

# Teil C

## Der Umstieg auf Visual Basic 2005 ...

---

<b>143</b>	<b>Der Umstieg von Visual Basic 6.0</b>
<b>185</b>	<b>Der Umstieg von Visual Basic.NET 2002 oder 2003</b>

---

Mit einiger Überraschung musste ich im Verlauf der letzten Monate feststellen, dass auch große Firmen mit einigermaßen großen Entwicklungsabteilungen tatsächlich auf das Erscheinen von Visual Basic 2005 gewartet haben, um den »Transit« in die .NET-Welt zu vollziehen. Als ich das Umsteigerkapitel zum Vorläufer dieses Buches schrieb, war ich der Meinung, ein Kapitel mit nicht so langem »Haltbarkeitsdatum« zu kreieren. Ich hatte Recht, aber anders, als ich es dachte.

Beide Kapitel in diesem Buchteil sind nämlich weitgehend neu. Sie konzentrieren sich mit kleinen Beispielen auf das Wesentliche, was Sie als Allererstes wissen müssen, um nicht auf Überraschungen bei Ihren ersten Projekten mit VB2005 zu stoßen. Und sie bauen aufeinander auf, was nicht zuletzt der Grund dafür war, das VB6-Umsteigerkapitel nicht 1:1 aus dem Vorgängerwerk dieses Titels zu verwenden.

Kommen Sie also aus der VB6-Welt, dann lesen Sie am besten beide Kapitel. Haben Sie sich jedoch zuvor schon in Visual Basic .NET (Versionen 2002 oder 2003) eingearbeitet, können Sie das nächste Kapitel vielleicht überblättern und direkt einen Blick in das übernächste werfen – wobei ein Blick in das erste Kapitel dieses Teils mit Sicherheit auch nicht schaden kann, um das eine oder andere aufzufrischen oder sogar was Neues zu entdecken.



# 5 Der Umstieg von Visual Basic 6.0

---

144	<b>Unterschiede in der Variablenbehandlung</b>
158	<b>Alles ist ein Objekt oder »let Set be«</b>
159	<b>Direktdeklaration von Variablen in For-Schleifen</b>
161	<b>Gültigkeitsbereiche von Variablen</b>
165	<b>Die Operatoren += und -= und ihre Verwandten</b>
167	<b>Fehlerbehandlung</b>
174	<b>Kurzschlussauswertungen mit OrElse und AndAlso</b>
175	<b>Variablen und Argumente auch an Subs in Klammern!</b>
176	<b>Namespaces und Assemblies</b>

---

Um es vorweg zu nehmen: Visual Basic 2005 verfügt wie Visual Basic 2003 über einen Upgrade-Assistenten, der aus Ihrem VB6-Projekt ein VB2005-Projekt machen und die dafür erforderlichen Umbauten vornehmen soll. Für kleinere Projekte oder Algorithmen, die projektunabhängig formuliert wurden, ist dieser Upgrade-Assistent sicherlich sinnvoll.

Doch eines möchte ich ohne Umschweife sagen: Wenn Sie planen, eine größere Entwicklung, die Sie ursprünglich unter VB6 vorgenommen haben, auf .NET 2.0 zu portieren, tun Sie sich, wenn es Budget und Zeit erlauben, selber den Gefallen, und überlegen Sie sich eine Strategie, komplett von vorn zu beginnen, und lassen Sie den Upgrade-Wizard einfach außen vor. Er ist sicherlich mit Überlegung entwickelt und für das eine oder andere sinnvoll einsetzbar. Doch kann er leider keine Entwicklungskonzepte anpassen, und so wird er eine in VB6 nach prozeduralen Vorgehensweisen entwickelte Lösung bestimmt nicht in ein objektorientiertes Modell umwandeln können. Doch genau das sollte Ihr Ziel bei einer Migration gerade bei größeren Softwareentwicklungen sein.

Sollte das aus zeit- und geldabhängigen Gründen vorläufig unmöglich erscheinen, überlegen Sie sich am besten eine Strategie, wenigstens einzelne Module nach und nach zu ersetzen, beide Welten für eine Weile nebeneinander existieren zu lassen und diese neuen Module später zu einem vollständig migrierten Gesamtpaket zu verschnüren.

Und über etwas Weiteres sollten Sie sich im Klaren sein: Visual Basic 2005 hat natürlich eine gewisse Ähnlichkeit zu Visual Basic 6.0. Eine gar nicht so kleine sogar. Doch selbst bei den eigentlichen Sprachelementen gibt es Unterschiede, und gemessen an der Tatsache, dass Ihnen mit dem .NET-Framework 2.0 rund 8.000 Klassen zur Lösung der unterschiedlichsten Aufgaben zur Verfügung

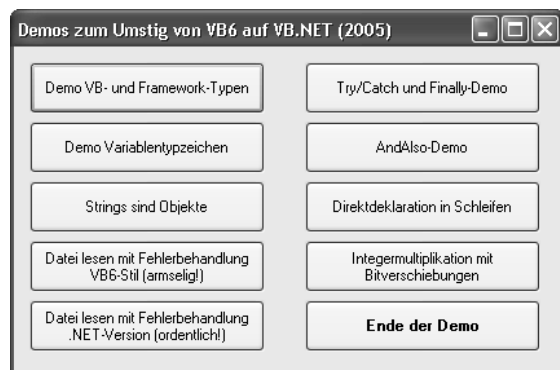
stehen, möchte ich VB2005 nicht nur als einfache Weiterentwicklung von VB6 bezeichnen. Wenn Sie mit der Vorstellung an das Erlernen von VB2005 herangehen, dass Sie im Grunde genommen eine komplett neue Sprache lernen, kommt das 1. nicht nur der Realität recht nahe, sondern Sie werden 2. auch schnelle Erfolgserlebnisse haben, da Sie mit Ihrem VB6-Können zumindest einen – na ja sagen wir – mittelgroßen Vorsprung genießen.

Die wichtigsten Unterschiede in diesem Bereich, also dort, wo es scheinbar eine Schnittmenge zwischen VB6 und VB2005 gibt, die aber dann doch gar keine ist, soll dieses Kapitel klären. Nicht mehr und nicht weniger. Die eigentlichen Neuerungen werden Sie als VB6-Programmierer dann im nächsten Buchteil erfahren.

---

**BEGLEITDATEIEN:** Übrigens: Die abgedruckten Codeausschnitte dieses Buchs vereinigt ein Projekt, das Sie im Pfad `.\VB 2005 - Entwicklerbuch\C - Ein- und Umstieg\VonVb6Zu2005` unter dem Projektmappennamen `VonVb6Zu2005.sln` finden. Es besteht aus einem Formular und mehreren Schaltflächen, die die jeweiligen Beispiele laufen lassen. Ausgaben werden dabei im Ausgabefenster angezeigt. Sollte dieses Fenster während des Programmablaufs nicht sichtbar sein, lassen Sie es einfach mit **Strg+Alt+O** anzeigen. Kleinere Codeschnipsel, VB6-Code oder Code, der zu Demonstrationszwecken mit Absicht Fehler enthält, sind dort natürlich nicht berücksichtigt.

---



**Abbildung 5.1:** Mit diesem Programm können Sie die Demos in diesem Kapitel ausprobieren und nachvollziehen

## Unterschiede in der Variablenbehandlung

Los geht es mit den Elementen des Entwickelns, die Sie am häufigsten benötigen, den primitiven Variablen und Arrays. »Primitive Variablen« meint dabei übrigens die »eingebauten und fest verdrahteten« Variablentypen in Visual Basic, wie Integer, Double, String, Boolean etc. Und schon hier gibt es eine nicht unerhebliche Anzahl an Unterschieden.

### Veränderungen bei primitiven Integer-Variablen – die Größen von Integer und Long und der neue Typ Short

Mal davon abgesehen, dass es einige Variablentypen in VB2005 nicht mehr gibt und viele neue hinzugekommen sind, haben sich einige Variablentypen auch verändert.

## Integer- und Short-Variablen

Integer-Variablen verfügen nunmehr über einen 32-Bit-breiten vorzeichenbehafteten Darstellungsbereich, anders als in VB6, wo sie mit 16-Bit auskommen mussten.

Damit stellen sie in VB2005 den Bereich von -2.147.483.648 bis 2.147.483.647 dar, im Gegensatz zu VB6, bei dem sich der Bereich nur von -32.768 bis 32.767 erstreckte. 16-Bit vorzeichenbehaftete Datentypen werden in VB2005 durch den Datentyp Short dargestellt – Short in VB2005 entspricht also Integer in VB6.

## Long-Variablen

Das was Long-Variablen in VB6 waren sind sie jetzt in Form von Integer in VB2005. Long gibt es aber auch in VB2005, nur basiert er hier auf 64 und nicht auf 32 Bit. Sein Darstellungsbereich erweitert sich also auf -9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807. Oder im Klartext gesprochen von rund minus 9,2 Trillionen auf plus 9,2 Trillionen.

## Anekdoten aus der Praxis

Ein Schulungsteilnehmer fragte mich in diesem Zusammenhang, wo denn bitteschön in der Praxis mit so hohen Werten hantiert werde. Ich musste diese Frage erst gar nicht beantworten, denn ein Kollege reagierte prompt und fragte ihn: »Wie viele Bytes hat denn deine 300 GByte-Festplatte, die du dir gestern neu bestellt hast?« Er antwortete triumphierend: »300 Millionen, und da komme ich doch locker mit 32-Bit hin.« Sein Kollege entgegnete ihm: »300 Millionen, wow, da kannst du ja dann eine halbe CD drauf speichern!«

Der gute Mann hatte kurzerhand drei Zehnerpotenzen unterschlagen und hätte die Kapazität seiner Festplatte in Bytes natürlich nicht mit einem 32-Bit-Wert darstellen können.

## Typen, die es nicht mehr gibt ...

Von einigen Variablentypen hat sich VB2005 (wie VB.NET 2002, 2003 auch schon) trennen müssen – in erster Linie übrigens, um mit dem so genannten CTS – dem Common Type System – das unter anderem den Standard aller primitiven Datentypen aller auf dem Framework basierenden Programmiersprachen regelt, kompatibel zu bleiben.

## Good bye Currency, welcome Decimal!

Den Datentypen Currency, den es in VB6 zur Verarbeitung von Währungsdaten, sprich: Geldbeträgen gab, gibt es nicht mehr. Stattdessen verwenden Sie Decimal. Im Gegensatz zu Double oder Single gibt es bei Decimal keine zahlenkonvertierungsbedingten Rundungsfehler. Für das Berechnen von Geldbeträgen ist das natürlich eine sinnvolle Voraussetzung. Decimal wird andererseits komplett »manuell« durch den Prozessor berechnet, und nicht durch dessen Mathe-Logik. Damit ist Decimal gleichzeitig auch der langsamste numerische Datentyp.

## And good bye Variant – hello Object!

Für Variant gilt das Gleiche – es gibt diesen Typ im .NET-Framework nicht mehr, na ja, sagen wir, nicht mehr an der Oberfläche.<sup>1</sup> Aber auch das ist keine Schikane der Framework-Entwickler, damit Sie Ihre Programme, die Sie bisher in V6.0 entwickelt hatten, auch wirklich gründlich umschreiben müssen – dieser Datentyp entspricht in seiner damaligen Funktionsweise einfach nicht dem Standard in Sachen Typsicherheit. In vielen Fällen können Sie aber Object verwenden, um den eigentlichen Datentyp zu kapseln. Doch zu diesem Thema erfahren Sie im Teil »OOP« mehr.

---

**HINWEIS:** Falls Sie aus Gewohnheit versucht sein sollten, einen Variant-Datentyp im Codeeditor zu deklarieren, wandelt der Editor Variant automatisch in Object um.

---

## ... und die primitiven Typen, die es jetzt gibt

Und das sind eine ganze Menge. Die folgende Tabelle zeigt Ihnen eine kurze Übersicht – neu hinzugekommene sind in Fettschrift dargestellt.

Detailliertes zu diesem Thema gibt's in Teil D dieses Buches.

Typname	.NET-Typname	Aufgabe	Wertebereich
<b>Byte</b>	System.Byte	Speichert vorzeichenlose Integer-Werte mit 8 Bit Breite	0 bis 255
<b>SByte</b>	System.SByte	Speichert vorzeichenbehaftete Integer-Werte mit 8 Bit Breite	-127 bis 128
<b>Short</b>	System.Int16	Speichert vorzeichenbehaftete Integer-Werte mit 16 Bit (2 Byte) Breite	-32.768 bis 32.767
<b>UShort</b>	System.UInt16	Speichert vorzeichenlose Integer-Werte mit 16 Bit (2 Byte) Breite	0 bis 65.535
Integer	System.Int32	Speichert vorzeichenbehaftete Integer-Werte mit 32 Bit (4 Byte) Breite <b>HINWEIS:</b> Auf 32-Bit-Systemen ist dies der am schnellsten verarbeitete Integer-Datentyp	-2.147.483.648 bis 2.147.483.647
<b>UInteger</b>	System.UInt32	Speichert vorzeichenlose Integer-Werte mit 32 Bit (4 Byte) Breite	0 bis 4.294.967.295

---

<sup>1</sup> Wobei das nicht ganz richtig ist, denn es gibt ihn Framework-intern noch (für diejenigen, die es genau wissen wollen: System.Variant befindet sich in der *mscorlib*, also der Common Language Runtime-Bibliothek), Sie können ihn nur nicht mehr verwenden. Variant wird intern aus Kompatibilitätsgründen zur Zusammenarbeit mit COM-Objekten verwendet. Kleine Randnotiz: In einer der ersten Betas von Visual Basic 2002 konnten Sie ihn sogar noch verwenden; da Object seine Funktionalität jedoch weitestgehend übernehmen kann, beschlossen die Entwickler des Frameworks, ihn für die Außenwelt unzugänglich zu machen, weil sie eine zu große Konfusion bei den Entwicklern zwischen Variant und Object befürchteten.

Typename	.NET-Typname	Aufgabe	Wertebereich
Long	System.Int64	Speichert vorzeichenbehaftete Integer-Werte mit 64 Bit (8 Byte) Breite	-9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807
ULong	System.UInt64	Speichert vorzeichenlose Integer-Werte mit 64 Bit (8 Byte) Breite	0 bis 18.446.744.073.709.551.615
Single	System.Single	Speichert Fließkommazahlen mit einfacher Genauigkeit. Benötigt 4 Bytes zur Darstellung.	-3,4028235E+38 bis -1,401298E-45 für negative Werte; 1,401298E-45 bis 3,4028235E+38 für positive Werte
Double	System.Double	Speichert Fließkommazahlen mit doppelter Genauigkeit. Benötigt 8 Bytes zur Darstellung. <b>HINWEIS:</b> Dies ist der schnellste Datentyp zur Fließkommazahlenberechnung, da er direkt an die Mathe-Einheit des Prozessors zur Berechnung delegiert wird.	1,79769313486231570E+308 bis -4,94065645841246544E-324 für negative Werte; 4,94065645841246544E-324 bis 1,79769313486231570E+308 für positive Werte
Decimal	System.Decimal	Speichert Fließkommazahlen im binärcodierten Dezimalformat. <b>HINWEIS:</b> Dies ist der langsamste Datentyp zur Fließkommazahlenberechnung, seine besondere Speicherform schließt aber typische Computerrundungsfehler aus.	0 bis +/-79.228.162.514.264.337.593.543.950.335 (+/-7,9...E+28) ohne Dezimalzeichen; 0 bis +/-7,9228162514264337593543950335 mit 28 Stellen rechts vom Dezimalzeichen; kleinste Zahl ungleich 0 (null) ist +/-0,0000000000000000000000000000001 (+/-1E-28)
Boolean	System.Boolean	Speichert boolesche Zustände.	True oder False
Char	System.Char	Speichert ein Unicode-Zeichen mit einem Speicherbedarf von 2 Byte.	Ein Unicodezeichen im Bereich von 0-65535
Date	System.DateTime	Speichert ein Datumswert, bestehend aus Datum und Zeitanteil	0:00 Uhr am 01.01.0001 bis 23:59:59 Uhr am 31.12.9999
Object	System.Object	Speichert die Referenz auf Daten bestimmten Typs im Managed Heap	Belegt entweder 32-Bit (4 Bytes) auf 32-Bit-Betriebssystemen oder 64 Bit auf 64-Bit-Betriebssystemen und dient – wie jede andere Objektvariable (Referenztyp) – als Zeiger auf die eigentlichen Objektdaten im Managed Heap
String	System.String	Speichert Unicode-Zeichenfolgen	Variable Länge. 0 bis ca. 2 Milliarden Unicode-Zeichen

**Tabelle 5.1:** Die primitiven Datentypen in .NET 2.0 bzw. VB2005

**HINWEIS:** Übrigens, eigentlich kann man die Regel aufstellen, dass jeder Typ, den der Codeeditor blau einfärbt, ein primitiver Datentyp ist. Das funktioniert natürlich nur so lange, wie Sie die Visual Basic-Äquivalente der primitiven .NET-Datentypen verwenden. Sie müssen das aber nicht, wie die in Fettschrift gesetzten Zeilen des folgenden Listings zeigen. Der folgende Code würde nämlich durchaus funktionieren:

```

Public Class Form1
    Private Sub btnVbUndFrameworkTypen_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles btnVbUndFrameworkTypen.Click
        'Ein im VB-deklarierte Datentyp unterscheidet...
        Dim locDatum1 As Date
        '...sich nicht von seiner Framework-Version!
        Dim locDatum2 As System.DateTime

        'Das ging in VB6 übrigens auch nicht!
        locDatum1 = #12/24/2005 6:30:00 PM#

        'Hier ist der Beweis: Keiner meckert.
        locDatum2 = locDatum1

        'Übrigens: {0} und {1} werden durch die darauffolgenden
        'Variableninhalte ersetzt. Hinter dem Doppelpunkt folgt
        'jeweils die Formatierungszeichenfolge.
        Debug.Print("Dieses Jahr am {0:dd.MM.yy}, also Heiligabend essen wir um {1:HH:mm}", _
            locDatum2.Date, _
            locDatum2.TimeOfDay)
    End Sub

```

## Deklaration von Variablen und Variablentypzeichen

Anders als in VB6 können Variablen gleichen Typs in einem Rutsch deklariert werden.

Darüber hinaus gibt es in VB2005, wie in VB6, Variablentypzeichen. Die kennen Sie, wenn Sie, wie beispielsweise ich, seit Jahrzehnten Basic programmieren, von der ersten Stunde an. Solche Zeichen definieren, welchen Typs eine Variable sein soll, wenn Sie die Typanweisung (beispielsweise As Integer) nicht mit ausformulieren.

Betrachten Sie das folgende Codebeispiel, das diese Zusammenhänge genauer verdeutlichen soll:

```

Public Class Form1

    Private Sub btnVariablentypzeichen_Click(ByVal sender As System.Object, _
        ByVal eArgs As System.EventArgs) Handles btnVariablentypzeichen.Click

        'Beide sind Integer - anders als in VB6!
        Dim locInt1, locInt2 As Integer

        'Auch das geht...
        Dim locDate1, locDate2 As Date, locLong1, locLong2 As Long

        'Und das hier auch:
        Dim a%, b&, c@, d!, e#, f$
        f$ = "Muss was drin sein!"

        'Alle Variablentypen in Klartext ausgeben:
        Debug.Print("locInt1 ist: " & locInt1.GetType.ToString)
        Debug.Print("locInt2 ist: " & locInt2.GetType.ToString)
    End Sub

```

```

    Debug.Print("locDate1 ist: " & locDate1.GetType.ToString)
    Debug.Print("locDate2 ist: " & locDate2.GetType.ToString)
    Debug.Print("locLong1 ist: " & locLong1.GetType.ToString)
    Debug.Print("locLong2 ist: " & locLong2.GetType.ToString)

    Debug.Print("a ist: " & a.GetType.ToString)
    Debug.Print("b ist: " & b.GetType.ToString)
    Debug.Print("c ist: " & c.GetType.ToString)
    Debug.Print("d ist: " & d.GetType.ToString)
    Debug.Print("e ist: " & e.GetType.ToString)
    Debug.Print("f ist: " & f.GetType.ToString)
End Sub
End Class

```

Wenn Sie dieses Beispiel laufen lassen, wird folgende Ausgabe im Ausgabefenster angezeigt:

```

locInt1 ist: System.Int32
locInt2 ist: System.Int32
locDate1 ist: System.DateTime
locDate2 ist: System.DateTime
locLong1 ist: System.Int64
locLong2 ist: System.Int64
a ist: System.Int32
b ist: System.Int64
c ist: System.Decimal
d ist: System.Single
e ist: System.Double
f ist: System.String

```

Im Grunde genommen verdeutlicht dieses Beispiel dreierlei:

- Zum Ersten sehen Sie abermals, dass die .NET-Datentypen und die Visual Basic-Datentypen absolut dasselbe sind.
- Zweitens sehen Sie, dass – anders als in VB6 – das gleichzeitige Deklarieren von Variablen vom selben Typ möglich ist. In VB wäre `locInt1` automatisch vom vorgegebenen Standarddatentyp oder – falls dieser mit `DefXXX` nicht festgelegt worden wäre – vom Typ `Variant` gewesen.

---

**HINWEIS:** Standarddatentypendefinitionen mit `DefXXX`, die beim Weglassen des Typqualifizierers bei einer Variablen-deklaration automatisch den zu verwendenden Datentyp festgelegt haben (also beispielsweise `DefInt A-Z`, um alle Variablen, die nicht explizit als bestimmter Typ ausgewiesen wurden, automatisch als Integer zu deklarieren), gibt es in keinem der Visual Basic.NET- Derivate mehr.

---

- Und drittens: Variablentypzeichen sind optional. Sie können sie zwar verwenden, aber Sie sollten sie nicht verwenden. Dabei spielt es im Übrigen keine Rolle, auf welche Weise eine Variablen zu »ihrem Typ geworden ist« – nur durch das Typzeichen oder durch die ausformulierte Angabe des Typs mit `as VarType`.

**TIPP:** Eines ist aber klar: Ihr Code ist in jedem Fall einfacher lesbar, wenn Sie die Variablentypen explizit angeben. Typzeichen sind meiner Meinung nach Relikte aus alter Zeit und eigentlich überflüssig. Im Übrigen gibt es sie bis auf eine kleine Ausnahme<sup>2</sup>, in keiner anderen »Erwachsenenprogrammiersprache« (wie C, C++, C# oder Java), und wir wollen ja schließlich alle, das Basic endlich gemeinhin bescheinigt werden kann, bei den Großen mitspielen zu dürfen. Und wenn Sie sich in einer umfangreichen Prozedur einmal nicht daran erinnern können, von welchem Typ eine bestimmte Variable war, fahren Sie einfach mit dem Mauszeiger im Editor darauf. Der gibt Ihnen mit einem Tooltip darüber sofort Auskunft.

```
'Und das hier auch:
Dim a%, b&, c@, d!, e#, f$
f$ = "Muss Dim c As Decimal sein!"
```

**Abbildung 5.2:** Ein simples Darauffahren mit dem Mauszeiger zur Entwurfszeit genügt, um den Typ einer Variablen zu erfahren

## Typsicherheit und Typliterale zur Typdefinition von Konstanten

Typliterale – meiner Meinung nach sollten diese »Typerzwingungsliterale für Konstanten« heißen – dienen dazu, eine Konstante dazu zu zwingen, ein bestimmter Typ zu sein.

Sie kennen das auch von Visual Basic 6.0. Wenn Sie eine numerische Konstante dazu zwingen wollen, eine Zeichenkette zu sein, damit Sie diese ohne sie umwandeln zu müssen an eine String-Variable zuweisen können, dann hüllen Sie sie in Anführungszeichen ein. Das Anführungszeichen (oder *die* Anführungszeichen in diesem Fall) dienen also dazu, aus einer Zahl eine Zeichenkette zu machen.

In Visual Basic .NET (also in allen Versionen seit VB2002) beschränken sich Typliterale nicht nur auf Strings – es gibt sie auch für andere Datentypen. Eines haben Sie bereits im ersten Listing dieses Kapitels kennen gelernt – das Typliteral für Datumswerte »#«. Neben dem Anführungszeichen bei Strings, ist dies das einzige weitere, das eine Konstante quasi einklammert.

Andere Typliterale werden der Konstante einfach nachgestellt. In einigen Fällen bestehen diese übrigens nicht nur aus einem Buchstaben sondern aus zweien.

Die folgende Tabelle zeigt, wie Sie Konstanten mit Typliteralen definieren und gibt Beispiele. Sollte es Variablentypzeichen für einen bestimmten Typ der Tabelle geben, finden Sie diese ebenfalls in der Tabelle angegeben.

Typname	Typzeichen	Typliteral	Beispiel
Byte	–	–	Dim var As Byte = 128
SByte	–	–	Dim var As SByte = -5
Short	–	S	Dim var As Short = -32700S
UShort	–	US	Dim var As UShort = 65000US ▶

<sup>2</sup> In C# 2005 gibt es die Möglichkeit, einen primitiven Datentyp auf sein nullbares Pendant durch ein Typzeichen festzulegen.

Typname	Typzeichen	Typliteral	Beispiel
Integer	%	I	Dim var% = -123I oder Dim var As Integer = -123I
<b>UInteger</b>	-	UI	Dim var As UInteger = 123UI
Long	&	L	Dim var& = -123123L oder Dim var As Long = -123123L
<b>ULong</b>	-	UL	Dim var As ULong = 123123UL
Single	!	F	Dim var! = 123.4F oder Dim var As Single = 123.4F
Double	#	R	Dim var# = 123.456789R oder Dim var As Double = 123.456789R
<b>Decimal</b>	@	D	Dim var@ = 123.456789123D oder Dim var As Decimal = 123.456789123D
Boolean	-	-	Dim var As Boolean = True
<b>Char</b>	-	C	Dim var As Char = "A"C
Date	-	#dd/MM/yyyy HH:mm:ss# oder #dd/mm/yyyy hh:mm:ss am/pm#	Dim var As Date = #12/24/2005 04:30:15 PM#
<b>Object</b>	-	-	In einer Variable vom Typ Object kann jeder beliebige Typ gekapselt („geboxed“) werden oder mit dieser referenziert werden.
String	\$	"Zeichenkette"	Dim var\$ = "Zeichenkette" oder Dim var As String = "Zeichenkette"

**Tabelle 5.2:** Typliterale und Variablentypzeichen der primitiven Datentypen in Visual Basic 2005

Vertiefen wir das in der Tabelle Gezeigte ein wenig, damit deutlicher wird, worum es geht.

Betrachten Sie sich den folgenden Visual Basic 6.0-Code:

**Achtung! VB6-Code!**

```
Dim einString As String
einString = "1.23"

Dim einAndererString As String
einAndererString = 1.23

Debug.Print (einString & ", " & einAndererString)
```

Wenn Sie diesen kleinen Codeausschnitt laufen ließen, würde er folgendes Ergebnis produzieren:

1.23, 1,23

Warum, ist jedem erfahrenen VB6-Programmierer klar:

einString hat seinen Wert durch eine direkte Zuweisung einer Konstanten bekommen, die als sein ureigener Typ zu erkennen war: Die Anführungszeichen sorgen dafür, dass die Konstante als Zei-

chenkette behandelt wird, und eben da es sich um Typengleichheit handelt, enthält die Stringvariable exakt den angegebenen Typ.

Die Konstante, die an einAndererString zugewiesen wird, ist nicht vom Typ String, denn sie ist nicht in Anführungszeichen eingefasst. Bei ihr handelt es sich um eine Fließkommakonstante. Also muss zunächst eine implizite Typkonvertierung bei der Zuweisung stattfinden – die Fließkommazahl wird in eine Zeichenkette umgewandelt. Und da die meisten von uns wohl auf einem deutschen Betriebssystem arbeiten, wird die deutsche Kultur bei der Umwandlung berücksichtigt: Anders als die Amerikaner trennen wir Nachkomma- von Vorkommastellen mit einem Komma und nicht mit einem Punkt. So wird aus der Konstante 1.23 die Zeichenkette 1,23.

Sie sehen also, dass die Bestimmung von Typen bei Konstanten schon im alten Visual Basic durchaus wichtig sein konnte.

## .NET ist Typsicher

In .NET ist das noch viel, viel wichtiger, denn .NET ist grundsätzlich typsicher. Typsicher bedeutet, dass Sie unterschiedliche Typen bei Zuweisungen nicht »einfach so« mischen und damit die folgende Zeile eigentlich gar nicht kompiliert sondern mit einer Fehlermeldung quittiert bekämen:

```
Dim einAndererString As String
einAndererString = 1.23
```

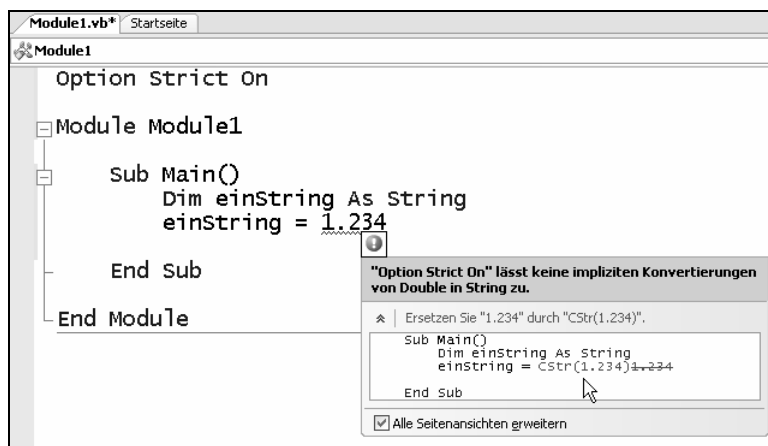


Abbildung 5.3: Typsicherheit unter .NET verlangt, dass implizit nur gleiche oder sichere Typen einander zugewiesen werden können

Die Autokorrektur für intelligentes Kompilieren zeigt Ihnen, was das Problem ist: Implizit, also ohne weiteres Zutun (oder anders gesagt »von selbst«) können Sie unter .NET im Grunde genommen nur gleiche Typen zuweisen, oder Typen die zwar unterschiedlich sind, bei denen eine implizite Konvertierung also auf jeden Fall sicher ist.

Und was heißt »auf jeden Fall sicher«?

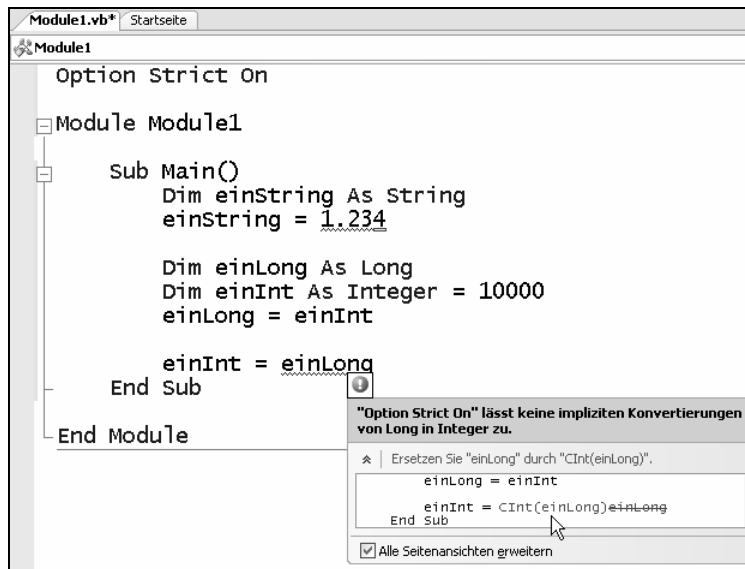
Bei den Strings haben wir es bereits schon im oben stehenden VB6-Beispiel erlebt. Diese implizite Konvertierung war nicht sicher. Auf einem amerikanischen System hätte man ein anderes Resultat als auf einem deutschen System gehabt. Die implizite Konvertierung ist also nicht sicher, weil es offensichtlich Einflussgrößen gibt (hier: die Kultureinstellungen), die das Ergebnis beeinflussen können.

Anders ist es zum Beispiel, wenn Sie einen Integer-Typ einem Long-Typ zuweisen:

```
Dim einLong As Long
Dim einInt As Integer = 10000
einLong = einInt
```

Kein .NET-Compiler würde hier was zu meckern haben, denn bei dieser Typkonvertierung kann nichts schief gehen. Integer deckt in jedem Fall einen viel kleineren Zahlenbereich als Long ab, und deswegen ist hier eine implizite Konvertierung gefahrlos möglich.

Andersherum sieht es schon wieder anders aus, wie die folgende Abbildung zeigt:



**Abbildung 5.4:** Während »kleinere« Typen sicher in »größere« konvertiert werden können, ist das umgekehrt nicht typsicher und deswegen auch nicht implizit gestattet

Bei einer Konvertierung von Long zu Integer können Informationen verloren gehen, deswegen stuft .NET diese Art der Konvertierung als nicht typsicher ein. Natürlich können Sie eine derartige Konvertierung vornehmen, nur eben nicht implizit. Sie müssen .NET explizit mitteilen, dass Sie sich sozusagen »der Gefahr bewusst sind«, und entsprechende Aktionen vornehmen, um die Konvertierung vornehmen zu können. Die Autokorrektur für intelligentes Kompilieren macht Ihnen auch direkt einen entsprechenden Vorschlag: »Setzen Sie CInt (etwa: *Convert to Integer – in Integer konvertieren*) ein«, schlägt sie vor, »und zeigen Sie dem Compiler damit, dass Sie sich der Konvertierung von Long in Integer bewusst sind.«

Und vielleicht ahnen Sie jetzt auch schon, wozu die Typliterale in VB.NET dienen. Typsicherheit muss auch für Konstanten gelten. Und damit muss es auch einen Weg geben, eine Konstante dazu zu bringen, ein bestimmter Typ zu sein. Und eben genau das geschieht mit Typliteralen. Funktioniert das folgende implizit?

```
Dim einChar As Char
Dim einString As String = "Hallo"
einChar = einString
```

Nein. Denn dabei würde »a11o« auf der Strecke bleiben. Ein Char-Typ kann nur ein einzelnes Unicode-Zeichen und keinen ganzen String aufnehmen. Auch wenn dieser Codeausschnitt folgendermaßen lauten würde:

```
Dim einChar As Char = "H"
```

String ist String, und Char ist Char. Nur in Anführungszeichen definieren Sie eben einen String, egal wie viele Zeichen der hat. Und selbst wenn es passen würde (so wie in dieser Zeile), wäre die Typsicherheit dennoch verletzt. Sie müssen aus dem »H« ein Char-Typ machen, und das erreichen Sie durch das Hintenanstellen eines Typliterals, wie die folgende Abbildung zeigt:

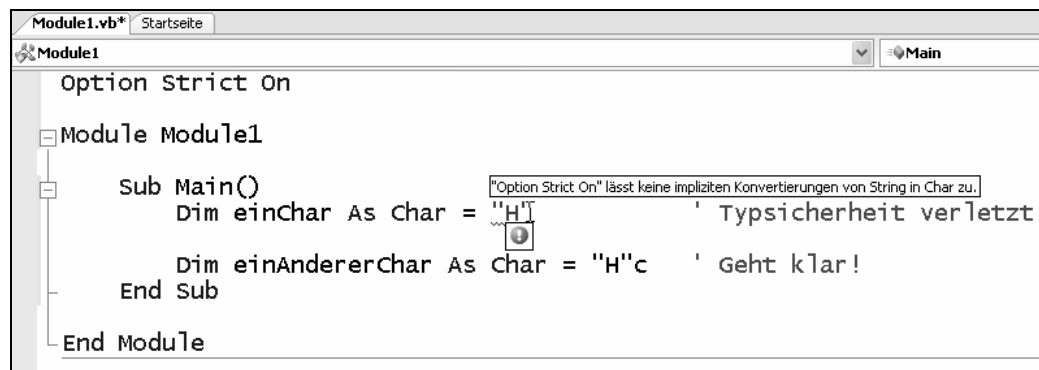


Abbildung 5.5: Mit Typliteralen zwingen Sie Konstanten in einen bestimmten Typ

Das gilt natürlich nicht nur für String und Char, sondern auch für die unterschiedlichen numerischen Datentypen.

Der folgende Codeausschnitt zeigt ein paar Beispiele, die demonstrieren, wann der Einsatz von Typliteralen Sinn macht:

```
'Fehler: Von Double nach Decimal geht nicht implizit.
Dim decimal1 As Decimal = 1.2312312312312
'Hier ist es ein Decimal durch das D am Ende
Dim decimal2 As Decimal = 1.2312312312312D

Dim decimal3 As Decimal = 9223372036854775807 ' OK.
' Überlauf - Ohne Typliteral ist es implizit ein
' Long-Wert, und in diesem Fall liegt er außerhalb seines Wertebereichs.
Dim decimal4 As Decimal = 9223372036854775808
' Mit dem Typliteral "D" wird Decimal als Konstante erzwungen, und es passt.
Dim decimal5 As Decimal = 9223372036854775808D ' Kein Überlauf.

'Fehler: Ohne Typliteral ist's wieder implizit ein Long,
'aber außerhalb des Long-Wertebereichs.
Dim ushort1 As ULong = 9223372036854775808

' Mit Typliteral wird Decimal als Konstante erzwungen, und es passt.
Dim ushort2 As Decimal = 9223372036854775808UL ' Kein Überlauf mehr.
```

## Deklaration und Definition von Variablen »in einem Rutsch«

Vielleicht haben Sie es beim Betrachten der vorherigen Beispiele schon gemerkt: Anders als in VB6 können Variablen in VB2005 (und auch in den Vorgängerversionen) in einer Zeile nicht nur deklariert sondern auch definiert werden.

Ob Sie also schreiben

```
Dim einInteger As Integer  
einInteger = 10
```

oder

```
Dim einInteger As Integer = 10
```

bleibt sich absolut gleich.

Das gilt in VB2005 auch für Instanzierungen<sup>3</sup> von Objekten, wie das folgende Beispiel zeigt.

```
Dim einObject As Object  
Dim einObject = New Object
```

Bleibt sich gleich mit

```
Dim einObject As New Object
```

---

**WICHTIG:** In VB6 gab es einen großen Unterschied zwischen den beiden Versionen der Objektinstanzierung, auf den der nächste Abschnitt eingeht.

---

## Vorsicht: New und New können zweierlei in VB6 und VB.NET sein!

Grundsätzlich gibt es zwei Möglichkeiten, ein neues Objekt in Visual Basic (sowohl in 6.0 als auch in 2002, 2003 und 2005) zu erstellen. Typischer VB6-Code, der beispielsweise eine Auflistung vom Typ Collection instanziiert, könnte folgendermaßen ausschauen, und das haben viele von der Riege der alten VB6-Programmierer auch so gemacht:

### Achtung! VB6-Code!

```
Dim locObj As Collection  
Set locObj = New Collection  
Debug.Print (locObj Is Nothing)
```

```
Set locObj = Nothing  
Debug.Print (locObj Is Nothing)
```

Wenn Sie diesen Code unter VB6 laufen lassen, zeigt Ihnen das Direktfenster erwartungsgemäß Folgendes an:

```
Falsch  
Wahr
```

---

<sup>3</sup> Falls Ihnen der Begriff »Instanzierung« im Moment nichts sagt: Sie kennen diesen Vorgang beispielsweise aus VB6, wenn Sie eine Collection für die Aufnahme von anderen Variablen oder Objekten benötigten.

Denn: Eine neue Collection wird in den ersten beiden Zeilen definiert; die Abfrage auf Nothing ist damit in der ersten Ausgabe Falsch. Anschließend wird das Objekt auf Nothing zurückgesetzt, und die folgende Ausgabe gibt wahrheitsgemäß Wahr aus.

Doch schauen Sie, was passiert, wenn Sie dieses kleine Programm folgendermaßen abändern:

### **Achtung! VB6-Code!**

```
Dim locObj As New Collection
'Set locObj = New Collection
Debug.Print (locObj Is Nothing)
Set locObj = Nothing
Debug.Print (locObj Is Nothing)
```

Nun erscheint folgendes Ergebnis im Direktfenster:

```
Falsch
Falsch
```

Warum? Weil es anders als unter .NET in VB6 *einen Unterschied* macht, wie Sie eine neue Instanz einer Klasse erstellen. Deklarieren und instanzieren Sie eine Objektvariable in VB6 in einem Rutsch, wird sie niemals Nothing werden können!

---

**WICHTIG:** Der VB6-Compiler sorgt beim Deklarieren und Instanzieren von Objektvariablen in einer Zeile immer dafür, dass diese mit New (»unter der Haube« sozusagen) neu instanziiert werden, sollten sie einmal Nothing geworden sein. **Das ist in VB.NET nicht so.**

---

## **Überläufe bei Fließkommazahlen und nicht definierte Zahlenwerte**

Überläufe bei Fließkommazahlen führten bei VB6 zu Laufzeitfehlern. Zu den Überläufen zählt – auch wenn das mathematisch nicht ganz korrekt ist – in diesem Zusammenhang eine Division durch 0. Der folgende Code unter VB6 hätte also zur Laufzeit gleich zwei Fehler ausgelöst:

### **Achtung! VB6-Code!**

```
Dim einDouble As Double
einDouble = 9.7E+307
'Fehler: Überlauf!
einDouble = einDouble * 2

einDouble = 1
'Fehler: Division durch 0!
einDouble = einDouble / 0
```

VB.NET verhält sich hier anders. Wenn eine Fließkommazahl einen bestimmten Wert nicht mehr annehmen kann, dann wird ihr ein besonderer Wert zugewiesen, wie das folgende Beispiel zeigt:

```
Dim einDouble As Double
einDouble = 9.7E+307
einDouble = einDouble * 2
Debug.Print("einDouble hat den Wert:" & einDouble)
```

```
einDouble = 1
einDouble = einDouble / 0
Debug.Print("einDouble hat den Wert:" & einDouble)
```

Ließen Sie diesen Codeausschnitt laufen, würde er überhaupt keinen Fehler produzieren. Stattdessen zeigt er folgendes Ergebnis im Ausgabefenster an:

```
einDouble hat den Wert:+unendlich
einDouble hat den Wert:+unendlich
```

Dieser »Wert« tritt nicht nur unter bestimmten Umständen (Überlauf, Division durch 0) während des Programmablaufs auf. Sie können ihn auch manuell zuweisen oder ihn abfragen, wie das folgende Beispiel zeigt:

```
'Auf "unendlich" (Überlauf) prüfen
If Double.IsInfinity(einDouble) Then
    Debug.Print("einDouble ist unendlich!")
End If

'Auf "minus unendlich" (negativen Überlauf) prüfen
If Double.IsNegativeInfinity(einDouble) Then
    Debug.Print("einDouble ist minus unendlich!")
End If

'Gezielt auf "plus unendlich" (positiven Überlauf) prüfen
If Double.IsPositiveInfinity(einDouble) Then
    Debug.Print("einDouble ist plus unendlich!")
End If

'"plus unendlich" zuweisen
einDouble = Double.PositiveInfinity

'"minus unendlich" zuweisen
einDouble = Double.NegativeInfinity
```

Und noch einen Sonderfall decken die primitiven Fließkommatypen Single, Double und Decimal ab: Die Division von 0 und 0, die mathematisch nicht definiert ist und *keine gültige Zahl* ergibt:

```
'Sonderfall: 0/0 ist mathematisch nicht definiert und ergibt "Not a Number"
einDouble = 0
einDouble = einDouble / 0
If Double.IsNaN(einDouble) Then
    Debug.Print("einDouble ist keine Zahl!")
End If
```

Ließen Sie diesen Code laufen, würde der Text in der If-Abfrage ausgegeben werden.

---

**WICHTIG:** Überprüfungen auf diese Sonderwerte lassen sich nur durch die Eigenschaften testen, die (als so genannte statische<sup>4</sup> Funktionen bzw. statische Eigenschaften) direkt an den Typen »hängen«. Zwar können Sie beispielsweise mit der Konstante der Fließkommatypen NaN den Wert »keine gültige Zahl« einer Variablen zuweisen; diese Konstante eignet sich allerdings nicht, auf diesen Zustand (»Wert«) zu testen, wie das folgende Beispiel zeigt:

---

```
Dim einDouble As Double
```

```
'Sonderfall: 0/0 ist mathematisch nicht definiert und ergibt "Not a Number"  
einDouble = 0  
einDouble = einDouble / 0
```

```
'Der Text sollte erwartungsgemäß ausgegeben werden,  
'wird er aber nicht!  
If einDouble = Double.NaN Then  
    Debug.Print("Test 1:einDouble ist keine Zahl!")  
End If
```

```
'Nur so kann der Test erfolgen!  
If Double.IsNaN(einDouble) Then  
    Debug.Print("Test 2:einDouble ist keine Zahl!")  
End If
```

In diesem Beispiel würde nur der zweite Text ausgegeben.

## Alles ist ein Objekt oder »let Set be«

Wenn Sie den folgende VB.NET-Code betrachten,

```
Dim einString As String  
einString = "Ruprecht Dröge"
```

```
'Position des Leerzeichens ermitteln  
Dim spacePos As Integer = einString.IndexOf(" ")
```

```
'Nur den Vornamen ermittelt - das ging "früher" mit Mid$  
einString = einString.Substring(0, spacePos)  
Debug.Print(einString)
```

So können Sie anhand der Anweisung `einString.Substring(0, spacePos)` schon vermuten, dass es sich bei `einString` nicht um eine »bloße Grunddatentypenvariable« handelt, wie man das von VB6 kennt, sondern dass das, was dort verarbeitet wird, eigentlich eher wie ein Objekt aussieht – denn wie bei einem richtigen Objekt verfügt eine `String`-Variable offensichtlich um Methoden und Eigenschaften – etwa, wie hier zu sehen, über eine `Substring`-Methode.

Die Wahrheit ist: In .NET »ist alles ein Objekt« oder zumindest von diesem abgeleitet. Buchstäblich, denn `Object`, von dem schon die Rede war, ist der grundlegendste aller Datentypen.

---

<sup>4</sup> *Statisch* deswegen, weil Sie keine definierte Variable brauchen, um die Funktion bzw. Eigenschaft verwenden zu können, sondern Ihnen diese direkt über den Typnamen immer (statisch) zur Verfügung steht.

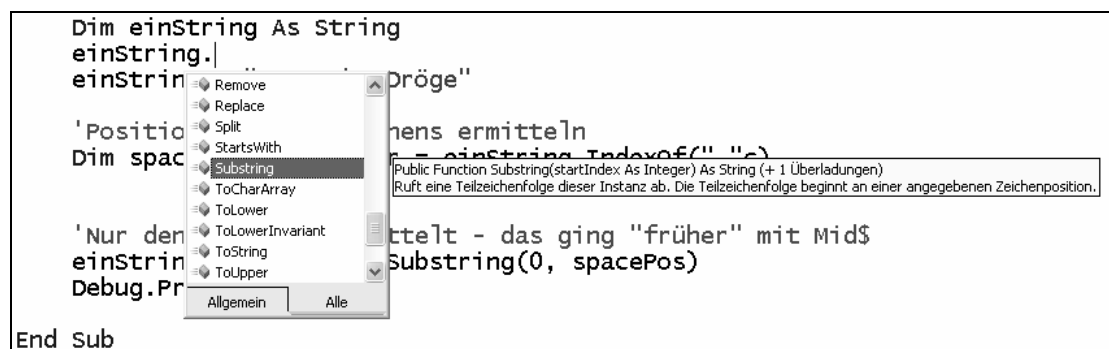
Reine Grunddatentypen wie in Visual Basic 6.0 gibt es im Framework.NET nur noch per Definition. In .NET heißen diese dann auch »primitive Datentypen«, und wie alles in .NET handelt es sich im Grunde genommen auch bei ihnen um Objekte. Und da man, hätte man das alte Konzept zur Verarbeitung von Objekten weiterverfolgt, bei der Zuweisung von Inhalten *immer* das Schlüsselwort »Set« hätte verwenden müssen, haben die VB.NET-Entwickler es schlicht hinausgeworfen.

Diese Tatsache hat weitere (und wie ich finde sehr angenehme) Konsequenzen: Auch die primitiven Datentypen (Integer, Double, String und wie sie alle heißen) haben Eigenschaften und Methoden. Das String-Objekt beispielsweise lässt sich zwar nach wie vor noch mit Left, Mid und Right buchstäblich auseinander nehmen, aber das sollten Sie damit nicht mehr machen. Das String-Objekt in .NET bietet viel elegantere Möglichkeiten für seine Verarbeitung (die Methode SubString ist beispielsweise das Mid-Pendant), und wenn Softwareentwickler, die in anderen .NET-Sprachen entwickeln, Ihre Programme lesen, dann helfen Sie ihnen, indem Sie die sprachübergreifenden Methoden der Objekte verwenden und nicht die proprietären von Visual Basic.

---

**TIPP:** Wenn Sie schnell herausfinden wollen, welche Methoden und Eigenschaften Ihnen die primitiven Datentypen anbieten, deklarieren Sie einfach irgendwo in einem einfachen Projekt einen primitiven Datentyp Ihrer Wahl, schreiben Sie den Variablen eine Zeile darunter, und recherchieren Sie mithilfe der Vervollständigungsliste, welche Methoden und Eigenschaften er Ihnen bietet.

---



**Abbildung 5.6:** Wenn Sie wissen wollen, welche Methoden und Eigenschaften ein primitiver Variablentyp zur Verfügung stellt – deklarieren Sie ihn, und benutzen Sie IntelliSense für eine Funktionsübersicht

## Direktdeklaration von Variablen in For-Schleifen

Um in Visual Basic 6.0 Schleifenvariablen zu verwenden, mussten Sie diese umständlich zuerst deklarieren und konnten diese erst danach verwenden:

### Achtung! VB6-Code!

```
Dim zähler As Integer
For zähler = 1 To 100
    Debug.Print zähler
Next
```

In allen .NET-Basic-Derivaten geht das eleganter:

```
Private Sub btnSchleifendemo_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles btnSchleifendemo.Click
    For zähler As Integer = 0 To 100
        Debug.Print("Wert:" & zähler)
    Next
    .
    .
    .
```

Das gilt im Übrigen auch für For Each-Schleifen, wie man Sie in VB6 überwiegend beim Iterieren durch Collection-Objekte verwendet hat:

---

**HINWEIS:** Die Variable ist dann zudem noch eine »Blockvariable«, d.h. sie ist nur in einem Block, hier der Next-Schleife gültig. Somit kennt .NET auch neue Gültigkeitsbereiche von Variablen – doch dazu mehr im Abschnitt »Gültigkeitsbereiche von Variablen« ab Seite 161.

---

### Achtung! VB6-Code!

```
Dim auflistung As Collection
Set auflistung = New Collection
auflistung.Add 5
auflistung.Add 10
auflistung.Add 15
auflistung.Add 20
```

```
Dim element As Variant
For Each element In auflistung
    Debug.Print element
Next
```

Das VB.NET-Pendant sieht typischerweise folgendermaßen aus:

```
.
.
.
    Dim auflistung As New ArrayList
    auflistung.Add(5)
    auflistung.Add(10)
    auflistung.Add(15)
    auflistung.Add(20)

    For Each element As Integer In auflistung
        Debug.Print("Wert:" & element)
    Next
End Sub
```

## Unterschiede bei verwendbaren Variablentypen für For/Each in VB6 und VB.NET

In diesem Zusammenhang eine kleine Anmerkung: In VB6 konnten innerhalb von For Each-Schleifen nur Variant- oder aus Klassen entstandenen Objektvariablen verwendet werden. Direkt einen Grunddatentyp zu verwenden war, wie im obigen .NET-Listing zu sehen, nicht möglich.

Diese Einschränkung muss natürlich in .NET wegfallen, weil, wie Sie bereits lesen konnten, jede Variable auf der Basisklasse aller Basisklassen Objekt basiert. Aus diesem Grund können Sie in For Each-Schleifen in jedem der .NET-Basic-Derivate auch primitive Datentypen wie Integer, Long, Single, Double, Date, Decimal, String, etc. verwenden.

## Gültigkeitsbereiche von Variablen

VB6 kannte nur drei Gültigkeitsbereiche innerhalb von Modulen, Klassen oder Formularen.

- Globale Variablen, die mit `Global` innerhalb von Modulen definiert wurden, und auf die Sie vom ganzen Projekt aus zugreifen konnten.
- Member-Variablen eines Moduls, eines Formulars oder einer Klasse, die innerhalb des ganzen Codefiles, aber nur da, gültig waren.
- Lokale Variablen, die innerhalb einer Sub, einer Function oder einer Property-Prozedur definiert wurden, und dann ab dem Zeitpunkt ihrer Definition für den kompletten Rest der Prozedur ihre Gültigkeit behielten.
- Blockvariablen, die innerhalb eines If-Blocks, einer For/Next- oder Do/Loop- oder While/Wend-Schleife gültig sind.

## Globale bzw. öffentliche (public) Variablen

Globale Variablen nach dem Vorbild von VB6 gibt es nicht mehr; jedenfalls nicht mit dem `Global`-Modifizierer von VB6. Möchten Sie, dass auf eine Variable eines Moduls von außen zugegriffen werden kann, definieren Sie sie als öffentlich, und das geschieht mit dem Modifizierer `Public`. VB6 konnte das im Übrigen auch schon – viele Umsteiger von DOS-Basic-Derivaten waren aber an die `Global`-Syntax gewöhnt und verwendeten sie daher auch weiter.

Anstatt also wie in VB6 vielfach gemacht auf Modulebene eine Variable mit

```
Global einInteger As Integer
```

zu deklarieren, verwendeten Sie in VB.NET

```
Public einInteger As Integer
```

---

**HINWEIS:** Auch in einer objektorientierten Programmiersprache gibt es die Möglichkeit, bestimmte Instanzen innerhalb seines Projektes zu haben, an denen allgemein zugängliche Variablen liegen. Bestimmte Programmeinstellungen, die von überall im Projekt abgerufen werden können, machen das auch bei der OOP erforderlich. Doch Sie sollten es auf jeden Fall vermeiden, eine komplette Kommunikation zwischen verschiedenen Programmeinheiten über diese Art und Weise abzuwickeln, denn das entspricht nicht der OOP.

---

---

Es ist aber typisch für die prozedurale Programmierung bzw. schlechte prozedurale Programmierung, da man den Austausch von Informationen auch hier über dezidierte Parameter abwickeln sollte.

Vermeiden Sie also das Offenlegen von Variablen auf die hier gezeigte Methode, wann immer es geht. Welche Alternativen sich Ihnen stattdessen bieten, werden Sie fast schon automatisch wissen, wenn Sie den OOP-Teil dieses Buches durchgearbeitet haben.

---

## Variablen mit Gültigkeit auf Modul, Klassen oder Formularebene

Hier hat sich im Gegensatz zu VB6 nichts Konzeptionelles getan. Eine Variable, die Sie beispielsweise als `Private` auf Modul, Formular oder Klassenebene definiert haben, war von außen zwar nicht sichtbar (eben »privat«), aber Sie konnten auf sie überall im Modul, Formular oder Ihrer Klasse zugreifen:

### Achtung! VB6-Code!

```
Private einIntegerMember As Integer
```

```
Private Sub Command1_Click()  
    'Von hieraus ist dranzukommen  
    einIntegerMember = 10  
End Sub
```

```
Private Sub Command2_Click()  
    'Von hieraus auch  
    einIntegerMember = 10  
End Sub
```

In den VB.NET-Derivaten hat sich an diesem Gültigkeitsbereich nichts geändert. In der OOP spricht man übrigens bei derartig deklarierten Variablen von »Membervariablen« (etwa: Mitgliedervariablen), und sie sind sogar ein ganz wichtiges Merkmal und elementarer Bestandteil von Klassen. Denn ein Formular ist in .NET (und war es auch schon in VB6) eine Klasse.

## Gültigkeitsbereiche von lokalen Variablen

Bei Variablen, die auf Prozedurebene deklariert wurden (also innerhalb von Subs, Functions oder Property-Prozeduren) gibt es eine ganz entscheidende, sehr wichtige Neuerung. Während in VB6 Variablen innerhalb von Prozeduren überall ab dem Zeitpunkt ihrer Deklaration gültig waren, sind sie das in VB.NET ab dem Zeitpunkt ihrer Deklaration nur innerhalb der Struktur, in der sie definiert sind. »Struktur« in diesem Zusammenhang bedeutet im Prinzip irgendetwas, was Code in irgendeiner Form einklammern kann – das können beispielsweise `If/Then/Else`-Abfragen, `For/Next`- oder `Do/Loop`-Schleifen aber auch `With`-Blöcke sein. Es gilt dabei die Regel: Jede Strukturanweisung, die dazu führt, dass der Editor den dazwischen platzierten Code einrückt, begrenzt automatisch den Gültigkeitsbereich von Variablen, die in diesem Block definiert sind. Ein Beispiel zeigt Abbildung 5.7.

```

Option Strict On

Module Module1
    Sub Main()
        For zähler As Integer = 0 To 10
            Dim fünfGefunden As Boolean
            If zähler = 5 Then
                fünfGefunden = True
            End If
        Next
        If fünfGefunden Then
            Debug.Print("Die Zahl 5 kam in der Zahlenreihe vor!")
        End If
    End Sub
End Module

```

Abbildung 5.7: Variablen sind nur in dem Strukturblock gültig, in dem sie deklariert wurden

Wie in der Abbildung zu sehen, versucht das Programm beim zweiten Mal auf fünfGefunden zuzugreifen, nach dem der Strukturbereich und damit auch der Gültigkeitsbereich der Variable verlassen wurde – und das führt zu einem Fehler.

Wollten Sie fünfGefunden in jedem Strukturblock zugreifbar machen, müssten Sie folgende Änderungen vornehmen:

```

Sub Main()

    Dim fünfGefunden As Boolean

    For zähler As Integer = 0 To 10
        'Dim fünfGefunden As Boolean
        If zähler = 5 Then
            fünfGefunden = True
        End If
    Next

    If fünfGefunden Then
        Debug.Print("Die Zahl 5 kam in der Zahlenreihe vor!")
    End If
End Sub

```

Die Änderungen sind hier im Listing durch Fettschrift gekennzeichnet.

Diese Regel für Gültigkeitsbereiche führt dazu, dass Variablen innerhalb mehrerer Strukturblocke mehrfach unter gleichem Namen deklariert werden können, wie das folgende Beispiel zeigt:

```

Sub Main()
    For zähler As Integer = 0 To 10
        Dim fünfGefunden As Boolean
        If zähler = 5 Then
            fünfGefunden = True
        End If
    Next

```

```

For zähler As Integer = 0 To 10
  Dim fünfGefunden As Boolean
  If zähler = 5 Then
    fünfGefunden = True
  End If
Next
End Sub

```

Hier wird die Variable fünfGefunden innerhalb einer Prozedur zweimal deklariert – dennoch meldet Visual Basic keinen Fehler, weil sich beide Deklarationen in unterschiedlichen Gültigkeitsbereichen befinden.

Allerdings:

```

Option Strict On
Module Module1
  Sub Main()
    For zähler As Integer = 0 To 10
      Dim fünfGefunden As Boolean
      If zähler = 5 Then
        fünfGefunden = True
      End If
      For zweiterZähler As Integer = 0 To 10
        Dim fünfGefunden As Boolean
        If zähler = 5 Then
          fünfGefunden = True
        End If
      Next
    Next
  End Sub
End Module

```

**Abbildung 5.8:** Variablen eines übergeordneten Gültigkeitsbereichs dürfen solche eines untergeordneten nicht verbergen

Das Deklarieren von Variablen gleichen Namens in einem Gültigkeitsbereich der einen anderen kapselt, funktioniert natürlich nicht, denn Variablen in einem übergeordneten Gültigkeitsbereich sind ohnehin immer von einem untergeordneten aus zugreifbar. Eine entsprechende Fehlermeldung würde, falls Ihnen das passiert, dann so ausschauen, wie in Abbildung 5.8 zu sehen.

## Arrays

In Visual Basic 6.0 konnten Array-Grenzen frei definiert werden. Das folgende Codekonstrukt hatte dort also durchaus Gültigkeit:

## Achtung! Visual Basic 6.0-Code!

```
Dim einArray(-5 To 10) As Integer
For z% = -5 To 10
    einArray(z%) = 10
Next z%
```

Das geht in den .NET-Basic-Derivaten nicht mehr. Arrays in VB.NET sind grundsätzlich 0-basierend, und können grundsätzlich nicht mehr mit To definiert werden. Anders als in anderen .NET-Programmiersprachen geben Sie aber in Visual Basic .NET nicht die Anzahl der zu dimensionierenden Elemente sondern die höchste Array-Grenze an. Die Anweisung:

```
Dim einArray(10) As Integer
```

definiert also nicht 10 Elemente, sondern die Elemente 0–10 – und das sind 11 Elemente.

Der besseren Lesbarkeit halber erlaubt deswegen Visual Basic 2005 auch wieder die Angabe von To bei der Dimensionierung – der untere Wert muss dabei aber grundsätzlich 0 sein. Die erneute Verwendungsmöglichkeit von To bei der Dimensionierung ist also reine Kosmetik:

```
'Ab VB2005 möglich:
Dim einArray(0 To 10) As Integer ' Definiert 11 Elemente
```

---

**HINWEIS:** Wie schon erwähnt gilt diese Art der Dimensionierung von Elementen eines Arrays nur in Visual Basic .NET. In C# beispielsweise wird, wie in allen anderen .NET-Programmiersprachen, die standardmäßig von Microsoft mit Visual Studio ausgeliefert werden, die Anzahl der Elemente angegeben, und das sieht dann beispielsweise so aus:

---

```
// 10 Elemente definieren
int[] einArray=new int[10];
// Alle 10 Elemente (0-9) durchlaufen...
for (int zähler=0; zähler<10; zähler++)
    // ...und jedem den Wert 10 zuweisen.
    einArray[zähler]=10;
```

Das ist beispielsweise dann wichtig zu wissen, wenn Sie ein Beispiel für die Lösung eines Problems im Internet nur als C#-Version gefunden haben. Hier werden häufig Fehler beim »Übersetzen« von C# nach VB gemacht.

## Die Operatoren += und -= und ihre Verwandten

Mit += und -= ist Visual Basic um Operatoren für numerische Berechnungen und bei &= auch für Stringverkettungen reicher geworden – das folgende Beispiel verdeutlicht ihre Verwendung.

```
If Not IsRangeOkProper(txtValue.Text) Then
    MsgBox("Wertebereich wurde überschritten!" & vbCrLf & "(Nur im Integerbereich von 0 bis 32768)" & vbCrLf _
        & "Info: Das war die " & Str(locErrorCount + 1) & ". Fehleingabe", _
        MsgBoxStyle.OKOnly Or MsgBoxStyle.Exclamation, "Falsche Eingabe")
    locErrorCount += 1
    txtValue.Focus()
Exit Sub
End If
```

Es gibt andere Operatoren in VB.NET, die das ebenfalls können – die folgende Tabelle zeigt, welche das sind:

Operation	Kurzform	Beschreibung
<code>var = var + 1</code>	<code>var += 1</code>	Den Variableninhalt um eins erhöhen.
<code>var = var - 1</code>	<code>var -= 1</code>	Den Variableninhalt um eins verringern.
<code>var = var * 2</code>	<code>var *= 2</code>	Den Variableninhalt verdoppeln (mal zwei nehmen).
<code>var = var / 2</code>	<code>var /= 2</code>	Den Variableninhalt halbieren (durch zwei teilen – Fließkommadivision).
<code>var = var \ 2</code>	<code>var \= 2</code>	Den Variableninhalt ohne Rest halbieren (durch zwei teilen – Integerdivision).
<code>var = var ^ 3</code>	<code>var ^= 3</code>	Den Variableninhalt mit drei potenzieren.
<code>varString = VarString &amp; "Klaus"</code>	<code>varString &amp;= "Klaus"</code>	An den Inhalt des String <i>varString</i> die Zeichenkette »Klaus« anhängen.

**Tabelle 5.3:** Kurzformen von Operatoren in Visual Basic

**HINWEIS:** Auch wenn die Verwendung der Kurzformen weniger Tipparbeit macht – eine schnellere Codeausführung bewirkt sie nicht.

Also ganz egal ob Sie

```
intvar = intvar + 1
```

oder

```
intvar += 1
```

schreiben – der Compiler wird aus beiden Angaben den gleichen Code erzeugen.

## Die Bitverschiebeoperatoren << und >>

Zusätzlich zu den genannten Operatoren gibt es ab VB.NET 2003 Bitverschiebeoperatoren, mit denen die einzelnen Bits von Integerwerten schrittweise nach links oder nach rechts verschoben werden können. Ohne im Detail auf die Funktionsweise des Binärsystems eingehen zu wollen: Eine Verschiebung der Bits eines Integerwertes nach links verdoppelt seinen Wert, eine Verschiebung nach rechts teilt seinen Wert ohne Rest durch 2.

Aus binär 101 (dezimal 5) wird durch Rechtsverschiebung also binär 10 (dezimal 2) und durch Linksverschiebung binär 1010 (dezimal 10).

Der folgende Code demonstriert einen Multiplikationsalgorithmus auf Bitebene. Die Älteren unter Ihnen erinnern sich sicherlich noch an Ihre Commodore 64-Zeiten. Dort galt das Anwenden solcher Algorithmen in Assembler zur täglichen Praxis!

```
Private Sub btnMultiplikationMitBitverschiebung_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnMultiplikationMitBitverschiebung.Click
    Dim wert1, wert2, ergebniswert, hilfswert As Integer
    wert1 = 10
    wert2 = 5
```

```

ergebniswert = 0
hilfswert = wert2

'Dieser Algorithmus funktioniert so, wie Sie
'auch im Dezimalsystem "zu Fuß" multiplizieren:
'
'(10) (5)
'1010 * 101 =
'-----
'      1010 +
'      0000 +
'      1010
'-----
'      101010 = 50

'Die "5" wird dazu bitweise nach rechts verschoben,
'um ihr rechts äußeres Bit zu testen. Ist es gesetzt,
'wird der Wert von 10 zunächst addiert, und dann sein
'kompletter "Bitinhalt" um eine Stelle nach links verschoben;
'ist es hingegen nicht gesetzt, passiert gar nichts.
'Dieser Vorgang wiederholt sich solange, bis alle
'Bits von "5" verbraucht sind - die Variable hilfswert,
'die diesen Wert verarbeitet, also 0 geworden ist.
'Für eine Multiplikation sind also gerade so viele
'Additionen nötig, wie Bits im zweiten Wert gesetzt sind.
Do
  If (hilfswert And 1) = 1 Then
    ergebniswert += wert1
  End If
  wert1 = wert1 << 1
  hilfswert = hilfswert >> 1
Loop Until hilfswert = 0
Debug.Print("Das Ergebnis lautet:" & ergebniswert)
End Sub

```

## Fehlerbehandlung

Mir persönlich war die Fehlerbehandlung im alten Visual Basic immer ein Gräuel. Trat in einer sehr langen Routine im fertigen Programm ein Fehler auf, war es vergleichsweise schwierig oder mit großem Aufwand verbunden, die genaue Stelle des Fehlers zu lokalisieren. Zwar konnten Sie mit der Systemvariablen `Err` die Zeile in der Fehlerbehandlungsroutine anzeigen lassen, in der der Fehler aufgetreten war, aber dazu mussten Sie manuell vor jede Zeile eine Zeilennummer setzen – selbst für »damalige« Verhältnisse war das doch eine eher vorsintflutliche Vorgehensweise.

Das geht heute viel, viel einfacher – auch wenn die VB.NET-Entwickler die ursprüngliche Verfahrensweise auch noch zulassen. Wahrscheinlich wollten sie nicht zu viele Inkompatibilitäten schaffen. Doch schauen wir uns abermals das Vorher und das Nachher an – hier am Beispiel einer kleinen VB6-Routine, die eine Datei in eine String-Variable liest, oder dies zumindest versucht. Achten Sie im Listing auf die Kommentare, die mit den Ziffern den Programmverlauf im Falle eines Fehlers kennzeichnen.

## Achtung! Visual Basic 6.0-Code!

```
Private Sub Command1_Click()

    Dim DateiNichtGefundenFlag As Boolean
    On Local Error GoTo 1000

    Dim ff As Integer
    Dim meinDateiInhalt As String
    Dim zeilenspeicher As String
    ff = FreeFile
    '1: Hier tritt der Fehler auf
    Open "C:\EineTextdatei.TXT" For Input As #ff
    '3: dann wieder hier hin
    If DateiNichtGefunden Then
        '4: um dann diese Meldung auszugeben.
        MsgBox ("Die Datei existiert nicht")
    Else
        'Dieser Block wird nur ausgeführt,
        'wenn alles OK war.
        Do While Not EOF(ff)
            Line Input #ff, zeilenspeicher
            meinDateiInhalt = meinDateiInhalt & zeilenspeicher
        Loop
        Close ff
        Debug.Print meinDateiInhalt
    End If
    'Und das ist auch nötig, damit das
    'Programm nicht in die Fehleroutine rennt.
    Exit Sub

    '2: Hier springt dann das Programm hin
1000 If Err.Number = 53 Then
    DateiNichtGefunden = True
    Resume Next
End If

End Sub
```

Unfassbar, oder? Um einen simplen Fehler abzufangen, muss sich das Programm kreuz und quer durch den Programmcode quälen, und – einem Steinbock im Himalaja gleich – hin und her springen, was das Zeug hält. Und wir fangen hier gerade mal einen einzigen Fehler ab!

Das noch Unfassbarere ist: Prinzipiell ist dieser On Error GoTo-Quatsch auch noch in allen VB.NET-Derivaten möglich, wenn auch alles andere als nötig – wie Sie gleich noch sehen werden. Das Einlesen einer Textdatei funktioniert hier zwar ein wenig anders, da es Open und Close in dieser VB6-Form nicht mehr gibt. Aber darum geht es an dieser Stelle auch gar nicht.

---

**WICHTIG:** Der einzige Unterschied, jedenfalls was die Fehlerbehandlung angeht: Zeilennummern müssen, wie andere Sprungmarken, mit Doppelpunkten versehen werden. Wollte man also das obige Programm in VB.NET übersetzen, käme vielleicht dieses (leider) mögliche Ergebnis dabei heraus:

---

```

Private Sub btnDateiLesenDotNet_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles btnDateiLesenDotNet.Click
    'WICHTIG:
    'Um auf die IO-Objekte zuzugreifen, muss "Imports System.IO"
    'am Anfang der Codedatei stehen!

    Dim DateiNichtGefundenFlag As Boolean
    'Selbe Blödsinn hier!
    On Error GoTo 1000

    Dim meinDateiInhalt As String
    '1: Wenn hier ein Fehler auftritt
    Dim dateiStromLeser As New StreamReader("C:\EineTextdatei.txt")
    '3: um das durch Resume Next hier wieder zu landen
    If DateiNichtGefundenFlag Then
        '4: und den Fehler abzufangen
        MessageBox.Show("Datei wurde nicht gefunden!")
    Else
        'Trat kein Fehler auf, wird dieser Block ausgeführt
        meinDateiInhalt = dateiStromLeser.ReadToEnd()
        dateiStromLeser.Close()
        'Und der Dateiinhalt ausgegeben.
        Debug.Print(meinDateiInhalt)
    End If
    Exit Sub

    '2: Fährt der Programmablauf hier fort
1000: If Err.Number = 53 Then
        DateiNichtGefundenFlag = True
        Resume Next
    End If
End Sub

```

## Elegantes Fehlerabfangen mit Try/Catch/Finally

Es gibt aber eine viel, viel elegantere Möglichkeit in VB.NET, Fehlerbehandlungen zu implementieren. Im Gegensatz zu `On Error GoTo` erlaubt die Struktur `Try/Catch/Finally` Fehler an den Stellen zu behandeln, an denen sie auch tatsächlich auftreten können. Schauen Sie sich zur Verdeutlichung das folgende Beispiel an, das die auf `Try/Catch` adaptierte Version des obigen Beispiels enthält:

```

Private Sub btnDateiLesenDotNet_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles btnDateiLesenDotNet.Click
    'WICHTIG:
    'Um auf die IO-Objekte zuzugreifen, muss "Imports System.IO"
    'am Anfang der Codedatei stehen!

    Dim meinDateiInhalt As String
    Dim dateiStromLeser As StreamReader

```

```

Try
'Diese folgenden Befehle probieren (try=versuchen, ausprobieren).
dateiStromLeser = New StreamReader("C:\meineTextdatei.txt")
meinDateiInhalt = dateiStromLeser.ReadToEnd()
dateiStromLeser.Close()
Debug.Print(meinDateiInhalt)
Catch ex As FileNotFoundException
'Hier werden nur FileNotFoundExceptions abgefangen
MessageBox.Show("Datei wurde nicht gefunden!" & vbNewLine & _
vbNewLine & "Der Ausnahmetext lautete:" & vbNewLine & ex.Message, _
"Ausnahme", MessageBoxButtons.OK, MessageBoxIcon.Error)
Catch ex As Exception
'Hier werden alle anderen bis zu diesem Zeitpunkt nicht
'behandelten Ausnahmen abgefangen
'Hier werden alle anderen bis zu diesem Zeitpunkt nicht
'behandelten Ausnahmen abgefangen
MessageBox.Show("Beim Verarbeiten der Datei trat eine Ausnahme auf!" & vbNewLine & _
"Die Ausnahme war vom Typ:" & ex.GetType.ToString & vbNewLine & _
vbNewLine & "Der Ausnahmetext lautete:" & vbNewLine & ex.Message, _
"Ausnahme", MessageBoxButtons.OK, MessageBoxIcon.Error)
End Try
End Sub

```

Was passiert hier? Alle Anweisungen, die zwischen der Try- und der ersten Catch-Anweisung platziert sind, befinden sich in einer Art »Versuchsmodus«. Tritt bei der Ausführung einer dieser Anweisungen ein Fehler auf, springt das Programm automatisch in den »zutreffenden« Catch-Block. Und was bedeutet dabei »zutreffend«?

### Das Ausnahmenabfangen ist nicht nur auf einen Ausnahmetyp beschränkt

Wenn Sie in den Editor einfach nur die Zeichenfolge »Try« eingeben und anschließend **Eingabe** drücken, fügt dieser automatisch den folgenden Block ein:

```

Try

Catch ex As Exception

End Try

```

Exception lautet die in der Vererbungshierarchie zuoberst stehende Klasse, deren Instanzen (hier durch ex referenziert) ausnahmslos alle Ausnahmen abfangen können. Doch es könnte, wie im Beispielcode zu sehen, sinnvoll sein, zwischen den vielen verschiedenen Ausnahmentypen zu differenzieren. Ihr Programm soll nämlich vielleicht auf eine Ausnahme, die durch eine nicht vorhandene Datei ausgelöst wird, anders reagieren, als auf eine Datei, die zwar vorhanden, aber gerade mit einem anderen Programm verarbeitet wird.

Sie können das am Beispielcode nachvollziehen. Wenn Sie diesen starten, wird eine Ausnahme ausgelöst, die sich FileNotFoundException nennt. Ein Catch mit diesem Klassennamen als Argument, fängt nur Ausnahmen dieses Typs ab. Dementsprechend wird der dort darunter stehende Code ausgeführt, und der Code sorgt dafür, dass ein entsprechender Dialog angezeigt wird.

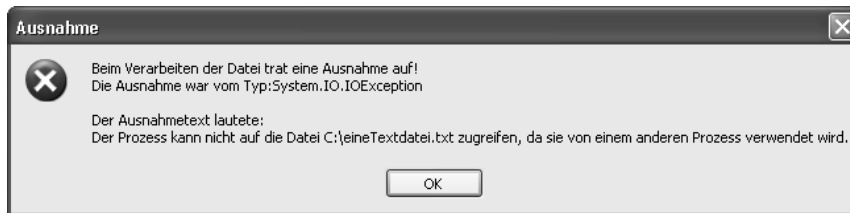


**Abbildung 5.9:** Im Beispielcode wird diese Meldung ausgegeben, wenn die Datei nicht vorhanden war, also eine *FileNotFoundException* vorlag

Wenn Sie nun aber auf Laufwerk C: eine Textdatei mit diesem Namen anlegen, und diese Datei anschließend beispielsweise in Microsoft Word öffnen, dann produziert die Zeile

```
dateiStromLeser = New StreamReader("C:\meineTextdatei.txt")
```

abermals eine Ausnahme – doch dieses Mal keine vom Typ *FileNotFoundException* sondern vom Typ *IOException*:



**Abbildung 5.10:** Jetzt ist die Datei zwar vorhanden, aber durch Word bereits geöffnet

Und wieso wird diese Ausnahme durch `Catch as Exception` abgefangen, obwohl Sie vom Typ *IOException* ist? Das liegt daran, dass *IOException* auf *Exception* basiert. In der objektorientierten Programmierung können durch Vererbung aus Klassen erweiterte Klassen entstehen, aus diesen wiederum nochmals spezialisierte, und so fort. Genau so ist das bei Ausnahmeklassen. *Exception* selbst ist die Basisklasse. Davon abgeleitet ist die Ausnahmeklasse *SystemException*. Und davon wiederum ist *IOException* abgeleitet. Da wir im Beispiel aber *IOException* nicht gesondert behandeln, wird der `Catch`-Block angesteuert (so vorhanden), der zumindest eine der Basisklassen der aufgetretenen Ausnahmeklassen repräsentiert. Bei der Verwendung von *Exception* sind Sie dabei immer auf der sicheren Seite, denn auf dieser basieren *alle* anderen Ausnahmeklassen. Und so wird zwar die Ausnahme *FileNotFoundException* gesondert behandelt, aber bei allen anderen Ausnahmen der `Catch`-Block ausgeführt, der mit *Exception* selbst arbeitet.

---

**WICHTIG:** Sie können in einem `Try-Catch`-Block so viele `Catch`-Zweige implementieren, wie Sie möchten, und damit alle denkbaren Ausnahmetypen gezielt abfangen. Achten Sie dabei aber darauf, dass Sie die spezialisierten Ausnahmetypen nach oben stellen, und die, auf denen diese basieren, erst weiter unten kommen. Anderenfalls handeln Sie sich Ärger mit dem Compiler ein, der das schlichtweg nicht zulässt. Denn eine Basisausnahmeklasse würde in diesem Fall eine spezialisierte Ausnahme schon längst verarbeitet haben, bevor der `Catch`-Block mit der spezialisierten Ausnahme zum Zuge käme. Und diesen »Un-Sinn« unterbindet der Compiler von vornherein:

---

```

Try
    'Diese folgenden Befehle probieren (try=versuchen, ausprobieren).
    dateiStromLeser = New StreamReader("C:\eineTextdatei.txt")
    meinDateiInhalt = dateiStromLeser.ReadToEnd()
    dateiStromLeser.Close()
    Debug.Print(meinDateiInhalt)
Catch ex As Exception
    'Hier werden alle anderen bis zu diesem Zeitpunkt nicht
    'behandelten Ausnahmen abgefangen
    MessageBox.Show("Beim Verarbeiten der Datei trat eine Ausnahme auf!" & vbCrLf & _
        "Die Ausnahme war vom Typ:" & ex.GetType.ToString & vbCrLf & _
        vbCrLf & "Der Ausnahmetext lautete:" & vbCrLf & ex.Message, _
        "Ausnahme", MessageBoxButtons.OK, MessageBoxIcon.Error)
Catch ex As FileNotFoundException
    'Der Catch-Block wird niemals erreicht, da "System.IO.FileNotFoundException" von "System.Exception" erbt.
    'abfangen
    MessageBox.Show("Datei wurde nicht gefunden!" & vbCrLf & _
        vbCrLf & "Der Ausnahmetext lautete:" & vbCrLf & ex.Message, _
        "Ausnahme", MessageBoxButtons.OK, MessageBoxIcon.Error)
End Try

```

**Abbildung 5.11:** Catch-Blöcke mit spezialisierteren Ausnahmeklassen müssen vor den Basisausnahmeklassen platziert werden, sonst meckert der Compiler

## Und wozu dient Finally?

Programmcode, den man in einem Finally-Block platziert, *wird in jedem Fall ausgeführt*. Auch dann, wenn ein Fehler auftritt, mit Catch abgefangen wird und die Anweisung ergeht, die gesamte Prozedur beispielsweise mit Return *eigentlich* noch im Catch-Block zu verlassen.

Denn normalerweise ist ja Return der letzte Befehl in einer Prozedur, ganz gleich, wo Return platziert wurde. Nach Return ist Schicht. Doch in einigen Fällen ist das nicht sinnvoll, gerade wenn es um Fehlerbehandlungen geht, und deswegen bildet Finally in diesem Fall die goldene Ausnahme.

Angenommen, Sie lesen aus einer Datei, und diese Datei haben Sie dazu vorher geöffnet. Beim Öffnen der Datei ist zwar kein Fehler aufgetreten, aber beim Lesen – vielleicht ist der Fall eingetreten, dass Sie über das Dateiende hinaus gelesen haben, und nun sind Zeilen, die Sie verarbeiten wollen, leer (also Nothing). Diesen Fall (das Abfangen einer Null-Referenz) haben Sie mit einem entsprechenden Catch-Block behandelt, und da es nichts Weiteres zu tun gibt, möchten Sie nach der Anzeige einer Fehlermeldung Ihre Lesenroutine mit Return direkt aus dem Catch-Block heraus verlassen. Das Problem: die Datei ist noch geöffnet, und Sie sollten sie schließen. Mit Finally lässt sich ein solcher Vorgang elegant implementieren.

Das folgende Beispiel simuliert einen solchen Prozess. Es liest »c:\eineTextdatei.txt« zeilenweise aus und wandelt die einzelnen Zeilen in Großbuchstaben um, bevor es sie zu einem Gesamttextblock zusammenführt. Doch da es viel zu viele Zeilen liest, funktioniert die ToUpper-Methode des Strings, die die Zeile in Großbuchstaben umwandeln soll, irgendwann nicht mehr, denn die ReadLine-Funktion greift ins Leere und liefert Nothing zurück:

```

Private Sub btnTryCatchFinally_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles btnTryCatchFinally.Click
    'WICHTIG:
    'Um auf die IO-Objekte zuzugreifen, muss "Imports System.IO"
    'am Anfang der Codedatei stehen!

    Dim meinDateiInhalt As String
    Dim dateiStromLeser As StreamReader

```

```

Try
'Diese folgenden Befehle probieren (try=versuchen, ausprobieren).
dateiStromLeser = New StreamReader("C:\eineTextdatei.txt")
Dim locZeile As String
meinDateiInhalt = ""
'Jetzt lesen wir zeilenweise aber viel zu viele Zeilen,
'und schießen daher irgendwann über das Ende der Datei hinaus:
Try
' Wenn Ihre Datei "C:\eineTextdatei" nicht gerade
' 1001 Zeilen enthält, knallt es hier, denn:
For zeilenzähler As Integer = 0 To 1000
locZeile = dateiStromLeser.ReadLine()
'locZeile ist jetzt Nothing, und dann kann
'die Konvertierung in Großbuchstaben nicht mehr
'funktionieren.
locZeile = locZeile.ToUpper
meinDateiInhalt &= locZeile
Next
Catch ex As NullReferenceException
MessageBox.Show("Die Zeile konnte nicht in umgewandelt werden, weil sie leer war!")
'Return? Aber die Datei ist doch noch geöffnet!!!
Return
Finally
'Egal! Auch bei Return im Catch-Block wird Finally in jedem Fall noch ausgeführt!
dateiStromLeser.Close()
End Try
'Aber diese Zeile wird nur im Erfolgsfall abgearbeitet:
Debug.Print(meinDateiInhalt)
Catch ex As FileNotFoundException
'Hier werden nur FileNotFoundExceptions abgefangen
MessageBox.Show("Datei wurde nicht gefunden!" & vbNewLine & _
vbNewLine & "Der Ausnahmetext lautete:" & vbNewLine & ex.Message, _
"Ausnahme", MessageBoxButtons.OK, MessageBoxIcon.Error)
Catch ex As Exception
'Hier werden alle anderen bis zu diesem Zeitpunkt nicht
'behandelten Ausnahmen abgefangen
MessageBox.Show("Beim Verarbeiten der Datei trat eine Ausnahme auf!" & vbNewLine & _
"Die Ausnahme war vom Typ:" & ex.GetType.ToString & vbNewLine & _
vbNewLine & "Der Ausnahmetext lautete:" & vbNewLine & ex.Message, _
"Ausnahme", MessageBoxButtons.OK, MessageBoxIcon.Error)

End Try
End Sub

```

---

**TIPP:** Sie können diesen Vorgang übrigens gut nachvollziehen, in dem Sie einen Haltepunkt mit **F9** in der Zeile setzen, die Return beinhaltet, und das Programm starten. Wenn Sie anschließend mit **F11** schrittweise durch das Programm steppen, werden Sie feststellen, dass nach dem Return der Code im Finally-Block noch abgehandelt wird.

---

# Kurzschlussauswertungen mit OrElse und AndAlso

Betrachten Sie den folgenden Codeblock:

```
'Kurzschlussauswertung beschleunigt den Vorgang.  
If locChar < "0" OrElse locChar > "9" Then  
    locIllegalChar = True  
    Exit For  
End If
```

Auffällig ist hier das Schlüsselwort `OrElse`. Es gibt ein weiteres, das nach dem gleichen Prinzip funktioniert: `AndAlso`. Beide entsprechen den Befehlen `Or` bzw. `And`, und auch sie dienen dazu, boolesche Ausdrücke logisch miteinander zu verknüpfen und auszuwerten – nur viel schneller. Ein Beispiel aus dem täglichen Leben soll das verdeutlichen:

Wenn Sie sich überlegen, ob Sie einen Regenschirm zu einem Spaziergang mitnehmen, da es vielleicht regnet *oder auch (or else)* zumindest sehr verhangen aussieht, dann machen Sie sich – berechtigterweise – schon keine Gedanken mehr, wie der Himmel aussieht, wenn Sie bereits festgestellt haben, *dass* es regnet. Sie brauchen das zweite Kriterium also gar nicht zu prüfen – der Schirm muss mit, sonst wird's nass! Genau das macht `OrElse` (bzw. `AndAlso`), und man nennt diese Vorgehensweise »Kurzschlussauswertung«.

Gerade bei Objekten bzw. dem Aufruf von Methoden können Ihnen Kurzschlussauswertungen helfen, Ihre Programme sicherer zu machen, wie das folgende Beispiel zeigt:

```
Private Sub btnAndAlsoDemo_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _  
    Handles btnAndAlsoDemo.Click  
    Dim einString As String = "Klaus ist das Wort, mit dem diese Zeile beginnt"  
    If einString IsNot Nothing AndAlso einString.Substring(0, 5).ToUpper = "KLAUS" Then  
        MessageBox.Show("Die Zeichenfolge beginnt mit Klaus!")  
    End If  
  
    If einString IsNot Nothing And einString.Substring(0, 5).ToUpper = "KLAUS" Then  
        MessageBox.Show("Die Zeichenfolge beginnt mit Klaus!")  
    End If  
End Sub
```

Wie erwartet, zeigt dieser Programmcode zweimal einen Meldungstext an. Denn `einString` hat in beiden Fällen einen Inhalt, und beide Male beginnt die Zeichenfolge (es ist ja auch dieselbe) mit »Klaus«. Doch ersetzen Sie nun Zeile

```
    Dim einString As String = "Klaus ist das Wort, mit dem diese Zeile beginnt"
```

durch,

```
    Dim einString As String = Nothing
```

und lassen Sie das Programm ein weiteres Mal laufen. Das Ergebnis zeigt die folgende Abbildung:

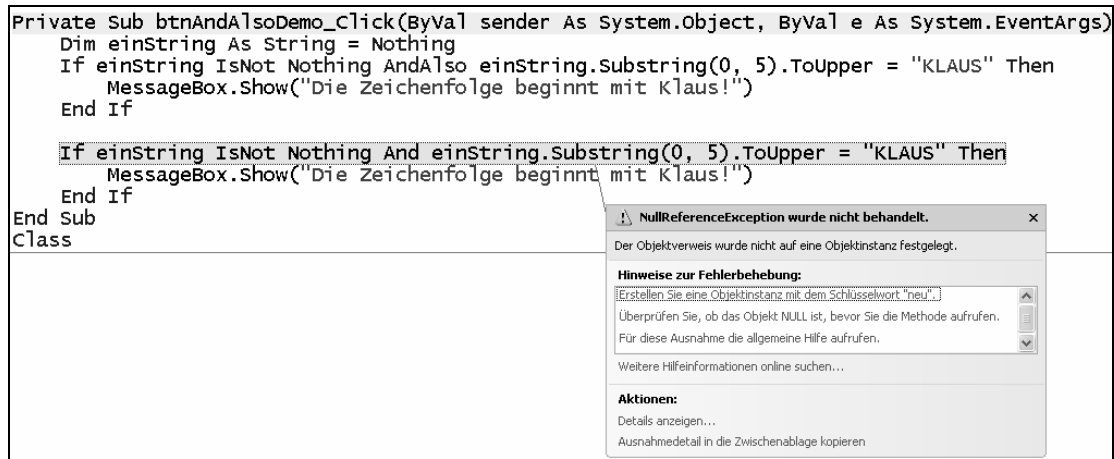


Abbildung 5.12: *AndAlso* dient Ihnen bei kombinierten Abfragen auf *Nothing* und Verwendungen von Instanzmethoden

Hier wird die Arbeitsweise von *AndAlso* deutlich. Die erste Abfrage funktioniert, weil der zweite, durch *AndAlso* verknüpfte Teil `einString.Substring(0, 5).ToUpper = "KLAUS"` schon gar nicht mehr abgehandelt wird. Das Objekt `einString` war nämlich *Nothing*, und bei der Verwendung von *AndAlso* interessieren alle folgenden Prüfteile nicht mehr.

In der zweiten Variante nur mit *And* wird, obwohl es in diesem Zusammenhang keinen Sinn macht, der zweite Part sehr wohl abgehandelt. Aber eben weil `einString` den Wert *Nothing* aufweist, können Sie seine Instanzfunktionen (`SubString`, `ToUpper`) nicht verwenden; das Programm bricht mit einer `NullReferenceException` ab.

## Variablen und Argumente auch an Subs in Klammern!

Klammern bei der Übergabe von Parametern an Funktionen waren schon in VB6 Usus. Das gilt nunmehr in allen VB.NET-Derivaten auch für die Übergabe von Variablen an Subs. Es mag Ihnen am Anfang lästig erscheinen, aber der Codeeditor nimmt Ihnen die Klammerei sogar ab, falls Sie sie nicht selber durchführen wollen. Man hätte dieses Verhalten sicherlich beim Alten lassen können. Aber auf diese Weise nähert sich Visual Basic dem Standard. Sowohl in C++ als auch in C# als auch in J# werden Argumente an »Subs« nicht ohne Klammern übergeben. Es gibt dort nämlich gar keine Subs, sondern nur Funktionen vom Typ »kein Rückgabewert«, der dort paradoxerweise obendrein noch einen speziellen Namen hat, nämlich *Void* (engl. etwa für »Hohlraum«, »Leere«, »Lücke«).

Streng genommen ist

```
Sub Methode(Übergabe as Irgendwas)
```

im Grunde also

```
'Das funktioniert natürlich nicht:
Function Methode(Übergabe as Irgendwas) as Void
```

damit eine Funktion, und ihre Parameter werden ergo auch in Klammern übergeben.

# Namespaces und Assemblies

Umsteigern von Visual Basic 6.0 macht das Konzept von Namespaces und Assemblies oftmals zu Anfang Schwierigkeiten, obwohl die dahinter stehenden Konzepte im Grunde genommen recht einfach zu verstehen sind. Die folgenden Abschnitte demonstrieren, was Assemblies und Namespaces sind, und wie sie bei .NET-Framework-Entwicklungen zur Anwendung kommen.

## Assemblies

Genau genommen ist eine Assembly ist eine Zusammenfassung von so genannten Modulen. Innerhalb eines Moduls können sich ausführbare Dateien oder Klassenbibliotheken befinden, die dann in einer Assembly zusammengefasst werden.

Doch in der Regel befindet sich – wenn Sie nicht explizit anderes sagen – entweder *ein* ausführbares Programm oder *eine* Klassenbibliothek in einer Assembly. Mit anderen, vereinfachenden Worten:

- Eine .EXE-Datei, die aus Ihrem Programmprojekt hervorgeht, ist eine Assembly.
- Eine .DLL-Datei, die aus einem Klassenbibliotheksprojekt hervorgeht, ist ebenfalls eine Assembly.
- Das Framework selbst besteht aus ganz vielen verschiedenen Klassenbibliotheken; und bei ihnen handelt es sich ebenfalls um Assemblies.

Die Nutzung einer Assembly, bei der es sich um eine ausführbare .EXE-Datei handelt, ist am einfachsten. Wenn eine .EXE-Datei aus Ihrem Projekt hervorgegangen ist, können Sie sie – vorausgesetzt das Framework ist auf dem entsprechenden Computer installiert – direkt starten.

Bei der Nutzung von Assembly-DLLs ist das anders. Assemblies, die in DLLs residieren, können nicht direkt gestartet werden – sie stellen nur eine gewisse Grundfunktionalität für verschiedene Themenbereiche zur Verfügung, derer sich andere Assemblies bedienen können. Zu diesem Zweck müssen Assemblies, die sich anderer Assemblies bedienen wollen, diese referenzieren.

Wie sieht das in der Praxis aus?

Schauen wir uns das an einem einfachen Beispiel an:

---

**BEGLEITDATEIEN:** Laden Sie zu diesem Zweck das Projekt *AssemblyDemo.sln*, das Sie im Pfad `.\VB 2005 - Entwicklerbuch\C - Ein- und Umstieg\` finden.

---

Dieses Projekt ist eine Windows Forms-Anwendung, und sie besteht aus nichts weiter als einem Formular und einer Schaltfläche. Beim genauen Hinsehen werden Sie jedoch feststellen, dass direkt nach dem Laden des Projektes in der Fehlerliste eine Reihe von Fehlern auftaucht. Und wenn Sie versuchen, das Formular zu öffnen, bekommen Sie statt des Formulars im Designer eine Fehlermeldung zu sehen.

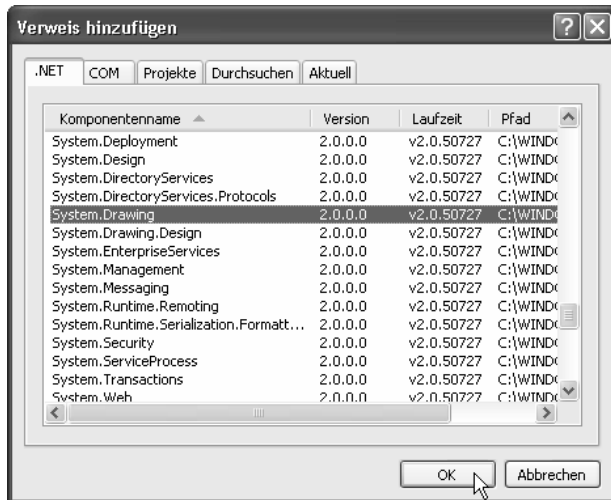


**Abbildung 5.13:** Irgendetwas hindert das Formular daran, dass es korrekt angezeigt werden kann

Und woran liegt das? Ganz einfach: Das Projekt muss auf eine Klassenbibliothek, auf eine Assembly aus dem Framework zurückgreifen. Doch das kann sie nicht, denn ich habe beim Erstellen des Projektes die Assembly-Verweisliste sabotiert. Das Projekt benötigt zur korrekten Darstellung des Formulars bestimmte Funktionalitäten aus der Assembly *System.Drawing.dll*, die Teil des Frameworks ist. Doch diese Funktionalitäten stehen dem Projekt momentan nicht zur Verfügung, weil sich die Assembly eben nicht in der Verweisliste befindet, und das Projekt sich der Funktionalitäten des Assembly auch nicht bedienen kann.

Sie können den Fehler folgendermaßen beheben:

- Schließen Sie zunächst den Designer, der die Fehlermeldung zeigt.
- Klicken Sie im Projektmappen-Explorer auf das Symbol *Alle Dateien anzeigen* – der entsprechende Tooltip, der erscheint, wenn Sie den Mauszeiger auf ein Symbol bewegen, hilft Ihnen, das richtige Symbol zu identifizieren.
- Der Projektmappen-Explorer blendet nun weitere Zweige mit entsprechenden Dateien und anderen Elementen ein – unter anderem einen namens *Verweise*.
- Öffnen Sie diesen Zweig. Dieser Zweig enthält alle Assemblies, auf die eine Windows Forms-Anwendung standardmäßig verweist; in diesem Fall aber eben nicht auf die fehlende *System.Drawing.dll*-Assembly.
- Klicken Sie mit der rechten Maustaste auf den Zweig *Verweise*, und wählen Sie aus dem Kontextmenü, das nun erscheint, *Verweis hinzufügen*.
- Es kann nun einen Augenblick dauern, bis der Dialog erscheint, den Sie auch in Abbildung 5.14 sehen. In diesem Dialog suchen Sie anschließend die .NET-Framework-Assembly *System.Drawing.dll*, klicken Sie an und klicken anschließend auf *OK*.



**Abbildung 5.14:** Mithilfe dieses Dialogs fügen Sie einen Verweis auf eine externe Assembly hinzu

Sie sehen, dass nun alle Fehler aus der Fehlerliste verschwunden sind. Ein Doppelklick auf das Formular öffnet dieses jetzt ordnungsgemäß, da der Designer nun auf die Objekte aus der gerade eingebundenen Assembly zurückgreifen kann, die für die korrekte Darstellung des Formulars und seiner Elemente erforderlich sind.

## Namespaces

Nun gibt es nach einer Grobzählung nicht weniger als rund 8.000 verschiedene Klassen im .NET-Framework. Und alle Klassen bieten verschiedene Methoden, Eigenschaften und Ereignisse – die Anzahl an Elementen, mit denen Sie es bei größeren Projekten zu tun haben, ist also schier gewaltig.

Aus diesem Grund macht es Sinn, die Klassen des Frameworks, die ja alle in verschiedenen Assemblies zu Hause sind,<sup>5</sup> in irgendeiner Form zu kategorisieren. Hier kommen die Namespaces ins Spiel.

Ein Namespace ist nichts weiter als eine »Sortier- und Wiederfindhilfe« für Klassen, die sich in irgendwelchen Assemblies befinden. Namespaces lassen auch überhaupt keinen Aufschluss auf den Namen einer Assembly zu. Eine Klasse, die sich in der Assembly *xyz.dll* befindet, kann sich im gleichen Namespace befinden, wie eine ganz andere Klasse der Assembly *zyx.dll*. Genauso gut kann eine Assembly Klassen für gleich mehrere Namespaces hervorbringen.

Um zu schauen, wie es sich mit den Namespaces beim Programmieren verhält, öffnen Sie das Formular aus dem vorangegangenen Beispiel (was sich ja nun öffnen lassen sollte), und doppelklicken Sie im Designer auf die Schaltfläche.

Wir möchten nun Code entwickeln, der zur Laufzeit eine zweite Schaltfläche im Formular platziert.

<sup>5</sup> Abbildung 5.14 gibt Ihnen auch einen Eindruck, wie viele verschiedene Assemblies es gibt, denn die Liste ist ja schon eindrucksvoll lang.

Dazu deklarieren wir eine Objektvariable vom Typ Button und fügen diesen der Controls-Auflistung des Formulars hinzu. Im Ergebnis sollte nach dem Programmstart ein Klick auf die schon vorhandene Schaltfläche bewirken, dass ein zweiter Button (Schaltfläche) im Formular erscheint.

Doch wenn Sie die folgende Abbildung betrachten, stellen Sie fest, dass schon die Deklaration eines Button-Objektes Probleme macht:

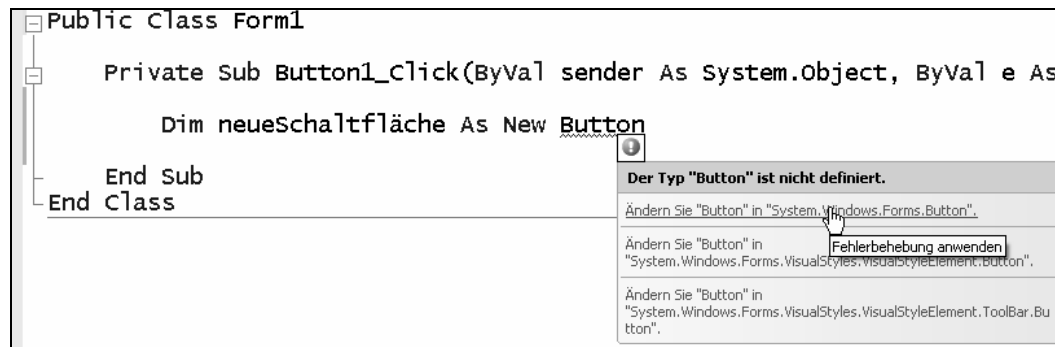


Abbildung 5.15: Prinzipiell ist diese Zeile nicht falsch, doch findet der Compiler die Button-Klasse aus irgendwelchen Gründen nicht

Die *Autokorrektur für intelligentes Kompilieren* gibt Ihnen, wie in der Abbildung zu sehen, einen Hinweis darauf, was schief läuft: Sie meint, es müsse System.Windows.Forms.Button und nicht einfach nur Button heißen. Der Hintergrund: Die Button-Klasse befindet sich in einer Assembly namens System.Windows.Forms.dll. Und diese Assembly definiert gleichzeitig, dass sich die Button-Klasse in einem Namespace (etwa: Namensbereich) namens System.Windows.Forms befindet.

Nun können Sie gleich drei Dinge tun, um den Fehler zu korrigieren:

- Sie geben den vollqualifizierten Namen der Klasse ein, also nicht nur seinen Klassennamen sondern auch den Namen des Namespaces. Das entspräche dem Korrekturvorschlag der *Autokorrektur für intelligentes Kompilieren*. Die Zeile müsste dann:

```
Dim neueSchaltfläche As New System.Windows.Forms.Button
lauten.
```

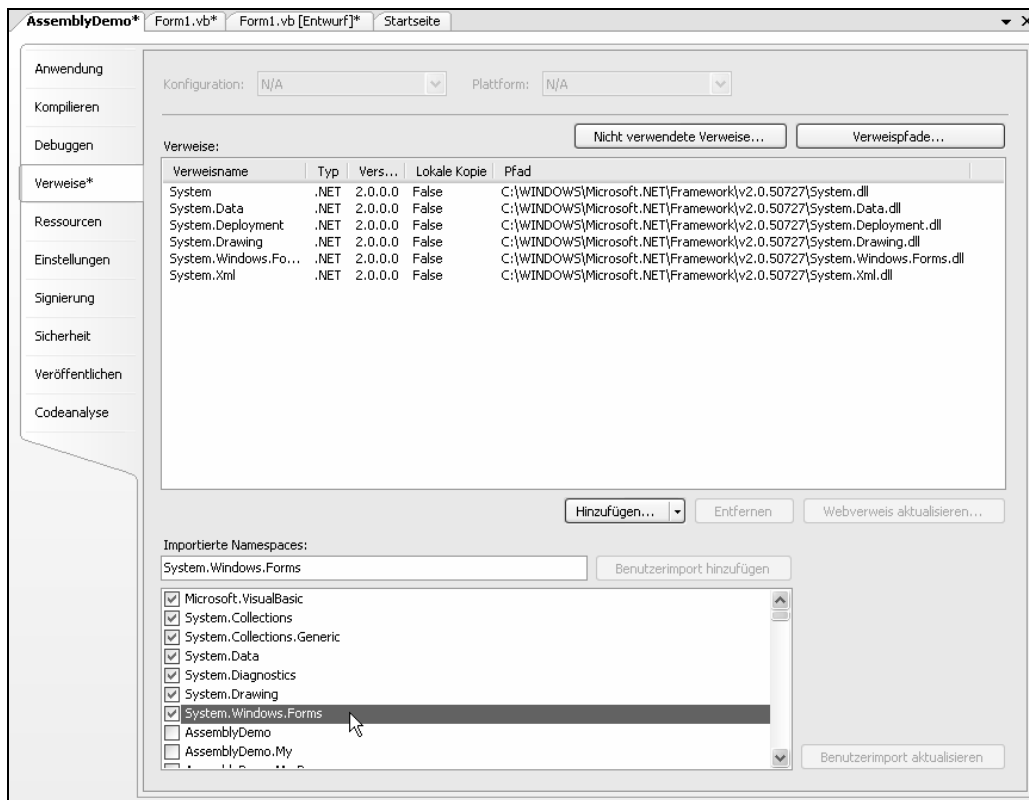
- Oder: Sie setzen eine Imports-Anweisung an den Anfang der Codedatei, die den Namensbereich für diese Codedatei einbindet. Dann können Sie innerhalb der Codedatei auf alle Klassen des importierten Namensbereiches zugreifen, ohne ständig den vollqualifizierten Namen eingeben zu müssen. Beispiel:

```
Imports System.Windows.Forms
```

```
Public Class Form1
    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
        Handles Button1.Click
        Dim neueSchaltfläche As New Button ' geht jetzt ohne Fehler, wegen 'Imports'
    End Sub
End Class
```

- Oder: Sie importieren den erforderlichen Namespace global für das ganze Projekt. Das funktioniert aber nicht mit einer Anweisung im Code, sondern lässt sich über die Projekteigenschaften

steuern. Rufen Sie dazu die Eigenschaften des Projektes ab (Menüpunkt *Projekt/AssemblyDemo-Eigenschaften*). Auf der Registerkarte *Verweise* finden Sie eine Liste aller eingebundenen Assemblies und darunter die für das Projekt global importierten Namespaces.



**Abbildung 5.16:** In dieser Liste bestimmen Sie, welche Namespaces global für das ganze Projekt eingebunden werden sollen

- Suchen Sie in dieser Liste den Namespace `System.Windows.Forms`. Anders als in der (zugegebenermaßen leicht frisierten) Abbildung wird dieser nicht oben in der Liste sondern viel weiter unten zu finden sein.
- Klicken Sie den Eintrag zweimal an (erst beim zweiten Mal erscheint das Häkchen).
- Klicken Sie auf das *Speichern*-Symbol in der Symbolleiste, und schließen Sie das Eigenschaftenfenster.

Nun können Sie die `Imports`-Anweisung wieder aus der obersten Zeile der Codedatei löschen; der Compiler wird die Deklaration dennoch akzeptieren, weil es nun durch die projektglobalen Imports-Einstellungen über den zu verwendenden `System.Windows.Forms`-Namespace Bescheid weiß.

Übrigens: Abbildung 5.15 gibt Ihnen einen weiteren Hinweis, wieso die Sortierung von Objekten in Namespaces so wichtig ist. Bei über 8.000 zu verwaltenden Objekten kann es nämlich schon einmal vorkommen, dass es Klassen mit gleichen Namen gibt. So gibt es ja nicht nur einen »normalen« Button, sondern auch einen `ToolBar.Button` – also eine Schaltfläche, die in einer Symbolleiste ihr zu

Hause hat. Deswegen kann Ihnen die Autokorrektur für intelligentes Kompilieren auch an dieser Stelle nicht exakt sagen, wie Sie den Fehler korrigieren sollen. Sie weiß nämlich selber nicht, welche der drei möglichen Button-Klassen aus den verschiedenen Namespaces gemeint ist.

Der Vollständigkeit halber sei hier übrigens noch der Rest des Codes nachgereicht, der die neue Schaltfläche tatsächlich zur Laufzeit in das Formular zaubert.

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click

    'Neue Schaltfläche instanzieren
    Dim neueSchaltfläche As New Button

    'Ein paar Eigenschaften setzen
    With neueSchaltfläche
        'Position:
        .Location = New Point(20, 20)
        'Größe
        .Size = New Size(150, 40)
        'Beschriftung:
        .Text = "Neue Schaltfläche"
        'Beschriftungsausrichtung:
        .TextAlign = ContentAlignment.MiddleCenter
    End With
    ' "Me" ist das Formular. Und sobald
    ' dessen Controls-Auflistung eine
    ' gültige Control-Instanz hinzugefügt
    ' wird, spiegelt sich das durch das Erscheinen des
    ' dahinter steckenden Steuerelement im Formular wider.
    Me.Controls.Add(neueSchaltfläche)
End Sub
```

---

**HINWEIS:** Wie die fehlende Assembly, war auch der fehlende Imports-Verweis eine gewollte Sabotage des Projektes meinerseits. Wenn Sie ein neues Windows Forms-Projekt erstellt hätten, wäre natürlich die Assembly-Referenz auf *System.Windows.Forms.dll* vorhanden gewesen. Auch der entsprechende Namespace *System.Windows.Forms* wäre importiert gewesen. Bei größeren Projekten, in denen Sie in weit größerem Umfang von der Framework-Klassenbibliothek machen werden, sind aber möglicherweise nicht alle erforderlichen Assembly-Referenzen oder Namespace-Importe von vornherein vorhanden. Aber jetzt wissen Sie ja, wie Sie sich in solchen Fällen helfen können.

---

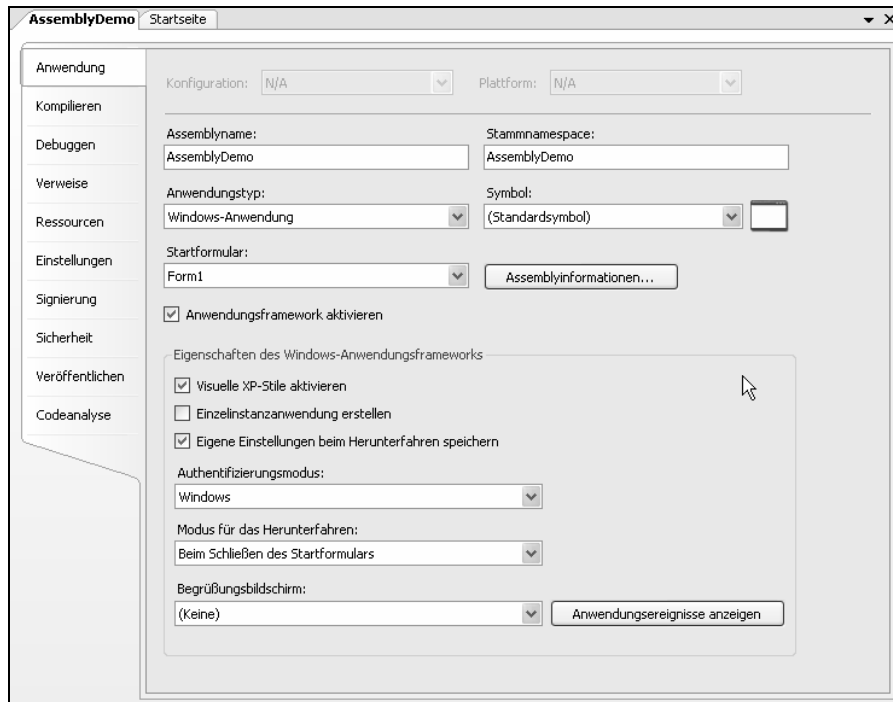
Um es übrigens nochmals zu wiederholen. Dass die *System.Windows.Forms.dll*-Assembly die Button-Klasse in einem Namespace definiert, der genau so heißt, wie die Assembly selbst, ist in diesem Fall zwar so, muss aber nicht so sein. So definiert dieselbe Assembly etwa auch eine *ResXResourceSet*-Klasse (es soll gar nicht interessieren, wozu die da ist), die sich aber im Namespace *System.Resources* befindet.

## So bestimmen Sie Assemblynamen und Namespace für Ihre eigenen Projekte

Genauso, wie bestimmte Assemblies des Frameworks bestimmte Namen tragen und ihre Klassen bestimmten Namespaces zuordnen, können Sie das auch mit Ihren eigenen Projekten machen.

Grundsätzlich trägt die Assembly, die aus Ihrem Projekt hervorgeht, den gleichen Namen, wie das Projekt selbst. Und das gilt gleichermaßen für den Namespace. Das muss aber nicht so sein. Verfahren Sie folgendermaßen, um Assembly-Namen oder Namespace-Namen für ein Projekt zu ändern:

- Rufen Sie dazu die Eigenschaften des Projektes ab (Menüpunkt *Projekt/AssemblyDemo-Eigenschaften*).



**Abbildung 5.17:** Unter *Assemblyname* definieren Sie den Namen der Assembly, der aus dem Projekt hervorgeht. Alle Klassen, die Ihre Assembly definiert, landen standardmäßig im unter *Stamnamespace* definierten Namespace

- Auf der Registerkarte *Anwendung* finden Sie die Felder *Assemblyname* und *Stammnamespace*, mit deren Hilfe Sie die entsprechenden Namen für Assembly und Namespace erfassen können.

## Verschiedene Namespaces in einer Assembly

Und zu guter Letzt: Wie das Beispiel der *System.Windows.Forms.dll* zeigt, können Assemblies unterschiedlichen Klassen verschiedene Namespaces zuweisen. Das funktioniert auch in Ihren eigenen Projekten. Wenn Sie das Beispielprojekt der vergangenen Abschnitte noch parat haben, nehmen Sie unterhalb des Codes von *Form1* folgende Änderungen vor:

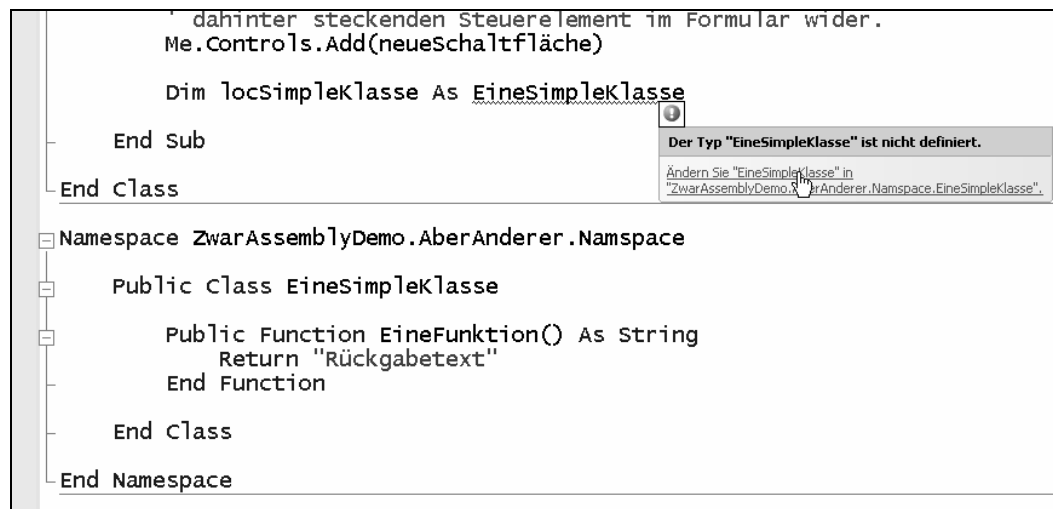
```
.
.
    ' dahinter steckenden Steuerelement im Formular wider.
    Me.Controls.Add(neueSchaltfläche)
End Sub
End Class
```

Namespace ZwarAssemblyDemo.AberAnderer.Namespace

```
Public Class EineSimpleKlasse
    Public Function EineFunktion() As String
        Return "Rückgabertext"
    End Function
End Class
```

End Namespace

Hier nun sehen Sie, wie Sie eine neue Klasse erstellen können, die innerhalb Ihrer Assembly aber in einem anderen Namespace liegt. Um diese Klasse, die zugegebenermaßen nicht das meiste kann, beispielsweise im Formularcode zu verwenden, gelten die gleichen Konventionen wie für externe Assemblies, die andere Namespace-Bereiche definieren: Sie müssen also entweder den vollqualifizierten Namen der Klasse eingeben, wollen Sie ein Objekt aus diesem Namespace verwenden, oder die Imports-Anweisung verwenden, um den Namespace in der Codedatei (oder einer anderen Code-datei des Projektes) zu verwenden.



**Abbildung 5.18:** Um Zugriff auf eine Klasse zu nehmen, die zwar in der gleichen Assembly, aber in einem anderen Namespace liegt, müssen Sie den vollqualifizierten Klassennamen oder *Imports* verwenden

