

Kapitel 6

Sprachsyntax Visual Basic 2010 (VB.NET 10.0) und C# 2010 (C# 4.0)

In diesem Kapitel:

Einleitung	305
Überblick über die Neuerungen	305
Allgemeines zu Visual Basic (.NET) 2010	306
Allgemeines zu C# 2010	309
Compiler	310
Datentypen	313
Operatoren	324
Klassendefinition	327
Felder (Field-Attribute)	329
Eigenschaften (Property-Attribute)	329
Methoden	331
Erweiterungsmethoden (Extension Methods)	335
Konstruktoren und Destruktoren	337
Objektinitialisierung	338
Beispiel für eine Klasse mit diversen Mitgliedern	339
Generische Klassen	340
Objektmengen	346 ►

In diesem Kapitel (Fortsetzung):

Partielle Klassen	348
Partielle Methoden	349
Anonyme Typen	351
Implementierungsvererbung	352
Ereignisse	356
Schnittstellen (Interfaces)	358
Namensräume (Namespaces)	359
Operatorüberladung	362
Schleifen	363
Verzweigungen	367
Funktionszeiger (Delegates)	368
Funktionale Programmierung mit Lambda-Ausdrücken	370
Annotationen (.NET-Attribute)	373
Fehlerbehandlung	374
Eingebaute Objekte und Funktionen	376
Kommentare und XML-Dokumentation	377
Zeigerprogrammierung (Unsicherer Code)	378
Abfrageausdrücke/Language Integrated Query (LINQ)	379
Vergleich: C# 2010 versus Visual Basic (.NET) 2010	379

Einleitung

Es ist ein wahrer Glaubenskrieg in der .NET-Entwicklergemeinde um die Wahl der »richtigen« Programmiersprache. C# oder Visual Basic heißt die Frage, die schon viele Projektteams bewegt hat. Immer scheint C# zu gewinnen, dabei ist Visual Basic gar nicht schlechter, sondern nur mit dem schlechten Ruf der Vergangenheit belastet.

Die Unterschiede zwischen Visual Basic und C# sind nämlich eher syntaktischer Natur; hinsichtlich der Möglichkeiten gibt es nur marginale Vorteile auf der einen oder anderen Seite. Dies gilt auch für die Geschwindigkeit. Da beide Sprachen MSIL-Code erzeugen, sind keine großen Unterschiede vorhanden. Der Glaubenskrieg, der um die Sprachwahl entbrannt ist, ist also eigentlich überflüssig. Details dazu lesen Sie am Ende des Kapitels.

Gerade durch .NET 4.0 rücken C# (jetzt in der Version 4.0) und Visual Basic (jetzt in der Version 10.0) noch enger zusammen.

HINWEIS

Zwei Hinweise vorweg:

1. Genau wie in der vorherigen Auflage dieses Buchs sind in dieser Auflage die Darstellung der Sprachsyntax der beiden vorherrschenden .NET-Programmiersprachen C# und Visual Basic .NET zu einem Kapitel zusammengefasst. Zu verschiedenen Unterthemen werden die Sprachfeatures direkt miteinander verglichen. Dies hat zwei Vorteile:
 - Die Unterschiede zwischen den Sprachen und damit der Wechsel zwischen den Sprachen fällt leichter.
 - Der neue Ansatz spart wertvolle Buchseiten, denn inhaltliche Wiederholungen in Fällen, wo die Sprachen gleich oder sehr ähnlich sind, werden vermieden.
2. Dieser Crashkurs bietet nicht die Möglichkeit, die Programmiersprachen komplett mit allen syntaktischen Feinheiten und mit detaillierten Beispielen zu beschreiben. Vorausgesetzt wird, dass Sie gute Kenntnisse in der objektorientierten Programmierung besitzen und eine objektorientierte Programmiersprache wie C++, Delphi, Java oder Visual Basic (im objektorientierten, nicht im prozeduralen Programmierstil!) bereits beherrschen.

Dieses Buch beschreibt nur kurz die in der Praxis wichtigsten Sprachmerkmale. Wenn Sie alle syntaktischen Details kennenlernen wollen, empfiehlt sich die Lektüre eines der zahlreichen Bücher zur Sprachsyntax, die in der Regel mehrere Hundert Seiten dick sind.

Überblick über die Neuerungen

Auch die Sprachversionen zählt Microsoft in .NET 4.0 hoch: Auf C# 4.0 und Visual Basic 10.0.

Dieser Abschnitt listet nur die Neuerungen auf. Die Besprechung erfolgt in den folgenden themenbezogenen Abschnitten.

Neu in beiden Sprachen ist:

- Ko- und Kontravarianz (für Delegates und Generische Mengen)

Neu in C# 4.0 ist:

- Dynamische Datentypen (siehe Abschnitt »Datentypen«)
- Optionale und benannte Parameter (siehe Abschnitt »Methoden«)

Neu in Visual Basic 10.0 ist:

- Implizite Zeilenfortsetzungen (siehe Abschnitt »Allgemeines zu Visual Basic (.NET) 2010«)
- Automatische Eigenschaften (siehe Abschnitt »Eigenschaften (Property-Attribute)«)
- Optionale Parameter dürfen wertelose Wertetypen sein (siehe Abschnitt »Methoden«)
- Typherleitung bei Arrays (siehe Abschnitt »Objektmengen«)
- Mengeninitialisierer (Collection Initializer, siehe Abschnitt »Objektmengen«)
- Mehrzeilige Lambda-Ausdrücke (siehe Abschnitt »Objektmengen«)
- Lambda-Ausdrücke ohne Rückgabewert (siehe Abschnitt »Funktionale Programmierung mit Lambda-Ausdrücken«)

Die neuen Sprachfeatures halten sich also in Grenzen (verglichen mit den großen Wellen, die es in .NET 2.0 und .NET 3.5 gab). Interessant ist die Aussage von Mads Torgersen, Produktmanager für C#, dass C# und Visual Basic in Zukunft hinsichtlich ihrer Funktionalität noch mehr im Gleichschritt gehen sollen als bisher. »Die Sprachen sollen sich in Stil und Gefühl unterscheiden, nicht in ihrem Funktionsumfang« [MT01].

Allgemeines zu Visual Basic (.NET) 2010

Visual Basic hat eine lange Tradition in der Windows-Entwicklerwelt. Die Sprache galt in den Versionen 1.0 bis 6.0 als einfach und sehr produktiv, war aber gleichzeitig auch als unsauber und unstrukturiert verpönt (gerade bei den Entwicklern, die mit C++, Delphi oder Java gearbeitet haben). Visual Basic 6.0 war allenfalls eine objektbasierte Sprache. Es fehlten insbesondere Konstrukte zur Implementierungsvererbung.

Mit dem Versionswechsel von Visual Basic 6.0 zur Version 7.0 hat Microsoft jedoch einen radikalen Wandel vollzogen. Die 7.0-Version wurde eine echte objektorientierte Sprache mit Vererbung, fast ebenso mächtig wie die zeitgleich neu entwickelte Sprache C#. Bis zur Version 9.0 gab es weitere Angleichungen, z.B. Operatorüberladung.

VB.NET folgt ab Version 7.0 der Common Language Specification (CLS) und ist daher eine mit dem .NET Framework kompatible Sprache. In Visual Basic 7.1 wurden nur Kleinigkeiten ergänzt. Visual Basic hat dann in den Versionen Version 8.0 (alias 2005) und 9.0 (alias 2008) nochmals eine erhebliche Aufwertung durch zusätzliche Sprachkonstrukte erfahren und kommt der Sprache C# noch näher. Die verbliebenen Unterschiede zwischen C# und Visual Basic sind primär syntaktischer Natur und für die meisten Problemszenarien unerheblich.

Die Sprache Visual Basic ist in der Version 10.0 Teil des .NET Framework 4.0. Der offizielle Produktname ist *Visual Basic 2010 (VB 2010)*. Auf die Sprache wird aber zum Teil auch als *Visual Basic 10.0 (VB10)* Bezug genommen.

.NET Framework	Version der Sprachsyntax mit Versionsnummer	Version der Sprachsyntax mit Jahreszahl	Interne Versionsnummer des C#-Compilers
1.0	Visual Basic .NET 7.0	Visual Basic .NET 2002	Visual Basic .NET 7.0
1.1	Visual Basic .NET 7.1	Visual Basic .NET 2003	Visual Basic .NET 7.1
2.0	Visual Basic 8.0	Visual Basic 2005	Visual Basic 8.0 ►

.NET Framework	Version der Sprachsyntax mit Versionsnummer	Version der Sprachsyntax mit Jahreszahl	Interne Versionsnummer des C#-Compilers
3.0	Visual Basic 8.0	Visual Basic 2005	Visual Basic 8.0
3.5	Visual Basic 9.0	Visual Basic 2008	Visual Basic 9.0
4.0	Visual Basic 10.0	Visual Basic 2010	Visual Basic 10.0

Tabelle 6.1 Versionsnummernzählungen für die Sprache Visual Basic .NET

Bereits seit Visual Basic .NET 7.0 ist Visual Basic eine echte objektorientierte Sprache mit Vererbung, die bis auf wenige Ausnahmen ebenso mächtig ist wie die Sprache C#. Microsoft hat bei VB.NET ganz bewusst auf die Kompatibilität mit der Vorgängerversion VB 6.0 verzichtet. Inkompatibilitäten wurden aus zwei Gründen hingenommen:

1. Alle Features aus VB 6.0, die signifikante Verwirrung bei den Entwicklern stifteten, wurden bereinigt.
2. Alle Features aus VB 6.0, die nicht kompatibel mit dem .NET Framework waren, wurden geändert oder entfernt.

Eine Aussage von COM- und .NET-Guru Don Box über VB7, die bereits aus dem Jahr 2001 stammt, sei hier zitiert: »Visual Basic .NET bedeutet, dass man sich nicht länger schämen muss, ein VB-Entwickler zu sein!«

Etwas konfus war die Namensgebung. Aus Visual Basic 6.0 wurde Visual Basic .NET 7.0. Jedoch hat Microsoft beginnend mit Version 8.0 in .NET 2.0 wieder auf den Zusatz *.NET* verzichtet, sodass die Versionen seitdem *Visual Basic 8.0* (alias *Visual Basic 2005*) und *Visual Basic 9.0* (alias *Visual Basic 2008*) hießen. Die offizielle Begründung war, dass es nun hinlänglich bekannt sei, dass die neuen Versionen .NET-basiert seien. Der Autor dieses Beitrags vermutet jedoch, dass dies eher passiert ist, um den Widerstand einiger Entwickler zu brechen, die nicht von Visual Basic 6.0 auf .NET umsteigen wollten. Befürchten kann man nur, dass – nachdem Microsoft eine Zeit lang die Buchstaben *.NET* auf alles geklebt hat, egal ob .NET drin war oder nicht – nun das Ruder in das andere unvernünftige Extrem umschlägt.

Seit dem Entfallen von *.NET* im Namen fällt es schwer, in der Schriftsprache zwischen dem »alten« Visual Basic (Versionen 1.0 bis 6.0) und dem neuen Visual Basic (Versionen ab 7.0) zu unterscheiden. Es bietet sich an, vom »klassischen Visual Basic« oder »COM-basierten Visual Basic« im Kontrast zum .NET-basierten Visual Basic zu sprechen. Auch »Visual Basic (.NET)« schreibe ich häufig.

HINWEIS In diesem Buch wird häufig aus Platzgründen die Abkürzung *VB* verwendet. *VB.NET* wird verwendet, wenn die Abgrenzung zu dem klassischen Visual Basic (Versionen 6.0 und früher) wichtig ist.

Die nachfolgende Abbildung zeigt das Verhältnis der existierenden Visual Basic-Dialekte hinsichtlich ihres Funktionsumfangs. Im Zuge von VB.NET hat Microsoft die drei Geschwister Visual Basic 6.0, Visual Basic for Applications (VBA) und VBScript zu einer Sprache fusioniert. Fusionsopfer in VB.NET sind nicht nur viele alte VB-Zöpfe (Goto etc.), sondern leider auch sinnvolle Funktionen zur Laufzeitkompilierung wie Eval und Execute aus VBScript.

HINWEIS Die Neugestaltung der Sprache Visual Basic ab Version 7.0 gefällt nicht allen Anwendern. Einige haben im Internet zu einer Petition an Microsoft aufgerufen, das alte Visual Basic 6.0 unter dem Namen *VB.COM* weiterzuentwickeln [ClassicVB01].

Das .NET Framework 3.0 enthielt weder eine neue Sprachsyntax noch eine neue Version des Sprach-Compilers. In .NET 3.0 galten weiterhin alle Aussagen zu Visual Basic 2005. Eine neue Sprachsyntax nebst Compiler (Visual Basic 9.0) ist zusammen mit dem .NET Framework 3.5 erschienen. In .NET 4.0 und Visual Basic 10.0 gibt es kleinere Aktualisierungen, insbesondere hinsichtlich »Parität« mit C#.

Die nachfolgende Abbildung spiegelt die Menge der Unterschiede zwischen den Versionen wieder.

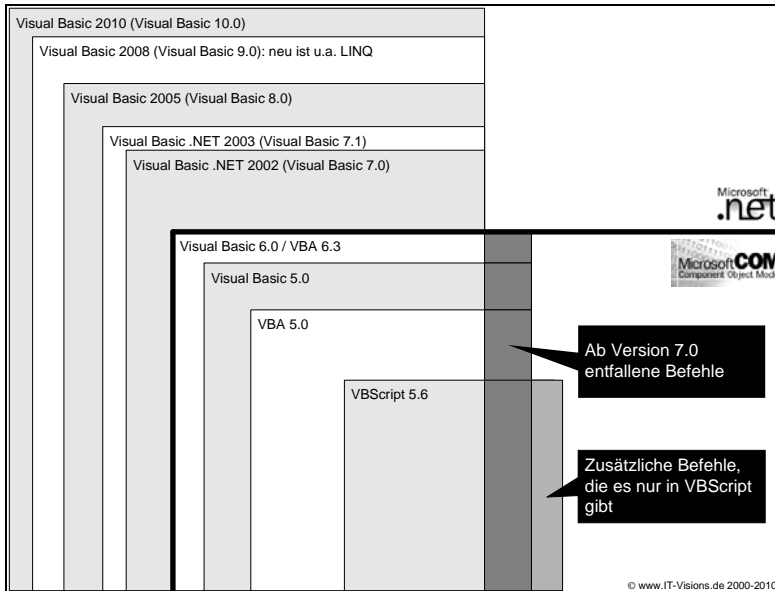


Abbildung 6.1 Vergleich der Visual Basic-Dialekte

Syntaktische Grundlagen

An den grundsätzlichen Syntaxregeln von Visual Basic hat sich seit VB6 nichts geändert:

- Die Sprache unterscheidet nicht zwischen Groß- und Kleinschreibung. Die Schlüsselwörter der Sprache werden aber üblicherweise mit einem großen Anfangsbuchstaben geschrieben.
- Blöcke werden durch End (z.B. End Sub, End Function) oder andere Schlüsselwörter wie Next und Loop abgeschlossen
- Befehle werden durch Zeilenumbruch (bevorzugt) oder einen Doppelpunkt (weniger verbreitet) getrennt
- Zwischen den Elementen eines Ausdrucks kann ein Zeilenumbruch eingefügt werden, wenn man die umzubrechende Zeile auf einen Unterstrich enden lässt

Neu ab Visual Basic 10.0 ist, dass in den nachfolgend gelisteten Fällen der Unterstrich nun auch weggelassen werden kann:

- Nach einer Annotation (Attribut)
- Nach einem Komma
- Nach einem Punkt
- Nach einem binären Operator

- Nach einer LINQ-Abfrage-Klausel
- Nach einem der folgenden Zeichen: ({ <%=
- Vor einem der folgenden Zeichen:) } %>

Zur Veranschaulichung beinhaltet das folgende Listing einige implizite Zeilenfortsetzungen.

```
Function ImplicitLineContinuation(  
ByVal einInteger As Integer,  
ByVal einAndererInteger As Integer  
) As Integer  
Return einInteger +  
einAndererInteger  
End Function
```

Listing 6.1 Beispiel mit impliziten Zeilenfortsetzungen

Objektorientierung

Bereits Visual Basic 6.0 unterstützte Konzepte der Objektorientierung wie Klassen, Objekte, Schnittstellen, Schnittstellenvererbung und einfache Formen des Polymorphismus. Da Visual Basic 6.0 aber keine Implementierungsvererbung unterstützte, konnte es nur das Prädikat *objektbasiert* bekommen. Seit Version 7.0 gilt Visual Basic jedoch als *objektorientierte* Sprache.

Allgemeines zu C# 2010

C# ist eine Programmiersprache, die mit .NET Framework neu entwickelt wurde. Das »#« könnte man auch in ein vierfaches Pluszeichen aufspalten (also C++++). Konzeptionell wurde C# vor allem von C++ und Java beeinflusst; man kann aber auch Parallelen zu Visual Basic und Delphi finden.

C# ist das Ergebnis eines Projekts bei Microsoft, das gestartet wurde, nachdem die Firma Sun Microsoft Ende der 1990er Jahre die Weiterentwicklung von J++, einer Microsoft-eigenen Anpassung der von Sun entwickelten Programmiersprache Java, verboten hatte. Ursprünglich sollte Microsofts neue Sprache dann *Cool* heißen. »Vater« von C# ist Anders Hejlsberg, der zuvor Entwickler von Turbo Pascal und Borland Delphi war.

Inzwischen ist C# standardisiert bei der ECMA (ECMA Standard 334, Arbeitsgruppe TC39/TG2) und bei der ISO (ISO/IEC 23270). Weitere Informationen zu den Standards finden Sie unter [ECMA01] und [MSDN07]. C# 4.0 ist aber noch nicht standardisiert. Die Standardisierung ist auf Stand C# 2.0.

Hinsichtlich der Versionsnummern der Sprache C# herrscht etwas Verwirrung. Es gibt einerseits eine offizielle Zählung mit Versionsnummer (parallel zum .NET Framework), andererseits mit Jahreszahlen (parallel zu Visual Studio). Intern wird eine dritte Zählung für den Compiler verwendet. Die erste Version von C# im Rahmen des .NET Framework 1.0 trug intern die Versionsnummer 7.0. Zu .NET 1.1 gab es dann C# 7.1, im .NET Framework 2.0 und 3.0 meldet sich der C#-Compiler mit Version 8.0. Ab .NET Framework 3.5 hat Microsoft dies aber bereinigt. Dort meldet sich der Compiler nun auch mit Version 3.5.

.NET Framework	Version der Sprachsyntax mit Versionsnummer	Version der Sprachsyntax mit Jahreszahl	Interne Versionsnummer des C#-Compilers
1.0	C# 1.0	Visual C# 2002	C# 7.0
1.1	C# 1.1	Visual C# 2003	C# 7.1
2.0	C# 2.0	Visual C# 2005	C# 8.0
3.0	C# 2.0	Visual C# 2005	C# 8.0
3.5	C# 3.0	Visual C# 2008	C# 3.5
4.0	C# 4.0	Visual C# 2010	C# 4.0

Tabelle 6.2 Verschiedene Versionsnummernzählungen für die Sprache C#

Syntaktische Grundlagen

Ein wesentlicher Unterschied zwischen C# und VB ist die Tatsache, dass C# im Gegensatz zu VB zwischen Groß- und Kleinschreibung unterscheidet. Dies gilt sowohl für die Schlüsselwörter der Sprache als auch für alle Bezeichner (a und A sind verschiedene Variablen!). Die Schlüsselwörter der Sprache C# werden komplett in Kleinbuchstaben geschrieben. Blockbildung findet im C/C++-Stil statt, also mit geschweiften Klammern { }. Befehlstrenner ist das Semikolon (;). Ein Zeilenumbruch kann zwischen den Elementen des Ausdrucks auftreten, ohne dass besondere Vorkehrungen getroffen werden müssen.

Objektorientierung

Im Gegensatz zu C++, das eine hybride Sprache aus objektorientierten und nicht-objektorientierten Konzepten ist, ist C# ebenso wie Java eine rein objektorientierte Sprache, d. h., jegliche Form von Anwendungen basiert auf Klassen. C# unterstützt alle zentralen Konzepte der Objektorientierung einschließlich Schnittstellen, Vererbung und Polymorphismus. In C# 2005 wurde die Unterstützung für generische Klassen und partielle Klassen hinzugefügt.

Compiler

Die Kommandozeilencompiler für C# und Visual Basic sind Teil des .NET Framework Redistributable. Die Compiler können auch über die .NET-Klassenbibliothek angesprochen werden.

TIPP

Einige der neuen Spracheigenschaften von C# 3.0 und Visual Basic 9.0 sind auch dann verfügbar, wenn man Visual Studio mit Target Framework *.NET 2.0* oder *.NET 3.0* laufen lässt. Wie bereits im Kapitel 4 »Grundkonzepte des .NET Framework 4.0« erläutert, verwendet Visual Studio 2008 immer die Version 9.0 des C#-Compilers und des Visual Basic-Compilers. Im Untergrund arbeitet aber immer die CLR 2.0, d. h. die neuen Spracheigenschaften kann man komplett in MSIL 2.0 übersetzen und damit auch auf dem .NET Framework 2.0/3.0 betreiben. Die Möglichkeit, die neuen Spracheigenschaften auch in älteren Versionen des Frameworks zu nutzen, ist jedoch nicht von Microsoft dokumentiert. Nicht alle Spracheigenschaften sind auf diesem Wege verfügbar, da z. B. LINQ auch das Einbinden einer speziellen DLL (*System.Core.dll*) erfordert, die wiederum andere DLLs erfordert.

C#-Compiler

Der Kommandozeilencompiler für C# im .NET Framework Redistributable ist *csc.exe*. Er kann in der Klassenbibliothek durch die Klasse `Microsoft.CSharp.CSharpCodeProvider` angesprochen werden.

Der Befehl

```
csc Dateiname1.cs Dateiname2.cs DateinameX.cs
```

übersetzt die angegebenen Dateien in eine Konsolenanwendung. Eine Datei, die als Konsolenanwendung oder Windows-Anwendung kompiliert wird, muss genau eine Klasse mit folgendem Einstiegspunkt besitzen: `public static void Main()`.

```
class Hauptprogramm
{
    public static void Main()
    {
        System.Console.WriteLine("Hello World!");
    }
}
```

Listing 6.2 »Hello World« in C#

Der Kommandozeilencompiler bietet zahlreiche Optionen. Die wichtigsten davon sind:

- **/target:winexe** Der Compiler erzeugt eine Windows-Anwendung
- **/target:library** Der Compiler erzeugt eine DLL (kein `Main()` notwendig)
- **/r:Dateiliste** Die angegebenen Assemblies werden referenziert
- **/out:Dateiname** Name der Ausgabedatei
- **/doc:Dateiname** Der Compiler erzeugt zusätzlich eine XML-Dokumentationsdatei
- **/help** Anzeige der Hilfe zu den Compiler-Optionen

Anders als beim VB-Compiler *vbc.exe* müssen die Optionen `/target` und `/out` bei *csc.exe* vor den Namen der Quelldateien in der Parameterliste erscheinen.

HINWEIS Es gibt eine in C++ geschriebene Shared Source-Version des C#-Compilers im Rahmen der ECMA-Referenzimplementierung *Rotor* [MSDN07]. Mono [MON001] bietet seit der ersten Version einen C#-Compiler, denn C# ist die Entwicklungssprache für die Mono-Bibliotheken. Auch dieser Compiler ist in C++ geschrieben.

Visual Basic .NET-Compiler

Der Compiler für Visual Basic .NET kann über das Kommandozeilenwerkzeug *vbc.exe* oder sogar per Programmcode aus der .NET-Klassenbibliothek über die Klasse `Microsoft.VisualBasic.VBCodeProvider` aufgerufen werden. Er erzeugt immer Managed Code und bietet keine Option zur Erzeugung von Unmanaged Code.

Den Einsprungpunkt definiert man mit `Sub Main()` in einem Modul:

```
Module Hauptprogramm
    Sub Main()
        System.Console.WriteLine("Hello World!")
    End Sub
End Module
```

Listing 6.3 »Hello World« in Visual Basic .NET

Die Übersetzung erfolgt mit

`vbc Dateiname1.vb Dateiname2.vb DateinameX.vb`

Die o.g. Kommandozeilenparameter für den C#-Compiler gibt es für `vbc.exe`.

Neu im Compiler seit VB 2005 im Vergleich zu VB 7.x ist, dass er die Kompatibilität mit der Common Language Specification (CLS) überprüft und warnt, wenn Sie Typen (z.B. `UInteger`) verwenden, die nicht CLS-konform sind. Außerdem prüft der VB-Compiler nun genau wie der C#-Compiler, ob deklarierte Variablen überhaupt jemals verwendet werden und ob verwendete Variablen bereits initialisiert sind.

Auch wenn die Hintergrundkompilierung des C#-Compilers deutlich besser geworden ist: Der VB Compiler ist in der 2010 (!)-Version dem C#-Compiler immer noch überlegen.

Viele Fehler werden unterschlängelt (siehe Abbildung) und in der Aufgabenliste angezeigt. Korrigierte Fehler werden sofort aus der Liste entfernt. In C# ist in beiden Fällen immer eine explizite Kompilierung notwendig, was das Codieren zeitaufwändiger macht.

```
Dim a As Byte
a = 3434344
Constant expression not representable in type 'Byte'.
```

Abbildung 6.2 Hintergrundkompilierung für VB in Visual Studio

HINWEIS Der Visual Basic-Compiler erzeugt immer automatisch eine Referenz auf die Assembly `Microsoft.VisualBasic.dll`. Seit VB 9.0 ist es möglich, auf diese Referenz zu verzichten. Dies erreicht man durch den Compilerparameter `/vbruntime-321`

. Mithilfe von `/vbruntime:abc.dll` kann man dann eine eigene Implementierung der zur Laufzeit benötigten Klassen und Methoden im Namensraum `Microsoft.VisualBasic.CompilerServices` angeben. Microsoft nennt diese Möglichkeit *Runtime Agility*. Der Charme von *Runtime Agility* liegt darin, dass man nur die im konkreten Fall wirklich benötigten Methoden implementieren kann. Die meisten Entwickler werden allerdings keine Zeit und Lust für eine solche eigene Implementierung haben.

Mono bot nicht von Anfang an eine Unterstützung für Visual Basic .NET. Doch seit Version 1.2.3 ist der Compiler fertig gestellt [MONO07]. In Rotor gibt es keinen Visual Basic .NET-Compiler.

Datentypen

Die Datentypen orientieren sich in allen .NET-Programmiersprachen am Typsystem des Common Type System (CTS). Erläuterungen dazu finden Sie bereits im Kapitel 4 »Grundkonzepte des .NET Framework 4.0«.

Art	Visual Basic	C#	Visual J#	JScript .NET	Visual C++
Ganzzahl 1 Byte	Byte	byte	byte	byte	BYTE, bool
Ganzzahl Boolean	Boolean	bool	boolean	boolean	VARIANT_BOOL
Ganzzahl 2 Bytes	Short	short	short	short	signed short int, __int16
Ganzzahl 4 Bytes	Integer	int	int	int	long, (long int, signed long int)
Ganzzahl 8 Bytes	Long	long	long	long	__int64
Zahl 4 Bytes	Single	float	float	float	float
Zahl 8 Bytes	Double	double	double	double	double
Zahl 12 Bytes	Decimal	decimal	–	decimal	DECIMAL
Zeichen 1 Byte oder 2 Bytes	Char	char	char	char	signed char, __int8
Zeichenkette	String	string	java.lang.String oder System.String	String	n/a
Datum/Uhrzeit	Date	DateTime	java.util.Date oder System.DateTime	Date	DATE

Tabelle 6.3 Vergleich der Datentypen in verschiedenen .NET-Sprachen

Datentypen in VB

Variablendeklarationen erfolgen in VB.NET wie in VB6 mit Dim und As. Bei Variablendeklarationen können nun mehrere Variablen eines Typs in einer Zeile ohne Wiederholung des Datentyp-Schlüsselworts deklariert werden. In VB6 musste das für jede Variable wiederholt werden.

Wenn mehrere Variablen in einer Zeile deklariert werden, muss im Gegensatz zu VB6 der Typ nicht hinter jeder Variablen genannt werden.

```
Dim x as String
Dim a,b,c as Integer
```

Hinter einer Variablendeklaration kann direkt die Initialisierung mit einem Wert erfolgen.

```
Dim Name as String = "Holger Schwichtenberg"
```

Alle Datentypen wurden bereits mit VB7 dem .NET-Typsystem angepasst. So umfasst beispielsweise der Datentyp Integer 32 statt 16 Bit wie in VB6. Dafür vergrößert sich Long auf 64 Bit. Außerdem gibt es neue Typen, z.B. Char, Short und Decimal. Short (16 Bit), Integer (32 Bit) und Long (64 Bit) erlauben negative Zahlen, während Byte (8 Bit) nur positive Zahlen speichern kann. Mit VB 2005 wird das Typsystem mit den vorzeichenlosen Datentypen UShort (16 Bit), UInteger (32 Bit) und ULong (64 Bit) sowie dem vorzeichenbehafteten Byte (SByte) komplettiert.

In VB.NET wird zwischen einer Zeichenkette und einem einzelnen Zeichen (Datentyp Char) unterschieden. Um ein Literal vom Typ Char zu erzeugen, muss dem schließenden Anführungszeichen ein kleines »c« folgen, z.B. "A"c.

Das Schlüsselwort Variant wurde mit VB7 abgeschafft. Der Datentyp System.Object übernimmt die Rolle von Variant. Die aus VB6 bekannten Zustände Empty und Null für Variablen werden nicht mehr unterstützt. An deren Stelle tritt das Schlüsselwort Nothing.

Notwendigkeit der Typisierung

In VB.NET ist die Deklaration von Variablen im Standard Pflicht (kann aber zur Option gemacht werden, indem Option Explicit Off gesetzt wird).

TIPP

Option Explicit Off sollten Sie nicht nutzen, weil dadurch der Programmcode anfälliger für Fehler wird.

Die Typisierung ist aber in VB.NET von Hause aus etwas weniger streng als in C#.

Die beiden folgenden Zuweisungen akzeptiert der Compiler, zur Laufzeit steht das Programm dann aber in der zweiten Zeile mit einer InvalidCastException.

```
Dim s As String = 5
Dim i As Int16 = "5s"
```

Das vermeidet man aber einfach, indem man den Visual Basic-Compiler in den strengeren Modus schaltet (*Option Strict*), siehe folgende Bildschirmabbildung. In C# gibt es nur diesen strengen Modus.

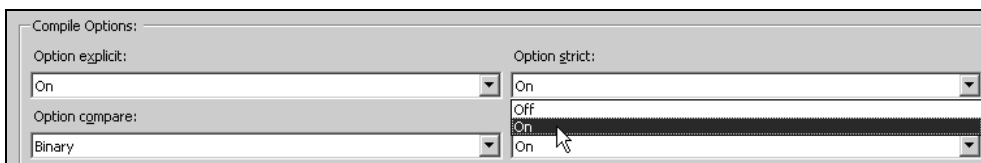


Abbildung 6.3 Aktivieren der strengeren Typprüfung im Visual Basic-Compiler

XML-Literale in VB

Visual Basic bietet ab Version 2008 eine einfache Möglichkeit, XML-Fragmente zu definieren und zu verwenden. Dabei hinterlegt man XML-Code ganz ohne Anführungszeichen im VB-Code. Diese Funktion nennt man XML-Literale und sie existiert nur in VB, nicht in C#.

```

Dim e As XElement = _
<Flug ID="347">
  <Abflugort>Madrid</Abflugort>
  <Zielort>Paris</Zielort>
  <FreiePlaetze>1</FreiePlaetze>
  <Details>
    <Nichtraucher>True</Nichtraucher>
    <Plaetze>250</Plaetze>
    <EingerichtetAm/>
  </Details>
</Flug>

```

Listing 6.4 Beispiel für ein XML-Literal in VB 2008

Im Programmcode kann man dieses XML-Literal dann ebenso einfach verwenden. Elemente werden mit <> angesprochen und Attribute mit @:

```

' Zugriff auf ein Attribut
Console.WriteLine(x.@ID)
' Zugriff auf Elemente
Console.WriteLine(x.<Abflugort>.Value) ' Madrid
Console.WriteLine(x.<Zielort>.Value) ' Paris
Console.WriteLine(x.<Details>.<Plaetze>.Value) ' 250
Console.WriteLine(x.<Passagiere>.<Passagier>(1).Value) ' = Meier!
' Schleife über Elemente
For Each p As XElement In x.<Passagiere>.<Passagier>
  Console.WriteLine(p.Value)
Next

```

Listing 6.5 Beispiel zur Verwendung eines XML-Literals in VB 2008

HINWEIS Voraussetzungen sind die Referenzierung der Assemblys *System.Core.dll* und *System.Xml.Linq.dll* sowie der Namensraum-Import `System.Xml.Linq`.

Visual Studio 2008 bietet IntelliSense für XML-Elemente, wenn man ein entsprechendes XML-Schema einbindet (siehe Kapitel zu 5 Visual Studio 2010).

Weitere Beispiele für XML-Literale finden Sie im Zusammenhang mit LINQ to XML im Kapitel zu »Datenzugriff mit System.Xml und LINQ to XML«.

Datentypen in C#

C# unterstützt die gleichen Datentypen wie Visual Basic .NET, teilweise jedoch mit etwas anderen Namen, beispielsweise `int` statt `Integer` oder `bool` statt `Boolean` (siehe obige Tabelle). In C# steht der Typ am Anfang jeder Deklaration. Mehrfachdeklarationen sind möglich durch Kommatrennung.

```

int a, b, c;
string x, y, z;
System.Guid g1, g2, g3;

```

Eine mit Option `Explicit Off` vergleichbare Möglichkeit gibt es in C# (zum Glück!) nicht.

TIPP

Sonderzeichen in Zeichenketten werden – wie in C++ – durch einen Backslash (\) eingeleitet (z. B. steht \n für einen Zeilenumbruch). Zu Schwierigkeiten kommt es bei Windows-Pfadangaben. In Pfaden ist entweder jeder \ durch einen doppelten \\ zu ersetzen oder aber der Zeichenkette ist ein @ voranzustellen. Synonym sind daher: "c:\\ordner\\datei.txt" und @"c:\ordner\datei.txt".

Lokale Typableitung (Local Variable Type Inference)

In VB9 und C# 3.0 wurde die Typableitung neu eingeführt. *Typableitung* bedeutet, dass der Entwickler in seinem Programmcode keinen expliziten Typ vergibt, sondern der Compiler den Typ während der Übersetzung festlegt. Typableitung darf nicht mit *Variant* aus Visual Basic 5.0/6.0 verwechselt werden (auch wenn in C# 2008 das Schlüsselwort *var* heißt): Bei einem Variant konnte man jederzeit im Programmablauf den Typ ändern und ein Variant war eine sehr speicherfressende Datenstruktur. Variablen, die mit Typableitung erzeugt wurden, erhalten hingegen zur Übersetzungszeit einen festen Typ, der im Programmablauf nicht mehr geändert werden darf und verbrauchen nicht mehr Speicher als bei einer expliziten Deklaration.

HINWEIS

Die Typableitung heißt *lokal*, weil sie nur für lokale Variablen in Methoden möglich ist. Ein Einsatz als Attribut einer Klasse bzw. Parameter oder Rückgabewert einer Methode ist ausgeschlossen. Eine Typableitung muss immer mit einer Wertinitialisierung verbunden sein, da sonst keine Typableitung möglich ist. *null* bzw. *nothing* ist nicht erlaubt, da hier keine Typableitung möglich ist.

Man kann die Typableitung auch für Laufvariablen in Schleifen verwenden.

WICHTIG

Bei vielen Entwicklern herrscht zunächst Skepsis über den Sinn der lokalen Typableitungen. Tatsächlich machen Typableitungen für sich isoliert betrachtet nur einen begrenzten Sinn. Typableitungen sind jedoch absolut notwendig im Zusammenhang mit anonymen Typen (später in diesem Kapitel) und LINQ-Projektionen (siehe eigenes Kapitel 10 zu »Language Integrated Query (LINQ)«). In beiden Szenarien entstehen Klassen, deren Namen der Entwickler nicht kennen kann.

Man darf Typableitung nicht mit dem Einsatz der allgemeinen Klasse `System.Object` verwechseln. Eine mit `System.Object` (alias `object` oder `Object`) deklarierte Variable kann tatsächlich im Programmablauf verschiedenartigste Inhalte aufnehmen. Eine mit lokaler Typableitung deklarierte Variable hingegen hat einen festen, unveränderbaren Typ.

Typableitung in VB

Typableitungen werden in Visual Basic .NET durch `Dim` ohne Datentypangabe, aber mit Initialisierung festgelegt.

```
' Local Variable Type Inference für String
Dim Heimatflughafen = "Essen/Mülheim"
Console.WriteLine(Heimatflughafen.GetType().FullName)

' Local Variable Type Inference für Int32
Dim Anzahl = Vorstandsmitglieder.Count
Console.WriteLine(Anzahl.GetType().FullName)

' Local Variable Type Inference für die Klasse Vorstandsmitglied
Dim Vorstandvorsitzender = Vorstandsmitglieder(0)
Console.WriteLine(Vorstandvorsitzender.GetType().FullName)
```

Listing 6.6 Drei Typableitungen in VB

Typableitung in C#

Typableitungen werden in C# durch das neue Schlüsselwort `var` anstelle des Datentyps, aber mit Initialisierung festgelegt.

```
// Local Variable Type Inference für String
var Heimatflughafen = "Essen/Mülheim";
Console.WriteLine(Heimatflughafen.GetType().FullName);

// Local Variable Type Inference für Int32
var Anzahl = Vorstandsmitglieder.Count;
Console.WriteLine(Anzahl.GetType().FullName);

// Local Variable Type Inference für die Klasse Vorstandsmitglied
var Vorstandschef = Vorstandsmitglieder[0];
Console.WriteLine(Vorstandschef.GetType().FullName);
```

Listing 6.7 Drei Typableitungen in C#

TIPP Gerade bei der Objektinstanziierung in C# kann man durch die Typableitung die überflüssige Doppelnennung des Klassennamens vermeiden, denn man schreibt nun statt

```
Vorstandsmitglied v1 = new Vorstandsmitglied();
```

kürzer:

```
var v2 = new Vorstandsmitglied();
```

Die neue Schreibweise hat keinen Nachteil!

```
Vorstandsmitglied v1 = new Vorstandsmitglied();
Vorstandsmitglied v2 = new Vorstandsmitglied();
```

Abbildung 6.4 Beim Betrachten mit dem Decompiler .NET Reflector sieht man, dass der Compiler beide Zeilen gleich übersetzt hat

Dynamische Typisierung

Dynamische Typisierung bedeutet, dass die Einsprungsstelle für einen Attributzugriff oder einen Methodenaufruf nicht zur Kompilierzeit feststeht (statische Typisierung), sondern erst zur Laufzeit ermittelt wird. Grundsätzlich ist statische Typisierung erstrebenswert, aber nicht immer ist dies möglich. Unmöglich ist die statische Typisierung zum Beispiel bei der Verwendung von COM-Bibliotheken, die als Datentypen Variant verwenden. Oder beim Zusammenspiel mit dynamischen Sprachen wie IronPython.

ACHTUNG Bei dynamischer Typisierung kann Visual Studio keine IntelliSense-Eingabeunterstützung bieten. Dynamische Typisierung birgt immer die Gefahr, dass die entsprechende Aktion nicht verfügbar ist, sei es durch einen Tippfehler oder weil ein anderes Objekt geliefert wird, als erwartet wurde. Wenn die Bindung nicht möglich ist, kommt es zum Laufzeitfehler (`RuntimeBinderException`).

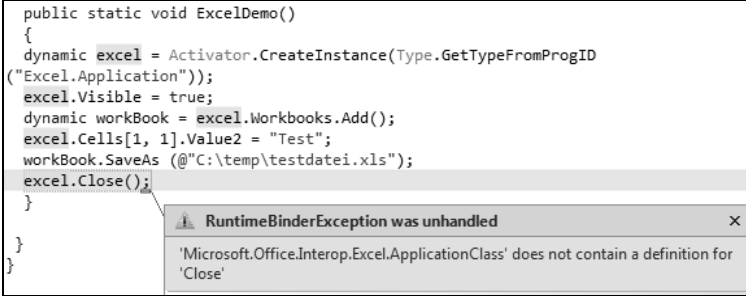


Abbildung 6.5 Laufzeitfehler, denn die Methode zum Schließen wäre Quit() statt Close() gewesen

Dynamische Typisierung in VB

In Visual Basic .NET ist die dynamische Typisierung schon seit Version 7.0 einfach möglich. In VB.NET wird die dynamische Typisierung durch den Typ object ermöglicht.

```

Imports Microsoft.VisualBasic

Public Class CS10_Dynamic
    ''' <summary>
    ''' Beispiel für dynamische Nutzung einer COM-Bibliothek (hier: Microsoft Excel)
    ''' </summary>
    Public Shared Sub ExcelDemo()
        Dim excel As Object = CreateObject("Excel.Application")
        excel.Visible = True
        Dim workbook As Object = excel.Workbooks.Add()
        excel.Cells(1, 1).Value2 = "Test"
        workbook.SaveAs("C:\temp\testdatei.xls")
        excel.Quit()
    End Sub
End Class

```

Listing 6.8 Verwendung der dynamischen Typisierung in C# [VB10_Dynamic.vb]

Dynamische Typisierung in C# (neu in C# 4.0)

In C# wurde die dynamische Typisierung erst in C# 4.0 auf einfache Weise ermöglicht. Vorher musste man sehr umständlich mit dem .NET-Reflection-Mechanismus arbeiten. C# 4.0 bietet dafür das Schlüsselwort `dynamic`.

Um `dynamic` in C# zu nutzen, muss man die Assembly `Microsoft.CSharp.dll` referenzieren. Sonst kommt es zum Fehler »Predefined type 'Microsoft.CSharp.RuntimeBinder.Binder' is not defined or imported.«

```

/// <summary>
/// Beispiel für dynamische Nutzung einer COM-Bibliothek (hier: Microsoft Excel)
/// </summary>
public static void ExcelDemo()
{
    dynamic excel = Activator.CreateInstance(Type.GetTypeFromProgID("Excel.Application"));
    excel.Visible = true;
    dynamic workbook = excel.Workbooks.Add();
    excel.Cells[1, 1].Value2 = "Test";
}

```



```
workBook.SaveAs (@\"C:\temp\testdatei.xls");  
excel.Quit();  
}
```

Listing 6.9 Verwendung der dynamischen Typisierung in C# [CS10_Dynamic.cs]

Typkonvertierung

Typkonvertierung bezeichnet die Umwandlung von einem Datentyp in einen anderen, z.B. Umwandeln einer Zahl in eine Zeichenkette oder Extrahieren einer Zahl aus einer Zeichenkette.

HINWEIS Typkonvertierungen sind in der .NET-Klassenbibliothek hinterlegt, insbesondere in der Klasse `System.Convert`. Darüber hinaus bieten alle Klassen für Zahlen (`System.Byte`, `System.Int16`, `System.Int32`, etc.) die Möglichkeit, eine Zahl aus einer Zeichenkette zu extrahieren mithilfe der Methoden `Parse()` und `TryParse()`, siehe dazu Kapitel 9 zur .NET-Klassenbibliothek 4.0. Hier in diesem Kapitel werden sprach eigene Konvertierungsoptionen genannt, die intern zum Teil auf die gleichen FCL-Klassen abgebildet werden.

Typkonvertierung in VB

In der Standardeinstellung verwendet Visual Basic nur eine schwache Typprüfung, d.h., die Zuweisung einer Zahl an eine Zeichenkettenvariable wird nicht bemängelt.

```
Dim zahl As Integer = 1  
Dim text As String = zahl
```

Seit VB 7.0 existiert eine Compiler-Option, die eine strikte Typprüfung erzwingt. Mit der Option `Strict On` müssen alle Typkonversionen explizit ausgeführt werden. Implizite Typkonversionen führen zu einem Fehler. Wenn Sie die neue Option `Strict` nutzen, müssen alle Variablen zwingend typisiert werden (mit `As...`).

Typkonvertierungen können mit verschiedenen eingebauten (schon vor dem .NET-Zeitalter verwendeten) Funktionen ausgeführt werden, die mit einem großen C beginnen, z. B.

- `i = CLng(s)`
- `s = CStr(i)`
- `pass = CType(pers, Passagier)`

`CType` ist dabei die allgemeinste Variante, da man hier einen Zieltyp angeben kann. Selbstverständlich sind nur Konvertierungen zwischen Klassen möglich, wenn diese in einer Vererbungsbeziehung zueinander stehen.

Typkonvertierung in C#

In C# kommt immer eine sehr strenge Typprüfung zum Einsatz, wohingegen sie in VB explizit eingeschaltet werden muss. Für

```
int zahl = 1;
```

sind folgende Konstrukte nicht gültig:

```
// falsch: string text = zahl;  
// falsch: string text = ((string) zahl);  
// falsch: string text = zahl as string;
```

Die Konvertierung von Zahl zu Text ist nur möglich über die ToString()-Methode oder über die FCL-Klasse System.Convert.

```
string text1 = zahl.ToString();  
string text2 = Convert.ToString(zahl);
```

Zwischen polymorphen Klassen gibt es zwei Syntaxformen für die Typumwandlung:

```
pass = ((Passagier)a[0]);  
pass = (a[0] as Passagier);
```

Der Unterschied zwischen der Schreibweise mit dem vorangestellten Typnamen und der Verwendung des as-Operators ist, dass in dem ersten Fall eine Ausnahme (InvalidCastException) erzeugt wird, wenn die Konvertierung nicht möglich ist, während der as-Operator in diesem Fall null zurückliefert.

Typinitialisierung

Ebenfalls sehr streng ist C# hinsichtlich der Initialisierung von Variablen. Während der VB 2005-Compiler in seiner Standardeinstellung folgende Anweisung immer durchgehen lässt,

```
Dim a As Integer  
a = a + 1
```

weil a mit 0 vorinitialisiert wurde, erfordert der C#-Compiler die explizite Initialisierung bei allen lokalen (methodeninternen) Variablen (nicht aber bei Klassenmitgliedern).

```
int a = 0;  
a = a + 1;
```

HINWEIS

Für die gebrochenen Zahlen gibt es in C# besondere Kürzel, die in Literalen zu verwenden sind:

```
float x = 0.12345f;  
double x = 0.12345d;  
Decimal x = 0.12356m;
```

Der C#-Compiler und der VB.NET-Compiler ab Version 2005 erzeugen Warnungen bei deklarierten, aber nicht verwendeten Variablen.

Wertelose Wertetypen (Nullable Types)

Während Referenztypen bereits in .NET 1.x den Zustand `null` (alias *nothing*) als Repräsentanz des Zustands *nicht vorhanden/nicht gesetzt* annehmen konnten, war dies für Wertetypen nicht vorgesehen. Ab .NET 2.0 existiert ein Hilfskonstrukt, um auch Wertetypen den Wert `null` zuweisen zu können.

In .NET (ab Version 2.0) ist ein auf `null` setzbarer Wertetyp eine generische Struktur (`System.Nullable`), die aus dem eigentlichen Wert (`Value`) und einem Hilfs-Flag `HasValue` (`Typ boolean`) besteht, das anzeigt, ob der Wert des Typs `Null` ist.

Wertelose Wertetypen in VB

Visual Basic in der Version 2005 enthielt nur eine rudimentäre Unterstützung für wertelose Wertetypen (vgl. Erläuterungen im Kapitel 4 »Grundkonzepte des .NET Framework 4.0«), die weit weniger elegant ist als die Unterstützung in C# (siehe nächstes Hauptkapitel). Dies wurde in Visual Basic 2008 geändert, sodass die Ergebnisse in VB jetzt fast Äquivalent zu denen in C# sind.

Durch ein Fragezeichen als Suffix eines Wertetyps in einer Typdeklaration sorgt der VB 2008-Compiler automatisch dafür, dass der Wertetyp in die generische `System.Nullable`-Struktur verpackt wird. Möglich ist auch eine explizite Deklaration mit `System.Nullable`.

```
' Wertetyp ohne Nothing
Integer a = 1
Integer b = 0
' Wertetyp mit Nothing erlaubt
Integer? x = 2;
System.Nullable(Of Integer) y = 6
```

Die folgende Tabelle zeigt verschiedene Ergebnisse für Operationen mit den obigen Variablen.

Operation	Ergebnis, wenn x den Wert 2 hat	Ergebnis, wenn x <i>nothing</i> ist
<code>Dim s1 As string = x.HasValue.ToString()</code>	True	False
<code>Dim s2 As string = x</code>	X	Laufzeitfehler
<code>Dim s3 As string = x.Value.ToString()</code>	X	Laufzeitfehler
<code>Dim s4 As string = x.ToString()</code>	X	Leere Zeichenkette
<code>Dim z As System.Nullable(Of Integer) = x + 10</code>	12	Nothing
<code>Dim a1 As Integer = x</code>	2	Laufzeitfehler
<code>Dim a2 As Integer = CType(y, Integer)</code>	2	Laufzeitfehler

Tabelle 6.4 Verschiedene Operationen mit wertelosen Wertetypen in Visual Basic 2008

TIPP In zwei Fällen bemerkt VB das Problem erst zur Laufzeit. Dies können Sie jedoch verbessern, indem Sie `Option Strict On` setzen.

HINWEIS Bereits Visual Basic 7.x erlaubte die Zuweisung `Dim b As Integer = Nothing`. Dies war gleichbedeutend mit `Dim b As Integer = 0`.

Wertelose Wertetypen in C#

C# unterstützt *Nullable Value Types* bereits seit Version 2005 durch ein besonderes Sprachkonstrukt: Durch ein Fragezeichen als Suffix eines Wertetyps in einer Typdeklaration sorgt der C#-Compiler automatisch dafür, dass der Wertetyp in die generische `System.Nullable`-Struktur verpackt wird. Möglich ist auch eine explizite Deklaration mit `System.Nullable`.

```
// Wertetyp ohne Null
int a = 1;
int b = 0;
// Wertetyp mit Null erlaubt
int? x = 2;
System.Nullable<Int32> y = 6;
```

Die folgende Tabelle zeigt verschiedene Ergebnisse für Operationen mit den obigen Variablen.

Operation	Ergebnis, wenn x den Wert 2 hat	Ergebnis, wenn x null ist
<code>string s1 = x.HasValue.ToString();</code>	True	False
<code>string s2 = x;</code>	X	Kompilierungsfehler
<code>string s3 = x.Value.ToString();</code>	X	Laufzeitfehler
<code>string s4 = x.ToString();</code>	X	Leere Zeichenkette
<code>int? z = x + 10;</code>	12	Null
<code>int a1 = x;</code>	2	Kompilierungsfehler
<code>int a2 = (int)x;</code>	2	Laufzeitfehler
<code>int a3 = x ?? 0;</code>	True	0

Tabelle 6.5 Verschiedene Operationen mit wertelosen Wertetypen in C# 2005 und C# 2008

```
public void NullableTypes()
{
    int a = 1;
    // Elegante Deklaration in C#
    int? b = 2;
    // a = null; // verboten!
    b = null; // Erlaubt
    // Explizite Deklaration
    System.Nullable<Int32> c = null;
    c = 100;
    Demo.Print(c.HasValue.ToString());
    Demo.Print(c.Value.ToString()); // Achtung: Geht nur, wenn c tatsächlich einen Wert hat!
    // Besser: "Null" abfangen
    Demo.Print ("b = " + ( b.HasValue ? b.Value.ToString() : "null"));
}
```

Listing 6.10 Verschiedene Beispiele mit Nullable Types

Vergleich

Die Tabelle zeigt einen zusammenfassenden Vergleich der Möglichkeiten für wertelose Wertetypen.

	C#	Visual Basic
Deklaration eines normalen Wertetyps	<code>int a;</code>	<code>Dim a As Integer</code>
Zuweisung des <i>nicht vorhandenen</i> an einen normalen Wertetyp	Nicht möglich (Kompilierungsfehler)	<code>a = nothing</code> setzt den Wert auf die Zahl 0 bzw. anderen Startwert (z. B. <code>DateTime.MinValue</code>)
Deklaration eines wertelosen Wertetyps in Langform	<code>System.Nullable<Int32> x = null</code>	<code>Dim x As System.Nullable(Of Integer) = Nothing</code>
Deklaration eines wertelosen Wertetyps in Kurzform	<code>int? x = null;</code>	<code>Integer? x = nothing;</code>
Ausdruck <code>x</code>	Liefert Wert oder null	VB 2005: Nicht möglich (Kompilierungsfehler) Ab VB 2008: Liefert Wert oder <code>null</code>
Ausdruck <code>x.Value</code>	Liefert Wert oder Laufzeitfehler (»Das Objekt mit Nullwert muss einen Wert haben.«)	Liefert Wert oder Laufzeitfehler (»Das Objekt mit Nullwert muss einen Wert haben.«)
Ausdruck <code>x.HasValue</code>	Liefert <code>true</code> oder <code>false</code>	Liefert <code>true</code> oder <code>false</code>
Ausdruck <code>x + 1</code>	Liefert null, wenn <code>x</code> gleich null	VB 2005: Nicht möglich (Kompilierungsfehler) Ab VB 2008: Liefert null, wenn <code>x</code> gleich <code>null</code>
Zuweisung <code>x = a</code>	Erlaubt, liefert <code>a</code>	Erlaubt, liefert <code>a</code>
Zuweisung <code>a = x</code>	Kompilierungsfehler: Verbotene Typkonvertierung	Mit <code>Option Strict</code> : Verbotene Typkonvertierung Ohne <code>Option Strict</code> : Laufzeitfehler (»Das Objekt mit Nullwert muss einen Wert haben.«), wenn <code>x</code> gleich <code>null</code>
Zuweisung <code>a = (int) x</code> bzw. <code>a = CType(x, Integer)</code>	Laufzeitfehler (»Das Objekt mit Nullwert muss einen Wert haben.«), wenn <code>x</code> gleich <code>null</code>	Laufzeitfehler (»Das Objekt mit Nullwert muss einen Wert haben.«), wenn <code>x</code> gleich <code>null</code>
Konvertierung eines wertelosen Wertetyps in einen normalen Wertetypen mit der Semantik: liefert <code>x</code> , wenn <code>x</code> einen Wert hat oder Zahl 0, wenn <code>x</code> gleich <code>null</code> .	<code>a = x ?? 0</code>	<code>If x.HasValue Then</code> <code>a = x.Value</code> <code>Else</code> <code>a = 0</code> <code>End If</code>

Tabelle 6.6 Gegenüberstellung der Behandlung von wertelosen Wertetypen in C# und Visual Basic

ACHTUNG Bitte beachten Sie, dass man den Typ `string` (`System.String`) nicht als wertelosen Wertetyp verwenden kann, da `String` kein Wertetyp ist, sondern ein Referenztyp, der sich in einigen Punkten (z. B. Wertzuweisungen) verhält wie ein Wertetyp. Richtig ist also `string i = null;` statt `string? i = null;`

Null, Nothing und default

Kleine Unterschiede gibt es bei der Zurücksetzung von Variablen. Referenztypen (und wertelose Wertetypen) kann man aus der Sicht der CLR auf *null* setzen. Dies geht mit `null` in C# und `Nothing` in Visual Basic. `Nothing` kann man in Visual Basic auch auf normale Wertetypen anwenden. Sie setzen dann den Wertetyp auf den Standardwert, z.B. 0 bei Zahlen.

```
Dim x As Integer = Nothing
```

In C# geht dies mit

```
int x = default(Int32);
```

Operatoren

Bei den Operatoren gibt es einige erhebliche Unterschiede zwischen den Sprachen (insbesondere hinsichtlich der Bedeutung der Operatoren `+` und `=`).

Operatoren in VB

Die Standardoperatoren entsprechen den Operatoren in VB6. Mit VB7 neu eingeführt wurden folgende Operatoren:

- `AndAlso` und `OrElse` unterstützen die *Short-Circuit-Auswertung* (etwa: *Kurzschlussauswertung*) in Bedingungen. Dadurch wird die Auswertung eines Ausdrucks abgebrochen, sobald das Ergebnis feststeht. Bei den Operatoren `And` und `Or` werden immer alle Teile ausgewertet.
- Zuweisungen können direkt innerhalb der Variablendeklarationszeile vorgenommen werden.

```
Dim v As String = "Holger"
Dim n As String = "Schwichtenberg"
```

- Basisoperatoren können mit dem Zuweisungsoperator kombiniert werden:

```
v &= " " & n
```

- Seit VB7 nicht mehr im Sprachumfang enthalten sind die Operationen `Set` und `Let`. Sie werden aber vom Editor erkannt und automatisch entfernt, wenn sie fälschlicherweise eingegeben werden.

ACHTUNG Auch in der aktuellen Version von VB.NET gibt es weiterhin für Zeichenkettenverknüpfungen sowohl die Möglichkeit, mit dem Pluszeichen (`+`) als auch dem kaufmännische Und (`&`) zu arbeiten. Das kaufmännische Und ist aber vorzuziehen, da das Pluszeichen durch die ebenfalls noch vorhandene mathematische Bedeutung zu unerwünschten Ergebnissen führen kann.

Neu seit VB 2005 ist der Operator `isNot`, der den Ausdruck `Not x is y` verkürzt auf `x isNot y`.

TIPP

Nur eine Kleinigkeit wurde bei den Operatoren in VB 2008 verbessert: VB 7.0, 7.1 und 8.0 kannten wie VB 5.0/6.0 den IIF-Operator für einen *wenn ... dann ... sonst*-Ausdruck. Da allerdings in VB 7.0, 7.1 und 8.0 dieser Operator kein echter Operator, sondern eine Methode in der VB-Laufzeitumgebung war, wurden immer alle Teile der Ausdrucks ausgewertet (da diese ja Parameter waren), also auch der nicht zutreffende Fall. In dem nachfolgenden Fall führte dies zu einem Laufzeitfehler (»Object variable or With block variable not set.«), denn `Vorstandsvorsitzender.Name` wird ausgewertet, obwohl `Vorstandsvorsitzender` ja auf *nothing* zeigt. Neu ab VB 2008 ist `If` (nur mit einem »!*!*«), der ein echter Operator ist und sich auch so verhält, d. h. nur den zutreffenden Teil auswertet.

```
' Besetzung des Vorstandes
StellvertretenderVorstandsvorsitzer = MM
Vorstandsvorsitzender = Nothing

' Ausgabe des amtierenden Chefs (alte Variante, führt zu Laufzeitfehler "Object variable or
' With block variable not set.")
' Dim NameDesAmtierendenChefs As String = IIf(Vorstandsvorsitzender IsNot Nothing,
' Vorstandsvorsitzender.Name, StellvertretenderVorstandsvorsitzer.Name)

' Ausgabe des amtierenden Chefs (neu ab 2008 mit echtem IF-Operator)
Dim NameDesAmtierendenChefs As String = If(Vorstandsvorsitzender IsNot Nothing,
Vorstandsvorsitzender.Name, StellvertretenderVorstandsvorsitzer.Name)
```

Operatoren in C#

Es gibt drei wichtige Unterschiede zwischen den Operatoren in VB und C#, die bei Portierungen von Code zu beachten sind:

- Das Gleichheitszeichen `=` ist in C# immer der Zuweisungsoperator. Zum Vergleichen müssen immer zwei Gleichheitszeichen `==` verwendet werden.
- Zeichenkettenverknüpfungen erfolgen immer mit dem Pluszeichen `+`. Das kaufmännische Und `&` ist nicht erlaubt.
- Die logischen Operatoren Und `&&` und Oder `||` verwenden immer die Short-Circuit-Auswertung, d. h., die folgenden Teile eines Ausdrucks werden nicht mehr ausgewertet, sobald feststeht, dass der Ausdruck nicht mehr wahr werden kann
- Für den Objektvergleich verwendet C# die normalen Vergleichsoperatoren `==` und `!=`
- Bei der Division ist es vom Typ der Operanden abhängig, ob die Division als Ganzzahldivision ausgeführt wird

Ein C#-Operator, für den es keine Entsprechung in VB gibt, ist das doppelte Fragezeichen. `??` liefert (ab C# 2005) den Wert des vorangestellten Ausdrucks, wenn dieser nicht Null ist. Wenn der Wert Null ist, wird der Wert des nachfolgenden Ausdrucks übergeben.

```
// Umwandlung eines Nullable Int in einen Int
int? d = null;
int e = d ?? -1;
// Behandlung eines String
string s = null;
Demo.Print ("s = " + (s ?? "(kein Inhalt)"));
```

Listing 6.11 Einsatz des `??`-Operators

Leider ist der Operator nicht hilfreich, wenn man einen wertelosen Zahlenwert ausgeben möchte, weil beide Operanden den gleichen Typ besitzen müssen.

```
Demo.Print("d = " + (d ?? "null")); // geht leider nicht :-)
```

	Visual Basic	C#	Visual J#	C++	JScript
Mathematik					
Addition	+	+	+	+	+
Subtraktion	–	–	–	–	–
Multiplikation	*	*	*	*	*
Division	/	/	/	/	/
Ganzzahldivision	\	/	n/a	n/a	n/a
Modulus	Mod	%	%	%	%
Potenz	^	n/a	n/a	n/a	n/a
Negation	Not	~	~	~	~
Inkrement	n/a	++	++	++	++
Dekrement	n/a	--	--	--	--
Zuweisung					
Einfache Zuweisung	=	=	=	=	=
Addition	+=	+=	+=	+=	+=
Subtraktion	-=	-=	-=	-=	-=
Multiplikation	*=	*=	*=	*=	*=
Division	/=	/=	/=	/=	/=
Ganzzahl-Division	\=	/=	n/a	n/a	n/a
Zeichenkettenverbindung	&=	+=	+=		+=
Modulus (Divisionsrest)	n/a	%=	%=	%=	%=
Bit-Verschiebung nach links	<<=	<<=	<<=	<<=	<<=
Bit-Verschiebung nach rechts	>>=	>>=	>>=	>>=	>>=
Bit-weises UND	n/a	&=	&=	&=	&=
Bit-weises XOR	n/a	^=	^=	^=	^=
Bit-weises OR	n/a	=	=	=	=
Vergleich					
Kleiner	<	<	<	<	<
Kleiner gleich	<=	<=	<=	<=	<=
Größer	>	>	>	>	>
Größer gleich	>=	>=	>=	>=	>=
Gleich	=	=	=	=	=

	Visual Basic	C#	Visual J#	C++	JScript
Nicht gleich	< >	!=	!=	!=	!=
Objektvergleich	Is	= =	= =	n/a	= =
Objektvergleich (negativ)	IsNot	!=	!=	n/a	!=
Objekttypvergleich	TypeOf x Is Class1	x is Class1	x instanceof Class1	n/a	Instanceof
Zeichenkettenvergleich	=	= =	n/a	n/a	= =
Zeichenkettenverbindung	&	+	+	n/a	+
Logische Operatoren					
UND	And	&&	&&	&&	&&
ODER	Or				
NICHT	Not	!	!	!	!
Short-circuited UND	AndAlso	&&	&&	&&	&&
Short-circuited ODER	OrElse				
Bit-Operatoren					
Bit-weises UND	And	&	&	&	&
Bit-weises XOR	Xor	^	^	^	^
Bit-weises OR	Or				
Bit-Verschiebung nach links	<<	<<	<<	<<	<<
Bit-Verschiebung nach rechts	>>	>>	>>	>>	>>, >>>
Sonstiges					
Bedingt	IIF-Funktion und If-Operator	?:	?:	?:	?:
Bedingt (für Nullable Types)	n/a	?? :	n/a	n/a	n/a

Tabelle 6.7 Vergleich der Operatoren in verschiedenen .NET-Sprachen

Klassendefinition

Klassen sind in .NET das zentrale Konzept zur Aufnahme von Daten und Programmcode. Eine Klassendefinition erstellt eine neue Klasse.

Klassen können folgende Elemente enthalten:

- Attribute in Form von Feldern oder Property-Routinen
- Methoden mit und ohne Rückgabewerte (Function/Sub)
- Ereignisse (Events)

Geschachtelte Typen (vgl. Kapitel 4) werden sowohl in C# als auch in VB unterstützt.

HINWEIS

Sowohl in C# als auch in VB gilt: Anders als in Java darf eine Quellcodedatei beliebig viele Klassen enthalten und der Name der Quellcodedatei muss nicht dem in der Datei implementierten Klassennamen entsprechen. Die in Visual Studio integrierten Refactoring-Funktionen (Funktionen zur nachträglichen Umgestaltung von Programmcode) werden für C#-Klassen allerdings automatisch tätig, wenn eine Quellcodedatei umbenannt wird, die eine Klasse mit gleichem Namen enthält. In diesem Fall wird auch die Klasse umbenannt.

Klassendefinitionen in VB

Die Definition von Klassen in VB.NET entspricht der Definition von Klassen in VBScript 5.x mit dem Konstrukt `Class...End Class` (während Klassen in VB 6.0 und früher durch eigene `.cls`-Dateien definiert wurden, es aber kein Sprachkonstrukt gab).

Neben Klassen existieren in VB auch weiterhin Module, wobei das Konstrukt `Module...End Module` zum Einsatz kommt. Module entsprechen nicht vererbbaaren Klassen, bei denen alle Mitglieder statisch und daher ohne eine Instanziierung zugänglich sind.

Klassendefinitionen in C#

Klassen werden in C# durch das Schlüsselwort `class` und einen Block mit geschweiften Klammern gebildet.

An die Stelle des VB-Schlüsselworts `Module` tritt in C# ab Version 2005 das Konstrukt `static class`. Eine solche Klasse darf nur statische Mitglieder besitzen und nicht instanziiert werden, weil der Konstruktor automatisch als `private` deklariert ist. Die Klasse darf nur von `System.Object` erben.

```
static class StatischeKlasse
{
    public static void StatischesMitglied() { ... }
    // Nicht erlaubt: Instanzmitglied
    // public void InstanzMitglied();
}
```

Listing 6.12 Beispiel für eine statische Klasse in C#

Sichtbarkeiten / Zugriffsmodifizierer

Klassendefinitionen können mit den Sichtbarkeitsmodifizierern (alias Zugriffsmodifizierer) `public` oder `friend` (entspricht `internal` in C# bzw. der allgemeinen Sichtbarkeit *assembly*, vgl. Kapitel 4) versehen werden. Klassenmitglieder können folgende Sichtbarkeiten haben:

- **Private** Das Mitglied kann nur innerhalb der Klasse genutzt werden
- **Protected** Das Mitglied kann innerhalb der Klasse und in abgeleiteten Klassen genutzt werden
- **friend (C#: internal)** Das Mitglied kann in allen Klassen innerhalb der Assembly genutzt werden
- **Public** Das Mitglied kann in allen Klassen auch in referenzierenden Assemblys genutzt werden

HINWEIS VB und C# unterscheiden sich bei den Klassendefinitionen außer bei `friend/internal` nur hinsichtlich der Groß-/Kleinschreibung der Schlüsselwörter. In C# müssen die Schlüsselwörter klein geschrieben werden. In VB ist dies egal, der Editor schreibt die Wörter allerdings automatisch groß.

Felder (Field-Attribute)

Attribute ohne Codehinterlegung werden durch einfache Variablendeklarationen erzeugt. Felder können *Public* (sichtbar für die Klasse und alle Nutzer), *Private* (sichtbar nur für die Klasse) oder *Protected* (sichtbar für die Klasse und geerbte Klassen) sein.

Felder in VB

Die nachfolgenden drei Zeilen zeigen gültige Feld-Deklarationen in VB. Die Sichtbarkeitsmodifizierer treten an die Stelle von `Dim`.

```
Private PersonalausweisNr As String
Public Vorname, Nachname As String
Protected Geburtstag As DateTime, Geburtsort As String = "unbekannt"
```

Felder in C#

In C# werden die Sichtbarkeitsmodifizierer vorangestellt.

```
private string PersonalausweisNr;
public string Vorname, Nachname;
Protected System.DateTime Geburtstag;
Protected string Geburtsort = "unbekannt";
```

Eigenschaften (Property-Attribute)

Ein Property dient im (.NET-)Programm dazu, ein Attribut einer Klasse zu deklarieren, bei dem aber Programmcode sowohl beim Setzen des Werts als auch beim Lesen des Werts ausgeführt wird. Ein Property ist damit eine Mischung aus Attribut und Methode: Der Aufrufer sieht das Property als Attribut, die Klasse intern sieht zwei Methoden: Die Get-Methode (alias *Getter*) zum Lesen und die Set-Methode (alias *Setter*) zum Schreiben des Attributs.

Was tatsächlich in Getter und Setter ausgeführt wird, ist dem Entwickler überlassen. Typische Beispiele sind:

- Im Getter wird ein Wert berechnet, statt ihn aus dem Speicher zu lesen. Der Setter fehlt, weil es keinen Sinn macht, einen berechneten Wert zu speichern (z. B. *Alter*: Diese Property würde im Getter das Alter aus Geburtstag und aktuellem Datum errechnen. Einen Setter gäbe es nur für *Geburtsort*, aber nicht für *Alter*).
- Im Setter wird geprüft, ob der Wert Sinn macht (z. B. Geburtstag darf nicht in Zukunft liegen)
- Man darf einen Wert setzen, aber nicht wieder auslesen (z. B. Kennwort)

Im deutschen verwendet Microsoft *Eigenschaft* als Übersetzung für *Property*, im Gegensatz zu den normalen (»direkten«) Attributen, die Microsoft *field* bzw. *Feld* nennt.

Ein Indexer ist eine Property mit Parametern, z.B. `Personenliste.Mitglieder[10]`.

Beide Sprachen unterstützen Eigenschaften (alias Property-Attribute) mit Getter- und Setter-Methoden.

HINWEIS Die *Automatischen Eigenschaften* (engl. *Automatic Property*) machen die Syntax prägnanter für solche Property-Attribute, die nichts anderes tun als ein privates *Field*-Attribut zu lesen und zu beschreiben. In diesem Fall kann man sich die explizite Definition des privaten *Field*-Attributs sparen und die Erzeugung dem Compiler überlassen. Damit verkürzt sich auch die Schreibweise von Getter und Setter radikal. Automatische Eigenschaften gibt es in C# seit Version 2008 und in Visual Basic seit Version 2010.

Eigenschaften in VB

Die Definition von Property-Routinen wurde ab VB7 syntaktisch geändert. Seit VB 2005 können die Get- und Set-Methoden sogar verschiedene Sichtbarkeiten besitzen. In dem nachfolgenden Beispiel ist der Getter für die Clients zugänglich (*Public*), der Setter steht aber nur in den von dieser Klasse abgeleiteten Klassen zur Verfügung (*Protected*).

```
Private _FlugStunden As Long
Public Property Flugstunden() As Long
    Get
        Return Me._FlugStunden
    End Get
    Protected Set(ByVal Value As Long)
        Me._FlugStunden = Value
    End Set
End Property
```

Listing 6.13 Ein Property in VB mit zugehörigem Field in normaler Schreibweise

Den obigen Programmcode könnte man (ab Visual Basic 2010) auch als automatische Eigenschaft schreiben:

```
Public Property Flugstunden As Long
```

Listing 6.14 Ein Property in VB mit zugehörigem Field als Automatische Eigenschaft

Dann darf es aber kein Mitglied `_Flugstunden` geben, denn dies erzeugt der Compiler automatisch.

In Visual Basic .NET ist – anders als in VB6 – nicht mehr erlaubt, ein Standardattribut in einer Klasse festzulegen, das automatisch verwendet wird, wenn ohne nähere Angabe eines Attributnamens Bezug auf eine Objektvariable genommen wird. Nur noch parametrisierte Attribute (genannt *Indexer*) können Standardmitglieder einer Klasse werden.

Es ist nicht erlaubt, dass es neben einer Property mit gleichem Namen auch noch explizite Getter-/Setter-Routinen mit `get_` bzw. `set_` (vgl. Java) gibt.

HINWEIS Visual Basic kennt darüber hinaus auch Property-Signaturen, die beliebig viele Parameter aufnehmen können. Dies ist insbesondere der Abwärtskompatibilität zu VB6 bzw. VBA-Code geschuldet. Sie sollten von der Verwendung mehrerer Parameter in Eigenschaften allerdings Abstand nehmen, um die Kompatibilität zu anderen .NET-Sprachen zu wahren.

```
Public Property Flugstunden()
    Get
        property "Flugstunden" definiert implizit "set_Flugstunden", was einen Konflikt mit einem Member mit dem gleichen Namen in class "misc" verursacht.
    End Get
    Set(ByVal value)
    End Set
End Property
Sub get_Flugstunden()
End Sub
Sub set_Flugstunden()
End Sub
```

Abbildung 6.6 Restriktionen für Getter- und Setter-Implementierung

Eigenschaften in C#

In C# ist die Syntax für Eigenschaften prägnanter.

```
private long _FlugStunden;
public long FlugStunden
{
    get
    {
        return this._FlugStunden;
    }
    protected set
    {
        this._FlugStunden = value;
    }
}
```

Listing 6.15 Ein Property in C# mit zugehörigem Field in normaler Schreibweise

Automatische Eigenschaften gibt es in C# seit Version 2008.

```
public long FlugStunden { get; set; }
```

Listing 6.16 Ein Property in C# mit zugehörigem Field als Automatische Eigenschaft

HINWEIS Bei einer automatischen Eigenschaft kann man das zugehörige private *Field*-Attribut nicht im Code getrennt ansprechen. Alle Zugriffe laufen über das *Property*-Attribut. Daher muss man immer Getter und Setter definieren. Diese können aber unterschiedliche Sichtbarkeiten haben, z. B.

```
public long FlugStunden { get; protected set; }
```

Methoden

Methoden sind Operationen in Klassen, die innerhalb der Klasse oder von Nutzern aufgerufen werden können. Methoden können einen Rückgabewert liefern. Parameter von Methoden können optional sein. Weggelassene Parameter werden durch Vorgabewerte ersetzt, die in der Methodendeklaration stehen müssen. Der Aufrufer gibt in der Regel die Parameter in der in der Deklaration vorgegebenen Reihenfolge an. Durch eine spezielle Syntax kann man aber die Parameter in einer beliebigen Reihenfolge angeben. Optionale Parameter dürfen Wertelose Wertetypen (Nullable Types) sein.

Methoden in VB

UnterROUTinen werden in VB mit Sub oder Function eingeleitet. Sub sind Methoden ohne Rückgabewert, Function sind Methoden mit Rückgabewert. Zur Rückgabe von Werten aus Funktionen kann ab VB7 das Schlüsselwort Return(wert) verwendet werden. Bei einem UnterROUTinenaufruf muss der Entwickler die Parameter immer in runde Klammern setzen. In VB6 musste er zwischen Funktionen mit Rückgabewert und Prozeduraufrufen ohne Klammern unterscheiden.

HINWEIS Parameterlose Methoden kann man theoretisch weiterhin ohne Klammern aufrufen. Der Compiler akzeptiert dies; die Entwicklungsumgebung Visual Studio wird aber immer die »vergessenen« Klammern ergänzen.

Seit VB7 muss ein optionaler Parameter auch einen Standardwert besitzen.

```
Public Overloads Sub BuchenMitRaucherAngabe(ByVal Flugnummer As String, Optional ByVal NichtRaucher As Boolean = True)
```

Ab VB 2010 dürfen optionale Parameter auch wertelose Wertetypen (Nullable Types) sein.

Mit dem Schlüsselwort Overloads können mehrere gleichnamige (überladene) UnterROUTinen erstellt werden, sofern sich diese hinsichtlich der Anzahl und / oder Datentypen der Parameter unterscheiden. VB entscheidet bei einem Funktionsaufruf anhand der Typsignatur, welche der gleichnamigen Implementierungen aufzurufen ist.

```
Public Overloads Sub Buchen(ByVal Flug As Flug)
Public Overloads Sub Buchen(ByVal Flugnummer As String)
```

Seit VB7 werden alle Parameter standardmäßig mit ihrem Wert übergeben (*Call by Value*). Bis VB6 war die Übergabe von Zeigern auf die Parameter die Voreinstellung (*Call by Reference*). In VB.NET muss man Referenzparameter explizit durch ByRef kennzeichnen.

```
Public Shared Sub Run()
    Dim a As Integer = 1
    Dim b As Integer = 2
    Dim c As Integer = 2
    Dim Ergebnis As String = Test(a, b, c)
    Console.WriteLine(Ergebnis)
    Console.WriteLine(a & ";" & b & ";" & c)
End Sub

Public Shared Function Test(ByVal WertValue As Integer, ByRef WertRef As Integer,
<System.Runtime.InteropServices.Out()> ByRef WertOut As Integer) As String
    WertValue += 1
    WertRef += 1
    ' nicht erlaubt, da noch nicht initialisiert: WertOut+=1
    WertOut = 10
    Return WertValue.ToString() & ";" & WertRef.ToString() & ";" & WertOut.ToString()
End Function
End Class
```

Listing 6.17 Beispiel für Parameterübergaben an Methoden

HINWEIS Die Annotation (zu Annotationen siehe gleichnamiger Abschnitt in diesem Kapitel) `<System.Runtime.InteropServices.Out(>)` bedeutet, dass der Aufrufer nur leeren (nicht initialisierten) Speicherplatz in die Methode hereingibt. Der Wert muss zwangsläufig von der Methode selbst initialisiert werden und wird dann dem Aufrufer geliefert. Diese Annotation ist das Äquivalent in Visual Basic zum Schlüsselwort `out` in C# (siehe unten).

Methoden in C#

In C# beginnt eine Methodendefinition ebenfalls mit der Sichtbarkeit. Danach folgt aber der Datentyp des Rückgabewerts. In C# gibt es kein direktes Pendant zu `Sub` und `Function`. Methoden ohne Rückgabewerte werden durch den Datentyp `void` signalisiert. Für überladene Methoden gibt es kein Schlüsselwort. Der Rückgabewert wird ebenfalls mit `return` signalisiert, aber klein geschrieben.

```
public void Buchen(Flug Flug)
public void Buchen(string FlugNummer)
```

ACHTUNG Beim Methodenaufruf sind immer runde Klammern zu verwenden, auch wenn es keine Parameter gibt!

Für die Übergaberichtung gibt es in C# für den Call by Value-Fall kein Schlüsselwort und für den Call by Reference-Fall zwei Wörter:

- Der Zusatz `ref` vor einem Parameter entspricht `ByRef` in VB und bedeutet, dass der Wert von außen hereingegeben wird und innerhalb der Methode verändert werden darf
- Der Zusatz `out` vor einem Parameter bedeutet, dass der Aufrufer nur leeren (nicht initialisierten) Speicherplatz hereingibt. Der Wert muss zwangsläufig von der Methode selbst gesetzt werden und wird dann dem Aufrufer geliefert.

HINWEIS Wichtig ist, dass man nicht nur in der Methodensignatur selbst `out` und `ref` verwenden muss, sondern auch beim Aufruf der Methode.

```
public static void Run()
{
    int a = 1;
    int b = 2;
    int c = 2;
    string Ergebnis = Test(a, ref b, out c);
    Console.WriteLine(Ergebnis);
    Console.WriteLine(a + ";" + b + ";" + c);
}

public static string Test(int WertValue, ref int WertRef, out int WertOut)
{
    WertValue++;
    WertRef++;
    // nicht erlaubt, da noch nicht initialisiert: WertOut++;
    WertOut = 10;
    return WertValue.ToString() + ";" + WertRef.ToString() + ";" + WertOut.ToString();
}
```

Listing 6.18 Beispiel zum Einsatz von `ref` und `out` in C#. Die Ausgabe ist 2;3;10 und 1;3;10

Erst seit C# 4.0 gibt es optionale und benannte Parameter. Zuvor musste man optionale Parameter durch Methodenüberladung nachbilden. Optionale Parameter werden in C# 4.0 durch einen Vorgabewert in dem Methodenkopf angezeigt. Optionale Parameter dürfen nur am Ende der Parameterliste erscheinen.

```
/// <summary>
/// Methode mit zwei optionalen Parametern
/// </summary>
public void Print(string text, ConsoleColor Farbe = ConsoleColor.Gray, bool Datum = false)
{
    if (Datum) text = DateTime.Now.ToShortTimeString() + ": " + text;
    ConsoleColor bisherigeFarbe = Console.ForegroundColor;
    Console.ForegroundColor = Farbe;
    Console.WriteLine(text);
    Console.ForegroundColor = bisherigeFarbe;
}
```

Listing 6.19 Methode mit zwei optionalen Parametern

Die obige Methode kann man wie folgt aufrufen:

```
CS10_Parameter obj = new CS10_Parameter();
obj.Print("Ausgabe ohne spezielle Farbe und ohne Datum.");
obj.Print("Ausgabe in grün und ohne Datum.", ConsoleColor.Green);
obj.Print("Ausgabe in grün und mit Datum.", ConsoleColor.Green, true);
```

Benannte Parameter erlauben die Angabe der Parameter in beliebiger Reihenfolge unabhängig von der Reihenfolge in der Deklaration. Ein benannter Parameter ist allein Sache des Aufrufers, d.h. hierzu sind keine Änderungen in der Deklaration notwendig. Der Aufrufer gibt durch Parametername und Doppelpunkt an, welchen Parameter er übergeben will.

```
obj.Print(text: "Ausgabe ohne spezielle Farbe und mit Datum.", Datum: true);
```

ACHTUNG Wenn man das Kompilat eines optionalen Parameterruufs mit einem Decompiler betrachtet, wird man überrascht: Die Aufrufe erfolgen gar nicht mit weniger Parametern, vielmehr werden die Vorgabewerte mit in den Aufruf hineinkompiliert. Das gilt sowohl für C# als auch Visual Basic.

```
obj = new CS10_Parameter();
obj.Print("Ausgabe ohne spezielle Farbe und ohne Datum.", 7, 0);
obj.Print("Ausgabe in gr\x00fc und ohne Datum.", 10, 0);
obj.Print("Ausgabe in gr\x00fc und mit Datum.", 10, 1);
CS$0$0000 = "Ausgabe ohne spezielle Farbe und mit Datum.";
CS$0$0001 = 1;
obj.Print(CS$0$0000, 7, CS$0$0001);
```

Listing 6.20 Dekompilat mit .NET Reflector [<http://www.red-gate.com/products/reflector>]

In der Verwendung optionaler Parameter besteht also eine Gefahr: Wenn die optionale Methode in einer anderen Assembly als der Aufrufer ist und diese beiden Assemblys unabhängig voneinander kompiliert werden (also nicht in einer Projektmappe sind), dann kann es zu Inkonsistenzen kommen. Nach einer Änderung der Vorgabewerte würden nicht erneut kompilierte Aufrufer weiterhin die alten Werte verwenden.

Erweiterungsmethoden (Extension Methods)

Eine Erweiterungsmethode ermöglicht einer Klasse, extern eine Methode anzuheften. *Extern* heißt, dass dies nicht im Rahmen der Klassendefinition selbst erfolgt, sondern in einer anderen Klasse. Damit ist es möglich, Klassen zu erweitern, die man selbst nicht geschrieben hat (z. B. Klassen der .NET-Klassenbibliothek *FCL*). Ein solches Konzept ist bereits aus JavaScript vielen Entwicklern bekannt. Zu beachten ist, dass die Methoden gemäß dem Prinzip der Kapselung nur auf die öffentlichen Attribute und Methoden der Klasse zugreifen können. Durch Einsatz von Reflection (Metadatenutzung) kann diese Beschränkung jedoch umgangen werden (durch Reflection kann man immer auch auf private Mitglieder zugreifen!). Erweiterungen können nur Methoden sein; Fields und Properties können leider nicht nachträglich ergänzt werden.

TIPP

Erweiterungsmethoden können auch auf Schnittstellen angewendet werden, sodass man auf einfache Weise alle Klassen erweitern kann, die eine bestimmte Schnittstelle anbieten. Microsoft hat dies im Rahmen von Language Integrated Query (siehe Kapitel 10 »Language Integrated Query (LINQ)«) auf die Schnittstelle *IEnumerable* angewendet, um alle Objektmengenklassen »LINQ-fähig« zu machen.

HINWEIS

Mit den Erweiterungsmethoden hat man nun eine dritte syntaktische Möglichkeit, bestehende Klassen zu erweitern:

- **Vererbung** Möglich seit .NET 1.0, aber nur für Klassen, die Vererbung zulassen (also nicht *sealed* bzw. *NotInheritable* sind)
- **Partielle Klassen** Möglich seit .NET 2.0, aber nur für Klassen im gleichen Projekt, die als *Partiell* gekennzeichnet sind
- **Erweiterungsmethoden** Möglich seit .NET 3.5, für alle Klassen und auch anwendbar auf Schnittstellen

ACHTUNG

Wichtig ist, dass in der Klasse, in der die Erweiterungsmethode verwendet wird, der Namensraum der Klasse, in der die Erweiterungsmethode implementiert wurde, durch `using` bzw. `imports` eingebunden wird. Sonst kann die Erweiterungsmethode vom Compiler nicht gefunden werden. Dies ist auch der Grund dafür, dass LINQ-Abfrageausdrücke nur zur Verfügung stehen, wenn der Namensraum `System.Linq` eingebunden wurde (siehe Kapitel 10 zu LINQ »Language Integrated Query (LINQ)«).

Der Name der Klasse, in der die Erweiterungsmethode implementiert wird, ist im Übrigen egal. Auf diese Weise ist die Anzahl der Erweiterungsmethoden für eine Klasse nicht räumlich und der Menge nach beschränkt. Erweiterungsmethoden können überladen werden, wobei hier die gleichen Bedingungen wie bei normalen Methoden gelten. Erweiterungsmethoden müssen keinen Rückgabewert haben (d. h. *void* bzw. *Sub* sind erlaubt).

Definition von Erweiterungsmethoden in VB

Erweiterungsmethoden werden durch die Annotation `System.Runtime.CompilerServices.Extension` gekennzeichnet. Der erste Parameter der Methode ist der Datentyp, der erweitert werden soll. Die weiteren Parameter sind die tatsächlichen Parameter der Methode.

Das Beispiel zeigt die Implementierung einer Erweiterungsmethode `Print()` für die Schnittstelle `IEnumerable`. Dadurch erhalten alle Objektmengenklassen in .NET die Methode `Print()`, die alle enthaltenen Objekte in einer bestimmten Farbe an der Konsole ausgibt (die Ausgabe erfolgt mit `ToString()` und ist daher darauf angewiesen, dass `ToString()` in den Objekten sinnvoll implementiert wurde. Sonst wird nur die Standardimplementierung aufgerufen, d.h. der Name der Klasse ausgegeben).

HINWEIS

In VB.NET muss die Implementierung einer Erweiterungsmethode in einem Modul (`Module ... End Module`) erfolgen. Klassen (`Class ... End Class`) sind leider nicht erlaubt. Dies ist optisch zwar unschön, weil die Syntax an das nicht-objektorientierte Programmieren in VB 6.0 erinnert, ist aber sonst nicht weiter schlimm.

```
Imports System.Runtime.CompilerServices
Imports System
Imports System.Collections

Module WWWingsCollectionExtensions

    ' --- Erweiterungsmethode für IEnumerable
    <Extension()>
    Public Sub Print(ByVal Menge As IEnumerable, ByVal Farbe As ConsoleColor)
        Dim VorherigeFarbe As ConsoleColor = Console.ForegroundColor
        Console.ForegroundColor = Farbe
        For Each o As Object In Menge
            Console.WriteLine(o.ToString())
        Next
        Console.ForegroundColor = VorherigeFarbe
    End Sub

End Module
```

Listing 6.21 Implementierung der Erweiterungsmethode `Print()` für die Schnittstelle `IEnumerable` (in VB)

```
Dim Vorstandsmitglieder As New List(Of Vorstandsmitglied)
Vorstandsmitglieder.Add(MM)
Vorstandsmitglieder.Add(HM)
Vorstandsmitglieder.Add(HS)

' Verwendung einer Erweiterungsmethode
Vorstandsmitglieder.Print(ConsoleColor.DarkYellow)
```

Listing 6.22 Anwendung der Methode `Print()` auf eine Menge, die mit der generischen Mengenkasse `List` erzeugt wurde (in VB)

Definition von Erweiterungsmethoden in C#

Die Syntax für Erweiterungsmethoden in C# ist ganz anders. Wenn man die Annotation `Extension` verwendet, quittiert der Compiler dies mit »Do not use 'System.Runtime.CompilerServices.ExtensionAttribute'. Use the 'this' keyword instead.«. Gemeint ist damit, dass man vor den ersten Parameter (also den Namen der zu erweiternden Klasse) das Schlüsselwort `this` schreiben soll. Dies ist leider wenig intuitiv, zumal `this` schon mehrere andere Bedeutungen in C# hat. Außerdem muss die Erweiterungsmethode statisch deklariert sein, wenngleich sie nachher eine Instanzmethode ist. Ebenso muss die Klasse statisch sein.

```
using System.Runtime.CompilerServices;
using System;
using System.Collections;

namespace de.WWWings.Library
{
    internal static class WWWingsCollectionExtensions
    {
        // --- Erweiterungsmethode für IEnumerable
        public static void Print(this IEnumerable Menge, ConsoleColor Farbe)
        {
            ConsoleColor VorherigeFarbe = Console.ForegroundColor;
            Console.ForegroundColor = Farbe;
            foreach (object o in Menge)
                Console.WriteLine(o.ToString());
            Console.ForegroundColor = VorherigeFarbe;
        }
    }
}
```

Listing 6.23 Implementierung der Erweiterungsmethode Print() für die Schnittstelle IEnumerable (in C#)

```
Imports de.WWWings.Library
...
List<Vorstandsmitglied> Vorstandsmitglieder = new List<Vorstandsmitglied> { HS, HM, MM };

// Verwendung einer Erweiterungsmethode
Vorstandsmitglieder.Print(ConsoleColor.DarkYellow);
```

Listing 6.24 Anwendung der Methode Print() auf eine Menge, die mit der generischen Mengenkasse List erzeugt wurde (in C#)

Konstruktoren und Destruktoren

Ein Konstruktor ist eine Methode, die beim Instanziiieren aufgerufen wird. Echte Destruktoren, die beim Löschen eines Objekts aufgerufen werden, kennt das .NET Framework hingegen nicht. Der Aufruf des Destruktors ist im .NET Framework nicht deterministisch. Daher spricht man oft auch von Finalizern statt von Destruktoren.

ACHTUNG Ein parameterloser Konstruktor, der nichts tut, scheint auf den ersten Blick überflüssig zu sein. Sofern kein parameterbehafteter Konstruktor vorhanden ist, generiert der Compiler – sowohl von C# als auch von VB – automatisch einen parameterlosen Konstruktor. Wird jedoch ein parameterbehafteter Konstruktor explizit implementiert, so wird der parameterlose Konstruktor nicht automatisch erzeugt. Wenn dieser benötigt wird, ist er also ebenfalls explizit zu implementieren.

Konstruktoren und Destruktoren in VB

Neu ab VB7 ist die Möglichkeit, Konstruktoren mit Parametern zu definieren. Dazu dient eine öffentliche Unteroutine mit dem Namen Sub New(). New() ohne Parameter entspricht dem Class_Initialize() in VB6. Class_Terminate() findet eine ähnliche, aber nicht ganz exakte Entsprechung in Sub Finalize() (vgl. Erläuterungen zu Destruktoren / Finalizern in Kapitel 4). Ab VB7 erfolgt die Instanziierung bei einem New-Operator sofort und nicht erst bei der ersten Verwendung des Objekts. Das Schlüsselwort Set wird bei der Instanziierung nicht mehr benutzt.

Konstruktoren und Destruktoren in C#

Konstruktoren besitzen den Namen der Klasse und haben keinen Rückgabetyt (auch nicht void). Der Bezeichner für den Finalizer besteht aus ~, gefolgt vom Klassennamen. Es kann nur höchstens einen Finalizer geben, aber beliebig viele überladene Konstruktoren. Wie in VB.NET wird der parameterlose Konstruktor nur dann automatisch erzeugt, wenn kein anderer Konstruktor explizit implementiert wird.

Objektinitialisierung

Bisher konnte man Objekte nur prägnant und elegant bei der Instanziierung initialisieren, wenn die Klassen entsprechende Parameter im Konstruktor anboten. Seit C# 3.0 und VB 9.0 kann nun jedes öffentliche Attribut (egal ob *Field* oder *Property*) bei der Instanziierung initialisiert werden. C# bietet dazu eine Schreibweise mit geschweiften Klammern an, VB das Schlüsselwort *with*. In VB ist außerdem zu beachten, dass immer dem Attributnamen ein Punkt voranzustellen ist.

HINWEIS Man kann nur öffentliche und beschreibbare Attribute initialisieren. Man muss keineswegs alle Attribute initialisieren. Man darf aber jedes Attribut nur einmal initialisieren.

```
Dim MM As New Vorstandsmitglied() With {.Name = "Max Müller", .Aufgabengebiet = "Flugbetrieb", .Alter = 33}
Dim HM As New Vorstandsmitglied() With {.Name = "Hans Meier", .Aufgabengebiet = "Personal", .Alter = 42}
Dim HS As New Vorstandsmitglied() With {.Name = "Hubert Schmidt", .Aufgabengebiet = "Marketing",
                                         .Alter = 35, .Ort = "Essen" }
```

Listing 6.25 Initialisierung von Objekten bei der Instanziierung (VB 9.0)

```
Vorstandsmitglied MM = new Vorstandsmitglied() { Name = "Max Müller", Aufgabengebiet = "Flugbetrieb",
                                                    Alter = 33 };
Vorstandsmitglied HM = new Vorstandsmitglied() { Name = "Hans Meier", Aufgabengebiet = "Personal",
                                                    Alter = 42 };
Vorstandsmitglied HS = new Vorstandsmitglied() { Name = "Hubert Schmidt", Aufgabengebiet = "Marketing",
                                                    Alter = 35, Ort = "Essen" };
```

Listing 6.26 Initialisierung von Objekten bei der Instanziierung (C# 3.0)

HINWEIS Man kann die Objektinitialisierung auch zusätzlich verwenden, wenn es einen parameterbehafteten Konstruktor gibt, z. B.

```
Vorstandsmitglied HS = new Vorstandsmitglied("Hubert Schmidt") { Aufgabengebiet = "Marketing",
                                                                    Alter = 35, Ort = "Essen" };
```

Beispiel für eine Klasse mit diversen Mitgliedern

Das erste Listing zeigt die Implementierung der Klasse Person mit zwei Konstruktoren, einem parameterlosen Konstruktor und einem Konstruktor mit zwei Parametern.

```
Namespace de.WWWings
    Public Class Person
        ' ===== Attribute (Fields)
        Public PersonalausweisNr As String
        Public Vorname As String
        Public Nachname As String
        ' ===== Errechnete Attribute (Properties)
        Public ReadOnly Property GanzerName() As String
            Get
                Return Me.Vorname & " " & Me.Nachname
            End Get
        End Property
        ' ===== Konstruktoren
        Public Sub New()
            End Sub
        Public Sub New(ByVal Nachname As String, ByVal Vorname As String)
            Me.Vorname = Vorname
            Me.Nachname = Nachname
        End Sub
        ' ===== Methoden
        Public Overridable Sub Info()
            Console.WriteLine("Person: " & Me.GanzerName)
        End Sub
    End Class
End Namespace
```

Listing 6.27 Implementierung der Klasse Person in VB.NET

```
namespace de.WWWings
{
    public class Person
    {
        // ===== Attribute (Fields)
        public string PersonalausweisNr;
        public string Vorname;
        public string Nachname;
        // ===== Errechnete Attribute (Properties)
        public string GanzerName
        {
            get
            {
                return this.Vorname + " " + this.Nachname;
            }
        }
        // ===== Konstruktoren
        public Person() { }
        public Person(string Nachname, string Vorname)
        {
            this.Vorname = Vorname;
            this.Nachname = Nachname;
        }
    }
}
```

```
// ===== Methoden
public virtual void Info()
{ Console.WriteLine("Person: " + this.GanzerName); }
}
}
```

Listing 6.28 Implementierung der Klasse Person in C#

Generische Klassen

Generische Klassen (Generics) erlauben es, einen oder mehrere Typen, die die Klasse intern verarbeitet, variabel zu halten (Typparameter). Ein typischer Einsatzfall sind generische Objektmengen (siehe auch Erläuterungen zu System.Collections im Kapitel 9 ».NET-Klassenbibliothek 4.0«). Generische Objektmengen ermöglichen es, dass der Entwickler einen allgemeinen Mengentyp so prägt, dass die Menge nur Mitglieder einer bestimmten Klasse akzeptiert und dafür eine Typprüfung bereits zur Entwicklungszeit stattfindet.

Neben den in der FCL implementierten generischen Objektmengen kann man in VB und C# auch selbst generische Klassen erzeugen. In diesem Kapitel wird die Definition und Verwendung eigener generischer Klassen besprochen.

Definition generischer Klassen in VB

Zur Definition einer generischen Klasse in VB gibt man bei der Klassendeklaration im Anschluss an den Klassennamen – in Klammern und eingeleitet durch das neue Schlüsselwort `Of` – Namen als Platzhalter für die bei der Deklaration einer Objektvariablen anzugebenden Objekttypen (Typparameter) an.

Beispiel: Die Klasse Mitarbeiterzuordnung erwartet bei der Deklaration einer Objektvariablen zwei Typen, einen `ChefTyp` und einen `AssistentTyp`.

```
Namespace de.WWWings.MitarbeiterSystem
    Public Class Mitarbeiterzuordnung(Of ChefTyp, AssistentTyp)
        Public Chef As ChefTyp
        Public Assi As AssistentTyp

        Sub New(ByVal Chef As ChefTyp, ByVal Assi As AssistentTyp, ByVal flug As de.WWWings.Flug)
            Me.Chef = Chef
            Me.Assi = Assi
        End Sub
    End Class
End Namespace
```

Listing 6.29 Deklaration einer generischen Klasse in VB

Definition generischer Klassen in C#

Die Unterstützung für generische Klassen wurde in C# ebenso wie in VB im Rahmen von .NET 2.0 hinzugefügt. Wie in vielen anderen Punkten auch, ist der Unterschied rein syntaktisch: An die Stelle des `Of`-Operators in runden Klammern tritt ein Klammernpaar aus spitzen Klammern. Die Bedingungen für die generischen Typparameter (Generic Constraints) definiert man mit dem Schlüsselwort `where`.

```

public class Mitarbeiterzuordnung<ChefTyp, AssistentTyp>
    where ChefTyp : Mitarbeiter
    where AssistentTyp : Mitarbeiter
{
    ChefTyp Chef;
    AssistentTyp Assi;

    public Mitarbeiterzuordnung(ChefTyp Chef, AssistentTyp Assi)
    {
        this.Chef = Chef;
        this.Assi = Assi;
    }
}

```

Listing 6.30 Implementierung einer generischen Klasse in C#

Verwendung generischer Klassen in VB

Bei der Verwendung einer generischen Klasse müssen sowohl bei der Deklaration der Objektvariablen als auch bei der Instanziierung mit dem Schlüsselwort `Of` die zu gebrauchenden Typen angegeben werden. In dem folgenden Beispiel wird ein Team aus zwei Piloten gebildet.

```

Dim pass1 As New Passagier("Schwichtenberg", "Holger")
Dim pilot1 As New Pilot("Müller", "Max")
Dim pilot2 As New Pilot("Meier", "Hans")
Dim CockpitTeam As Mitarbeiterzuordnung(Of Pilot, Pilot)
CockpitTeam = New Mitarbeiterzuordnung(Of Pilot, Pilot)(pilot1, pilot2)
' Fehler: CockpitTeam = New Mitarbeiterzuordnung(Of Pilot, Pilot)(pilot1, Pass1)

```

Listing 6.31 Gebrauch der generischen Klasse in VB

Die letzte Zeile ist ungültig, weil gemäß der Deklaration von `CockpitTeam` zwei Piloten erwartet werden. Die Übergabe eines `Passagier`-Objekts als Copilot wird bereits vom Compiler abgelehnt.

Ohne generische Klassen gäbe es zwei (unbefriedigende) Implementierungsalternativen:

- Für jeden möglichen Typ von Mitarbeiterkombination muss eine eigene Klasse deklariert werden (z. B. Pilot-Pilot, Pilot-Ingenieur, Purser-Flugbegleiter)
- Die ersten beiden Parameter des Konstruktors der Klasse `Mitarbeiterzuordnung` werden als `System.Object` deklariert, sodass jede Kombination möglich ist. Dann bemerkt der Compiler aber unsinnige Kombinationen (z. B. Pilot-Flugzeug) nicht mehr.

Verwendung generischer Klassen in C#

In C# kommen anstelle von runden Klammern und dem Schlüsselwort `Of` die spitzen Klammern zum Einsatz, um die von der Klasse erwarteten Typparameter anzugeben.

```

Passagier Pass1 = new Passagier("Schwichtenberg", "Holger")
Mitarbeiterzuordnung<Pilot,Pilot> CockpitTeam;
Pilot Pilot1 = new Pilot("Müller", "Max")
Pilot Pilot2 = new Pilot("Meier", "Hans");
CockpitTeam = new Mitarbeiterzuordnung<Pilot, Pilot>(Pilot1, Pilot2);
' Fehler: CockpitTeam = new Mitarbeiterzuordnung<Pilot, Pilot>(Pilot1, Pass1);

```

Listing 6.32 Nutzung einer generischen Klasse in C#

Einschränkungen für generische Typparameter (Generic Constraints)

Ein Problem verbleibt bei der Nutzung generischer Typen: Bei der Deklaration einer Variablen für einen generischen Typ könnte ein Entwickler (versehentlich) Typparameter angeben, für die die Klasse gar nicht vorgesehen ist, beispielsweise ein `File`-Objekt und ein `Directory`-Objekt bei der Klasse `Mitarbeiterzuordnung`.

```
' Das ist Unsinn:
Dim DateiTeam As New Mitarbeitersystem.Mitarbeiterzuordnung(Of System.IO.FileInfo,
                                                             System.IO.DirectoryInfo)
```

Um dies zu verhindern, können Bedingungen für die Typparameter (so genannte Generic Constraints) definiert werden. In Visual Basic erfolgt die Festlegung solcher Generic Constraints mit dem Schlüsselwort `As` hinter dem Typparameternamen in der `Of`-Deklaration. Nach dem `As` dürfen in geschweiften Klammern beliebig viele Schnittstellennamen, aber maximal ein Klassename genannt werden, da die angegebenen Namen additiv wirken und eine Klasse maximal eine Basisklasse besitzen darf. In C# verwendet man das Schlüsselwort `where`.

```
Public Class Mitarbeiterzuordnung(Of ChefTyp As {Mitarbeiter, New}, AssistentTyp As {Mitarbeiter, New})
    Public Chef As ChefTyp
    Public Assi As AssistentTyp
    Sub New(ByVal Chef As ChefTyp, ByVal Assi As AssistentTyp, ByVal flug As de.WWWings.Flug)
        Me.Chef = Chef
        Me.Assi = Assi
    End Sub
End Class
```

Listing 6.33 Deklaration einer generischen Klasse in VB mit Generic Constraints

```
public class Mitarbeiterzuordnung<ChefTyp, AssistentTyp> where ChefTyp: Mitarbeiter, new()
    where AssistentTyp: Mitarbeiter, new()
{
    public ChefTyp Chef;
    public AssistentTyp Assi;
    public Mitarbeiterzuordnung(ChefTyp Chef, AssistentTyp Assi, de.WWWings.Flug flug)
    {
        this.Chef = Chef;
        this.Assi = Assi;
    }
}
```

Listing 6.34 Deklaration einer generischen Klasse in C# mit Generic Constraints

HINWEIS In Generic Constraints sind folgende Angaben erlaubt:

- eine oder mehrere Schnittstellen
- eine Basisklasse
- Schlüsselwort `new` (steht für Typen mit parameterlosem Konstruktor)
- Schlüsselwort `class` (steht für Referenztypen)
- Schlüsselwort `structure` (steht für Wertetypen)

Ko- und Kontravarianz

Gastautor dieses Abschnitts: Manfred Steyer, entnommen aus »NET 4.0 Update« [HS07]

Bisher fehlte es in .NET im Allgemeinen sowie in C# im Speziellen an ko- und kontravariantem Verhalten für Generics. Da es sich hierbei um keine einfach zu verstehenden Konzepte handelt, beschreibt dieses Kapitel anhand einer plastischen Problemstellung die dahinterliegenden Ideen, bevor die damit einhergehenden Schlüsselwörter *in* bzw. *out* vorgestellt werden.

Kovarianz

In Listing 6.35 werden eine Klasse `Tier` sowie die Klassen `Hamster` und `Krokodil`, welche von `Tier` erben, deklariert. In der Methode `Kovarianz` werden zwei `Hamster` instanziiert und an die Methode `Fotographiere()` übergeben. Diese Methode erwartet zwar »lediglich« ein `Tier`, da `Hamster` jedoch von `Tier` erbt, kann es auch als solches verwendet werden. Anschließend wird ein `IEnumerable<Hamster>` erzeugt, welches einem `IEnumerable<Tier>` zugewiesen wird. Letzteres wird anschließend an die Methode `GruppenFoto` übergeben.

```
class Tier { public String Name { get; set; } }
class Hamster : Tier { public void LaufeImRad() { } }
class Krokodil : Tier { public void boeseSein() { } }
[...]
```

```
private static void Kovarianz()
{
    Hamster h1 = new Hamster() { Name = "Krümel" };
    Hamster h2 = new Hamster() { Name = "Goldy" };

    Fotographiere(h1);
    Fotographiere(h2);

    IEnumerable<Hamster> kaefig = new List<Hamster>() { h1, h2 };
    IEnumerable<Tier> tiere = kaefig;
    GruppenFoto(tiere);
}

private static void Fotographiere(Tier t)
{
    Console.WriteLine("Fotographiere „ + t.Name);
}

private static void GruppenFoto(IEnumerable<Tier> tiere)
{
    foreach (var t in tiere)
    {
        Console.WriteLine("Fotographiere „ + t.Name);
    }
}
```

Listing 6.35 Kovariantes Verhalten

Da ein `Hamster` in diesem Beispiel als `Tier` verwendet werden kann, erscheint es auf den ersten Blick auch logisch, dass eine Menge von `Hamstern` (zum Beispiel ein `IEnumerable<Hamster>`) als eine Menge von `Tieren` (zum Beispiel als `IEnumerable<Tier>`) angesprochen werden kann. Bei einer genaueren Betrachtung fällt auf, dass dem nicht zwangsläufig so ist, denn dies würde ja bedeuten, dass es möglich ist, zu einer Menge von `Hamstern`, welche als Menge von `Tieren` angesprochen wird, ein `Krokodil` hinzuzufügen, da es sich dabei

schließlich auch um ein Tier handelt. Da ein Krokodil jedoch kein Hamster ist, ist dies im Sinne der Typsicherheit (sowie im Sinne der Sicherheit der betroffenen Hamster) nicht wünschenswert. Aus diesem Grund waren solche Zuweisungen vor C# 4.0 auch nicht erlaubt. Die Tatsache, dass bestimmte Klassen, wie `IEnumerable<Tier>`, lediglich *lesend* auf die beinhalteten Instanzen zugreifen, und die zuvor geäußerten Bedenken somit gegenstandslos sind, wurde dabei ignoriert.

C# 4.0 berücksichtigt nun diesen Umstand, indem mit dem Schlüsselwort *out* für Typparameter in *Delegates* und *Interfaces* festgelegt werden kann, dass diese lediglich als Rückgabewert und nicht als Übergabeparameter verwendet werden dürfen. Die integrierten Typen `IEnumerable` und `IEnumerator` machen davon bereits Gebrauch. Dies ist auch der Grund dafür, dass Listing 6.35 mit C# 4.0 funktioniert. Listing 6.36 demonstriert die Verwendung dieses neuen Schlüsselwortes anhand der Deklaration der Schnittstellen `IEnumerable` und `IEnumerator` in .NET 4.0.

```
public interface IEnumerable<out T> : IEnumerable
{
    IEnumerator<T> GetEnumerator();
}
public interface IEnumerator<out T> : IEnumerator
{
    bool MoveNext();
    T Current { get; }
}
```

Listing 6.36 Verwendung von *out*

HINWEIS Per Definition kann das Schlüsselwort *out* lediglich für Schnittstellen und Delegationen verwendet werden.

Kontravarianz

Listing 6.37 zeigt eine Klasse `Tier` sowie eine Klasse `Hamster`, welche von `Tier` erbt. Danach wird eine Klasse `TierComparer` deklariert, welche `IComparer<Tier>` implementiert. Implementierungen von `IComparer` werden verwendet, um zwei Instanzen eines Typs im Hinblick auf eine bestimmte Reihenfolge, meist im Zuge eines Sortiervorgangs, zu vergleichen. Die durch diese Schnittstelle vorgegebene Methode `Compare()` nimmt zwei Instanzen entgegen und liefert per Definition einen negativen Wert, wenn die erste Instanz kleiner als die zweite ist. Ist die zweite Instanz größer als die erste, wird ein positiver Wert geliefert; bei Gleichheit Null (0).

In der danach dargestellten Methode `ContraVariance()` wird eine Liste mit zwei `Hamstern` sowie eine Instanz von `TierComparer` erstellt. Um die Liste mit diesem Comparer zu sortieren, wird er nach `IComparer<Hamster>` konvertiert und an die Methode `Sort()` der Liste übergeben. Die Typkonvertierung wird dabei nur durchgeführt, um anzudeuten, dass `Sort()` in diesem Fall einen `IComparer<Hamster>` erwartet.

```
class Tier {
    public int Id { get; set; }
    public String Name { get; set; }
}

class Hamster : Tier {
    public void LaufeImRad() { }
}
```

```

class TierComparer : IComparer<Tier>
{
    public int Compare(Tier x, Tier y)
    {
        if (x.Id < y.Id) return -1;
        if (x.Id > y.Id) return 1;
        return 0;
    }
}
[...]
```

```

private static void ContraVariance()
{
    Hamster h1 = new Hamster() { Name = "Krümel" };
    Hamster h2 = new Hamster() { Name = "Goldy" };

    List<Hamster> kaefig = new List<Hamster>() { h1, h2 };
    TierComparer tierComparer = new TierComparer();
    IComparer<Hamster> comparer = tierComparer;
    kaefig.Sort(comparer);
}

```

Listing 6.37 Kontravariantes Verhalten

Somit werden Hamster mit einem TierComparer, welcher IComparer<Tier> implementiert und innerhalb von Sort() als IComparer<Hamster> angesprochen wird, sortiert. Da ein Hamster auch ein Tier ist, scheint dies kein Problem zu sein. Vor C# 4.0 wäre die dazu nötige Typkonvertierung von IComparer<Tier> auf IComparer<Hamster> jedoch nicht möglich gewesen, denn dies würde ja bedeuten, dass Methoden, von denen erwartet wird, dass sie einen Hamster zurückliefern, stattdessen irgendein Tier, zum Beispiel ein Krokodil, zurückliefern. Dies wäre jedoch im Sinne der Typsicherheit nicht korrekt. In Fällen, in denen ein Typparameter, wie im Falle von IComparer<...>, lediglich für Übergabeparameter und nicht für Rückgabewerte verwendet wird, ist diese Befürchtung jedoch unberechtigt. Ab C# 4.0 können solche Fälle berücksichtigt werden, indem mit dem Schlüsselwort *in* für Typparameter in Schnittstellen oder Delegaten angegeben wird, dass diese nur für Übergabeparameter verwendet werden dürfen. Die integrierte Schnittstelle IComparer<...> macht von dieser Möglichkeit ab .NET 4.0 Gebrauch. Dies ist auch der Grund dafür, dass Listing 6.37 mit C# 4.0 funktioniert. Listing 6.38 demonstriert die Verwendung von *in* anhand dieser in .NET 4.0 enthaltenen Schnittstelle.

```

public interface IComparer<in T>
{
    Compare(T x, T y);
}

```

Listing 6.38 Deklaration von IComparer<...> in .NET 4.0

HINWEIS Per Definition kann das Schlüsselwort *in* lediglich für Schnittstellen und Delegaten verwendet werden.

Objektmengen

Es gibt drei Arten von Objektmengen in C# und VB:

- Einfache Arrays (typisiert)
- Untypisierte Objektmengen
- Typisierte Objektmengen

Einfache Arrays

Einfache Arrays sind Instanzen der Klasse `System.Array`. Alle Arrays sind nun dynamisch bezüglich der Größe, jedoch muss man sie explizit erweitern. Die Anzahl der Dimensionen muss bei der Deklaration festgelegt werden.

TIPP

Die Handhabung der Objektmengen aus dem Namensraum `System.Collections` ist einfacher als die Verwendung von Arrays. Jedoch erwarten einige Methoden in der .NET-Klassenbibliothek Arrays als Parameter. Man kann aber alle Objektmengen in Arrays umwandeln und so mit Objektmengen arbeiten bis zur Parameterübergabe.

Einfache Arrays in VB

Arrays werden deklariert mit runden Klammern hinter dem Typnamen oder dem Variablennamen. Mit geschweiften Klammern ist eine Initialisierung möglich. Erlaubte und gleichwertige Deklarationen sind alle folgenden vier Deklarationen, wobei die letzte Form (automatische Ableitung) seit VB 2010 möglich ist.

```
Dim lottozahlen1 As Byte() = New Byte(6) {23, 48, 3, 19, 20, 6, 9}
Dim lottozahlen2() As Byte = New Byte() {23, 48, 3, 19, 20, 6, 9}
Dim lottozahlen3 As Byte() = {23, 48, 3, 19, 20, 6, 9}
Dim lottozahlen4 = {23, 48, 3, 19, 20, 6, 9}
```

Auch mehrdimensionale Arrays kann man so initialisieren, z.B.

```
Dim DreiLottoReihen = {
    {23, 48, 3, 19, 20, 6, 9},
    {2, 6, 9, 24, 28, 26, 29}, {7, 8, 10, 11, 34, 40, 42}}
```

Listing 6.39 Array Literale

Man kann die Deklaration und die Dimensionierung trennen:

```
Private lottozahlen2() As Integer
myIntegerArray = New lottozahlen2(Anzahl-1) {}
```

HINWEIS

Wichtig ist, dass Sie bei der Neudefinition von Arrays daran denken, dass a) auch hier nicht die Anzahl der Elemente, sondern die Array-Obergrenze angegeben wird, und b), dass Sie die geschweiften Leerklammern hinter der Dimensionsangabe nicht vergessen anzugeben. Für Letzteres sieht der Compiler nämlich anderenfalls den Wert in den Klammern als Argument für einen parametrisierten Konstruktor des entsprechenden Wertetyps, den es aber gar nicht gibt, und er generiert eine Fehlermeldung.

Mit `ReDim` lässt sich die Größe der Dimensionen ändern (auch unter Beibehaltung des bestehenden Inhalts):

```
ReDim Preserve lottozahlen2(20)
```

Arrays beginnen immer bei 0. Die Möglichkeit, ein Array bei einer anderen Untergrenze als 0 beginnen zu lassen, wurde abgeschafft. Ab VB 2005 darf allerdings die Untergrenze 0 explizit angegeben werden. Bei der Dimensionierung zu nennen ist die Obergrenze. Für ein Array mit n Elementen ist also $n-1$ anzugeben (während in C# n angegeben werden muss). Eine Menge von Werten kann in geschweiften Klammern einem Array zugewiesen werden.

Einfache Arrays in C#

Zur Kennzeichnung der Indizes in Arrays verwendet C# die eckigen Klammern. Die Initialisierung erfolgt ebenso wie in VB mit geschweiften Klammern. In .NET-Arrays beginnt die Zählung der Elemente immer bei 0. Einen wichtigen Unterschied gibt es jedoch zwischen VB und C#: In C# ist in der Deklaration die *Anzahl* der Elemente zu nennen, in Visual Basic der *höchste Index* (also Anzahl - 1). Erlaubte und gleichwertige Deklarationen sind:

```
byte[] lottozahlen1 = new byte[7] { 23, 48, 3, 19, 20, 6, 9 };  
byte[] lottozahlen2 = new byte[] { 23, 48, 3, 19, 20, 6, 9 };  
byte[] lottozahlen3 = { 23, 48, 3, 19, 20, 6, 9 };
```

TIPP

Da es für die Schlüsselwörter `ReDim` und `Preserve` kein Äquivalent in C# gibt, muss man hier auf die .NET-Klassenbibliothek zurückgreifen:

```
Array.Resize<byte>(ref lottozahlen3, 20);
```

Objektmengen (untypisiert und typisiert)

Neben den einfachen Arrays kennt .NET Objektmengen im FCL-Namensraum `System.Collections`, die einfacher zu bedienen bzw. mächtiger sind. So ist häufig die Verwendung von `System.Collections.ArrayList` komfortabler als ein einfaches Array, da man bei den Objektmengen Elemente hinzufügen und entfernen kann, ohne die Größe der Menge explizit anpassen zu müssen. Die Objektmengen in `System.Collections` werden nicht durch spezielle Schlüsselwörter in den Sprachen unterstützt.

Während die ursprünglich in .NET 1.0 eingeführten Objektmengen alle untypisiert waren (die Elemente der Liste wurden mit dem allgemeinen Typ `System.Object` verwaltet und dadurch konnte es Typfehler geben), hat Microsoft mit .NET 2.0 so genannte generische Objektmengen eingeführt, die typisiert sind. Bei den generischen Objektmengen (FCL-Namensraum `System.Collections.Generic`) wird durch einen Typparameter bei Deklaration bzw. Instanziierung festgelegt, was die Menge aufnehmen darf. `List<Typ>` bzw. `List<of Typ>` ist das Pendant zur Klasse `ArrayList`.

Mengen werden häufig durch die Methode `Add()` befüllt. C# ab 2008 und Visual Basic ab 2010 bieten hier eine verkürzte Schreibweise mit geschweiften Klammern wie bei einfachen Arrays an (*Collection Initializer*). Diese Verkürzung funktioniert nur, wenn es eine `Add()`-Methode gibt!

```
// Collection Initializer
List<Vorstandsmitglied> = new List<Vorstandsmitglied> { HS, HM, MM };
Vorstandsmitglieder.Add(HP);
```

Listing 6.40 Initialisierung einer typisierten Objektmenge in C# mit vier Objekten, davon drei als Collection Initializer

```
Dim Vorstandsmitglieder As New List(Of Vorstandsmitglied)() From {HS, HM, MM}
Vorstandsmitglieder.Add(HP)
```

Listing 6.41 Initialisierung einer typisierten Objektmenge in VB mit vier Objekten, davon drei als Collection Initializer

Partielle Klassen

Neu seit .NET 2.0 ist auch die Unterstützung für partielle Klassen. Eine partielle Klasse ermöglicht dem Entwickler, den Quellcode einer Klasse auf mehrere Dateien aufzuteilen. Diese Funktionalität wird in Visual Studio verwendet, um in ASP.NET-Webforms den Code vom Layout zu trennen (vgl. Zusatzkapitel »ASP.NET«, das Sie als PDF auf dem Leser-Portal herunterladen können) und um in Windows Forms den durch den Designer erzeugten Code von dem Entwicklercode zu trennen. Entwickler können partielle Klassen auch dazu benutzen, den Code übersichtlicher zu halten oder mit verschiedenen Personen parallel an einer Klasse zu arbeiten.

Partielle Klassen in VB

Während in C# beide Teile der Klasse mit dem Zusatz `partial` versehen werden müssen, reicht es in Visual Basic aus, den Zusatz ab dem zweiten Teil zu verwenden. Verbunden werden können auf diese Weise aber nur Klassen im Quellcode einer Assembly; Sie können keine Klasse in einer referenzierten Assembly erweitern. Letzteres ist nur mit Vererbung möglich.

```
Namespace de.WWings.PassagierSystem
    Partial Public Class Flug
        ' ===== Klassenmitglieder
        Public Shared Fluege As New Fluege
        ' ===== Attribute
        Public FlugNr As String
        Public AbflugOrt As String
        Public ZielOrt As String
        Public Pilot As de.WWings.MitarbeiterSystem.Pilot
    End Class
End Namespace
```

Listing 6.42 Teil 1 der Klasse Flug (Attribute)

```
Namespace de.WWings.PassagierSystem
    Partial Public Class Flug
        ' ===== Konstruktor
        Public Sub New(ByVal FlugNr As String, ByVal AbflugOrt As String, ByVal ZielOrt As String)
            Me.FlugNr = FlugNr
            Me.AbflugOrt = AbflugOrt
        End Sub
    End Class
End Namespace
```

```
Me.ZielOrt = ZielOrt
Fluege.Add(Me.FlugNr, Me)
End Sub
' ===== Operatorüberladung
Shared Operator (ByVal flug As Flug, ByVal pass As Passagier) As Flug
    pass.Buchen(flug)
    Return flug
End Operator
End Class
End Namespace
```

Listing 6.43 Teil 2 der Klasse Flug (Methoden)

Partielle Klassen in C#

Ebenso wie in VB ist die Unterstützung für partielle Klassen in C# vorhanden. Hinsichtlich der Einsatzgebiete gibt es keinen Unterschied. Syntaktisch gibt es zu VB drei kleine Unterschiede:

- `partial` muss klein geschrieben werden
- `partial` muss hinter den Sichtbarkeitsmodifizierern der Klasse stehen
- `partial` muss bei allen Teilklassen angegeben werden

Beispiel

```
public partial class Flug {...}
```

Partielle Methoden

Neu seit .NET 3.5 sind partielle Methoden. Im Rahmen eines Teils einer partiellen Klasse kann man eine Methode deklarieren (ohne Implementierung). Im Rahmen eines anderen Teils kann man die Implementierung liefern. So lassen sich die Deklaration und die Implementierung trennen. Die partielle Methode kann gleichwohl in dem Teil, in dem sie nur deklariert ist, aufgerufen werden. Wenn es keine Implementierung in einem anderen Teil gibt, kommt es aber nicht zu einem Fehler. Der Compiler wird vielmehr den Aufruf entfernen. Damit kann man partielle Methoden als *Hooks* einsetzen, um sich in Programmcode einzuklinken. Gerne wird dies benutzt bei Programmcode, der von einem Codegenerator (Assistenten oder Designer) erzeugt wurde. Zum ersten Mal eingesetzt wird diese Vorgehensweise im LINQ to SQL-Designer (siehe Kapitel zum Objektrelationalen Mapping – ORM/als PDF!).

HINWEIS

Partielle Attribute (Propertyts) gibt es leider bisher nicht.

Partielle Methoden in VB

Es gibt folgende Bedingungen für partielle Methoden in VB:

- Die Methode darf keinen Rückgabewert (nur `Sub` nicht `Function`) haben
- Nur die Deklaration darf `Partial` verwenden

- Die Methode muss explizit als Private deklariert sein
- Sie können statisch (Shared) sein

```

Partial Friend Class Vorstandsmitglied
...
    Public Overrides Function ToString() As String
        ' Partielle Methode - Verwendung
        OnToString()
        Return Me.Name
    End Function

    ' Partielle Methode - Deklaration
    Partial Private Sub OnToString()

    End Sub
End Class

Partial Class Vorstandsmitglied
    ' Partielle Methode - Implementierung
    Private Sub OnToString()
        Console.WriteLine("ToString aufgerufen!")
    End Sub
End Class

```

Listing 6.44 Beispiel für eine partielle Methode in VB 9.0

Partielle Methoden in C#

Es gibt folgende Bedingungen für partielle Methoden in C#:

- Die Methode darf keinen Rückgabewert (void) haben
- Beide Teile müssen partial verwenden
- Die Methode ist automatisch private. Sie dürfen nicht öffentlich sein.
- Eine Sichtbarkeit darf nicht angegeben sein (also auch nicht private)
- Sie können statisch sein

```

public partial class Vorstandsmitglied
{
    // Automatic Properties
    public string Name { get; set; }
    public string Aufgabengebiet { get; set; }
    public int Alter { get; set; }
    public string Ort;

    public override string ToString()
    {
        // Partielle Methode - Verwendung
        OnToString();
        return Name;
    }
}

```



```
// Partielle Methode - Deklaration
partial void OnToString();
}

public partial class Vorstandsmitglied
{
    // Partielle Methode - Implementierung
    partial void OnToString()
    {
        Console.WriteLine("ToString aufgerufen!");
    }
}
```

Listing 6.45 Beispiel für eine partielle Methode in C# 3.0

Anonyme Typen

Neu seit C# 3.0 und VB 9.0 ist, dass man Objekte ohne eine explizite Klassendefinition erzeugen kann. Solche Klassen erhalten automatisch einen Klassennamen vom dem Compiler. Dieser Name ist recht kompliziert und nicht zur Verwendung durch den Entwickler gedacht.

Ein anonymer Typ entsteht in C# durch Verwendung von `new` ohne Klassennamen und in VB durch `New With`.

```
// Anonyme Typen
var Fluggesellschaft = new { Name = "World Wide Wings",
                             Gruendungsdatum = new DateTime(2005, 01, 01),
                             Vorstand = Vorstandsmitglieder };
Console.WriteLine(Fluggesellschaft.GetType().FullName);

// 2., gleich aufgebauter anonymer Typ
var Flugzeugbauer = new { Name = "Strong Winds Corp.", Gruendungsdatum = new DateTime(1972, 08, 01),
                          Vorstand = new List<Vorstandsmitglied>() };
Console.WriteLine(Flugzeugbauer.GetType().FullName);
```

Listing 6.46 Anonyme Typen in C# 3.0

```
' Anonyme Typen
Dim Fluggesellschaft = New With {Key .Name = "World Wide Wings",
                                .Gruendungsdatum = New DateTime(2005, 1, 1), .Vorstand = Vorstandsmitglieder}
Console.WriteLine(Fluggesellschaft.GetType().FullName)

' 2., gleich aufgebauter anonymer Typ
Dim Flugzeugbauer = New With {Key .Name = "Strong Winds Corp.",
                              .Gruendungsdatum = New DateTime(1972, 8, 1), .Vorstand = Nothing}
Console.WriteLine(Flugzeugbauer.GetType().FullName)
```

Listing 6.47 Anonyme Typen in VB 9.0

Durch die obigen Listings entsteht ein anonymer Typ mit diesem Namen:

```
<?f AnonymousType0`3[[System.String, mscorlib, Version=2.0.0.0, Culture=neutral
, PublicKeyToken=b77a5c561934e089],[System.DateTime, mscorlib, Version=2.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089],[System.Collections.Generic.Li
st`1[[NET3.SpracheCSharp.Demo Sprachfeatures.Vorstandsmitglied, WWings.Verschie
deneDemos, Version=0.5.0.0, Culture=neutral, PublicKeyToken=null]], mscorlib, Ve
rsion=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089]]
```

HINWEIS

Dabei ist Folgendes zu beachten:

- Die Initialisierung kann mit statischen Werten oder Variablen erfolgen
- Zwei auf die o.g. Weise instanziierte Objekte gehören zur gleichen Klasse, wenn sie die gleiche Anzahl und Reihenfolge von Attributen bei der Instanziierung besitzen
- Auf diese Weise instanziierte Objekte können nicht mehr verändert werden, weil alle Property-Attribute nur für den Lesezugriff erzeugt werden
- Auf diese Weise instanziierte Objekte sind nicht serialisierbar, weil es keinen parameterlosen Standardkonstruktor gibt
- Der Name eines anonymen Typen wird bei jedem Kompilierungsvorgang neu vergeben. Man darf sich daher nicht auf das Ergebnis von `GetType()` verlassen.
- Man kann komplexe anonyme Typen durch Verschachtelung erzeugen
- Man kann auch ein Array aus anonymen Typen bilden und – mit einem hier aus Platzgründen nicht gezeigten Trick – auch anonyme Typen in andere Objektmengen aufnehmen
- Anonyme Typen sind nur für lokale Variablen erlaubt. Sie sind nicht einsetzbar als Klassenmitglieder, Parameter von Methoden und Rückgabewerte von Methoden.

Implementierungsvererbung

Anders als in C++, aber wie in Java und C#/Visual Basic ist die Mehrfachvererbung, also die gleichzeitige Ableitung einer Klasse von mehreren anderen Klassen, *nicht* möglich. Die Implementierungsvererbung stellt alle Attribute, Methoden und Ereignisse auch für die erbende Klasse bereit. Nicht vererbt werden jedoch die Konstruktoren. Zirkuläres Erben (Class A : Inherits B ... Class B : Inherits A) ist nicht sinnvoll und daher auch nicht erlaubt.

Sowohl auf Klassen als auch auf Mitgliederebene kann eine Klasse steuern, wie man von ihr erben kann. Im Standard kann man von einer Klasse erben, man muss es aber nicht. Auf Klassenebene bedeutet `MustInherit` (C#: `abstract`), dass eine Klasse nicht direkt verwendet werden kann, sondern nur der Vererbung dient. `NotInheritable` (C#: `sealed`) bedeutet, dass ein Erben nicht möglich ist.

Für Methoden gelten etwas andere Spielregeln: `Overridable` (C#: `virtual`) legt fest, dass eine Unterklasse eine Methode überschreiben (also reimplementieren) darf (siehe Methode `Info()` im Listing). `MustOverride` (C#: `abstract`) bedeutet, dass die Unterklasse die Methode überschreiben muss (abstrakte Methode). `NotOverridable` (C#: `sealed`) legt fest, dass eine Methode versiegelt ist, also nicht überschrieben werden kann. Da dies die Grundeinstellung ist, müssen `NotOverridable` bzw. `sealed` nicht explizit genannt werden.

Vererbung in VB

Vor VB7 unterstützt Visual Basic keine Implementierungsvererbung. In Visual Basic .NET dient dazu das neue Schlüsselwort `Inherits`. Mithilfe des Doppelpunkts als Befehlstrennzeichen können `Class` und `Inherits` in einer Zeile stehen. Wenn Vererbung verwendet wird, muss `Inherits` die erste Anweisung nach dem Klassennamen sein.

```
Class B : Inherits A
```

Beispiel 1

Das folgende Beispiel zeigt die Implementierung der Klasse `Passagier`, die von `Person` erbt. Die Klasse enthält ein Klassenmitglied `Passagiere`, in dem alle Instanzen der Klasse verwaltet werden. Der Konstruktor sorgt dafür, dass jeder neu erzeugte `Passagier` in die Objektmenge aufgenommen wird. Ein als `ReadOnly` markiertes Attribut kann nur innerhalb des Konstruktors gesetzt werden und eignet sich daher für Daten, die später weder durch Clients noch durch das Objekt selbst geändert werden dürfen. Ein `private` Mitglied kann hingegen von den Clients nicht verwendet werden.

Der Konstruktor ruft mithilfe des Schlüsselwortes `MyBase` den Konstruktor der zu Grunde liegenden Basis-Klasse `Person` auf. Dieser Befehl muss immer der erste Befehl in der Implementierung eines Konstruktors sein. `MyClass` ermöglicht den Rückbezug auf Konstruktoren in der aktuellen Klasse. Wenn `MyBase.New()` oder `MyClass.New()` verwendet werden, müssen diese Befehle die ersten im Konstruktor sein.

`Me` bezeichnet die aktuelle Instanz der Klasse. Das Schlüsselwort `Overrides` ist für `Sub Info` notwendig, um die vorhandene Implementierung der Oberklasse zu überschreiben.

Mit VB7 eingeführt wurde auch die Möglichkeit, Methoden zu überladen. In diesem Beispiel existiert `Buchen()` mit zwei unterschiedlichen Methodensignaturen.

```
Namespace de.WWWings.PassagierSystem
    Public Class Passagier
        Inherits de.WWWings.Person
        ' ===== Klassenmitglieder
        Public Shared Passagiere As New Passagiere
        ' ===== Attribute
        Public Fluege As New Fluege
        Public ReadOnly PID As Long
        Private _AktuellerFlug As Flug
        ' ===== Ereignisse
        Public Shared Event CheckInStart(ByVal pass As Passagier)
        Public Shared Event CheckInEnde(ByVal pass As Passagier)
        ' ===== Konstruktoren
        Public Sub New(ByVal Name As String, ByVal vorname As String)
            MyBase.New(Name, vorname)
            Me.PID = Passagier.Passagiere.Add(Me)
        End Sub
        ' ===== Errechnete Attribute (Properties)
        Public ReadOnly Property AktuellerFlug() As Flug
            Get
                Return Me._AktuellerFlug
            End Get
        End Property
        ' ===== Methoden
        Public Overloads Sub Buchen(ByVal flug As Flug)
            Me.Fluege.Add(flug.FlugNr, flug)
        End Sub
        Public Overloads Sub Buchen(ByVal Flugnummer As String)
            If Not Flug.Fluege.ContainsKey(Flugnummer) Then
                Throw New FalscheFlugnummer(Me.PID & "/" & Flugnummer)
            Else
                Me.Buchen(Flug.Fluege.Item(Flugnummer))
            End If
        End Sub
    End Class
    ''' <summary>Einchecken eines Passagiers für einen Flug</summary>
    ''' <param name="Text">Die Methode erwartet eine Flugnummer</param>
    ''' <returns>Die Methode liefert als Rückgabewert das Flugobjekt, wenn das
    ''' Einchecken erfolgreich war</returns>
```

```

Public Function CheckIn(ByVal Flugnummer As String) As Flug
    RaiseEvent CheckInStart(Me)
    If Not Me.Fluege.ContainsKey(Flugnummer) Then
        Throw New PassagierNichtAufFlugGebucht(Me.PID & "/" & Flugnummer)
    Else
        RaiseEvent CheckInEnde(Me)
        Return Me.Fluege.Item(Flugnummer)
    End If
End Function

' ===== Methoden
Public Overrides Sub Info()
    Console.WriteLine("Passagier: " & Me.GanzerName)
End Sub

End Class
End Namespace

```

Listing 6.48 Implementierung der Klasse *Passagier*

Beispiel 2

Im zweiten Beispiel wird die Objektmenge Passagiere durch Vererbung von der in der .NET-Klassenbibliothek definierten generischen Mengenklasse `System.Collections.Generic.SortedDictionary` erzeugt. Generische Mengenklassen (seit .NET 2.0) erlauben zur Entwicklungszeit die Festlegung der möglichen Inhaltstypen, während die in .NET 1.x vorhandenen Mengenklassen immer untypisiert waren. Zur Festlegung der Typen einer generischen Klasse kommt wieder das neue Schlüsselwort `of` zum Einsatz.

Neben der generischen Basisklasse ist in dem Beispiel das Schlüsselwort `Shadows` interessant, mit dem in einer Unterklasse ein in der Oberklasse vorhandenes Klassenmitglied verdeckt werden kann. In diesem Fall könnte `Overrides` nicht eingesetzt werden, weil die neue `Add()`-Methode nur einen Parameter besitzt, aber `Add()` mit zwei Parametern verdecken soll. `MyBase` kann innerhalb der verdeckenden Methode genutzt werden, um auf die verdeckte Methode zurückzugreifen.

```

Public Class Passagiere
    Inherits System.Collections.Generic.SortedList(Of Long, Passagier)
    Private Function NaechstePassagierNummer() As Long
        If MyBase.Count = 0 Then
            Return 1
        Else
            Return MyBase.Keys(MyBase.Count - 1) + 1
        End If
    End Function
    Public Shadows Function Add(ByVal P As Passagier) As Long
        Dim pnr As Long = Me.NaechstePassagierNummer
        MyBase.Add(pnr, P)
        Return pnr
    End Function
End Class

```

Listing 6.49 Implementierung der Klasse *Passagiere*

In der Grundeinstellung kann von jeder Klasse geerbt werden. Eine Klasse, die nicht vererbbar sein soll, muss mit dem Schlüsselwort `NotInheritable` versehen werden. `MustInherit` bezeichnet dagegen eine abstrakte Oberklasse, von der keine Instanzen direkt erzeugt werden können und deren ausschließlicher Zweck die Vererbung darstellt.

Vererbung in C#

C# unterstützt ebenso wie VB die Einfachvererbung, nicht aber Mehrfachvererbung. Die Implementierungsvererbung wird angezeigt durch einen Doppelpunkt nach dem Klassennamen. Der Doppelpunkt dient auch der Anzeige von Schnittstellenvererbung, entspricht also sowohl dem VB-Schlüsselwort `Inherits` als auch `Implements`. Zum Dritten wird der Doppelpunkt eingesetzt, um in einem Konstruktor einen anderen Konstruktor aufzurufen. Nach dem Doppelpunkt kann auf `this` (aktuelle Klasse) und `base` (Basisklasse) Bezug genommen werden. Durch diese Syntaxform wird sichergestellt, dass der Aufruf des anderen Konstruktors immer der erste Befehl in einem Konstruktor ist. Die Anforderung, dass der Aufruf eines anderen Konstruktors der erste Befehl sein muss, existiert auch in C#; dort jedoch gibt es dafür keine spezielle Syntax, sondern die Befehlsreihenfolge wird durch den Compiler geprüft.

```
#region Using directives

using System;
using System.Collections.Generic;
using System.Text;
using de.WWWings.PassagierSystem;
using de.WWWings;

#endregion

namespace de.WWWings.PassagierSystem
{
    public class Passagier : de.WWWings.Person
    {
        // ===== Klassenmitglieder
        public static de.WWWings.PassagierSystem.Passagiere Passagiere = new Passagiere();
        // ===== Attribute (Fields)
        public de.WWWings.Fluege Fluege = new de.WWWings.Fluege();
        public readonly long PID;
        private de.WWWings.Flug _AktuellerFlug;
        // ===== Errechnete Attribute (Properties)
        public Flug AktuellerFlug
        {
            get
            {
                return this._AktuellerFlug;
            }
        }
        // ===== Konstruktoren
        public Passagier(string Name, string Vorname) : base(Name, Vorname)
        {
            this.PID = Passagier.Passagiere.Add(this);
        }
        // ===== Methoden
        public void Buchen(de.WWWings.Flug flug)
        {
            this.Fluege.Add(flug.FlugNr, flug);
        }
        public void Buchen(string Flugnummer)
        {
            if (!(Flug.Fluege.Contains(Flugnummer)))
            {
                throw new de.NETFly.PassagierSystem.FalscheFlugnummer(this.PID + "/" + Flugnummer);
            }
        }
    }
}
```

```

else
{
    this.Buchen(de.WWWings.Flug.Fluege[Flugnummer]);
}
public Flug CheckIn(string Flugnummer)
{
    if (!(this.Fluege.ContainsKey(Flugnummer)))
    {
        throw new de.NETFly.PassagierSystem.PassagierNichtAufFlugGebucht(this.PID + "/" + Flugnummer);
    }
    else
    {
        return this.Fluege[Flugnummer];
    }
}
public override void Info()
{
    Console.WriteLine("Passagier: " + this.GanzerName);
}
}
}

```

Listing 6.50 Implementierung der Klasse *Passagier* in C#

Ereignisse

Klassen oder einzelne Objekte können Ereignisse auslösen, die von anderen abonniert werden können. Zu einem Ereignis kann es beliebig viele Abonnenten in beliebig vielen Objekten geben. In diesem Fall ruft das Objekt Unterroutrinen in allen Abonnenten auf, wenn eine bestimmte Situation eintritt.

Ereignisse in VB

Für den Ereignismechanismus im .NET Framework sind für VB vier Bausteine wichtig:

- Deklaration eines Ereignismitglieds in der das Ereignis auslösenden Klasse (ggf. mit Parametern)

```
Public Shared Event CheckInStart(ByVal pass As Passagier)
```

- Auslösen des Ereignisses in einer Methode der das Ereignis definierenden Klasse (ggf. mit Werten für die Parameter)

```
RaiseEvent CheckInStart(Me)
```

- Deklaration einer Ereignisbehandlungsroutine im Client gemäß der Signatur des Ereignisses

```
Shared Sub CheckInStartHandler(ByVal pass As Passagier)
```

- Bindung der Ereignisbehandlungsroutine an das Ereignis im Client

```
AddHandler Passagier.CheckInStart, AddressOf CheckInStartHandler
```

Für den letzten Schritt (Ereignisbindung) existiert in Visual Basic syntaktisch die Alternative über `WithEvents` und `Handles`:

```
Dim WithEvents pass As Passagier
...
Shared Sub CheckInEndeHandler(ByVal pass As Passagier) Handles pass.CheckInEnde
```

Diese Möglichkeit ist eleganter, ermöglicht aber nur, die Ereignisse in einer bestimmten Instanz zu behandeln. Ereignisse, die statisch zur Klasse gehören, können darüber nicht gebunden werden.

Ereignisse in C#

Die Definition und die Behandlung von Ereignissen ist in C# komplizierter im Vergleich zu der Vorgehensweise in Visual Basic. In C# 2005 wurde lediglich eine kleine Verbesserung eingeführt.

Definition von Ereignissen

Wenn eine Klasse ein Ereignis auslösen möchte, muss sie zunächst für jeden Ereignistyp einen so genannten *Delegaten* deklarieren:

```
public delegate void CheckInStartHandler(Passagier p);
public delegate void CheckInEndeHandler(Passagier p);
```

Dann muss die Klasse wie in VB 2005 die Ereignisse als Klassenmitglieder deklarieren, wobei die Parameter hier durch Bezug auf einen Delegaten festgelegt werden.

```
public static event CheckInStartHandler CheckInStart;
public static event CheckInEndeHandler CheckInEnde;
```

TIPP

Eine Vereinfachung bei der Deklaration von Ereignissen ist möglich durch die generische Klasse `EventHandler<>`.

```
public static event EventHandler<Passagier> CheckInEndeAlternativ;
```

Hier kann man auf die Deklaration eines Delegaten verzichten. Dann gelten aber folgende Nebenbedingungen:

- Die Methodensignatur für das Ereignis ist dann `CheckInEndeAlternativ(object Sender, Passagier p)`
- Die Klasse `Passagier` muss von `System.EventArgs` abgeleitet sein

Ereignis auslösen

Ein spezielles Schlüsselwort zum Auslösen eines Ereignisses (vgl. `RaiseEvent` in VB) existiert in C# nicht. Zum Auslösen des Ereignisses kann zwar das Ereignis wie eine Methode aufgerufen werden; zuvor muss aber der Entwickler prüfen, ob überhaupt jemand für das Ereignis registriert ist.

```
if (CheckInStart != null) { CheckInStart(this); }
```

Ereignisbehandlung

Auch für die Ereignisbehandlung existieren in C# keine speziellen Schlüsselwörter wie `AddHandler`, `WithEvents` und `Handles` in VB. In C# muss der Delegat instanziiert werden mit der Ereignisbehandlungsroutine als Parameter und diese so gebildete Instanz muss der Ereignisvariablen der Klasse mit dem Operator `+` hinzugefügt werden.

```
Passagier.CheckInStart += new Passagier.CheckInStartHandler(CheckInGestartet);
...
static void CheckInGestartet(Passagier pass)
{
    Demo.Print("Check-In beginnt... für " + pass.GanzerName); }
}
```

Listing 6.51 Bindung einer Ereignisbehandlungsroutine

Neuerungen seit C# 2005

C# unterstützt ab Version 2005 zur Ereignisbehandlung auch so genannte *anonyme Methoden*, mit denen Programmcode direkt einem Delegaten zugewiesen werden kann. Anstelle des Verweises auf eine entsprechende Ereignisbehandlungsroutine kann der Entwickler mit dem Schlüsselwort `delegate` nun direkt einen Codeblock (anonyme Methode) binden. Wenn mehrere Ereignisse den gleichen Code ausführen sollen, ist die Implementierung der anonymen Methode auf den Aufruf einer Methode zu beschränken.

```
public static void Run()
{
    ...
    Passagier.CheckInEnde += delegate (Passagier CheckedInPassagier)
    {
        Int16 AnzahlPass = 0;
        AnzahlPass += 1;
        Demo.Print(AnzahlPass + ". Passagier: " + CheckedInPassagier.GanzerName);
    };
    Passagier p1 = new Passagier("Schröder", "Gerhard");
    p1.CheckIn("NF1234");
    ...
}
```

Listing 6.52 Beispiel für die Zuweisung einer anonymen Methode an das Ereignis `CheckInEnde()` in der Klasse `Passagier`

Schnittstellen (Interfaces)

Während das .NET Framework nur die einfache Implementierungsvererbung unterstützt, gibt es Mehrfachvererbung für Schnittstellen, d.h., eine Klasse kann optional eine oder mehrere Schnittstellen implementieren. Eine Schnittstelle kann auch von mehreren anderen Schnittstellen erben.

Schnittstellen in VB

Schnittstellen können seit VB7 direkt, ohne Umweg über Klassen, definiert werden. Es gibt dazu ein neues Schlüsselwort `Interface`, das ähnlich wie `Class` einen Block (`Interface...End Interface`) bildet.


```
Imports System
Namespace de.WWwings.MitarbeiterSystem
    Interface IPilot
        Property FlugscheinSeit() As DateTime
        Property FlugscheinTyp() As String
        Property Flugstunden() As Long
        Sub FlugZuweisen(ByVal Flug As de.WWwings.Flug)
    End Interface
End Namespace
```

Klassen zeigen wie in VB6 durch Implements an, dass sie eine Schnittstelle implementieren wollen. Wie in vielen anderen Programmiersprachen auch üblich, sollten Schnittstellennamen mit einem großen I beginnen.

```
Public Class Pilot
    Implements IPilot
```

Schnittstellen in C#

Eine Schnittstelle wird in C# ebenfalls durch einen interface-Block deklariert und darf sowohl Attribute als auch Methoden enthalten. Modifizierer hinsichtlich der Sichtbarkeit (public, protected, private etc.) sind nicht erlaubt.

```
interface IPilot
{
    // ===== Attribute
    DateTime FlugscheinSeit { get; set; }
    string FlugscheinTyp { get; set; }
    long Flugstunden { get; set; }
    // ===== Methoden
    void FlugZuweisen(de.WWwings.Flug Flug);
}
```

Listing 6.53 Implementierung der Schnittstelle IPilot in C#

Eine Klasse zeigt durch einen Doppelpunkt hinter dem Namen an, dass sie eine Schnittstelle implementieren will.

```
public class Pilot : Mitarbeiter, IPilot
```

HINWEIS Der Compiler unterscheidet dabei automatisch, ob der Bezeichner nach dem Doppelpunkt eine Klasse oder eine Schnittstelle ist.

Namensräume (Namespaces)

Namensräume dienen der hierarchischen Benennung von Klassen (vgl. allgemeine Erläuterungen im Kapitel 4 »Grundkonzepte des .NET Framework 4.0«).

Namensräume deklarieren

Die Deklaration eines Namensraums dient dazu, eine Klasse einem Namensraum zuzuordnen. Jede Klasse gehört nur zu genau einem Namensraum.

Namensräume deklarieren in VB

Die Festlegung des Namensraums für eine Klasse erfolgt in Visual Basic durch das Konstrukt `Namespace...End Namespace`.

```
''' <summary>Diese Klasse repräsentiert einen Passagier.</summary>
<System.Serializable> _
Public Class Passenger
...
End Class
End Namespace
```

WICHTIG

Bei der Verwendung von Visual Studio ist zu beachten, dass in den Projekteigenschaften ein Wurzelnamensraum festgelegt werden kann, der dem nach dem Schlüsselwort `Namespace` genannten Namen vorangestellt wird. Der Standardnamensraum entspricht dem Projektnamen beim Anlegen des Projekts, kann aber jederzeit geändert werden. Oft ist es sinnvoll, den Standardnamensraum auf eine leere Zeichenkette zu setzen.

Namensräume deklarieren in C#

Die Festlegung des Namensraums für eine Klasse erfolgt in C# durch das Konstrukt `namespace { ... }`.

```
namespace de.WWWings.PassagierSystem
{
    public class Passagier : de.WWWings.Person
    { ... }
}
```

HINWEIS

Anders als bei Visual Basic-Projekten kann man in Visual Studio für C#-Projekte in den Projekteigenschaften keinen Wurzelnamensraum definieren, der allen Namensraumdeklarationen vorangestellt wird, sondern nur einen Standardnamensraum, der beim Anlegen neuer Klassen verwendet wird. Der Standardnamensraum wird nicht automatisch allen Namensraumdeklarationen vorangestellt.

Import von Namensräumen

Im Normalfall müssen Klassen in .NET immer mit ihrem vollen Namensraum genannt werden. Das Importieren von Namensräumen hat das Ziel, einen Klassennamen mit verkürztem oder ganz ohne Namensraum zu verwenden.

Import von Namensräumen in VB

Der Import von Namensräumen erfolgt in Visual Basic mit dem Schlüsselwort `Imports`.

Imports-Anweisung	Typnutzung
Ohne	System.Collections.Generic.SortedList(of string, of Flug)
Imports System.Collections.Generic	SortedList(of string, of Flug)

Tabelle 6.8 Einsatz der Imports-Anweisung für Klassen

Der Visual Basic-Compiler referenziert immer automatisch die *Microsoft.VisualBasic.dll*, in der die Implementierung aller Hilfsfunktionen steckt, die seit VB 7.x nicht mehr zum Sprachumfang gehören und nur aus Kompatibilitätsgründen aus VB6 übernommen wurden. Zusätzlich sollten Sie immer zu Beginn einer jeden Visual Basic-Codedatei den Namensraum *Microsoft.VisualBasic* importieren. Ohne diese Anweisung müssten Sie eine Funktion wie *MsgBox()* mit ihrem vollqualifizierten Namen aufrufen. In Visual Studio wird für Visual Basic-Projekte automatisch ein globaler Import dieses Namensraums in das Projekt eingetragen.

Mit Import-Anweisung	Ohne Import-Anweisung
Imports Microsoft.VisualBasic Msgbox("Hello") Msgbox("World")	Microsoft.VisualBasic.Msgbox("Hello") Microsoft.VisualBasic.Msgbox("World")

Tabelle 6.9 Einsatz der Imports-Anweisung für Funktionen

Import von Namensräumen in C#

Das Importieren von Namensräumen erfolgt in C# mit dem Schlüsselwort *using*. Dabei ist es möglich, einen Alias-Namen für einen Namensraum zu vergeben.

```
using System.Collections.Generic;
using GenCol = System.Collections.Generic;
```

Import-Anweisung	Typnutzung
Ohne	System.Collections.Generic.SortedList<string, Flug>
using System.Collections.Generic;	SortedList<string, Flug>
using GenCol = System.Collections.Generic;	GenCol.SortedList<string, Flug>

Tabelle 6.10 Beispiele für den Einsatz von Import

Verweis auf Wurzelnamensräume

Wurzelnamensräume sollten eindeutig sein. Deshalb ist es empfehlenswert, dem Namensraum die Internet-Domain voranzustellen (z. B. *de.WWWings.PassagierSystem*). Dabei sollte man Namensdopplungen auch für untergeordnete Namensräume vermeiden, weil es sonst unter bestimmten Bedingungen zweideutige Interpretationen einer Anweisung geben könnte. Insbesondere sollte man die Begriffe *System* und *Microsoft* vermeiden, weil damit die FCL-Namensräume verdeckt werden.

Beispiel

Wenn man »versehentlich« einen Namensraum wie `de.WWWings.System` definiert hat, kann man aus diesem Namensraum heraus nicht mehr auf den FCL-Namensraum `System` zugreifen (siehe Abbildung),

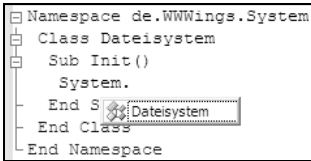


Abbildung 6.7 Der FCL-Namensraum `System` ist durch den Namensraum `de.WWWings.System` verdeckt

Verweis auf Wurzelnamensräume in VB

Seit Visual Basic 2005 gibt es eine Lösung für diese missliche Situation: Über die Voranstellung des Präfix `Global` ist es noch möglich, die FCL-Klassen zu verwenden (z.B. `Global.System.Int32`).

Verweis auf Wurzelnamensräume in C#

Das Schlüsselwort `global::` übernimmt ab C# 2005 die gleiche Funktion wie `global.` ab Visual Basic 2005: Mit diesem dem Namensraum vorangestellten Schlüsselwort adressiert man einen Wurzelnamensraum, wenn dieser durch einen untergeordneten Namensraum verdeckt ist.

Operatorüberladung

Operatorüberladung bedeutet, einem der Standardoperatoren wie `+`, `-`, `*` und `=` im Zusammenhang mit selbstdefinierten Klassen eine neue Bedeutung zu geben, z.B. ein Flug-Objekt und ein Passagier-Objekt zu addieren, um daraus ein neues Objekt des Typs `Buchung` zu gewinnen.

WICHTIG Zum Thema Operatorüberladung gibt es geteilte Meinungen. Von einigen Entwicklern wird sie geliebt wegen der Prägnanz. Von anderen Entwicklern wird sie gehasst wegen der Mehrfachbedeutung der Operatoren, die die Lesbarkeit des Programmcodes erschwert. Festzuhalten ist auf jeden Fall, dass man Operatorüberladung nicht zwingend braucht; alles was Operatorüberladung kann, kann man auch durch eine Methode mit einem sprechenden Namen ausdrücken.

Operatorüberladung in VB

Seit VB Version 2005 war es auch möglich, Operatoren zu überladen.

Die letzte Zeile in dem nachfolgenden Code-Ausschnitt würde in VB 7.x zu einem Fehler führen.

```
Dim f1 As New Flug("NF1234", "Düsseldorf", "München")
Dim f2 As New Flug("NF5678", "Düsseldorf", "New York")
Dim p1 As New Passagier("Schröder", "Gerhard")
Dim p2 As New Passagier("Merkel", "Angela")
p1.Buchen("NF1234")
p1.Buchen("NF5678")
f1 = f1 + p2 ' Entspricht p2.Buchen("NF1234")
```

Ab VB 2005 ist dagegen die *Addition* eines Passagier-Objekts zu einem Flug-Objekt möglich, wenn in der Klasse Flug der Operator + neu definiert wird mit dem Konstrukt `Operator...End Operator`. In dem nachstehenden Fall wird der Operator auf einen Methodenaufruf in der Passagier-Klasse abgebildet.

```
Public Class Flug
...
    ' ===== Operatorüberladung
    Shared Operator +(ByVal flug As Flug, ByVal pass As Passagier) As Flug
        pass.Buchen(flug)
        Return flug
    End Operator
End Class
```

Listing 6.54 Beispiel für Operatorüberladung in VB

Operatorüberladung in C#

C# bietet seit seiner ersten Version eine prägnante Syntax für die Definition einer Operatorüberladung.

```
// ===== Operatorüberladung
public static Flug operator +(Flug flug, de.WWWings.PassagierSystem.Passagier pass)
{
    pass.Buchen(flug);
    return flug;
}
```

Listing 6.55 Beispiel für Operatorüberladung in C#

Schleifen

Sowohl VB als auch C# unterstützen vier Typen von Schleifen:

- Kopfgeprüfte bedingte Schleifen
- Fußgeprüfte bedingte Schleifen
- Zählschleifen: Schleife mit einer bestimmten Anzahl von Durchläufen
- Mengenschleifen: Schleifen über alle Mitglieder eines Arrays oder andere Objektmenge, welche die `IEnumerable`-Schnittstelle unterstützen (insbesondere die Klassen aus dem FCL-Namensraum `System.Collections`).

HINWEIS

Um eine aufzählbare Objektmengenkategorie zu implementieren, leitet man diese von einer bestehenden aufzählbaren Klasse (aus dem Namensraum `System.Collections`) ab oder implementiert `IEnumerable` selbst unter Verwendung des Schlüsselworts `yield`, das mit C# 2005 neu eingeführt wurde.

Schleifen in VB

Bei den Schleifen sind in Visual Basic 2008 folgende Punkte zu beachten:

- Erlaubte bedingte Schleifenkonstrukte sind `Do...Loop` und `While...End While` (`While...Wend` ist seit VB7 nicht mehr erlaubt)
- `For...Next` ist die Zählschleife
- `For Each...Next` dient der Iteration über Arrays und Objektmengen
- Eine innerhalb eines Schleifenblocks deklarierte Variable ist nur innerhalb der Schleife gültig, nicht in der ganzen Unteroutine
- Ab VB 7.1 ist es möglich, eine Laufvariable innerhalb des Schleifenkopfes zu deklarieren

```
For Each p As Passagier In Passagier.Passagiere.Values
...
Next
```

Die Laufvariable ist dabei nur innerhalb der Schleife gültig. Generell ist in dem .NET-basierten Visual Basic die Gültigkeit (der *Scope*) einer Variable auf den aktuellen Block begrenzt, in dem die Variable deklariert wurde.

- Eine Schleife kann mit den Konstrukten `Exit For`, `Exit Do` und `Exit While` vorzeitig verlassen werden
- Ab VB 2005 existiert das `Continue`-Schlüsselwort, um eine Schleife vorzeitig fortzusetzen

Schleifen in C#

C# unterstützt diese Schleifentypen:

- **Bedingte Schleifen** `while (bedingung) { ... }` sowie `do { ... } while(Bedingung)`
- **Zählschleife** `for ([Initialisierung];[Abbruchbedingung];[Iteration]) { ... }`
- **Mengenschleife** `foreach (x in y) { ... }`

Das Besondere an der `for`-Schleife ist, dass alle drei Bestandteile der runden Klammer optional sind. Das nachfolgende Beispiel enthält daher eine gültige `for`-Schleife, bei der Initialisierung, Abbruchbedingung und Iteration in eigenen Codezeilen enthalten sind. Eine innerhalb eines Anweisungsblocks einer Schleife deklarierte Variable ist nur innerhalb des Blocks gültig, nicht in der ganzen Unteroutine.

Normale For-Schleife	For-Schleife ohne Inhalt in den runden Klammern
<pre>for (int a = 0; a <= 10; a++) { ... }</pre>	<pre>int b = 0; for (; ;) { b++; if (b > 10) break; ... }</pre>

Tabelle 6.11 Beispiele für For-Schleifen in C#

Iterator-Implementierung mit Yield (Yield Continuations)

Iteratoren sind ein .NET-Entwurfsmuster zur Erzeugung aufzählbarer Mengen, die mit `for...each` durchlaufen werden können. Das in C# 2005 eingeführte Schlüsselwort `yield` vereinfacht die Iterator-Implementierung erheblich. `Yield` liefert ähnlich wie `return` einen Wert an den Aufrufer zurück. Anders als beim Einsatz von `return` beginnt die CLR beim nächsten Aufruf der Methode nicht am Anfang der Routine, sondern setzt die Bearbeitung nach dem `yield` fort. Das nächste Listing zeigt eine einfache Iterator-Klasse, die die deutschen Bundeskanzler aufzählt. Sinn macht ein solcher Iterator, wenn zwischen den Schritten irgendeine Art von Verarbeitung stattfindet, wenn z. B. die Daten aus einem Datenspeicher geholt oder dynamisch berechnet werden.

```
public class KanzlerListe : IEnumerable
{
    public IEnumerator GetEnumerator()
    {
        // Logik !!!
        yield return "Adenauer";
        // Logik !!!
        yield return "Erhard";
        // Logik !!!
        yield return "Kiesinger ";
        // Logik !!!
        yield return "Brandt";
        // Logik !!!
        yield return "Schmidt";
        // Logik !!!
        yield return "Kohl";
        // Logik !!!
        yield return "Schröder";
        // Logik !!!
        yield return "Merkel";
        // Ende
        yield break;
    }
}

class Iteratoren
{
    public static void run()
    {
        KanzlerListe k2 = new KanzlerListe();
        foreach (string s in k2)
        {
            Console.WriteLine(s);
        }
    }
}
```

Listing 6.56 Iterator-Implementierung und -Nutzung in C# 2005

Praxisbeispiel für Yield

Das vorstehende Beispiel ist nur ein Lernbeispiel. Eine Schleife über eine Menge von Zeichenketten hätte man auch einfacher realisieren können. Ein echtes Praxisbeispiel für den Einsatz von `Yield` finden Sie in der World Wide Wings-Anwendung in Form der Klasse `FlugMengePaging`. Diese Klasse implementiert `IEnumerable<Flug>`, um die in der Datenbank vorhandenen Flüge seitenweise aus der Datenbank auszulesen,

wobei die Seitengröße definierbar ist. Der Client soll von dem Paging nichts mitbekommen, wenn er nicht will: Der Client kann mit einer ganz normalen For/Each-Schleife über die Datensätze iterieren. Optional kann der Client das Ereignis `SeitenWechsel`, das die Klasse `FlugMengePaging` auslöst, abonnieren und damit über den Seitenwechsel informiert werden.

```

C:\file:///H:/WWW/ConsoleUI_CS/bin/Debug/ConsoleUI_CS.EXE
Flug 152 von Hamburg nach Köln/Bonn am 13.01.2006 01:53:03.
Flug 153 von Hamburg nach Rom am 13.01.2006 20:34:03.
Flug 154 von Hamburg nach London am 12.01.2006 23:28:03.
Flug 155 von Hamburg nach Paris am 13.01.2006 18:09:03.
#### Datensseite 11 von 47 mit 5 von insgesamt 235 Elementen:
Flug 156 von Hamburg nach Mailand am 14.01.2006 12:50:03.
Flug 157 von Hamburg nach Prag am 15.01.2006 07:32:03.
Flug 158 von Hamburg nach Moskau am 12.01.2006 14:53:03.
Flug 159 von Hamburg nach New York am 12.01.2006 14:53:03.
Flug 160 von Hamburg nach Seattle am 13.01.2006 09:34:03.
#### Datensseite 12 von 47 mit 5 von insgesamt 235 Elementen:
Flug 161 von Hamburg nach Essen/Mülheim am 14.01.2006 04:15:03.
Flug 162 von Hamburg nach Kapstadt am 14.01.2006 22:57:03.
Flug 163 von Hamburg nach Madagid am 13.01.2006 00:59:03.
Flug 164 von Köln/Bonn nach Berlin am 14.01.2006 14:22:03.
Flug 165 von Köln/Bonn nach Frankfurt am 14.01.2006 14:22:03.
#### Datensseite 13 von 47 mit 5 von insgesamt 235 Elementen:
Flug 166 von Köln/Bonn nach München am 15.01.2006 09:03:03.
Flug 167 von Köln/Bonn nach Hamburg am 12.01.2006 16:24:03.
Flug 169 von Köln/Bonn nach Rom am 14.01.2006 05:47:03.
Flug 170 von Köln/Bonn nach London am 15.01.2006 00:28:03.
Flug 171 von Köln/Bonn nach Paris am 12.01.2006 07:49:03.
#### Datensseite 14 von 47 mit 5 von insgesamt 235 Elementen:
Flug 172 von Köln/Bonn nach Mailand am 12.01.2006 07:49:03.
Flug 173 von Köln/Bonn nach Prag am 13.01.2006 02:31:03.

```

Abbildung 6.8 Nutzung der Klasse `FlugMengePaging`

Das folgende Listing zeigt die Implementierung der Klasse `FlugMengePaging`, die zwei Generische Klassen der .NET-Klassenbibliothek verwendet:

- Zum einen die generische Variante von `IEnumerable`: `IEnumerable<Flug>`
- Zum anderen die generische Klasse `EventHandler<>` zur Deklaration eines Ereignisses.

```

/// <summary>
/// Klasse für Ereignisparameter beim Paging in der Geschäftslogik
/// </summary>
public class PagingInfo : System.EventArgs
{
    public long AnzahlObjekteGesamt;
    public long SeitenGroesse;
    public long AnzahlSeiten;
    public long AktuelleSeite;
    public long AnzahlObjekteInAktuellerSeite;

    public PagingInfo(long AnzahlObjekteGesamt, long AnzahlSeiten, long SeitenGroesse, long
AktuelleSeite, long AnzahlInAktuellerSeite)
    {
        this.AnzahlObjekteGesamt = AnzahlObjekteGesamt;
        this.AnzahlSeiten = AnzahlSeiten;
        this.SeitenGroesse = SeitenGroesse;
        this.AnzahlObjekteInAktuellerSeite = AnzahlInAktuellerSeite;
        this.AktuelleSeite = AktuelleSeite;
    }
}

/// <summary>
/// FlugMenge ist die typisierte Menge von Flug-Objekten, die mithilfe der Klasse
/// <see cref="System.Collections.Generic.List"/> implementiert ist. Diese Variante holt immer
/// nur eine definierbare Menge (Attribut SeitenGroesse) aus der Datenbank.

```



```

/// </summary>
public class FlugMengePaging : IEnumerable<Flug>
{
    private int _SeitenGroesse = 10;
    /// <summary>
    /// Maximale Anzahl von Objekten, die in einer Datenseite abgeholt werden
    /// </summary>
    public int SeitenGroesse
    {
        get { return _SeitenGroesse; }
        set { _SeitenGroesse = value; }
    }
    // Ereignis beim Wechsel der Datenseite
    public event EventHandler<PagingInfo> SeitenWechsel;
    public FlugMengePaging(int SeitenGroesse)
    {
        this.SeitenGroesse = SeitenGroesse;
    }
    #region IEnumerable<Flug> Members
    public IEnumerator<Flug> GetEnumerator()
    {
        int Anzahl = new FlugBLManager().Count();
        int Seiten = Anzahl / SeitenGroesse;

        for (int i = 0; i < Seiten; i++)
        {
            // Nächste Datenseite abholen
            FlugMenge ff = FlugBLManager.HoleAlle(SeitenGroesse, i * SeitenGroesse + 1);
            // Ereignis auslösen
            if (SeitenWechsel != null) SeitenWechsel(this, new PagingInfo(Anzahl, Seiten,
                                                                    SeitenGroesse, i + 1, ff.Count));
            // Elemente der aktuellen Seite in einer Schleife zurückgeben
            foreach (Flug f in ff)
            {
                yield return f;
            }
            yield break;
        }
    }
}

```

Listing 6.57 Praxisbeispiel zum Einsatz von Yield, Ereignissen und Generics

Verzweigungen

Bei den Verzweigungen werden einfache Verzweigungen und Mehrfachverzweigungen unterstützt.

Verzweigungen in VB

Für die Verzweigung im Programmcode unterstützt Visual Basic die bereits aus VB6 bekannten Konstrukte:

- If...Then...Else...End If
- Select Case...Case...Case Else...End Select

Eine innerhalb eines Anweisungsblocks einer Bedingung deklarierte Variable ist nur innerhalb des Blocks gültig, nicht in der ganzen Unterroutine.

Verzweigungen in C#

Für die Verzweigung im Programmcode unterstützt C# 2005 die gleichen Konstrukte wie VB, jedoch mit etwas anderer Syntax:

- `if (Bedingung) {...} else {...}`
- `switch (Bedingung) { case Wert:... default:... }`

Bei der `switch`-Anweisung sind im Vergleich zu der `Select`-Anweisung in VB folgende Punkte zu beachten:

- Jeder Fall muss mit einer `break`-Anweisung abgeschlossen werden
- Anders als in VB kann man bei C# keine Wertebereiche nach `case` angeben

```
switch (a)
{
    case 1: Console.WriteLine("Fall 1"); break;
    case 2: Console.WriteLine("Fall 2"); break;
    case 3: Console.WriteLine("Fall 3"); break;
    case 4: Console.WriteLine("Fall 4"); break;
    case 5: Console.WriteLine("Fall 5"); break;
    case 6:
    case 7: Console.WriteLine("Sonderfall!"); break;
    default: Console.WriteLine("Nicht unterstützter Fall!");break;
}
```

Listing 6.58 Verwendung der `switch`-Anwendung in C#

Eine innerhalb eines Anweisungsblocks einer Bedingung deklarierte Variable ist nur innerhalb des Blocks gültig, nicht in der ganzen Unteroutine.

Funktionszeiger (Delegates)

Delegaten (engl. *Delegates*) sind typsichere Zeiger auf Funktionen. Durch Delegaten kann der aufzurufende Code variabel gehalten werden. Sie kommen insbesondere zum Einsatz für die Ereignisbehandlung und für asynchrone Methodenaufrufe. Ein Delegat kann auf mehrere Funktionen zeigen (*Multicast Delegate*). Beim Aufruf des Delegaten werden alle an den Delegaten gebundenen Funktionen aufgerufen.

Jeder deklarierte Delegat erhält automatisch die Methoden `Invoke()`, `BeginInvoke()` und `EndInvoke()`.

Funktionszeiger in VB

VB unterstützt .NET-Funktionszeiger durch das Schlüsselwort `Delegate`.

Funktion	Syntax
Methodendeklaration	<code>Public Function HoleWert(ByVal Parameter As Long) As String</code>
Deklaration eines Funktionszeigertyps	<code>Public Delegate Function HoleWertDelegate(ByVal Parameter As Long) As String</code>
Erstellung eines Zeigers auf die Funktion	<code>Dim del As New HoleWertDelegate(AddressOf Me.HoleWert)</code>

Tabelle 6.12 Beispiele für den Einsatz von Delegaten

Ein gutes Anwendungsbeispiel für Delegates ist der asynchrone Methodenaufruf. Hierfür sind neben dem Funktionszeiger auch eine Rückrufroutine (Callback-Routine) und ein AsyncCallback-Objekt notwendig, das auf die Rückrufroutine verweist und beim Aufruf von `BeginInvoke()` übergeben werden muss. In der Rückrufroutine kann über `EndInvoke()` das Ergebnis abgerufen werden.

```
Public Class DelegateBeispiel
    ' == Funktion
    Public Function HoleWert(ByVal Parameter As Long) As String
        Console.WriteLine("Methodenaufruf...")
        Return "Wert " & Parameter
    End Function
    ' == Definition eines Funktionszeigertyps
    Public Delegate Function HoleWertDelegate(ByVal Parameter As Long) As String
    ' == Hauptprogramm
    Public Sub Test()
        ' --- Synchroner Aufruf
        Console.WriteLine("Asynchroner Aufruf...")
        Console.WriteLine("Ergebnis: " & HoleWert(2))
        ' --- Asynchroner Aufruf
        Console.WriteLine("Asynchroner Aufruf...")
        Dim del As New HoleWertDelegate(AddressOf Me.HoleWert)
        Dim Callback As New AsyncCallback(AddressOf Fertig)
        del.BeginInvoke(123, Callback, del)
        ' --- Warten
        For a As Integer = 1 To 10
            Console.WriteLine(a)
            System.Threading.Thread.Sleep(100)
        Next
        Console.ReadLine()
    End Sub
    ' == Callback-Handler
    Public Sub Fertig(ByVal CallbackResult As IAsyncResult)
        Console.WriteLine("Aufruf fertig...")
        Dim del As HoleWertDelegate = CType(CallbackResult.AsyncState, HoleWertDelegate)
        Dim Ergebnis As String = del.EndInvoke(CallbackResult)
        Console.WriteLine("Ergebnis: " & Ergebnis)
    End Sub
End Class
```

Listing 6.59 Asynchroner Methodenaufruf unter Einsatz von Delegates

Funktionszeiger in C#

Die Tabelle und das nachfolgende Listing zeigen den Einsatz eines Funktionszeigers zum asynchronen Aufruf einer Methode in C#.

Funktion	Syntax
Methodendeklaration	<code>public string HoleWert(long Parameter)</code>
Deklaration eines Funktionszeigertyps	<code>public delegate string HoleWertDelegate(long Parameter);</code>
Erstellung eines Zeigers auf die Funktion	<code>HoleWertDelegate del = new HoleWertDelegate(this.HoleWert);</code>

Tabelle 6.13 Beispiele für den Einsatz von Delegates

```

public class DelegateBeispiel
{
    // === Funktion
    public string HoleWert(long Parameter)
    {
        Console.WriteLine("Methodenaufruf...");
        return "Wert " + Parameter;
    }
    // === Definition eines Funktionszeigertyps
    public delegate string HoleWertDelegate(long Parameter);
    // === Hauptprogramm
    public void Test()
    {
        // --- Synchroner Aufruf
        Console.WriteLine("Asynchroner Aufruf...");
        Console.WriteLine("Ergebnis: " + HoleWert(2));
        // --- Asynchroner Aufruf
        Console.WriteLine("Asynchroner Aufruf...");
        HoleWertDelegate del = new HoleWertDelegate(this.HoleWert);
        AsyncCallback Callback = new AsyncCallback(Fertig);
        del.BeginInvoke(123, Callback, del);
        // --- Warten
        for (int a = 1; a <= 10; a++)
        {
            Console.WriteLine("");
            System.Threading.Thread.Sleep(100);
        }
        Console.ReadLine();
    }
    // === Callback-Handler
    public void Fertig(IAsyncResult CallbackResult)
    {
        Console.WriteLine("Aufruf fertig...");
        HoleWertDelegate del = (HoleWertDelegate)CallbackResult.AsyncState;
        string Ergebnis = del.EndInvoke(CallbackResult);
        Console.WriteLine("Ergebnis: " + Ergebnis);
    }
}

```

Listing 6.60 Asynchroner Methodenaufruf unter Einsatz von Delegates

Funktionale Programmierung mit Lambda-Ausdrücken

Ein Lambda-Ausdruck ist eine stark verkürzte Schreibweise für eine Methode, die einen einzelnen Ausdruck auswertet. Technisch gesehen handelt es sich bei den Lambda-Ausdrücken um eine verkürzte Schreibweise von Funktionszeigern (Delegates) und zugleich um anonyme Delegates, da kein expliziter Name für die Delegate-Klasse vergeben wird. Dies erledigt wie bei anonymen Typen der Compiler.

Lambda-Ausdrücke gibt es in zwei Formen: Einzeilige Lambda-Ausdrücke und mehrzeilige Lambda-Ausdrücke.

HINWEIS Komplexe Lambda-Ausdrücke lassen sich durch so genannte Ausdrucksbäume (Expression Trees) auch anders darstellen. Dies würde jedoch den Fokus dieses Buchs völlig überschreiten, zumal sich mit diesem Thema nur wenige Entwickler befassen werden. Erläuterungen zu Ausdrucksbäumen finden Sie unter [MSDN19].

Lambda-Ausdrücke in VB

Zur Deklaration der Variablen verwendet man die Klasse `Func` aus dem Namensraum `System.Linq` in der `System.Core.dll`. Zur Definition des Rumpfs wird in VB 9.0 das Schlüsselwort `Function` »wiederverwendet«. Der Rumpf wird direkt dahinter ohne Blockbegrenzung geschrieben.

```
' Lambda-Ausdrücke deklarieren
Dim f1 As Func(Of Integer, Integer) = Function(x) x + 1
Dim f2 As Func(Of String, String) = Function(s) s.ToUpper()
Dim f3 As Func(Of String, Integer) = Function(s) s.Length
Dim f4 As Func(Of String, Integer, String) = Function(s, i) s.Substring(0, i)
' Lambda-Ausdrücke verwenden
Console.WriteLine(f1(10)) ' ergibt 11
Console.WriteLine(f2("World Wide Wings")) ' ergibt "HALLO"
Console.WriteLine(f3("World Wide Wings")) ' ergibt WORLD WIDE WINGS
Console.WriteLine(f4("World Wide Wings", 10)) ' ergibt "World Wide"
```

Listing 6.61 Beispiele für Lambda-Ausdrücke in VB (ab VB 2008)

Während in VB 2008 Lambda-Ausdrücke lediglich aus einer einzigen Zeile bestehen durften, können diese nun in VB 2010 auch aus mehreren Zeilen bestehen. Wie aus Listing 6.62 ersichtlich, werden diese mit *End Function* abgeschlossen. Der Rückgabewert wird dabei unter Verwendung von *Return* zurückgeliefert. Wenn es keinen Rückgabewert gibt, kommt *Sub... End Sub* zum Einsatz.

```
' Deklaration Lambda-Ausdruck mit Rückgabewert
Dim ZahlenKleiner10 = Function(x As Integer)
    Console.WriteLine("Prüfe Zahl: " & x)
    Return x < 10
End Function

' Datenmenge
Dim Zahlen() As Integer = {1, 30, 5, 10, 15, 20, 3, 9}
' Verwendung Lambda-Ausdruck
Dim Ergebnis = Array.FindAll(Zahlen, ZahlenKleiner10)
' Ausgabe
For Each Zahl As Integer In Ergebnis
    Console.WriteLine(Zahl)
Next
```

Listing 6.62 Beispiel für einen mehrzeiligen Lambda-Ausdruck mit Rückgabewert in VB 2010

```
' Deklaration Lambda-Ausdruck ohne Rückgabewert
Dim Ausgabe = Sub(n)
    Trace.WriteLine(n)
    Console.WriteLine(n)
End Sub

' Datenmenge
Dim ZahlenReihe() As Integer = {1, 30, 5, 10, 15, 20, 3, 9}
' Verwendung Lambda-Ausdruck
Array.ForEach(ZahlenReihe, Ausgabe)
```

Listing 6.63 Beispiel für einen mehrzeiligen Lambda-Ausdruck ohne Rückgabewert in VB 2010

Lambda-Ausdrücke in C#

In C# kommt ebenfalls die Klasse `Func` zum Einsatz. Zu beachten ist, dass `Func` kein Schlüsselwort der Sprache C#, sondern eine generische Klasse ist. Daher muss der Name groß geschrieben werden. Der Rumpf wird durch den Operator `=>` knapp gehalten. `=>` könnte man hier lesen als »wird abgebildet auf«.

HINWEIS Lambda-Ausdrücke, die einen Typ auf einem anderen Typ abbilden (also Beispiele 2 bis 4 in dem folgenden Listing), nennt man eine Projektion.

```
// Lambda-Ausdrücke deklarieren
Func<int, int> f1 = x => x + 1;
Func<string, string> f2 = s => s.ToUpper();
Func<string, int> f3 = s => s.Length;
Func<string, int, string> f4 = (s, i) => s.Substring(0, i);

// Lambda-Ausdrücke verwenden
Console.WriteLine(f1(10)); // ergibt 11
Console.WriteLine(f2("World Wide Wings")); // ergibt WORLD WIDE WINGS
Console.WriteLine(f3("World Wide Wings")); // ergibt 16
Console.WriteLine(f4("World Wide Wings", 10)); // Ergibt "World Wide"
```

Listing 6.64 Beispiele für Lambda-Ausdrücke in C# (ab C# 2008)

```
Predicate<int> ZahlenKleiner10 = x =>
{
    Console.WriteLine("Prüfe Zahl: " + x);
    return x < 10;
};

// Datenmenge
int[] Zahlen = {1,30, 5, 10, 15, 20, 3, 9};
// Verwendung Lambda-Ausdruck
var Ergebnis = Array.FindAll(Zahlen, ZahlenKleiner10);
// Ausgabe
foreach (object Zahl in Ergebnis)
{
    Console.WriteLine(Zahl);
}
```

Listing 6.65 Beispiel für einen mehrzeiligen Lambda-Ausdruck mit Rückgabewert in C# (ab C# 2008)

```
// Deklaration Lambda-Ausdruck ohne Rückgabewert
Action<int> Ausgabe = x =>
{
    Trace.WriteLine(x);
    Console.WriteLine(x);
};
// Datenmenge
int[] ZahlenReihe = {1,30, 5, 10, 15, 20, 3, 9};
// Verwendung Lambda-Ausdruck
Array.ForEach(ZahlenReihe, Ausgabe);
```

Listing 6.66 Beispiel für einen mehrzeiligen Lambda-Ausdruck ohne Rückgabewert in C# (ab C# 2008)

Prädikate

Ein Einsatzbeispiel für Lambda-Ausdrücke ist der Einsatz als *Prädikat*. Ein *Prädikat* ist ein Funktionszeiger (Delegat) auf eine Methode, die true oder false liefert. Prädikate werden zur Auswahl von Elementen in Listen verwendet. Die Objektmengenklassen in der FCL (siehe auch Kapitel 9 zur .NET Klassenbibliothek 4.0) stellen Methoden bereit, die Prädikate erwarten. Das folgende Listing zeigt drei verschiedene Schreibweisen, um alle Vorstandsmitglieder aus einer Liste zu filtern, die eine bestimmte Bedingung erfüllen; die letzte Schreibweise mit Lambda-Ausdrücken ist die kürzeste und eleganteste. Im Kapitel 10 zu Language Integrated Query (LINQ) werden Sie dann noch eine elegantere (wenn auch nicht kürzere) Schreibweise sehen.

```
{
...
// Prädikate klassische Schreibweise
List<Vorstandsmitglied> JungeVorstandsmitglieder1 = Vorstandsmitglieder.FindAll(AuswahlJunge);
Console.WriteLine("Junge Vorstandsmitglieder: " + JungeVorstandsmitglieder1.Count);

// Prädikate mit anonymen Methoden
List<Vorstandsmitglied> JungeVorstandsmitglieder2 = Vorstandsmitglieder.FindAll(delegate(Vorstandsmitglied v) { return v.Alter < 40; });
Console.WriteLine("Junge Vorstandsmitglieder: " + JungeVorstandsmitglieder2.Count);

// Prädikate mit Lambda-Ausdruck
List<Vorstandsmitglied> JungeVorstandsmitglieder3 = Vorstandsmitglieder.FindAll(v => v.Alter < 40);
Console.WriteLine("Junge Vorstandsmitglieder: " + JungeVorstandsmitglieder3.Count);
}

// gehört zu Prädikat klassische Schreibweise!
static public bool AuswahlJunge(Vorstandsmitglied v)
{
    return (v.Alter < 40);
}
```

Listing 6.67 Prädikate in C#

Annotationen (.NET-Attribute)

Der Entwickler selbst kann Komponenten, Klassen und Klassenmitglieder mit zusätzlichen Informationen (Metadaten) versehen, die entweder während der Kompilierung oder zur Laufzeit der Anwendung ausgewertet werden können. Details wurden bereits im Kapitel 4 »Grundkonzepte des .NET Framework 4.0« erläutert.

Annotationen in VB

Annotationen werden in Visual Basic in spitzen Klammern den Klassen bzw. Klassenmitgliedern vorangestellt. Sie müssen in der gleichen Befehlszeile stehen wie das `Class`-Schlüsselwort (bzw. `Sub`, `Function` etc.).

In dem folgenden Beispiel wird die vordefinierte Annotation `System.Obsolete` einer Methode zugewiesen. `System.Obsolete` sorgt dafür, dass der Compiler den Entwickler warnt, wenn er eine derart deklarierte Methode aufruft. Das zweite Beispiel zeichnet die Klasse `Passagier` als serialisierbar aus, d. h., ihre Instanzen können persistiert oder in einen anderen Prozess übertragen werden.

```
<System.Obsolete("Benutzen Sie bitte den überladenen Operator '+'.")>
Sub PassagierHinzufuegen(ByVal pass As de.WWWings.PassagierSystem.Passagier)
    pass.Buchen(Me)
End Sub
```

Listing 6.68 Beispiel für die Anwendung der Annotation `System.Obsolete` in VB

```
<System.Serializable()>
Public Class Passagier
    ...
End Class
```

Listing 6.69 Beispiel für die Anwendung der Annotation `System.Serializable` in VB

Annotationen in C#

Annotationen können in C# den Typen und den Typmitgliedern in eckigen Klammern vorangestellt werden.

```
[System.Serializable()]
public class Passagier : de.WWWings.Person
{...}
```

Listing 6.70 Beispiel für die Anwendung der Annotation `System.Serializable` in C#

Fehlerbehandlung

Das Erzeugen und Behandeln von Ausnahmen ist in der CLR verankert und daher für alle .NET-Sprachen gleich. *Exceptions* (Ausnahmen) sind .NET-Objekte, wobei es verschiedene Klassen von Ausnahmen geben kann, die in einer Vererbungshierarchie zueinander stehen. Basisklasse ist `System.Exception`. Jede Ausnahme stellt Informationen wie eine Fehlerbeschreibung (`Message`) und die Aufrufliste der Methoden (`StackTrace`) bereit.

Fehlerbehandlung VB

An die Stelle der unschönen Fehlerbehandlung mit `On Error Goto` in VB6 tritt die elegante Fehlerbehandlung auf Basis von Ausnahmen (*Exceptions*). Ab VB7 unterstützt die Sprache das Konstrukt `Try...Catch...Finally`, um Laufzeitfehler abzufangen. Dabei kann es mehrere `Catch`-Blöcke mit unterschiedlichen Ausnahmeklassen geben. Eine `Catch ex As Exception` fängt alle Fehler ab, weil `System.Exception` die Oberklasse aller Ausnahmen ist.


```

Try
    p2.CheckIn("NF5678")
Catch ex As PassagierNichtAufFlugGebucht
    Demo.Print("Einchecken nicht möglich, da Passagier nicht auf diesen Flug gebucht ist!")
Catch ex As Exception
    Demo.Print("Es ist ein unerwarteter Fehler aufgetreten: " & ex.Message)
End Try

```

Listing 6.71 Fehlerbehandlung

Throw *ExceptionKlasse* erzeugt eine Ausnahme. Neben den in der .NET-Klassenbibliothek vordefinierten Ausnahmen (z. B. *System.ArithmeticException*, *System.ArgumentException*, *System.FormatException*) können eigene anwendungsspezifische Ausnahmeklassen durch Ableitung von *System.ApplicationException* erzeugt werden.

```

Namespace de.WWWings.PassagierSystem
    Public Class FalscheFlugnummer
        Inherits System.ApplicationException
        Public Sub New(ByVal Beschreibung As String)
            MyBase.New(Beschreibung)
        End Sub
    End Class
    Public Class PassagierNichtAufFlugGebucht
        Inherits FalscheFlugnummer
        Public Sub New(ByVal Beschreibung As String)
            MyBase.New(Beschreibung)
        End Sub
    End Class
End Namespace

```

Listing 6.72 Definition eigener Ausnahmen

ACHTUNG Eine .NET-Klasse kann – anders als in Java – nicht deklarieren, welche Fehlertypen sie erzeugt und welche vom Nutzer abgefangen werden müssen (Konzept der *Checked Exceptions*). Der .NET-Entwickler kann Wissen über mögliche Fehlerarten nur aus der Dokumentation entnehmen.

Fehlerbehandlung in C#

C# unterstützt das Konstrukt `try...catch...finally`, um Laufzeitfehler abzufangen. Dabei kann es mehrere Catch-Blöcke mit unterschiedlichen Ausnahmeklassen geben. Ein `catch (Exception ex)` fängt alle Fehler ab, weil *System.Exception* die Oberklasse aller Ausnahmen ist.

```

try
{
    p2.CheckIn("NF5678");
}
catch (de.NETFly.PassagierSystem.PassagierNichtAufFlugGebucht ex)
{
    Demo.Print("Einchecken nicht möglich, da Passagier nicht auf diesen Flug gebucht ist!")
}
catch (Exception ex)
{
    Demo.Print("Es ist ein unerwarteter Fehler aufgetreten: " + ex.Message);
}

```

Listing 6.73 Fehlerbehandlung in C#

```
public class FalscheFlugnummer : System.ApplicationException
{
    public FalscheFlugnummer(string Beschreibung) : base(Beschreibung) { }
}
public class PassagierNichtAufFlugGebucht : FalscheFlugnummer
{
    public PassagierNichtAufFlugGebucht(string Beschreibung) : base(Beschreibung) { }
}
```

Listing 6.74 Definition eigener Ausnahmen in C#

Eingebaute Objekte und Funktionen

Im .NET Framework ist es nicht so üblich, dass Programmiersprachen eingebaute Objekte und Funktionen besitzen, da aus Gründen der Vereinheitlichung diese Aufgaben durch die allen Sprachen gemeinsame .NET-Klassenbibliothek wahrgenommen werden soll.

Eingebaute Objekte und Funktionen in VB

Traditionell wurde in Visual Basic sehr viel Funktionalität durch in die Sprache integrierte Objekte und Funktionen angeboten, weil es eine getrennte Klassenbibliothek für Visual Basic gab. Durch die Integration in das .NET Framework sollte an diese Stelle grundsätzlich die .NET-Klassenbibliothek treten. Aus Gründen der Kompatibilität hat Microsoft jedoch zahlreiche Funktionen in ihrer früheren Form belassen und lediglich anders implementiert.

Typprüfungsfunktionen wie `isNumeric()` und `isDate()`, Typkonvertierungsfunktionen wie `CInt()` und `CStr()` sowie sonstige Ein-/Ausgabe-Funktionen wie `MsgBox()` und `InputBox()` werden nun durch den Namensraum `Microsoft.VisualBasic` (*Microsoft.VisualBasic.dll*) als globale Funktionen bereitgestellt. Einzelne Funktionen wurden hingegen entfernt, beispielsweise wird aus `DoEvents` das Konstrukt `System.Windows.Forms.Application.DoEvents`.

Um eine klare Trennung zwischen Sprache und Klassenbibliothek und eine leichtere Konvertierbarkeit in andere .NET-Sprachen sicherzustellen, sollten Sie die Verwendung der Klassen aus der .NET-Klassenbibliothek präferieren, z.B. `System.String.IndexOf()` statt `InStr()`.

My-Objekte

Seit Version 2005 enthält VB eine eigene kleine unabhängige Klassenbibliothek, die sich hinter dem eingebauten Objekt `My` verbirgt und den Zugriff auf Informationen über Computer, Nutzer, Dateisystem und vieles mehr ermöglicht. Letztlich stellen die `My`-Klassen Abkürzungen sowie Vereinfachungen bestehender .NET-Klassen dar. Zwar sind die `My`-Klassen auch von anderen .NET-Sprachen aus referenzierbar; es stellt sich jedoch die Frage, warum Microsoft diese Klassen in die *Microsoft.VisualBasic.dll* (Namensraum `Microsoft.VisualBasic.MyServices`) verbannt hat.

HINWEIS

Dazu sei ein kritischer Kommentar erlaubt: Microsoft mindert sicherlich den Spott nicht, den manche Visual Basic-Entwickler auch in .NET-Zeiten von ihren mit geschweiften Klammern arbeitenden Kollegen erfahren, indem die Firma eine Klassenbibliothek schafft, die nahe legt, dass die .NET-Klassenbibliothek für VB-Entwickler an manchen Stellen zu kompliziert ist. Die `My`-Klassen sollten Teil der FCL sein!

Für einige Funktionen der My-Bibliothek (Zugriff auf Einstellungen und Ressourcen) ist eine Unterstützung auf Projektebene notwendig. Zu jedem VB-Projekt (seit Version 2005) gehört daher ein Ordner *My Project* mit einigen generierten Dateien. Diese generierten Dateien können im *Solution Explorer* von Visual Studio durch ein Symbol sichtbar gemacht werden.

Eingebaute Objekte und Funktionen in C#

Anders als in Visual Basic existieren in C# keine eingebauten Funktionen zur Typprüfung, Typumwandlung und Ausgabe. Auch die My-Klassenbibliothek ist nicht vorhanden. Grundsätzlich ist es möglich, die in Visual Basic eingebauten Funktionen und die My-Bibliothek durch Referenzierung der *Microsoft.VisualBasic.dll* auch in C# zu nutzen. Dies sollte jedoch vermieden werden, um sprachunabhängig zu bleiben. Alle Visual Basic-Funktionen und -Objekte sind allerdings auch in der .NET-Klassenbibliothek enthalten, zum Teil dort aber komplexer.

Kommentare und XML-Dokumentation

Beide Programmiersprachen unterstützen neben einfachen Kommentaren, die nicht vom Compiler beachtet werden, auch spezielle Kommentare in XML-Form. Der Compiler erzeugt dann aus diesen XML-Fragmenten zusammen mit den Klassendeklarationen eine XML-Datei, die als Eingabedatei für die Generierung von Hilfedokumenten verwendet werden kann. Außerdem zeigt Visual Studio den <Summary>-Text im Tooltip bei der Eingabe einer Methode an (leider nur bei C#).

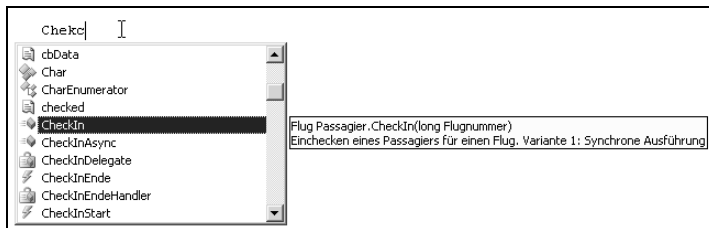


Abbildung 6.9 Anzeige eines XML-Kommentars bei der Eingabe eines Methodennamens

Kommentare in VB

Kommentarzeilen werden wie in den COM-basierten VB-Dialekten mit einem einfachen Anführungszeichen `'` eingeleitet. Seit Version 2005 unterstützt VB auch die bereits seit der ersten C#-Version beliebte Möglichkeit, eine Klasse oder ein Klassenmitglied mit Kommentaren in XML-Form zu annotieren. Die XML-Kommentare sind mit drei einfachen Hochkommata `'''` zu beginnen.

```
''' <summary>Einchecken eines Passagiers für einen Flug</summary>
''' <param name="Text">Die Methode erwartet eine Flugnummer</param>
''' <returns>Die Methode liefert als Rückgabewert das Flugobjekt, wenn das Einchecken erfolgreich war.
</returns>
Public Function CheckIn(ByVal Flugnummer As String) As Flug
    If Not Me.Fluege.ContainsKey(Flugnummer) Then
        Throw New PassagierNichtAufFlugGebucht(Me.PID & "/" & Flugnummer)
```

```
Else
    Return Me.Fluege.Item(Flugnummer)
End If
End Function
```

Listing 6.75 XML-Dokumentation in VB

HINWEIS Im Gegensatz zu C# werden die XML-Codekommentare bei der Nutzung einer VB-Klasse leider nicht in der IntelliSense-Funktion von Visual Studio angezeigt.

Kommentare in C#

C# unterstützt drei Arten von Kommentaren:

- Zeilenkommentare, bei denen jede Zeile mit einem `//` eingeleitet wird
- Blockkommentare, bei denen der Codeblock in `/* ... */` eingerahmt wird
- XML-Kommentare, bei denen jede Zeile mit `///` beginnt.

```
/// <summary>Einchecken eines Passagiers für einen Flug</summary>
/// <param name="Flugnummer">Die Methode erwartet eine Flugnummer.</param>
/// <returns>Die Methode liefert als Rückgabewert das Flugobjekt, wenn das Einchecken erfolgreich
war.</returns>
public Flug CheckIn(string Flugnummer)
...
```

Listing 6.76 Beispiel für XML-Codekommentare in C#

Zeigerprogrammierung (Unsicherer Code)

Niemand möchte *unsicheren Code* schreiben, doch die Programmiersprache C# kennt eine gleichnamige Option (`unsafe`). Innerhalb von unsicherem Code können in C# Zeiger und Zeigerarithmetik verwendet werden. Diese Operationen werden dann nicht von der Common Language Runtime verifiziert und können zu Programmabstürzen führen. Bei Visual Basic gibt es keine in die Sprachsyntax eingebaute Möglichkeit, Zeiger und Zeigerarithmetik zu nutzen. Das wäre nur über Umwege über die Klassenbibliothek möglich. Wenn Sie derartige *Low-Level-Funktionen* wirklich nutzen wollten, sollten Sie C# oder C++ / CLI verwenden.

ACHTUNG Es gibt nur wenige sinnvolle Einsatzgebiete für Zeigerarithmetik im .NET Framework. Ein solcher Fall liegt bei sehr umfangreichen Array-Operationen vor. Da die CLR bei jedem Array-Zugriff die Array-Grenzen prüft, kann durch Einsatz von Zeigerarithmetik ein erheblicher Leistungsgewinn erzielt werden – allerdings auf Kosten der Zuverlässigkeit der Anwendung.

Mit dem Schlüsselwort `unsafe` können ganze Unterrouтины markiert werden; es besteht auch die Möglichkeit, einen `unsafe`-Block innerhalb einer Unterroutine zu erzeugen. Voraussetzung für die Kompilierung einer Anwendung mit unsicherem Code ist die Verwendung der Compiler-Option `/unsafe`.

```
class Zeiger
{
    unsafe static void ZeigerTest(int* x) // x ist ein Zeiger auf ein Integer32
    {
        int* y; // y ist ein Zeiger auf ein Integer32
        int z = 10; // z ist ein Integer32
        y = &z; // y zeigt auf den Speicherplatz von z
        *x = *x * *y; // Der Platz, auf den x zeigt, soll mit dem Ergebnis des Produktes aus dem Inhalt von x
        und y gefüllt werden
        int* r; // y ist ein Zeiger auf ein Integer32
        // Achtung: Das produziert Unsinn!
        r = y + 1; // r soll nun auf den Speicherplatz zeigen, der 4 Plätze hinter y liegt
        Demo.Print(*r); // gebe den Inhalt aus, auf den r zeigt
    }
    public static void run()
    {
        int i = 5;
        unsafe
        {
            ZeigerTest(&i); // Rufe ZeigerTest mit einem Zeiger auf den Speicherplatz von i auf
        }
        Demo.Print(i);
    }
}
```

Listing 6.77 Unsicherer Code in C#

HINWEIS Die Ausführung von unsicherem Code kann explizit durch die Code Access Security (CAS) verboten werden. Die Einstellung können Sie in dem entsprechenden Berechtigungssatz vornehmen (vgl. Kapitel 4 »Grundkonzepte des .NET Framework 4.0«).

Abfrageausdrücke / Language Integrated Query (LINQ)

Abfrageausdrücke alias Language Integrated Query (LINQ) sind eine zentrale Neuerung seit .NET 3.5, die sich auch – aber nicht nur – auf die Syntax der Programmiersprachen C# und Visual Basic .NET auswirkt. Aufgrund seiner Bedeutung ist LINQ nicht hier, sondern in einem eigenen Kapitel in diesem Buch behandelt. Dabei werden Sie feststellen, dass viele damals in C# 3.0 und VB 9.0 eingeführte syntaktische Erweiterungen (nur) Hilfsmittel für LINQ sind. Dies gilt insbesondere für anonyme Typen, Typableitungen, Lambda-Ausdrücke, Objektinitialisierungen und Erweiterungsmethoden.

Vergleich: C# 2010 versus Visual Basic (.NET) 2010

C# oder Visual Basic – das ist für viele ein Glaubenskrieg. Dieses Kapitel soll abschließend einige Aspekte dazu beitragen, diesen Krieg beizulegen.

Vergleichstabelle

Eine detaillierte syntaktische Vergleichstabelle der beiden Sprachen finden Sie im Anhang dieses Buchs. Diese Tabelle können Sie auf der Leser-Website auch als PDF-Dokument herunterladen, um diese als Ausdruck neben Ihre Tastatur zu legen oder an die Pinwand zu hängen. Die folgende Tabelle zeigt nur die wichtigsten Vergleichspunkte in der Zusammenfassung.

	VB(.NET) 10.0	C# 4.0
Verwandte Sprachen	VB6, VBScript, VBA	C++, Java
Objektorientierung	++ (kann aber mit »Modulen« umgangen werden)	++
Zusätzliche Features	Vereinfachte Ereignisbindung Select mit Bereichen und Operationen (My-Namensraum) XML-Literate	Zeigerarithmetik Mehrzeilenkommentare Anonyme Methoden Yield
Case Sensitive	Nein	Ja
Sehr strenge Typisierung	Optional	vorgeschrieben
Unterstützung in Visual Studio / Qualität des Editors	++	++
Plattformunabhängigkeit	+ (nicht bei Micro Framework)	+ (nicht bei SQL Server Integration Services, nicht bei Windows Workflow 4.0-Ausdrücken)
Geschwindigkeit	++	++ (minimaler Vorteil von C#)
Ruf	0	++

Tabelle 6.14 C# versus VB

Visual Basic-Code ist tendenziell etwas länger als C#-Code, denn Blöcke werden hier nicht mit geschweiften Klammern, sondern mit Wörtern (z.B. *Class... End Class*, *For...Next*) gebildet. Viele Wörter sind in Visual Basic aber sprechender: *MustInherit* statt *abstract*, *NotInheritable* statt *sealed*, *Overridable* statt *virtual*.

Die Objektorientierung ist in beiden Sprachen gleich ausgeprägt. In Visual Basic könnte man den OO-Stil aber durch die exzessive Verwendung des Schlüsselwortes *Module* an vielen Orten umgehen. In C# gibt es aber das gleiche Konstrukt, dort heißt es *static class*. Da man zusätzlich in C# dann vor jedes Klassenmitglied *static* schreiben muss, ist die Hemmschwelle etwas höher als in Visual Basic.

C# steht in der Tradition von C++ und Java und unterscheidet daher natürlich – anders als Visual Basic – zwischen Groß- und Kleinschreibung. Da es gemäß Common Language Specification (CLS) sowieso keine Schnittstellen geben darf, in denen man zwischen zwei Bezeichnernamen nur durch Groß- und Kleinschreibung unterscheidet, ist die Groß- und Kleinschreibung in C# kein funktionaler Vorteil, sondern nur eine Frage des Geschmacks.

Der in Visual Studio mitgelieferte Editor für Visual Basic war immer schon etwas besser als der C#-Editor. Beispielsweise fügt der Visual Basic-Editor automatisch das Blockende in den Code ein, während man in C# explizit eine schließende Klammer eingeben muss (dafür braucht man Zusatzwerkzeuge wie den Resharper [<http://www.jetbrains.com/resharper/>]). Dies ist erst durch den Zusatz »Productivity Power Tools« bereinigt werden (siehe Kapitel 5 »Visual Studio 2010«).

Bei der Geschwindigkeit gibt es keine Unterschied: Beide Sprachen erzeugen die Microsoft Intermediate Language (alias Common Intermediate Language), die der Just-In-Time-Compiler ausführt. Hier und da gibt es kleine Unterschiede in dem erzeugten Zwischencode, die aber zu vernachlässigen sind.

Ein echter funktionaler Unterschied ist die Unterstützung für Zeigerarithmetik im Stil von C++, die es in C# gibt, nicht aber in Visual Basic. Die Tatsache, dass der Entwickler aber vor jede Verwendung von Zeigerarithmetik das Schlüsselwort `unsafe` verwenden und zudem den Compiler in einen »unsafe«-Modus schalten muss, sagt schon alles über die Bedeutung dieser Funktion aus.

Der Rest der Unterschiede sind rein syntaktische Zuckerstückchen, d.h. vereinfachte Ausdrucksformen für Funktionen, die auch anders zu erreichen sind.

In .NET 4.0 haben beide Sprachen nochmals einen großen Schritt aufeinander zu gemacht. Visual Basic hatte zuvor klare Vorteile beim dynamischen Programmieren/späten Binden. C# hatte die prägnanten Automatischen Properties, die man in VB vermisste. Beide Unterschiede sind nun nicht mehr vorhanden.

HINWEIS Interessant ist die Aussage von Mads Torgersen, Produktmanager für C#, im Dokument »New features in C# 4.0«, dass C# und Visual Basic in Zukunft hinsichtlich ihrer Funktionalität noch mehr im Gleichschritt gehen sollen als bisher. »Die Sprachen sollen sich in Stil und Gefühl unterscheiden, nicht in ihrem Funktionsumfang« [MT01].

Marktindikatoren

Zuverlässige Daten darüber, ob mehr C# oder Visual Basic (.NET) eingesetzt wird, gibt es nicht. Aber es gibt Indikatoren, die für C# sprechen. Im Gulpometer [GU01], einer Statistik über die Nachfrage nach Freiberuflern, liegt C# vor Basic und das, obwohl »Basic« hier auch die .NET-Varianten umfasst. Auf der Fachkonferenz BASTA fragt der Autor regelmäßig in seinen Keynotes, wer womit programmiert. Von rund 600 Leuten meldeten sich nur rund 30 bei Visual Basic. Dabei steht »BASTA« eigentlich für »Basic Tage«. Gut möglich aber, dass sich hier viele VB-Entwickler nicht getraut haben, angesichts der zu erwartenden Übermacht von C#.

Im TIOBE INDEX liegt Visual Basic noch knapp vor C#. Der TIOBE INDEX versucht die Beliebtheit der Programmiersprachen aufgrund der Erwähnung der Begriffe auf der Website zu messen. Dabei werden verschiedene Suchmaschinen ausgewertet. Das Ergebnis sagt weder etwas über die beste Sprache noch über die Anzahl der Zeilen Programmcode aus, die in der Sprache geschrieben wurden, sondern nur darüber, über welche Sprachen am meisten geredet wird.

Position Oct 2010	Position Oct 2009	Delta in Position	Programming Language	Ratings Oct 2010	Delta Oct 2009	Status
1	1	=	Java	18.166%	-0.48%	A
2	2	=	C	17.177%	+0.33%	A
3	4	↑	C++	9.802%	-0.08%	A
4	3	↓	PHP	8.323%	-2.03%	A
5	5	=	(Visual) Basic	5.650%	-3.04%	A
6	6	=	C#	4.963%	+0.55%	A
7	7	=	Python	4.860%	+0.96%	A
8	12	↑↑↑↑	Objective-C	3.706%	+2.54%	A
9	8	↓	Perl	2.310%	-1.45%	A
10	10	=	Ruby	1.941%	-0.51%	A
11	9	↓↓	JavaScript	1.659%	-1.37%	A
12	11	↓	Delphi	1.558%	-0.58%	A
13	17	↑↑↑↑	Lisp	1.084%	+0.48%	A-
14	24	↑↑↑↑↑↑↑↑	Transact-SQL	0.820%	+0.42%	A-
15	15	=	Pascal	0.771%	+0.10%	A-
16	18	↑↑	RPG (OS/400)	0.708%	+0.12%	A-
17	29	↑↑↑↑↑↑↑↑	Ada	0.704%	+0.40%	A-
18	14	↓↓↓	SAS	0.664%	-0.14%	B
19	19	=	MATLAB	0.627%	+0.05%	B
20	-	↑↑↑↑↑↑↑↑	Go	0.626%	+0.63%	B

Abbildung 6.10 TIOBE INDEX Stand
Oktober 2010 (Quelle: [TIOBE01])

Die Entwicklung von 2002 bis 2010 zeigt im TIOBE INDEX deutliche Tendenzen:

- Weniger Java
- Weniger Visual Basic
- Weniger C++
- Mehr C#
- Mehr Ruby
- Mehr Objective-C (durch die iPhone & Co-Welle)
- Mehr Python

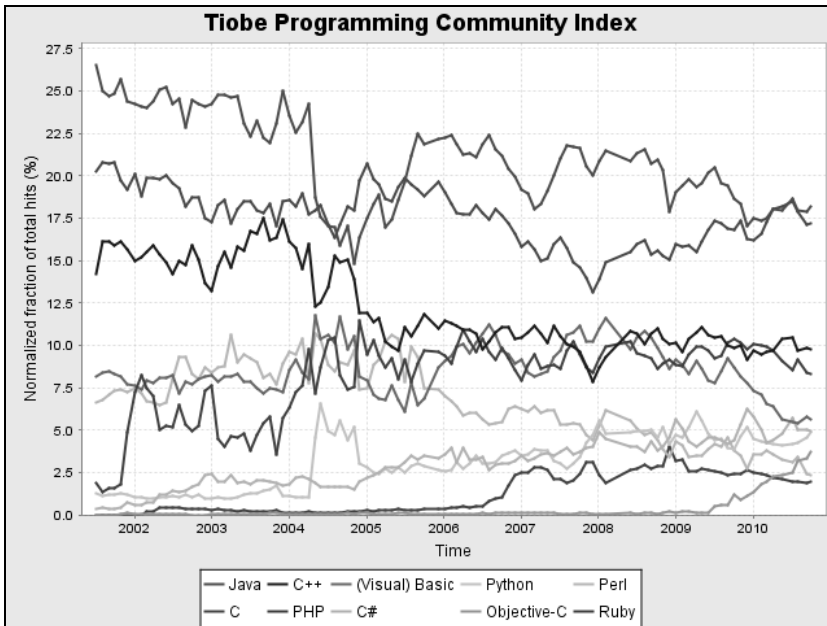


Abbildung 6.11 Veränderungen im TIOBE INDEX zwischen 2002 und 2010 (Quelle: [TIOBE01])

Honorare

In der Vergangenheit gab es ein paar Statistiken, die besagten, dass ein C#-Entwickler deutlich mehr bekommt (nicht: verdient) als ein Visual Basic-Entwickler. Das aktuelle Gulpometer besagt aber, dass C# mit 64 Euro/h für einen Freiberufler nur noch einen Vorteil von einem Euro gegenüber Visual Basic hat.

Die Meinung des Autors

C# oder Visual Basic? Es gibt nur eine Antwort: Es ist egal. Die Firma des Autors (www.IT-Visions.de) arbeitet zu 80% mit C# und zu 20% mit Visual Basic, meist getrieben durch den Wunsch der Kunden, die die Software nachher weiterentwickeln wollen.

Wenn Sie bisher mit Visual Basic 6.0 gearbeitet haben, dann fällt es vielen Entwicklern vom Kopf her leichter, jetzt mit Visual Basic .NET zu arbeiten. Bei C++ und Java liegt C# näher.

Der Autor sagt sowohl Bewerbern bei uns als auch Kunden, die er berät, immer nur eins: Wenn Sie Entwickler einstellen ist es nicht wichtig, ob sie C# oder Visual Basic .NET können. Nur sollte man niemals (!) jemanden einstellen, der darauf beharrt, nur eine der beiden Sprachen programmieren zu können oder zu wollen. Diese Form der »Beharrlichkeit« ist nämlich ein eindeutiges Zeichen von Inkompetenz.

