

Das ist keineswegs überraschend. Die `Variant`-Funktionalität ist Bestandteil von OLE und die `VariantTest`-Prozedur greift wahrscheinlich auf die Visual-Basic-Runtime-Datei zurück. Auch beim Native-Code-Kompilat müssen hier die Runtime-Funktionen aufgerufen werden, so daß der Unterschied zwischen Native Code und P-Code kaum ins Gewicht fällt (dies stützt meine Aussage aus dem vorhergehenden Kapitel, daß in vielen Fällen ein Native-Code-Kompilat nur geringen Vorteil bringt). Auf der anderen Seite entspricht die `LongTest`-Prozedur genau solchen rechenintensiven Operationen, bei denen Native Code seine Vorteile ausspielen kann. Dies ist keineswegs ein an den Haaren herbeigezogenes Beispiel – Schleifen gehören nunmal zum Standardwerkzeug eines jeden Programmierers.

Wir schließen daraus:

- Wenn Sie Zähler-Schleifen in Ihrem Code verwenden, verwenden Sie möglichst `Long`- oder `Integer`-Variablen.
- Wenn Sie viele derartige Schleifen verwenden, oder diese sehr umfangreich sind, wird Ihre Anwendung ihre Stärke am besten als Native-Code-Kompilat ausspielen können. Auf jeden Fall ist es einen Versuch wert.
- Wenn es keinen ausdrücklichen Grund für die Verwendung von Variants gibt, lassen Sie die Finger davon.

10.1.11 Optionale Parameter und Parameter-Arrays

Es lohnt sich, einen kurzen Blick auf zwei weitere Visual-Basic-Features in Sachen Funktions-Parameter zu werfen: optionale Parameter und Parameter-Arrays. Diese Features werden in der Beispiel-Projektgruppe `Misc2.vbg` demonstriert, die zwei Projekte enthält: `Misc2.vbp` und `Misc2tst.vbp`.

Sie können Funktions-Parameter optional halten, indem Sie das Schlüsselwort »`optional`« voranstellen. Ein oder mehrere optionale(r) Parameter kann/können als Parameterliste einer Funktion übergeben werden. Jedoch kann auf einen optionalen Parameter kein nichtoptionaler Parameter folgen. Das bedeutet, daß alle optionalen Parameter als letzte in der Liste erscheinen müssen.

Optionale Parameter können von jedem Datentyp sein. Visual Basic macht jedoch einen Unterschied bei der Behandlung von `Variant`-Datentypen und anderen Datentypen. Ist ein optionaler Parameter vom Datentyp `Variant`, kann die Funktion `IsMissing` zur Prüfung verwendet werden, ob an dieser Stelle tatsächlich ein Parameter übergeben worden ist. Dies verdeutlicht das Beispiel `Optional1`, in dem die Funktion `Optional1` der Klasse `clsMisc2` wie folgt erscheint:

```
Public Function Optional1(Optional vr As Variant) _
 As Variant
   If IsMissing(vr) Then
      Debug.Print "Variable vr is missing"
   Else
      Debug.Print "Variable vr is present"
   End If
End Function
```

Der Test erfolgt in der Ereignis-Prozedur `cmdOptional1_Click` im Hauptformular von `Misc2tst`:

```
Private Sub cmdOptional1_Click()
   Dim obj As New clsMisc2B
   Debug.Print "Calling with no Parameter"
   obj.Optional1
   Debug.Print "Calling with Parameter"
   obj.Optional1 "Hello"
End Sub
```

Das Ganze sieht etwas anders aus, wenn ein anderer Datentyp verwendet wird, wie etwa im Beispiel `Optional2` und der Test-Prozedur `cmdOptional2_Click`:

```
Public Function Optional2(Optional vr As String) _
 As Variant
   If IsMissing(vr) Then
      Debug.Print "Variable vr is missing"
   Else
      Debug.Print "Variable vr is: " & vr
```

```
   End If
End Function

' IsMissing funktioniert nicht
' mit andere Datentypen als Variant
Private Sub cmdOptional2_Click()
   Dim obj As New clsMisc2B
   Debug.Print "Calling with no Parameter"
   obj.Optional2
   Debug.Print "Calling with Parameter"
   obj.Optional2 "Hello"
End Sub
```

Hier wird niemals die Meldung »Variable vr is missing« im Direktfenster erscheinen. Der optionale Parameter wird statt dessen mit dem Standard-Wert des angegebenen Datentyps initialisiert (etwa mit 0 oder einem Leerstring).

Mit Visual Basic 5.0 wurden auch Vorgabewerte für Parameter eingeführt, wie es im Beispiel `Optional3` und der Test-Prozedur `cmdOptional3_Click` gezeigt wird:

```
' Mit Vorgabewerten
Public Function Optional3(Optional vr As String = _
 "A Default Value") As Variant
   If IsMissing(vr) Then
      Debug.Print "Variable vr is missing"
   Else
      Debug.Print "Variable vr is: " & vr
   End If
End Function

' Demonstration von Vorgabewerten für Parameter
Private Sub cmdOptional3_Click()
   Dim obj As New clsMisc2B
   Debug.Print "Calling with no Parameter"
   obj.Optional3
   Debug.Print "Calling with Parameter"
   obj.Optional3 "Hello"
End Sub
```

Hier wird der String mit dem angegebenen Vorgabewert initialisiert, wenn Sie der Funktion keinen `String`-Parameter übergeben.

Fall Sie nun in Versuchung kommen sollten, ausgiebigen Gebrauch von optionalen Parametern zu machen, möchte ich Ihnen davon abraten. Optionale Parameter sind am nützlichsten, wenn sie eine Funktion haben, die in der Regel mit festen Parametern und nur selten mit den optionalen Parametern aufgerufen wird. Eine zu große Anzahl von optionalen Parametern erschwert das Verständnis und die

Wartbarkeit des Codes. Schließlich haben Sie ja bereits gesehen, daß es `Variant`-Parameter zu vermeiden gilt.

An einer Stelle können Sie gelegentlich eine Vielzahl an optionalen Parametern bei öffentlichen Methoden antreffen. So etwa bei Anwendungen wie Microsoft Word, wo die Funktionalität von komplexen Dialogen offengelegt wird und jeder Parameter einem Feld oder einem Control eines Dialogs zugeordnet ist. In den meisten Fällen wäre ein Ansatz mit einem neuen, eigenen Objekt besser, das dem Dialog zugeordnet ist und die Felder des Dialogs über Eigenschaften offenlegt. Dann würden wenige Methoden zum Aufruf der Dialog-Funktionalität ausreichen.

Neben optionalen Parametern unterstützt Visual Basic Parameter-Arrays, in denen eine beliebige Anzahl von Parametern jedes beliebigen Typs an eine Funktion übergeben werden kann (`ParamArray`-Argumente müssen Variants sein). Dies wird in der Methode `ParamArrayDemo` und der Prozedur `cmdparamArray_Click` gezeigt:

```
Public Function ParamArrayDemo(ParamArray vr())
   Debug.Print "Parameters range from: " & LBound(vr) _
    & " to " & UBound(vr)
End Function

' Demonstration von ParamArray
Private Sub cmdparamArray_Click()
   Dim obj As New clsMisc2B
   obj.ParamArrayDemo "Hello", 1, 2.5
End Sub
```

Während schon optionale Parameter die Lesbarkeit einer Funktion beeinträchtigen, verringern Parameter-Arrays diese noch zusätzlich. Zumindest sollten Sie dafür sorgen, daß die Parameterliste einen klaren Bezug hat, etwa als zusätzliche Parameterliste, in der etwa eine Liste von Elementen übergeben wird, die die Funktion beispielsweise sortieren soll usw.

10.1.12 Parameter für größere Datenmengen: Arrays und benutzerdefiniert

Gelegentlich wird es nötig sein, einen schnellen Zugriff auf größere, in einem Objekt gekapselte Datenmengen oder -einheiten zu bekommen. Schließlich steht ja hinter dem Konzept von Objekten die Kapselung von Daten hinter einem klar definierten Interface. Der einzige Weg, auf die Daten in einem Objekt zuzugreifen, sollte über die Methoden und Eigenschaften des Interfaces führen.

Was ist jedoch, wenn das Objekt eine große Menge Daten enthält? Vielleicht ein großes Bild oder Multimedia-Daten? Oder wenn es über Hunderte von verschiedenen Eigenschaften statt nur über ein paar Dutzend verfügt?

Die Sache ist bei einem Bild oder einer Multimedia-Datei, die als Array eines Standard-Datentyps repräsentiert werden können (wie etwa durch ein `Byte`-Array), recht einfach in den Griff zu bekommen. Sie könnten eine `Byte`-Eigenschaft verwenden, die einen Index-Wert akzeptiert (ein parametrisiertes Array also) und die Daten Byte für Byte kopieren. Jedoch wäre der Overhead des Aufrufs der Methode für jedes einzelne Byte erheblich – sogar katastrophal, falls sich das Objekt in einem anderen Prozeßraum befinden sollte (etwa in einem ActiveX-EXE-Server). Jedoch erlaubt Visual Basic 6.0 die Verwendung von Arrays als Funktions-Parameter und damit den Transport über eine Eigenschaft (das Array kann aber auch nach wie vor in einen Variant verpackt werden).

Die Beispiel-Projektgruppe `misc2.vbg` zu diesem Kapitel enthält die folgende Methode in der Klasse `clsMisc2`:

```
Public Function ArrayParam(param() As Byte)
    Debug.Print "Param() range is: " & LBound(param) _
     & " to " & UBound(param)
 End Function
```

Diese Methode wird von folgendem Test-Code im Formular `frmMisc2` des Projekts `misc2tst.vbp` aufgerufen:

```
' Demonstration von Array-Parametern
Private Sub cmdArray_Click()
   Dim obj As New clsMisc2B
   Dim x(5) As Byte
   obj.ArrayParam x()
End Sub
```

Wie Sie sehen, berücksichtigt die Methode korrekt die untere und die obere Grenze des Arrays. Seit Visual Basic 6 können auch Arrays als Rückgabewert von Eigenschaften-Prozeduren zurückgegeben werden, so daß Sie tatsächlich auch Array-Eigenschaften anlegen können.

Was machen Sie jedoch, wenn die Daten des Objekts komplexerer Natur sind? Zum Beispiel könnte es vorkommen, daß ein Objekt ein Gerät in einer Labor-Versuchsanordnung repräsentiert, das vielleicht 250 Einstellungen in einer benutzerdefinierten Struktur zusammenfaßt, damit sie vom Treiber des Geräts gelesen werden können. Es könnte aber auch sein, daß Daten nicht in einem `Byte`-Array, sondern in einem Array eines benutzerdefinierten Datentyps enthalten sind. In solchen Fällen kann der Zugriff auf Hunderte von Eigenschaften oder Elemente die Performance erheblich in den Keller drücken, vor allem dann, wenn sich das Objekt in einem anderen Prozeß befinden sollte.

In solchen Fällen bietet es sich an, die Daten des Objekts in einer benutzerdefinierten Struktur (benutzerdefinierte Variable in Visual Basic) zu speichern und diese Struktur über einen öffentlichen Datentyp der Klasse offenzulegen. Dann kann die ganze Struktur über eine einzige Eigenschaft übertragen werden. Bevor

Sie sich nun auf diese Möglichkeit stürzen, sollten Sie jedoch noch über folgendes nachdenken:

- Könnte das Objekt möglicherweise in mehrere Teil-Objekte zerlegt werden? Könnte ein Array eines benutzerdefinierten Datentyps eventuell durch ein Objekt-Array ersetzt werden (Objekte können sowohl in einem Array als auch in einer Collection transportiert werden)?
- Ist der Zugriff auf die Daten wirklich notwendig? Vielleicht könnte das Objekt die notwendigen Operationen selbst ausführen. Ein Beispiel hierfür wäre das Schreiben der Daten in eine Datei bzw. das Zurücklesen, was am besten im Code des Objekts selbst zu implementieren wäre.
- Muß der Zugriff auf die Daten öffentlich sein? Solange der Zugriff nur aus der gleichen Anwendung heraus zu erfolgen braucht, können Sie eine Methode auch als `Friend` deklarieren. `Friend`-Funktionen können auch nicht öffentlich deklarierte benutzerdefinierte Datentypen als Parameter verarbeiten. Diese können einfach in einem globalen Standard-Modul der Anwendung deklariert werden.
- Brauchen Sie wirklich den blockweisen Zugriff? Immerhin sind In-Process-Zugriffe sehr schnell.

Man könnte auch sagen, daß die Offenlegung großer Datenblöcke ein Zeichen für schlechtes Design sein kann.

Es gibt auch Fälle, in denen der Zugriff auf große Datenblöcke kritisch ist. Es kann durchaus deutlich schneller gehen, ein Objekt zu kopieren. Es kann natürlich vorkommen, daß ein schneller Treiber auch einen schnellen Zugriff auf ein Objekt über Datenblöcke in einem bestimmten Format erfordert. Und der Zugriff auf große Datenblöcke kann bei der Arbeit mit größeren Windows-Strukturen sinnvoll sein.

Der einfachste Weg ist, den benutzerdefinierten Datentyp öffentlich in einem Klassen-Modul zu deklarieren. Damit wird die Definition des Datentyps in die Typbibliothek der Komponente aufgenommen, so daß er als Datentyp für Eigenschaften und Parameter verwendet werden kann. Der folgende Code zeigt dies:

```
Public Type usertype
   x(20) As Byte
   y As Integer
End Type

Public Property Get UserProperty() As usertype
   Dim TempUserType As usertype
   TempUserType.y = 77  ' Test-Wert
   UserProperty = TempUserType
End Property
```

```
Public Property Let UserProperty(vNewValue As usertype)
    Dim TempUserType As usertype
    TempUserType = vNewValue
    Debug.Print TempUserType.y
End Property
```

Der folgende Code im Formular `frmMisc2` demonstriert die Verwendung einer solchen Eigenschaft.

```
Private Sub cmdUserType_Click()
   Dim TempUserType As usertype
   Dim obj As New clsMisc2B
   TempUserType.y = 55  ' Test Wert
   obj.UserProperty = TempUserType
   TempUserType = obj.UserProperty
   Debug.Print TempUserType.y
End Sub
```

Ein anderer Ansatz wird bei früheren Visual-Basic-Versionen notwendig, in denen öffentliche Datentypen noch nicht unterstützt wurden. Dieser Ansatz wird bei der `UserTypeEquiv`-Eigenschaft in der Klasse `clsMisc2` und der `UserType2`-Anweisung im Formular `frmMisc2` gezeigt. Das Beispiel greift auf eine Speicherkopie-Funktion zurück, um die Daten aus einer und in eine Eigenschaft des Datentyps `Variant` zu kopieren.

10.2 Prozedur-Attribute

In Visual Basic können Sie eine ganze Reihe von Attributen für Methoden und Eigenschaften einer Klasse, eines Formulars, eines ActiveX-Controls oder eines ActiveX-Dokuments setzen. Sie können zwar derartige Attribute auch bei Funktionen in Standard-Modulen setzen. Doch da diese Attribute keinerlei Bedeutung für Methoden außerhalb des Moduls haben, die COM-Objekte unterstützen, wäre dies eigentlich sinnlos – außer zu reinen Dokumentationszwecken. Die Attribute werden im Dialog PROZEDURATTRIBUTE gesetzt. Sie können diesen Dialog über das Menü EXTRAS/PROZEDURATTRIBUTE öffnen, wenn ein Code-Fenster eines Moduls aktiv ist. In Abbildung 10.1 sind die erweiterten Attribute zu sehen.

Viele der Attribute (vor allem die erweiterten Attribute) betreffen lediglich ActiveX-Controls. Diese werden im Teil III dieses Buches behandelt.

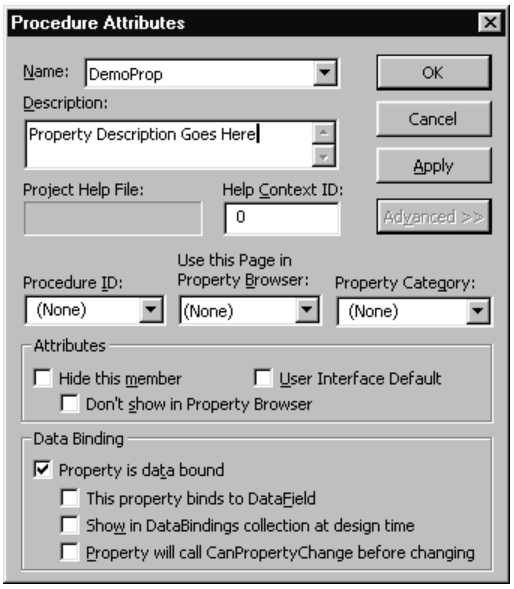

Abb. 10.1: Der Prozedurattribute-*Dialog*

10.2.1 Beschreibung

Sie sollten hier für jede öffentliche Methode und Eigenschaft eine Beschreibung eingeben. Auch für `private` Methoden und Eigenschaften empfiehlt es sich, Beschreibungen einzugeben, vor allem bei umfangreicheren Klassen. Diese Beschreibung wird im Objekt-Katalog und im Eigenschaften-Fenster angezeigt.

10.2.2 Hilfekontext-ID

Geben Sie hier die Kontext-ID für die betreffende Methode oder Eigenschaft aus der Hilfe-Datei (falls eine existiert) der Komponente an. Diese Hilfekontext-ID wird vom Objekt-Katalog und vom Eigenschaften-Fenster (bei ActiveX-Controls) verwendet.

10.2.3 Prozedur-ID

In Kapitel 4, »Das Component Object Model: Interfaces, Automation und Bindung«, haben wir besprochen, wie Dispatch-Interfaces funktionieren. Sie werden sich daran erinnern, daß jeder Methode und jeder Eigenschaft eines Dispatch-Interfaces eine Kennung zugeordnet ist, die Dispatch-ID oder Prozedur-ID genannt wird. Diese Dispatch-ID wird für den Aufruf von Methoden des Interfaces benötigt, um festzulegen, welche Methode oder Eigenschaft aufgerufen werden soll. Beachten Sie, daß diese Kennungen zwar eindeutig innerhalb eines Interfaces sein müssen, aber keine Regel etwas darüber aussagt, ob diese Kennun-

gen in sequentieller Reihenfolge zu vergeben sind oder ob sie überhaupt irgendwelche bestimmten Werte sein müssen.

OLE definiert eine Reihe von Standard-Dispatch-IDs, die jeweils eine spezielle Bedeutung haben. Die Verwendung dieser IDs betrifft nicht die Art und Weise des Aufrufs von Methoden und Eigenschaften eines Objekts. Es gibt jedoch Objekt-Container, die beim Antreffen dieser speziellen Dispatch-IDs auf besondere, festgelegte Weise reagieren können.

Ein Beispiel hierfür ist die Dispatch-ID 0 (Null). Die Methode oder Eigenschaft, die dieser ID zugeordnet ist, wird von Visual Basic (und vielen anderen Containern) als Standard-Methode bzw. -Eigenschaft betrachtet. Dies bedeutet, daß bei einem Zugriff auf ein Objekt ohne Angabe einer Methode oder Eigenschaft standardmäßig auf diese Methode oder Eigenschaft zugegriffen wird.

Beispielsweise sieht der Zugriff bei einem Objekt namens `OBJ` mit der Eigenschaft `MeineEigenschaft` normalerweise so aus:

```
MeineVariable = OBJ.MeineEigenschaft
```

Ist jedoch `MeineEigenschaft` als Standard-Eigenschaft gesetzt, können Sie auch einfach so darauf zugreifen:

```
MeineVariable = OBJ
```

Viele Entwickler wie auch die Dokumentation von Microsoft betonen, wie wichtig und sinnvoll es ist, eine logische, naheliegende Eigenschaft als Standard-Eigenschaft zu wählen. So ist beispielsweise beim `TextBox`-Control die Eigenschaft `Text` die Standard-Eigenschaft. Beim `Label`-Control ist es die `Caption`-Eigenschaft.

Wenn ich Ihnen auch nicht guten Gewissens von der Festlegung einer Standard-Eigenschaft bei Ihren Objekten abraten kann, empfehle ich im allgemeinen dennoch, darauf zu verzichten. Der Grund hierfür liegt einzig und allein in einer besseren Lesbarkeit und Wartbarkeit des Codes.

Wenn Sie etwa die Code-Zeile `MeineVariable = MeinObjekt` sehen, können Sie nicht auf den ersten Blick erkennen, auf welche Eigenschaft tatsächlich zugegriffen wird. Sie müßten es schon auswendig wissen oder erst umständlich nachschlagen. Auf jeden Fall bedeutet es zusätzlichen Aufwand.

Die Form `MeineVariable = MeinObjekt.XYEigenschaft` ist dagegen auf den ersten Blick eindeutig. Man weiß sofort, auf welche Eigenschaft zugegriffen wird. Der geringe zusätzliche Aufwand, den Namen der Eigenschaft hinzuzufügen (und durch die QuickInfo-Fähigkeiten seit Visual Basic 5 ist der Aufwand wirklich nur minimal), wird wegen der besseren Lesbarkeit des Codes mehr als aufgewogen.

Mehr zu Prozedur-IDs erfahren Sie in Teil III im Zusammenhang mit ihrer Verwendung bei ActiveX-Controls.

10.2.4 Dieses Mitglied ausblenden

Diese Option setzt ein spezielles Flag in der Typbibliothek eines Objekts, das dafür sorgt, daß die Methode oder Eigenschaft verborgen bleibt. Dann wird diese Methode oder Eigenschaft in einem Objekt-Katalog nicht angezeigt. Bei ActiveX-Controls erscheint eine verborgene Eigenschaft nicht im Eigenschaften-Fenster. Sie können auf verborgene Methoden oder Eigenschaften dennoch per Code zugreifen.

Die Verwendung dieser Option soll einen einfachen Schutzmechanismus bieten. Nur wenn jemand über die Dokumentation zu verborgenen Eigenschaften und Methoden verfügt, können diese problemlos verwendet werden. Gelegentlich werden auch als öffentlich deklarierte Methoden und Eigenschaften verborgen, die von einer Anwendung zu speziellen internen Zwecken benötigt werden, jedoch nicht unbedingt zur Verwendung durch jedermann gedacht sind.

Auf die verbleibenden erweiterten Attribute gehen wir in Teil III im Zusammenhang mit ihrer Verwendung bei ActiveX-Controls näher ein.

10.3 Objekt-Prozeduren: Public, Private oder Friend

Die Verwendung des Worts `Public` ist in Visual Basic fast genauso verwirrend wie die des Worts `Object` (Objekt). Die Bedeutung hängt nämlich stark vom jeweiligen Kontext ab. Ist die Rede von einer DLL-Deklaration? Von einer Variablen-Deklaration? Oder von einem Klassen-Modul? In diesem Abschnitt befassen wir uns damit, wie in Visual Basic die Aspekte der Sichtbarkeit von Objekt-Methoden und -Eigenschaften gehandhabt werden, und mit einigen Regeln dazu, die die Variablen innerhalb einer Anwendung betreffen.

10.3.1 Public ist und bleibt Public

In Kapitel 8, Das »Projekt«, haben wir den Unterschied zwischen privaten und öffentlichen Objekten herausgearbeitet. Der Typ eines Objekts wird über die `Instancing`-Eigenschaft einer Klasse festgelegt.

Die Verwendung des Begriffs »`Public`« an dieser Stelle betrifft die Sichtbarkeit von Methoden und Eigenschaften eines Klassen-Moduls. Ist die `Instancing`-Eigenschaft einer Klasse auf irgendeinen der öffentlichen Typen gesetzt, kann das von dieser Klasse implementierte Objekt von anderen Anwendungen eingesetzt werden. Alle oder auch nur einzelne Eigenschaften des Objekts können ebenso von anderen Anwendungen gebraucht werden. Ist die `Instancing`-Eigenschaft einer Klasse jedoch auf `Private` gesetzt, kann das von ihr implementierte Objekt nicht von anderen Anwendungen verwendet werden.

Die `Instancing`-Eigenschaft kann man als übergeordnete Einstellung betrachten, die festlegt, ob das Objekt außerhalb der Anwendung eingesetzt werden kann. Sie betrifft nicht den Einsatz eines Objekts innerhalb der gleichen Anwendung. In einem Projekt können alle Objekte frei instanziert und verwendet werden, die von

Klassen innerhalb der gleichen Anwendung implementiert werden, unabhängig von der Einstellung der `Instancing`-Eigenschaft.

Methoden und Eigenschaften

Die Schlüsselwörter `Public` und `Private` können bei der Deklaration einer Methode oder Eigenschaft in einem Modul verwendet werden (dies betrifft Klassen-, Standard- als auch Formular-Module). Damit wird die Sichtbarkeit einer Methode oder Eigenschaft innerhalb einer Anwendung festgelegt.

Es gilt eine einfache Regel: Ist die Methode/Eigenschaft/Funktion als `Public` (öffentlich) deklariert, kann auf sie von anderen Modulen der Anwendung zugegriffen werden. Ist sie als `Private` deklariert, kann auf sie nur innerhalb desselben Moduls zugegriffen werden.

Diese ziemlich einfache Regel zieht einen Seiteneffekt nach sich, an den manche Visual-Basic-Programmierer, die Visual Basic schon sehr lange kennen, oft nicht denken. Da sich diese Regel gleichfalls auf Formulare bezieht, können auch Formulare öffentliche Methoden und Eigenschaften enthalten. So kann ein Formular ein zweites Formular laden und vom Entwickler zusätzlich definierte Eigenschaften dieses Formulars setzen (oder Methoden aufrufen), bevor es angezeigt wird. Öffentliche Methoden und Eigenschaften wurden in früheren Versionen von Visual Basic noch nicht unterstützt – viele Entwickler haben noch gar nicht mitbekommen, daß das mittlerweile geht. Ein Beispiel für dieses Feature sehen Sie im Beispiel-Projekt `Misc3.vbp`, das Sie im Ordner zu Kapitel 10 auf der Buch-CD finden.

Die besagte simple Regel definiert die Sichtbarkeit von Funktionen innerhalb eines Projekts – wie steht es jedoch mit der Sichtbarkeit außerhalb des Projekts? Ist eine Methode oder Eigenschaft als `Public` deklariert und zugleich das Objekt über die `Instancing`-Eigenschaft der Klasse ebenfalls als `Public` deklariert, ist die betreffende Methode oder Eigenschaft ebenfalls nach außen hin öffentlich.

Dies wirft ein interessantes Problem bezüglich öffentlicher Klassen auf: Was ist, wenn Sie wollen, daß eine Methode oder Eigenschaft öffentlich innerhalb des Projekts der Komponente sein soll, aber für Clients des Objekts trotzdem nicht sichtbar, also privat sein soll? Anders gefragt: Wie macht man eine Methode oder Eigenschaft privat nach außen hin und gleichzeitig öffentlich innerhalb der Komponenten-Anwendung?

Dies wird über die Deklaration mit dem Schlüsselwort `Friend` erreicht. Abbildung 10.2 illustriert die Sichtbarkeiten von als `Public`, `Private` und `Friend` deklarierten Methoden und Eigenschaften von öffentlichen und privaten Klassen.

Eine interessante Technik ergibt sich, wenn Sie beispielsweise bei einer Eigenschaft die `Property Let`- und/oder die `Property Set`-Prozedur als `Private` deklarieren, die `Property Get`-Prozedur hingegen als `Public`. Damit können Sie Eigenschaften anlegen, die zwar von anderen Anwendungen gelesen, aber nur innerhalb des Komponenten-Projekts gesetzt werden können.

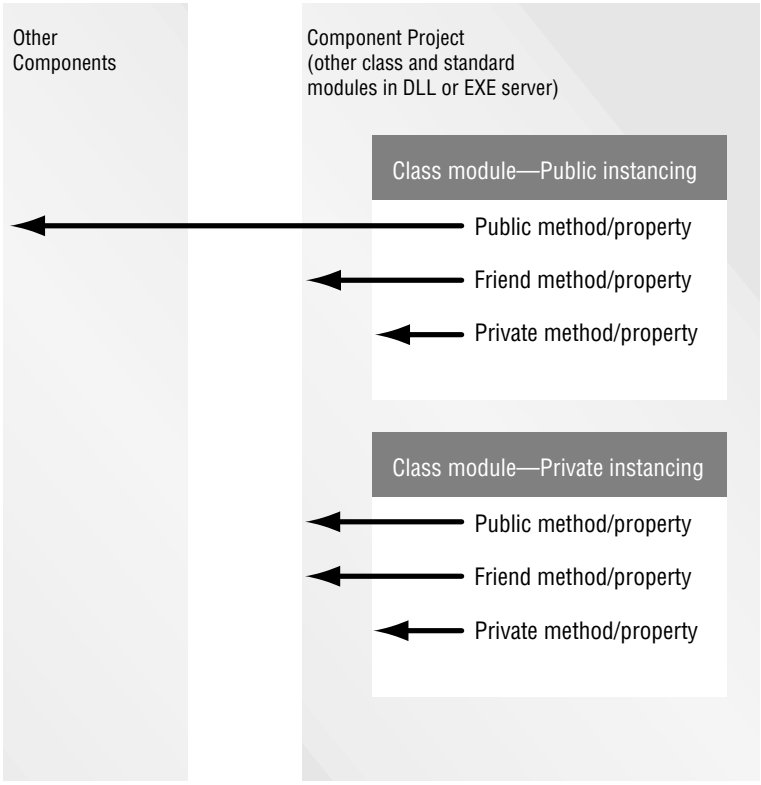

Abb. 10.2: Festlegung der Sichtbarkeit von Methoden und Eigenschaften

Es gibt einen kleinen Haken an der `Friend`-Deklaration: Der Zugriff auf diese Funktionen muß über frühe Bindung erfolgen. Der Zugriff auf eine `Friend`-Funktion eines Objekts, das einer als `As Object` deklarierten Variablen zugewiesen ist, ist leider nicht möglich.

Variablen

Für Variablen gelten nahezu die gleichen Regeln wie für Methoden und Eigenschaften. Private Variablen sind nur innerhalb des Moduls sichtbar, in dem sie deklariert sind. Öffentliche Variablen einer öffentlichen Klasse werden für andere Anwendungen offengelegt (Visual Basic erzeugt automatisch die entsprechenden `Property Get`- und `Property Let/Set`-Methoden für eine Variable). Bei Variablen ist eine `Friend`-Deklaration allerdings nicht möglich.

Privatheit als höchstes Gut

Ein wesentlicher Punkt beim Entwurf eines Objekt-Modells ist die Festlegung, welche Methoden und Eigenschaften öffentlich sein sollen. Generell gilt, daß Sie soviel Privatheit wie möglich wahren sollten.

Stellen wir uns eine Klasse mit der Funktion `MeineFunktion` mit mehreren Parametern vor.

Nehmen wir an, die Funktion sei privat und wird mehrmals innerhalb der Klasse aufgerufen. Somit können Sie die Implementierung und das Testen der Funktion darauf aufsetzen lassen, wie sie verwendet werden soll.

Wird die Funktion immer nur mit bekannten Parameter-Werten aufgerufen, können Sie den Aufwand einer Fehlerbehandlung für den Fall der Übergabe anderer Parameter-Werte sparen. Sie können es dabei bei einigen wenigen Kommentaren zu möglichen und sinnvollen Parameter-Werten für die künftige Wartung der Funktion belassen.

Ist die Funktion öffentlich innerhalb des Projekts (als `Public` in einer privaten Klasse oder als `Friend` in einer öffentlichen Klasse deklariert), ist die Situation dieselbe, jedoch vor einem leicht erweiterten Hintergrund. Solange Sie Bugs in Ihrem Programm-Entwurf und den Funktions-Aufrufen ausschließen können, können Sie davon ausgehen, daß die Funktion immer ihre Arbeit verrichten wird, und daß keine aufwendige Fehlerbehandlung im Code der Funktion notwendig wird.

Ist die Funktion jedoch eine öffentliche Methode einer öffentlichen Klasse, steht alles in den Sternen. Jede Anwendung, die Ihre Komponente einsetzt, kann die Methode aufrufen, mit jedem von der Methode unterstützten Datentyp und jedem beliebigen dem Datentyp entsprechenden Wert. Das bedeutet, daß `Variant`-Parameter immer auf Datentyp und Wert hin geprüft werden müssen. Bei einigen Eigenschaften wird die Prüfung des Wertes notwendig. Ihre Funktion muß mit allem fertig werden können, was in sie hineingestopft wird.

Reduzieren Sie die Sichtbarkeit von Variablen, Methoden und Eigenschaften in Ihrer Anwendung so weit es geht. Ihr Code wird so modularer, zuverlässiger und wartbarer.

10.3.2 Gültigkeitsbereichsregeln

Die Sichtbarkeit ist eines der Merkmale einer Funktion oder Variablen, die unter den Begriff Gültigkeitsbereich fallen. Die Sichtbarkeit legt fest, von wo aus auf eine Funktion oder Variable zugegriffen werden kann. Kann auf sie zugegriffen werden, ist sie auch sichtbar. Der Gültigkeitsbereich einer Variablen oder Funktion bezieht sich sowohl auf die Sichtbarkeit als auch auf die Lebensdauer – die Zeit, während der eine Variable existiert.

Sie haben gesehen, daß die Sichtbarkeit letztlich von der Verwendung der Schlüsselwörter `Public`, `Private` oder `Friend` bei der Deklaration abhängt. Die Sichtbarkeit einer Variablen hängt zudem vom Ort der Variablen-Deklaration ab. Variablen-Deklarationen innerhalb einer Funktion oder als Funktions-Parameter sind den Variablen-Deklarationen gleichen Namens auf Modul-Ebene übergeordnet. Dies wird im folgendem trivialen Programm deutlich:

```
Option Explicit

Private x As Integer

Private Sub Command1_Click()
   Dim x As Integer
   x = 6 ' Dies hat keine Auswirkung auf die modulweit
         ' gültige Variable
   Debug.Print x
End Sub

Private Sub Command2_Click()
   Debug.Print x
End Sub

Private Sub Form_Load()
   x = 5
End Sub
```

Beträgt der Wert der Variablen `x` nun 5 oder 6? Innerhalb der Prozedur `Command1_Click` beträgt er 6, da die innerhalb dieser Funktion mit der `Dim`-Anweisung deklarierte Variable sichtbar ist. Da sie den gleichen Namen wie die modulweit gültige Variable `x` trägt, bleibt letztere innerhalb der Funktion verborgen. Die Variable in der Funktion kann auf 6 gesetzt werden, was jedoch keine Auswirkung auf die modulweit gültige Variable hat. In der Prozedur `Command2_Click` wird keine weitere Variable deklariert – somit gibt es hier auch nichts zu verbergen und die modulweite Variable bleibt sichtbar.

Sie haben auch gesehen, daß die Sichtbarkeit ebenfalls Methoden und Eigenschaften von Objekten betrifft und in erster Linie sowohl von den Schlüsselwörtern `Public`, `Private` und `Friend` als auch der `Instancing`-Eigenschaft einer Klasse festgelegt wird. Die Sichtbarkeit von Objekt-Methoden und -Eigenschaften sowie einiger Anweisungen und Konstanten auf Anwendungsebene oder globaler Ebene kann gleichfalls von der Reihenfolge der in eine Anwendung aufgenommenen Typbibliothek-Verweise abhängen. Über diesen Punkt hatten wir bereits in Kapitel 9 gesprochen.

Lebensdauer

Die Lebensdauer einer Variablen hängt davon ab, wie lange sie existiert. Die Lebensdauer ist nicht dasselbe wie die Sichtbarkeit, auch wenn sie damit zusammenhängt. So ist es beispielsweise offensichtlich, daß ein Objekt, das nicht mehr existiert, auch nicht mehr sichtbar sein kann. Die Lebensdauer hängt auch von der Verwendung des Visual-Basic-Schlüsselworts `Static` ab. C- und C++-Programmierer müssen hier aufpassen: Auch wenn das Schlüsselwort `Static` in Visual Basic einige Aspekte des Schlüsselworts `Static` in C und C++ widerspiegelt, gibt es dennoch einige wesentliche Unterschiede.

In der Regel existieren Variablen, die innerhalb einer Prozedur deklariert sind, nur während des Aufrufs dieser Prozedur. Die Variable wird auf dem Stack angelegt, und wenn die Prozedur verlassen wird, wird der Speicher an das System zurückgegeben.

Wird eine Variable als `Static` innerhalb einer Prozedur deklariert, wird sie anstatt auf dem Stack im Datensegment der Anwendung angelegt. Der Wert der Variablen bleibt so zwischen den Aufrufen der Prozedur erhalten. Wird eine Prozedur als `Static` deklariert, bleiben die Werte aller innerhalb dieser Prozedur deklarierten Variablen zwischen den Aufrufen erhalten.

Tabelle 10.1 faßt die Lebensdauer von Variablen in Abhängigkeit vom Ort der Deklaration und der Verwendung des Schlüsselworts `Static` zusammen.

Deklaration	**Ort der Deklaration**	**Lebensdauer**	**Erstellung**
Modul-Ebene – nicht `Static`	Standard-Modul	Dauer der Anwendung bzw. des Threads*	Einmal während der Dauer der Anwendung*
	Formular-Modul	Lebensdauer des Formulars	Einmal je Formular
	Klassen-Modul	Lebensdauer des Klassen-Objekts	Einmal je Klassen-Objekt
Modul-Ebene – mit `Static`-Schlüsselwort	Nicht erlaubt		
Innerhalb nicht als Static deklarierter Prozedur – nicht `Static`, auch nicht bei Prozedur-Parametern	In jedem Modul-Typ	Während des Prozedur-Aufrufs	Einmal je Prozedur-Aufruf
Innerhalb einer nicht als `Static` deklarierten Prozedur mit Schlüsselwort `Static`, oder innerhalb einer als Static deklarierten Prozedur	Standard-Modul	Dauer der Anwendung oder des Threads*	Einmal während der Dauer der Anwendung*

Tab. 10.1: *Lebensdauer von Variablen*

Deklaration	Ort der Deklaration	Lebensdauer	Erstellung
	Formular-Modul	Lebensdauer des Formulars	Einmal je Formular
	Klassen-Modul	Lebensdauer des Klassen-Objekts	Einmal je Klassen-Objekt

Tab. 10.1: *Lebensdauer von Variablen*

* Bei einem Multithreading-Server kommen bestimmte Regeln zur Geltung. Näheres siehe Kapitel 14.

Die Lebensdauer von globalen Variablen entspricht der Dauer der Anwendung. Bei der Implementierung eines Multithreading-Servers kommen jedoch bestimmte Regeln zur Geltung. Auf diese wird in Kapitel 14 näher eingegangen.

Wenn Sie eine Prozedur als `Static` deklarieren, sind alle Variablen innerhalb dieser Prozedur als `Static` definiert. Daraus ergibt sich das Problem, daß Sie in dieser Prozedur keine dynamischen Variablen mehr anlegen können – die meisten Programmierer sind es jedoch gewohnt, Prozedur-Variablen dynamisch anzulegen. Außerdem ist die Deklaration einer Prozedur als `Static` so selten, daß andere Leute beim Lesen des Codes davon nur allzuleicht verwirrt werden.

Ich kann mich nicht daran erinnern, in meinen fünf Jahren der Visual-Basic-Programmierung jemals eine als `Static` deklarierte Prozedur verwendet zu haben. Als `Static` deklarierte Variablen innerhalb einer Prozedur habe ich jedoch des öfteren verwendet. Diese bieten einen Mechanismus zum Anlegen von Variablen, die effektiv privat für eine einzelne Prozedur sind. Das ist viel besser als das Anlegen von globalen Variablen, da so verhindert wird, daß eine andere Prozedur versehentlich den Wert der Variablen ändern kann. Damit wird deutlich, daß das Schlüsselwort `Static` die Lebensdauer einer Variablen der einer Variablen auf Modul-Ebene gleichsetzt, jedoch die Sichtbarkeit der Variablen auf die Prozedur beschränkt, in der sie deklariert ist.

Die Lebensdauer eines Objekts legt fest, wann eine Variable jeweils angelegt wird. Eine Variable in einem Standard-Modul eines Nicht-Multithreading-Servers wird angelegt, wenn die Anwendung geladen wird und sie existiert während der gesamten Dauer der Anwendung.

Eine Variable auf Formular-Ebene gehört zum Formular-Objekt und nicht zum Fenster des Formulars. Wenn Sie also das Formular entladen, ohne die Referenz auf das Objekt freizugeben, wird die Variable weiterhin existieren. Dies kann Verwirrung stiften, wenn Sie ein Formular entladen, später wieder laden und dann feststellen, daß die Formular-Variablen nicht neu initialisiert worden sind. Formular-Variablen leben bis zum Auftreten des `Form_Terminate`-Ereignisses.

Ist eine Variable in einem Formular-Objekt deklariert, wird auch bei jeder neuen Instanz des Objekts eine neue Instanz der Variablen angelegt.

Jedes Formular-Objekt verfügt so über eine eigene Instanz der Variablen. Jedoch können beide Variablen sichtbar sein! Angenommen, ein Formular enthält eine Variable namens `ThisFormName` und Sie haben zwei globale Formular-Objekte in einem Standard-Modul, die den gleichen Formular-Typ referenzieren: `MyForm1` und `MyForm2`. Sie können dies anhand Codes im Standard-Modul `Misc4.bas` des Beispiels `Misc4.vbp` nachvollziehen:

```
' Guide to Perplexed - Misc4 sample
' Copyright (c) 1997 by Desaware Inc. All Rights Reserved

Option Explicit

Public MyForm1 As frmMisc4
Public MyForm2 As frmMisc4

Sub Main()
   ' Objekt-Variablen initialisieren
   Set MyForm1 = New frmMisc4
   Set MyForm2 = New frmMisc4
   MyForm1.ThisFormName = "MyForm1"
   MyForm2.ThisFormName = "MyForm2"
   ' Nun beide Formulare anzeigen
   MyForm1.Show
   MyForm2.Show
End Sub
```

Bei diesem Projekt wird die Prozedur `Sub Main` zuerst ausgeführt. `ThisFormName` ist eine öffentliche Variable des Formulars auf Modul-Ebene. Das Schlüsselwort `Public` macht die Variable anwendungsweit sichtbar. Die Deklaration der Variablen auf Modul-Ebene des Formulars (im Allgemein-Abschnitt des Code-Fensters) sorgt dafür, daß die Lebensdauer der Variablen der des Formulars entspricht. Dabei wird für jedes Formular-Objekt eine neue Variable angelegt (siehe Tabelle 10.1).

Das Formular enthält eine Schaltfläche und folgenden Code:

```
' Guide to Perplexed - Misc4 sample
' Copyright (c) 1997 by Desaware Inc. All Rights Reserved

Option Explicit

Public ThisFormName As String

Private Sub Command1_Click()
   Debug.Print "Clicked" & ThisFormName
   Debug.Print MyForm1.ThisFormName
   Debug.Print MyForm2.ThisFormName
End Sub
```

```
Private Sub Form_Load()
   Caption = ThisFormName
End Sub

' Globale Variablen aufräumen, wenn die Formulare
' geschlossen werden
Private Sub Form_Unload(Cancel As Integer)
   If MyForm1 Is Me Then Set MyForm1 = Nothing
   If MyForm2 Is Me Then Set MyForm2 = Nothing
End Sub
```

Wenn Sie auf die Schaltfläche klicken, verweist die Variable `ThisFormName` auf diejenige im aktuellen Formular. Für jedes Formular gibt es die verborgene Objekt-Referenz namens Me, die auf das Formular-Objekt verweist, das gerade ausgeführt wird. Der Zugriff auf `ThisFormName` ohne Angabe einer Objekt-Referenz führt zu einem automatischen Zugriff auf `Me.ThisFormName`. Da `ThisFormName` öffentlich ist, kann auf diese Variable von jedem Formular-Objekt aus zugegriffen werden, wie Sie in diesem Beispiel sehen können. Die Lebensdauer der Variablen ist von der des Formulars abhängig und die Sichtbarkeit ist projektweit.

Sie können exakt dasselbe auch mit Klassen-Objekten machen. Legen Sie einfach eine öffentliche Variable in einer Klasse an und ersetzen Sie die Formular-Objekte durch Klassen. Versuchen Sie dann einmal, auf die Objekt-Variablen sowohl von innerhalb des Standard-Moduls als auch von Methoden des Klassen-Moduls aus zuzugreifen. Sie werden die gleichen Resultate erhalten.

Geht das auch, daß eine Variable privat für eine Klasse, dabei jedoch öffentlich für alle Instanzen dieser Klasse sein kann? Daß also die Sichtbarkeit einer Variablen auf eine bestimmte Klasse beschränkt ist, diese Variable jedoch nur einmal angelegt wird und während der gesamten Dauer der Anwendung existiert? Auf diese Weise könnten Daten von allen Objekten einer Klasse gemeinsam genutzt werden – und eben nur von diesen. In C und C++ kann dies über die Verwendung des Schlüsselworts `Static` auf Modul-Ebene erreicht werden (Klassen-Deklarations-Ebene). In Visual Basic ist dies jedoch nicht direkt möglich. Statt dessen können Sie einfach eine öffentliche globale Variable in einem Standard-Modul anlegen. Achten Sie jedoch darauf, auf diese Variable nur aus den entsprechenden Klassen-Modulen heraus zuzugreifen. Dies ist keine ideale Lösung, da trotzdem von überall her in der Anwendung auf diese Variable zugegriffen und ihr Wert verändert werden kann. Wenn Sie jedoch einen klar identifizierbaren Namen für diese Variable verwenden und sorgsam damit umgehen, wird die Gefahr erheblich geringer.

Denken Sie daran, daß eine Variable, die in einer nicht `Static` deklarierten Prozedur deklariert ist, jedesmal angelegt wird, wenn die Prozedur aufgerufen wird und so lange existiert, bis die Prozedur verlassen wird. Dies gilt genauso für Parameter der Prozedur, die so behandelt werden, als wären Sie lokal angelegt und von

der aufrufenden Routine initialisiert worden. Für `ByRef`-Parameter gilt dies ebenfalls – diese können sich auf eine Variable in der aufrufenden Funktion beziehen, wobei die referenzierte Variable selbst auf dem Stack übergeben wird und somit lokal für die aufgerufene Prozedur ist. In allen Fällen gilt, daß bei einem Aufruf einer Prozedur durch sich selbst (dies ist eine Technik, die »Rekursion« genannt wird) eine ganze Menge an Variablen angelegt wird.

Ein klassisches Beispiel einer Rekursion zeigt das folgende Listing. Die Anweisung `Debug.Print` läßt Sie das Anlegen und Freigeben der Variablen auf Prozedur-Ebene (der Prozedur-Parameter) verfolgen:

```
' Guide to Perplexed - Recursion example
' Copyright (c) 1997 by Desaware Inc. All Rights Reserved

Option Explicit

Private Sub cmdCalculate_Click()
   Dim N As Integer
   Dim result As Double
   N = Val(txtValue.Text)
   result = Factorial(N)
   lblResult.Caption = "Factorial of " & N & " is " _
    & Str$(result)
End Sub

' Fakultät N ist N * (N-1) * (N-2), usw.
' d.h. Fakultät 5 ist 5 * 4 * 3 * 2 * 1
Public Function Factorial(ByVal N As Integer) As Double
   Debug.Print "N is created with value " & N
   If N <= 1 Then
      Factorial = 1
   Else
      Factorial = CDbl(N) * Factorial(N - 1)
   End If
   Debug.Print "N with value " & N & _
    " is about to be destroyed "
End Function
```

10.4 Verschiedenes

Der Abschnitt »Verschiedenes« eignet sich hervorragend dazu, dies und jenes zu beschreiben, das nicht umfangreich genug für ein eigenes Kapitel ist und das auch sonst nicht so recht irgendwohin paßt.

10.4.1 Enumerationen

An dieser Stelle erwähne ich das erste Mal in diesem Buch Enumerationen, einem mit Visual Basic 5.0 neu eingeführten Feature. Sie werden in Teil III über ActiveX-Controls mehr dazu erfahren, wo sie im Zusammenhang mit ActiveX-Control-Eigenschaften besprochen werden.

Auf der einen Seite stellen Enumerationen eine nette Technologie dar, auf der anderen Seite sind sie völlig überflüssig. Denn eigentlich nützen Sie in Code-Komponenten gar nichts. Und da sie ihren eigentlichen praktischen Nutzen nur im Zusammenhang mit ActiveX-Controls entfalten, sind sie einigen Beschränkungen unterworfen, wie Sie später noch sehen werden. Doch zunächst ein wenig Hintergrund.

Sie haben gesehen, daß ActiveX-Controls einiges an Informationen zur Verfügung stellen können, was von anderen Anwendungen genutzt werden kann. Zu diesen Informationen gehören die Methoden und Eigenschaften eines Objekts, wie auch die ProgID und der GUID des Objekts. Sie haben auch gelernt, daß das Zurverfügungstellen dieser Informationen über Einträge in der System-Registrierung erfolgt (die bei der Registrierung einer Komponente vorgenommen werden). Zu diesen Informationen gehört auch der Ort der Typbibliothek zu der Komponente, die eine Ressource darstellt, die alle Eigenschaften und Methoden der Komponente definiert. Diese Ressource wird von Visual Basic automatisch der ausführbaren Datei hinzugefügt (EXE, DLL oder OCX – je nach Komponenten-Typ).

In einer Typbibliothek kann jedoch mehr als nur die Methoden und Eigenschaften beschrieben werden. Dort können auch Konstanten offengelegt werden. Sie können beispielsweise folgende Enumeration anlegen:

```
Public Enum TestEnum
      Test1 = 1
      Test2 = 2
End Enum
```

Jede Anwendung, die über einen Verweis auf Ihre Komponente verfügt, kann nun `Test1` und `Test2` zur Repräsentation der Werte 1 und 2 verwenden. Sie können sogar Eigenschaften anlegen, deren Datentyp die Enumeration ist, zum Beispiel:

```
Public Property Get TestEnumProp() As TestEnum
 '...
End Property

 Public Property Let TestEnumProp(ByVal vNewValue _
  As TestEnum)
    '...
 End Property
```

Nun könnten Sie denken, daß dies eine hübsche Sache sei – eine Eigenschaft, die sich automatisch auf die Werte der Enumeration beschränkt. Doch damit lägen Sie falsch.

Enumerationen dienen lediglich zur Definition von Konstanten – das ist alles. Jeder Wert einer Enumeration ist ein 32-Bit-Long-Wert. Und somit ist die Eigenschaft `TestEnumProp` letztlich bloß als `Long`-Eigenschaft deklariert worden. Visual Basic nimmt keinerlei Bereichsprüfung vor. Bei einem Objekt `MeinObjekt`, das den obenstehenden Code enthält, würden die folgenden beiden Zeilen funktionieren:

```
MeinObjekt.TestEnumProp = Test1
MeinObjekt.TestEnumProp = 528249
```

Wofür sind Enumerationen denn dann nützlich? Sie sparen Zeit und vermindern Tippfehler. Wenn Sie in den Visual-Basic-Optionen im Register EDITOR die Option MITGLIEDER AUTOMATISCH AUFLISTEN gesetzt haben, wird automatisch eine Liste der Elemente der Enumeration angezeigt, sobald Sie `MeinObjekt.TestEnumProp =` im Code-Fenster eingetippt haben. Dann können Sie einfach ein Element auswählen, ohne lange irgendwo nachschauen zu müssen.

Enumeration verbessern also die Lesbarkeit des Codes und erleichtern ein wenig die Programmierarbeit, haben ansonsten jedoch keine Wirkung.

Sie können Enumerationen problemlos verwenden, sollten jedoch immer daran denken, den Wertebereich bei öffentlichen Eigenschaften zu prüfen, die als Enumerations-Datentyp deklariert sind.

Hier ein paar weitere Punkte, die Sie über Enumerationen wissen sollten:

- In ActiveX-Komponenten können Sie einem Enumerations-Namen einen Unterstrich voranstellen, um die Enumeration zu verbergen. ich sehe jedoch keinen praktischen Verwendungszweck dafür. Beispiel: `[_Test1] = 1`.
- Verwenden Sie eckige Klammern für die Definition von Enumerations-Elementen, die den Namensregeln für Variablen zuwiderlaufen (die etwa ungültige Zeichen oder Leerzeichen enthalten). Beispiel: `[Mein enumerierter Wert] = 5`.
- In einer Enumeration müssen die Werte nicht streng aufeinanderfolgen. Sie können so beliebige Werte in einer Enumeration zusammenfassen.
- Die Werte der Elemente einer Enumeration brauchen nicht eindeutig zu sein. Gültig ist beispielsweise: `Test1 = 1 : Test2 = 1`.
- Hüten Sie sich davor, den enumerierten Konstanten Namen zu verleihen, die bereits von anderen Komponenten oder von Visual Basic selbst verwendet werden. Im Falle eines Konflikts nimmt Visual Basic die Konstante, die in der am weitesten oben stehenden Typbibliothek der Verweis-Liste enthalten ist. Microsoft empfiehlt, jeden Enumerations-Namen mit einem eindeutigen Präfix

zu versehen. So haben beispielsweise die in Visual Basic und VBA verwendeten enumerierten Konstanten das Präfix `vb`, wie etwa in `vbArrow`.

Nun noch eine abschließende Bemerkung zur Verwendung öffentlicher Enumerationen. Wie Sie sich vorstellen können, stellt Visual Basic eine ziemlich komplexe Umgebung dar. Wenn auch deren Qualität im allgemeinen hervorragend ist, so kommt es doch hin und wieder zu seltsamen Erscheinungen im Verhalten der Umgebung. Manchmal werden Fehler angezeigt, die nur sehr schwer nachzuvollziehen sind. Mir ist es beispielsweise untergekommen, daß ein kompiliertes Projekt sporadisch abstürzte, sich nicht einwandfrei kompilieren ließ oder nicht unter binärer Kompatibilität rekompiliert werden konnte.

In den meisten Fällen konnte ich derartige Probleme beseitigen, indem ich die Verwendung von öffentlichen Enumerationen als Datentyp für Eigenschaften oder Parameter entfernte. Ich kann dieses Phänomen selbst nicht richtig nachvollziehen, geschweige denn Ihnen ein nachvollziehbares Beispiel geben. Bei den Desaware-Produkten, die in Visual Basic geschrieben sind, vermeiden wir allerdings aufgrund dieser Erfahrungen die Verwendung von öffentlichen Enumerationen als Datentyp für Parameter oder Eigenschaften. Ich will jetzt nicht so weit gehen, Ihnen zu raten, unserem Beispiel zu folgen. Jedoch sollten Sie vor der Freigabe eines neuen Produkts versuchen, das Projekt unter Binär-Kompatibilität neu zu kompilieren, um zu prüfen, ob die neue Version korrekt erstellt wird. Die genannten Probleme tauchten allerdings nur unter Visual Basic 5 auf. In Visual Basic 6 sind sie mir noch nicht begegnet. Doch da sie in Hunderten von Programmierstunden nur hin und wieder sporadisch auftraten, könnte es für eine Aussage noch zu früh sein, ob die Probleme in Visual Basic 6 weiterhin vorhanden sind. Ich hoffe, daß sie mittlerweile doch erkannt und beseitigt worden sind.

10.4.2 Assistenten verwenden – ja oder nein?

Visual Basic enthält eine Reihe von Assistenten, darunter auch einen Klassengenerator. Ob Sie solche Assistenten verwenden, ist eine Frage des persönlichen Geschmacks. Ich habe festgestellt, daß der Klassengenerator einen guten Überblick über Methoden, Eigenschaften und Ereignisse eines Objekts bietet, im allgemeinen jedoch mehr Nacharbeit erfordert, als er Zeit spart. Klassen-Code ist nicht so allgemeingültig, als daß der von dem Klassengenerator erzeugte Code auch immer wirklich Ihren Vorstellungen entspricht. Ich erstelle Klassen lieber von Hand, anstatt hinter dem Generator aufräumen zu müssen. Doch lassen Sie mich betonen, daß das meine persönliche Ansicht ist und keinen Rat meinerseits darstellen soll. Versuchen Sie es mit oder ohne Klassengenerator und schauen Sie, wie Sie am besten zurechtkommen.

Der ActiveX-Schnittstellen-Assistent, der bei der Erstellung von ActiveX-Controls helfen soll, ist weitaus nützlicher. Er hilft vor allem bei der Implementierung von Standard-Eigenschaften und zeigt auch, wie einige übliche Techniken implementiert werden. Sie werden mehr über ihn in Teil III, »ActiveX-Controls«, erfahren.