Visual C# 2008

Windows-Programmierung mit dem .NET Framework 3.5

JÜRGEN BAYER

KOMPENDIUM

Einführung | Arbeitsbuch | Nachschlagewerk

Teil 1
Grundlagen

Seite	Kapitel	Nr.
35	Einführung	1
87	Einführung in die Arbeit mit Visual Studio 2008	2
141	Die Sprache C#	3
231	Grundlegende OOP	4
281	Weiterführende OOP	5
341	OOP-Specials	6
373	Arrays und Auflistungen	7
441	Grundlegende Programmiertechniken	8
543	Fehler debuggen, testen und protokollieren	9
599	Arbeiten mit Dateien, Ordnern und Streams	10
649	LINQ	11

3 Die Sprache C#

Nachdem Sie in Kapitel 2 erfahren haben, wie Sie einfache C#-Programme mit Visual Studio entwickeln und kompilieren, beschreibe ich in diesem Kapitel die Grundlagen der Sprache C#. Ich zeige, wie Sie Anweisungen schreiben, mit Typen umgehen und mit Schleifen und Verzweigungen arbeiten. Um die Beispiele möglichst einfach zu halten, verwende ich dazu einfache Konsolenanwendungen.

Methoden werden übrigens noch nicht in diesem Kapitel, sondern erst bei der objektorientierten Programmierung in Kapitel 4 beschrieben.

Die Stichworte dieses Kapitels sind:

- Grundlagen von Windows- und Konsolenanwendungen
- Grundlagen zu (Daten)Typen
- Wert- und Referenztypen
- Nullables: Werttypen ohne Wert
- Grundlagen generischer Typen
- Die Standardtypen
- Über- und Unterläufe und deren Behandlung
- Typumwandlungen und Konvertierungen
- Variablen und Konstanten
- Ausdrücke und Operatoren
- Verzweigungen
- Schleifen
- Präprozessor-Direktiven und bedingte Kompilierung

Die C#-3.0-Neuigkeiten sind:

- Objektinitialisierer (Seite 155)
- Implizit typisierte lokale Variablen (Seite 198)

Neu im .NET Framework 3.5:

- Die Struktur `DateTimeOffset` (Seite 181)

3.1 Die Grundlage einer C#-(Windows-)Anwendung

Konsolen- und `Windows.Forms`-Windows-Anwendungen bestehen mindestens aus einer Klasse, die in einer Textdatei mit der Endung *.cs* gespeichert ist. Je nach Art der Anwendung unterscheidet sich der Aufbau dieser Klasse. WPF- und Webanwendungen sind prinzipiell anders aufgebaut. WPF-Anwendungen beschreibe ich ab Kapitel 12.

3.1.1 Konsolenanwendungen

Eine Konsolenanwendung ist die einfachste Anwendung in C#. Eine solche Anwendung besteht mindestens aus einer Datei, in der eine Klasse deklariert ist.

Anwendungen benötigen immer einen Einsprungpunkt (der Punkt der Anwendung, mit der diese startet). Solch einen Einsprungpunkt können Sie in einer Konsolen- oder `Windows.Forms`-Anwendung definieren, indem Sie einer Klasse eine Methode `Main` hinzufügen, die einen festgelegten Aufbau besitzen muss:

Listing 3.1: Eine einfache Konsolenanwendung

```
using System;

namespace Kompendium.Samples.Applications
{
   class Program
   {
      static void Main(string[] args)
      {
         Console.WriteLine("Hello World");
      }
   }
}
```

Die erste Anweisung (`using System;`) bindet den Inhalt des `System`-Namensraums so ein, dass bei der Verwendung der darin enthaltenen Typen und Namensräume »System.« nicht immer wieder mit angegeben werden muss.

namespace erzeugt einen Namensraum

Die `namespace`-Anweisung erzeugt einen neuen Namensraum. Alle in den geschweiften Klammern enthaltenen Typen gehören diesem Namensraum an. Diese Anweisung ist nicht unbedingt notwendig. Das Namensraum-Konzept ist aber sehr sinnvoll, weswegen Sie Ihre eigenen Anwendungen auch in Namensräumen organisieren sollten. Das Beispiel erzeugt einen übergeordneten Namensraum `Kompendium`, einen untergeordneten Namensraum `Samples` und einen weiter untergeordneten Namensraum `Applications`.

class erzeugt eine Klasse

Die `class`-Anweisung erzeugt eine neue Klasse (die für die Start-Klasse üblicherweise `Program` heißt). Diese Klasse enthält die `Main`-Methode. Das Schlüsselwort `static` bewirkt, dass diese Methode statisch ist, d. h. ohne Instanz der Klasse aufgerufen werden kann. `Main` muss statisch sein, da die CLR bei der Ausführung von `Windows.Forms`- und Konsolenanwendungen keine Instanz der Startklasse der Anwendung erzeugt.

Das in den Klammern angegebene `string[] args` deklariert einen Parameter der `Main`-Methode. An diesem Parameter werden eventuell beim Aufruf der Anwendung angegebene Befehlszeilenargumente übergeben. Befehlszeilenargumente werden in Kapitel 8 behandelt. Das `args`-Argument muss nicht unbedingt angegeben sein. Die `Main`-Methode kann also auch so aussehen:

```
static void Main()
{
   Console.WriteLine("Hello World")
}
```

Wie Sie bereits in Kapitel 2 erfahren haben, können Sie in einer Konsolenanwendung über die `Write`- oder die `WriteLine`-Methode der `Console`-Klasse einen Text an der Konsole ausgeben. Das Beispiel nutzt die `WriteLine`-Methode um den Text »Hello World« auszugeben. Der Unterschied zwischen `WriteLine` und `Write` ist, dass `WriteLine` hinter dem Text noch einen Zeilenvorschub ausgibt.

Zusätzlich können Sie über die ReadLine-Methode auf Eingaben warten und diese auswerten. ReadLine wartet, bis der Anwender die [↵]-Taste betätigt hat, und gibt das, was der Anwender bis dahin eingegeben hat, zurück. Kapitel 2 enthält dazu ein Beispiel.

3.1.2 Windows.Forms-Windows-Anwendungen

Windows.Forms-Windows-Anwendungen sind grundlegend ähnlich aufgebaut wie Konsolenanwendungen. Der Unterschied ist, dass eine solche Anwendung (normalerweise) zumindest aus zwei Quellcodedateien besteht. Eine Datei verwaltet die Klasse, die die Main-Methode enthält, mit der das Programm startet, die andere verwaltet das Formular, das als erstes Formular geöffnet wird, wenn die Anwendung startet.

Der Aufbau einer Formular-Klasse ist relativ komplex. Die Start-Klasse, die üblicherweise Program heißt, sieht aber fast so aus wie die Start-Klasse einer Konsolenanwendung. Im Unterschied zu einer solchen Anwendung wird hier lediglich eine Instanz des Start-Formulars erzeugt und der (statischen) Run-Methode der Application-Klasse übergeben. Die Anweisungen, die davor stehen, sind nicht unbedingt notwendig, schalten aber erweiterte Features wie die XP- oder Vista-übliche Darstellung der Steuerelemente ein:

Listing 3.2: Aufbau der Start-Klasse bei Windows.Forms-Anwendungen

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;

namespace Kompendium.Samples.Applications:
{
    static class Program
    {
        /* Der Haupteinstiegspunkt für die Anwendung. */
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new StartForm());
        }
    }
}
```

Im Unterschied zu einer Konsolenanwendung sind oben in der Quellcodedatei per Voreinstellung mehr Namensräume eingebunden. Diese sind nicht alle für die Start-Klasse wirklich notwendig. Es handelt sich dabei aber um die Standard-Namensräume, die Visual Studio in jede neue Klasse einfügt.

Die Main-Methode ist prinzipiell dieselbe Methode, die auch in einer Konsolenanwendung verwendet wird. Visual Studio hat das args-Argument weggelassen, weil Windows-Anwendungen nur in den seltensten Fällen Befehlszeileargumente übergeben werden. Sie können dieses Argument aber auch von Hand in die Parameterliste der Main-Methode einfügen wenn die Anwendung Befehlszeilenargumente unterstützen soll.

Ein anderer Unterschied ist, dass die Methode mit dem STAThread-Attribut markiert ist. Attribute, die in Kapitel 6 vertieft behandelt werden, ermöglichen die Angabe von Metadaten für Typen und deren Elemente. Diese speziellen Zusatzdaten werden vom

Compiler, von der CLR oder von Programmen ausgewertet und können die unterschiedlichsten Bedeutungen besitzen. Das `STAThread`-Attribut ist enorm wichtig (wie ich im nächsten Abschnitt noch näher ausführe), löschen Sie dieses also niemals.

Der letzte Unterschied ist, dass die `Main`-Methode speziell kommentiert ist. Diese Kommentare beginnen mit drei Schrägstrichen. Dabei handelt es sich um Dokumentationskommentare, aus denen später eine Programmdokumentation erzeugt werden kann. Programmdokumentationen werden in Kapitel 8 behandelt.

Das STAThread-Attribut

Das STAThread-Attribut ist wichtig für COM

Das `STAThread`-Attribut hat eine Bedeutung für das seit langer Zeit in Windows eingesetzte COM-Modell. Es führt dazu, dass die Anwendung in einem »Single Threaded Apartment« (STA) ausgeführt wird. Eine Erläuterung dieses Begriffs möchte ich Ihnen ersparen, schließlich lesen Sie gerade ein Buch über .NET und nicht über das sehr komplexe COM-Modell (das glücklicherweise fast tot ist). Da aber viele der vom Betriebssystem zur Verfügung gestellten Features, wie z. B. der Datei-Öffnen-Dialog, COM-Komponenten sind, kommen Sie mit dem COM-Modell zumindest indirekt in Kontakt.

HALT

Die Kommunikation zwischen `Windows.Forms` *und COM ist nur korrekt möglich, wenn die Anwendung in einem Single Threaded Apartment ausgeführt wird. Geben Sie das* `STAThread`*-Attribut nicht an, läuft die Anwendung in einem freien Thread. Das kann dann in einigen Fällen zu Kommunikations-Problemen zwischen COM und der Anwendung führen. Der Datei-Öffnen-Dialog zeigt z. B. für einige Ordner keinen Inhalt mehr an, wenn das* `STAThread`*-Attribut nicht angegeben ist. Glücklicherweise warnt Visual Studio beim Ausführen des Programms im Debugger bei der Verwendung von COM-Objekten, wenn dieses Attribut fehlt. Da diese Warnung in der kompilierten Datei aber nicht ausgegeben wird, sollten Sie immer darauf achten, dass das* `STAThread`*-Attribut angegeben ist.*

Eigentlich sollte das auch für Konsolenanwendungen gelten, zumindest wenn Sie darin auf die `Windows.Forms`*-Assembly verweisen und Klassen dieses Namensraums verwenden.*

3.2 Assemblys und Namensräume

Assemblys müssen referenziert werden

Wenn Sie die in einer Assembly enthaltenen Typen in Ihren Programmen verwenden wollen, muss das jeweilige Projekt diese Assembly referenzieren. Die einzige Assembly, die nicht referenziert werden muss, weil das der Compiler automatisch macht, ist *mscorlib.dll*. Diese Assembly enthält die grundlegenden Typen des .NET Framework in verschiedenen Namensräumen.

In Visual Studio verwenden Sie zur Referenzierung von Assemblys den `Referenzen`-Eintrag im Projektmappen-Explorer, so wie ich es bereits in Kapitel 2 gezeigt habe. Wenn Sie den Kommandozeilencompiler verwenden, geben Sie die zu referenzierenden Assemblys im `reference`-Argument an.

Typen können voll qualifiziert verwendet werden

Wenn Sie die in einem Namensraum enthaltenen Typen in Ihrem Programm verwenden wollen, können Sie diese unter Angabe des Namensraums voll qualifiziert angeben. Wollen Sie beispielsweise über die `Show`-Methode der `MessageBox`-Klasse eine Meldung ausgeben, können Sie die entsprechende Anweisung so schreiben:

```
System.Windows.Forms.MessageBox.Show("Hello World");
```

Sie können den Namensraum aber auch über eine `using`-Direktive, die ganz oben im Quellcode stehen muss, quasi importieren:

using erleichtert die Verwendung von Typen

```
using System.Windows.Forms;
```

Dann können Sie alle darin enthaltenen Typen ohne Angabe des Namensraums verwenden:

```
MessageBox.Show("Hello World");
```

Falls der Compiler eine Kollision mit einem Typnamen eines anderen Namensraums meldet, können Sie diesen Typen trotz importiertem Namensraum auch voll qualifiziert (also mit Namensraum) angeben, um den Konflikt zu lösen.

Mit der `using`-Direktive können Sie für einen Namensraum oder eine Klasse auch noch einen Alias einrichten:

```
using mb = System.Windows.Forms.MessageBox;
```

Den Namensraum bzw. die Klasse können Sie dann über den Alias referenzieren:

```
mb.Show("Hello World");
```

Ich rate Ihnen allerdings von dieser Technik ab, weil es schwer zu erkennen ist, welche Klasse bzw. welcher Namensraum tatsächlich verwendet wird. Mit der Hilfe von IntelliSense und Codeausschnitten sollte es auch kein Problem sein, lange Typnamen ohne Alias zu verwenden.

Der globale Namensraum

Klassen, Strukturen und andere Typen müssen nicht unbedingt einem Namensraum zugeordnet sein. Wenn Sie einen Typen ohne Namensraum deklarieren, wird dieser dem so genannten globalen Namensraum zugeordnet. Der globale Namensraum ist nichts Besonderes, er enthält lediglich alle weiteren Namensräume und eben Typen, die keinem Namensraum zugeordnet sind. Da im .NET Framework alle Typen einem speziellen Namensraum angehören, sollten Sie darauf verzichten, Typen ohne Namensraum zu deklarieren.

3.3 Bezeichner und Schlüsselwörter

In C# müssen die Bezeichner für Variablen, Klassen, Strukturen, Felder, Eigenschaften, Methoden und andere Elemente, die Sie bei Programmierung deklarieren, den Standardregeln entsprechen: Bezeichner dürfen nur Buchstaben, Zahlen und den Unterstrich enthalten und müssen mit einem Buchstaben oder dem Unterstrich beginnen.

Bezeichner dürfen Buchstaben, Zahlen und den Unterstrich enthalten

Die folgenden Variablendeklarationen sind demnach gültig:

Listing 3.3: Variablendeklarationen mit gültigen Bezeichnern

```
int number;
int number1;
int _number;
```

Die folgenden Deklarationen sind ungültig:

Listing 3.4: Variablendeklarationen mit ungültigen Bezeichnern

```
int number 1; // Leerzeichen nicht erlaubt
int number@1; // Sonderzeichen nicht erlaubt
int 1Number;  // Zahl am Anfang nicht erlaubt
```

Der Compiler beschwert sich außerdem, wenn Sie für einen Bezeichner eines der reservierten C#-Schlüsselwörter verwenden (Tabelle 3.1).

Tabelle 3.1: Die reservierten C#-Schlüsselwörter

abstract	do	In	protected	true
as	double	Int	public	try
base	else	interface	readonly	typeof
bool	enum	internal	ref	uint
break	event	Is	return	ulong
byte	explicit	Lock	sbyte	unchecked
case	extern	Long	sealed	unsafe
catch	false	namespace	short	ushort
char	finally	New	sizeof	using
checked	fixed	Null	stackalloc	virtual
class	float	Object	static	void
const	for	operator	string	volatile
continue	foreach	Out	struct	while
decimal	goto	override	switch	
default	if	Params	this	
delegate	implicit	private	throw	

@ erlaubt auch Schlüsselwörter als Bezeichner

Sie können ein Schlüsselwort aber trotzdem als Bezeichner einsetzen, indem Sie diesem ein @ voranstellen:

Listing 3.5: Deklaration einer Variablen mit Namen eines Schlüsselworts

```
int @in;
```

Sinn macht das aber nur in den seltensten Fällen.

C# enthält zudem einige kontextuelle Schlüsselwörter, deren Bedeutung vom Kontext abhängt (Tabelle 3.2). Diese können Sie jederzeit auch als Bezeichner einsetzen (auch wenn das ebenfalls in der Regel keinen Sinn macht ...).

Tabelle 3.2: Die kontextuellen C#-Schlüsselwörter

add	from	join	Remove	where
ascending	get	let	Select	yield
by	global	on	Set	
descending	group	orderby	Value	
equals	into	partial	Var	

3.4 Anweisungen

Nun geht es (endlich) los mit der eigentlichen Sprache. Dieser Abschnitt behandelt dazu zunächst elementare Anweisungen, Kommentare und den Aufruf von Methoden. Strukturanweisungen, über die Sie Ihre Programme strukturieren können, werden erst im Abschnitt »Verzweigungen und Schleifen« ab Seite 216 behandelt, nach den dafür notwendigen Grundlagen.

3.4.1 Elementare Anweisungen

Die Mutter aller Programme, die elementare Anweisung, entspricht in C# syntaktisch der Anweisung von C++ oder Java. Eine C#-Anweisung wird immer mit einem Semikolon abgeschlossen:

```
Console.WriteLine("Hello World");
```

Daraus folgt, dass Sie Anweisungen einfach in die nächste Zeile umbrechen können:

```
Console.WriteLine(
   "Hello World");
```

Das Umbrechen eines Zeichenketten-Literals ist allerdings nicht so einfach. Die folgende Anweisung:

```
Console.WriteLine("Hello
   World");
```

ergibt direkt mehrere Syntaxfehler. U. a. meldet der Compiler (bzw. Visual Studio) den Fehler, dass eine schließende Klammer erwartet wird.

Zeichenketten müssen prinzipiell immer in der aktuellen Zeile abgeschlossen und in einer neuen Zeile wieder begonnen werden (außer, Sie verwenden so genannte wortwörtliche Zeichenketten, die ich im Abschnitt »Zeichen und Zeichenketten« auf Seite 182 beschreibe). Wollen Sie eine Zeichenkette umbrechen (was in der Praxis sehr häufig vorkommt), schließen Sie diese ab und verbinden sie mit der Teilzeichenkette in der nächsten Zeile über den Plus-Operator.

Die folgende Anweisung ist korrekt:

```
Console.WriteLine("Hello " +
   "World");
```

Elementare Anweisungen sind entweder Zuweisungen von arithmetischen oder logischen Ausdrücken an Variablen:

```
i = 1 + 1;
```

oder Aufrufe von Methoden:

```
Console.WriteLine("Hello World");
```

Arithmetische und logische Ausdrücke arbeiten mit Operatoren, die in diesem Kapitel später (ab Seite 201) behandelt werden. Das Beispiel verwendet die Operatoren = (Zuweisung) und + (Addition). Den Aufruf von Methoden behandelt dieses Kapitel ab Seite 152.

C# unterscheidet Groß- und Kleinschreibung

Bei allen Anweisungen müssen Sie beachten, dass C# Groß- und Kleinschreibung unterscheidet. Jedes Element einer Anweisung muss genauso geschrieben werden, wie es ursprünglich deklariert wurde. Sie können z. B. nicht

```
console.writeline("Hello World");
```

schreiben.

3.4.2 Anweisungsblöcke

Anweisungen müssen häufig zusammengefasst werden. Eine Methode enthält z. B. einen Block von Anweisungen:

```
private static void SayHello()
{
   string name;
   Console.Write("Ihr Name: ");
   name = Console.ReadLine();
   Console.WriteLine("Hallo " + name + ", wie geht's?");
}
```

Blöcke werden in C#, wie in C++ und in Java, mit geschweiften Klammern umschlossen. Wie in anderen Sprachen auch, kann ein Block überall dort eingesetzt werden, wo eine einzelne Anweisung erwartet wird. Die `if`-Verzweigung kann z. B. mit einzelnen Anweisungen arbeiten.

Die folgende Anweisung überprüft, ob heute Sonntag ist, und gibt in diesem Fall einen entsprechenden Text an der Konsole aus:

```
if (System.DateTime.Now.DayOfWeek == 0)
   Console.WriteLine("Heute ist Sonntag.");
```

Wenn für einen Fall, dass heute Sonntag ist, nun aber mehrere Anweisungen ausgeführt werden sollen, müssen Sie einen Block einsetzen:

```
if (System.DateTime.Now.DayOfWeek == 0)
{
   Console.WriteLine("Heute ist Sonntag.");
   Console.WriteLine("Heute wird nicht gearbeitet.");
   Console.WriteLine("Heute gehen wir snowboarden.");
}
```

INFO

Falls Sie die letzte Anweisung nicht verstehen, weil Sie irgendwo gelesen haben, dass ich am Niederrhein wohne (zurzeit allerdings in meinem anderen Wohnsitz in Dublin): Auch am Niederrhein kann man snowboarden. In Neuss. In der Jever Skihalle. Sogar im Sommer ☺.

Würden Sie die Blockklammern weglassen,

```
if (System.DateTime.Now.DayOfWeek == 0)
   Console.WriteLine("Heute ist Sonntag.");
   Console.WriteLine("Heute wird nicht gearbeitet.");
   Console.WriteLine("Heute gehen wir snowboarden.");
```

würde der Compiler nur die erste Anweisung der `if`-Verzweigung zuordnen und die zweite und dritte immer ausführen, auch dann, wenn kein Sonntag ist. Ich fände das zwar recht nett (jeden Tag snowboarden ...), korrekt ist die Programmierung aber nicht.

Die Blockklammern teilen dem Compiler also mit, dass die darin enthaltenen Anweisungen zusammengehören.

Verwenden Sie grundsätzlich für alle Strukturanweisungen (wie `if` *und* `while`*) für die enthaltenen Anweisungen Blockklammern, auch wenn nur eine Anweisung enthalten ist. Damit erhöhen Sie die Lesbarkeit, Wartbarkeit und Erweiterbarkeit Ihrer Programme und sorgen wahrscheinlich gleich auch noch dafür, dass diese fehlerfreier sind.*

Schreiben Sie also besser:

```
if (System.DateTime.Now.DayOfWeek == 0)
{
   Console.WriteLine("Heute ist Sonntag.");
}
```

an Stelle von:

```
if (System.DateTime.Now.DayOfWeek == 0)
   Console.WriteLine("Heute ist Sonntag.");
```

Ich setze diese Konvention in diesem Buch (hoffentlich ☺) konsequent ein.

3.4.3 Sichere und unsichere Anweisungen

Normale Anweisungen sind in C# sicher. Die CLR überprüft sichere Anweisungen daraufhin, ob diese u. U. Operationen ausführen, die Speicherbereiche beeinflussen würden, die nicht zum aktuellen Kontext gehören. Die CLR erlaubt deswegen z. B. keine Zeiger, weil diese es einem Programm erlauben, beliebig in den Speicher zu schreiben und daraus zu lesen.

Wenn Sie unsichere Features wie Zeiger verwenden wollen (was nur in absoluten Ausnahmefällen sinnvoll ist), können dazu unsichere Anweisungsblöcke einsetzen. Diese Blöcke leiten Sie mit dem Schlüsselwort `unsafe` ein:

```
unsafe
{
   ...
}
```

Alternativ können Sie ganze Methoden als unsicher kennzeichnen:

```
unsafe void UnsafeDemo()
{
 ...
}
```

In einem unsicheren Block können Sie alle Anweisungen unterbringen, auch solche, die die CLR als unsicher erkennen würde. Unsichere Blöcke benötigen Sie immer dann, wenn Sie direkt auf den Speicher zugreifen wollen.

Normalerweise müssen Sie dies nur tun, wenn Sie extrem schnelle Programme schreiben wollen. Mit Zeigern können Sie eben direkt mit dem Speicher arbeiten. Zeiger verursachen aber auch massive Probleme, da Sie damit versehentlich Speicherbereiche überschreiben oder auslesen können, die Sie gar nicht reserviert haben. Da dieses Thema wohl eher die C++ -Programmierer interessiert, die systemnahe Programme wie z. B. Treiber entwickeln, behandle ich unsichere Programmierung in diesem Buch nicht weiter.

3.4.4 Kommentare

Über Kommentare wird Quellcode kommentiert

Kommentare gehören auch zu den Anweisungen, allerdings werden Kommentare vom Compiler nicht berücksichtigt. Sie dienen lediglich der Kommentierung des Quellcodes.

C# kennt vier Arten von Kommentaren: *Einfache Kommentare*, *mehrzeilige einfache Kommentare*, *Aufgabenkommentare* und *Dokumentationskommentare*. Unabhängig von der Art des Kommentars kompiliert der Compiler diese nicht mit in die Assembly.

TIPP

Kommentare im Quellcode sind in den meisten Programmen der Schlüssel zum Verständnis eines Programms. Gute Programme enthalten etwa so viele Kommentarzeilen wie Programmzeilen. Kommentieren Sie also immer ausführlich.

Einfache Kommentare

Einfache Kommentare enthalten lediglich einen beschreibenden Text, der das Verständnis eines Programmteils erleichtert. Einfache Kommentare können Sie an das Ende einer Zeile anfügen, indem Sie diese mit zwei Schrägstrichen einleiten:

```
Console.WriteLine("Hello World"); /* Das ist ein einfacher Kommentar
```

Sie können diese aber natürlich auch in separaten Zeilen unterbringen (was ich bevorzuge):

```
// Das ist ein einfacher Kommentar
Console.WriteLine("Hello World");
```

Mehrzeilige einfache Kommentare werden mit /* eingeleitet und mit */ beendet:

```
Console.WriteLine("Hello World"); /* Das ist ein
   mehrzeiliger Kommentar */
```

Visual Studio beginnt die zweite Zeile eines mehrzeiligen Kommentars automatisch mit einem Stern:

```
/* Das ist ein mehrzeiliger Kommentar,
 * der mit Visual Studio
 * erzeugt wurde. */
```

Die Sterne vor den einzelnen Zeilen dienen lediglich der optischen Darstellung und haben keine weitere Bedeutung.

Aufgabenkommentare

Aufgabenkommentare werden in der Aufgabenliste angezeigt

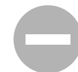
Wie ich bereits in Kapitel 2 beschrieben habe, können Sie den Text eines Kommentars automatisch in die Aufgabenliste übernehmen, wenn Sie den Kommentar mit einem Aufgabentoken beginnen:

```
static void Main()
{
   // TODO: Den Rest programmieren :-)
}
```

Visual Studio fügt neuen Projekten per Voreinstellung die Aufgabentoken `HACK`, `TODO`, `UNDONE` und `UnresolvedMergeConflict` hinzu. Ich nutze in meinen Programmen lediglich `TODO` für Stellen, an denen noch gearbeitet werden muss, und `HACK` für Workarounds um Probleme, für die ich keine zufrieden stellende Lösung gefunden habe.

Sie sollten diese speziellen Aufgabenkommentare immer dann nutzen, wenn irgendwo im Quellcode noch irgendetwas zu tun ist. Da diese Kommentare immer dann automatisch in der Aufgabenliste erscheinen, wenn die entsprechende Datei

geöffnet ist, und Sie diese Liste auch gut filtern können (etwa nur die Aufgabenkommentare anzeigen lassen), haben Sie eine gute Übersicht über die noch zu erledigenden Aufgaben.


HALT

Sie sollten allerdings beachten, dass Visual Studio nur die Aufgabenkommentare der aktuell geöffneten Dateien anzeigt. Aufgabenkommentare von nicht geöffneten Dateien werden (unverständlicherweise) nicht in der Aufgabenliste angezeigt. Eine Lösung dieses Problems ist die zusätzliche Definition einer Verknüpfung mit STRG *+* K *+* H *auf einem Aufgabenkommentar. Verknüpfungen werden in der Aufgabenliste auch dann angezeigt, wenn die betroffene Datei geschlossen ist. Zur Anzeige von Verknüpfungen müssen Sie allerdings den Filter der Aufgabenliste entsprechend einstellen.*

Dokumentationskommentare

Aus Dokumentationskommentaren kann eine externe Dokumentation erzeugt werden

Zur Dokumentation eines Programms gehören nicht nur einfache Kommentare, die den Quelltext verständlicher machen. Ein wichtiger Teil der Dokumentation ist der, der die Klassen eines Programms und deren Methoden, Eigenschaften und andere Elemente für andere Programmierer beschreibt. Die .NET Framework-Dokumentation ist ein gutes Beispiel dafür.

Diese Art der Dokumentation ist besonders wichtig für Klassen, die in Klassenbibliotheken veröffentlicht und somit auch von anderen Programmierern verwendet werden. Sind die Klassen verständlich und ausführlich dokumentiert, müssen Programmierer, die Ihre Klassenbibliotheken anwenden, sich nicht an Sie wenden, um zu erfahren, wie Ihre Klassen verwendet werden. Und Sie haben dann wieder etwas mehr Zeit zum Windsurfen, Snowboarden oder für Ihre Kinder.

Eine gute Dokumentation zu erstellen erfordert einiges an Arbeit. Die meiste Arbeit haben Sie beim Schreiben der Dokumentationskommentare. Das Erstellen einer Dokumentation selbst ist dann relativ einfach, allerdings leider nicht (mehr[1]) mit Visual Studio möglich.

Die Basis für diese Dokumentation ist eine XML-Datei, die vom Compiler erzeugt werden kann. In Visual Studio geben Sie dazu einfach den Namen der Datei in den Projekteigenschaften an. Diese XML-Datei ist für Menschen wenig hilfreich, enthält aber alle Informationen, die zum Erstellen einer auch für Menschen lesbaren Dokumentation notwendig sind. Über ein externes Tool (z. B. Doxygen) können Sie aus der XML-Datei dann eine ansprechend formatierte Dokumentation erzeugen. Wie das geht, zeige ich in Kapitel 8.

Dokumentationskommentare werden mit drei Schrägstrichen eingeleitet. Mit diesen Kommentaren können Sie alle Typen (Klassen, Strukturen, Aufzählungen etc.) und deren Elemente (Methoden, Felder, Eigenschaften, Ereignisse etc.) dokumentieren. Innerhalb der Kommentare können Sie einfachen Text und XML-Elemente unterbringen. Über XML-Elemente können Sie eine eigene XML-Struktur aufbauen (was aber in der Praxis nur sehr selten verwendet wird). Sie können eigene XML-Elemente einfügen (die Sie dann aber auch selbst in der XML-Datei auswerten müssen) oder einige der vordefinierten (die von den Tools zur Erzeugung einer HTML-Dokumentation verwendet werden). Das Element `summary` wird z. B. verwendet, um einen Typen oder ein Element zusammenfassend zu beschreiben:

1 In Visual Studio 2003 war das Erstellen einer HTML-Dokumentation noch möglich.

```
/// <summary>
/// Der Einstiegspunkt der Anwendung
/// </summary>
[STAThread]
static void Main()
...
```

Zur Erstellung der XML-Datei geben Sie deren Dateinamen in den Projekteigenschaften im Register ERSTELLEN an. Wählen Sie dazu die Option XML-DOKUMENTATIONSDATEI und ändern Sie ggf. den vorgeschlagenen Dateipfad. Beachten Sie, dass Sie diese Einstellung für jede Konfiguration (*Debug*, *Release*) einzeln vornehmen können bzw. müssen. Wenn Sie das Projekt erstellen oder mit der F5-Taste starten, wird die XML-Dokumentationsdatei automatisch erzeugt.

In Kapitel 8 erfahren Sie dann mehr über Dokumentationskommentare. Außerdem stelle ich in diesem Kapitel ein Tool vor, über das Sie ansprechende Dokumentationen erzeugen können.

3.4.5 Der Aufruf von Methoden

Methoden enthalten, wie Sie ja wahrscheinlich bereits wissen, vorgefertigte Programme, die Sie über den Namen der Methode aufrufen können. Der Aufruf von Methoden ist ein wichtiger Teil beim Schreiben von Anweisungen.

Klassen- und Instanzmethoden

Das Verständnis des Unterschieds zwischen Klassen- und Instanzmethoden ist ein wichtiger Schlüssel zum Verständnis des Aufrufs von Methoden. Dummerweise erfordert dies schon ein wenig OOP-Kenntnisse. Dieses Buch behandelt die OOP ab Kapitel 4. Ich will aber bereits hier versuchen, den Unterschied zwischen Klassen- und Instanzmethoden zu erläutern. Wenn Sie den Unterschied bereits kennen, können Sie diesen Abschnitt (natürlich) auslassen.

Eine Klasse ist ein Bauplan für Objekte

Wie ich in der Einführung bereits beschrieben habe, enthalten Klassen in der reinen OOP die Beschreibung von Objekten, die später, im Programm aus diesen Klassen erzeugt werden. Eine Klasse ist damit so etwas wie ein Bauplan für Objekte. Im Wesentlichen beschreibt eine Klasse, welche Methoden, Eigenschaften und Ereignisse die Objekte besitzen, die später aus der Klasse erzeugt werden. Wenn Sie einem Formular einer Windows-Anwendung z. B. `TextBox`-Steuerelemente hinzufügen, führt das in Wirklichkeit dazu, dass in der Laufzeit der Anwendung Instanzen der Klasse `TextBox` erzeugt und dem Formular hinzugefügt werden. Die Klasse `TextBox` beschreibt, welche Eigenschaften, Methoden und Ereignisse diese `TextBox`-Objekte besitzen.

Objekte (Instanzen von Klassen) können Sie aber auch in Ihren Programmen erzeugen, was in diesem Buch noch sehr ausführlich genutzt wird.

Instanzmethoden gehören zu Instanzen

Instanzmethoden sind nun alle Methoden, die ein *Objekt* besitzt. Diese Methoden können nur über das Objekt (eine Instanz) aufgerufen werden. `TextBox`-Instanzen besitzen z. B. die Methode `Focus`, die bewirkt, dass der Eingabecursor in die `TextBox` gestellt wird.

Klassenmethoden gehören zu Klassen

Klassenmethoden (die auch als *statische Methoden* bezeichnet werden) haben eine andere Bedeutung. Sie gehören nicht zu den Objekten, die aus einer Klasse erzeugt werden, sondern zu der *Klasse*. Das ist vielleicht ein wenig schwer zu verstehen, aber im Wesentlichen geht es bei Klassenmethoden darum, dass diese aufgerufen werden können, ohne dass Instanzen der Klasse erzeugt werden müssen. Ein gutes Beispiel für solche Methoden sind die der `Math`-Klasse, die mathematische Berech-

nung erlauben. Es wäre ziemlich unsinnig, wenn Sie zur Berechnung des Minimums von zwei Werten zunächst eine Instanz der `Math`-Klasse erzeugen müssten, um dann die `Min`-Methode des `Math`-Objekts aufrufen zu können. Für solch allgemeine Aufgaben wie mathematische Berechnungen sind Klassenmethoden, die eben direkt über die Klasse aufgerufen werden können, wesentlich besser geeignet.

Ich hoffe, ich habe den Unterschied damit einigermaßen gut erläutert ☺.

Grundlagen zum Aufruf von Methoden

Methoden werden immer über deren Namen aufgerufen, gefolgt von Klammern, in denen Sie der Methode in vielen Fällen Argumente übergeben, die die Ausführung steuern. Handelt es sich um eine Klassenmethode, geben Sie den Namen der Klasse als Präfix an. Den Präfix trennen Sie durch einen Punkt vom Methodennamen. Das folgende Beispiel ruft die Klassenmethode `WriteLine` der `Console`-Klasse auf:

```
Console.WriteLine("Hello World");
```

Handelt es sich um eine Instanzmethode, geben Sie an Stelle der Klasse die Variable an, die die Instanz der Klasse (das Objekt) verwaltet. Das folgende Beispiel erzeugt zwei Instanzen der `StreamWriter`-Klasse (die zum Schreiben von Texten in Dateien verwendet wird) und ruft deren `WriteLine`-Methode auf.

Listing 3.6: Aufrufen von Instanzmethoden am Beispiel von StreamWriter-Instanzen

```
StreamWriter sw1 = new StreamWriter("C:\\Demo1.txt");
sw1.WriteLine("Hello World");
StreamWriter sw2 = new StreamWriter("C:\\Demo2.txt");
sw1.WriteLine("Und hier auch: Hello World");
```

Das Erzeugen von Instanzen soll an dieser Stelle noch nicht behandelt werden. Das Beispiel benötigt jedoch eine Instanz, da eine Instanzmethode aufgerufen wird. Die `WriteLine`-Methode, die hier verwendet wird, ist in der Klasse `StreamWriter` definiert, aber eben nicht als Klassenmethode (wie die gleichnamige Methode in der `Console`-Klasse), sondern als Instanzmethode. Deswegen kann sie nur über eine Instanz der Klasse (ein Objekt) aufgerufen werden.

EXKURS

Falls Sie sich wundern, dass die `Console`- und die `StreamWriter`-Klasse eine gleich aussehende `WriteLine`-Methode besitzen: Das Schreiben von Daten basiert in beiden Klassen auf dem Stream-Konzept, und dieses stellt eben eine `WriteLine`-Methode zur Verfügung, die beide Klassen auf identische Weise nutzen. Die `Console`-Klasse schreibt die Daten aber an die Konsole, die `StreamWriter`-Klasse schreibt die Daten in eine Datei (bzw. in einen beliebigen Stream).

An diesen Beispielen können Sie auch gut den Unterschied zwischen Klassen- und Instanzmethoden erkennen: In einer Konsolenanwendung existiert die Konsole nur genau ein Mal. Deswegen wäre es unsinnig, wenn zunächst Instanzen der `Console`-Klasse erzeugt werden müssten, um mit der Konsole zu arbeiten. Dateien können hingegen (in mehreren `StreamWriter`-Instanzen) beliebig viele geöffnet werden. Es ist also absolut notwendig, dass die Methoden der `StreamWriter`-Klasse sich auf die Datei beziehen, die das jeweilige `StreamWriter`-Objekt verwaltet. Deswegen *müssen* diese Methoden Instanzmethoden sein.

Sie geben also normalerweise beim Aufruf von Klassenmethoden den Namen der Klasse und beim Aufruf von Instanzmethoden den Namen der das Objekt referenzierenden Variable als Präfix an. Eine Ausnahme von dieser Regel ist, wenn Sie Methoden aufrufen, die in der Klasse deklariert sind, in der der Aufruf erfolgt. Dann *können* Sie

den Klassen- bzw. Instanznamen weglassen. Sie können (und sollten) in diesem Fall bei Klassenmethoden aber auch den Klassennamen und bei Instanzmethoden eine Referenz auf die Instanz der Klasse angeben. Eine Referenz auf das Objekt, das später aus einer Klasse erzeugt wird, erhalten Sie innerhalb der Klasse über das Schlüsselwort `this`. Darüber, und über Klassenmethoden, erfahren Sie noch mehr in Kapitel 4.

Methoden und Namensräume

Wie Sie ja bereits wissen, sind Klassen in Namensräumen organisiert. Wenn Sie den Namensraum nicht über die `using`-Direktive »importiert« haben, müssen Sie dessen kompletten Namen beim Aufruf einer Methode angeben. Die `Console`-Klasse gehört z. B. zum Namensraum `System`. Den vollständigen Aufruf der `WriteLine`-Methode zeigt der folgende Quellcode:

```
System.Console.WriteLine("Hello World");
```

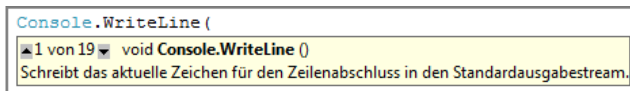
Binden Sie den `System`-Namensraum über `using` ein, müssen Sie nur noch die Klasse beim Aufruf angeben:

```
Console.WriteLine("Hello World");
```

Argumente übergeben

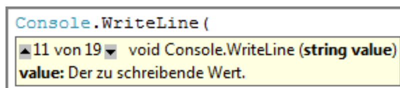
Die Deklaration der Methode legt fest, wie viele Argumente welchen Typs übergeben werden müssen. In Visual Studio zeigt IntelliSense beim Schreiben einer Anweisung die Deklaration(en) der Methode und damit die zu übergebenden Argumente an (Abbildung 3.1).

Abbildung 3.1: IntelliSense beim Schreiben von Anweisungen mit Methoden



Viele Methoden liegen in mehreren, so genannten »überladenen« Varianten vor. Das Überladen von Methoden erläutere ich noch näher für eigene Methoden in Kapitel 4, fürs Erste reicht es aus, dass Sie wissen, dass eine Methode auch in mehreren Varianten vorkommen kann. Die `WriteLine`-Methode kommt z. B. (zurzeit) in 19 Varianten vor. Die einzelnen Varianten können Sie im IntelliSense-Fenster über die Cursortasten oder einen Klick auf die Pfeile anzeigen lassen. Die Variante 11 erwartet z. B. nur einen einfachen String (eine Zeichenkette) als Argument (Abbildung 3.2).

Abbildung 3.2: Die Variante 11 der WriteLine-Methode



Gibt die Methode einen Wert zurück, können Sie diesen Wert in Zuweisungen, in arithmetischen oder logischen Ausdrücken oder als Argument einer anderen Methode verwenden. Stellen Sie sich einfach vor, dass der Compiler Methoden immer zuerst aufruft, bevor er weitere Teile der Anweisung auswertet und das Ergebnis der Methode dann als Wert in der Anweisung weiterverarbeitet. So sind einfache Zuweisungen möglich:

```
sinY = Math.Sin(y);
```

oder Ausdrücke:

```
result = Math.Min(x1, x2) / Math.Max(y1, y2);
```

oder geschachtelte Aufrufe:

```
result = Math.Min(y1, Math.Max(y2, y3));
```

Ein wichtiger Grundsatz beim Aufruf von Methoden ist: Überall da, wo ein bestimmter Datentyp erwartet wird, können Sie neben Literalen, Konstanten und Variablen immer auch eine Methode einsetzen, die diesen Datentyp zurückgibt.

3.5 (Daten)Typen

Typen kommen in einem Programm sehr häufig vor. Ein Ausdruck wie `1 + 1.5` beinhaltet z. B. Operanden, die einen bestimmten Typ besitzen. Im Beispiel sind das die »Typen« *Ganzzahl* (eigentlich: `int`) und *Dezimalzahl* (eigentlich: `double`). Und auch das Ergebnis eines Ausdrucks besitzt einen Typ (im Beispiel: `double`).

Daneben besitzen Variablen, Konstanten und Literale (einfache Wertangaben) einen Typ, genau wie Parameter und Rückgabewerte von Methoden.

C# unterstützt die üblichen Typen wie z. B. `int` (Integer) und `double`. Die in C# normalerweise verwendeten Typ-Schlüsselwörter sind allerdings nur Aliasnamen für Typen, die im .NET Framework definiert sind. `int` steht z. B. für `System.Int32`. Prinzipiell können Sie immer die .NET-Typen verwenden, z. B. um eine Variable zu deklarieren:

```
System.Int32 i;
```

Verwenden Sie aber lieber die C#-Aliasnamen, damit Ihr Quellcode besser lesbar wird.

3.5.1 Typsicherheit und der Typ Object

C# ist typsicher

C# ist eine typsichere Sprache. Wenn irgendwo ein bestimmter Typ erwartet wird, können Sie nur einen Typ einsetzen, der zum erwarteten passt. Die Zuweisung eines String-Literals an eine `int`-Variable ist z. B. nicht möglich:

```
string s = "10";
int i = s; // Fehler
```

Das Beispiel resultiert im Compilerfehler »Eine implizite Konvertierung vom Typ "string" in "int" ist nicht möglich«. C# sichert damit ab, dass Sie nicht versehentlich einen falschen Typ verwenden. Wollen Sie trotzdem einen anderen Typ einsetzen, als der Compiler erwartet, müssen Sie diesen explizit konvertieren. Wie das geht, zeige ich in Abschnitt »Konvertierungen« ab Seite 188. Deswegen folgt hier nur ein kleines Beispiel:

```
int i = Convert.ToInt32(s);
```

Ist der zugewiesene Typ kompatibel und kann er die zugewiesenen Daten ohne Verlust aufnehmen, wird er normalerweise automatisch (implizit) konvertiert. Die Zuweisung eines `byte`-Typs an einen `int`-Typ ist z. B. problemlos möglich, weil `byte` immer in einen `int`-Typ hineinpasst:

```
byte b = 10;
int i = b;
```

Alles ist ein Objekt

Alle Typen sind von Object abgeleitet

In C# sind alle Typen (auch eigene!) von der Basisklasse `Object` abgeleitet. `Object` stellt den .NET-Typen einige Methoden zur Verfügung, die ich im Abschnitt »Der Typ Object« (Seite 185) erläutere.

Object stellt einige Methoden zur Verfügung

In den von `Object` abgeleiteten Typen werden diese Methoden normalerweise (d. h. nicht unbedingt immer) mit einer neuen Implementierung überschrieben. Die `ToString`-Methode gibt z. B. bei einem `int`-Datentyp die gespeicherte Zahl als Zeichenkette zurück:

```
int i = 10;
Console.WriteLine(i.ToString());
```

Bei einem `double`-Typ wird die Zeichenkette unter Berücksichtigung einer Ländereinstellung zurückgegeben. `ToString` können Sie ohne Argument aufrufen, dann wird die Systemeinstellung verwendet:

```
double x = 1.234;
Console.WriteLine(x.ToString());
```

Auf deutschen Systemen kommt dabei der String »1,234« heraus. Alternativ können Sie diese Methode bei einem `double`-Wert auch mit einem Format-String aufrufen:

```
Console.WriteLine(x.ToString("0.00"));
```

Die Zahl wird nun auf zwei Stellen hinter dem Komma formatiert ausgegeben. Den Umgang mit Zeichenketten und deren Formatierung beschreibe ich in Kapitel 8.

Die Verwaltung aller Typen als Objekt ist sogar so konsequent, dass Sie auch für einfache Konstanten die Methoden des zugrunde liegenden Typs aufrufen können:

```
string s = 10.ToString();
```

Die verschiedenen Typen des .NET Framework bieten meist noch zusätzliche Methoden und Eigenschaften. Der Typ `String` besitzt z. B. eine Vielzahl an Methoden zur Arbeit mit Zeichenketten. Ein Beispiel dafür ist die `Replace`-Methode, über die Sie einen Teilstring durch einen anderen String ersetzen können.

Einige der weiteren Methoden der einzelnen Typen sind statisch. Diese (Klassen-) Methoden können Sie (wie Sie ja bereits wissen) aufrufen, ohne eine Instanz des Datentyps zu besitzen. Die `Format`-Methode der `String`-Klasse ist z. B. eine solche statische Methode:

```
Console.WriteLine(String.Format("{0:0.00}", 1.234));
```

Ich beschreibe die wichtigsten dieser Methoden in Kapitel 8.

3.5.2 Wert- und Referenztypen

C# unterscheidet bei den Typen grundsätzlich Wert- und Referenztypen. Zu den Werttypen gehören alle Standardtypen (außer `string`), Strukturen und Aufzählungen. Alle anderen Typen (um genau zu sein, außer Methoden und Delegaten, die auch Typen sind) sind Klassen und damit Referenztypen.

Ein wesentlicher Unterschied zwischen Wert- und Referenztypen ist, dass die Daten von Werttypen im Stack gespeichert werden, die Daten von Referenztypen werden auf dem Heap gespeichert.

EXKURS

Der Stack ist ein spezieller Speicherbereich, den der Compiler für jede aufgerufene Methode neu reserviert. Alle lokalen Daten einer Methode werden, sofern es sich um Werttypen handelt, auf dem Stack abgelegt. Der Stack wird auch intern verwendet, um Argumente an eine Methode zu übergeben. Der aufrufende Programmteil legt die Argumente, die an die Methode übergeben werden sollen, auf dem Stack ab, die Methode liest diese Argumente dann aus dem Stack aus.

Der Heap ist ein anderer Speicherbereich, der allerdings global für das gesamte Programm gilt und so lange besteht, wie das Programm läuft. Auf dem Heap werden üblicherweise programmglobale Daten, aber eben auch Referenztypen abgelegt.

Strukturen sind Werttypen, Klassen sind Referenztypen

Wenn Sie selbst Typen entwickeln, können Sie nur bei den strukturierten Typen (nicht bei Aufzählungen, die per Definition Werttypen sind) entscheiden, ob Sie einen Wert- oder einen Referenztypen implementieren: Entweder Sie programmieren eine Struktur (also einen Werttypen) oder eine Klasse (also einen Referenztypen). Wie das geht, zeige ich in Kapitel 4. Ich benötige für diesen Abschnitt allerdings Beispiele, deswegen zeige ich kurz, wie Sie eine einfache Struktur und eine einfache Klasse deklarieren. Die Typen im Beispiel verwalten Personendaten:

Listing 3.7: Struktur (Werttyp) und Klasse (Referenztyp), die Personendaten speichern

```
public struct PersonStruct
{
   public string FirstName;
   public string LastName;
}

public class PersonClass
{
   public string FirstName;
   public string LastName;
}
```

Wenn der Typ einer Variablen (oder eines Feldes oder einer Eigenschaft) ein Werttyp ist, speichert die Variable den Wert direkt (auf dem Stack). Ist der Typ einer Variablen ein Referenztyp, speichert die Variable nicht den Wert, sondern lediglich die Adresse des Speicherbereichs, der den Wert verwaltet. Die Adresse (der Wert der Variablen) wird auf dem Stack verwaltet, der eigentliche Wert auf dem Heap.

Wenn Sie z. B. auf den oben deklarierten Beispieltypen in einer Methode je zwei Objekte erzeugen, sieht der Speicher schematisch dargestellt aus wie in Abbildung 3.3.

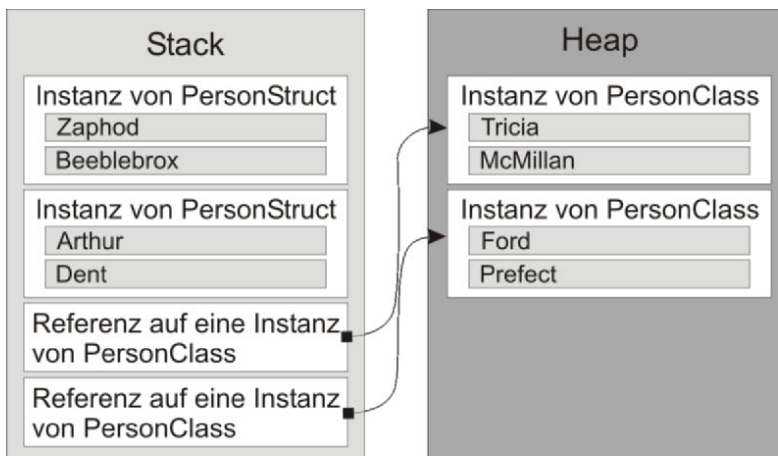


Abbildung 3.3: Schematische Darstellung des Stack und des Heap nach der Erzeugung von je zwei PersonStruct- und PersonClass-Objekten

Unterschiede zwischen Wert- und Referenztypen

Werttypen werden schneller gelesen und geschrieben

Da Werttypen direkt im Stack gespeichert sind, kann ein Programm den Wert dieser Typen sehr performant lesen und schreiben. Um auf einen Referenztypen zuzugreifen, benötigt ein Programm hingegen zwei Schritte: Es muss zunächst die Adresse der Daten aus dem Stack auslesen und kann erst dann mit Hilfe dieser Adresse auf die Daten im Heap zugreifen. Referenztypen werden also prinzipiell etwas langsamer bearbeitet als Werttypen. Ich denke aber nicht, dass dieser Unterschied zwischen Wert- und Referenztypen in der Praxis (bei den heutigen schnellen Rechnern) eine große Rolle spielt.

EXKURS

In einem Test (den Sie auch in den Beispielen auf der Buch-DVD finden) habe ich ermittelt, dass auf meinem Rechner das Erzeugen von 1.000.000 einfachen Objekten ca. 0,0026 Sekunden benötigte, wenn es sich um Werttypen handelte. Bei Referenztypen benötigte das Erzeugen ca. 0,052 Sekunden. Werttypen wurden in diesem Test also etwa 20-mal schneller erzeugt als Referenztypen. Das Schreiben einer Eigenschaft dieser Typen benötigte ca. 0,0019 Sekunden bei Wert- und ca. 0,0089 Sekunden bei Referenztypen (jeweils natürlich auch wieder bei 1.000.000 Instanzen). Werttypen waren auch hier schneller, allerdings nur etwa 4,7-mal. Beim Lesen war das Verhältnis etwas ausgeglichener (0,0017 Sekunden zu 0,0042 Sekunden, was dem Faktor 2,4 entspricht).

Wert- und Referenztypen verhalten sich bei Zuweisungen unterschiedlich

Der wesentliche Unterschied zwischen Wert- und Referenztypen basiert auf ihrem Verhalten bei Zuweisungen und bei der Übergabe an Methoden: Wenn Sie einen Werttypen auf einen anderen Werttypen zuweisen oder einen Werttypen an ein Argument einer Methode übergeben, erzeugt der Compiler immer eine *Kopie* der gespeicherten Daten (des Objekts). Die Übergabe von Werttypen, die große Datenmengen speichern, an Methoden, kann somit recht viel Zeit in Anspruch nehmen.

Bei Referenztypen wird bei Zuweisungen oder beim Übergeben an Methoden allerdings keine Kopie des Objekts erzeugt. Die Referenz, die den Referenztypen referenziert (die Variable), speichert schließlich nur die *Adresse* des Objekts und nicht das Objekt selbst. Wenn Sie eine Variable, die einen Referenztypen verwaltet, auf eine andere Variable oder an ein Argument einer Methode zuweisen, wird nur die Adresse des Objekts kopiert, nicht das Objekt selbst. Die Übergabe von Referenztypen an Methoden ist deswegen wesentlich performanter und speicherschonender als die Übergabe von Werttypen. Referenztypen sind somit zur Übergabe an Methoden besser geeignet als Werttypen.

Referenztypen erlauben beliebig viele Referenzen

Ein weiteren Unterschied ist, dass auf Referenztypen mehrere beliebige Referenzen zeigen können. Das ergibt sich schon daraus, dass bei Zuweisungen oder bei der Übergabe eines Referenztyps an eine Methode nur die Referenzen kopiert werden. Dieses Feature wird bei der objektorientierten Programmierung sehr häufig verwendet. Mit Werttypen ist das prinzipiell nicht möglich, weil der Compiler immer die Werte kopiert[2].

INFO

Wenn Sie mit Typen arbeiten, sollten Sie das Verhalten von Wert- und Referenztypen bei Zuweisungen und bei der Übergabe an Methoden immer im Auge haben.

2 Eine Ausnahme ist die Übergabe von Argumenten an eine Methode mit der Übergabeart »By Reference« (siehe Kapitel 4).

Werttypen sind von ValueType abgeleitet

Ein letzter Unterschied ist schließlich, dass Werttypen von der Klasse `ValueType` abgeleitet sind und Referenztypen nicht. `ValueType` überschreibt die von `Object` geerbte Methode `Equals`, die das Objekt, auf dem sie aufgerufen wird, mit dem übergebenen Objekt auf Gleichheit überprüft. In der originalen `Object`-Variante überprüft diese Methode lediglich, ob die Referenzen der beiden Objekte gleich sind. In der `ValueType`-Variante vergleicht diese Methode alle Felder auf Gleichheit (wozu Reflection verwendet wird, weswegen diese Methode unter Umständen relativ langsam ist).

Eine in einigen besonderen Situationen hilfreiche Technik ist, dass Sie bei einem Objekt über eine Abfrage auf `ValueType` ermitteln können, ob es sich um einen Werttypen handelt (was u. U. wichtig ist, wenn Sie mit einer Variablen vom Typ `object` arbeiten):

Listing 3.8: Abfrage darauf, ob ein Objekt ein Wert- oder Referenztyp ist

```
object o = 10;
if (o is ValueType)
{
   Console.WriteLine("o ist ein Werttyp");
}
else
{
   Console.WriteLine("o ist ein Referenztyp");
}
o = new PersonClass();
if (o is ValueType)
{
   Console.WriteLine("o ist jetzt ein Werttyp");
}
else
{
   Console.WriteLine("o ist jetzt ein Referenztyp");
}
```

EXKURS

Was mich anfangs immer wunderte, war die Tatsache, dass Werttypen genau wie Referenztypen von `Object` abgeleitet sind. `Object` ist aber eine Klasse und damit ein Referenztyp. Wird normalerweise von einem Referenztyp abgeleitet, resultiert immer auch ein Referenztyp. .NET lässt auch gar nicht zu, dass ein Werttyp (eine Struktur) von einem Referenztyp (einer Klasse) abgeleitet wird, da Strukturen nicht abgeleitet werden können.

Die Lösung dieses Problems habe ich nicht recherchieren können und deshalb selbst hergeleitet ☺:

Werttypen sind in Wirklichkeit auch Instanzen von Klassen. Eine Variable, deren Typ ein Werttyp ist, ist in Wirklichkeit auch eine Referenz auf ein Objekt. Werttypen werden aber vom Compiler und von der CLR anders behandelt als Referenztypen. Werttypen werden im Stack angelegt, Referenztypen im Heap. Bei Zuweisungen werden Werttypen kopiert, Referenztypen allerdings nicht. Wie aber erkennen der Compiler und die CLR, dass es sich bei einem Objekt um einen Werttypen handelt?

Die Lösung ist einfach: Werttypen sind, wie ich ja bereits beschrieben habe, nicht direkt von `Object` abgeleitet, sondern von der Klasse `ValueType`. Der Compiler und die CLR erkennen einen Werttypen daran, dass dieser mehr oder weniger direkt von `ValueType` abgeleitet ist. Der Compiler legt die Daten eines Objekts, das (mehr oder weniger direkt) von `ValueType` abgeleitet ist, im Stack an (statt im Heap wie bei Refe-

renztypen). Die CLR erzeugt bei der Weitergabe einer Referenz auf ein Werttyp-Objekt implizit eine neue Instanz, kopiert die Daten des Objekts in diese und schreibt die neue Referenz in die Variable, das Feld oder Argument, auf das zugewiesen wurde.

Erzeugen von Werttypen

Werttypen werden implizit erzeugt

Werttypen werden implizit erzeugt, wenn Sie Variablen (Felder etc.) der entsprechenden Wertypen deklarieren. Das folgende Beispiel deklariert eine `int`-Variable und eine Instanz der `PersonStruct`-Struktur:

```
int i;
PersonStruct personStructInstance1;
```

Der Compiler reserviert an der Stelle der Deklaration einen Speicherbereich im Stack, der für die Standardtypen (nicht für Strukturen!) allerdings noch uninitialisiert ist. Beim Kompilieren von Programmen, in denen uninitialisierte Variablen gelesen werden, meldet der Compiler den Fehler »Verwendung der nicht zugewiesenen lokalen Variablen *x*« bzw. »Verwendung des möglicherweise nicht zugewiesenen Feldes *y*«.

Die Initialisierung können Sie bei den Standardtypen direkt bei der Deklaration vornehmen:

```
int i = 0;
```

Strukturen werden automatisch mit Leerwerten initialisiert

Strukturen werden (wie Klassen) vom Compiler bei der Erzeugung automatisch mit Leerwerten initialisiert. Näheres dazu erfahren Sie im Abschnitt »Initialisierung von Objekten bei der Erzeugung« (Seite 161). Strukturen können sich bei der Erzeugung (in ihrem Konstruktor) aber auch selbst initialisieren, was allerdings in der Regel nur dann der Fall ist, wenn bei der Erzeugung Initialisierungswerte übergeben werden.

Sie können die Felder und Eigenschaften einer Struktur aber natürlich auch selbst initialisieren. Eine Möglichkeit dazu ist das Schreiben in die Eigenschaften des Objekts:

```
personStructInstance1.FirstName = "Zaphod";
personStructInstance1.LastName = "Beeblebrox";
```

Die andere (und bessere) Möglichkeit über Objektinitialisierer zu initialisieren zeigt der Abschnitt »Initialisierung von Objekten bei der Erzeugung«.

Der Compiler sichert damit ab, dass Werttypen immer einen definierten Wert speichern.

Schließlich können Werttypen wie Referenztypen auch explizit erzeugt werden:

```
int i = new int();
```

Dieses Feature wird genutzt, wenn der Konstruktor des Werttypen Argumente besitzt und damit eine Initialisierung bei der Erzeugung erlaubt.

Erzeugen von Referenztypen

Referenztypen müssen explizit erzeugt werden

Wenn Sie eine Variable (Feld etc.) eines Referenztyps deklarieren, verweist dieser noch nicht auf eine Instanz des Typs:

```
PersonClass personClassInstance1;
```

Wie bei den Standardtypen (aber anders als bei Strukturen) meldet der Compiler einen Fehler, wenn Sie uninitialisierte Referenztypen im Programm verwenden. Sie können der verwendeten Variablen aber den Wert `null` zuweisen. Dieser Wert steht dafür, dass eine Referenz auf keine Instanz zeigt:

```
PersonClass personClassInstance2 = null;
```

Beim Versuch, mit einer Referenz, die `null` »speichert«, zu arbeiten, generiert jedoch die CLR eine Ausnahme (eine `NullRefererenceException`). Sie können allerdings abfragen, ob ein Referenztyp auf eine Instanz zeigt, indem Sie mit `null` vergleichen:

Listing 3.9: Abfrage darauf, ob eine Referenz null speichert

```
if (personClassInstance2 != null)
{
    Console.WriteLine(personClassInstance2.FirstName);
}
else
{
    Console.WriteLine("Die Referenz personClassInstance2 " +
       "referenziert kein Objekt");
}
```

Möglicherweise werden Sie sich fragen, was das Ganze soll. In der Praxis kommt es aber sehr häufig vor, dass Referenztypen nicht auf eine Instanz zeigen. Gut, wenn Sie dann damit umgehen können ☺.

Referenztypen müssen über new erzeugt werden

Wenn Sie Referenztypen verwenden wollen, müssen Sie diese über das Schlüsselwort `new` erzeugen:

```
personClassInstance1 = new PersonClass();
personClassInstance2 = new PersonClass();
```

Initialisierung von Objekten bei der Erzeugung

Klassen werden wie Strukturen mit Leerwerten initialisiert

Klassen werden (wie Strukturen) vom Compiler bei der Erzeugung automatisch mit Leerwerten initialisiert. Numerische Felder werden auf 0 gesetzt, boolesche Felder auf `false`, Datumsfelder erhalten das minimale Datum (1.1.0001 00:00:00), Strings. Referenztypen und Nullables (Seite 167) erhalten den Wert `null`. Ist der Typ des Feldes eine Struktur, wird diese entsprechend initialisiert.

Konstruktoren erlauben die explizite Initialisierung

Wenn Sie Instanzen von Typen (Wert- oder Referenztypen) explizit erzeugen, können (oder müssen) Sie in den Klammern in vielen Fällen Argumente übergeben, die das Objekt direkt bei der Erzeugung initialisieren. Welche Werte übergeben werden können, legen die Konstruktoren dieser Typen fest. Ein Konstruktor ist eine spezielle Methode, die bei der Erzeugung eines Objekts automatisch aufgerufen wird. Konstruktoren werden in Kapitel 4 behandelt.

Die `DateTime`-Struktur besitzt z. B. (u. a.) einen Konstruktor, dem das Jahr, der Monat und der Tag des gewünschten Datums übergeben werden können:

```
DateTime date = new DateTime(2010, 12, 31);
```

Die meisten Typen besitzen mehrere Konstruktoren. IntelliSense unterstützt Sie bei der Auswahl des Konstruktors, der Ihren Ansprüchen gerecht wird.

NEU

In C# 3.0 können Sie Objekte bei der Erzeugung allerdings auch über Objektinitialisierer initialisieren. Dazu geben Sie statt den runden Klammern geschweifte an. In den Klammern können Sie nun die einzelnen öffentlichen (und nicht schreibgeschützten) Felder und Eigenschaften der Struktur bzw. Klasse angeben und diese mit passenden Werten versehen. Bei der `PersonClass`-Klasse sieht das z. B. so aus:

Listing 3.10: Initialisieren mit Objektinitialisierern

```
PersonClass personClassInstance1 = new PersonClass
{
   FirstName = "Tricia",
   LastName = "McMillan"
};
PersonClass personClassInstance2 = new PersonClass
{
   FirstName = "Ford",
   LastName = "Prefect"
};
```

Objekte können auch über Objektinitialisierer initialisiert werden

Die einzelnen Initialisierungen trennen Sie mit Kommata. Da Sie die Namen der Felder bzw. Eigenschaften angeben, spielt die Reihenfolge keine Rolle. Außerdem müssen Sie nicht alle Felder bzw. Eigenschaften initialisieren, sondern nur die, die Sie initialisieren wollen. Sie sollten aber beachten, dass Felder und Eigenschaften auch schreibgeschützt sein können. Solche Felder und Eigenschaften können Sie dann natürlich nicht initialisieren.

Wenn der verwendete Typ keinen parameterlosen Konstruktor besitzt, müssen Sie wie beim normalen Erzeugen hinter der Typangabe Klammern angeben und in den Klammern die Parameter übergeben, die einer der Konstruktoren des Typs erwartet. Objektinitialisierer können Sie trotzdem verwenden. Nur machen diese dann in der Regel keinen Sinn, weil zumindest einer der Konstruktoren Ihre Initialisierungs-Bedürfnisse abdecken sollte.

EXKURS

Falls Sie interessiert, was dahintersteckt: Wenn Sie Objektinitialisierer verwenden, erzeugt der C#-Compiler daraus CIL-Code, der das Objekt ganz normal instanziert, gefolgt von separaten Initialisierungen der in der Initialisierungsliste verwendeten Eigenschaften und Feldern. Das Erzeugen der ersten Instanz der `PersonClass`-Klasse sieht im erzeugten CIL-Code prinzipiell so aus:

```
PersonClass personClassInstance1 = new PersonClass();
personClassInstance1.FirstName = "Tricia";
personClassInstance1.LastName = "McMillan";
```

Falls Sie weiterhin interessiert, wie ich das herausgefunden habe: Dazu habe ich die erzeugte Assembly zunächst über den Reflector von Lutz Roeder (`www.aisto.com/roeder/dotnet`) wieder in Quellcode zurückgewandelt, wobei ich allerdings in den Optionen eine Optimierung nach C# 2.0 eingestellt hatte, damit die neuen Features nicht berücksichtigt werden. Über den Microsoft CIL-Code-Disassembler (ildasm.exe) habe ich daraufhin überprüft, ob der vom Reflector erzeugte C#-Code dem CIL-Code entspricht.

Zuweisungen an Wert- und Referenztypen

Den Standardtypen und Aufzählungen (deren Basis ein Integer-Typ ist) können Sie passende Werte direkt zuweisen, weil diese nur einen Wert speichern:

```
int i;
i = 10; // Zuweisung des Integer-Literals 10 an die Integer-Variable
```

Strukturen und Klassen sind hingegen strukturierte Typen, die in der Regel mehrere Werte verwalten. Es ist zwar prinzipiell möglich, dass eine Struktur oder Klasse auch die Zuweisung eines einfachen Werts erlaubt, in diesem Fall müssen aber spezielle (Konvertierungs-)Operatoren für die Klasse überschrieben worden sein. Dieses Thema wird in Kapitel 5 behandelt.

Normalerweise müssen Sie bei Instanzen von Strukturen und Klassen deren Felder bzw. Eigenschaften beschreiben, um diese mit (neuen) Werten zu versorgen.

```
PersonStruct personStructInstance1
personStructInstance1.FirstName = "Zaphod";
personStructInstance1.LastName = "Beeblebrox";
```

Sie können einer Variablen (bzw. einem Feld oder einer Eigenschaft) aber nicht nur Literale, sondern auch andere Variablen zuweisen. Dabei verhalten sich Wert- und Referenztypen aber grundlegend anders.

Das Zuweisen von Werttypen kopiert die Werte

Wenn Sie einen Werttypen einem anderen zuweisen, werden die Werte kopiert:

```
int i, j;
i = 10;
j = i;
```

Am Ende dieses Beispiels besitzt *j* den Wert 10, ist aber immer noch ein eigener Speicherbereich. Wenn Sie *j* danach verändern, wird *i* nicht davon beeinflusst:


```
j = 11; // i ist immer noch 10
```

Das funktioniert natürlich auch mit Struktur-Instanzen:

```
PersonStruct personStructInstance3;
personStructInstance3.FirstName = "Marvin";
personStructInstance3.LastName = "The Robot";
PersonStruct personStructInstance4;
personStructInstance4 = personStructInstance3;
```

Das Objekt personStructInstance4 speichert in diesem Fall die gleichen Werte wie das Objekt personStructInstance3. Es handelt sich aber um zwei verschiedene Objekte. Das können Sie sehen, wenn Sie eines der Objekte verändern und deren Daten ausgeben:

```
personStructInstance4.FirstName = "Prostetnik";
personStructInstance4.LastName = "Vogon Jeltz";
Console.WriteLine("Person-Struktur-Instanz 3: " +
   personStructInstance3.FirstName + " " +
   personStructInstance3.LastName);
Console.WriteLine("Person-Struktur-Instanz 4: " +
   personStructInstance4.FirstName + " " +
   personStructInstance4.LastName);
```

Das Beispiel gibt die folgenden Texte an der Konsole aus:

```
Person-Struktur-Instanz 3: Marvin The Robot;
Person-Struktur-Instanz 4: Prostetnik Vogon Jeltz
```

Das Zuweisen von Referenztypen kopiert die Referenz

Referenztypen verhalten sich beim Aufeinander-Zuweisen aber anders. Wenn Sie einen Referenztypen einem anderen Referenztypen zuweisen, werden nicht die Werte, sondern die Referenzen kopiert.

Das folgende Beispiel erzeugt zunächst eine Instanz der `PersonClass`-Klasse und weist diese dann einer weiteren Variablen zu:

```
PersonClass personClassInstance3 = new PersonClass();
personClassInstance3.FirstName = "Marvin";
personClassInstance3.LastName = "The Robot";
PersonClass personClassInstance4;
personClassInstance4 = personClassInstance3;
Console.WriteLine("Person-Klassen-Referenz 3: " +
   personClassInstance3.FirstName + " " + personClassInstance3.LastName);
Console.WriteLine("Person-Klassen-Referenz 4: " +
   personClassInstance4.FirstName + " " + personClassInstance4.LastName);
```

Das Beispiel gibt den folgenden Text an der Konsole aus:

```
Person-Klassen-Referenz 3: Marvin The Robot
Person-Klassen-Referenz 4: Marvin The Robot
```

Dann wird das Objekt, das über personClassInstance4 referenziert wird, geändert:

```
personClassInstance4.FirstName = "Prostetnik";
personClassInstance4.LastName = "Vogon Jeltz";
Console.WriteLine("Person-Klassen-Referenz 3: " +
   personClassInstance3.FirstName + " " + personClassInstance3.LastName);
Console.WriteLine("Person-Klassen-Referenz 4: " +
   personClassInstance4.FirstName + " " + personClassInstance4.LastName);
```

Da beide Referenzen dasselbe Objekt referenzieren, wird an der Konsole nun Folgendes ausgegeben:

```
Person-Klassen-Referenz 3: Prostetnik Vogon Jeltz
Person-Klassen-Referenz 4: Prostetnik Vogon Jeltz
```

INFO

Das Verhalten bei Zuweisungen sollten Sie immer im Auge behalten, wenn Sie mit Wert- und Referenztypen arbeiten. Dies gilt besonders auch deswegen, da es ebenso die Übergabe von Typen an Methoden betrifft: Wird einer Methode ein Werttyp übergeben, erzeugt der Compiler eine Kopie des Objekts. Ändert die Methode die Daten des (kopierten) Objekts, betreffen diese Änderungen nicht das Objekt, das übergeben wurde. Wird hingegen ein Referenztyp übergeben, übergibt der Compiler lediglich eine Referenz auf das Objekt. Änderungen, die die Methode vornimmt, betreffen dann das eine Objekt und sind natürlich nach außen sichtbar.

3.5.3 Standardwerte der verschiedenen Typen

Der C#-Compiler initialisiert Felder von Klassen oder Strukturen immer mit einem Standardwert. Dabei handelt es sich um einen Leerwert: Numerische Felder erhalten den Wert 0, boolesche Felder werden auf `false` gesetzt, Datumsfelder werden mit dem minimalen Datum (1.1.0001 00:00:00) initialisiert, `char`-Instanzen werden auf das 0-Zeichen ('\0') gesetzt, Strings. Referenztypen und Nullables (Seite 167) erhalten den Wert `null`. Ist der Typ des Feldes selbst wieder eine Struktur, wird diese entsprechend initialisiert.

Zum Initialisieren von Variablen können Sie default verwenden

Variablen werden allerdings nicht implizit initialisiert. Variablen müssen Sie immer initialisieren, bevor Sie diese verwenden können. Die Verwendung einer uninitialisierten Variablen lässt der Compiler nicht zu. Zum Initialisieren verwenden Sie normalerweise einen zur Variablen passenden Wert. Sie können aber auch das `default`-Schlüsselwort verwenden um eine Variable (oder ein Feld) mit dem Standardwert des Typs zu initialisieren. Diesem Schlüsselwort übergeben Sie den Typ in Klammern:

```
int i = default(int);
```

3.5.4 Generische Typen

Der Typ der verwalteten Daten wird bei generischen Typen erst bei der Verwendung festgelegt

Das .NET Framework enthält eine große Anzahl an so genannten generischen Typen. Generische Typen sind zunächst einmal Typen, die andere Objekte verwalten oder verwenden. Anders als bei »normalen« Typen ist der Typ der verwalteten bzw. verwendeten Objekte aber nicht festgelegt. Erst bei der Verwendung eines generischen Typs in einem Programm legt der Programmierer den Typ der verwalteten Daten fest, indem er diesen in spitzen Klammern angibt.

Ein sehr gutes Beispiel für den sinnvollen Einsatz generischer Typen ist eine Auflistung. Eine Auflistung verwaltet eine Liste von Objekten. So könnten Sie z. B. Instan-

zen einer eigenen Person-Klasse in einem Programm aus den Daten einer XML-Datei ermitteln und in einer Auflistung verwalten um diese weiterverarbeiten zu können.

Würden keine generischen Typen existieren, könnten Sie dazu eine der alten (.NET-1.0-)Auflistungen verwenden, die Referenzen vom Typ object verwalten. Da eine Referenz vom Typ object alle möglichen Objekte referenzieren kann, wäre auch die Verwaltung von Address-Objekten möglich.

Der massive Nachteil dieser alten Auflistungen ist aber, dass Sie beim Lesen zum einen die referenzierten Objekte über eine Typumwandlung in den erwarteten Typ (in unserem Beispiel in Person-Instanzen) umwandeln müssen. Zum anderen können Sie beim Lesen nie sicher sein, dass wirklich nur Instanzen der erwarteten Klasse in der Auflistung gespeichert sind (glauben Sie mir: In komplexen Projekten passieren so einige Unfälle ☺).

Eine andere Möglichkeit wäre die eigene Implementierung einer typsicheren Auflistung, die nur Person-Objekte aufnehmen kann. Der Nachteil hier wäre, dass dies eine Menge Arbeit macht.

Glücklicherweise bietet .NET ab der Version 2.0 eine Vielzahl an generischen Auflistungen (die in Kapitel 7 noch genauer besprochen werden). Die List-Klasse aus dem Namensraum System.Collections.Generic ist z. B. eine Standard-Auflistung, die das Durchgehen über einen Integer-Index erlaubt. Bei der Deklaration und der Erzeugung einer List-Instanz müssen Sie (wie bei allen generischen Typen) den Typ, den die Auflistung verwalten soll, in spitzen Klammern angeben. Eine typsichere Auflistung von Person-Objekten erhalten Sie also z. B. so:

```
List<Person> personList = new List<Person>();
```

Dieser Auflistung können Sie nun nur Person-Instanzen hinzufügen und beim Lesen erhalten Sie immer auch Person-Referenzen zurück:

```
Person person = new Person();
person.FirstName = "Zaphod";
person.LastName = "Beeblebrox";
...
personList.Add(person);
...
for (int i = 0; i < personList.Count; i++)
{
   Console.WriteLine(personList[i].FirstName + " " +
      personList[i].LastName);
}
```

Wenn Sie generische Typen verwenden, zeigt IntelliSense zunächst über die spitzen Klammern am Namen des Typs an, dass es sich um einen generischen Typ handelt (Abbildung 3.4).

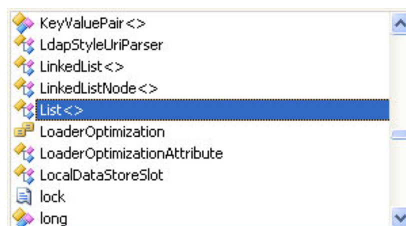
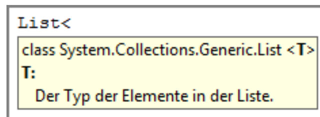


Abbildung 3.4: IntelliSense zeigt den Namen der List-Klasse an

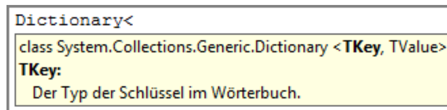
Wenn Sie den Namen eines generischen Typs und eine öffnende spitze Klammer zur Angabe des oder der verwalteten Typen schreiben, zeigt IntelliSense die erwarteten Typen an und deren Beschreibung (Abbildung 3.5).

Abbildung 3.5: IntelliSense zeigt Informationen zu Typparametern an



Generische Typen können beliebig viele Typparameter besitzen. Mehrere Typparameter werden durch Kommata voneinander getrennt. Ein Beispiel dafür ist die `Dictionary`-Klasse (Abbildung 3.6)

Abbildung 3.6: IntelliSense für die Dictionary-Klasse



EXKURS

Ein `Dictionary` ist eine so genannte assoziative Auflistung. In einer solchen werden die aufgelisteten Objekte mit einem Schlüssel assoziiert. Über den Schlüssel können Sie später auf die Objekte zugreifen. Ein Beispiel für den sinnvollen Einsatz einer `Dictionary`-Auflistung wäre das Speichern von Kundendaten, wobei die Kundennummer als Schlüssel verwendet wird. In Kapitel 7 erfahren Sie mehr über diese Klasse.

`Dictionary` erwartet zwei Typparameter: *TKey* und *TValue*. *TKey* ist der Typ des Schlüssels, *TValue* der Typ der verwalteten Objekte. Der Name der Typparameter ist übrigens beim eigenen Programmieren generischer Typen frei vergebbar. Die .NET-Klassen benennen Typparameter üblicherweise mit einem einfachen »T« (für »Type«), wenn die Klasse nur einen Typparameter besitzt, und mit einem Namen, der mit »T« beginnt, wenn die Klasse mehrere Typparameter besitzt.

Wenn Sie eine `Dictionary`-Instanz erzeugen, müssen Sie also zwei Typparameter angeben. Für das Kunden-Beispiel mit einer String-Kundennummer wäre das:

```
Dictionary<string, Customer> customerList =
   new Dictionary<string, Customer>();
```

Einige generische Typen schränken die für die Typparameter verwendbaren Typen auch ein. So kann es z. B. sein, dass ein Typ eine bestimmte Schnittstelle implementieren muss. Der Compiler beschwert sich in diesem Fall, wenn Sie einen Typen einsetzen, der den Einschränkungen nicht entspricht. Über dieses (etwas komplexere) Thema erfahren Sie mehr in Kapitel 6, bei der Programmierung eigener generischer Typen.

Generische Typen machen das Programmierer-Leben damit in vielen Fällen um einiges leichter. Kapitel 7 setzt sich noch näher mit den generischen Auflistungen auseinander, Kapitel 6 behandelt die Programmierung eigener generischer Typen.

3.5.5 Nullables: Werttypen mit der Möglichkeit nichts (null) zu speichern

Normalen Werttypen müssen Sie immer einen zum Typ passenden Wert zuweisen, was in manchen Fällen ein Nachteil ist. Wenn Sie z. B. in einer Instanz der DateTime-Struktur ein Datum verwalten, aber ermöglichen wollen, dass auch gespeichert werden kann, dass *kein* Datum angegeben ist, haben Sie ein Problem. Dieses Problem können Sie allerdings mit *Nullable Types* (Nullables) lösen.

Nullables sind Werttypen, die null speichern können

Nullables sind spezielle Werttypen, denen Sie neben dem zum Typ passenden Wert auch null zuweisen können. Nullables basieren auf der speziellen generischen Struktur System.Nullable, die die Zuweisung, den Vergleich und arithmetische Operationen mit Standardtypen ermöglicht.

Das folgende Beispiel deklariert einen Nullable-Integer, eine Nullable-DateTime- und eine Nullable-PersonStruct-Instanz (aus dem Abschnitt »Wert- und Referenztypen« ab Seite 155):

```
Nullable<int> i1;
Nullable<DateTime> date1;
Nullable<PersonStruct> person1;
```

C# ermöglicht eine verkürzte Schreibweise, indem Sie dem Typen ein Fragezeichen anhängen:

```
int? i2 = null;
DateTime? date2 = null;
PersonStruct? person2;
```

Diese spezielle Syntax wird vom C#-Compiler einfach in die eigentliche Deklaration mit der Nullable-Struktur umgesetzt.

Mit Nullables, die auf Standardtypen basieren, können Sie nun prinzipiell so arbeiten wie mit normalen Typen. Der Compiler unterstützt Sie dabei, da er bei allen Typen eine Initialisierung erzwingt. Sie können also gar keine uninitialisierten Nullables verwenden.

```
i1 = 10;
i2 = i1 * 10;
Console.WriteLine("i2: " + i2);

date1 = DateTime.Now;
Console.WriteLine(date1);
```

Sie können aber jetzt auch null zuweisen:

```
i1 = null;
date1 = null;
person1 = null;
```

Sie können (und sollten) vor der Verwendung von Nullable-Objekten nachfragen, ob diese null speichern. Dazu können Sie einfach mit null vergleichen:

```
if (date1 != null)
{
   Console.WriteLine("Datum: " + date1);
}
else
{
   Console.WriteLine("Kein Datum angegeben");
}
```

Alternativ dazu können Sie die `HasValue`-Eigenschaft der `Nullable`-Struktur abfragen:

```
if (date1.HasValue)
{
   Console.WriteLine("Datum: " + date1);
}
else
{
   Console.WriteLine("Kein Datum angegeben");
}
```

Sie können alle Werttypen (alle Standardtypen, Aufzählungen und Strukturen) mit der `Nullable`-Struktur einsetzen (auch eigene). Der Zugriff auf die Daten des Objekts kann für Standardtypen und Aufzählungen über den Namen der entsprechenden Variablen erfolgen. Dabei sollten Sie jedoch beachten, dass Ausdrücke, die Nullables enthalten, immer einen `Nullable` zurückgeben. Darauf gehe ich noch ein.

Nullables und Strukturen und die Value-Eigenschaft

`Nullable`-Strukturen können (in der Regel) nicht so einfach verwendet werden wie Standardtypen und Aufzählungen. Das liegt daran, dass diese Felder und/oder Eigenschaften besitzen. Ein Zugriff wie der folgende:

```
Console.WriteLine(person1.FirstName);
```

ist nicht möglich. Das erkennen Sie schon daran, dass IntelliSense die Felder und Eigenschaften der Struktur gar nicht anbietet. Das ist auch korrekt so, denn die Variable, mit der Sie arbeiten, ist nicht vom Typ des eigentlich gespeicherten Objekts, sondern vom Typ der `Nullable`-Struktur. Und die besitzt nur einige spezielle Eigenschaften.

Value speichert den eigentlichen Wert

Über die Eigenschaft `Value` der `Nullable`-Struktur erhalten Sie allerdings Zugriff auf das eigentlich gespeicherte Objekt. Damit können Sie die Daten des Objekts lesen:

```
Console.WriteLine(person1.Value.FirstName);
```

Speichert die `Nullable`-Variable, mit der Sie arbeiten, allerdings `null`, resultiert in der Laufzeit eine Ausnahme vom Typ `InvalidOperationException` mit der Meldung »Das Objekt mit Nullwert muss einen Wert haben«. Sie müssen also vor dem Zugriff immer abfragen, ob die `Nullable`-Variable einen Wert besitzt:

Listing 3.11: Abfrage darauf, ob ein Nullable einen Wert besitzt

```
if (person1.HasValue) // oder if (person1 != null)
{
   Console.WriteLine(person1.Value.FirstName);
}
else
{
   Console.WriteLine("person1 ist null");
}
```

Der schreibende Zugriff ist nur möglich, wenn es sich um einen Standardtypen oder eine Aufzählung handelt oder wenn eine Struktur Konvertierungsoperatoren besitzt, die einfache Werte in Instanzen der Struktur umwandeln. Beim Versuch, in eine Eigenschaft oder ein Feld einer Struktur zu schreiben, meldet der Compiler den Fehler, dass der Rückgabewert von `Value` nicht modifiziert werden kann, weil es keine Variable ist:

```
person1.Value.FirstName = "Zaphod"; // Compilerfehler »Der Rückgabewert
   // "System.Nullable<Nullables.PersonStruct>.Value" kann nicht geändert
   // werden, da er keine Variable ist«
```

Prinzipiell würde aber die folgende Anweisung funktionieren, wenn die `PersonStruct`-Struktur einen Konvertierungsoperator zur Verfügung stellen würde, der Strings in `PersonStruct`-Instanzen konvertiert:

```
person1.Value = "Zaphod Beeblebrox";
```

Ansonsten können Sie eine `Nullable`-Strukturvariable nur mit einer neuen Instanz der Struktur beschreiben. Da eine nachträgliche Änderung der Daten nicht mehr möglich ist, müssen Sie das Objekt bei der Erzeugung initialisieren. Dazu können Sie einen ggf. vorhandenen Konstruktor verwenden oder einen Objektinitialisierer:

```
person1 = new PersonStruct { FirstName = "Zaphod",
   LastName = "Beebebrox" };
```

Nullables in Ausdrücken

Sie können Nullables ohne Probleme in Ausdrücken verwenden, sofern die verwendeten Basistypen die verwendeten Operatoren anbieten. Die Standardtypen überschreiben z. B. alle arithmetischen Operatoren. Deswegen können Sie mit diesen Typen rechnen, auch dann, wenn es sich um Nullables handelt:

Listing 3.12: Rechnen mit Nullables

```
int? i3 = 1;
int? j3 = 2;
double? result1 = (i3 + j3) * 0.5;
Console.WriteLine(result1); // 1,5
```

Nullable-Ausdrücke ergeben Nullable-Werte

Das Ergebnis eines Ausdrucks, der Nullables enthält, ist immer auch ein Nullable. Deswegen weist das Beispiel das Ergebnis einer `Nullable-double`-Variablen zu. Die Zuweisung an eine normale Variable würde der Compiler nicht zulassen.

Der Grund dafür ist, dass ein Ausdruck mit `Nullable`-Werten auch `null` ergeben kann. Das ist dann der Fall, wenn mindestens einer der Operanden `null` speichert:

```
int? i4 = null;
int? j4 = 2;
double? result2 = (i4 + j4) * 0.5;
Console.WriteLine(result2); // result2 ist null
```

Speichert einer der Operanden `null`*, ist das Ergebnis eines Ausdrucks mit Nullables immer ebenfalls* `null`.

Wollen Sie allerdings mit den Basistypen rechnen oder das Ergebnis einem solchen zuweisen, müssen Sie die Nullables in normale Werttypen konvertieren. Das erledigen Sie idealerweise, indem Sie vor der Berechnung abfragen, ob die Operanden nicht null speichern. In der Berechnung können Sie dann über die `Value`-Eigenschaft auf die Werte zugreifen:

Listing 3.13: Abfragen der HasValue-Eigenschaft um zu ermitteln, ob ein Nullable einen Wert verwaltet

```
int? i3 = null;
int? j3 = 2;
if (i3.HasValue && j3.HasValue)
{
   double result3 = (i3.Value + j3.Value) * 0.5;
   Console.WriteLine(result3);
}
```

```
else
{
   Console.WriteLine("Mindestens einer der Operanden ist null");
}
```

3.5.6 Freigeben von Objekten

Irgendwann einmal muss jedes Objekt (leider) sterben. Ok, ich glaube an Wiedergeburt, aber das gehört hier wohl nicht hin …

Spätestens dann, wenn ein Programm beendet ist, werden alle Speicherbereiche freigegeben, die das Programm reserviert hat. Darum müssen Sie sich also nicht kümmern.

Während der Laufzeit einer Anwendung werden aber immer wieder Instanzen erzeugt, die später nicht mehr benötigt werden. Würden diese nicht freigegeben werden, würden .NET-Programme immer mehr Speicher benötigen und ggf. (mit einer `OutOfMemoryException`) abstürzen.

Wie lange ein Objekt benötigt wird, hängt davon ab, wie lange es referenziert wird. Wenn Sie in einer Methode Objekte erzeugen und nur dort (über lokale Variablen) referenzieren, werden diese freigegeben, sobald die Methode beendet ist. Handelt es sich um Werttypen, werden diese sofort aus dem Speicher »entfernt«, weil das Programm den Teil des Stacks, den die Methode reserviert hat, wieder freigibt. Referenztypen bleiben allerdings noch ein wenig im Heap.

Wird ein Objekt allerdings nicht nur in einer Methode referenziert, sondern global (über eine statische Eigenschaft einer Klasse oder über ein Objekt, das selbst global verwaltet wird), wird das Objekt erst dann freigegeben, wenn es nicht mehr referenziert wird. bzw. wenn das Objekt, das das andere Objekt beinhaltet, freigegeben wird.

Der GC räumt Objekte auf

Das Aufräumen von Objekten auf dem Heap, die nicht mehr benötigt werden, übernimmt der Garbage Collector. Dieser geht immer dann, wenn eine Anwendung nicht besonders beschäftigt ist, durch den Heap, sucht »verwaiste« Objekte und gibt deren Speicher frei[3]. Die Freigabe geschieht allerdings nach einem komplexen Muster, das sicherstellt, dass die Performance der Anwendung nicht leidet.

Für Sie bedeutet dies, dass Sie nichts weiter tun müssen, als die Referenzen auf ein Objekt freizugeben bzw. sich darauf zu verlassen, dass diese automatisch freigegeben werden, wenn die Objektvariable ihren Gültigkeitsbereich verlässt. Wenn Sie Objekte selbst freigeben wollen, setzen Sie die Objektreferenz auf `null`:

```
p = null;
```

Dieses explizite »Freigeben« wird allerdings nur in sehr seltenen Fällen sinnvoll sein. Wenn Sie z. B. in einer aufwändigen Methode Objekte einsetzen und nach deren Verwendung Arbeitsspeicher sparen wollten, während die Methode noch weiter ausgeführt wird, könnten Sie auf die Idee kommen, die Objektreferenzen von Hand freizugeben. Dummerweise werden diese aber erst aus dem Arbeitsspeicher entfernt, wenn der Garbage Collector dafür Zeit hat. Und das ist mit Sicherheit nicht mitten in der Ausführung einer Methode.

3 Der Garbage Collector wird dazu in einem Thread ausgeführt, der parallel zum Hauptthread der Anwendung ausgeführt wird.

Der Garbace Collector macht zwar, wie Sie sehen, seine Arbeit. Das Problem ist aber, dass die Freigabe von Objekten zu einem unbestimmten Zeitpunkt geschieht. Deswegen ist zum Freigeben von externen Ressourcen, die Objekte verwenden, unter .NET ein Modell vorgesehen, das auf der `IDisposable`*-Schnittstelle beruht. Typen, die diesem Modell entsprechen, bieten eine Methode an, die* `Dispose` *heißt. Diese Methode sollte – wenn vorhanden – immer aufgerufen werden, wenn ein Objekt nicht mehr benötigt wird. In Kapitel 4 gehe ich im Abschnitt »Konstruktoren, Finalisierer, Dispose und using« noch näher darauf ein.*

3.5.7 Der Typ String als Ausnahme

String-Instanzen können auch ohne new erzeugt werden

Der Typ `string`, der Zeichenketten verwaltet, ist eigentlich auch ein Referenztyp. Der Compiler erlaubt für diesen Typ aber auch eine Erzeugung ohne `new`:

```
string s1 = "Das ist ein String";
```

Alternativ können Sie Strings auch mit `new` erzeugen. Dem Konstruktor können Sie dazu verschiedene Argumente übergeben. U. a. können Sie den String mit einer bestimmten Anzahl von Zeichen initialisieren:

```
string s2 = new string('*', 1024);
```

Bei Zuweisungen eines Strings auf einen anderen verhält sich ein String ebenfalls etwas anders als ein normaler Referenztyp: Wenn Sie zwei Stringvariablen oder Eigenschaften einander zuweisen, kopiert der Compiler zunächst die Referenz.

```
s1 = "Zaphod";
s2 = s1;
```

Nach der Ausführung dieser Anweisungen referenziert *s2* dieselbe Zeichenkette wie *s1*. So weit stimmen Strings noch mit normalen Referenztypen überein. Da die Zuweisung eines String-Literals aber immer zur Erzeugung eines neuen String-Objekts führt, führt die »Änderung« eines Strings nicht dazu, dass auch Strings, die über andere Variablen referenziert werden, geändert werden:

```
s2 = "Ford";
```

Nun referenziert *s2* einen anderen String als *s1*. Eine »Änderung« des Strings in *s2* bewirkt keine Änderung des Strings in *s1*.

Strings sind unveränderbar

Strings sind in Wirklichkeit unveränderbar (immutable). Wenn Sie einen String neu beschreiben, führt das immer dazu, dass eine neue Instanz erzeugt und die alte freigegeben wird. Die `String`-Klasse sorgt damit dafür, dass Strings ähnlich wie Werttypen behandelt werden können, obwohl es sich um Referenztypen handelt.

3.5.8 Übersicht über die Standardtypen

Jede Programmiersprache besitzt Standardtypen zur Speicherung von numerischen Werten, Zeichenketten, Datumswerten und anderen häufig benötigten Daten. Auch C# besitzt solche Typen. Allerdings handelt es sich dabei nicht um C#-eigene Typen, sondern um Typen, die die CLR zur Verfügung stellt und die deswegen in allen .NET-Sprachen zur Verfügung stehen. Diese Typen gehören zum Namensraum `System` und heißen z. B. `Int32`. In C# können Sie für diese Typen aber auch Aliasnamen verwenden. `System.Int32` entspricht z. B. dem Alias `int`.

Tabelle 3.3 beschreibt die Standardtypen, inklusive des für API-Aufrufe wichtigen Typs `IntPtr`, für den es keinen Aliasnamen gibt.

Beachten Sie, dass die vorzeichenlosen Integer-Typen nicht CTS-kompatibel sind. Wenn Sie Komponenten entwickeln, die von anderen .NET-Programmiersprachen verwendet werden sollen, sollten Sie diese Typen nur für interne, private Zwecke verwenden.

Tabelle 3.3: Die C#-Standardtypen

C#-Datentyp	CLR-Datentyp	Größe	Wertebereich
sbyte	SByte	8 Bit	-128 bis 127
byte	Byte	8 Bit	0 bis 255
char	Char	16 Bit	ein beliebiges Unicode-Zeichen (Unicode wird in Kapitel 1 beschrieben)
short	Int16	16 Bit	-32.768 bis 32.767
ushort	UInt16	16 Bit	0 bis 65.535
int	Int32	32 Bit	-2.147.483.648 bis 2.147.483.647
uint	UInt32	32 Bit	0 bis 4.294.967.295
long	Int64	64 Bit	-9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807
ulong	UInt64	64 Bit	0 bis 18.446.744.073.709.551.615
float	Single	32 Bit	$\pm 1{,}5 \times 10^{-45}$ bis $\pm 3{,}4 \times 10^{38}$ mit 7 bis 8 Ziffern Genauigkeit
double	Double	64 Bit	$\pm 5{,}0 \times 10^{-324}$ bis $\pm 1{,}7 \times 10^{308}$ mit 15 bis 16 Ziffern Genauigkeit
decimal	Decimal	128 Bit	$1{,}0 \times 10^{-28}$ bis $7{,}9 \times 10^{28}$ mit maximal 28-29 Dezimalstellen
bool	Boolean	8 Bit	true oder false
string	String	variabel	beliebige Unicode-Zeichenketten
	IntPtr	32 Bit	mit int kompatibler Zeiger-Typ, der als Argument von Windows-API-Funktionen verwendet wird

3.5.9 Instanzmethoden der Standardtypen

Die Standardtypen besitzen wie alle Typen die Instanzmethoden, die sie von Object geerbt haben (siehe Seite 155). Der ToString-Methode können Sie bei numerischen Typen allerdings auch einen Format-String oder einen Format-Provider übergeben, um die gespeichert Zahl formatiert auszugeben. Formatierungen werden in Kapitel 8 behandelt. Ich zeige hier nur kurz, wie Sie z. B. einen double-Wert auf zwei Stellen hinter dem Komma formatieren können:

```
double d = 1.2345;
Console.WriteLine(d.ToString("0.00"));
```

Neben den geerbten Methoden besitzen die Standardtypen noch die in Tabelle 3.4 dargestellten zusätzlichen Instanzmethoden.

Methode	Beschreibung
int CompareTo(*Typ value*)	vergleicht ein Objekt mit einem anderen. Ist die Rückgabe kleiner 0, ist das Objekt kleiner als das andere. Wird 0 zurückgegeben, sind beide Objekte gleich. Bei einer Rückgabe größer 0 ist das Objekt größer als das andere.
TypeCode GetTypeCode()	ermittelt den Typ des Objekts als Wert der Aufzählung TypeCode. TypeCode enthält Konstanten, die so benannt sind wie die Typen. TypeCode.Byte steht z. B. für einen Byte-Typ.

Tabelle 3.4: Die zusätzlichen Methoden der Standardtypen

Die CompareTo-Methode wird hauptsächlich implizit verwendet, wenn ein Typ in einer Auflistung gespeichert ist, die sortierbar ist. Explizit brauchen Sie diese Methode eigentlich nie aufrufen, da Sie für Vergleiche Vergleichsoperatoren verwenden können.

Der Typ String besitzt noch eine Vielzahl weiterer Methoden. Ich beschreibe die wichtigsten davon separat in Kapitel 8.

3.5.10 Klassenmethoden und -eigenschaften der Standardtypen

Alle Standardtypen besitzen neben den Instanzmethoden auch noch Klassenmethoden und -eigenschaften, die Sie ohne Instanz der Klasse verwenden können. Die MaxValue-Eigenschaft der numerischen Typen gibt z. B. den größten speicherbaren Wert zurück:

```
Console.WriteLine("double kann maximal " +
   double.MaxValue + " speichern.");
```

Über die Parse-Methode können Sie einen String in den entsprechenden Typ umwandeln:

```
d = double.Parse("1.234");
```

Tabelle 3.5 zeigt die wichtigsten dieser Eigenschaften und Methoden.

Eigenschaft/Methode	Beschreibung
MinValue	liefert den kleinsten speicherbaren Wert.
MaxValue	liefert den größten speicherbaren Wert.
Typ Parse(string *s* [...])	Über diese Methode können Sie einen String in den Typ umwandeln, auf dem Sie die Methode anwenden. Parse erlaubt bei numerischen Typen zusätzlich die Angabe der zu verwendenden Kultur. Kulturen werden in Kapitel 15 behandelt.
bool TryParse(string *s*, NumberStyles *style*, IFormatProvider *provider*, out double *result*)	Über diese komplexe Methode können Sie überprüfen, ob eine Umwandlung eines Strings in einen speziellen Typ möglich ist.

Tabelle 3.5: Die wichtigsten Klasseneigenschaften und -methoden der Standardtypen

Die TryParse-Methode will ich hier nicht näher beschreiben, weil Sie dazu wissen müssen, wie Sie mit Schnittstellen umgehen (IFormatProvider) und was Kultur-Informationen sind. Schnittstellen werden in Kapitel 5 behandelt, Kultur-Informa-

tionen (Globalisierung) in Kapitel 15. Das folgende Beispiel überprüft einen String darauf, ob der mit den aktuellen Ländereinstellungen in einen `double`-Wert konvertierbar ist:

Listing 3.14: Überprüfung eines Strings darauf, ob dieser in einen double-Wert konvertiert werden kann

```
string input = "abc";
double result;
if (double.TryParse(input, out result) == false)
{
   Console.WriteLine(input + " kann nicht in einen " +
      "double-Wert umgewandelt werden.");
}
```

3.5.11 Integer-Typen

Integer-Typen speichern Ganzzahlen

C# unterscheidet einige Typen für die Speicherung von Integer-Werten (Werte ohne Dezimalanteil): `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long` und `ulong`. Bei der Auswahl dieser Typen müssen Sie eigentlich nur beachten, dass der Datentyp groß genug ist für Ihre Anwendung und ob Sie ein Vorzeichen benötigen. Machen Sie sich aber nicht zu viel Gedanken: Wenn Sie für einzelne Daten einen zu großen Datentyp verwenden, macht das dem Programm nichts aus. Der Arbeitsspeicher ist heute ja wohl immer groß genug. Verwenden Sie lieber einen größeren Typ, damit Sie später bei der Speicherung von Daten keine Probleme haben. Wenn Sie z. B. für die Speicherung einer Kundennummer `ushort` verwenden, und die Nummer wird größer als 65.535, gibt es Probleme (siehe unter »Über- und Unterläufe und spezielle Werte« auf Seite 177). Achten Sie aber auf die Größe, wenn Sie sehr viele Daten speichern (z. B. in einem Array).

Die gängigen Literale für Integer-Werte sind dezimale Zahlwerte:

```
int number1 = 1234;
```

Alternativ können Sie auch die hexadezimale Schreibweise verwenden. Stellen Sie dazu 0x vor den Wert:

```
byte number2 = 0xff; // = 255
```

Wenn Sie eine Zahl ohne Dezimalstelle schreiben, erkennt der Compiler diese als `int`, `uint`, `long` oder `ulong`, je nachdem, ob der Wert in den Typ passt:

```
ulong number3 = 10; // die Zahl 10 ist vom Typ int
ulong number4 = 4294967295; // die Zahl 4294967295 ist vom Typ uint
```

Ausprobieren können Sie dies, indem Sie den Typ eines Literals ermitteln:

```
Console.WriteLine("Das Literal 4294967295 besitzt " +
   "den Typ '" + 4294967295.GetType().Name + "'");
```

Sie können ein Integer-Literal aber auch einer kleineren oder größeren Variablen zuweisen:

```
byte number5 = 10; // Zuweisung des int-Werts 10 auf byte
long number6 = 10; // Zuweisung des int-Werts 10 auf long
```

Wenn Sie versuchen, einem Datentyp ein zu großes oder nicht konvertierbares Literal zuzuweisen, meldet der Compiler einen Fehler:

```
byte number7;
number7 = 1024; // Compilerfehler "Konstantenwert '1024' kann
                // nicht nach 'byte' konvertiert werden"
number7 = 10.5; // Compilerfehler "Implizite Konvertierung
                // des Typs 'double' zu 'byte' nicht möglich"
```

Konvertierungen werden ab Seite 188 behandelt.

Sie können einem Literal einen Suffix anhängen um damit den Datentyp zu bestimmen. Groß- und Kleinschreibung spielt dabei keine Rolle.

Tabelle 3.6: Die Suffixe für Integer-Literale

Suffix	Datentyp
`l`	`long`
`u`	`uint` oder `ulong`, je nach Größe des Werts
`ul`	`ulong`

3.5.12 Fließkommatypen und der Typ decimal

Fließkommatypen (`float` und `double`) und der Typ `decimal` speichern Dezimalzahlen mit einer festgelegten maximalen Anzahl an Stellen (unabhängig vom Komma).

Fließkommatypen besitzen eine eingeschränkte Genauigkeit

Der Typ `float` besitzt eine Genauigkeit von sieben Stellen. Dies bedeutet, dass maximal sieben Stellen genau gespeichert werden (die restlichen werden ungenau gespeichert). Handelt es sich um eine Dezimalzahl und steht eine 0 vor dem Komma, wird die Angabe der Genauigkeit allerdings um diese Null erhöht, beträgt dann also acht Stellen.

`double` besitzt eine Genauigkeit von 15 bis 16 Stellen (16 wieder dann, wenn eine Null vor dem Komma steht). `decimal` besitzt eine Genauigkeit von 29 Stellen. Die Genauigkeit und die Probleme, die daraus entstehen, behandle ich gleich noch.

Als Literal für einen Fließkommatypen können Sie eine Zahl in der englischen Schreibweise verwenden. Der Compiler wertet Zahlen mit Dezimalstellen grundsätzlich als `double` aus:

```
double number1 = 1.234;
```

Wenn Sie einer `float`-Variablen ein Literal zuweisen wollen, das einen Nachkommaanteil besitzt, müssen Sie dieses explizit als `float` kennzeichnen. Hängen Sie dazu das Suffix *f* an:

```
float number2 = 1.234f;
```

Integer-Werte können Sie allerdings ohne Konvertierung zuweisen:

```
float number3 = 1;
```

Dasselbe gilt, wenn Sie einem `decimal`-Typ ein Literal zuweisen wollen. Verwenden Sie zur Konvertierung den Suffix *m* (von »Money«):

```
// Decimal erfordert eine Konvertierung bei Dezimal-Literalen ...
decimal number4 = 1.234m;

// ... aber nicht bei Integer-Literalen
decimal number5 = 1234;
```

Für Dezimalzahlen können Sie wie bei Ganzzahlen die wissenschaftliche Schreibweise verwenden:

```
double number6 = 5E-2; // 0.05
```

`5E-2` steht im Beispiel für: $5 * 10^{-2}$.

Für die Festlegung des Typs können Sie eines der in Tabelle 3.7 dargestellten Suffixe verwenden.

Tabelle 3.7: Suffixe für die Festlegung des Datentyps eines Fließkomma-Literals

Suffix	Datentyp
d	Double
f	Float
m	decimal

Die (Un)Genauigkeit der Fließkommatypen und von decimal

Ein `float`-Wert besitzt eine Genauigkeit von sieben bis acht Stellen. Unabhängig davon, an welcher Stelle das Komma steht, sind die Ziffern ab der Position 9 (mit 0 vor dem Komma) bzw. 8 also ungenau. Die Zahl 1234567*890* wird ebenso ungenau gespeichert wie die Zahl 1,234567*890*. Bei der ersten Zahl resultiert 1234567*936*, bei der zweiten 1,234567*8806304932*.

Zum Beweis der Ungenauigkeit speichert das folgende Beispiel zunächst einen möglichst genauen Wert für PI in einer `float`-Variablen. Dieser Wert wird dann an der Konsole ausgegeben. Schließlich wird der Inhalt der `float`-Variablen in eine `double`-Variable geschrieben und noch einmal ausgegeben:

Listing 3.15: Beweis der Ungenauigkeit von float

```
float pi1 = 3.14159265358979323846264338327950F;
Console.WriteLine(pi1); // 3,141593

double pi2 = pi1;
Console.WriteLine(pi2); // 3,1415927410125732
```

Wie Sie an dem Beispiel erkennen, speichert die `float`-Variable den ungenauen Wert 3,141592*7410125732*. Wenn Sie Fließkommawerte ausgeben, gibt die implizit aufgerufene `ToString`-Methode den Wert allerdings auf die signifikanten Stellen gerundet aus. Deswegen erscheint an der Konsole der Wert 3,141593. Das Beispiel beweist dann aber, dass die `float`-Variable tatsächlich einen ungenauen Wert speichert, nämlich in etwa den, der in die `double`-Variable geschrieben und von dort aus mit einer größeren Genauigkeit ausgegeben wird.

Als letzten Beweis der Ungenauigkeit schreibt das Beispiel den PI-Wert in eine `double`-Variable, gibt diesen Wert aus und multipliziert schließlich den `float`-Wert mit 2 um zu zeigen, dass tatsächlich der angegebene Wert gespeichert ist:

Listing 3.16: Letzter Beweis der Ungenauigkeit von float

```
double pi3 = 3.14159265358979323846264338327950;
Console.WriteLine(pi3); // 3,1415926535897931

double test = pi1 * 2;
Console.WriteLine(test); // 6,2831854820251465
```

Der beim Multiplizieren resultierende Wert 6,2831854820251465 entspricht ziemlich genau dem Zweifachen des Werts 3,141592*7410125732* (und nicht des Werts 3,141593, den `ToString` der `float`-Variablen suggeriert), was endgültig beweist, dass das Rechnen mit Fließkommazahlen bei Zahlen mit mehr als sieben bzw. 15 Ziffern ungenau ist.

Ungenauigkeiten treten aber häufig schon mit Zahlen auf, die eigentlich noch in den speicherbaren Bereich fallen:

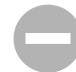
```
float price1 = 4.99F;
int quantity = 17;
float total1 = price1 * quantity;
Console.WriteLine("Gesamtpreis: " + total1); // 84,8299961090087890
```

Obwohl 4,99 * 17 eigentlich 84,83 ergibt, resultiert der Wert 84,8299961090087890. Erst, wenn Sie `double` zum Speichern verwenden, ist das Ergebnis genau:

```
double price2 = 4.99;
double total2 = price2 * quantity;
Console.WriteLine("Gesamtpreis: " + total2); // 84,83
```

REF

Die Ungenauigkeit von Fließkommatypen basiert auf der (komplexen) Technik, mit der sie gespeichert werden. Diese Technik wird im IEEE-Standard 754 beschrieben (`de.wikipedia.org/wiki/IEEE_754`*,* `754r.ucbtest.org/standards/754.pdf`*).*


HALT

`float` *ist wegen seiner Genauigkeit von lediglich sieben Stellen normalerweise nicht besonders zur Speicherung von Zahlen geeignet. Verwenden Sie lieber* `double`*, denn dieser Datentyp besitzt eine Genauigkeit von 15 bis 16 Stellen. Um eine noch höhere Genauigkeit zu erreichen können Sie auch mit dem Festkommatyp* `decimal` *arbeiten, der eine Genauigkeit von 29 Stellen besitzt.*

decimal besitzt immer 29 Stellen

`decimal` ist ein spezieller Datentyp mit einer festgelegten Anzahl von Ziffern (29). Damit können Sie sich also recht schnell ausrechnen, wie viele Dezimalstellen dieser Datentyp speichern kann, wenn eine bestimmte Anzahl Ziffern vor dem Komma verwaltet wird. Diesen Datentyp können Sie verwenden, wenn Sie hochgenaue Berechnungen ausführen wollen:

```
decimal pi4 = 3.14159265358979323846264338327950288419m;
Console.WriteLine(pi4); // 3,1415926535897932384626433833
```

INFO

Verwenden Sie `decimal` *immer für Währungsberechnungen, da die für diese Berechnungen erforderliche Genauigkeit von vier Stellen hinter dem Komma mit* `decimal` *auch bei recht großen Zahlen gegeben ist.*

3.5.13 Über- und Unterläufe und spezielle Werte

Typen, die Zahlen speichern, können nur einen bestimmten Bereich verwalten. Wenn dieser Wert bei einer Zuweisung oder bei einer Berechnung unter- oder überschritten wird, resultieren bei Integer-Typen ein Unter- bzw. Überlauf und bei Fließkommatypen der spezielle Wert $+\infty$ bzw. $-\infty$. `decimal` hingegen generiert eine Ausnahme vom Typ `OverflowException`. Das Verhalten des Programms in einem solchen Fall wird davon beeinflusst, ob es sich um einen Integer-, einen Fließkommatyp oder um `decimal` handelt, und bei Integer-Typen zusätzlich von einer Option, die der Compiler auswertet.

Über- und Unterläufe bei Integer-Typen

Wenn Sie einem Integer-Typen einen zu großen Wert zuweisen, resultiert daraus ein Überlauf:

Listing 3.17: Beispiel für einen Überlauf

```
byte number1 = 255;
number1 += 2;
Console.WriteLine(number1); // Ergebnis: 1
```

Der Wert in der Variablen *number1* ist durch die Addition mit 2 übergelaufen und beträgt danach 1. Dieser Wert resultiert daraus, dass der maximale `byte`-Wert (255) überschritten wurde. Bei diesem Wert sind alle Bits gesetzt. Die Addition von 2 führt dazu, dass bei der Addition von 1 zunächst alle Bits zurückgesetzt werden. Das folgende Beispiel erläutert diesen Umstand. Bei der Addition von 1 resultiert theoretisch der Dualwert 100000000:

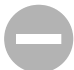
$11111111_2 + 00000001_2 = 100000000_2$

Da `byte` aber nur acht Bit verwaltet, wird das linke Bit verworfen. Das Resultat ist also 00000000. Die zusätzliche Addition von 1 führt dann dazu, dass die Variable den Wert 1 verwaltet.

Ein Unterlauf wird erzeugt, wenn Sie einen zu kleinen Wert zuweisen:

Listing 3.18: Beispiel für einen Unterlauf

```
byte number2 = 0;
number2 -= 2;
Console.WriteLine(number2); // Ergebnis: 254
```

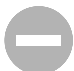

HALT

Wenn Sie nicht darauf achten, Ihre Typen ausreichend groß zu dimensionieren, kann ein Über- oder Unterlauf zu enormen logischen Fehlern führen, nach denen Sie u. U. tagelang (oder nächtelang) suchen.

Der Compiler kann so eingestellt werden, dass er bei Über- und Unterläufen eine Ausnahme generiert

Sie können den Compiler aber auch so einstellen, dass dieser bei einem Über- oder Unterlauf von Integer-Werten eine Ausnahme generiert. In Visual Studio stellen Sie dazu die Option AUF ARITHMETISCHEN ÜBER-/UNTERLAUF PRÜFEN ein. Sie finden diese Option in den Projekteigenschaften im ERSTELLEN-Register in dem Dialog, den Sie über den ERWEITERT-Schalter öffnen können. Achten Sie darauf, dass Sie die richtige Konfiguration einstellen (*Debug* oder *Release*). Sinnvoll ist die Überprüfung auf jeden Fall für die Debug-Konfiguration. Wenn Sie Ihr Programm in der Entwicklungsphase mit dieser Konfiguration fleißig testen (lassen), werden Über- und Unterläufe auf jeden Fall gemeldet, sodass Sie darauf reagieren und Fehler beseitigen können.

Im Release sollten Sie die Überprüfung u. U. ebenfalls einschalten. Dabei sollten Sie allerdings bedenken, dass die Überprüfung auf Unter- und Überlauf etwas Rechenzeit kostet (was aber bei den heutigen Rechnern kein Problem darstellen sollte, vor allen Dingen, wenn Sie nicht sehr intensiv mit Integer-Werten rechnen). Denken Sie daran, dass die Suche nach logischen Fehlern, die durch Über- oder Unterläufe verursacht werden, sehr aufwändig werden kann (sofern diese Fehler überhaupt erkannt werden).


HALT

Ich setze in meiner Praxis immer ausreichend große (und manchmal auch zu große) Integer-Typen ein. Damit minimiere ich die Gefahr von Über- und Unterläufen. `byte`-*Werte verwende ich nur in Sonderfällen als Methodenparameter. Meine Anwendungen verwalten Integer-Werte in der Regel in* `int`- *oder* `long`-*Variablen. Gehen auch Sie besser den sicheren Weg. Moderne Systeme haben kein Problem mit den paar Byte, die Ihre Programme in diesem Fall zusätzlich benötigen.*

Alternativ können Sie auch den checked-Block verwenden, um einzelne Anweisungen bei ansonsten abgeschalteter Prüfung auf einen Über- oder Unterlauf zu überprüfen:

Ein checked-Block überprüft explizit auf Unter- und Überläufe

```
checked
{
   number1 += 2;
}
```

Alle im checked-Block enthaltenen Anweisungen werden nun überprüft. Tritt ein Über- oder Unterlauf ein, wird eine Ausnahme generiert. Diese Ausnahme können Sie natürlich abfangen (wie ich in Kapitel 8 beschreibe), aber alleine die Tatsache, dass eine Ausnahme generiert wird, hilft bei der Erkennung logischer Fehler.

Wenn Sie bei eingeschalteter Prüfung einzelne Anweisungen explizit *nicht* überprüfen wollen, verwenden Sie den unchecked-Block:


```
unchecked
{
   number1 += 2;
}
```

Alternativ können Sie für einzelne Ausdrücke auch den checked- oder unchecked-Operator verwenden:

```
number1 = unchecked((byte)(number1 + 2));
```

Die Typumwandlung (byte) konvertiert in diesem Beispiel das int-Ergebnis des Ausdrucks in den Datentyp byte.

Keine Über- und Unterläufe bei Fließkommatypen

Fließkommatypen erzeugen keine Unter- und Überläufe

Bei Fließkommatypen treten Über- und Unterläufe nicht auf. Wenn Sie einem Fließkommatypen eine größere oder kleinere Zahl zuweisen, als dieser speichern kann, resultieren die speziellen Werte $+\infty$ bzw. $-\infty$, die ich im nächsten Abschnitt beschreibe. Allerdings hängt das davon ab, ob Sie mit dem Wert überhaupt in den Bereich der signifikanten Stellen gelangen. Im folgenden Listing wird die gespeicherte Zahl z. B. bei der Addition mit 2 nur im Bereich der nicht signifikanten Stellen geändert:

Listing 3.19: Demonstration des Verhaltens von Fließkommatypen bei der Zuweisung von zu großen Werten

```
float number4 = float.MaxValue;
Console.WriteLine(((double)number4).ToString("N"));
// 340.282.346.638.529.000.000.000.000.000.000.000.000,00

number4 += 2;
// Die gespeicherte Zahl wurde nicht verändert, da die Addition
// nicht den Bereich der signifikanten Stellen betraf
Console.WriteLine(((double)number4).ToString("N"));
// 340.282.346.638.529.000.000.000.000.000.000.000.000,00

// Eine Multiplikation mit 2 betrifft aber auch
// den signifikanten Bereich und führt zu +∞
number4 = number4 * 2;
Console.WriteLine(number4); // Ausgabe: +unendlich
```

HALT

Das im Listing demonstrierte Verhalten sollten Sie bei Fließkommatypen, gemeinsam mit der eingeschränkten Genauigkeit, immer im Auge behalten. Mein Tipp hier ist noch einmal: Verwenden Sie statt float *lieber* double *und bei größeren Zahlen bzw. wenn Sie mehr Nachkommastellen benötigen* decimal.

+0, -0, +∞, -∞, NaN

Fließkommatypen können spezielle Werte speichern

Über- und Unterläufe treten bei Fließkommatypen also nicht auf. Dafür können die Typen `float` und `double` (nicht `decimal`!) spezielle Werte speichern. Wenn Sie z. B. einen `double`-Typ, der eine positive Zahl speichert, durch 0 teilen, resultiert der Wert +∞(positiv unendlich):

Listing 3.20: Beispiel für einen Ausdruck, der +∞ergibt

```
double number5 = 10;
double number6 = 0;
double number7 = number5 / number6;
Console.WriteLine(number7); // Ausgabe: +unendlich
```

Teilen durch Null führt zu dem Wert »Unendlich« oder NaN

Teilen Sie einen negativen Wert durch 0, resultiert der Wert -∞(negativ unendlich). Wenn Sie 0 durch 0 teilen, ergibt das den Wert `NaN` (Not a Number). In einigen speziellen Fällen resultiert auch der Wert -0 oder +0. Die Regeln dazu sind ziemlich kompliziert und im Standard für Fließkommaoperationen beschrieben (IEEE 754). Sie finden diesen Standard unter der Adresse `754r.ucbtest.org/standards/754.pdf`. Wikipedia erläutert den Standard vielleicht etwas »verstehbarer«: `de.wikipedia.org/wiki/IEEE_754`. In der C#-1.2-Sprachspezifikation finden Sie im Abschnitt 7.7.2 bei der Beschreibung des Divisions-Operators eine Tabelle, die die einzelnen Möglichkeiten auflistet. Die C#-1.2-Sprachspezifikation finden Sie im Visual-Studio-Ordner im Unterordner *VC#\Specifications\1033*.

Nur für den Fall, dass Sie sich über die Versionsnummer wundern: Der Divisionsoperator ist bereits in C# 1.0 enthalten. Die Sprachspezifikations-Dokumente der Versionen 2.0 und 3.0 behandeln lediglich die Neuerungen.

Über einige statische Methoden des `double`- und des `float`-Typen können Sie herausfinden, ob ein Objekt einen dieser Werte speichert:

Listing 3.21: Verwendung der statischen Methoden des double-Typs zur Ermittlung der besonderen Fließkommawerte

```
if (double.IsInfinity(number6))
{
   Console.WriteLine("number6 ist unendlich.");
}
if (double.IsNegativeInfinity(number6))
{
   Console.WriteLine("number6 ist negativ unendlich.");
}
if (double.IsPositiveInfinity(number6))
{
   Console.WriteLine("number6 ist positiv unendlich.");
}
if (double.IsNaN(number6))
{
   Console.WriteLine("number6 ist NaN.");
}
```

INFO

Die Bedeutung dieser Werte ist für die Praxis wohl eher gering. Wenn Sie das Teilen durch 0 vermeiden, treten die Unendlich-Werte lediglich dann auf, wenn Sie einem Fließkommatyp eine zu kleine oder zu große Zahl zuweisen.

decimal

Der Typ `decimal` verhält sich anders als Integer- und Fließkommatypen. Er generiert bei Über- oder Unterläufen grundsätzlich eine Ausnahme vom Typ `OverflowException`.

Listing 3.22: decimal generiert bei Über- oder Unterläufen eine OverflowException

```
decimal number9 = decimal.MaxValue;
number9 += 2; // OverflowException
number9 -= 2; // OverflowException
```

Diese Ausnahme können Sie auf verschiedene Weise abfangen, wie ich in Kapitel 8 zeige.

3.5.14 Datumswerte

Datumswerte werden in einer Instanz der Struktur `DateTime` oder `DateTimeOffset` gespeichert:

```
DateTime date1;
DateTimeOffset date2
```

`DateTimeOffset` ist die neuere Struktur. Diese behebt einige Probleme, die mit `DateTime` entstehen. Beide Strukturen speichern ein Datum mit Zeitangabe und sind prinzipiell identisch. Der Unterschied ist allerdings, dass `DateTimeOffset` immer den aktuellen Offset zur UTC-Zeit verwaltet, und `DateTime` diesen Offset nur dann verwaltet, wenn es sich um ein Datum vom Typ `DateTimeKind.Local` oder `DateTimeKind.Utc` handelt. Da es bei `DateTime` sehr schnell passiert, dass ein unspezifiziertes Datum erzeugt wird (vom Typ `DateTimeKind.Unspecified`), ist dieser Typ nicht zur Speicherung oder Übertragung von Daten geeignet, wenn verschiedene Zeitzonen im Spiel sind. Auf dieses Problem gehe ich in Kapitel 8 noch näher ein. In diesem Abschnitt verwende ich den neuen Typ, der diese Probleme nicht aufweist.

Ich würde Ihnen empfehlen, grundsätzlich die neue `DateTimeOffset`*-Struktur zu verwenden und auf* `DateTime` *zu verzichten. Damit vermeiden Sie Probleme, die entstehen können, wenn Daten mit Datumswerten zwischen Systemen ausgetauscht werden, die verschiedenen Zeitzonen angehören. Leider besitzt* `DateTimeOffset` *aber nicht alle Möglichkeiten, die* `DateTime` *bietet, wie Sie im Verlauf dieses Abschnitts noch sehen.*

Dem Konstruktor der `DateTimeOffset`-Struktur können Sie verschiedene Werte übergeben, mit denen Sie das Datum genau spezifizieren können:

```
DateTimeOffset date = new DateTimeOffset(
   2099, 12, 31, 12, 0, 0, TimeSpan.Zero); // 31.12.2099 12:00 UTC-Zeit
```

Das letzte Argument im Beispiel ist ein `TimeSpan`-Wert, der eine Zeitspanne definiert. Diese Zeitspanne gibt den Offset zur UTC-Zeit an. Im Beispiel habe ich eine 0-Zeitspanne gewählt, weil ich ein UTC-Datum angeben wollte.

Das Datum 31.12.2099 habe ich übrigens gewählt, weil ich hoffe, dass sich dieses Buch dann immer noch verkauft. Dann muss ich das Datum in den einzelnen Auflagen nicht immer wieder ändern ☺.

Alternativ können Sie auch eine der statischen Eigenschaften der `DateTimeOffset`-Struktur verwenden, um das aktuelle oder ein bestimmtes Datum einzustellen:

```
// Aktuelles Datum einstellen
date = DateTimeOffset.Now;

// Datum aus einem String ermitteln
date = DateTimeOffset.Parse("31.12.2099");
```

Now liefert das aktuelle Datum

`Now` liefert dabei das bis auf die Millisekunde genaue aktuelle Datum. `Parse` ermöglicht das Parsen eines Strings mit einer Datumsangabe.

ToString formatiert ein Datum

Um ein Datum auszugeben, können Sie die `ToString`-Methode verwenden. Dieser können Sie einen Formatierungs-String übergeben. Das Beispiel zeigt, wie Sie das Datum in verschiedenen Varianten ausgeben:

Listing 3.23: Formatierte Ausgabe eines Datums

```
Console.WriteLine(date.ToString("g")); // Langes Datum inkl. Zeit
Console.WriteLine(date.ToString("d")); // Kurzes Datum
Console.WriteLine(date.ToString("T")); // Lange Zeit (12:00:00)
Console.WriteLine(date.ToString("t")); // Kurze Zeit (12:00)
```

`DateTime` besitzt zudem u. a. die Methoden `ToLongDateString`, `ToShortDateString`, `ToLongTimeString` und `ToShortTimeString`, die `DateTimeOffset` (leider) fehlen. In Kapitel 6 erfahren Sie im Abschnitt zu den Erweiterungsmethoden, wie Sie diese Methoden in `DateTimeOffset` einblenden können.

Beachten Sie bitte, dass die Formatierungen des Datums sich auf die Datums- und Zeitformat-Einstellungen im aktuellen System beziehen. Es kann z. B. sein, dass das lange Datum auf einem Windows-System in einer anderen Form ausgegeben wird, weil das Format (über die Systemsteuerung) geändert wurde.

Den Umgang mit Datumswerten beschreibe ich in Kapitel 8.

3.5.15 Zeichen und Zeichenketten

Zeichen werden in char verwaltet, Strings in string

Zur Speicherung einzelner Zeichen verwenden Sie den Typ `char`, für Zeichenketten den Typ `string`. `char` ist ein Werttyp, `string` ein Referenztyp (der aber ähnlich einem Werttyp ausgewertet wird, siehe in »Der Typ String als Ausnahme« auf Seite 171).

Zeichen und Zeichenketten werden im Speicher in der Unicode-Codierung UTF-16 verwaltet. Unicode wurde bereits in Kapitel 1 beschrieben.

Als Literal für Zeichen verwenden Sie ein in einfache Anführungsstriche eingeschlossenes Zeichen:

```
char c1 = 'A';
```

Alternativ können Sie auch den Unicode-Wert des Zeichens angeben (wenn Sie diesen kennen). C# bietet dazu die Escape-Sequenzen \x und \u, denen Sie den hexadezimalen Wert des Zeichens anhängen. \x und \u unterscheiden sich scheinbar nur dadurch, dass \u einen vierstelligen (hexadezimalen) Unicode-Wert erwartet, \x aber auch Unicode-Hexadezimalwerte mit weniger als vier Ziffern zulässt.

```
char c2 = '\u20AC'; // 0x20AC = '_'
char c3 = '\u0041'; // 0x0041 = 'A'
char c4 = '\x20AC'; // 0x20AC = '_'
char c5 = '\x41';   // 0x41 = 'A'
```

Wenn Sie den ISO-8859-1-Code eines Zeichens kennen, setzen Sie diesen im ersten Byte ein: Der ISO-8859-1-Zeichensatz ist im ersten Byte des Unicode-Zeichensatzes abgebildet.

`string`-Literale schließen Sie in Anführungszeichen ein:

```
string s1 = "Das ist ein String";
```

Innerhalb einer Zeichenkette können Sie auch die Unicode- oder Hexadezimal-Darstellung der Zeichen verwenden:

```
string s2 = "\x0041\x0042\x0043"; // "ABC"
```

Escape-Sequenzen

Escape-Sequenzen ermöglichen besondere Zeichen

`char`- und `string`-Typen können so genannte Escape-Sequenzen speichern. Eine Escape-Sequenz wird immer mit einem Backslash eingeleitet.

Einige Escape-Sequenzen sorgen dafür, dass Zeichen mit einer besonderen Bedeutung im String als normale Zeichen verwendet werden können. Das Anführungszeichen können Sie innerhalb eines String-Literals z. B. nur über die Escape-Sequenz \" verwenden, da der Compiler dieses Zeichen ansonsten als Stringbegrenzer auswertet:

```
Console.WriteLine("Sein Name war \"Old Trashbarg\"");
```

Andere Escape-Sequenzen stehen für spezifische Zeichen, wie z. B. einen Zeilenumbruch:

```
Console.WriteLine("Das ist Zeile 1\r\nDas ist Zeile 2");
```

Tabelle 3.8 listet die Escape-Sequenzen von C# auf.

Tabelle 3.8: Die C#-Escape-Sequenzen

Escape-Sequenz	Bedeutung
\'	Apostroph
\"	Anführungszeichen
\\	Backslash
\0	Null-Zeichen
\a	Warnung (alert). Erzeugt einen Piepston.
\b	Backspace
\f	Seitenvorschub (Form Feed)
\n	Zeilenumbruch (New Line)
\r	Wagenrücklauf (Carriage Return)
\t	Horizontaler Tabulator
\v	Vertikaler Tabulator

TIPP

In Zeichenketten werden häufig Zeilenumbrüche benötigt, z. B. weil mehrere Zeilen in einem `Label` *ausgegeben werden sollen. Sie könnten dazu die Escape-Sequenzen \r und \n verwenden. Dummerweise wird ein Zeilenumbruch je nach Betriebssystem aber unterschiedlich interpretiert. Auf Windows-Systemen ist das ein Carriage Return (Wagenrücklauf, '\r') gefolgt von einem Line Feed (Zeilenvorschub, '\n'). Die UNIX- und Linux-Entwickler haben die Sinnlosigkeit dieses von der Schreibmaschine übernommenen Verhaltens erkannt und nur das Line-Feed-Zeichen als Zeilenumbruch verwendet.*

Zeilenumbrüche sollten über Environment.NewLine erzeugt werden

Da .NET-Anwendungen prinzipiell (u. U. mit Einschränkungen) auch auf anderen Systemen ausgeführt werden können, sollten Sie statt den Escape-Sequenzen für Zeilenumbrüche immer die statische Eigenschaft `NewLine` *der* `Environment`*-Klasse verwenden. Auf Windows-Systemen gibt diese Eigenschaft den String "\r\n" zurück. Auf UNIX- oder Linux-Systemen wird allerdings der String "\n" zurückgegeben.*

```
Console.WriteLine(
   "Das ist Zeile 1" + Environment.NewLine +
   "Das ist Zeile 2");
```

Und nur für den Fall, dass Sie sich fragen, wie .NET-Anwendungen auf UNIX oder Linux laufen können: Das Mono-Projekt (`www.mono-project.com`*) implementiert das .NET Framework für diese (und andere) Betriebssysteme.*

Wortwörtliche Stringliterale

Wortwörtliche Stringliterale werten keine Escape-Sequenzen aus

In normalen Stringliteralen werden vorhandene Escape-Sequenzen entsprechend ihrer Bedeutung ausgewertet. In einigen Fällen kann das zum Problem werden. Wenn Sie z. B. einen Dateipfad angeben, wollen Sie die im Pfad enthaltenen Backslash-Zeichen nicht als Escape-Zeichen verwenden. Häufig beschwert sich schon der Compiler:

```
string filename1 = "c:\files\books\C#\C#.doc";
```

In diesem Fall meldet der Compiler eine »Nicht erkannte Escape-Folge«.

Enthält die Zeichenfolge zufällig nur gültige Escape-Sequenzen, ist ein Programmfehler vordefiniert:

```
string filename2 = "c:\files\new\release.doc";
Console.WriteLine(filename2);
```

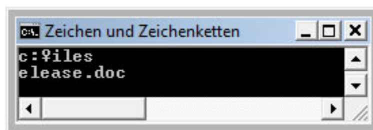


Abbildung 3.7: Ein Dateiname, der aufgrund der Escape-Sequenzen fehlerhaft ausgewertet wird

Sie können Escape-Sequenzen zwar über doppelte Backslashs deaktivieren:

```
string filename3 = "c:\\files\\new\\release.doc";
```

Einfacher und besser ist aber, stattdessen »wortwörtliche« Stringliterale (Verbatim String Literals) zu verwenden. Diese Literale, die keine Escape-Sequenzen auswerten, leiten Sie über ein @ ein:

```
string filename4 = @"c:\files\new\release.doc";
```

Wortwörtliche Stringliterale können sogar in mehreren Zeilen umbrochen werden, ohne dass Sie diese in jeder Zeile abschließen und mit + addieren müssen:

```
string s3 = @"Das ist ein
wortwörtliches Stringliteral, das einfach
umbrochen wurde";
```

Dabei müssen Sie allerdings beachten, dass nun alle Zeilenvorschübe und Leerzeichen mit in den String übernommen werden. Der String hat also genau dasselbe Format, wie es im Quellcode angegeben ist.

Der Typ char

Der Typ `char` wird in einem 16-Bit-Speicherbereich verwaltet (da es sich ja um Unicode-Zeichen in der UTF-16-Codierung handelt). Eigentlich ist zu diesem Typen nichts Besonderes zu sagen, außer, dass er mit Integer-Werten kompatibel ist. Ein `char`-Wert kann also in einen Integer-Wert umgewandelt oder konvertiert werden und umgekehrt. Das Konvertieren wird ab Seite 188 behandelt. Deshalb folgt hier nur ein kleines Beispiel:

Listing 3.24: Konvertieren von char-Werten in int und umgekehrt

```
char c4 = 'A';
int integerValue = (int)c4;
Console.WriteLine(integerValue); // 65
integerValue = 97;
c4 = (char)integerValue;
Console.WriteLine(c4); // a
```

Strings

Strings werden dynamisch verwaltet

Strings werden automatisch dynamisch verwaltet. Die Größe des Strings ist lediglich durch den verfügbaren Speicher begrenzt. Wenn Sie einen String neu zuweisen oder verändern, erzeugt das Programm immer eine neue Instanz des Strings:

```
string s4 = "Das ist ein ";
s4 = s4 + "Test"; // erzeugt eine neue Instanz
```

Wenn Sie viel mit einem String arbeiten, ist das nicht sehr effizient. In diesem Fall können Sie alternativ eine Instanz der `StringBuilder`-Klasse verwenden, die das Umkopieren vermeidet (siehe Kapitel 8).

In Stringausdrücken ruft der Compiler implizit die `ToString`-Methode anderer Typen auf, wenn dies nicht bereits im Programmcode der Fall ist. Deshalb können Sie beispielsweise einen `int`-Typen mit einem String ohne Konvertierung addieren:

```
int i = 42;
string s5 = i + " ist die Antwort auf die Frage aller Fragen";
```

Die `String`-Klasse besitzt eine große Anzahl an Methoden zur Arbeit mit Zeichenketten. In Kapitel 8 beschreibe ich, wie Sie diese nutzen.

3.5.16 Der Typ Object

Der Typ `Object`, der in C# mit dem Alias `object` verwendet wird, ist ein besonderer Typ. `Object` ist zunächst der Basistyp aller anderen .NET-Typen. Mit anderen Worten sind alle .NET-Typen mehr oder weniger direkt von `Object` abgeleitet. .NET-Typen besitzen also grundsätzlich immer die Elemente, die in `Object` enthalten sind.

Object stellt allen Typen öffentliche Methoden zur Verfügung

`Object` besitzt vier öffentliche Methoden, zwei geschützte und eine statische. Die öffentlichen Methoden werden an die anderen .NET-Typen (auch an selbst entwickelte) so vererbt, dass diese auch mit Instanzen dieser Typen aufgerufen werden können. Die `ToString`-Methode liefert z. B. in der Regel einen String zurück, der den Inhalt des Objekts darstellt. Tabelle 3.9 beschreibt die öffentlichen `Object`-Methoden.

Tabelle 3.9: Die öffentlichen Instanzmethoden des Basistyps Object

Methode	Beschreibung
`bool Equals(object` *obj*)	vergleicht die Daten eines Objekts mit den Daten eines anderen Objekts auf Gleichheit (d. h., dass die in den Objekten gespeicherten Werte gleich sind).
`int GetHashCode()`	ermittelt einen Hashcode für das Objekt. Ein Hashcode ist ein aus den Daten des Objekts ermittelter Code, der das Objekt in verkürzter Form identifiziert.
`Type GetType()`	ermittelt den Typ des Objekts. Gibt ein Objekt der Klasse `Type` zurück. Aus diesem Objekt können Sie sehr viele Informationen auslesen. Die Eigenschaft `Name` liefert z. B. den Namen des Typs, die Eigenschaft `Namespace` gibt den Bezeichner des Namensraums zurück, in dem der Typ verwaltet wird.
`string ToString()`	Diese Methode gibt die Daten des Objekts als Zeichenkette zurück.

HALT

Wenden Sie diese Methoden auf einem Objekt an, sollten Sie beachten, dass nur `GetType` *garantiert in allen Fällen korrekt funktioniert. Alle anderen Methoden müssen in den entsprechenden Typen mit einer neuen Implementierung überschrieben sein, damit diese einen korrekten Wert zurückliefern. In den Typen des .NET Framework ist dies zwar meistens, aber nicht immer der Fall. Die* `ToString`*-Methode liefert in der originalen Implementierung der* `Object`*-Klasse z. B. nur den Namen der Klasse zurück. In vielen Typen des .NET Framework (und in vielen externen Typen) wird diese Methode (leider) nicht überschrieben und liefert auch dort lediglich den Namen der Klasse zurück.*

`Object` stellt abgeleiteten Typen zudem zwei geschützte Methoden zur Verfügung. Geschützte Methoden können nicht von außen, über eine Instanz des Typs, aufgerufen werden. Wenn Sie aber eine eigene Struktur oder Klasse entwickeln, können Sie diese innerhalb der Methoden der Struktur bzw. Klasse für eigene Zwecke aufrufen.

Tabelle 3.10: Die geschützten Instanzmethoden des Basistyps object

Methode	Beschreibung
`object MemberwiseClone()`	erzeugt einen Klon des Objekts, indem eine neue Instanz erzeugt wird und die Werte aller Felder kopiert werden. Bei Feldern, deren Typ ein Referenztyp ist, wird die Referenz kopiert. Es wird keine Kopie des referenzierten Objekts erzeugt. Diese Einschränkung ist auch der Grund dafür, dass diese Methode nicht öffentlich ist. In einem abgeleiteten Typ können Sie `MemberwiseClone` verwenden, um einen Klon zu erzeugen, müssen aber gegebenenfalls Referenztyp-Felder separat behandeln.
`void Finalize()`	`Finalize` wird vom Garbage Collector aufgerufen, wenn ein Objekt zerstört wird. Über ein Überschreiben dieser Methode können Sie darauf reagieren, was aber in der Praxis eigentlich niemals notwendig sein sollte.

Neben den Instanzmethoden besitzt die `Object`-Klasse noch die statische Methode `ReferenceEquals`, der zwei `Object`-Referenzen übergeben werden und die über-

prüft, ob diese dasselbe Objekt referenzieren. Diese Methode kann für Referenztypen interessant sein, deren Vergleichsoperator (==) so überschrieben wurde, dass dieser nicht mehr die Referenzen vergleicht, sondern den Inhalt der Objekte.

Object kann als Typ verwendet werden

`Object` kann zudem als Typ eingesetzt werden. Einer `Object`-Instanz können Sie jeden anderen Typen zuweisen:

```
object o1;
o1 = 10;
o1 = "Hallo";
```

Sinnvoll ist diese Verwendung aber nicht, da Sie damit die sehr wichtige Typsicherheit unterlaufen.

Einige Methoden besitzen Object-Argumente

Einige der .NET-Methoden besitzen allerdings Argumente vom Typ `object`. Das ist – neben der Verwendung im veralteten COM-Modell – auch der Haupt-Anwendungsbereich dieses Typs. `object`-Argumenten können Sie beliebige Werte übergeben. Die `WriteLine`-Methode der `Console`-Klasse, die in einer Variante ein `object`-Argument erwartet, ist ein Beispiel dafür. Der `object`-Variante der `WriteLine`-Methode können Sie ein beliebiges Objekt übergeben. Methoden wie `WriteLine`, die den übergebenen Wert als String auswerten, nutzen dabei die `ToString`-Methode des übergebenen Objekts, das dieses von `object` geerbt hat.

Sie selbst sollten `object` *nicht verwenden. Wenn Sie an einem Methodenargument unterschiedliche Typen übergeben wollen, ist das Prinzip des Polymorphismus[4] dafür viel besser geeignet. Polymorphismus wird in Kapitel 5 behandelt.*

Object-Referenzen müssen in Ausdrücken konvertiert werden

Wenn Sie eine `object`-Referenz in einem Ausdruck verwenden, müssen Sie diesen in der Regel in einen Typ konvertieren, der zum Ausdruck passt:

```
object o2 = 10;
int i1 = (int)o2 * 10;
```

In Stringausdrücken ist das allerdings nicht notwendig, da der Compiler in diesem automatisch die `ToString`-Methode des `object`-Typen aufruft:

```
object o3 = "42";
string s1 = o3 + " ist die Antwort auf die Frage aller Fragen";
```

GetType ermittelt den eigentlichen Typ

Über die `GetType`-Methode können Sie den Typ ermitteln, der über die `object`-Referenz verwaltet wird:

```
Console.WriteLine(o3.GetType().Name); // String
```

Beachten Sie, dass `GetType` ein Objekt der Klasse `Type` zurückgibt, über das Sie noch wesentlich mehr Informationen über den Typ erhalten können, als nur dessen Namen. Die Eigenschaft `IsArray` ist z. B. `true`, wenn der Typ ein Array ist.

Verwenden Sie den Typ `object` *idealerweise nur in Ausnahmefällen. Die Typsicherheit von C# geht mit diesem Typ verloren. Wenn Sie* `object`*-Variablen verwenden, wissen Sie in größeren Programmen nie genau, welchen Typ diese tatsächlich speichern. Logische Programmfehler und Ausnahmefehler sind damit vorprogrammiert.*

4 OK, dass eine Object-Referenz alle anderen Typen verwalten kann, ist bereits Polymorphismus

Boxing und Unboxing

Boxing erlaubt, dass Object auch Werttypen verwalten kann

Der Datentyp `object` ist ein Referenztyp. Theoretisch könnte er deswegen auch nur Referenztypen verwalten. Über das so genannte Boxing und Unboxing ermöglicht die CLR es aber, dass `object` auch Werttypen verwalten kann.

Wenn ein Werttyp auf einen `object`-Typen zugewiesen wird, wird implizit Boxing verwendet:

```
int i2 = 10;
object o4 = i2; // i2 wird geboxt
```

Da `object` ein Referenztyp ist, werden die Werte, die eine `object`-Referenz verwaltet, folglich auf dem Heap gespeichert. Wenn Sie einen Werttypen auf eine `object`-Variable zuweisen oder an ein `object`-Argument übergeben, erzeugt der Compiler eine »Box« auf dem Heap und kopiert den Wert des Werttypen in diese. Die Box ist ein Referenztyp und kann deshalb über eine `object`-Variable referenziert werden.

Unboxing bedeutet dann lediglich den umgekehrten Vorgang, nämlich dass der Wert eines Werttypen aus der Box ausgelesen wird. Das geschieht aber nicht implizit, weil C# eine typsichere Sprache ist. Sie müssen die `object`-Referenz explizit in den Werttyp umwandeln:

```
object o5 = 10;
int i3 = (int)o5; // der Wert von o5 wird in den Werttyp »entboxt«
```

3.5.17 Konvertierungen

Wie Sie ja bereits wissen, ist C# eine typsichere Sprache. Sie können einem Objekt nur einen vom Typ her passenden Wert zuweisen, dessen Maximal- bzw. Minimalwert ohne Kürzungen in das Objekt passt. Einer `int`-Variablen können Sie z. B. ohne Probleme einen `byte`-Wert zuweisen:

```
byte b1 = 255;
int i1 = b1;
```

Dasselbe gilt, wenn Sie einem `Nullable`-Objekt einen passenden Werttypen zuweisen:

```
Nullable<int> i2 = b1;
i2 = i1;
```

Der Typ wird in diesem Fall implizit konvertiert. Wenn der Grundtyp aber nicht passt oder der Wert nicht in den zugewiesenen Typ passen würde, erzeugt der Compiler einen Fehler:

```
string s1 = "255";

int i3 = s1; // Fehler »Eine implizite Konvertierung vom Typ "string" in
             // "int" ist nicht möglich«

float f1 = 1.234f;

i3 = f1;     // Fehler »Der Typ "float" kann nicht implizit in "int"
             // konvertiert werden ...«

i3 = i2;     // Fehler »Der Typ "int?" kann nicht implizit in "int"
             // konvertiert werden ...«
```

Konvertierungen, die nicht implizit vorgenommen werden, können Sie bei Literalen über die Datentypzeichen explizit vornehmen. Das `double`-Literal 1.234 habe ich im Beispiel über das Zeichen f zu einem `float`-Typ umgewandelt. Andere Typen müs-

sen Sie über eine Typumwandlung (englisch: »Typecast« oder nur »Cast«) oder mit Hilfe der Methoden der `Convert`-Klasse explizit konvertieren.

Implizite und explizite Konvertierungen (Typumwandlungen)

Jeder Typ kann spezielle Operatoren für implizite und explizite Konvertierungen zur Verfügung stellen. Solch ein Operator ist so etwas wie eine Methode, die allerdings automatisch aufgerufen wird, wenn der Typ konvertiert werden muss. Wenn Sie eine eigene Klasse oder Struktur erzeugen, können Sie beliebig viele Konvertierungs-Operatoren implementieren und damit die implizite oder explizite Konvertierung von verschiedenen Typen in Ihren Typ erlauben. Kapitel 5 zeigt, wie das programmiert wird.

Implizite Konvertierungen werden automatisch vorgenommen

Implizite Konvertierungen werden, wie der Name schon sagt, implizit vorgenommen. Eine Voraussetzung dafür ist allerdings, dass der Typ, dem ein Ausdruck zugewiesen wird, einen Konvertierungsoperator für den Typ des Ausdrucks besitzt. Ist ein solcher vorhanden (was bei den Standardtypen für die möglichen impliziten Konvertierungen der Fall ist), müssen Sie nichts weiter programmieren.

Ein `int`-Typ lässt z. B. (u. a.) die implizite Konvertierung eines `byte`- und eines `short`-Typs zu:

```
int i4 = 0;
byte b2 = 10;
i4 = b2; // Implizite Konvertierung
```

Typumwandlungen erlauben eine explizite Konvertierung

Wenn ein Typ spezielle Operatoren besitzt, die die explizite Konvertierung eines anderen Typs erlauben, können Sie diesen anderen Typ über eine Typumwandlung explizit konvertieren. Dazu setzen Sie den neuen Typ in Klammern vor den zu konvertierenden Ausdruck. Die Typen `int` und `float` erlauben z. B. die explizite Konvertierung eines `double`-Werts.

Listing 3.25: Explizite Typumwandlung

```
double d1 = 1.2345678901234567890;

// Explizite Konvertierung. Die Dezimalstellen gehen verloren.
int i5 = (int)d1;

// Explizite Konvertierung. Die Genauigkeit wird eingeschränkt.
float f2 = (float)d1;
```

Wie bereits gesagt, kann es beim expliziten Konvertieren vorkommen, dass Informationen verloren gehen. Die `int`-Variable des Beispiels speichert nach der Ausführung z. B. den Wert 1, die `float`-Variable den Wert 1,23456788.

HALT

Ein wichtiger Merksatz ist, dass bei impliziten Konvertierungen normalerweise[5] nichts verloren geht (bzw. »gehen sollte«, schließlich ist es dem Programmierer eines Typs überlassen, wie er implizite Konvertierungsoperatoren implementiert). Bei expliziten Konvertierungen kann es aber sein, dass Informationen verloren gehen.

Implizite Konvertierungen können auch explizit verwendet werden

Für einige Fälle ist auch interessant, dass Sie eine vorhandene implizite Konvertierung auch explizit einsetzen können, um den Typ eines Ausdrucks festzulegen. Wenn Sie z. B. eine Integer-Variable durch eine andere teilen, erhalten Sie normalerweise eine Ganzzahldivision (siehe Seite 203). Wollen Sie aber eine normale Division ausführen, müssen Sie eine der Variablen nach `float` oder `double` umwandeln:

5 Die Konvertierung eines long-Werts in einen double-Wert führt allerdings dann zu ungenauen Werten, wenn es sich um große long-Werte handelt.

Listing 3.26: Explizite Verwendung einer impliziten Konvertierung

```
int i6 = 1;
int i7 = 3;
double d2 = i6 / (double)i7;
```

Konvertieren von Nullables

Wenn Sie einem Werttypen ein `Nullable`-Objekt zuweisen wollen, müssen Sie diesen ebenfalls explizit konvertieren. Dabei müssen Sie beachten, dass eine Ausnahme generiert wird, wenn das `Nullable`-Objekt `null` speichert. Vor der Typumwandlung sollten Sie also auf null abfragen:

Listing 3.27: Explizite Typumwandlung bei Nullables

```
int? i8 = null;
int i9;
if (i8 != null)
{
   i9 = (int)i8;
}
```

Änderung des Typs eines Ausdrucks

Wenn Sie den Typ von Ausdrücken ändern wollen, müssen Sie diese klammern. Der Typumwandlungsoperator gilt immer nur für den Operanden rechts von ihm:

```
float f3 = (float)(1.234567890 * 1.234567890);
```

Konvertierungen über die Convert-Klasse

Typen, die nicht denselben Grundtyp besitzen, können nicht einfach über eine Typumwandlung konvertiert werden. Ein String kann z. B. nicht in einen `int`-Wert umgewandelt werden:

Listing 3.28: Explizite Typumwandlungen von Typen, die nicht denselben Grundtyp besitzen, sind nicht möglich

```
string s2 = "10";
int i10 = (int)s2; // Fehler »Eine Konvertierung vom Typ "string" in
                   // "int" ist nicht möglich«
```

Convert erlaubt die Konvertierung unterschiedlicher Typen

Zur Konvertierung von Typen mit unterschiedlichen Grundtypen können Sie die Klassenmethoden der `Convert`-Klasse verwenden. Der Grund dafür liegt darin, dass Konvertierungen zwischen grundverschiedenen Typen auch fehlschlagen können. Die Methoden der `Convert`-Klasse erzeugen in diesem Fall eine Ausnahme.

Einen String konvertieren Sie z. B. über die `ToInt32`-Methode in einen Integer-Wert:

Listing 3.29: Verwendung der Convert-Klasse zum Konvertieren grundverschiedener Typen

```
string s3 = "10";
int i11 = Convert.ToInt32(s3);
```

Schlägt die Konvertierung fehl, erzeugt die Methode eine Ausnahme vom Typ `FormatException`:

```
string s4 = "10a";
int i12 = Convert.ToInt32(s4); // Ausnahme
```

Diese Ausnahme können Sie abfangen, wie ich es noch in Kapitel 8 zeige.

Tabelle 3.11 zeigt die wichtigsten Klassenmethoden der `Convert`-Klasse. Diese Methoden liegen in mehreren Varianten vor, die die unterschiedlichen Typen übergeben bekommen. Ich beschreibe nicht alle diese Varianten und erläutere deswegen die Grundlagen:

Die Varianten, die keinen String übergeben bekommen, besitzen nur ein Argument. Der `ToInt32`-Methode können Sie z. B. im ersten Argument (u. a.) einen `float`-, `short`- oder `double`-Wert übergeben. Die Variante, die einen String übergeben bekommt, kann zusätzlich noch mit einem zweiten Argument aufgerufen werden, an dem Sie einen Format-Provider übergeben, der die länderspezifische Formatierung des Strings festlegt. Übergeben Sie diesen nicht, wird die Systemeinstellung berücksichtigt.

Methode	konvertiert in
`ToBoolean`(...)	`bool`
`ToByte`(...)	`byte`
`ToChar`(...)	`char`
`ToDateTime`(...)	`DateTime`
`ToDecimal`(...)	`decimal`
`ToDouble`(...)	`Double`
`ToInt16`(...)	`Short`
`ToInt32`(...)	`Int`
`ToInt64`(...)	`Long`
`ToSByte`(...)	`Sbyte`
`ToSingle`(...)	`Float`
`ToString`(...)	`String`
`ToUInt16`(...)	`ushort`
`ToUInt32`(...)	`uint`
`ToUInt64`(...)	`ulong`

Tabelle 3.11: Die wichtigsten Klassenmethoden der Convert-Klasse

TIPP

Für Konvertierungen in Strings können Sie auch die `ToString`-Methode verwenden, die alle Typen von `object` geerbt haben. Ob der zurückgegebene String den Inhalt des Objekts darstellt, hängt aber davon ab, ob der jeweilige Typ die `ToString`-Methode mit einer eigenen Implementierung überschreibt. Bei den Standardtypen und vielen anderen .NET-Typen ist das der Fall. Bei Typen, die `ToString` nicht überschreiben, wird dann allerdings die Implementierung verwendet, die in `object` definiert ist. Und die gibt einfach nur den vollen Klassennamen (inkl. Namensraum) zurück.

3.5.18 Aufzählungen (Enumerationen)

Aufzählungen bestehen aus mehreren Konstanten

Aufzählungen sind Typen, die aus mehreren benannten Zahlkonstanten bestehen. Eine Aufzählung wird nach dem folgenden Schema deklariert:

```
[Attribute] [Modifizierer] enum Name [: Basistyp]
{
   Konstantenliste
};
```

In der Liste geben Sie einen oder mehrere Bezeichner an.

Das folgende Beispiel deklariert eine Aufzählung, die Himmelsrichtungen verwaltet:

Listing 3.30: Einfache Aufzählung

```
private enum Direction
{
   North,
   South,
   East,
   West
}
```

Wenn Sie keinen Typ angeben, besitzt die Aufzählung den Typ `int`. Sie können den Typ aber auch auf einen anderen festlegen, indem Sie diesen durch einen Doppelpunkt getrennt anhängen:

Listing 3.31: Aufzählung mit expliziter Datentypangabe

```
private enum TrafficLightColor: byte
{
   Red,
   Yellow,
   Green
}
```

Als Typ können Sie lediglich die Integer-Typen (`byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` und `ulong`) angeben. Andere Typen, und damit (leider) auch `string`, sind nicht möglich.

Die neu erzeugten Aufzählungstypen können Sie nun überall da einsetzen, wo diese gültig sind. Zur Gültigkeit erfahren Sie mehr in Kapitel 4. Eine Methode kann z. B. ein Argument dieses Typs besitzen:

```
static void Move(Direction direction)
{
   ...
}
```

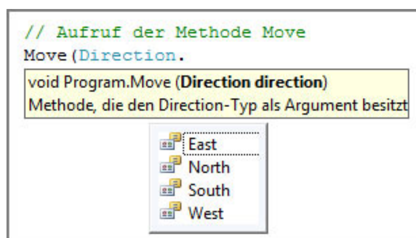


Abbildung 3.8: IntelliSense bei der Verwendung einer Aufzählung

Beim Aufruf der Methode muss nun ein passender Wert übergeben werden. Dabei greifen Sie über den Namen der Aufzählung auf die einzelnen Konstanten zu:

```
Move(Direction.North);
```

Visual Studio unterstützt Sie dabei mit IntelliSense.

TIPP

Verwenden Sie Aufzählungen immer da, wo nur eine begrenzte Anzahl an Möglichkeiten besteht. Die Methode mit den Himmelsrichtungen ist dafür ein gutes Beispiel. Sie könnten aber auch auf die Idee kommen, solche Methoden mit einfachen Integer-Argumenten auszustatten. Sie müssten dann innerhalb der Methode im Programm festlegen, dass mit 0 Norden, mit 1 Osten etc. gemeint ist. Das Problem mit einer solchen Programmierung (die ich in meiner Praxis immer wieder sehe) ist aber, dass beim Aufruf der Methode nicht klar ist, was da eigentlich übergeben werden muss. Hinzu kommt, dass der Aufrufer (der auch ein anderer Programmierer sein kann) natürlich auch vollkommen ungeeignete Werte übergeben kann. Aufzählungen verhindern diese Probleme, da die verwendeten Werte zum einen für den Menschen lesbar sind und der C#-Compiler zum anderen nur die in der Aufzählung angegebenen Werte erlaubt (mit der Ausnahme, dass Sie auch ungültige Integer-Werte in die Aufzählung konvertieren können, wie ich unten beschreibe).

Explizite Vergabe der Aufzählungswerte und bitweise Aufzählungen (Flag-Aufzählungen)

Wenn Sie bei der Deklaration der Auflistung keinen Wert für die einzelnen Konstanten angeben, erhalten diese einen bei 0 beginnenden Wert. Sie können den Wert allerdings auch festlegen:

Listing 3.32: Aufzählung, bei der die Werte festgelegt sind

```
private enum FavoriteSports
{
   Windsurfing = 1,
   Snowboarding = 2,
   Mountainboarding = 4,
   Landkiting = 8,
   Biking = 16
}
```

Aufzählungen können Flags verwalten

Wenn Sie die Werte wie im Beispiel so definieren, dass diese die einzelnen Bits des Typs repräsentieren, können Sie die Auflistung bitweise einsetzen. Dazu ist etwas Wissen notwendig, das erst in Abschnitt »Bitoperationen« (Seite 205) vermittelt wird:

Listing 3.33: Beschreiben einer Aufzählungsvariablen, die Werte bitweise verwalten kann

```
FavoriteSports mySports =
   FavoriteSports.Windsurfing |
   FavoriteSports.Snowboarding |
   FavoriteSports.Landkiting |
   FavoriteSports.Biking;
```

Sie können solche Aufzählungen natürlich auch auswerten. Das Auswerten setzt wieder Wissen aus dem Abschnitt »Bitoperationen« (Seite 205) voraus:

Listing 3.34: Auswerten einer Aufzählungsvariablen, die Werte bitweise verwalten kann

```
if ((mySports & FavoriteSports.Windsurfing) > 0)
{
   Console.WriteLine("Ich mag Windsurfen");
}
if ((mySports & FavoriteSports.Snowboarding) > 0)
{
```

```
    Console.WriteLine("Ich mag Snowboarden");
}
if ((mySports & FavoriteSports.Mountainboarding) > 0)
{
    Console.WriteLine("Ich mag Mountainboarden");
}
if ((mySports & FavoriteSports.Landkiting) > 0)
{
    Console.WriteLine("Ich mag Landkiten");
}
if ((mySports & FavoriteSports.Biking) > 0)
{
    Console.WriteLine("Ich mag Biken");
}
```

TIPP

Die `ToString`*-Methode einer Aufzählung gibt normalerweise den Namen der Konstante zurück, wenn nur der Wert einer Konstanten gespeichert ist. Sind aber in einer Aufzählung, die die Konstantenwerte bitweise verwaltet, mehrere Werte gesetzt, wird der numerische Wert zurückgegeben. Im Beispiel wäre das 27. Wenn Sie allerdings Aufzählungen, die Werte bitweise verwalten, mit dem* `Flags`*-Attribut belegen, führt dies dazu, dass* `ToString` *die Namen der gespeicherten Konstanten kommagetrennt zurückgibt:*

Listing 3.35: Flags-Aufzählung

```
[Flags]
private enum FavoriteSports
{
    Windsurfing = 1,
    Snowboarding = 2,
    Mountainboarding = 4,
    Landkiting = 8,
    Biking = 16
}
```

REF

Attribute werden in Kapitel 6 behandelt.

INFO

Wenn Sie Flags verwalten wollen, können Sie dazu auch die Klassen `BitArray` *und* `BitVector32` *verwenden. Diese Klassen werden in Kapitel 7 beschrieben.*

Zusammenfassen von Flag-Werten

In Flags-Aufzählungen können Sie die einzelnen Werte auch zusammenfassen um diese einfacher setzen und abfragen zu können:

Listing 3.36: Flags-Aufzählung mit zusammengefassten Werten

```
[Flags]
private enum FavoriteSports
{
    Windsurfing = 1,
    Snowboarding = 2,
    Mountainboarding = 4,
    Landkiting = 8,
    Biking = 16,
    WindSports = Windsurfing | Landkiting,
    MountainSports = Snowboarding | Mountainboarding | Biking
}
```

Konvertieren von Aufzählungen

Aufzählungen können natürlich auch in ihren Basistyp umgewandelt werden und umgekehrt. Dazu verwenden Sie eine explizite Konvertierung:

Listing 3.37: Umwandeln eines Aufzählungswerts in seinen Basistyp und umgekehrt

```
Direction direction = Direction.North;
byte directionValue = (byte)direction;
Console.WriteLine("Der Wert von " + direction +
   " ist " + directionValue); // Der Wert von North ist 0
directionValue = 2;
direction = (Direction)directionValue;
Console.WriteLine("Der Direction-Name von " + directionValue +
   " ist " + direction); // Der Direction-Name von 2 ist East
```

Beim expliziten Konvertieren können Sie die Typsicherheit unterlaufen. So können Sie der `Direction`*-Aufzählung z. B. den eigentlich ungültigen Wert 42 zuweisen:*

```
direction = (Direction)42;
Console.WriteLine(direction); // 42
```

Auch wenn das philosophisch betrachtet vielleicht korrekt wäre (42 ist schließlich die Antwort auf die Frage aller Fragen, warum sollte diese wichtige Antwort nicht auch gleich die richtige Richtung (fürs Leben) sein ☺), das Programm funktioniert in einem solchen Fall mit Sicherheit nicht korrekt. Passen Sie bei Konvertierungen von Integer-Werten in Aufzählungswerte also auf.

3

Ein Tipp

Der folgende Tipp für die Arbeit mit Aufzählungen ist für die Praxis sehr wichtig, setzt allerdings Programmiertechniken voraus, die erst später behandelt werden.

Wenn Sie alle Werte einer Aufzählung ermitteln wollen, können Sie dazu die Methode `GetValues` der `Enum`-Klasse verwenden.

Listing 3.38: Ermitteln aller Werte einer Aufzählung

```
foreach (Direction d in Enum.GetValues(typeof(Direction)))
{
   Console.WriteLine(d);
}
```

Über `Enum.GetNames` können Sie die Namen der Aufzählungskonstanten ermitteln. Das ist aber eigentlich nicht notwendig, da die `ToString`-Methode eines Aufzählungstyps den Namen der Konstante sowieso bereits zurückgibt, wie es im obigen Beispiel der Fall ist.

3.6 Variablen und Konstanten

In Variablen und Konstanten können Sie Werte speichern und Objekte referenzieren. Eine Variable oder eine Konstante besitzt wie Literale oder Ausdrücke einen Typ. Variablen können im Programm verändert werden, der Wert einer Konstanten ist unveränderlich.

C# unterscheidet die üblichen Bereiche für die Lebensdauer und die Gültigkeit: Variablen und Konstanten können Sie innerhalb einer Methode deklarieren. Dann gilt diese nur innerhalb der Methode und lebt nur so lange, wie die Methode ausgeführt

wird. Eine andere Möglichkeit der Deklaration ist außerhalb von Methoden auf der Ebene der Klasse oder Struktur. »Variablen« werden dann allerdings als *Felder* bezeichnet. Felder können statisch sein, dann leben sie, solange das Programm ausgeführt wird. »Normale« Felder sind Instanzfelder und leben nur so lange, wie die Instanz der Klasse bzw. Struktur (das Objekt) lebt.

Ich beschreibe hier nur die Deklaration von Variablen und Konstanten innerhalb einer Methode, weil Kapitel 4 ausführlich auf normale und statische Felder (und Eigenschaften) eingeht.

3.6.1 Die Deklaration von Variablen

Bei der Deklaration einer Variablen geben Sie optionale Attribute, einen optionalen Modifizierer, den Datentyp und den Bezeichner an. Eine Variable können Sie direkt bei der Deklaration initialisieren:

```
[Attribute] [Modifizierer] {Typ | var} Bezeichner [= Ausdruck];
```

Über Attribute können Sie zusätzliche Informationen zu einer Variablen definieren, die entweder vom Compiler oder von externen Programmen ausgewertet werden können. Attribute werden in Kapitel 6 behandelt.

Der Modifizierer wird bei Feldern (und Eigenschaften) verwendet, die Sie innerhalb von Klassen deklarieren (siehe Kapitel 4). Er legt den Gültigkeitsbereich des Feldes bzw. der Eigenschaft fest.

Als Typ für die Variable wird entweder ein vorhandener Typ (z. B. `int`) oder das Schlüsselwort `var` angegeben. `var` führt zu einer implizit typisierten Variable (Seite 198).

Innerhalb einer Methode wird kein Modifizierer verwendet. Die folgende Anweisung deklariert eine `int`-Variable und initialisiert deren Wert auf 0:

```
int i1 = 0;
```

Sie können in einer Anweisung gleich mehrere Variablen deklarieren, die Sie durch Kommata getrennt angeben:

```
int i2 = 0, j1 = 0, k1 = 0;
```

Die so deklarierten Variablen erhalten natürlich alle denselben Typ.

Blockweise Gültigkeit

Variablen gelten blockweise

In C# gelten Variablen blockweise. Wenn Sie eine Variable in einem Block deklarieren, können Sie außerhalb des Blocks nicht auf die Variable zugreifen:

```
// Deklaration innerhalb einer Abfrage
if (i1 == 0)
{
   string s1 = "i1 ist 0";
}
Console.WriteLine(s1); // Dieser Zugriff ist nicht möglich
```

Daneben können Sie innerhalb einer Methode keine Variable erneut deklarieren, die in der Methode bereits deklariert wurde. Dazu gehören auch Variablen, die bereits in Blöcken deklariert wurden, folglich eigentlich nur in dem Block gelten:

```
// Deklaration innerhalb einer Abfrage
if (i1 == 0)
{
```

```
   string s2 = "i1 ist 0";
   Console.WriteLine(s2);
}
string s2 = ""; // Diese Deklaration ist nicht möglich
```

Sie können aber Felder und Eigenschaften deklarieren, die denselben Namen tragen wie Variablen in einer Methode. Oder andersherum: Die Namen von Variablen können dieselben sein wie die von Feldern oder Eigenschaften. Wenn Sie innerhalb einer Methode auf eine solche Variable zugreifen, verwendet der Compiler immer die auf der niedrigsten Gültigkeitsebene, also die Variable, die innerhalb der Methode deklariert ist.

TIPP

Wenn Sie in einer Methode explizit auf ein Feld oder eine Eigenschaft zugreifen wollen, das/die denselben Namen trägt wie eine Variable, verwenden Sie den `this`*-Operator als Präfix vor dem Namen. Wenn Sie sich an die Konvention halten, diesen Operator beim Zugriff auf Felder und Eigenschaften immer zu verwenden, haben Sie prinzipiell kein Problem beim Zugriff auf Variablen, Felder und Eigenschaften.*

3

3.6.2 Konstanten

Konstanten werden ähnlich wie Variablen deklariert:

```
[Attribute] [Modifizierer] const Typ Name = Ausdruck;
```

Bei einer Konstantendeklaration müssen Sie immer einen Wert zuweisen. Die Modifizierer gelten wieder für die Deklaration auf Klassenebene. Den Wert einer Konstanten können Sie im Programm nicht mehr ändern (das ist ja auch der Sinn einer Konstanten ☺).

Wenn Sie konsequent Konstanten einsetzen, verzichten Sie damit automatisch auf Magic Numbers (Magische Zahlen). Magische Zahlen sind in Programmen verwendete Zahlliterale. Das Magische an diesen Zahlen ist, dass deren Bedeutung an Hand des Werts nicht erfasst werden kann. Setzen Sie stattdessen Konstanten (oder Aufzählungen) ein, ist deren Bedeutung hingegen klar, sofern Sie aussagekräftige Namen verwenden.

Konstanten erleichtern die Wartung eines Programms

Ein anderer Grund für den Verzicht auf magische Zahlen ist, dass die Werte, mit denen Sie arbeiten, sich im Laufe der Weiterentwicklung des Programms ändern können. Wenn Sie eine magische Zahl für einen Wert verwenden, der an vielen Stellen im Programm benötigt wird, ist eine Änderung nur sehr schwer möglich und führt in vielen Fällen zu Fehlern. Setzen Sie stattdessen eine aussagekräftig benannte Konstante ein, ist die Änderung einfach.

Auch die spätere Umsetzung einer Konstanten in eine Variable, in ein Feld oder in eine Eigenschaft ist einfach. Diese Umsetzung ist z. B. dann notwendig, wenn Sie den Wert in der Laufzeit aus der Konfiguration der Anwendung auslesen und diesen so für den Anwender änderbar machen (wie das geht, zeigt Kapitel 8).

3.6.3 Array-Grundlagen

Arrays speichern Listen von Variablen

Arrays erlauben das zusammenhängende Speichern mehrerer gleichartiger Informationen. Sie verhalten sich wie eine Liste einzelner Variablen, auf die Sie über den Arraynamen und einen Integer-Index zugreifen können. Arrays sind in C# komplexe Objekte und besitzen einige Möglichkeiten, die ich in diesem Kapitel noch nicht besprechen will. In Kapitel 7 erfahren Sie wesentlich mehr über Arrays und daneben

über die für die Speicherung von Daten-Listen wesentlich flexibleren Auflistungen. Hier beschreibe ich lediglich, wie Sie einfache (eindimensionale) Arrays erzeugen und damit arbeiten.

Einfache (eindimensionale) Arrays erzeugen Sie nach dem folgenden Schema:

```
Typ[] Name = new Typ[Anzahl der Elemente];
```

Als Typ können Sie jeden der verfügbaren Typen einsetzen, auch komplexe Typen wie Strukturen oder Klassen.

Der Name eines Arrays entspricht den Namen einer Variablen und muss den Bedingungen für Bezeichner genügen.

Arrays sind in C# Referenztypen, die mit `new` erzeugt werden müssen. Dabei geben Sie hinter dem Typ die Anzahl der Elemente in eckigen Klammern an.

So können Sie z. B. ein Array deklarieren, das drei Integer-Werte verwalten kann:

Listing 3.39: Deklaration eines Integer-Arrays

```
int[] numbers = new int[3];
```

Der Index eines Arrays beginnt bei 0

Die einzelnen Elemente eines Arrays können Sie wie Variablen lesen und schreiben. Der Unterschied ist lediglich, dass Sie den Index des Elements angeben müssen. Dazu verwenden Sie die C#-Indexer-Syntax: Schreiben Sie den Index in eckige Klammern hinter den Namen der Variablen. Der Index eines Arrays (und einer Auflistung) beginnt in C# immer mit 0:

Listing 3.40: Zugriff auf ein Array

```
numbers[0] = 3;
numbers[1] = 7;
numbers[2] = 42;
```

Arrays können durchlaufen werden

Arrays haben gegenüber normalen Variablen zur Speicherung zusammenhängender Daten die Vorteile, dass die Daten über eine Variable erreichbar sind und dass diese über eine Schleife durchlaufen werden können. Die `Length`-Eigenschaft, die die Anzahl der gespeicherten Elemente zurückgibt, ergibt dabei den größten Index (subtrahiert mit 1):

Listing 3.41: Durchgehen eines Arrays

```
for (int i = 0; i < numbers.Length; i++)
{
   Console.WriteLine(numbers[i]);
}
```

Schleifen werden ab Seite 223 behandelt.

3.6.4 Implizit typisierte lokale Variablen

NEU

C# 3.0 ermöglicht die Deklaration von Variablen ohne Typangabe über das `var`-Schlüsselwort.

```
var number = 11.5;
```

Auf diese Weise deklarierte Variablen sind allerdings nicht (wie z. B. in JavaScript) typenlos. Der Compiler ermittelt beim Kompilieren den Typ des zugewiesenen Ausdrucks und erzeugt im CIL-Code eine Variable von diesem Typ. Die Variable in der obigen Anweisung besitzt z. B. den Typ `double`. Beweisen können Sie dies, indem Sie den Typ ausgeben:

Implizit typisierte lokale Variablen sind nicht typlos

```
Console.WriteLine(number.GetType().Name); // Double
```

Ein weiterer Beweis ist der erzeugte CIL-Code:

Listing 3.42: Auszug aus dem CIL-Code einer Konsolenanwendung mit den oben angegebenen Anweisungen

```
method private hidebysig static void  Main(string[] args) cil managed
{
  .entrypoint
  // Code size       34 (0x22)
  .maxstack  1
  .locals init ([0] float64 number)
  IL_0000:  nop
  IL_0001:  ldc.r8     11.5
  IL_000a:  stloc.0
  IL_000b:  ldloc.0
  ...
```


Für implizit typisierte lokale Variablen gilt eigentlich nur eine Regel: Der Compiler muss die Möglichkeit haben, den Typ der Variablen aus dem zugewiesenen Ausdruck zu erkennen. Der Typ des Ausdrucks ist dabei unerheblich, es kann sich um die Standardtypen, um sonstige Werttypen und um Referenztypen handeln. Objektinitialisierer (und die in Kapitel 7 behandelten Auflistungs-Initialisierer) sind nicht direkt möglich, da ein solcher Ausdruck keinen Typ ergibt. Sie können aber neue Objekte über das `new`-Schlüsselwort erzeugen und damit Objekt- oder Auflistungs-Initialisierer verwenden:

Der Compiler muss den Typ einer implizit typisierten lokalen Variablen erkennen

Listing 3.43: Mögliche Zuweisungen an implizit typisierte lokale Variablen

```
// OK, der Ausdruck ergibt float
var v1 = 10 * 1.5F;

// OK, der Ausdruck ergibt ein DateTime-Objekt
var v2 = new DateTime(2010, 12, 31);

// OK, der Ausdruck ergibt ein PersonStruct-Objekt
var v3 = new PersonStruct { FirstName = "Zaphod",
   LastName = "Beeblebrox" };

// Anmerkung: PersonStruct entstammt dem Abschnitt
// »Wert- und Referenztypen« auf Seite _Ref179608779
```

Listing 3.44: Nicht mögliche Zuweisungen an implizit typisierte lokale Variablen

```
// Nicht OK, da Array-Initialisierer nicht unterstützt werden
var v4 = { 1, 2, 3 };

// Nicht OK, da null keinen Typ besitzt
var v5 = null;
```

Das automatische Erkennen des Variablentyps wird übrigens als *Typrückschluss* (Type inference) bezeichnet.

Implizit typisierte lokale Variablen sind bei anonymen Typen (Kapitel 4) obligatorisch und werden intensiv mit LINQ (Kapitel 11) eingesetzt. Außerdem können Sie

diese Variablen sehr gut mit der `foreach`-Schleife einsetzen (Seite 226), was der `foreach`-Codeschnipsel von Visual Studio bereits macht.

3.7 Namensrichtlinien

Quellcode wird über Namensrichtlinien verständlicher

Bei der Programmierung macht es immer Sinn, sich an gewisse Richtlinien bei der Vergabe von Bezeichnern zu halten. Damit machen Sie es anderen Programmierern (und auch sich selbst) leichter, Ihren Quelltext zu verstehen.

Für die verschiedenen Programmiersprachen haben sich unterschiedliche Notationen für die Benennung entwickelt. C++ -Programmierer verwenden z. B. meist die so genannte »Ungarische Notation«, Visual Basic-Programmierer wenden die »Reddik-Konvention« an.

Für C# beschreibt Microsoft in der C#-Dokumentation eine Richtlinie für die Benennung von Bezeichnern, die recht übersichtlich ist. Diskussionen in verschiedenen Newsgroups und die im Internet verfügbaren C#-Programme zeigen, dass sich diese Richtlinien durchgesetzt haben.

Bezeichner werden mit Pascal und Camel Casing benannt

Grundlage der Benennung von Bezeichnern sind das so genannte *Pascal Casing* und das *Camel Casing*. Beim Pascal Casing beginnt der Bezeichner mit einem Großbuchstaben und wird klein weitergeschrieben. Besteht der Bezeichner aus mehreren Worten, wird jedes Wort wieder mit einem Großbuchstaben begonnen:

```
public int LeftMargin;
```

Das Camel Casing ist ähnlich, nur dass der Bezeichner mit einem Kleinbuchstaben beginnt:

```
int leftMargin;
```

Pascal Casing wird hauptsächlich bei öffentlichen Elementen verwendet, Camel Casing bei privaten oder lokalen Elementen. Tabelle 3.12 fasst die Richtlinien zusammen.

Tabelle 3.12: Namensrichtlinien für C#-Programme

Element	Namensrichtlinie
Klasse	– Pascal Casing
Schnittstelle	– Pascal Casing – »I« als Präfix
Aufzählungen (Enums)	– Pascal Casing (für den Namen der Aufzählung und die Namen der Werte)
Eigenschaften	– Bennennung mit Substantiven (Hauptwörtern) oder Substantiv-Phrasen (z. B. *Color*, *FirstName*) – Pascal Casing für öffentliche Eigenschaften – Camel Casing für private und geschützte Eigenschaften
Methoden	– Bennennung mit Verben oder Verb-Phrasen (z. B. *Remove*, *RemoveAll*) – Pascal Casing

Element	Namensrichtlinie
Ereignisse	– Benennen Sie die Delegaten für Ereignisse mit dem »EventHandler«-Suffix (z. B. *MouseEventHandler*) – Verwenden Sie die zwei Argumente *sender* und *e* – Benennen Sie Ereignisargument-Klassen mit dem Suffix »EventArgs« (z. B. *MouseEventArgs*) – Verwenden Sie Pascal Casing
Argumente	– Verwenden Sie aussagekräftige Namen – Verwenden Sie Camel Casing

Tabelle 3.12: Namensrichtlinien für C#-Programme (Forts.)

Keine Präfixe für Variablen

Präfixe werden (von fast allen C#-Programmierern, wie Recherchen in Newsgroups zeigten) grundsätzlich nicht verwendet. Die C#-Konvention weicht damit erheblich von anderen Konventionen ab, die eine Präfix vor den eigentlichen Namen setzen, der den Datentyp kennzeichnet. Diese in C++ als ungarische Notation und in Visual Basic als Reddik-Konvention bezeichnete Konvention bringt in der Praxis aber unter C# nicht allzu viele Vorteile. C# ist typsicher, Sie müssen also nicht wissen, welchen Typ eine Variable besitzt: Der Compiler meldet bei falschen Zuweisungen einen Fehler und Sie müssen explizit konvertieren, wenn der Typ nicht implizit konvertiert werden kann.

Ich benenne die Steuerelemente auf `Windows.Forms`-, *WPF- und ASP.NET-Formularen (und anderen Containern) allerdings mit einem zwei- bis dreistelligen, kleingeschriebenen Präfix, den ich dem Namen der Klasse entnehme. Der Name einer* `Button`-*Instanz beginnt in meinen Anwendungen (und im Buch) z. B. immer mit* `btn`. *Der Vorteil dieser Konvention ist, dass ich die verwendeten Steuerelemente im Programm über IntelliSense sehr schnell finde, da ich nur den Präfix eingeben muss, um alle Steuerelemente einer bestimmten Klasse untereinander aufgelistet zu bekommen. Das hat sich in der Praxis bewährt. Sie finden meine Namenskonvention im Anhang.*

3.8 Ausdrücke und Operatoren

Die Typen, die Sie in den vorhergehenden Kapiteln kennen gelernt haben, setzen Sie beim Programmieren häufig in Ausdrücken als Operanden ein, die Sie über Operatoren miteinander verknüpfen. Ausdrücke sind entweder arithmetische (die etwas berechnen) oder logische (die eine Bedingung überprüfen).

3.8.1 Der Typ eines Ausdrucks

Ein Ausdruck ergibt eigentlich immer einen Wert, der einen Typ besitzt. Eigentlich deswegen, weil es eine Sonderform eines Ausdrucks gibt, die keinen Wert ergibt. Das ist z. B. dann der Fall, wenn eine Methode aufgerufen wird, die nichts zurückgibt, wie z. B. `Console.WriteLine`. Der einfachste Leer-Ausdruck ist allerdings ein einfaches Semikolon.

Um Ausdrücke zu verstehen, müssen Sie wissen, wie der Compiler den Typ eines Ausdrucks bestimmt.

Der Compiler bestimmt den Typ eines Ausdrucks

Der Compiler muss beim Kompilieren den Typ eines Ausdrucks bestimmen, weil das Ergebnis eines Ausdrucks (und jedes Teilergebnis) in einem temporären Speicherbereich gespeichert wird, bevor dieses weiterverarbeitet wird. Dieser Speicherbereich

muss dazu eine für das Ergebnis (wahrscheinlich) ausreichende Größe besitzen und im weiteren (CIL-)Programm mit dem richtigen Typ ausgewertet werden.

Zur Bestimmung des Typs eines Ausdrucks untersucht der Compiler die einzelnen Operanden. Als Ergebnistypen verwendet er einen der enthaltenen Typen, der das Ergebnis am wahrscheinlichsten aufnehmen kann. Der Typ des Ausdrucks (byte)1 + 10 ist demnach int, der Typ des Ausdrucks 1 + 10.5 ist double, bei 1 * 10F kommt ein float-Wert heraus. Dabei werden aber immer nur die im Ausdruck enthaltenen Typen als Basis verwendet.

Das kann natürlich auch zu Problemen führen. Wenn Sie z. B. zwei int-Werte addieren oder multiplizieren, kann der vom Compiler verwendete int-Speicherbereich das tatsächliche Ergebnis häufig nicht verwalten und es kommt zu einem Unter- oder Überlauf. Das passiert sogar dann, wenn Sie das Ergebnis einem größeren Typen zuweisen:

Listing 3.45: Überlauf in einem Integer-Ausdruck

```
int number1 = int.MaxValue;
long result1 = number1 * 2;
Console.WriteLine(result1); // -2
```

Das Ergebnis des Ausdrucks ist hier -2, weil die Multiplikation des Maximalwerts von int mit 2 einen Überlauf verursacht.

HALT

Dieses Verhalten sollten Sie bei Ausdrücken immer im Auge behalten. Lösen können Sie das Problem, indem Sie einen der Operanden im Ausdruck zum erwarteten Typ konvertieren:

```
int number2 = int.MaxValue;
long result2 = number2 * (long)2;
Console.WriteLine(result2); // -2
```

3.8.2 Arithmetische Ausdrücke und Operatoren

Arithmetische Ausdrücke ergeben einen Wert

Arithmetische Ausdrücke ergeben einen Wert (eine Zahl, ein Datum, eine Zeichenkette), der in weiteren Ausdrücken oder in Zuweisungen verwendet werden kann. Der Ausdruck

```
1 + 1
```

ergibt z. B. den Wert 2 (wenn ich richtig gerechnet habe ...).

Arithmetische Ausdrücke verwenden die in Tabelle 3.13 beschriebenen Operatoren, von denen einige unär sind (d. h. nur einen Operanden besitzen) und einige binär (d. h. zwei Operanden besitzen).

Tabelle 3.13: Die arithmetischen Operatoren

Operator	Bedeutung
()	Klammern; Verschieben der Rechenpriorität
+	Addition zweier Operanden
++	Addition eines Operanden mit 1
-	Subtraktion zweier Operanden

Operator	Bedeutung
--	Subtraktion von 1 von einem Operanden
*	Multiplikation
/	Division
%	Modulo-Division

Tabelle 3.13: Die arithmetischen Operatoren (Forts.)

Klammern verschieben die Priorität

Klammern verwenden Sie immer dann, wenn Sie die Rechenpriorität verschieben wollen. In Computern gilt natürlich auch die in der Mathematik verwendete Rechenpriorität (»Punktrechnung vor Strichrechnung«). Der Ausdruck 1 + 2 * 2 ergibt demnach 5, weil erst die Multiplikation ausgerechnet wird.

Ich könnte jetzt an dieser Stelle eine Prioritätsliste der Operatoren liefern. Stattdessen gebe ich aber lediglich einen wichtigen Praxistipp:

HALT

*Verwenden Sie immer Klammern um die Priorität in Ausdrücken explizit festzulegen. Verlassen Sie sich nie auf die vordefinierte Priorität. Sie müssten ansonsten jeden Ausdruck hinsichtlich der Priorität der einzelnen Operatoren untersuchen um den Ausdruck interpretieren zu können. Sind Teilausdrücke hingegen geklammert, vermeiden Sie zum einen Fehler und machen den Ausdruck zum anderen wesentlich lesbarer. Auch wenn Sie beim Ausdruck 1 + 2 * 2 die Standard-Rechenpriorität verwenden wollen, setzen Sie Klammern: 1 + (2 * 2).*

/ führt u. U. zu einer Ganzzahldivision

Die Operatoren +, –, *, / und ^ müssen wohl nicht erläutert werden … Außer, dass der Operator / nicht unbedingt das vom Programmierer geplante Ergebnis liefert ☺. Das Problem liegt an der im vorigen Abschnitt beschriebenen Auswertung der Operanden eines Ausdrucks zur Ermittlung des Ergebnis-Typs. Wenn Sie zwei Integer-Werte teilen, wird das Ergebnis demnach als Ganzzahl ausgewertet. Deswegen ergeben 1 / 3 und 2 / 3 den Wert 0 und 5 / 3 den Wert 1:

Listing 3.46: Beispiele für Ganzzahldivisionen

```
Console.WriteLine("1 / 3 ergibt " + 1 / 3); // 0
Console.WriteLine("2 / 3 ergibt " + 2 / 3); // 0
Console.WriteLine("5 / 3 ergibt " + 5 / 3); // 1
```

Ganzzahl- und Restwertdivisionen

Der Divisionsoperator arbeitet also als Ganzzahldivisionsoperator bei Integer-Werten. Ist im Ausdruck eine Dezimalzahl enthalten, resultiert hingegen ein `float`-, `double`- oder `decimal`-Wert:

Listing 3.47: Beispiele für Divisionen mit Dezimalzahlen

```
Console.WriteLine("1 / 3F ergibt " + 1 / 3F); // 0,3333333
Console.WriteLine("2 / 3F ergibt " + 2 / 3F); // 0,6666666
Console.WriteLine("5 / 3F ergibt " + 5 / 3F); // 1,6666666
```

In vielen Fällen ist eine Ganzzahldivision aber auch erwünscht. Wenn Sie z. B. in einem Programm berechnen wollen, wie viele Lagerbehälter für eine bestimmte Anzahl zu lagernde Objekte benötigt werden, wenn fünf Objekte in ein Paket passen, verwenden Sie dazu eine Ganzzahldivision:

```
int numberOfObjects = 552;
int numberOfStorgageContainers = numberOfObjects / 5;
```

Das Ergebnis hier sind 110 Lagerbehälter, die vollständig gefüllt werden.

Dabei bleiben allerdings zu lagernde Objekte übrig (im Beispiel 2). Diese können Sie über eine einfache Subtraktion berechnen:

```
int remainingObjects = numberOfObjects -
   (numberOfStorgageContainers * 5);
```

Restwertdivisionen ergeben den Restwert

Sie können aber auch eine Restwertdivision über den Modulo-Operator ausführen:

```
int remainingObjects = numberOfObjects % 5;
```

Potenzierungen

Für Potenzierungen liefert C# keinen Operator. Potenzierungen können Sie über die `Pow`-Methode der `Math`-Klasse vornehmen. Die Potenzierung 2^5 berechnen Sie z. B. so:

```
double result = Math.Pow(2, 5);
```

String-Verkettungen

Über den Operator + können Sie auch Strings verketten:

Listing 3.48: String-Verkettung

```
string firstName = "Donald";
string lastName = "Duck";
Console.WriteLine(firstName + " " + lastName);
```

INFO

Wenn Sie reine String-Literale verketten, erzeugt der Compiler daraus einen einzigen String. Aus `"a" + "b"` *wird im CIL-Code* `"ab"`. *Deswegen können Sie größere String-Literale (z. B. SQL-Anweisungen) im Quellcode beruhigt in mehrere Zeilen auftrennen, ohne befürchten zu müssen, dass Ihr Programm aufgrund der 1/10000000000000 Sekunde, die eine String-Verkettung ansonsten benötigt, zu langsam wird (ok, das war jetzt ironisch an die Performance-Freaks gerichtet …). Bei Verkettungen mit String-Variablen oder bei dynamisch ausgeführten Verkettungen sollten Sie aus Performancegründen allerdings ab etwa fünf Verkettungen die* `StringBuilder`*-Klasse verwenden. Diese wird in Kapitel 8 behandelt.*

Die unären Operatoren ++ und – –

Die Operatoren ++ und `--` arbeiten unär. Die Anweisung

```
i++;
```

addiert z. B. den Wert 1 zu der Variablen i. Diese Operatoren können Sie vor (Präfix-Notation) oder hinter den Operanden setzen (Postfix-Notation). Ein Unterschied wird allerdings erst dann sichtbar, wenn die Operatoren in Ausdrücken verwendet werden:

Listing 3.49: Beispiele für die Präfix- und Postfix-Notation des unären Operators ++

```
int i = 1;
Console.WriteLine(++i); // 2, da i vor der Auswertung des Ausdrucks
                        // inkrementiert wird
Console.WriteLine(i);   // 2
i = 1;
```

```
Console.WriteLine(i++); // 1, da i erst nach der Auswertung des Ausdrucks
                        // inkrementiert wird
Console.WriteLine(i);   // 2
```

Die Präfix-Notation bewirkt, dass der Compiler zuerst den Wert des Operanden addiert bzw. subtrahiert und danach den Ausdruck auswertet. Der Ausdruck verwendet also den veränderten Wert. Die erste und die zweite `WriteLine`-Anweisung geben folglich den Wert 2 aus.

Wenn Sie die Postfix-Notation verwenden, wird der Wert des Operanden erst addiert bzw. subtrahiert, nachdem der Ausdruck vom Compiler ausgewertet wurde. Die dritte `WriteLine`-Anweisung gibt also 1 aus, die vierte dann wieder 2.

Ich halte diese, von C++ übernommene Auswertung der unären arithmetischen Operatoren für überflüssig, verwirrend und gefährlich. Sehr leicht können Sie in Ihren Programmen logische Fehler produzieren, wenn Sie diese Operatoren unbedacht einsetzen. Achten Sie immer darauf, dass die Präfix-Notation den Wert vor und die Postfix-Notation den Wert erst nach der Auswertung des Ausdrucks verändert, wenn Sie die Operatoren `++` *und* `--` *in Ausdrücken einsetzen. Idealerweise vermeiden Sie den Einsatz dieser Operatoren in Ausdrücken. Sie können die Addition/Subtraktion ja auch vor dem Ausdruck ausführen.*

3.8.3 Bitoperationen

C# enthält einige Operatoren, über die Sie die einzelnen Bits eines Integer- oder booleschen Typen bearbeiten können.

Tabelle 3.14: Die bitweisen Operatoren

Operator	Bedeutung
|	Or
&	And
^	Xor
~	Komplementär-Operation
<<	Linksverschiebung
>>	Rechtsverschiebung

Der Or-Operator (|)

Über den Or-Operator können Bits gesetzt werden

Mit Hilfe des Or-Operators können Sie einzelne Bits in einem Integer-Wert gezielt setzen, ohne die anderen Bits zu beeinflussen. Im Ergebnis eines Or-Ausdrucks sind alle Bits gesetzt, die entweder in dem einen *oder* im anderen Operanden gesetzt sind. Dual ausgedrückt ergibt $0101_2 \mid 1001_2$ z. B. den Wert 1101_2, weil die Bits 1 und 3 im ersten und die Bits 1 und 4 im zweiten Operanden gesetzt sind.

So können Sie z. B. gezielt das Bit 3 und 4 (das mit dem Wert 4 und das mit dem Wert 8) einer Variablen setzen, unabhängig vom bereits in der Variablen gespeicherten Wert:

Listing 3.50: Beispiel für das Setzen einzelner Bits

```
int i = 0;
i = i | 4; // Bit 3 setzen
i = i | 8; // Bit 4 setzen
// Alternativ: i = i | 4 | 8;
```

Eine solche Zuweisung kann noch etwas kürzer geschrieben werden (vgl. Seite 207):

```
i |= 4;
i |= 8;
```

Sie benötigen den |-Operator im Besonderen mit Aufzählungen, deren Konstanten die einzelnen Bits des Aufzählungswerts repräsentieren (siehe bei »Aufzählungen (Enumerationen)« ab Seite 192).

Der And-Operator (&)

Der And-Operator ermöglicht das Abfragen, ob Bits gesetzt sind

Im Ergebnis einer And-Operation sind nur die Bits gesetzt, die in dem einen *und* in dem anderen Operanden gesetzt sind. Im Dualsystem ausgedrückt ergibt z. B: 0101_2 And 0100_2 den Wert 0100_2, weil nur das dritte Bit (von rechts aus gesehen) in beiden Operanden gesetzt ist.

Den &-Operator benötigen Sie immer dann, wenn Sie überprüfen wollen, ob ein bestimmtes Bit in einem Integer-Wert gesetzt ist. Die folgende Anweisung überprüft z. B., ob das vierte Bit (das mit dem Wert 8) in der Variablen i gesetzt ist:

Listing 3.51: Beispiel für das Abfragen einzelner Bits

```
if ((i & 8) > 0)
{
   Console.WriteLine("Bit 4 ist gesetzt");
}
else
{
   Console.WriteLine("Bit 4 ist nicht gesetzt");
}
```

Genau wie den |-Operator benötigen Sie den &-Operator häufig bei Aufzählungen, deren Konstanten die einzelnen Bits des Aufzählungswerts repräsentieren, um die Konstanten der Aufzählung abzufragen.

Der XOr-Operator (^)

Der XOr-Operator ^ führt eine Exklusiv-Oder-Operation aus. Im Ergebnis ist ein Bit gesetzt, wenn dieses Bit in einem Operanden gesetzt und im anderen Operanden nicht gesetzt ist. Dual ausgedrückt ergibt z. B. 0101_2 ^ 1001_2 den Wert 1100_2. Mit diesem Operator können Sie einzelne Bits in einer Variablen gezielt umschalten. Gesetzte Bits werden so auf 0 gesetzt, Bits mit dem Wert 0 auf 1. Das folgende Beispiel schaltet das Bit 3 um:

```
i = i ^ 4;
```

Der Komplementär-Operator (~)

Mit dem Komplementär-Operator ~ können Sie das Einer-Komplement eines Wertes ermitteln. Dabei werden einfach alle Bits des Wertes umgekippt. Aus 1 wird 0 und aus 0 wird 1.

Links- und Rechtsschieben

Mit den Operatoren >> und << schieben Sie die Bits eines Operanden um eine anzugebende Anzahl Stellen nach links oder nach rechts. Die überstehenden Stellen werden dabei abgeschnitten, zur anderen Seite wird mit Nullen aufgefüllt. Wenn eine Byte-Variable den Wert 2 besitzt:

```
i = 2;
```

ist die duale Darstellung 00000010_2.

Linksschieben um eine Stelle:

```
i = i << 1;
```

ergibt die duale Zahl 00000100_2. Der Wert dieser Zahl ist jetzt 4. Linksschieben um eine Stelle entspricht einer Multiplikation mit 2. Rechtsschieben entspricht prinzipiell einer Ganzzahldivision durch 2. Die rechten Bits fallen beim Schieben heraus, weswegen immer eine ganze Zahl resultiert.

Das Ergebnis einer Links-Schiebeoperation kann auch einen größeren Datentyp besitzen als der Operand. Ist der Operand z. B. vom Typ `byte` und speichert den Wert 128 (10000000_2), ergibt das Schieben nach links um zwei Stellen keinen Überlauf, sondern den Wert 512 (1000000000_2).

Fragen Sie mich bloß nicht, wofür man diese Bit-Schiebereien benötigt. Ganz früher, als die Rechner noch langsam waren, war eine Multiplikation mit einer Zweier-Potenz wesentlich schneller, wenn diese durch Linksschieben implementiert wurde: Der Prozessor stellt für solche Operationen direkte Befehle zur Verfügung. Heute sehe ich im kommerziellen Bereich eigentlich keine Anwendung mehr. Möglicherweise benötigt die Mathematik solche Operationen. Aber davon habe ich keine Ahnung. Ich denke, die Schiebe-Operatoren sind hauptsächlich deswegen in C# enthalten, weil es diese auch in C++ gibt und schon immer gab.

3.8.4 Zuweisungen

Für Zuweisungen stellt C# nicht nur den Operator =, sondern auch noch einige spezielle Operatoren zur Verfügung.

Tabelle 3.15: Die Zuweisungsoperatoren

<table>
<tr><th>Operator</th><th>Bedeutung</th></tr>
<tr><td>=</td><td>einfache Zuweisung</td></tr>
<tr><td>+=</td><td>Additionszuweisung. i += 1 entspricht i = i + 1.</td></tr>
<tr><td>-=</td><td>Subtraktionszuweisung. I -= 1 entspricht i = i - 1.</td></tr>
<tr><td>*=</td><td>Multiplikationszuweisung. i *= 1 entspricht i = i * 1.</td></tr>
<tr><td>/=</td><td>Divisionszuweisung. i /= 1 entspricht i = i / 1.</td></tr>
<tr><td>%=</td><td>Modulo-Divisionszuweisung. i %= 1 entspricht i = i % 1.</td></tr>
<tr><td>&=</td><td>Bitweise-And-Zuweisung. i &= 1 entspricht i = i & 1.</td></tr>
<tr><td>|=</td><td>Bitweise-Or-Zuweisung. I |= 1 entspricht i = i | 1.</td></tr>
<tr><td>^=</td><td>Bitweise-XOr-Zuweisung. i ^= 1 entspricht i = i ^ 1.</td></tr>
</table>

Tabelle 3.15: Die Zuweisungsoperatoren (Forts.)

Operator	Bedeutung
<<=	Linksverschiebungszuweisung. `i <<= 1` entspricht `i = i << 1`.
>>=	Rechtsverschiebungszuweisung. `i >>= 1` entspricht `i = i >> 1`.

Eine einfache Zuweisung sieht z. B. so aus:

```
int i = 1;
```

C# erlaubt auch Mehrfachzuweisungen

C# erlaubt auch Mehrfachzuweisungen:

```
int j;
i = j  = 11;
```

i und *j* besitzen nach der Ausführung dieser Anweisung den Wert 11.

Die erweiterten Zuweisungsoperatoren erlauben die Zuweisung eines Ausdrucks bzw. Werts, den Sie gleich noch über eine arithmetische Operation mit den Operanden berechnen. Die Anweisung

```
i += 10;
```

addiert beispielsweise den Wert von j auf die Variable i und bedeutet soviel wie

```
i = i + 10;
```

3.8.5 Vergleiche

Vergleiche ergeben true oder false

Für Vergleiche bietet C# die üblichen Operatoren. Ein Vergleichsausdruck ergibt immer den booleschen Wert `true` (wenn der Vergleich wahr ist) oder `false` (wenn der Vergleich falsch ist).

Tabelle 3.16: Die Vergleichsoperatoren

Operator	Beschreibung
==	Gleichheit
!=	Ungleichheit
<	Kleiner
>	Größer
<=	Kleiner/Gleich
>=	Größer/Gleich

Lediglich die Operatoren == und != können Sie auf alle .NET-Typen anwenden.

INFO

Dabei müssen Sie aber beachten, dass diese Operatoren nur bei Werttypen, dem Typ `String` *und bei Referenztypen, die diese Operatoren speziell überladen haben (was das ist und wie das geht, zeigt Kapitel 5) einen Vergleich der gespeicherten Werte ergeben. Vergleichen Sie Referenztypen, vergleichen Sie prinzipiell daraufhin, ob diese dasselbe Objekt referenzieren.*

Vergleich von Referenztypen

Referenzieren zwei Variablen unterschiedliche Referenztyp-Instanzen, die aber dieselben Werte speichern, ergibt ein Vergleich mit == prinzipiell `false`. Da die Vergleichsoperatoren für einzelne Typen aber auch überladen sein können, kann es auch sein, dass ein Vergleich von Referenztypen doch die gespeicherten Werte miteinander vergleicht. Im .NET Framework ist das zwar laut der Microsoft-Dokumentation nur für den (Referenz-)Typ `String` der Fall. Für neue Referenztypen, besonders aus Assemblys externer Hersteller, können die Vergleichsoperatoren aber durchaus überschrieben sein. Lesen Sie also die Dokumentation zu den verwendeten Typen oder probieren Sie den Vergleich einfach aus. Oder:

Verwenden Sie dazu die von `Object` *geerbte* `Equals`*-Methode, wenn Sie den Inhalt zweier Referenztyp-Instanzen miteinander vergleichen wollen. Vorausgesetzt, die verwendeten Typen überschreiben diese Methode (was zwar bei den .NET-Typen, aber nicht unbedingt auch bei externen Typen der Fall ist), erhalten Sie einen Vergleich der gespeicherten Werte. Einige Typen wie* `String` *besitzen daneben eine* `CompareTo`*-Methode, über die Sie herausfinden können, ob ein Objekt kleiner, gleich oder größer ist als ein anderes. Die Rückgabe dieser Methode ist kleiner Null, wenn das Objekt kleiner ist, Null, wenn beide Objekte gleich groß sind, und größer Null, wenn das Objekt größer ist als das übergebene.*

Wollen Sie hingegen bei zwei Referenzen explizit überprüfen, ob diese dasselbe Objekt referenzieren, auch wenn der Typ die Vergleichsoperatoren überschreibt, setzen Sie die statische `ReferenceEquals`*-Methode der* `Object`*-Klasse ein.*

3

Vergleich von Strings

= = unterscheidet Groß- und Kleinschreibung

Beim Vergleich von Strings berücksichtigt C# Groß- und Kleinschreibung. Der Vergleich `"a" == "A"` ergibt also `false`. Eine einfache Lösung dieses Problems ist, Strings einfach vor dem Vergleich in Klein- oder Großschrift umzuwandeln:

Listing 3.52: Vergleich von Strings

```
s1 = "ZAPHOD";
s2 = "zaphod";
if (s1.ToLower() == s2.ToLower())
{
   Console.WriteLine("Beide Strings sind gleich");
}
```

Alternativ können Sie auch die `Compare`-Methode der `String`-Klasse verwenden:

Listing 3.53: Vergleich von Strings über String.Compare

```
string s1 = "ZAPHOD";
string s2 = "zaphod";
if (String.Compare(s1, s2, true) == 0)
{
   Console.WriteLine("Beide Strings sind gleich");
}
```

Am dritten Argument geben Sie an, ob Sie Groß- und Kleinschreibung ignorieren wollen.

Die Operatoren >, <, >= und <= können Sie nur auf numerische Typen, Aufzählungen (Enums) und spezielle Typen anwenden, die diese Operatoren überschreiben. Mit

Strings können Sie diese Operatoren leider nicht einsetzen. Warum Microsoft diese Operatoren nicht für die `String`-Klasse implementiert hat, ist absolut unverständlich.

Strings können mit CompareTo verglichen werden

Zum Vergleich zweier Strings können Sie aber stattdessen die `CompareTo`-Methode der `String`-Klasse verwenden. Diese Methode besitzen neben `String` viele Typen, die Vergleiche zulassen. Über die Rückgabe können Sie herausfinden, ob der Typ kleiner (Rückgabe kleiner Null), gleich (Rückgabe gleich Null) oder größer (Rückgabe größer Null) ist als ein anderer. So können Sie z. B zwei Strings daraufhin vergleichen, ob ein String kleiner, gleich oder größer ist als der andere:

Listing 3.54: Vergleich von Strings über CompareTo

```
String s1 = "abc";
String s2 = "abd";
int compareResult = s1.CompareTo(s2);
if (compareResult < 0)
{
   Console.WriteLine(s1 + " ist kleiner " + s2);
}
else if (compareResult == 0)
{
   Console.WriteLine(s1 + " ist gleich " + s2);
}
else
{
   Console.WriteLine(s1 + " ist größer " + s2);
}
```

Beim Vergleich von Strings wird per Voreinstellung ein Wort-Vergleich nach Regeln vorgenommen, die die aktuelle Kultur bestimmt. In der deutschen Kultur erhalten alle Zeichen z. B. einen aufsteigenden Wert, der der in Deutschland verwendeten Sortierung entspricht. Die Groß- und Kleinschreibung von Buchstaben spielt dabei zunächst keine Rolle. Beim Vergleich wird dem ersten Zeichen die höchste und folgenden Zeichen eine absteigende Priorität zugewiesen. Der String wird Zeichen für Zeichen verglichen. Wird dabei ein Zeichen mit einem höheren oder niedrigeren Wert gefunden, wird der Vergleich beendet. "abc" ist deshalb kleiner als "abd", weil das dritte Zeichen im zweiten String einen höheren Wert besitzt als das dritte Zeichen im ersten. Ist ein String länger als der andere, werden die zusätzlichen Zeichen dann berücksichtigt, wenn der Vergleich bis dahin gleiche Strings ergeben hat. Der längere String ist dann der größere. "abcd" ist deswegen größer als "abc". Enthalten beide String gleiche Zeichen, wird die Groß- und Kleinschreibung berücksichtigt. Großgeschriebene Zeichen werden dann höher bewertet. Deswegen ist "ABC" größer als "abc". Weil aber der eigentliche Zeichenwert höher bewertet wird, sind "abcd" oder "bcd" wieder größer als "ABC".

Beim Vergleich von Strings, die Zahl- oder Datumswerte speichern, führt dieser Zeichen-für-Zeichen-Vergleich in vielen Fällen zu Problemen. Der String "20" ist z. B. größer als der String "100". Das ist auch logisch, denn das Zeichen »2« besitzt in der deutschen Sortierung einen höheren Wert als das Zeichen »1«. Dieses Verhalten müssen Sie beim Vergleich von Strings immer im Auge behalten. Besonders kritisch wird dies, wenn Sie Listen mit Strings, die Zahl- oder Datumswerte enthalten, sortieren. Die Sortierung entspricht bei Zahlwerten nur dann der normalen Zahl-Sortierung, wenn die einzelnen Strings gleich lang sind und das Komma (wenn vorhanden) an

derselben Stelle steht. Sie können das Sortierungsproblem also lösen, indem Sie die Strings links vor dem Komma auf dieselbe Länge bringen (evtl., indem Sie Nullen vorne hinzufügen). Bei Datumswerten wird das Ganze sehr schwierig, da diese ja eigentlich in der Reihenfolge Jahr, Monat und Tag sortiert werden. Besser wäre allerdings, nicht Strings, sondern die eigentlichen Typen zu verwalten (also z. B. `int`, `double` *oder* `DateTime`*), diese zu vergleichen bzw. zu sortieren und für die Anzeige in ein String zu konvertieren. Auf Sortierungen (von Auflistungen) gehe ich in Kapitel 7 noch näher ein.*

3.8.6 Logische Operatoren

Bei Vergleichen von Typen können Sie mehrere Vergleichsausdrücke über die logischen Operatoren von C# verknüpfen. Der Ausdruck

Ausdruck1 `&&` *Ausdruck2*

ergibt z. B. nur `true`, wenn beide Teilausdrücke `true` ergeben. Tabelle 3.17 zeigt die logischen Operatoren von C#.

Tabelle 3.17: Die logischen Operatoren

Operator	Beschreibung
`&&`	Logisches And. Der rechte und der linke Ausdruck müssen `true` ergeben, damit der Gesamtausdruck `true` ergibt.
`||`	Logisches Or. Wenn einer der Ausdrücke `true` ergibt, resultiert `true` für den Gesamtausdruck.
`!`	Logisches Not. Der Ausdruck, der rechts von diesem Operator steht, wird negiert. Aus `true` wird `false` und aus `false` wird `true`.

HALT

Wie in allen anderen Sprachen auch, besitzen diese Operatoren eine Priorität. `!` *wird vor* `&&` *ausgewertet und* `&&` *vor* `||`*. Zur Sicherheit sollten Sie kombinierte logische Operationen immer klammern. Das gilt besonders dann, wenn Sie den* `!`*-Operator verwenden, der die höchste Priorität besitzt.*

Der folgende Quellcode überprüft, ob jetzt gerade Samstag oder Sonntag und nach 12 Uhr ist:

Listing 3.55: Überprüfung darauf, ob heute Samstag oder Sonntag und nach 12:00 ist

```
DateTime date = DateTime.Now;
if (((date.DayOfWeek == DayOfWeek.Saturday) ||
   (date.DayOfWeek == DayOfWeek.Sunday)) &&
   (date.TimeOfDay.Hours > 12))
{
   Console.WriteLine("Jetzt ist Samstag oder " +
      "Sonntag nach 12 Uhr");
}
else
{
   Console.WriteLine("Jetzt ist nicht Samstag " +
      "oder Sonntag nach 12 Uhr");
}
```

C# verwendet einen logischen Kurzschluss

C# wertet zusammengesetzte logische Ausdrücke von links nach rechts aus. Ergibt ein links stehender Ausdruck bereits, dass das Gesamtergebnis auf jeden Fall `false` ergibt, werden die rechts stehenden Ausdrücke nicht weiter ausgewertet. Dieses Kurzschluss-Verhalten ist für den Fall, dass (rechts stehende) Teilausdrücke nur dann ausgewertet werden sollen, wenn (links stehende) andere Teilausdrücke zum Erfolg führen, enorm hilfreich. Bei der Ermittlung eines Teilstrings aus einem String darf beispielsweise die Anzahl der extrahierten Zeichen die Länge des Strings nicht überschreiten:

```
string name = "Ford";
if (name.Substring(0, 6) == "Zaphod") // erzeugt eine Ausnahme, weil der
{                                     // String nicht genügend Zeichen
}                                     // speichert
```

Wenn Sie vor dem Extrahieren überprüfen, ob der String lang genug ist, wird keine Ausnahme erzeugt, weil der rechte Teilausdruck nicht mehr ausgewertet wird, wenn der String zu kurz ist:

Listing 3.56: Sinnvoller Einsatz des Kurzschlusses bei logischen Ausdrücken

```
string name = "Ford";
if (name.Length >= 6 && name.Substring(0, 6) == "Zaphod")
{
   Console.WriteLine("Hallo Zaphod");
}
```

HALT

Das Kurzschluss-Verhalten bei der Auswertung logischer Ausdrücke kann zu Problemen führen, wenn ein rechts stehender Teilausdruck den Aufruf einer Methode beinhaltet. Ergibt der links stehende Teilausdruck bereits, dass das Gesamtergebnis `false` *ergibt, wird damit die im rechten Ausdruck untergebrachte Methode nicht aufgerufen. Um solche Probleme zu vermeiden, sollten Sie in logischen Ausdrücken Methoden nicht oder nur im linken Teilausdruck aufrufen.*

3.8.7 ?:, ??, typeof, is, as, sizeof und =>

C# besitzt neben den Standardoperatoren noch ein paar spezielle. Über den Bedingungsoperator `?:` können Sie in vielen Fällen auf Abfragen mit `if` verzichten. `??` ermöglicht die Umwandlung von `null` in andere Werte. `typeof` ermittelt den Typ eines Objekts, was für erweiterte Techniken, die mit Reflektion arbeiten, wichtig ist. `is` vergleicht ein Objekt daraufhin, ob dieses einem angegebenen Typen entspricht. `as` ermöglicht die Umwandlung eines Referenztypen und, falls dieser nicht umwandelbar ist, die Rückgabe von `null`. `sizeof` ermittelt die Speichergröße eines Werttypen, was für den Aufruf von API-Funktionen (Funktionen, die in klassischen DLL-Dateien enthalten sind) oder bei der Arbeit mit COM-Objekten wichtig sein kann (auf die .NET-Programmierer nach Möglichkeit verzichten). Der Lambda-Operator => wird schließlich mit Lambda-Ausdrücken eingesetzt. Lambda-Ausdrücke werden gesondert in Kapitel 6 behandelt.

Der Bedingungsoperator ?:

?: ersetzt einfache Anfragen

Über den Bedingungsoperator `?:` können Sie einen Ausdruck schreiben, der eine Bedingung überprüft und der jeweils einen anderen Wert ergibt, wenn die Bedingung wahr oder falsch wird:

```
Bedingung ? Ausdruck1 : Ausdruck2
```

Der Ausdruck `number != 0d ? Math.Sin(number) : 1d` überprüft z. B., ob *number* ungleich 0 ist, und gibt in diesem Fall den Sinus von *number* zurück, ansonsten 1.

Das Ergebnis eines solchen Ausdrucks können Sie in weiteren Ausdrücken verwenden oder einer Variablen zuweisen:

```
double number = 0;
double result = number != 0d ? Math.Sin(number) : 1d;
```

Übersichtlicher wird dies, wenn Sie den Ausdruck klammern:

```
double number = 0;
double result = (number != 0d ? Math.Sin(number) : 1d);
```

Über den Bedingungsoperator können Sie in vielen Fällen auf `if`-Abfragen verzichten. Die `if`-Abfrage, die der obigen Zuweisung entspricht, wäre z. B. deutlich aufwändiger:

```
double result;
double number = 0;
if (number != 0)
{
   result = Math.Sin(number);
}
else
{
   result = 1;
}
```

Der Bedingungsoperator wird allerdings wesentlich schneller ausgeführt und ist einfacher anzuwenden. Richtig interessant wird der Bedingungsoperator, wenn Sie Teilausdrücke damit in anderen Ausdrücken einsetzen oder als Argument an eine Methode übergeben. Sie sollten jedoch beachten, dass Ihr Programmcode bei der Verwendung des Bedingungsoperators nicht unbedingt lesbarer wird.

Der Operator ??

Über den Operator `??` können Sie den Wert `null` in einen beliebigen, zum Ausdruck passenden Wert umwandeln. Der Ausdruck `x ?? y` ergibt *x*, wenn *x* nicht `null` ist, und *y*, wenn *x* `null` ist.

`??` ist ziemlich hilfreich, wenn Sie mit Referenztypen oder Nullables arbeiten und Sie bei der Speicherung von `null` einen anderen Wert verwenden wollen.

Listing 3.57: Der ??-Operator

```
string name = null;
...
Console.WriteLine("Name: " + (name ?? "Nicht angegeben"));
```

TIPP

In vielen Fällen haben Methoden (meist externer Programmierer) Probleme damit, dass an einem `String`*-Argument eine* `null`*-Referenz übergeben wird. Diese Methoden generieren dann eine* `NullReferenceException`*. Mit* `??` *können Sie für einen String statt* `null` *einen Leerstring übergeben. Ich demonstriere das an der* `WriteLine`*-Methode der* `Console`*-Klasse, die allerdings mit* `null`*-Werten keine Probleme hat:*

```
string name = null;
...
Console.WriteLine(name ?? String.Empty);
```

Der typeof-Operator

typeof ermittelt Informationen zu einem Typ

Über den typeof-Operator können Sie Informationen zu einem Typ ermitteln:

```
typeof(Typ)
```

typeof ergibt ein Objekt der Klasse Type. Ein solches Objekt wird auch von der GetType-Methode zurückgegeben, die jede Instanz eines Typs besitzt. typeof wird aber nicht auf Instanzen, sondern auf die Typen direkt angewendet.

Die Type-Klasse besitzt eine große Anzahl an Eigenschaften und Methoden, über die Sie Informationen zum Typ ermitteln können. Die Name-Eigenschaft gibt z. B. den Namen des Typs zurück. Wenn Sie beispielsweise eine Klasse besitzen, die die Daten einer Person speichert:

```
class Person
{
   public string FirstName;
   public string LastName;
}
```

können Sie mit typeof Informationen zu dieser Klasse ermitteln:

Listing 3.58: Einsatz von typeof zur Ermittlung von Informationen zu einem Typ

```
Type t = typeof(Person);
Console.WriteLine("Name: " + t.Name);
Console.WriteLine("Assembly: "+ t.Assembly.FullName);
Console.WriteLine("Basistyp: "+ t.BaseType.Name);
Console.WriteLine("Voller Name: "+ t.FullName);
Console.WriteLine("Typ ist " +
   (t.IsClass ? "eine" : "keine" ) + " Klasse");
Console.WriteLine("Typ ist " +
   (t.IsArray ? "ein" : "kein") + " Array");
Console.WriteLine("Typ ist " +
   (t.IsEnum ? "eine" : "keine") + " Aufzählung");
Console.WriteLine("Typ ist " +
   (t.IsInterface ? "eine" : "keine") + " Schnittstelle");
```

Dieser Vorgang wird übrigens als Reflektion (Reflection) bezeichnet. Über Reflektion können Sie u. a. jederzeit Informationen zu allen Typen erhalten, auf die Ihr Programm Zugriff hat. .NET stellt Ihnen dazu Klassen zur Verfügung, die Sie im Namensraum System.Reflection finden. Reflektion wird in Kapitel 22 behandelt.

Der is-Operator

is ermittelt, ob ein Ausdruck einem bestimmten Typ angehört

Über den is-Operator können Sie herausfinden, ob ein Ausdruck einen bestimmten Typ besitzt:

```
Ausdruck is Typ
```

Üblicherweise überprüfen Sie damit Variablen. Sinn macht das u. a., wenn Sie Daten mit object-Referenzen verwalten (bei allen anderen Typen kennen Sie und der Compiler den gespeicherten Typ):

Listing 3.59: Überprüfung eines Objekts daraufhin, ob dieses einen bestimmten Typ besitzt

```
object o = (int)10;
if (o is int)
{
   Console.WriteLine("o ist ein int.");
}
else
```

```
{
   Console.WriteLine("o ist kein int.");
}
```

is wird in der Praxis jedoch meist für die Überprüfung verwendet, ob ein Typ von einem anderen Typ abgeleitet ist oder ob dieser eine bestimmte Schnittstelle implementiert. Dazu vergleichen Sie einfach mit dem Basistyp oder dem Typ der Schnittstelle. Vererbung und Schnittstellen werden in Kapitel 5 behandelt.

Der as-Operator

Der as-Operator verbindet die Typumwandlung eines Referenztypen mit einer Abfrage, ob das umzuwandelnde Objekt dem Typ überhaupt entspricht. Mit

```
Variable = Referenztyp as Typ
```

können Sie einer Variablen eine Referenz auf den Typ zuweisen, falls der Referenztyp dem Typ entspricht. Entspricht der Referenztyp dem Typ nicht, wird automatisch null zugewiesen.

as wird wie is meist eingesetzt, um zu überprüfen, ob Objekte von einem Basistyp abgeleitet sind oder eine erwartete Schnittstelle implementieren. Der Vergleichstyp ist dann der Basistyp bzw. die Schnittstelle. as besitzt den (selten genutzten) Vorteil, dass automatisch konvertiert wird, wenn das Objekt vom Basistyp abgeleitet ist bzw. die Schnittstelle implementiert, und null zugewiesen wird, wenn das Objekt nicht vom Basistyp abgeleitet ist bzw. die Schnittstelle nicht implementiert. In einigen Programmiersituationen ist die Verwendung von as deswegen einfacher als die von is.

Aus Mangel an einem sinnvollen Beispiel, das nicht zu viel Wissen (aus Kapitel 5) voraussetzt, überprüft Listing 3.60, ob eine Instanz der StreamReader-Klasse (aus dem Namensraum System.IO) von TextReader abgeleitet ist und die Schnittstelle IDisposable implementiert (was beides der Fall ist).

Listing 3.60: Den as-Operator mit Basisklassen und Schnittstellen einsetzen

```
StreamReader streamReader = new StreamReader("C:\\boot.ini");

// Den StreamReader in den Basistyp TextReader umwandeln
TextReader textReader = streamReader as TextReader;
if (textReader != null)
{
   Console.WriteLine(textReader.ReadToEnd());
}

// Den StreamReader in die Schnittstelle IDisposable umwandeln
IDisposable disposable = streamReader as IDisposable;
if (disposable != null)
{
   disposable.Dispose();
}
```

Das Beispiel ist ziemlich sinnlos, weil Sie die verwendeten Methoden auch direkt über die StreamReader-Instanz aufrufen können. Der as-Operator wird in der Praxis wie is meist in Zusammenhang mit Polymorphismus (Kapitel 5) eingesetzt.

Der sizeof-Operator

sizeof ermittelt die Größe eines Typen

Über den sizeof-Operator können Sie die Größe eines Werttypen ermitteln:

```
sizeof(Typ)
```

Dieser Operator kann nur in einem unsicheren Bereich des Quellcodes verwendet werden:

Listing 3.61: Verwendung des sizeof-Operators

```
unsafe
{
   Console.WriteLine("int besitzt die Größe: " + sizeof(int));
   Console.WriteLine("DateTime besitzt die Größe: " + sizeof(DateTime));
}
```

Der Grund für den Zwang zur Verwendung in einem unsicheren Block ist, dass dieser Operator bereits zur Kompilierungszeit ausgewertet wird und dass die Größe von Typen sich in neueren Versionen des .NET Framework ändern kann. Der `DateTime`-Typ kann z. B. in neuen Versionen zwölf statt acht Byte groß sein. Ein altes Programm würde unter einer neuen .NET Framework-Version dann fehlerhaft ausgeführt werden.

Verwenden Sie diesen Operator also idealerweise erst gar nicht. Wenn Sie nur mit den Klassen des .NET Framework arbeiten, benötigen Sie `sizeof` *nicht. Nur wenn Sie (Windows-)API-Funktionen direkt aufrufen (was lediglich zur Lösung sehr seltener Programmier-Probleme notwendig ist), müssen Sie manchen dieser Funktionen Strukturen und deren Größe übergeben. Dann müssen Sie normalerweise* `sizeof` *einsetzen um die Größe zu ermitteln.*

3.9 Verzweigungen und Schleifen

C# kennt natürlich auch die gängigen Verzweigungen und Schleifen. Bevor ich diese beschreibe, ein Tipp, der eventuelle Probleme vermeidet:

Wenn Sie versehentlich eine Endlosschleife produzieren (eine Schleife, die nie beendet wird) und das Programm in Visual Studio ausführen, scheint Ihre Anwendung nicht mehr zu reagieren. Das liegt daran, dass der Prozessor in diesem Fall sehr stark ausgelastet ist und Ihre Anwendung keine Möglichkeit mehr hat, auf Benutzereingaben zu reagieren. Statt das Programm über den Task-Manager »abzuschießen«, sollten Sie die Anwendung in Visual Studio lieber anhalten. Dazu betätigen Sie `Strg` *+* `ALT` *+* `Pause`*, den Schalter Alles unterbrechen in der Symbolleiste oder den entsprechenden Befehl im* DEBUG*-Menü. Das Unterbrechen hat den Vorteil, dass Sie erkennen, in welcher Schleife Ihr Programm hängt.*

3.9.1 Die if-Verzweigung

if verzweigt ein Programm

Über die `if`-Verzweigung können Sie ein Programm bedingungsabhängig verzweigen. Die `if`-Verzweigung besitzt die folgende Syntax:

```
if (Bedingung)
   Anweisungsblock1
[else
   Anweisungsblock2]
```

Wenn die Bedingung wahr wird, verzweigt das Programm in den ersten Anweisungsblock. Wenn Sie den optionalen `else`-Block angeben, wird dieser ausgeführt, wenn die Bedingung `false` ergibt.

Das folgende Beispiel überprüft, ob der Wert einer Variablen größer ist als der einer anderen:

Listing 3.62: if-Verzweigung mit else-Block ohne Blockklammern

```
int number1 = 1;
int number2 = 2;
if (number1 > number2)
   Console.WriteLine("number1 ist größer als number2.");
else
   Console.WriteLine("number1 ist kleiner oder gleich number2.");
```

Wenn Sie mehrere Anweisungen in einem Block unterbringen wollen, müssen Sie diese in geschweifte Klammern einfügen, aber das wissen Sie ja bereits. Und wie Sie auch bereits wissen, sollten Sie der besseren Lesbarkeit Ihrer Programme zuliebe immer Blockklammern verwenden:

Listing 3.63: Die if-Verzweigung mit else-Block mit Blockklammern

```
int number1 = 1;
int number2 = 2;
if (number1 > number2)
{
   Console.WriteLine(number1 + " ist größer als " + number2);
}
else
{
   Console.WriteLine(number1 + " ist kleiner oder gleich " + number2);
}
```

Die Bedingung kann natürlich mit den logischen Operatoren auch komplex gestaltet werden. Das folgende Beispiel überprüft, ob *number1* größer als 1 und kleiner als 100 ist:

Listing 3.64: if-Verzweigung mit komplexer Bedingung

```
if (number1 > 1 && number1 < 100)
{
   Console.WriteLine(number1 + " ist größer als 1 und kleiner als 100");
}
else
{
   Console.WriteLine(number1 +
      " ist kleiner/gleich 1 oder größer/gleich 100");
}
```

if mit mehreren Fallüberprüfungen

if kann mehrere Fallüberprüfungen besitzen

Mit der `if`-Verzweigung können Sie natürlich auch mehrere Fälle überprüfen (auch wenn dazu häufig die `switch`-Verzweigung besser geeignet ist). Im Gegensatz zur `switch`-Verzweigung, die im nächsten Abschnitt behandelt wird, können Sie mit `if` beliebige, auch unterschiedliche Vergleiche verwenden. C# stellt dazu aber kein spezielles Schlüsselwort zur Verfügung (wie das `ElseIf` von Visual Basic). Sie können `if`-Verzweigungen allerdings schachteln:

Listing 3.65: Geschachtelte if-Abfrage, die überprüft, ob heute Weihnachten, Samstag, Sonntag oder ein anderer Tag ist

```
if (DateTime.Now.Day == 24 && DateTime.Now.Month == 12)
{
   Console.WriteLine("Heute ist Weihnachten");
}
else
```

```
{
   if (DateTime.Now.DayOfWeek == DayOfWeek.Saturday)
   {
      Console.WriteLine("Heute ist Samstag");
   }
   else
   {
      if (DateTime.Now.DayOfWeek == DayOfWeek.Sunday)
      {
         Console.WriteLine("Heute ist Sonntag");
      }
      else
      {
         Console.WriteLine("Heute ist weder Weihnachten " +
            "noch Samstag oder Sonntag");
      }
   }
}
```

Zur besseren Übersicht sollten Sie solche Abfragen allerdings (ausnahmsweise) ohne `else`-Blöcke (außer im letzten `else`-Block) und nicht eingerückt schreiben:

Listing 3.66: Besser lesbare, geschachtelte if-Abfrage mit mehreren Bedingungen

```
if (DateTime.Now.Day == 24 && DateTime.Now.Month == 12)
{
   Console.WriteLine("Heute ist Weihnachten");
}
else if (DateTime.Now.DayOfWeek == DayOfWeek.Saturday)
{
   Console.WriteLine("Heute ist Samstag");
}
else if (DateTime.Now.DayOfWeek == DayOfWeek.Sunday)
{
   Console.WriteLine("Heute ist Sonntag");
}
else
{
   Console.WriteLine("Heute ist weder Weihnachten " +
      "noch Samstag oder Sonntag");
}
```

3.9.2 Die switch-Verzweigung

switch erlaubt beliebig viele Fälle

Die `switch`-Verzweigung kann einen Ausdruck mit mehreren Werten vergleichen. Jeder erwartete Ergebniswert wird in einem `case`[6]-Block abgefangen, der ausgeführt wird, wenn der zu prüfende Ausdruck diesen Wert ergibt. Der optionale `default`-Block behandelt alle anderen Fälle.

```
switch (Prüfausdruck)
{
   case Wert1:
      /* Anweisungen für den Fall, dass der Prüfausdruck den
         Wert 1 ergibt */
      Sprunganweisung;

   [case Wert2:
      /* Anweisungen für den Fall, dass der Prüfausdruck den
         Wert 2 ergibt */
      Sprunganweisung;]
```

6 engl. für »Fall«

```
   [...]

   [default:
      /* Anweisungen, die ausgeführt werden sollen,
         wenn keiner der Fälle eingetreten ist */
      Sprunganweisung;]
}
```

Der Prüfausdruck kann ein beliebiger Ausdruck sein, der einen ordinalen Wert oder einen String ergibt. Ein ordinaler Wert ist eine Ganzzahl, ein boolescher oder ein Aufzählungswert.

Die in den einzelnen Fallblöcken angegebenen Werte müssen konstante Werte sein (keine Ausdrücke oder Variablen), deren Typ zum Prüfausdruck passt. Die am Ende eines Fallblocks angegebene Sprunganweisung ist obligatorisch. Damit geben Sie an, wohin das Programm nach der Abarbeitung eines Falls springen soll. In den meisten Fällen verwenden Sie hier die `break`-Anweisung. Diese bewirkt, dass das Programm aus dem `switch`-Block herausspringt und an der ersten Anweisung hinter diesem weiter abgearbeitet wird.

Das folgende Beispiel lässt den Anwender eine Zahl eingeben und wertet diese dann aus:

Listing 3.67: Standard-switch-Verzweigung mit break-Anweisungen

```
Console.Write("Ihre Lieblingszahl: ");
int number = Convert.ToInt32(Console.ReadLine());

switch (number)
{
   case 7:
      Console.WriteLine("7 ist eine gute Zahl.");
      break;

   case 42:
      Console.WriteLine("Cool. Trinken Sie einen " +
         "pangalaktischen Donnergurgler.");
      break;

   case 3:
      Console.WriteLine("3 ist OK.");
      break;

   default:
      Console.WriteLine("Denken Sie noch einmal darüber nach.");
      break;
}
```

Der Compiler geht bei der Auswertung von `switch`-Verzweigungen so vor, dass die einzelnen `case`-Blöcke von oben nach unten der Reihe nach überprüft werden. Entspricht einer der Vergleichswerte dem Ergebnis des Prüfausdrucks, verzweigt das Programm in den Block. Die normalerweise angebrachte Sprunganweisung `break` am Ende des Blocks bewirkt nach der Abarbeitung der im Block enthaltenen Anweisungen einen Sprung zum Ende der `switch`-Verzweigung. Weitere Fälle werden nicht mehr überprüft. Tritt keiner der angegebenen Fälle ein, wird der `default`-Block ausgeführt, falls dieser vorhanden ist.

Wenn der Prüfausdruck einen Aufzählungswert ergibt, können Sie sehr gut den `switch`*-Codeausschnitt einsetzen um automatisch* `case`*-Blöcke für alle Werte der Aufzählung einzufügen.*

1. Schreiben Sie dazu »sw« bzw. so viele Zeichen, dass IntelliSense den `switch`-Codeausschnitt markiert.
2. Betätigen Sie dann die [Tab]-Taste um den Codeausschnitt einzufügen.
3. Tragen Sie im grün markierten Bereich innerhalb der Klammern den Ausdruck ein, der den Aufzählungswert zurückgibt. Sie können den Ausdruck auch aus der Zwischenablage einfügen und sogar nach dem Einfügen des `switch`-Codeausschnitts zuerst einmal von einer anderen Stelle im Quellcode in die Zwischenablage kopieren. Solange der Bereich innerhalb der Klammern grün markiert bleibt (bis Sie den Quellcode ändern oder die [ESC]- oder [↵]-Taste betätigen), funktioniert der Codeausschnitt weiterhin.
4. Sobald Sie den grün markierten Bereich innerhalb der Klammern verlassen oder wenn Sie die [↵]-Taste betätigen, fügt Visual Studio `case`-Blöcke für alle Werte der Aufzählung und einen `default`-Block ein.

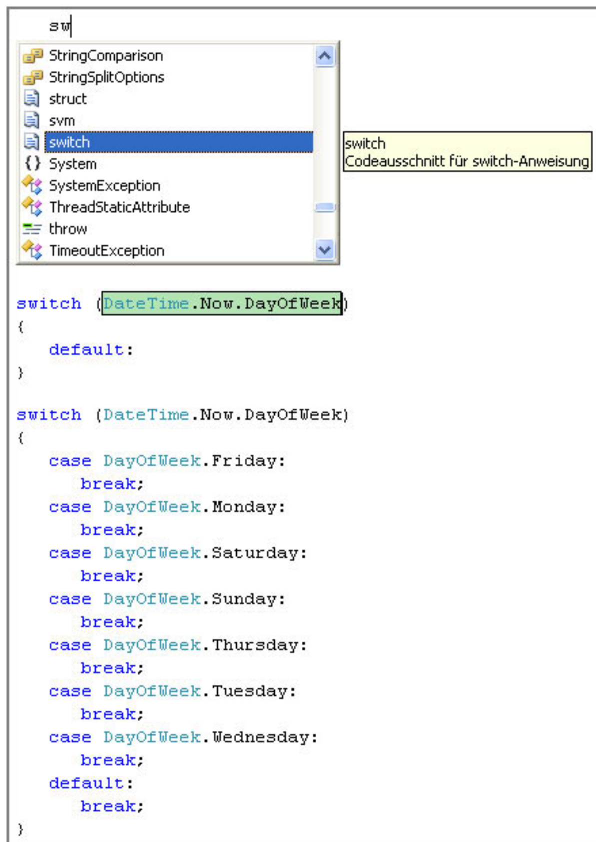


Abbildung 3.9: Die drei Stufen des switch-Codeausschnitts

case-Blöcke

Variablen können in case-Blöcken problematisch sein

Wenn Sie in den einzelnen `case`-Blöcken mehrere Anweisungen unterbringen, müssen Sie diese nicht in geschweifte Klammern einfügen. Das wird aber problematisch, wenn Sie in einem `case`-Block eine Variable deklarieren, deren Name Sie in einem anderen Block oder außerhalb der `switch`-Anweisung noch einmal benötigen:

Listing 3.68: Nicht mögliche switch-Anweisung mit der Deklaration mehrerer Variablen in den einzelnen Blöcken, die denselben Namen tragen

```
switch (number)
{
   case 7:
      string message = "7 ist eine gute Zahl.";
      Console.WriteLine(message);
      break;

   case 42:
      // Compilerfehler »Eine lokale Variable mit dem Namen "message" ist
      // bereits in diesem Bereich definiert«
      string message = "Cool. Trinken Sie einen " +
         "pangalaktischen Donnergurgler.";
      Console.WriteLine(message);
      break;

   case 3:
      // Compilerfehler »Eine lokale Variable mit dem Namen "message" ist
      // bereits in diesem Bereich definiert«
      string message = "3 ist OK.";
      Console.WriteLine(message);
      break;

   default:
      // Compilerfehler »Eine lokale Variable mit dem Namen "message" ist
      // bereits in diesem Bereich definiert«
      string message = "Denken Sie noch einmal darüber nach.";
      Console.WriteLine(message);
      break;
}
```

Der Compiler beschwert sich in diesem Fall für die letzten drei Fälle darüber, dass die Variable *message* bereits in diesem Bereich deklariert wurde.

Lösen können Sie das Variablen-Deklarations-Problem, indem Sie in den `case`-Blöcken Block-Klammern einsetzen:

Listing 3.69: switch-Verzweigung mit mehreren gleichnamigen Variablen in den einzelnen case-Blöcken

```
switch (number)
{
   case 7:
      {
         string message = "7 ist eine gute Zahl.";
         Console.WriteLine(message);
         break;
      }

   case 42:
      {
         string message = "Cool. Trinken Sie einen " +
            "pangalaktischen Donnergurgler.";
         Console.WriteLine(message);
         break;
      }
```

```
   case 3:
      {
         string message = "3 ist OK.";
         Console.WriteLine(message);
         break;
      }

   default:
      {
         string message = "Denken Sie noch einmal darüber nach.";
         Console.WriteLine(message);
         break;
      }
}
```

Zusammenfassen von mehreren Fällen

Sie können in einer `switch`-Verzweigung mehrere Fälle zusammenfassen, indem Sie die entsprechenden `case`-Anweisungen übereinander schreiben:

Listing 3.70: switch-Verzweigung mit mehreren zusammengefassten case-Blöcken

```
switch (number)
{
   case 7:
   case 42:
   case 3:
      Console.WriteLine("Gute Wahl");
      break;

   default:
      Console.WriteLine("Denken Sie noch einmal darüber nach.");
      break;
}
```

switch-Verzweigungen mit Strings und Methodenaufrufen

Anders als einige andere Sprachen erlaubt C# nicht nur die Überprüfung eines ordinalen Werts, sondern auch von Zeichenketten. Außerdem können Sie im Prüfausdruck auch Methoden aufrufen.

Das folgende Beispiel ruft die `ReadLine`-Methode der `Console`-Klasse direkt im Prüfausdruck auf, um den Anwender seinen Namen eingeben zu lassen und diesen dann auszuwerten:

Listing 3.71: switch-Verzweigung mit Aufruf einer Methode im Prüfausdruck und Auswertung eines Strings

```
Console.Write("Ihr Name: ");
switch (Console.ReadLine())
{
   case "Zaphod":
      Console.WriteLine("Oh, hallo. " +
         "Wie geht's dem Universum so?");
      break;

      case "Arthur":
      Console.WriteLine("Hast du Dein Handtuch dabei?");
      break;

   default:
      Console.WriteLine("Guten Tag, Fremder");
      break;
}
```

Mögliche Sprunganweisungen in switch-Verzweigungen

Als Sprunganweisung erlaubt C# die in Tabelle 3.18 angegebenen.

Sprunganweisung	Bedeutung
`break`	Sprung aus dem `switch`-Block
`return`	Sprung aus der Methode heraus
`throw` *Ausnahme*	Die `throw`-Anweisung wirft eine Ausnahme und führt damit auch zum Sprung aus einem `case`-Block.
`goto` *Labelname*	Sprung zu einem Label, das am Anfang einer Zeile mit *Labelname*: definiert ist. Nur der Vollständigkeit halber hier aufgeführt! Verwenden Sie `goto` nicht!
`goto case` *Vergleichswert*	Sprung zu einem bestimmten `case`-Block, ab dem erneut ausgewertet werden soll. Der Prüfausdruck des Blocks wird natürlich weiterhin ausgewertet. Die Verwendung von `goto case` macht nur dann Sinn, wenn ein `case`-Block die im Programm gespeicherten Daten so verändert, dass der Prüfausdruck einen anderen Wert ergibt als zuvor, und wenn dieser noch einmal ab einem bestimmten `case`-Block überprüft werden soll. Wenn Sie also Programme schreiben wollen, die kein normaler Entwickler versteht, setzen Sie diese verworrene Technik ein.
`goto default`	Sprung zum `default`-Block, der dann auf jeden Fall ausgeführt wird.

Tabelle 3.18: Die C#-Sprunganweisungen

In der Praxis werden hauptsächlich break, return und throw eingesetzt

In der Praxis werden in der Regel `break`, `return` und `throw` eingesetzt. `break` verwenden Sie immer dann, wenn das Programm nach Abarbeitung eines `case`-Blocks hinter dem `switch`-Block weiter ausgeführt werden soll. `return` können Sie einsetzen, wenn der entsprechende Fall dazu führen soll, dass die gesamte Methode beendet wird. `throw` wird dann verwendet, wenn der entsprechende Fall eine Ausnahme darstellt, die im normalen Programmablauf nicht vorkommen sollte und die in einer `Exception` gemeldet werden soll. Das eigene Werfen von Ausnahmen wird in Kapitel 8 behandelt.

Verzichten Sie auf goto

Das unstrukturierte, aus uralten Zeiten übernommene `goto` sollten Sie besser nicht verwenden. Wenn Sie dies trotzdem machen, wird der schwarze Mann Ihr Programm holen ...

3.9.3 Die while-Schleife

Die while-Schleife wiederholt Anweisungen

Die `while`-Schleife überprüft die Schleifenbedingung im Kopf und wiederholt die enthaltenen Anweisungen so lange, wie diese Bedingung erfüllt ist:

```
while (Bedingung)
{
   Anweisungen
}
```

Beachten Sie, dass diese Schleife nicht mit einem Semikolon abgeschlossen werden muss.

Das folgende Beispiel zählt eine Variable hoch, solange diese einen Wert kleiner als 10 besitzt:

Listing 3.72: Beispiel für die Verwendung der while-Schleife

```
int i = 1;
while(i < 10)
{
   Console.WriteLine(i);
   i++;
}
```

EXKURS

Die while-Schleife ist eine kopfgesteuerte Schleife. Solche Schleifen überprüfen ihre Bedingung im Kopf. Die Anweisungen, die im Schleifenkörper enthalten sind, werden nur dann ausgeführt, wenn die Bedingung beim Eintritt in die Schleife wahr ist. Im Gegensatz dazu prüft eine fußgesteuerte Schleife (die do-Schleife) ihre Bedingung im Fuß. Die Anweisungen im Block einer fußgesteuerten Schleife werden auf jeden Fall einmal ausgeführt, auch dann, wenn die Bedingung beim Eintritt in die Schleife nicht wahr ist. Bei der Programmierung entscheiden Sie sich an Hand dieses Kriteriums für eine kopf- oder fußgesteuerte Schleife.

while-Schleife mit Bedingungsprüfung im Körper

Eine while-Schleife können Sie (wie auch eine do-Schleife) mit break explizit abbrechen. So können Sie Schleifen erzeugen, die ihre Bedingung im Körper überprüfen (»bauchgesteuerte« Schleifen ☺). Als Bedingung für die eigentliche Schleife wird einfach true angegeben (was ohne break im Schleifenkörper zu einer Endlosschleife führen würde):

Listing 3.73: while-Schleife, die ihre Abbruchbedingung im Körper überprüft

```
int i = 0;
while (true)
{
   i++;
   if (i > 9)
   {
      break;
   }
   Console.WriteLine(i);
}
```

Solche Schleifen benötigen Sie dann, wenn die Bedingung so komplex ist, dass diese nicht im Kopf oder Fuß der Schleife überprüft werden kann.

3.9.4 Die do-Schleife

Die do-Schleife überprüft die Bedingung im Fuß

Die do-Schleife überprüft die Bedingung im Fuß:

```
do
{
   Anweisungen
}
while (Bedingung);
```

Im Unterschied zur while-Schleife wird die do-Schleife mindestens einmal durchlaufen, auch wenn die Bedingung zu Anfang der Schleife bereits falsch ist.

Das folgende Beispiel lässt den Anwender eine Antwort eingeben und schleift so lange, bis dieser die »richtige« Antwort eingegeben hat:

Listing 3.74: Verwendung der do-Schleife

```
string answer;
int i = 0;
do
{
   i++;
   if (i < 3)
   {
      Console.Write("Geben Sie die Antwort ein: " );
   }
   else
   {
        Console.Write("Geben Sie die Antwort ein " +
           "(Tipp: Die richtige Antwort ist 42): " );
   }
   answer = Console.ReadLine();
}
while (answer != "42");
```

Wie eine `while`-Schleife, können Sie eine `do`-Schleife explizit mit `break` abbrechen:

Listing 3.75: Abbrechen einer do-Schleife

```
int i = 0;
do
{
   i++;
   if (i > 9)
   {
      break;
   }
   Console.WriteLine(i);
} while (true);
```

3.9.5 Die for-Schleife

Die for-Schleife zählt eine Integer-Variable hoch oder herunter

Die `for`-Schleife wird verwendet, wenn die Schleife eine festgelegte Anzahl Durchläufe besitzen soll. Im Kopf der Schleife können Sie eine Zählvariable initialisieren, geben an, bis zu welchem Wert diese gezählt werden soll und welchen Wert der Compiler in jedem Durchlauf der Schleife zu der Variable hinzuzählen oder von dieser abziehen soll:

```
for ([Initialisierungsausdruck]; [Bedingung]; [Zählausdruck])
{
   Anweisungen
}
```

Der Initialisierungsausdruck kann eine Variablendeklaration enthalten.

Das folgende Beispiel zählt eine Variable von eins bis zehn in Einer-Schritten hoch:

Listing 3.76: Hochzählende for-Schleife

```
for (int i = 1; i < 11; i++)
{
   Console.WriteLine(i);
}
```

Rückwärts-Zählen ist natürlich auch möglich:

Listing 3.77: Rückwärts zählende for-Schleife

```
for (int i = 10; i > 0; i--)
{
   Console.WriteLine(i);
}
```

Genauso können Sie auch um mehr als den Wert 1 hoch- oder herunterzählen:

Listing 3.78: for-Schleife, die die Zählvariable bei jedem Durchlauf um zwei erhöht

```
for (int i = 1; i < 10; i += 2)
{
   Console.WriteLine(i);
}
```

Beachten Sie, dass alle drei Teile der `for`-Schleife optional sind. Sie können z. B. die Zählvariable außerhalb der `for`-Schleife deklarieren und initialisieren:

Listing 3.79: for-Schleife mit Zählvariable, die außerhalb der Schleife deklariert und initialisiert wird

```
int number = 0;
for (; number < 10; number++)
{
   Console.WriteLine(number);
}
```

Genauso können Sie aber auch die Bedingung im Schleifenkörper überprüfen (und die Schleife mit `break` beenden) und die Zählvariable im Schleifenkörper hochzählen:

Listing 3.80: for-Schleife mit Bedingung und Hochzählen im Schleifenkörper

```
for (int i = 0; ; )
{
   if (i >= 10)
   {
      break;
   }
   Console.WriteLine(i);
   i++;
}
```

Sie sollten in einem solchen Fall allerdings darüber nachdenken, ob eine `while`- oder `do`-Schleife nicht besser angebracht wäre.

3.9.6 Die foreach-Schleife

Die foreach-Schleife erleichtert das Durchgehen von Arrays und Auflistungen

Die `foreach`-Schleife ist eine sehr wichtige Schleife. Mit ihr können Sie die in einer Auflistung oder in einem Array gespeicherten Objekte sequenziell durchgehen. Auflistungen und Arrays werden zwar erst in Kapitel 7 behandelt, die `foreach`-Schleife gehört aber nun einmal zu den Schleifen und wird deshalb hier beschrieben.

Prinzipiell können Sie Auflistungen und Arrays auch mit einer der anderen Schleifen durchgehen, die `foreach`-Schleife besitzt aber den Vorteil, dass die verwendete Variable die durchlaufenen Objekte direkt referenziert und dass Sie keine Ende-Bedingung überprüfen müssen.

Die Syntax der `foreach`-Schleife ist die folgende:

```
foreach ({var | Typ} Variable in {Auflistung | Array})
   Anweisungen
```

Als Typ geben Sie normalerweise `var` an. In diesem Fall ermittelt der Compiler den Typ der in der Auflistung bzw. im Array gespeicherten Objekte und erzeugt eine Variable von genau dem richtigen Typ. Der Vorteil dieses Vorgehens ist, dass Sie nicht wissen müssen, welchen Typ die Auflistung bzw. das Array verwaltet (was in der Praxis eine große Hilfe ist). Nur wenn der Compiler den Typ nicht aus dem Ausdruck ermitteln kann oder wenn Polymorphismus im Spiel ist (was das ist, klärt Kapitel 5), ist es u. U. notwendig, einen Typen anzugeben (Das ist z. B. der Fall für die `Threads`-Eigenschaft der `Process`-Klasse, die zwar `ProcessThread`-Instanzen verwaltet, bei der der Compiler dies aber nicht erkennt).

Das folgende Beispiel geht auf diese Weise die Auflistung von `Process`-Objekten durch, die die `GetProcesses`-Methode der `Process`-Klasse (aus dem Namensraum `System.Diagnostics`) zurückgibt. `Process`-Objekte repräsentieren einen Prozess im (Windows-)System und geben Informationen zum Prozess und Möglichkeiten, diesen zu steuern. Das Beispiel liest lediglich die ID und den Namen des Prozesses aus:

Listing 3.81: Verwendung der foreach-Schleife zum Durchgehen aller Prozesse des aktuellen Systems

```
foreach (var process in Process.GetProcesses())
{
   Console.WriteLine("ID: " + process.Id);
   Console.WriteLine("Name: " + process.ProcessName);
   Console.WriteLine();
}
```

INFO

Die `foreach`-Schleife wird etwas langsamer ausgeführt als die `for`-Schleife. Das liegt daran, dass die `foreach`-Schleife intern Methodenaufrufe beinhaltet (wie ich bei der `IEnumerable`-Schnittstelle in Kapitel 7 noch näher beschreibe). Der Unterschied liegt zwar nur im Nanosekunden-Bereich, kann aber bei sehr (oder besser: bei extrem) vielen Schleifendurchläufen besonders in performance-kritischen Anwendungen schon einen Unterschied ausmachen.

for ist insignifikant schneller als foreach

In einem Performance-Test (den Sie im Projekt foreach-Performance-Test in den Beispielen zu diesem Kapitel finden) habe ich ermittelt, dass die `for`-Schleife auf meinem Laptop (2 GHz Dual Core, 2 GB RAM) für eine `int`-Liste bei 100.000 gespeicherten Elementen im Durchschnitt 0,21 ms und die `foreach`-Schleife im Mittel 0,73 ms benötigte. Das ist bei der `foreach`-Schleife zwar etwa 3,5-mal langsamer, aber bei der benötigten Zeit von 7,3 Nanosekunden für einen Schleifendurchlauf wohl kaum relevant. Deshalb würde ich der `foreach`-Schleife aufgrund der einfacheren Anwendung immer den Vorzug geben.

Falls Sie den Test nachvollziehen, beachten Sie, dass Sie das Release des Projekts unter Windows direkt ausführen sollten um korrekte Ergebnisse zu erhalten. Und um Kritik vorzubeugen: Ich habe natürlich auch damit experimentiert, die Schleifen in der anderen Reihenfolge als im Test (`foreach`-Schleife vor der `for`-Schleife) auszuführen ☺.

3.10 Präprozessor-Direktiven

Präprozessor-Direktiven steuern u. a. die Kompilierung

C# kennt einige Präprozessor-Direktiven, über die Sie die Kompilierung steuern und spezielle Features verwenden können. Präprozessor-Direktiven beginnen immer mit einem #. Das Wesentliche an diesen Direktiven ist, dass die nicht in das Programm übernommen werden. Sie stellen entweder spezielle Informationen für den Compiler oder für Visual Studio dar. Tabelle 3.19 beschreibt die wichtigen Präprozessor-Direktiven.

Tabelle 3.19: Die wichtigen C#-Präprozessor-Direktiven

Direktive	Bedeutung
`#define` *Konstante* **[=** **{**`true` **|** `false`**}]**	Definition einer Konstante für die bedingte Kompilierung.
`#if` *Bedingung* **[**`#elif` *Bedingung***]** **[**`#else`**]** `#endif`	In einem `#if`-Block können Sie Anweisungen unterbringen, die nur dann kompiliert werden, wenn die Bedingung wahr bzw. falsch wird. Die Bedingung kann alle Vergleichsoperatoren beinhalten und vergleicht mit den Konstanten, die mit `#define` oder in den Projekteigenschaften angelegt wurden.
`#warning` *Warnung* `#error` *Fehler*	Diese Anweisung können Sie in einen `#if`-Block einbauen, um beim Eintritt einer Präprozessor-Bedingung eine Compiler-Warnung oder einen Compiler-Fehler zu generieren.
`#region` *Bezeichnung* `#endregion`	In diese Direktive schließen Sie Anweisungen ein, die als Region verwaltet werden sollen. In Visual Studio können Sie Regionen auf- und zuklappen. Regionen werden in Kapitel 4 behandelt.
`#pragma` *pragma-name argumente*	Mit `#pragma` werden dem Compiler spezielle Informationen übergeben. Über `#pragma warning disable` *Warnungsliste* können Sie z. B. Warnungen, die der Compiler für eine Klasse ausgibt, deaktivieren. In der Warnungsliste geben Sie dazu die ID der Warnungen kommagetrennt an. Die ID erhalten Sie über das Fehlerfenster, das Visual Studio beim Auftreten von Fehlern oder Warnungen beim Kompilieren anzeigt.

#if erlaubt eine bedingte Kompilierung

Über einen `#if`-Block können Sie eine bedingte Kompilierung erreichen. Die Arbeitsweise entspricht der der `if`-Verzweigung, mit dem Unterschied, dass Code in Blöcken, deren Bedingung nicht zutrifft, vom Compiler nicht berücksichtigt wird. Dieser Code kann dann auch für die aktuelle Umgebung nicht benutzbare Anweisungen enthalten.

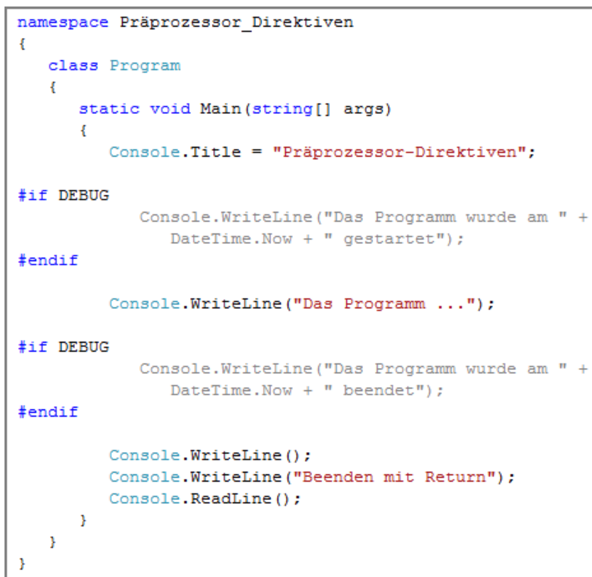
Der Bedingungsausdruck darf lediglich Literale, Operatoren und Konstanten für die bedingte Kompilierung enthalten. In der Debug-Konfiguration aller Projekte ist per Voreinstellung bereits die Konstante `DEBUG` definiert (was Sie allerdings in den Projekteigenschaften in der Debug-Konfiguration auch abschalten können). So können Sie im Programm sehr einfach abfragen, ob es sich um eine Debug- oder eine Release-Anwendung handelt, und in der Debug-Variante z. B. spezielle Protokollierungen vornehmen:

Listing 3.82: Kleine Demo-Konsolenanwendung mit bedingter Kompilierung

```
static void Main(string[] args)
{
   Console.Title = "Präprozessor-Direktiven";

#if DEBUG
   Console.WriteLine("Das Programm wurde am " +
      DateTime.Now + " gestartet");
#endif
```

```
   Console.WriteLine("Das Programm ...");

#if DEBUG
   Console.WriteLine("Das Programm wurde am " +
      DateTime.Now + " beendet");
#endif

   Console.WriteLine();
   Console.WriteLine("Beenden mit Return");
   Console.ReadLine();
}
```

Visual Studio zeigt die Anweisungen, die aufgrund der bedingten Kompilierung nicht kompiliert werden, in einer grauen Farbe an. Sie können dies ausprobieren, indem Sie das obige Beispiel (das Sie natürlich auf der Buch-DVD finden) in die Release-Konfiguration schalten.



```
namespace Präprozessor_Direktiven
{
   class Program
   {
      static void Main(string[] args)
      {
         Console.Title = "Präprozessor-Direktiven";

#if DEBUG
            Console.WriteLine("Das Programm wurde am " +
               DateTime.Now + " gestartet");
#endif

         Console.WriteLine("Das Programm ...");

#if DEBUG
            Console.WriteLine("Das Programm wurde am " +
               DateTime.Now + " beendet");
#endif

         Console.WriteLine();
         Console.WriteLine("Beenden mit Return");
         Console.ReadLine();
      }
   }
}
```

Abbildung 3.10: Visual Studio zeigt Anweisungen in einer bedingten Kompilierung, die in der aktuellen Konfiguration nicht kompiliert werden, in einer grauen Farbe an

Immer wenn Sie Anweisungen nur zum Test in Ihren Programmen unterbringen, sollten Sie diese in eine bedingte Kompilierung einschließen, die auf das `DEBUG`*-Symbol überprüft. Auf diese Weise verhindern Sie, dass diese Anweisungen in das Release übernommen werden, falls Sie vergessen, diese zu löschen.*

Sie können die bedingte Kompilierung aber auch für spezielle Zwecke, wie z. B. zur Erzeugung einer Shareware-Version eine Anwendung, verwenden. Die dazu zu verwendenden Konstanten können Sie im Kopf einer Datei über `#define` *Name* **[**= **{**`true` | `false`**}]** einrichten. Solche Konstanten gelten dann nur für die entsprechende Datei.

Sie können Konstanten für eine bedingte Kompilierung aber auch in den Eigenschaften des Projekts im ERSTELLEN-Register angeben. Diese gelten dann für das gesamte Projekt. Diese Technik wird interessant, wenn Sie spezielle Konfigurationen erstellen, in denen Sie die Konstanten angeben. So können Sie z. B. eine Konfiguration

Shareware erstellen, in der Sie die Konstante SHAREWARE unterbringen. Wenn Sie im Programm mit dieser Konstante bedingt kompilieren, können Sie über ein einfaches Umschalten der Projekt-Konfiguration ein normales Release oder eine spezielle Version (im Beispiel eine eingeschränkte Shareware-Version) erstellen.