2 Die Sprache C#

C# (sprich: *see sharp*) ist Microsofts neue Programmiersprache für die .NET-Plattform. Obwohl man .NET auch in anderen Sprachen programmieren kann (z.B. in Visual Basic .NET oder C++), ist C# die von Microsoft bevorzugte Sprache, die .NET am besten unterstützt und die von .NET am besten unterstützt wird.

C# ist keine revolutionär neue Sprache. Sie ist vielmehr eine Kombination aus Java, C++ und Visual Basic, wobei man versucht hat, von jeder Sprache bewährte Eigenschaften zu übernehmen und komplexe Eigenschaften zu vermeiden. C# wurde von einem relativ kleinen Team unter der Leitung von *Anders Hejlsberg* entworfen. Hejlsberg ist ein erfahrener Sprachdesigner. Er war bei Borland Chefentwickler von Delphi und ist dafür bekannt, seine Sprachen auf die Bedürfnisse von Praktikern zuzuschneiden.

In diesem Kapitel wird davon ausgegangen, dass der Leser bereits programmieren kann, am besten in Java oder C++. Während wir uns die Konzepte von C# ansehen, arbeiten wir auch die Unterschiede zu Java und C++ heraus.

2.1 Überblick

Ähnlichkeiten zu Java

Auf den ersten Blick sehen C#-Programme wie Java-Programme aus. Jeder Java-Programmierer sollte daher in der Lage sein, C#-Programme zu lesen. Neben der fast identischen Syntax wurden folgende Konzepte aus Java übernommen:

- *Objektorientierung*. C# ist wie Java eine objektorientierte Sprache mit einfacher Vererbung. Klassen können nur von einer einzigen Klasse erben, aber mehrere Schnittstellen (Interfaces) implementieren.
- *Typsicherheit*. C# ist eine typsichere Sprache. Viele Programmierfehler, die durch inkompatible Datentypen in Anweisungen und Ausdrücken entstehen, werden bereits vom Compiler abgefangen. Zeigerarithmetik oder ungeprüfte Typumwandlungen wie in C++ gibt es nicht. Zur Laufzeit wird sichergestellt, dass Array-Indizes im erlaubten Bereich liegen, dass Objekte

nicht durch uninitialisierte Zeiger referenziert werden und dass Typumwandlungen zu einem definierten Ergebnis führen.

- *Garbage Collection*. Dynamisch erzeugte Objekte werden vom Programmierer nie selbst freigegeben, sondern von einem Garbage Collector automatisch eingesammelt, sobald sie nicht mehr referenziert werden. Das beseitigt viele unangenehme Fehler, die z.B. in C++-Programmen auftreten können.
- *Namensräume*. Was in Java Pakete sind, nennt man in C# Namensräume. Ein Namensraum ist eine Sammlung von Deklarationen und ermöglicht es, gleichnamige Klassen, Variablen oder Methoden in unterschiedlichem Kontext zu verwenden.
- *Threads*. C# unterstützt leichtgewichtige parallele Prozesse in Form von Threads. Es gibt wie in Java Mechanismen zur Synchronisation und Kommunikation zwischen Prozessen.
- *Generizität*. Sowohl Java als auch C# kennen generische Typen und Methoden. Damit kann man Bausteine herstellen, die mit anderen Typen parametrisierbar sind (z.B. Listen mit beliebigem Elementtyp).
- *Reflection*. Wie in Java kann man auch in C# zur Laufzeit auf Typinformationen eines Programms zugreifen, Klassen dynamisch zu einem Programm hinzuladen, ja sogar Programme zur Laufzeit zusammenstellen und ausführen.
- *Attribute*. Der Programmierer kann beliebige Informationen an Klassen, Methoden oder Felder hängen und sie zur Laufzeit mittels Reflection abfragen. In Java heißt dieser Mechanismus *Annotationen*.
- *Bibliotheken*. Viele Typen der C#-Bibliothek sind denen der Java-Bibliothek nachempfunden. So gibt es vertraute Typen wie Object, String, Collection oder Stream, meist sogar mit den gleichen Methoden wie in Java.

Auch aus C++ wurden einige Dinge übernommen, zum Beispiel das Überladen von Operatoren, die Zeigerarithmetik in systemnahen Klassen (die als *unsafe* gekennzeichnet sein müssen) sowie einige syntaktische Details z.B. im Zusammenhang mit Vererbung. Aus Visual Basic stammt beispielsweise die foreach-Schleife.

Unterschiede zu Java

Neben diesen Ähnlichkeiten weist C# aber wie alle .NET-Sprachen auch einige Merkmale auf, die in Java fehlen:

- *Referenzparameter.* Parameter können nicht nur durch *call by value* übergeben werden, wie das in Java üblich ist, sondern auch durch *call by reference*. Dadurch sind nicht nur Eingangs-, sondern auch Ausgangs- und Übergangsparameter realisierbar.

- *Objekte am Keller.* Während in Java alle Objekte am Heap liegen, kann man in C# Objekte auch am Methodenaufrufkeller anlegen. Diese Objekte sind leichtgewichtig und belasten den Garbage Collector nicht.
- *Blockmatrizen.* Für numerische Anwendungen ist das Java-Speichermodell mehrdimensionaler Arrays zu ineffizient. C# lässt dem Programmierer die Wahl, mehrdimensionale Arrays entweder wie in Java anzulegen oder als kompakte Blockmatrizen, wie das in C, Fortran oder Pascal üblich ist.
- *Einheitliches Typsystem.* Im Gegensatz zu Java sind in C# alle Datentypen (also auch int oder char) vom Typ object abgeleitet und erben die dort deklarierten Methoden.
- *goto-Anweisung.* Die viel geschmähte goto-Anweisung wurde in C# wieder eingeführt, allerdings mit Einschränkungen, so dass man mit ihr kaum Missbrauch treiben kann.
- *Versionierung.* Klassen werden bei der Übersetzung mit einer Versionsnummer versehen. So kann eine Klasse gleichzeitig in verschiedenen Versionen vorhanden sein. Jede Applikation verwendet immer diejenige Version der Klasse, mit der sie übersetzt und getestet wurde.

Schließlich gibt es noch zahlreiche Eigenschaften von C#, die zwar die Mächtigkeit der Sprache nicht erhöhen, aber bequem zu benutzen sind. Sie fallen unter die Kategorie »*syntactic sugar*«, d.h., man kann mit ihnen Dinge tun, die man auch in anderen Sprachen realisieren könnte, nur dass es in C# eben einfacher und eleganter geht. Dazu gehören:

- *Properties* und *Events.* Diese Eigenschaften dienen der Komponententechnologie. Properties sind spezielle Felder eines Objekts. Greift man auf sie zu, werden automatisch get- und set-Methoden aufgerufen. Mit Events kann man Ereignisse definieren, die von Komponenten ausgelöst und von anderen behandelt werden.
- *Indexer.* Ein Index-Operator wie bei Array-Zugriffen kann durch get- und set-Methoden selbst definiert werden.
- *Delegates.* Delegates sind im Wesentlichen das, was man in Pascal *Prozedurvariablen* und in C *Function Pointers* nennt. Sie sind allerdings etwas mächtiger. Zum Beispiel kann man mehrere Prozeduren in einer einzigen Delegate-Variablen speichern.
- *foreach-Schleife.* Damit kann man bequem über Arrays, Listen oder Mengen iterieren.
- *Iteratoren.* Man kann Klassen mit speziellen Iterator-Methoden ausstatten, die eine Folge von Werten liefern, welche man dann mit einer foreach-Schleife durchlaufen kann.

Hello World

Nun wird es aber Zeit für ein erstes Beispiel. Das bekannte Hello-World-Programm sieht in C# folgendermaßen aus:

```
using System;

class Hello {

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

}
```

Es besteht aus einer Klasse Hello und einer Methode Main (Achtung: Groß- und Kleinschreibung ist in C# signifikant). Jedes Programm hat genau eine Main-Methode, die aufgerufen wird, wenn man es startet. Die Ausgabeanweisung heißt Console.WriteLine("..."), wobei WriteLine eine Methode der Klasse Console ist, die aus dem Namensraum System stammt. Um Console bekannt zu machen, muss man System in der ersten Zeile mittels using importieren. C#-Programme werden in Dateien mit der Endung .cs gespeichert.

Die einfachste Arbeitsumgebung für .NET ist das *Software Development Kit* (SDK) von Microsoft. Es ist kommandozeilenorientiert und bietet neben einem Compiler (csc) noch einige andere Werkzeuge (z.B. al, ildasm), die in Kapitel 8 beschrieben werden. Wenn wir unser Hello-World-Programm in eine Datei Hello.cs abspeichern, können wir es durch Eingabe von

```
csc Hello.cs
```

im Konsolenfenster übersetzen und mittels

```
Hello
```

aufrufen. Die Ausgabe erscheint wieder im Konsolenfenster.

Der Dateiname (z.B. Hello.cs) muss übrigens unter .NET nicht mit dem Klassennamen (z.B. Hello) übereinstimmen, obwohl es aus Lesbarkeitsgründen empfehlenswert ist. Eine Datei kann auch mehrere Klassen enthalten. In diesem Fall sollte sie nach der Hauptklasse benannt sein.

Gliederung von Programmen

Der Quelltext eines C#-Programms kann auf mehrere Dateien verteilt sein. Jede Datei kann aus einem oder mehreren Namensräumen bestehen, von denen jeder eine oder mehrere Klassen oder andere Typen enthalten kann. Abb. 2.1 zeigt diese Struktur.

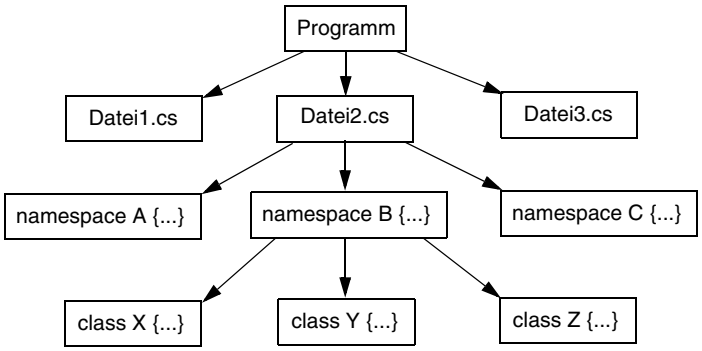


Abb. 2.1 *Gliederung von Programmen*

Unser Hello-World-Programm besteht nur aus einer einzigen Datei und einer einzigen Klasse. Namensraum wurde keiner angegeben, was bedeutet, dass die Klasse Hello zu einem namenlosen Standardnamensraum gehört, den .NET für uns bereithält. Namensräume werden in Abschnitt 2.5 und 2.13 behandelt, Klassen in Abschnitt 2.8.

Programme aus mehreren Dateien

Wenn ein Programm aus mehreren Dateien besteht, können wir diese entweder gemeinsam oder getrennt übersetzen. Im ersten Fall entsteht eine einzige ausführbare Datei, im zweiten Fall eine ausführbare Datei und eine DLL (*dynamic link library*).

Nehmen wir an, eine Klasse Counter in der Datei Counter.cs wird von einer Klasse Prog in der Datei Prog.cs benutzt:

```
public class Counter {      // Datei Counter.cs
    int val = 0;
    public void Add(int x) { val = val + x; }
    public int Val() { return val; }
}

using System;               // Datei Prog.cs
public class Prog {
    public static void Main() {
        Counter c = new Counter();
        c.Add(3); c.Add(5);
        Console.WriteLine("val = " + c.Val());
    }
}
```

Wir können diese beiden Dateien nun gemeinsam übersetzen:

```
csc Prog.cs Counter.cs
```

wodurch eine ausführbare Datei Prog.exe entsteht, die beide Klassen enthält. Alternativ dazu könnten wir aus Counter aber auch eine Bibliothek (DLL) machen, indem wir schreiben:

```
csc /target:library Counter.cs
```

Der Compiler erzeugt auf diese Weise eine Datei Counter.dll, die wir dann bei der Übersetzung von Prog.cs folgendermaßen angeben müssen:

```
csc /reference:Counter.dll Prog.cs
```

Aus dieser Übersetzung entsteht zwar auch eine Datei Prog.exe; sie enthält aber nur die Klasse Prog. Die Klasse Counter steht nach wie vor in der Datei Counter.dll und wird beim Aufruf von Prog dynamisch dazugeladen. Die verschiedenen Formen des Compileraufrufs werden in Abschnitt 8.2 genauer beschrieben.

2.2 Symbole

C#-Programme bestehen aus Namen, Schlüsselwörtern, Zahlen, Zeichen, Zeichenketten, Operatoren und Kommentaren.

Namen. Ein Name besteht aus Buchstaben, Ziffern und dem Zeichen "_". Das erste Zeichen muss ein Buchstabe oder ein "_" sein. Groß-und Kleinbuchstaben haben unterschiedliche Bedeutung (d.h. red ist ungleich Red). Da C# den Unicode-Zeichensatz benutzt [UniC], können Namen auch griechische, arabische oder chinesische Zeichen enthalten. Man muss sie allerdings auf unseren Tastaturen als Nummerncodes eingeben. Der Code \u03C0 bedeutet z.B. π, der Name b\u0061ck den Namen back.

Schlüsselwörter. C# kennt 77 Schlüsselwörter, Java nur 50. Das deutet schon darauf hin, dass C# komplexer ist als Java. Schlüsselwörter sind reserviert, d.h., sie dürfen nicht als Namen verwendet werden.

abstract	as	base	bool	break	byte
case	catch	char	checked	class	const
continue	decimal	default	delegate	do	double
else	enum	event	explicit	extern	false
finally	fixed	float	for	foreach	goto
if	implicit	in	int	interface	internal
is	lock	long	namespace	new	null
object	operator	out	override	params	private
protected	public	readonly	ref	return	sbyte
sealed	short	sizeof	stackalloc	static	string
struct	switch	this	throw	true	try
typeof	uint	ulong	unchecked	unsafe	ushort
using	virtual	void	volatile	while	

Namenskonventionen. Bei der Namenswahl und bei der Groß-/Kleinschreibung sollte man sich an die Regeln halten, die auch in der Klassenbibliothek von C# benutzt werden:

- Namen beginnen mit großen Anfangsbuchstaben (z.B. Length, WriteLine), außer bei lokalen Variablen und Parametern (z.B. i, len) oder bei Feldern einer Klasse, die von außen nicht sichtbar sind.
- In zusammengesetzten Wörtern beginnt jedes Wort mit einem Großbuchstaben (z.B. WriteLine). Die Trennung von Wörtern durch "_" wird in C# selten verwendet.
- Methoden ohne Rückgabewert sollten mit einem Verb beginnen (z.B. DrawLine). Alles andere sollte in der Regel mit einem Substantiv beginnen (z.B. Size, IndexOf, Collection). Felder oder Methoden mit booleschem Typ können auch mit einem Adjektiv beginnen, wenn sie eine Eigenschaft ausdrücken (z.B. Empty).
- Da Schlüsselwörter und Namen aus der .NET-Bibliothek englisch sind, sollte man auch seine eigenen Programmobjekte englisch benennen.

Zeichen und Zeichenketten. Zeichenkonstanten werden zwischen einfache Hochkommas eingeschlossen (z.B. 'x'), Zeichenkettenkonstanten zwischen doppelte Hochkommas (z.B. "John"). In beiden dürfen beliebige Zeichen vorkommen, außer das schließende Hochkomma, ein Zeilenende oder das Zeichen \, das als *Escape-Zeichen* verwendet wird. Folgende Escape-Sequenzen dienen zur Darstellung von Sonderzeichen in Zeichen- und Zeichenkettenkonstanten:

\'	'
\"	"
\\	\
\0	0x0000 (das Zeichen mit dem Wert 0)
\a	0x0007 (alert)
\b	0x0008 (backspace)
\f	0x000c (form feed)
\n	0x000a (new line)
\r	0x000d (carriage return)
\t	0x0009 (horizontal tab)
\v	0x000b (vertical tab)

Um zum Beispiel den Text

```
file "C:\sample.txt"
```

als Zeichenkette darzustellen, muss man schreiben:

```
"file \"C:\\sample.txt\""
```

Daneben können wie in Namen auch Unicode-Konstanten (z.B. \u0061) verwendet werden.

Wenn vor einer Zeichenkette das Zeichen @ steht, dürfen darin Zeilenumbrüche vorkommen, \ wird nicht als Escape-Zeichen interpretiert und das Hochkomma muss verdoppelt werden. Das obige Beispiel könnte man also auch so schreiben:

```
@"file ""C:\sample.txt"""
```

Ganze Zahlen. Ganze Zahlen können in Dezimalschreibweise (z.B. 123) oder in Hexadezimalschreibweise (z.B. 0x007b) vorkommen. Der Typ der Zahl ist der kleinste Typ aus int, uint, long oder ulong, zu dem der Zahlenwert passt. Durch Anhängen der Endung u oder U (z.B. 123u) erzwingt man den kleinsten passenden vorzeichenlosen Typ (uint oder ulong), durch Anhängen von l oder L (z.B. 0x007bl) den kleinsten passenden Typ aus der Menge long und ulong.

Gleitkommazahlen. Gleitkommazahlen bestehen aus einem ganzzahligen Teil, einem Kommateil und einem Exponenten (3.14E0 bedeutet z.B. $3.14 * 10^0$). Jeder dieser Teile kann fehlen, aber zumindest einer davon muss vorkommen. Die Zahlen 3.14, 314E-2 und .314E1 sind also gültige Schreibweisen desselben Werts. Der Typ einer Gleitkommakonstante ist double, durch die Endung f oder F (z.B. 1f) erzwingt man den Typ float, durch m oder M (z.B. 12.3m) den Typ decimal.

Kommentare. Es gibt zwei Arten von Kommentaren: *Zeilenendekommentare* beginnen mit // und erstrecken sich bis zum Zeilenende, z.B.:

```
// ein Kommentar
```

Klammerkommentare beginnen mit /* und enden mit */. Sie können sich auch über mehrere Zeilen erstrecken, dürfen aber nicht geschachtelt werden, z.B.:

```
/* ein Kommentar,
der zwei Zeilen einnimmt */
```

Zeilenendekommentare werden für kurze Erläuterungen verwendet, Klammerkommentare meist zum Auskommentieren von Code.

2.3 Typen

Die Datentypen von C# bilden eine Hierarchie, wie das in Abb. 2.2 gezeigt wird. Grundsätzlich gibt es Werttypen und Referenztypen. *Werttypen* sind einfache Typen wie char, int oder float, Enumerationen und Structs. Variablen dieser Typen enthalten direkt einen Wert (z.B. 'x', 123 oder 3.14). *Referenztypen* sind Klassen, Interfaces, Arrays und Delegates. Variablen dieser Typen enthalten eine Referenz auf ein Objekt, das in einem dynamisch wachsenden Speicherbereich (dem *Heap*) angelegt wird.

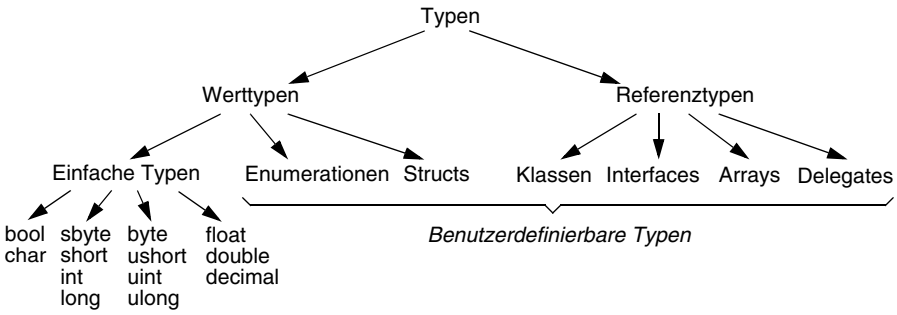


Abb. 2.2 *Typenhierarchie*

C# besitzt ein einheitliches Typsystem, d.h., alle Typen, ob Werttypen oder Referenztypen, sind mit dem Typ object kompatibel: Variablen dieser Typen können object-Variablen zugewiesen werden und verstehen object-Operationen. Tabelle 2.1 veranschaulicht den Unterschied zwischen Wert- und Referenztypen.

Tabelle 2.1 *Werttypen und Referenztypen*

	Werttypen	*Referenztypen*
Variable enthält	einen Wert	eine Referenz auf ein Objekt
Variable wird gespeichert	am Methodenkeller oder im enthaltenden Objekt	am Heap
Zuweisung	kopiert den Wert	kopiert die Referenz

Beispiel	int i = 17;
	int j = i;
	i 17
	j 17
		string s = "Hello";
		string s1 = s;
		s
		s1
		Hello

Der in Tabelle 2.1 verwendete Typ string ist eine vordefinierte Klasse und somit ein Referenztyp. Eigentlich ist string ein Schlüsselwort, das vom Compiler auf die Klasse System.String abgebildet wird (d.h. die Klasse String aus dem Namensraum System). Auch object wird auf die Klasse System.Object abgebildet.

2.3.1 Einfache Typen

Wie jede Sprache kennt C# einige vordefinierte Typen für Zahlen, Zeichen und boolesche Werte. Bei den Zahlen wird zwischen ganzzahligen Typen und Gleitkommatypen unterschieden und innerhalb dieser wieder nach Größe und Genauigkeit. Tabelle 2.2 zeigt eine Aufstellung aller einfachen Typen.

Tabelle 2.2 *Einfache Typen*

	Wertebereich	*abgebildet auf*
sbyte	-128 .. 127	System.SByte
short	-32768 .. 32767	System.Int16
int	-2147483648 .. 2147483647	System.Int32
long	-2^{63} .. $2^{63}-1$	System.Int64
byte	0 .. 255	System.Byte
ushort	0 .. 65535	System.UInt16
uint	0 .. 4294967295	System.UInt32
ulong	0 .. $2^{64}-1$	System.UInt64
float	±1.4E-45 .. ±3.4E38 (32 Bit, IEEE 754)	System.Single
double	±5E-324 .. ±1.7E308 (64 Bit, IEEE 754)	System.Double
decimal	±1E-28 .. ±7.9E28 (128 Bit)	System.Decimal
bool	true, false	System.Boolean
char	Unicode-Zeichen	System.Char

Die vorzeichenlosen Typen byte, ushort, uint und ulong dienen vor allem der Systemprogrammierung und der Kompatibilität zu anderen Sprachen. Der Typ decimal erlaubt die exakte Darstellung großer Dezimalzahlen mit großer Genauigkeit und wird vor allem in der Finanzmathematik verwendet.

Alle einfachen Typen werden vom Compiler auf Struct-Typen des Namensraums System abgebildet. Der Typ int entspricht z.B. dem Struct System.Int32. Alle dort definierten Operationen (einschließlich der von System.Object geerbten) sind somit auf int anwendbar.

Zwischen den meisten einfachen Typen besteht eine Kompatibilitätsbeziehung, die in Abb. 2.3 dargestellt ist. Ein Pfeil zwischen char und ushort bedeutet dabei, dass char-Werte einer ushort-Variablen zugewiesen werden dürfen (ushort schließt alle char-Werte ein). Die Beziehung ist transitiv, d.h., char-Werte dürfen auch int- oder float-Variablen zugewiesen werden. Eine Zuweisung an decimal ist allerdings nur nach einer expliziten Typumwandlung erlaubt (z.B. decimal d = (decimal) 3;). Bei der Zuweisung von long oder ulong an float kommt es zu einem Genauigkeitsverlust, falls die Bits der Mantisse nicht ausreichen, um das Ergebnis darzustellen.

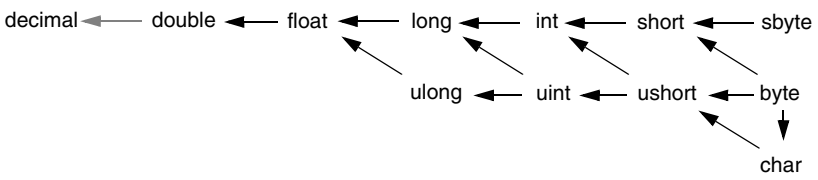


Abb. 2.3 *Kompatibilitätsbeziehung zwischen einfachen Typen*

2.3.2 Enumerationen

Enumerationen sind Aufzählungstypen aus benannten Konstanten. Ihre erlaubten Werte werden bei der Deklaration angegeben, z.B.:

```
enum Color {red, blue, green}
```

Variablen vom Typ Color können also die Werte red, blue oder green annehmen, wobei der Compiler diese Werte auf die Zahlen 0, 1 und 2 abbildet. Enumerationen sind aber keine Zahlentypen; man kann sie keiner Zahlenvariablen zuweisen und umgekehrt darf man keinen Zahlenwert einer Color-Variablen zuweisen. Auf Wunsch kann der Wert der Enumerationskonstanten bei der Deklaration spezifiziert werden, also

```
enum Color {red=1, blue=2, green=4}
enum Direction {left=0, right, up=4, down}  // left=0, right=1, up=4, down=5
```

Enumerationswerte sind in der Regel 4 Byte groß. Man kann allerdings auch eine andere Typgröße wählen, indem man hinter den Typnamen den gewünschten (numerischen) Basistyp schreibt, z.B.:

```
enum Access : byte {personal=1, group=2, all=4}
```

Variablen vom Typ Access sind also 1 Byte groß. Enumerationen können zum Beispiel wie folgt verwendet werden:

```
Color c = Color.blue;
Access a = Access.personal | Access.group;
if ((a & Access.personal) != 0) Console.WriteLine("access granted");
```

Bei der Verwendung müssen Enumerationskonstanten mit ihrem Typnamen qualifiziert werden. Wenn man wie beim Typ Access die Werte als Zweierpotenzen wählt, kann man durch logische Operationen (&, |, ~) Bitmengen bilden. In einer Enumerationsvariablen kann somit eine Menge von Werten stehen. Dass dabei Werte entstehen, die keiner erlaubten Enumerationskonstanten entsprechen, stört den Compiler nicht (Access.personal | Access.group ergibt z.B. den Wert 3). Mit Enumerationen sind folgende Operationen erlaubt:

```
==, !=, <, <=, >, >=   if (c == Color.red) ...
                       if (c > Color.red && c <= Color.green) ...
+, -                   c = c + 2;
++, --                 c++;
&                      if ((a & Access.personal) != 0) ...
|                      a = a | Access.group;
~                      a = ~ Access.all;  // Einerkomplement
```

Wie bei logischen Operationen kann auch bei arithmetischen Operationen ein Wert entstehen, der keiner Enumerationskonstanten entspricht. Der Compiler akzeptiert das.

Enumerationen erben alle Operationen von object, wie zum Beispiel Equals oder ToString (siehe Abschnitt 2.3.7). Es gibt auch eine Klasse System.Enum, die spezielle Operationen auf Enumerationen bereitstellt.

2.3.3 Arrays

Arrays sind ein- oder mehrdimensionale Vektoren von Elementen. Die Elemente werden durch einen Index angesprochen, wobei die Indizierung bei 0 beginnt.

Eindimensionale Arrays. Eindimensionale Arrays werden durch ihren Elementtyp und eine leere Indexklammer deklariert:

```
int[] a;                                // deklariert eine Array-Variable a
int[] b = new int[3];                   // Initialisierung mit einem leeren Array
int[] c = new int[] {3, 4, 5};          // Initialisierung mit den Werten 3, 4, 5
int[] d = {3, 4, 5};                    // Initialisierung mit den Werten 3, 4, 5
SomeClass[] e = new SomeClass[10];      // Array von Referenzen
SomeStruct[] f = new SomeStruct[10];    // Array von Werten (direkt im Array)
```

Durch die Deklaration eines Arrays wird noch kein Speicherplatz angelegt, weshalb man auch noch keine Array-Länge angibt. Der new-Operator erzeugt ein Array eines gewünschten Elementtyps und einer gewünschten Länge (new int[3] erzeugt z.B. ein Array aus drei int-Elementen). Die Werte werden dabei mit 0 (bzw. '\0', false, null) initialisiert, außer es wird eine andere Initialisierung in geschweiften Klammern angegeben. Bei der Deklaration eines Arrays kann die Initialisierung auch direkt (d.h. ohne new-Operator) angegeben werden, wobei der Compiler dann ein Array der passenden Größe erzeugt.

Beachten Sie bitte, dass in einem Array aus Klassen *Referenzen* stehen, während ein Array aus Structs direkt die *Werte* der Structs als Elemente enthält.

Mehrdimensionale Arrays. Bei mehrdimensionalen Arrays unterscheidet C# zwischen ausgefransten (*jagged*) und rechteckigen Arrays. Ausgefranste Arrays enthalten als Elemente wieder Referenzen auf andere Arrays, während die Elemente bei rechteckigen Arrays hintereinander im Speicher liegen (siehe Abb. 2.4). Rechteckige Arrays sind nicht nur kompakter, sondern erlauben auch eine effizientere Indizierung. Hier sind einige Beispiele mehrdimensionaler Arrays:

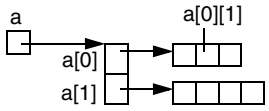
```
// ausgefranste Arrays (werden mit [][] deklariert)
int[][] a = new int[2][];           // 2 Zeilen, deren Spaltenanzahl noch undefiniert ist
a[0] = {1, 2, 3};                   // Zeile 0 hat 3 Spalten
a[1] = {4, 5, 6, 7, 8};             // Zeile 1 hat 5 Spalten

// rechteckige Arrays (werden mit [,] deklariert)
int[,] a = new int[2, 3];           // 2 Zeilen zu 3 Spalten
int[,] b = {{1, 2, 3}, {4, 5, 6}};  // Initialisierung der 2 Zeilen und 3 Spalten
int[,,] c = new int[2, 4, 2];       // 2 Blöcke zu 4 Zeilen zu 2 Spalten
```

In ausgefransten Arrays können die Zeilen also unterschiedlich lang sein. Dafür darf man bei ihrer Erzeugung nur die Länge der ersten Dimension angeben und nicht die Länge aller Dimensionen, wie das bei rechteckigen Arrays möglich ist. Abb. 2.4 zeigt den Unterschied zwischen den beiden Array-Arten grafisch.

Ausgefranst (wie in Java)

```
int[][] a = new int[2][];
a[0] = new int[3];
a[1] = new int[4];
int x = a[0][1];
```



Rechteckig

```
int[,] a = new int[2, 3];
int x = a[0, 1];
```

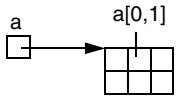


Abb. 2.4 *Ausgefranste und rechteckige mehrdimensionale Arrays*

Array-Operationen. Wie man aus Abb. 2.4 sieht, enthalten Array-Variablen in C# Referenzen. Eine Array-Zuweisung ist also eine *Zeigerzuweisung*. Das Array selbst wird dabei nicht kopiert. Neben der Indizierung, die in C# immer bei 0 beginnt, kann man mit Length die Array-Länge abfragen.

```
int[] a = new int[3];
int[][] b = new int[2][];
b[0] = new int[4];
b[1] = new int[4];
Console.WriteLine(a.Length);      // 3
Console.WriteLine(b.Length);      // 2
Console.WriteLine(b[0].Length);   // 4
```

Bei rechteckigen Arrays liefert Length die Gesamtanzahl der Elemente. Um die Anzahl der Elemente einer bestimmten Dimension zu bekommen, muss man die GetLength-Methode verwenden.

```
int[,] a = new int[3, 4];
Console.WriteLine(a.Length);            // 12
Console.WriteLine(a.GetLength(0));      // 3
Console.WriteLine(a.GetLength(1));      // 4
```

Die Klasse System.Array enthält noch einige nützliche Operationen wie das Kopieren, Sortieren und Suchen in Arrays.

```
int[] a = new int[2];
int[] b = {7, 2, 4};
Array.Copy(b, a, 2);          // kopiert b[0..1] nach a
Array.Sort(b);                // sortiert b aufsteigend
```

Arrays variabler Länge. Gewöhnliche Arrays haben eine feste Länge. Es gibt allerdings eine Klasse System.Collections.ArrayList, die Arrays *variabler* Länge anbietet (siehe Abschnitt 4.1.5). Die Operation Add kann dazu benutzt werden, Elemente beliebigen Typs an das Array anzufügen. Die Elemente können dann durch Indizierung angesprochen werden:

```
using System;
using System.Collections;

class Test {
    static void Main() {
        ArrayList a = new ArrayList();      // erzeugt ein leeres Array variabler Länge
        a.Add("Anton");                     // fügt "Anton" an das Array-Ende an
        a.Add("Berta");
        a.Add("Caesar");
        for (int i = 0; i < a.Count; i++)   // a.Count liefert die Anzahl der Elemente
            Console.WriteLine(a[i]);        // Ausgabe: "Anton", "Berta", "Caesar"
    }
}
```

Assoziative Arrays. Die Klasse System.Collections.Hashtable erlaubt es, Arrays nicht nur mit Zahlen, sondern z.B. auch mit Zeichenketten zu indizieren:

```
using System;
using System.Collections;

class Test {
    static void Main() {
        Hashtable phone = new Hashtable();      // erzeugt leeres assoziatives Array
        phone["Mueller"] = 4362671;
        phone["Maier"] = 2564439;
        phone["Huber"] = 6451162;
        foreach (DictionaryEntry x in phone) {  // foreach: siehe Abschnitt 2.6.9
            Console.Write(x.Key + " = ");       // Schluessel, z.B. "Mueller"
            Console.WriteLine(x.Value);         // Wert, z.B. 4362671
        }
    }
}
```

2.3.4 Strings

Zeichenketten (Strings) kommen so häufig vor, dass es für sie in C# einen eigenen Typ string gibt, der vom Compiler auf die Klasse System.String abgebildet wird. Einer Stringvariablen kann man Stringkonstanten oder andere Stringvariablen zuweisen:

```
string s = "Hello";
string s2 = s;
```

Strings können wie Arrays indiziert werden (z.B. s[i]), sind aber keine Arrays. Insbesondere können sie nicht verändert werden. Wenn man veränderbare Strings braucht, sollte man die Klasse System.Text.StringBuilder benutzen:

```
using System;
using System.Text;

class Test {
    static void Main(string[] arg) {
        StringBuilder buffer = new StringBuilder();
        buffer.Append(arg[0]);
        buffer.Insert(0, "myfiles\\");
        buffer.Replace(".cs", ".exe");
        Console.WriteLine(buffer.ToString());
    }
}
```

Dieses Beispiel zeigt auch, dass man die Methode Main mit einem String-Array als Parameter deklarieren kann, in dem eventuelle Kommandozeilenparameter übergeben werden. Wenn man das obige Programm mit

```
Test sample.cs
```

aufruft, erhält man als Ausgabe myfiles\sample.exe.

Strings sind Referenztypen, d.h., eine Stringvariable enthält eine Referenz auf ein Stringobjekt. Daher sind Stringzuweisungen *Zeigerzuweisungen*; der Wert des Strings wird dabei nicht kopiert. Die Operationen == und != sind allerdings im Gegensatz zu Java *Wertvergleiche*. Der Vergleich

```
(s + " World") == "Hello World"
```

liefert also true. Die Operationen <, <=, > und >= sind auf Strings nicht erlaubt; man muss dazu die Methode CompareTo verwenden (siehe Tabelle 2.3). Strings können mit + verkettet werden (z.B. s + " World" ergibt "Hello World"), wobei ein neues Stringobjekt entsteht (d.h. s wird nicht verändert). Die Länge eines Strings kann wie bei Arrays mit s.Length abgefragt werden. Die Klasse System.String bietet viele nützliche Operationen (siehe Tabelle 2.3):

Tabelle 2.3 *Stringoperationen (Auszug)*

s.CompareTo(s1)	liefert -1, 0 oder 1, je nachdem ob s < s1, s == s1 oder s > s1
s.IndexOf(s1)	liefert den Index des ersten Vorkommens von s1 in s
s.LastIndexOf(s1)	liefert den Index des letzten Vorkommens von s1 in s
s.Substring(from, length)	liefert den Teilstring s[from .. from + length - 1]
s.StartsWith(s1)	liefert true, wenn s mit s1 beginnt
s.EndsWith(s1)	liefert true, wenn s mit s1 endet
s.ToUpper()	liefert eine Kopie von s in Großbuchstaben
s.ToLower()	liefert eine Kopie von s in Kleinbuchstaben
String.Copy(s)	liefert eine Kopie von s

2.3.5 Structs

Structs sind benutzerdefinierte Typen bestehend aus Daten und eventuellen Methoden (Zugriffsoperationen). Sie werden wie folgt deklariert:

```
struct Point {
    public int x, y;                                          // Felder
    public Point(int a, int b) { x = a; y = b; }              // Konstruktor
    public void MoveTo(int x, int y) { this.x = x; this.y = y; }   // Methoden
}
```

Structs sind *Werttypen*. Variablen des Typs Point enthalten daher direkt die Werte der Felder x und y. Eine Zuweisung zwischen Structs ist eine Wertzuweisung und keine Zeigerzuweisung.

```
Point p;            // p ist noch uninitialisiert
p.x = 1; p.y = 2;   // Feldzugriff
Point q = p;        // alle Felder werden zugewiesen (q.x == 1, q.y == 2)
```

Structs können mit Hilfe eines *Konstruktors* initialisiert werden. Die Deklaration

```
Point p = new Point(3, 4);
```

erzeugt ein neues Struct-Objekt am Methodenkeller und ruft den Konstruktor von Point auf, der die Felder mit den Werten 3 und 4 initialisiert. Ein Konstruktor muss immer den gleichen Namen haben wie der Struct-Typ. Die *Methode* MoveTo wird wie folgt aufgerufen:

```
p.MoveTo(10, 20);
```

Im Code der aufgerufenen Methode kann das Objekt p, auf das die Methode angewendet wurde, mittels this angesprochen werden. this.x bezeichnet also das Feld x des Objekts p, während x den Parameter der Methode MoveTo bezeichnet. Wenn keine Verwechslungsgefahr besteht, kann this beim Feldzugriff weggelassen werden, wie das im Konstruktor von Point zu sehen ist.

Structs dürfen keinen parameterlosen Konstruktor deklarieren, sie dürfen ihn aber verwenden, da der Compiler für jeden Struct-Typ einen parameterlosen Konstruktor erzeugt. Der Konstruktor in der Deklaration

```
Point p = new Point();
```

initialisiert die Felder von p mit dem Wert 0. Wir werden in Abschnitt 2.8 noch näher auf Structs und auf Konstruktoren eingehen.

2.3.6 Klassen

Wie Structs sind Klassen benutzerdefinierte Typen aus Daten und eventuellen Zugriffsmethoden. Im Gegensatz zu Structs sind sie allerdings *Referenztypen*, d.h.,

eine Variable vom Typ einer Klasse enthält eine Referenz auf ein Objekt, das am Heap liegt. Klassen werden wie folgt deklariert:

```
class Rectangle {
    Point origin;  // linker unterer Eckpunkt
    public int width, height;
    public Rectangle() { origin = new Point(0, 0); width = height = 1; }
    public Rectangle(Point p, int w, int h) { origin = p; width = w; height = h; }
    public void MoveTo(Point p) { origin = p; }
}
```

Eine Rectangle-Variable kann erst benutzt werden, nachdem man in ihr ein Rectangle-Objekt installiert hat, das mit dem new-Operator erzeugt wurde:

```
Rectangle r = new Rectangle(new Point(10, 20), 5, 5);
int area = r.width * r.height;
```

Nach dem Erzeugen eines Objekts wird automatisch der passende Konstruktor aufgerufen, der die Felder des Objekts initialisiert. Die Klasse Rectangle hat zwei Konstruktoren, die sich in ihrer Parameterliste unterscheiden. Die Parameter des zweiten Konstruktors passen zu den aktuellen Parametern beim Erzeugen des Objekts, daher wird dieser Konstruktor gewählt. Die Deklaration gleichnamiger Konstruktoren oder Methoden in einer Klasse oder einem Struct nennt man *Überladen*. Wir werden darauf in Abschnitt 2.8 näher eingehen.

Erzeugte Objekte werden in C# nie explizit freigegeben. Stattdessen verlässt man sich auf einen so genannten *Garbage Collector*, der Objekte automatisch freigibt, wenn sie nicht mehr referenziert werden. Das vermeidet viele unangenehme Fehler, mit denen C++-Programmierer zu kämpfen haben: Werden Objekte dort zu früh freigegeben, zeigen manche Referenzen ins Leere. Vergisst man, sie freizugeben, bleiben sie als »Leichen« im Speicher zurück. Unter .NET können solche Fehler nicht auftreten, weil einem der Garbage Collector die Freigabe der Objekte abnimmt.

Da Variablen vom Typ einer Klasse Referenzen enthalten, ist die Zuweisung

```
Rectangle r1 = r;
```

eine *Zeigerzuweisung*: r und r1 zeigen anschließend auf dasselbe Objekt. Methoden werden wie bei Structs aufgerufen:

```
r.MoveTo(new Point(3, 3));
```

In der aufgerufenen Methode MoveTo bezeichnet this wieder das Objekt r, für das die Methode aufgerufen wurde. Da dort der Feldname origin aber eindeutig ist, braucht er nicht mit this.origin qualifiziert werden.

Tabelle 2.4 stellt nochmals Klassen und Structs gegenüber. Auf einige der Unterschiede wird erst in späteren Kapiteln eingegangen.

Tabelle 2.4 *Klassen versus Structs*

Klassen	*Structs*
Referenztypen (Variablen referenzieren Objekte am Heap)	Werttypen (Variablen enthalten Objekte)
unterstützen Vererbung (alle Klassen sind von object abgeleitet)	unterstützen keine Vererbung (sind aber zu object kompatibel)
können Interfaces implementieren	können Interfaces implementieren
parameterloser Konstruktor deklarierbar	kein parameterloser Konstruktor deklarierbar

Structs sind leichtgewichtige Typen, die oft für temporäre Daten verwendet werden. Da sie nicht am Heap angelegt werden, belasten sie den Garbage Collector nicht. Klassen werden meist für komplexere Objekte verwendet, die oft zu dynamischen Datenstrukturen verknüpft werden und die Methode überleben können, die sie erzeugt hat.

2.3.7 object

Der Typ object, der vom Compiler auf die Klasse System.Object abgebildet wird, hat in C# eine besondere Bedeutung. Er ist die Wurzel der gesamten Typhierarchie, d.h., alle Typen sind mit ihm kompatibel. Daher darf man einer object-Variablen auch beliebige Werte zuweisen:

```
object obj = new Rectangle();          // Zuweisung von Rectangle an object
Rectangle r = (Rectangle) obj;         // Typumwandlung
obj = new int[3];                      // Zuweisung von int[] an object
int[] a = (int[]) obj;                 // Typumwandlung
```

Insbesondere erlaubt object die Implementierung generischer Container-Klassen. Ein Keller (*Stack*), in dem beliebige Objekte gespeichert werden sollen, kann mit einer Methode

```
void Push(object x) {...}
```

deklariert werden, die man dann mit beliebigen Objekten als Parameter aufrufen kann:

```
Push(new Rectangle());
Push(new int[3]);
```

Die Klasse System.Object, die in Abschnitt 2.9.8 näher behandelt wird, enthält einige Methoden, die von allen Klassen und Structs geerbt (und meistens überschrieben) werden. Die wichtigsten Methoden sind:

```
class Object {
    public virtual bool Equals(object o) {...}      // Wertvergleich
    public virtual string ToString() {...}          // Umwandlung d. Objekts in einen String
    public virtual int GetHashCode() {...}          // Berechnung eines Hashcodes
    ...
}
```

Diese Methoden kann man auf Objekte beliebiger Typen anwenden, sogar auf Konstanten:

```
string s = 123.ToString();  // s == "123"
```

2.3.8 Boxing und Unboxing

Nicht nur Referenztypen, sondern auch Werttypen wie int, Structs oder Enumerationen sind mit object kompatibel. Bei der Zuweisung

```
object obj = 3;
```

wird der Wert 3 in ein Objekt einer implizit erzeugten Hilfsklasse eingewickelt (*boxing*), das dann der Variablen obj zugewiesen wird (siehe Abb. 2.5).

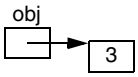


Abb. 2.5 *Boxing eines int-Werts*

Die Zuweisung in der Gegenrichtung erfordert eine Typumwandlung.

```
int x = (int) obj;
```

Dabei wird der Wert aus dem Hilfsobjekt wieder ausgepackt (*unboxing*) und als int-Wert behandelt.

Boxing und Unboxing sind besonders bei generischen Container-Typen nützlich, weil man diese nicht nur mit Elementen eines Referenztyps, sondern auch mit Elementen eines Werttyps füllen kann. Wenn man zum Beispiel eine Klasse Queue wie folgt deklariert,

```
class Queue {
    object[] values = new object[10];
    public void Enqueue(object x) {...}
    public object Dequeue() {...}
}
```

kann man sie folgendermaßen verwenden:

```
Queue q = new Queue();
q.Enqueue(new Rectangle());              // Referenztyp
q.Enqueue(3);                            // Werttyp (boxing)
...
Rectangle r = (Rectangle) q.Dequeue();   // Typumwandlung object -> Rectangle
int x = (int) q.Dequeue();               // Typumwandlung object -> int (unboxing)
```

Aufrufe von object-Methoden werden übrigens an die eingepackten Objekte weitergeleitet. Das Codestück

```
object obj = 123;
string s = obj.ToString();
```

liefert also in s den Wert "123".

2.4 Ausdrücke

Ausdrücke bestehen aus Operanden und Operatoren und berechnen Werte. Tabelle 2.5 zeigt die Operatoren von C# geordnet nach Priorität. Weiter oben genannte Operatoren haben Vorrang vor den weiter unten genannten. Operatoren auf gleicher Ebene werden von links nach rechts ausgewertet.

Tabelle 2.5 *Operatoren nach Priorität geordnet*

Klasse	*Operatoren*
primäre Operatoren	(x) x.y f(x) a[x] x++ x-- new typeof sizeof checked unchecked
unäre Operatoren	+ - ~ ! ++x --x (T)x
Multiplikationsoperatoren	* / %
Additionsoperatoren	+ -
Shift-Operatoren	<< >>
Vergleichsoperatoren	< > <= >= is as
Gleichheitsoperatoren	== !=
bitweises Und	&
bitweises exklusives Oder	^
bitweises Oder	|
bedingtes Und	&&
bedingtes Oder	||
bedingter Ausdruck	condition?value1:value2
Zuweisungsoperatoren	= += -= *= /= %= <<= >>= &= ^= |=

Arithmetische Ausdrücke

Arithmetische Ausdrücke werden mit den Operatoren +, -, *, /, % (Divisionsrest), ++ (Inkrement) und -- (Dekrement) gebildet. Ihre Operanden dürfen von einem beliebigen numerischen Typ oder vom Typ char sein. Bei ++ und -- sind auch Enumerationen erlaubt, allerdings dürfen diese Operatoren nur auf Variablen und nicht auf Konstanten oder Ausdrücke angewendet werden.

Der Ergebnistyp eines arithmetischen Ausdrucks ist der kleinste numerische Typ, der beide Operandentypen einschließt, zumindest aber int. Bei Enumerationen ist der Ergebnistyp wieder derselbe Enumerationstyp. Einige Beispiele mögen das veranschaulichen:

```
short s; int i; float f; char ch; Color c;
... s + f ...          // float
... s + s ...          // int
... i + ch ...         // int
... c++ ...            // Color
```

Beim Rechnen mit vorzeichenlosen Operanden sind einige Ausnahmen zu beachten: Die Negation eines uint-Werts sowie die Verknüpfung eines uint-Werts mit einem sbyte-, short- oder int-Wert gibt einen long-Wert, weil dabei der Wertebereich von uint überschritten werden könnte. Die Negation eines ulong-Werts sowie die Verknüpfung eines ulong-Werts mit einem sbyte-, short-, int- oder long-Wert ist verboten, ebenso wie die Verknüpfung eines decimal-Werts mit einem float- oder double-Wert.

Vergleichsausdrücke

Mit den Operatoren <, >, <= und >= dürfen numerische Operanden oder char-Werte verglichen werden. Auch der Vergleich zwischen zwei Enumerationswerten ist erlaubt. Die Operatoren == (Gleichheit) und != (Ungleichheit) dürfen zusätzlich auch auf boolesche Werte, Referenzen und Delegates angewendet werden. Wenn numerische Operanden von unterschiedlichem Typ sind (z.B. int und long), wird der kleinere Typ vor dem Vergleich in den größeren umgewandelt.

Die Operatoren is und as bewirken einen Typtest und werden in Abschnitt 2.9.2 näher erläutert. Im Ausdruck x is T oder x as T darf x ein Ausdruck beliebigen Typs sein, T muss ein Referenztyp sein, z.B.:

```
r is Rectangle  // referenziert r ein Rectangle-Objekt?
3 is object     // ist 3 vom Typ object?
a is int[]      // referenziert a ein int-Array?
```

Das Ergebnis eines Vergleichs ist immer vom Typ bool.

Boolesche Ausdrücke

Die Operatoren && (Und), || (Oder) und ! (Negation) sind auf boolesche Operanden anwendbar und liefern auch wieder ein Ergebnis vom Typ bool. In Ausdrücken mit && und || findet eine *bedingte Auswertung* (auch *Kurzschlussauswertung* genannt) statt, d.h., die Auswertung wird abgebrochen, sobald das Ergebnis des Ausdrucks feststeht. Formal bedeutet das:

```
a && b      ausgewertet als: wenn (!a) false sonst b
a || b      ausgewertet als: wenn (a) true sonst b
```

Diese Auswertungsreihenfolge ist nützlich, wenn zusammengesetzte Ausdrücke berechnet werden sollen, bei denen der zweite Term nur unter einer bestimmten Bedingung berechnet werden soll, die durch den ersten Term gegeben ist, wie z.B.:

```
if (p != null && p.val > 0) ...      // p.val darf nur berechnet werden, wenn p != null ist
if (y == 0 || x / y > 2) ...         // x / y darf nur berechnet werden, wenn y != 0 ist
```

Der bedingte Ausdruck *condition* ? *value1* : *value2* liefert den Wert *value1*, wenn *condition* wahr ist, sonst *value2*, zum Beispiel:

```
int max = x > y ? x : y;             // if (x > y) max = x; else max = y;
```

Bit-Ausdrücke

Die Operatoren & (bitweises Und), | (bitweises Oder), ^ (bitweises exklusives Oder) und ~ (bitweise Negation) dürfen auf ganzzahlige Operanden, char-Werte und Enumerationen angewendet werden. Die Operatoren &, | und ^ sind außerdem bei booleschen Operanden erlaubt. Es findet keine Kurzschlussauswertung statt. Bei unterschiedlich großen Operandentypen wird der kleinere Typ vor der Auswertung in den größeren konvertiert. Der Ergebnistyp ist der größere der beiden Operandentypen. Bei numerischen oder char-Operanden ist der Ergebnistyp aber zumindest int. Folgende Beispiele zeigen die Funktionsweise von Bit-Ausdrücken unter der Annahme, dass x den Wert 5 und y den Wert 6 hat (in Zweierkomplement-Darstellung):

```
x & y       // 00000101 & 00000110 => 00000100 (4)
x | y       // 00000101  |  00000110 => 00000111 (7)
x ^ y       // 00000101 ^  00000110 => 00000011 (3)
~ x         // ~ 00000101 => 11111010 (-6)
```

Shift-Ausdrücke

Shift-Ausdrücke werden vor allem in der Systemprogrammierung verwendet. Sie erlauben aber auch eine effiziente Multiplikation und Division mit Zweierpotenzen. Der Ausdruck x << y (bzw. x >> y) bedeutet, dass das Bitmuster von x um y Stellen nach links (bzw. nach rechts) verschoben wird. Bei << werden von rechts Nullen nachgeschoben. Bei >> werden von links Bits *b* nachgeschoben, wobei *b* bei vorzeichenlosen Typen 0 und bei vorzeichenbehafteten Typen gleich dem Vorzeichenbit ist. Der Typ von x muss ganzzahlig oder char sein, der Typ von y muss int sein. Der Ergebnistyp ist der Typ von x, zumindest aber int. Folgende Beispiele nehmen an, dass in den int-Variablen a und b die Werte 7 und -7 stehen:

```
a >> 1      // 00000111 >> 1 = 00000011 (3)    = a / 2
a << 3      // 00000111 << 3 = 00111000 (56)   = a * 8
b >> 2      // 11111001 >> 2 = 11111110 (-2)
b << 1      // 11111001 << 1 = 11110010 (-14)
```

Überlaufprüfung

Wird bei arithmetischen Operationen der zulässige Wertebereich überschritten, so tritt kein Fehler auf, sondern das Ergebnis wird einfach abgeschnitten, und das entstehende Bitmuster wird wieder als Zahl interpretiert, z.B.:

```
int x = 1000000;
x = x * x;  // -727379968, keine Fehlermeldung!
```

Dieses Verhalten ist zum Beispiel bei der Implementierung von Zufallszahlengeneratoren erwünscht. Wenn man aber den Überlauf abfangen möchte, kann man das mit dem checked-Operator tun, der wie folgt benutzt wird:

```
x = checked(x * x);       // liefert bei Überlauf System.OverflowException
...
checked {
    ...
    x = x * x;            // liefert bei Überlauf System.OverflowException
    ...
}
```

Man kann die Überlaufprüfung auch für eine ganze Übersetzungseinheit mittels einer Compiler-Option einschalten:

```
csc /checked Test.cs
```

typeof

Der typeof-Operator kann auf einen Typ angewendet werden und liefert das entsprechende Typobjekt (vom Typ System.Type) dazu. Dieses Objekt enthält Typinformationen und kann für *Reflection* verwendet werden (siehe Abschnitt 4.5).

```
Type t = typeof(int);
Console.WriteLine(t.Name);  // liefert Int32
```

Der Typ eines *Objekts* kann mit der von object geerbten Methode GetType abgefragt werden, z.B.:

```
Type t = 123.GetType();
Console.WriteLine(t.Name);  // liefert Int32
```

sizeof

Der sizeof-Operator kann auf Werttypen angewendet werden und liefert ihre Größe in Bytes. Da die Typgröße von Structs plattformabhängig sein kann, sind Programme, die sizeof verwenden, schwer portabel. Man sollte diesen Operator daher vermeiden. Wenn man ihn dennoch verwendet, muss man ihn in einen unsafe-Block einschließen:

```
unsafe {
   Console.WriteLine(sizeof(int));  // liefert 4
}
```

Außerdem muss man diesen Code mit der unsafe-Option übersetzen:

```
csc /unsafe Test.cs
```

2.5 Deklarationen

Deklarationen führen Namen ein, definieren deren Typ und geben manchmal auch bereits einen Anfangswert vor, z.B.:

```
int x = 3;            // deklariert eine int-Variable x mit dem Wert 3
void Foo() {...}      // deklariert eine Methode Foo ohne Rückgabewert
```

Jeder Name gehört zu einem bestimmten *Deklarationsbereich*, von denen es in C# vier gibt:

- *Namensraum*. Kann Klassen, Interfaces, Structs, Enumerationen, Delegates sowie weitere Namensräume enthalten.
- *Klasse, Interface, Struct*. Kann Felder, Methoden, Konstruktoren, Destruktoren, Properties, Indexers, Events und eingeschachtelte Typen enthalten.
- *Enum*. Enthält Enumerationskonstanten.
- *Block*. Kann lokale Variablen enthalten.

Für Deklarationen gelten folgende zwei Regeln:

1. Kein Name darf in einem Deklarationsbereich mehrfach deklariert werden.
2. Die Reihenfolge der Deklarationen ist beliebig, außer bei lokalen Variablen, die vor ihrer Verwendung deklariert werden müssen.

Der Deklarationsbereich, zu dem ein Name gehört, bestimmt seine *Sichtbarkeit*:

1. Ein Name ist in seinem gesamten Deklarationsbereich sichtbar (lokale Variablen allerdings erst ab ihrer Deklaration).
2. Kein Name ist außerhalb seines Deklarationsbereichs sichtbar.
3. Die Sichtbarkeit von Namen, die in Namensräumen, Klassen, Structs und Interfaces deklariert werden, kann durch Attribute wie public, private oder protected reguliert werden.

Deklarationen in Namensräumen

Namensräume (*namespaces*) dienen zur Gruppierung zusammengehöriger Typdeklarationen. Sie erlauben die Verwendung gleicher Namen in verschiedenen Teilen eines Programms. Indem man seine Klassen und anderen Typen in einem eigenen Namensraum deklariert, vermeidet man Kollisionen mit gleichnamigen Typen aus anderen Namensräumen. Namensräume können auch geschachtelt werden, wodurch man hierarchisch gegliederte Gültigkeitsbereiche von Namen bekommt.

```
namespace A {
    ... Klassen ...
    ... Interfaces ...
    ... Structs ...
    ... Enumerationen ...
    ... Delegates ...
    namespace B {  // voller Name: A.B
        ...
    }
}
```

Ein Namensraum wird mit dem Schlüsselwort namespace und seinem Namen deklariert. Darauf folgen in geschwungenen Klammern die enthaltenen Typen und eventuell geschachtelte Namensräume. Der volle Name des in A eingeschachtelten Namensraums B ist A.B.

Deklariert man einen Typ außerhalb eines Namensraums, gehört er zu einem namenlosen Standardnamensraum. Die Klasse Hello aus Abschnitt 2.1 ist ein Beispiel dafür.

Angenommen, wir hätten einen Namensraum Persons mit den zwei Klassen Student und Teacher.

```
namespace Persons {
    public class Student {...}
    public class Teacher {...}
}
```

Die in Persons deklarierten Typen sind zunächst einmal in Persons sichtbar. Da sie aber als public deklariert sind, können sie auch in einem anderen Namensraum benutzt werden, indem man Persons dort *importiert*.

```
namespace School {
    using Persons;        // importiert alle öffentlichen Typen aus Persons

    class Test {
        Student s;        // Student ist hier bekannt
        Teacher t;        // Teacher ist hier bekannt
        ...
    }
}
```

Anstatt Student und Teacher durch Importieren von Persons sichtbar zu machen, können wir sie auch mit ihrem Namensraum *qualifizieren*.

```
namespace School {
    class Test {
        Persons.Student s;      // Qualifikation eines Typs mit seinem Namensraum
        Persons.Teacher t;
        ...
    }
}
```

Die explizite Qualifikation eines Typnamens wird vor allem dann verwendet, wenn man gleichnamige Typen aus unterschiedlichen Namensräumen an der gleichen Stelle verwenden will.

Fast jedes Programm benötigt den Namensraum System (z.B. um den Typ Console für die Ein-/Ausgabe zu verwenden). Daher beginnt fast jedes Programm mit using System;.

Namensräume werden in Abschnitt 2.13 noch ausführlicher beschrieben.

Deklarationen in Klassen, Structs und Interfaces

Klassen, Structs und Interfaces können folgende Elemente enthalten:

```
class C {
    ... Felder, Konstanten ...
    ... Methoden ...
    ... Konstruktoren, Destruktoren ...
    ... Properties ...
    ... Indexers ...
    ... Events ...
    ... überladene Operatoren ...
    ... eingeschachtelte Typen (Klassen, Structs, Interfaces, Enumerationen, Delegates) ...
}
struct S {
    ... wie bei Klassen ...
}
interface I {  // siehe Abschnitt 2.10
    ... Methoden ...
    ... Properties ...
    ... Indexers ...
    ... Events ...
}
```

Werden Klassen mittels Vererbung erweitert (siehe Abschnitt 2.9) bilden Oberklasse und Unterklasse unterschiedliche Deklarationsbereiche. Man kann daher in beiden Klassen gleiche Namen deklarieren.

Interfaces dürfen keine Datenfelder enthalten. Im Gegensatz zu Java sind aber auch keine Konstanten erlaubt. Als Ersatz dafür können Properties verwendet werden.

Deklarationen in Enumerationstypen

Ein Enumerationstyp darf nur Enumerationskonstanten enthalten. Diese müssen bei der Verwendung mit dem Namen des Enumerationstyps qualifiziert werden.

```
enum E {
    ... Enumerationskonstanten ...
}
```

Deklarationen in Blöcken

Ein Block ist eine in geschweiften Klammern enthaltene Anweisungsfolge und bildet den Rumpf von Methoden oder zusammengesetzten Anweisungen. In einem Block dürfen nur lokale Variablen deklariert werden.

```
void Foo(int x) {                  // Methodenblock
    ... lokale Variablen ...
    if (...) {                     // geschachtelter Block einer if-Anweisung
        ... lokale Variablen ...
    }
    for (int i = 0; ...) {         // geschachtelter Block einer for-Anweisung
        ... lokale Variablen ...
    }
}
```

Wie man sieht, können Blöcke geschachtelt auftreten. Der Deklarationsbereich eines Blocks schließt die Deklarationsbereiche der in ihm eingeschachtelten Blöcke ein. Im Gegensatz zu vielen anderen Sprachen darf ein geschachtelter Block also keinen Namen deklarieren, der bereits im übergeordneten Block deklariert wurde.

Formale Parameter (z.B. `x` in der Methode `Foo`) gehören zum Deklarationsbereich des Methodenblocks, obwohl sie außerhalb von ihm deklariert werden. Ihr Name darf nicht mit anderen in der Methode deklarierten Namen kollidieren.

In ähnlicher Weise gehört der Name einer Laufvariablen einer for-Anweisung (z.B. die Variable `i` der obigen for-Schleife) zum Deklarationsbereich des for-Blocks. Daher dürfen verschiedene for-Anweisungen gleichnamige Laufvariablen deklarieren.

Wie bereits erwähnt, muss die Deklaration einer lokalen Variablen ihrer Verwendung vorausgehen. Folgendes Beispiel zeigt einige Fehler, die durch Nichtbeachtung der obigen Regeln entstehen können und vom Compiler gemeldet werden.

```
void Foo(int a) {
    int b;
    if (...) {
        int b;          // Fehler: b ist bereits in äußerem Block deklariert
        int c;          // soweit o.k., aber siehe später ...
        int d;
    } else {
        int a;          // Fehler: a ist bereits im äußeren Block deklariert
        int d;          // o.k.: keine Überschneidung mit d im letzten Block
    }
    for (int i = 0; ...) {...}
    for (int i = 0; ...) {...}  // o.k.: keine Überschneidung mit i aus letzter Schleife
    int c;              // Fehler: c ist bereits in einem inneren Block deklariert
}
```

2.6 Anweisungen

Die Anweisungsarten von C# unterscheiden sich nur unwesentlich von denen in anderen Programmiersprachen. Neben Zuweisungen und Methodenaufrufen gibt es diverse Arten von Verzweigungen und Schleifen sowie Sprunganweisungen und Anweisungen zur Fehlerbehandlung (*exception handling*).

2.6.1 Leeranweisung

Jede Anweisung wird in C# durch einen Strichpunkt abgeschlossen. Ein Strichpunkt alleine ist eine Leeranweisung, die keine Aktion bewirkt und zum Beispiel gebraucht wird, um einen leeren Schleifenrumpf auszudrücken.

2.6.2 Zuweisung

Eine Zuweisung berechnet einen Ausdruck und weist ihn einer Variablen zu. Die Zuweisung selbst ist ebenfalls ein Ausdruck, so dass Mehrfachzuweisungen möglich sind:

```
x = 3 * y + 1;      // x wird zu 3 * y + 1
a = b = 0;          // Mehrfachzuweisung: a und b werden beide auf 0 gesetzt
```

Der Typ des Ausdrucks muss mit dem Typ der Variablen *zuweisungskompatibel* sein. Das ist der Fall, wenn die beiden Typen gleich sind oder wenn der Typ der Variablen den Typ des Ausdrucks gemäß Abb. 2.3 einschließt (short-Werte dürfen z.B. an int-Variablen zugewiesen werden). Zuweisungskompatibilität ist auch gegeben, wenn der Typ des Ausdrucks eine Unterklasse des Typs der Variablen ist (siehe Abschnitt 2.9).

Zuweisungen können mit verschiedenen binären Operationen kombiniert werden. Zum Beispiel ist

```
x += y;
```

eine Kurzform für

```
x = x + y;
```

Diese Kurzformen sind vor allem dann nützlich, wenn statt x ein zusammengesetzter Name verwendet wird (z.B. a[i].f), weil man dann Schreibarbeit spart und der Compiler die Anweisung leichter optimieren kann. Kombinierte Zuweisungen sind mit den Operatoren +=, -=, *=, /=, %=, <<=, >>=, &=, |= und ^= möglich.

2.6.3 Methodenaufruf

Eine Methode wird durch ihren Namen und eine Parameterliste aufgerufen. Die Details des Methodenaufrufs und der Parameterübergabe behandeln wir in Abschnitt 2.8.3. Hier sehen wir uns nur einige Aufrufbeispiele für Methoden der Klasse String an.

```
string s = "a,b,c";
string[] parts = s.Split(',');          // Aufruf der Objektmethode s.Split (nicht static)
s = String.Join(" + ", parts);          // Aufruf der Klassenmethode String.Join (static)
char[] arr = new char[10];
s.CopyTo(0, arr, 0, s.Length);
```

Wie wir in Abschnitt 2.8.3 sehen werden, können Methoden *statisch* oder *nicht statisch* sein. Eine *nicht statische* Methode (z.B. Split) wird auf ein bestimmtes *Objekt* (z.B. s) angewendet. s.Split(',') bewirkt, dass der in s gespeicherte String bei jedem Vorkommen von ',' aufgespalten wird. Das Ergebnis ist ein Array von Teilstrings (hier "a", "b" und "c").

Eine *statische* Methode (z.B. Join) wird nicht auf ein Objekt, sondern auf eine *Klasse* (z.B. String) angewendet. Sie ist mit einer normalen Prozedur in C vergleichbar. String.Join(" + ", parts) bewirkt, dass die Strings im Array parts mit " + " verkettet werden. Als Ergebnis wird hier "a + b + c" geliefert.

Sowohl Split als auch Join sind *Funktionsmethoden*: Sie werden als Operanden eines Ausdrucks aufgerufen und liefern einen Wert als Ergebnis zurück. Daneben gibt es aber auch Methoden, die *kein* Funktionsergebnis liefern. Sie haben den Methodentyp void und werden als eigenständige Anweisung aufgerufen, z.B. s.CopyTo(from, arr, to, len). Diese Methode kopiert len Zeichen des Strings s ab der Position from in das Array arr beginnend an der Stelle to.

2.6.4 if-Anweisung

Eine if-Anweisung hat die Form

if (*BooleanExpression***)** *Statement* **else** *Statement*

Wenn der boolesche Ausdruck wahr ist, wird der Then-Zweig (das erste Statement) ausgeführt, sonst der Else-Zweig (das zweite Statement). Der Else-Zweig kann fehlen. Wenn der Then- oder Else-Zweig aus mehr als einer Anweisung besteht, muss er als *Anweisungsblock* in geschweiften Klammern geschrieben werden. Hier sind einige Beispiele von if-Anweisungen:

```
if (x > max) max = x;                    // ohne Else-Zweig
if (x > y) max = x; else max = y;        // mit Else-Zweig
if ('0' <= ch && ch <= '9')
   val = ch - '0';
else if ('A' <= ch && ch <= 'F')         // geschachteltes if
   val = 10 + ch - 'A';
else {                                   // Else-Zweig besteht aus einer Anweisungsfolge
   val = 0;
   Console.WriteLine("invalid character: " + ch);
}
```

Im Gegensatz zu C und C++ muss der if-Ausdruck vom Typ bool sein. Es ist z.B. nicht erlaubt, den Wert 0 oder null als false zu interpretieren.

2.6.5 switch-Anweisung

Die switch-Anweisung ist eine *Mehrfachverzweigung*. Sie besteht aus einem Ausdruck und mehreren Anweisungsfolgen, die durch case-Marken eingeleitet werden. Die switch-Anweisung verzweigt zu jener Marke, die dem Wert des Ausdrucks entspricht. Wenn keine der Marken passt, springt sie zur default-Marke, und wenn diese fehlt, an das Ende der switch-Anweisung. Hier ist ein Beispiel:

```
switch (country) {
   case "Germany": case "Austria": case "Switzerland":
      language = "German";
      break;
   case "England": case "USA":
      language = "English";
      break;
   case null:
      Console.WriteLine("no country specified");
      break;
   default:
      Console.WriteLine("don't know language of " + country);
      break;
}
```

Der Ausdruck im Kopf der switch-Anweisung muss numerisch, eine Enumeration oder vom Typ char oder string sein. Die case-Marken müssen disjunkte Konstantenwerte sein, deren Typ zum Typ des Ausdrucks zuweisungskompatibel ist.

Im Gegensatz zu den meisten anderen Sprachen erlaubt C# switch-Anweisungen mit Ausdrücken vom Typ string (einschließlich case-Marken mit dem Wert null). In diesem Fall wird die switch-Anweisung aber vom Compiler wie mehrere geschachtelte if-Anweisungen behandelt, während sie sonst durch einen direkten Sprung zur passenden case-Marke implementiert wird.

Jede Anweisungsfolge zwischen den case-Marken *muss* durch einen Aussprung beendet werden. Die häufigste Art dieses Sprungs ist die *break-Anweisung*, die zum Ende der switch-Anweisung springt. Es sind aber auch return-, goto- oder throw-Anweisungen erlaubt, die wir später behandeln werden. Im Gegensatz zu den meisten anderen Sprachen lässt es C# nicht zu, dass ein Programm über eine case-Marke hinweg in die nächste Anweisungsfolge läuft. Dieser »*Fall Through*« muss – falls gewünscht – durch eine goto-Anweisung implementiert werden.

2.6.6 while-Anweisung

Die while-Anweisung ist die häufigste Form einer Schleife. Sie besteht aus einem booleschen Ausdruck und einem Schleifenrumpf, der ausgeführt wird, solange der Ausdruck wahr ist. Der Ausdruck wird *vor* jedem Schleifendurchlauf geprüft. Wenn der Schleifenrumpf aus mehreren Anweisungen besteht, muss er als Anweisungsblock in geschweiften Klammern geschrieben werden.

```
while (x > y) x = x / 2;       // Schleifenrumpf besteht aus einer einzigen Anweisung
while (i < n) {                // Schleifenrumpf besteht aus einer Anweisungsfolge
   sum += i;
   i++;
}
```

2.6.7 do-while-Anweisung

Die do-while-Anweisung unterscheidet sich von der while-Anweisung nur dadurch, dass der boolesche Ausdruck *nach* jedem Schleifendurchlauf geprüft wird. Der Schleifenrumpf wird also zumindest einmal ausgeführt, weshalb diese Schleifenform auch *Durchlaufschleife* heißt, im Gegensatz zur while-Anweisung, die man *Abweisschleife* nennt, weil der Schleifenrumpf unter Umständen kein einziges Mal ausgeführt wird.

```
do i = 10 * i while (i < n);   // Schleifenrumpf besteht aus einer einzigen Anweisung
do {                           // Schleifenrumpf besteht aus einer Anweisungsfolge
   sum += a[i]; i--;
} while (i >= 0);
```

2.6.8 for-Anweisung

Die for-Anweisung ist die flexibelste, aber auch die komplizierteste aller Schleifenarten. Sie hat die Form

for (*Initialization***;** *Condition***;** *Increment***)** *Statement*

Vor dem ersten Schleifendurchlauf wird die Initialisierung ausgeführt, die meist aus der Zuweisung eines Werts an eine Laufvariable besteht. Vor jedem Schleifendurchlauf wird eine Bedingung geprüft und am Ende jedes Durchlaufs der Inkrementierungsteil ausgeführt. Der Schleifenrumpf wird so lange ausgeführt, solange die Bedingung wahr ist. Die Anweisung

```
for (int i = 0; i < n; i++)
    sum += i;
```

kann als Kurzform für die while-Schleife

```
int i = 0;
while (i < n) {
    sum += i;
    i++;
}
```

betrachtet werden und wird vom Compiler auch tatsächlich so behandelt (bis auf die Tatsache, dass i lokal zur for-Schleife ist). Sowohl der Initialisierungsteil als auch der Inkrementierungsteil kann aus mehreren Anweisungen bestehen, die aber durch Kommas getrennt und nicht durch Strichpunkte abgeschlossen werden, z.B.:

```
for (int i = 0, j = n-1; i < n; i++, j--)
    sum += a[i] + b[j];
```

2.6.9 foreach-Anweisung

Die foreach-Anweisung bietet eine bequeme Möglichkeit, über die Elemente eines Arrays, eines Strings oder einer sonstigen Sammlung von Elementen zu iterieren, die das Interface IEnumerable (Abschnitt 4.1.2) implementiert. Sie hat die Form

foreach (*ElementVarDecl* **in** *Collection***)** *Statement*

Hier sind zwei Beispiele für foreach-Anweisungen:

```
int[] a = {3, 17, 4, 8, 2, 29};
sum = 0;
foreach (int x in a) sum += x;

string s = "Hello";
foreach (char ch in s) Console.WriteLine(ch);
```

Die erste Schleife addiert alle Elemente des Arrays a, die zweite gibt alle Zeichen des Strings s aus. Das folgende Beispiel ist ebenfalls interessant:

```
Queue q = new Queue();
q.Enqueue("John"); q.Enqueue("Alice"); ...
foreach (string s in q) Console.WriteLine(s);
```

Die Elemente einer Queue werden als Daten vom Typ object gespeichert. Der Compiler erkennt aber, dass die Laufvariable s der foreach-Schleife vom Typ string ist und sorgt für eine geprüfte Umwandlung der Elemente von object nach string.

2.6.10 break- und continue-Anweisungen

Die break-Anweisung haben wir bereits im Zusammenhang mit der switch-Anweisung kennen gelernt. Sie kann aber auch verwendet werden, um eine Schleife vorzeitig zu verlassen:

```
for (;;) {
    int x = stream.ReadByte();
    if (x < 0) break;
    sum += x;
}
```

Dieses Beispiel zeigt auch die Verwendung der for-Anweisung als *Endlosschleife*. Wenn der Initialisierungsteil, die Abbruchbedingung und der Inkrementierungsteil fehlen, kreist die Schleife für immer, bis sie wie in diesem Fall durch eine break-Anweisung verlassen wird. Die break-Anweisung kann auch in while-, do-while- und foreach-Schleifen verwendet werden. Bei geschachtelten Schleifen bewirkt sie allerdings nur das Verlassen der innersten Schleife. Will man eine äußere Schleife verlassen, muss man das mit einer goto-Anweisung tun.

Die continue-Anweisung darf ebenfalls in jeder Schleifenart verwendet werden und bewirkt, dass der Rest des Schleifenrumpfes übersprungen wird. Es wird dann ein eventueller Inkrementierungsteil ausgeführt (nur bei der for-Schleife) und die Abbruchbedingung geprüft, bevor der nächste Schleifendurchlauf beginnt.

2.6.11 goto-Anweisung

Die goto-Anweisung springt zu einer Marke, die vor einer anderen Anweisung steht. Eine Marke besteht aus einem Namen gefolgt von einem Doppelpunkt. Die do-while-Anweisung könnte z.B. auch mit Sprüngen und Marken codiert werden:

```
top:                          //  do {
    sum += i;                 //      sum += i;
    i++;                      //      i++;
    if (i <= n) goto top;     //  } while (i <= n);
```

Dies ist aber nicht empfehlenswert, da man auf diese Weise die Programmstruktur nur schwer erkennt. Eine sinnvolle Anwendung der goto-Anweisung ist zum Beispiel der Aussprung aus einer geschachtelten Schleife, der mit einer break-Anweisung nicht zu bewerkstelligen ist.

Da die uneingeschränkte Verwendung von goto-Anweisungen die Programmqualität beeinträchtigen kann, unterliegen Sprünge gewissen Restriktionen. So ist es zum Beispiel verboten, in einen Block hineinzuspringen. Ebenso ist der Aussprung aus einem finally-Block bei der Fehlerbehandlung verboten (siehe Abschnitt 2.12).

Goto-Anweisungen kann man auch benutzen, um innerhalb einer switch-Anweisung zu einer case-Marke zu springen. Dies ist vielleicht die sinnvollste Anwendung eines goto, weil sich auf diese Weise so genannte *endliche Automaten* effizient implementieren lassen. Ein endlicher Automat besteht aus nummerierten Zuständen, zwischen denen Übergänge definiert sind, die durch Lesen bestimmter Symbole ausgelöst werden können. Abb. 2.6 zeigt so einen Automaten, bei dem die Zustände durch Kreise und die Übergänge durch Pfeile dargestellt sind.

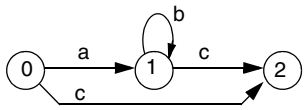


Abb. 2.6 *Endlicher Automat*

Dieser Automat kann mit Hilfe von goto-Anweisungen folgendermaßen implementiert werden:

```
int state = 0;                  // beginnt in Zustand 0
int ch = Console.Read();        // liest erstes Übergangssymbol
switch (state) {
    case 0:  if (ch == 'a') { ch = Console.Read(); goto case 1; }
             else if (ch == 'c') goto case 2;
             else goto default;
    case 1:  if (ch == 'b') { ch = Console.Read(); goto case 1; }
             else if (ch == 'c') goto case 2;
             else goto default;
    case 2:  Console.WriteLine("input valid!");
             break;
    default: Console.WriteLine("illegal character: " + (char) ch);
             break;
}
```

Noch kürzer geht es in diesem Fall, wenn man auf die switch-Anweisung verzichtet und direkt Marken mit dem Namen state0, state1, state2 und illegal anspringt.

2.6.12 return-Anweisung

Die return-Anweisung dient zum (vorzeitigen) Beenden von Methoden. Sie kann in zwei Formen auftreten. Methoden *ohne* Rückgabewert *können* durch eine return-Anweisung ohne Argument vorzeitig beendet werden, z.B.:

```
void P(int x) {
    if (x < 0) return;
    ...
}
```

Die Methode endet natürlich auch ohne return nach ihrer letzten Anweisung. Funktionsmethoden *mit* Rückgabewert *müssen* durch eine return-Anweisung beendet werden, die einen Rückgabewert als Argument enthält, z.B.:

```
int Max(int a, int b) {
    if (a > b) return a; else return b;
}
```

Der Typ des Funktionswerts muss zu dem im Methodenkopf deklarierten Rückgabetyp zuweisungskompatibel sein. Eine Funktionsmethode darf nicht ohne return enden, da ja ein Funktionswert zurückgegeben werden muss.

Auch die Hauptmethode Main eines Programms darf als Funktionsmethode deklariert werden. Der Funktionswert wird dann als Fehlercode interpretiert, der in einer Umgebungsvariablen des Betriebssystems (unter Windows die Variable errorlevel) gespeichert wird.

```
class Test {
    static int Main() {
        ...
        if (...) return -1;
        ...
    }
}
```

2.7 Ein-/Ausgabe

C# besitzt wie die meisten modernen Sprachen keine Anweisungen für die Ein- und Ausgabe von Daten. Vielmehr gibt es dafür Bibliotheksklassen. Wir sehen uns in diesem Kapitel an, wie man auf den Bildschirm schreibt, von der Tastatur liest sowie Dateien für die Ein- und Ausgabe benutzt. Abschnitt 4.2 geht noch ausführlicher auf die Ein-/Ausgabe ein.

2.7.1 Ausgabe auf den Bildschirm

Das Konsolenfenster auf dem Bildschirm wird durch die Klasse System.Console repräsentiert. Diese Klasse enthält zwei Ausgabemethoden:

```
Console.Write(x);       // schreibt den Wert von x in das Konsolenfenster
Console.WriteLine(x);   // schreibt den Wert von x in das Konsolenfenster und
                        // macht anschließend einen Zeilenvorschub
```

Der Parameter dieser beiden Methoden darf von jedem beliebigen Standardtyp (int, char, float, ...) sowie vom Typ string sein. Wenn der Parameter x von einem anderen Typ ist (z.B. ein Referenztyp), wird vorher die Methode x.ToString() aufgerufen, die x vor der Ausgabe in einen String konvertiert.

Formatierte Ausgabe

Write und WriteLine erlauben auch die formatierte Ausgabe einer variablen Anzahl von Werten. Zu diesem Zweck gibt man als Parameter neben den auszugebenden Werten auch einen Formatstring an, der nummerierte Platzhalter für die einzelnen Werte enthält. Der Aufruf

```
Console.WriteLine("{0} = {1}", x, y);
```

ersetzt {0} durch x und {1} durch y und gibt den Ergebnisstring dann aus. Hier ist ein Beispiel, das die Werte eines Arrays in ansprechender Form ausgibt:

```
using System;

class Test {
    static void Main() {
        string[] a = {"Anton", "Berta", "Caesar"};
        for (int i = 0; i < a.Length; i++)
            Console.WriteLine("a[{0}] = {1}", i, a[i]);
    }
}
```

Die Ausgabe sieht wie folgt aus:

```
a[0] = Anton
a[1] = Berta
a[2] = Caesar
```

Platzhalter-Syntax

Die Platzhalter in Formatstrings erlauben auch die Angabe einer Feldbreite, eines Ausgabeformats und der gewünschten Anzahl von Nachkommastellen. Ihre Syntax

```
"{" n ["," width] [":" format [precision]] "}"
```

hat folgende Bedeutung (die eckigen Klammern geben an, dass der geklammerte Teil fehlen kann):

n	Argumentnummer (beginnend bei 0)
width	Feldbreite (wenn zu klein, wird sie überschritten); positiv = rechtsbündig, negativ = linksbündig
format	Formatierungscode, z.B. d, f, e, x, ... (siehe unten)
precision	Anzahl der Nachkommastellen (manchmal Anzahl der Ziffern)

Der Platzhalter

```
{0,10:f2}
```

bewirkt z.B. die rechtsbündige Ausgabe des Arguments 0 in einem Feld der Breite 10 im Fixpunktformat f mit 2 Nachkommastellen. Tabelle 2.6 zeigt die wichtigsten Formatierungscodes und ihre Bedeutung.

Tabelle 2.6 *Die wichtigsten Formatierungscodes*

Code	*Bedeutung*	*Ausgabebild*
d, D	**Dezimalformat** (ganze Zahl mit führenden Nullen) precision = Ziffernanzahl	-xxxxx
f, F	**Fixpunktformat** precision = Anzahl Nachkommastellen (Standard = 2)	-xxxxx.xx
n, N	**Nummernformat** (mit Tausender-Trennzeichen) precision = Anzahl Nachkommastellen (Standard = 2)	-xx,xxx.xx
e, E	**Gleitkommaformat** (groß/klein ist signifikant) precision = Anzahl Nachkommastellen	-x.xxxE+xxx
c, C	**Währungsformat** precision = Anzahl Nachkommastellen (Standard = 2) Negative Werte werden in Klammern gesetzt	$xx,xxx.xx ($xx,xxx.xx)
x, X	**Hexadezimalformat** (groß/klein ist signifikant) precision = Anzahl Hex-Ziffern (eventuell führende Nullen)	xxxx
g, G	**General** (kompaktestes Format für gegebenen Wert; Standard)	

Folgende Beispiele zeigen die Verwendung der formatierten Ausgabe:

```
int x = 26;

Console.WriteLine("{0}", x);          // 26
Console.WriteLine("{0, 5}", x);       //    26
```

```
Console.WriteLine("{0:d}", x);          // 26
Console.WriteLine("{0:d5}", x);         // 00026

Console.WriteLine("{0:f}", x);          // 26.00
Console.WriteLine("{0:f1}", x);         // 26.0

Console.WriteLine("{0:E}", x);          // 2.600000E+001
Console.WriteLine("{0:e1}", x);         // 2.6e+001

Console.WriteLine("{0:X}", x);          // 1A
Console.WriteLine("{0:x4}", x);         // 001a
```

Stringformatierung

Numerische Werte können mittels ToString in eine Zeichenkette konvertiert werden. Dabei kann man wie bei Write und WriteLine Formatierungscodes angeben, die das Format, die Feldbreite und die Nachkommastellen der Zeichenkette spezifizieren. Hier sind einige Beispiele:

```
string s;
int i = 12;
s = i.ToString();           // "12"
s = i.ToString("x4");       // "000c"
s = i.ToString("f");        // "12.00"
```

ToString erlaubt sogar eine länderspezifische Formatierung, die sich z.B. auf das Währungsformat auswirkt:

```
s = i.ToString("c");                                    // "$12.00" -- US-Format
s = i.ToString("c", new CultureInfo("en-GB"));          // "£12.00" -- britisches Format
```

Die Klasse String besitzt schließlich noch eine Methode Format zur allgemeinen Formatierung von Zeichenketten mittels Formatstrings, z.B.:

```
s = String.Format("{0} = {1,6:x4}", name, i);   // "myName =    000c"
```

2.7.2 Ausgabe auf eine Datei

Die Methoden Write und WriteLine können auch zur formatierten Ausgabe auf eine Datei verwendet werden. Dazu muss man vorher einen Ausgabestrom vom Typ FileStream anlegen und einen StreamWriter darauf setzen, der wie die Klasse Console die Methoden Write und WriteLine besitzt. Das folgende Beispiel erzeugt im Applikationsverzeichnis eine Datei namens "myfile.txt" und gibt dort eine Tabelle mit den Quadraten der ersten 10 natürlichen Zahlen aus.

```
using System.IO;  // importiert FileStream und StreamWriter
class Test {
    static void Main() {
        FileStream s = new FileStream("myfile.txt", FileMode.Create);
        StreamWriter w = new StreamWriter(s);
        w.WriteLine("Table of squares:");
        for (int i = 0; i < 10; i++)
            w.WriteLine("{0,3}: {1,5}", i, i*i);
        w.Close();  // nötig, damit der letzte Ausgabepuffer geleert wird
    }
}
```

2.7.3 Eingabe von der Tastatur

Die Klasse System.Console kann nicht nur zur Ausgabe auf den Bildschirm, sondern auch zur Eingabe von der Tastatur verwendet werden. Die Methode

```
int ch = Console.Read();
```

liefert bei jedem Aufruf das nächste Zeichen der Eingabe als int-Zahl. Am Ende der Eingabe (das man unter Windows durch Strg-Z am Zeilenanfang eingibt) liefert sie -1. Read blockiert so lange, bis der Benutzer die Return-Taste drückt und liest dann nacheinander die Zeichen der eingegebenen Zeile. Lautet die Eingabe

```
abc
```

gefolgt von der Return-Taste, so liefert Read der Reihe nach 'a', 'b', 'c', '\r' und '\n'. Will man eine ganze Zeile auf einmal lesen, so kann man die Methode

```
string line = Console.ReadLine();
```

verwenden. Sie blockiert wieder so lange, bis der Benutzer die Return-Taste drückt, und gibt dann die eingegebene Zeile ohne '\r' und '\n' zurück. Am Ende der Eingabe liefert ReadLine den Wert null.

Leider gibt es in der derzeitigen Bibliothek keine Methoden zum formatierten Lesen von Zahlen oder booleschen Werten. Man muss solche Methoden selbst implementieren, z.B.:

```
static int ReadInt() {
    int ch = Console.Read();
    while (0 <= ch && ch <= ' ') ch = Console.Read();  // bedeutungslose Zeichen überlesen
    int val = 0;
    while ('0' <= ch && ch <= '9') {
        val = 10 * val + (ch - '0');
        ch = Console.Read();
    }
    return val;
}
```

2.7.4 Eingabe von einer Datei

Die Methoden Read und ReadLine können auch zum Lesen von einer Datei verwendet werden. Man muss dazu einen Eingabestrom vom Typ FileStream öffnen und einen StreamReader darauf setzen, zu dem Read und ReadLine gehören:

```
using System.IO;    // importiert FileStream und StreamReader
class Test {
    static void Main() {
        FileStream s = new FileStream("myfile.txt", FileMode.Open);
        StreamReader r = new StreamReader(s);
        string line = r.ReadLine();
        while (line != null) {
            ...
            line = r.ReadLine();
        }
        r.Close();
    }
}
```

Leider können nicht mehrere StreamReader gleichzeitig auf eine Datei gesetzt werden, um Daten von unterschiedlichen Stellen der Datei zu lesen.

2.7.5 Lesen der Kommandozeilenparameter

Ein Programm wird durch Angabe seines Namens aus der Kommandozeile aufgerufen. Hinter dem Programmnamen kann noch Text stehen, der in Worte zerlegt wird, die durch Leerzeichen voneinander getrennt sind. Diese *Kommandozeilenparameter* werden als String-Array an die Main-Methode übergeben:

```
using System;
class Test {
    static void Main(string[] arg) {
        for (int i = 0; i < arg.Length; i++)
            Console.WriteLine("{0}: {1}", i, arg[i]);
        foreach (string s in arg) Console.Write(s + " ");
    }
}
```

Ruft man dieses Programm in der Form

```
Test value = 3
```

auf, so liefert es

```
0: value
1: =
2: 3
value = 3
```

2.8 Klassen und Structs

Sowohl Klassen als auch Structs sind Typen, die Daten und dazugehörige Operationen zu einer Einheit zusammenfassen. Beide können folgende Elemente enthalten, die wir in diesem Abschnitt genauer betrachten:

- Felder und Konstanten
- Methoden
- Konstruktoren und Destruktoren
- Properties
- Indexer
- Events
- überladene Operatoren
- geschachtelte Typen (Klassen, Structs, Interfaces, Enumerationen, Delegates)

Hier ist ein Beispiel einer Klasse Counter, die Werte kumuliert und ihren Mittelwert berechnet:

```
class Counter {
   public int value = 0;                              // Felder
   private int n = 0;
   public void Add(int x) { value += x; n++; }        // Methoden
   public float Mean() { return (float) value / n; }
}
```

Bevor ein Counter-Objekt benutzt werden kann, muss es auf folgende Weise erzeugt werden:

```
Counter c = new Counter();
```

Anschließend kann man auf die öffentlichen Felder und Methoden zugreifen, also:

```
c.Add(3); c.Add(17); ...
Console.WriteLine("Summe: {0}, Mittelwert: {1}", c.value, c.Mean());
```

Klassen sind *Referenztypen*. Ihre Objekte liegen am Heap und werden über Zeiger referenziert. Structs sind hingegen *Werttypen*. Ihre Objekte liegen am Methodenkeller (oder sind in andere Objekte eingebettet) und werden direkt in Variablen gespeichert. Folgendes Beispiel zeigt einen Struct-Typ für Bruchzahlen:

```
struct Fraction {
   public int z, n;                                          // Felder (Zähler, Nenner)
   public Fraction(int z, int n) { this.z = z; this.n = n; }  // Konstruktor
   public Add(Fraction f) { z = z * f.n + n * f.z; n = n * f.n; }// Methoden
   public Multiply(Fraction f) { z = z * f.z; n = n * f.n; }
}
```

Deklariert man eine Struct-Variable, sind ihre Felder noch uninitialisiert. Man kann ihnen aber Werte zuweisen, z.B.:

```
Fraction f;
f.z = 1; f.n = 1;
```

Besser ist jedoch die Initialisierung mittels eines *Konstruktors* (siehe Abschnitt 2.8.4):

```
Fraction f = new Fraction();          // erzeugt hier keine neuen Objekte, sondern initialisiert
Fraction g = new Fraction(1, 2);      // die Objekte f und g am Methodenkeller;
g.Add(new Fraction(1, 3));            // erzeugt ein neues Objekt am Methodenkeller
```

Jeder Struct-Typ hat automatisch einen parameterlosen Standardkonstruktor, der die Felder mit 0 initialisiert. Der bei der Erzeugung von g aufgerufene Konstruktor setzt g.z auf 1 und g.n auf 2. Der Aufruf g.Add(new Fraction(1, 3)) erzeugt am Methodenkeller ein neues Fraction-Objekt mit dem Wert 1/3 und addiert es zu g, so dass g.z den Wert 5 und g.n den Wert 6 bekommt.

Die Initialisierung von Struct-Feldern bei ihrer Deklaration ist aufgrund einer Einschränkung der CLR verboten. Der Compiler würde also in folgendem Fall einen Fehler melden:

```
struct Fraction {
    public int z = 1;   // Fehler!
    public int n = 1;   // Fehler!
    ...
}
```

Objekte von Structs, die am Methodenkeller liegen, werden am Ende der Methode automatisch freigegeben. Objekte von Klassen, die am Heap liegen, werden nie explizit freigegeben, sondern vom Garbage Collector entfernt, sobald sie nicht mehr referenziert werden.

2.8.1 Sichtbarkeitsattribute

Klassen und Structs haben auch die Aufgabe, von der tatsächlichen Implementierung ihrer Daten zu abstrahieren (*Datenabstraktion*), indem sie diese vor anderen Programmteilen verbergen (*Information Hiding*). Zu diesem Zweck kann die Sichtbarkeit von Deklarationen mittels Attributen festgelegt werden.

public. Alles, was mit dem Sichtbarkeitsattribut public deklariert wurde, ist überall sichtbar, wo der deklarierende Namensraum bekannt ist. Die Elemente eines Interface und eines Enumerationstyps sind automatisch public. Die Elemente von Structs und Klassen müssen hingegen explizit als public deklariert werden, wenn man sie öffentlich zugänglich machen will, ansonsten sind sie private.

Typen auf äußerster Ebene eines Namensraums (Klassen, Structs, Interfaces, Enumerationen und Delegates) können ebenfalls als public deklariert werden. Andernfalls haben sie die Sichtbarkeit internal, was bedeutet, dass sie nur innerhalb des Assemblies bekannt sind, in dem sie deklariert sind. Ein *Assembly* besteht grob

gesagt aus all jenen Programmteilen, die zusammen übersetzt werden. In Abschnitt 2.13 werden wir uns Assemblies näher ansehen.

Öffentliche Elemente eines Typs sind natürlich nur dort sichtbar, wo auch der Typ, in dem sie enthalten sind, sichtbar ist.

private. Alles, was mit dem Sichtbarkeitsattribut private deklariert wurde, ist nur in der Klasse oder im Struct sichtbar, in dem es deklariert ist. Die Elemente einer Klasse oder eines Structs haben standardmäßig die Sichtbarkeit private. Will man sie öffentlich machen, muss man sie als public deklarieren. Hier ist ein Beispiel für die Verwendung von Sichtbarkeitsattributen:

```
struct Fraction {                       // internal
    public int z, n;                    // public
    private int auxiliary1;             // private
    int auxiliary2;                     // private
    void Reduce() {...}                 // private
    public void Add(Fraction f) {...}   // public
}
```

Folgendes Beispiel zeigt nochmals, wo überall auf private Elemente zugegriffen werden darf:

```
class A {
    private int x;
}

class C {
    private int y;

    public void F(C c) {
        y = ...;            // C-Methode darf auf private Elemente von C zugreifen.
        c.y = ...;          // C-Methode darf auf private Elemente eines anderen
                            // C-Objekts zugreifen.
        A a = new A();
        a.x = ...;          // Falsch! C-Methode darf nicht auf private Elemente
    }                       // eines A-Objekts zugreifen.
}
```

Neben den Sichtbarkeitsattributen public, private und internal gibt es noch die Attribute protected und protected internal, die in Abschnitt 2.9.1 besprochen werden.

2.8.2 Felder

Die Daten innerhalb von Klassen und Structs heißen *Felder* und können Variablen oder Konstanten sein. Folgendes Beispiel zeigt einige erlaubte Deklarationen:

```
class C {
    int value = 0;                                      // Variable
    const long size = ((long)int.MaxValue + 1) / 4);    // Konstante
    readonly DateTime date;                             // Readonly-Variable
    ...
}
```

Variablen. Feldvariablen wie value werden durch ihren Typ und ihren Namen deklariert. In Klassen dürfen sie auch wie im obigen Beispiel initialisiert werden, in Structs jedoch nicht, was mit einer Implementierungsbeschränkung der Common Language Runtime zu tun hat. Die Initialisierung von Struct-Feldern muss immer in einem Konstruktor erfolgen. Initialisiert man ein Feld bei seiner Deklaration, darf man dabei nicht auf andere Felder und Methoden des gleichen Objekts zugreifen.

Konstanten. Eine Konstante wie size ist ein Name für einen Wert, der zur Compilezeit berechnet und an allen Stellen im Programm eingesetzt wird, an denen dieser Name vorkommt. Konstanten werden mit dem Schlüsselwort const deklariert und müssen dabei initialisiert werden (auch in Structs).

Readonly-Felder. Wenn ein Feld mit dem Schlüsselwort readonly deklariert wird, darf man später nur lesend darauf zugreifen. Der Wert eines solchen Felds muss entweder bei der Deklaration oder in einem Konstruktor zugewiesen werden. Eine spätere Zuweisung ist nicht mehr möglich. Im Gegensatz zu Konstanten wird für Readonly-Felder eine Speicherzelle angelegt, auf die später bei der Verwendung zugegriffen wird. Readonly-Felder werden zur Laufzeit initialisiert, Konstantenfelder zur Compilezeit. Der Vorteil liegt darin, dass der Wert eines Readonly-Felds zum Beispiel von einer Datei eingelesen werden kann, was bei Konstanten nicht möglich ist.

Innerhalb der deklarierenden Klasse können Felder einfach über ihren Namen angesprochen werden (z.B. value oder this.value). Die Felder eines anderen Objekts müssen mit dem Objektnamen qualifiziert werden (z.B. c.value).

Statische Felder

Die Felder einer Klasse können entweder in jedem Objekt oder nur einmal pro Klasse vorhanden sein. Wenn sie nur einmal pro Klasse angelegt werden sollen, heißen sie *statische Felder* und müssen mit dem Zusatz static deklariert werden.

```
class Rectangle {
    static Color defaultColor;          // nur einmal pro Klasse vorhanden
    static readonly int scale = 1;      // nur einmal pro Klasse vorhanden
    int x, y, width, height;            // in jedem Objekt gespeichert
    ...
}
```

Die Felder defaultColor und scale gehören also zur Klasse Rectangle selbst und werden nur einmal angelegt. Die Felder x, y, width und height sind hingegen in jedem Rectangle-Objekt vorhanden (siehe Abb. 2.7).

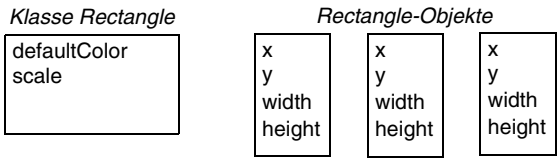


Abb. 2.7 *Statische und nicht statische Felder*

Alle Methoden einer Klasse können auf die statischen Felder dieser Klasse zugreifen (z.B. mittels defaultColor oder Rectangle.defaultColor). Beim Zugriff auf statische Felder einer fremden Klasse muss man den Feldnamen mit dem Klassennamen qualifizieren (z.B. String.Empty).

Für Konstantenfelder wird kein Speicherplatz angelegt. Daher hat der Zusatz static bei ihnen keinen Sinn und ist verboten.

2.8.3 Methoden

Die zu einer Klasse gehörenden Operationen heißen *Methoden*. Sie werden mit ihrem Namen, einer (eventuell leeren) Parameterliste und ihrem Code deklariert.

```
class Counter {
    int sum = 0, n = 0;

    public void Add(int x) {            // Prozedur
        sum = sum + x; n++;
    }

    public float Mean() {               // Funktion (muss Wert mit return zurückgeben)
        return (float)sum / n;
    }
}
```

Methoden können Prozeduren oder Funktionen sein. Eine *Prozedur* wird als eigenständige Anweisung aufgerufen, z.B.:

```
counter.Add(3);
```

Sie liefert keinen Funktionswert und wird daher als void deklariert. Eine *Funktion* wird dagegen als Operand eines Ausdrucks aufgerufen, z.B.:

```
float result = 10 + counter.Mean();
```

Sie liefert einen Rückgabewert (*Funktionswert*) und wird daher mit dem Typ dieses Rückgabewerts deklariert (Mean liefert z.B. ein float-Ergebnis). Das Ergebnis muss mittels einer return-Anweisung zurückgegeben werden, andernfalls meldet der Compiler einen Fehler.

Statische Methoden

Ähnlich wie Felder können auch Methoden entweder der Klasse selbst oder den Objekten dieser Klasse zugeordnet werden. Methoden, die der Klasse zugeordnet sind, heißen *statische Methoden* und werden mit dem Schlüsselwort static deklariert.

```
class Rectangle {
    static Color defaultColor;

    public static void ResetColor() {
        defaultColor = Color.white;
    }
    ...
}
```

Statische Methoden sind auf ihre Klasse anwendbar, während nicht statische Methoden auf Objekte ihrer Klasse angewendet werden. Meist benutzt man statische Methoden dazu, um statische Felder zu initialisieren. Innerhalb der deklarierenden Klasse können sie einfach mit ihrem Namen angesprochen werden (z.B. ResetColor oder Rectangle.ResetColor). Statische Methoden fremder Klassen müssen mit dem Klassennamen qualifiziert werden (z.B. String.Format).

Parameter

Parameter sind Werte, die vom Rufer an eine Methode übergeben oder von der Methode an den Rufer zurückgegeben werden. Bei der Deklaration einer Methode wird eine Liste *formaler Parameter* angegeben, die den *aktuellen Parametern* beim Aufruf der Methode entsprechen. Parameter werden wie Variablen deklariert, aber durch Kommas getrennt, statt durch Strichpunkte abgeschlossen, z.B.:

```
void PrintMessage (string msg, int x, int y) {...}
```

C# kennt drei Arten von Parametern, die auf unterschiedliche Weise übergeben werden.

Wertparameter. Ein Wertparameter ist ein *Eingangsparameter*, der vom Rufer an die Methode übergeben wird (*call by value*). Der aktuelle Parameter darf ein beliebiger Ausdruck sein, dessen Wert vor dem Aufruf berechnet und in den entsprechenden formalen Parameter kopiert wird. Im folgenden Beispiel

```
void Inc(int x) { x = x + 1; }

void F() {
    int val = 3;
    Inc(val);      // val hat nach diesem Aufruf noch immer den Wert 3
}
```

wird der Wert von val in den formalen Parameter x kopiert, bevor Inc aufgerufen wird. Wenn x anschließend erhöht wird, bleibt der Wert von val unverändert, weil x ja eine Kopie von val ist. Die Typen von aktuellen und formalen Parametern müssen übrigens wie bei Zuweisungen kompatibel sein, was hier der Fall ist, weil sowohl val als auch x vom Typ int sind.

Ref-Parameter. Ein Referenzparameter ist ein *Übergangsparameter*, der dazu verwendet werden kann, einen Wert an eine Methode zu übergeben und eventuelle Änderungen, die die Methode an diesem Wert vornimmt, im Rufer weiterzuverarbeiten. Der aktuelle Parameter muss eine Variable sein; der entsprechende formale Parameter kann als Alias-Name für dieselbe Variable betrachtet werden. Technisch wird das gelöst, indem die Adresse des aktuellen Parameters übergeben wird (*call by reference*); der entsprechende formale Parameter bekommt einfach dieselbe Adresse wie der aktuelle Parameter. Sowohl formale als auch aktuelle Referenzparameter müssen mit dem Zusatz ref versehen werden. Im Beispiel

```
void Inc(ref int x) { x = x + 1; }

void F() {
    int val = 3;
    Inc(ref val);          // val hat nach diesem Aufruf den Wert 4
}
```

bezeichnen val und x dieselbe Variable. Wenn also Inc den Wert von x erhöht, wird dadurch auch der Wert von val erhöht. Die Variable val muss bereits vor dem Aufruf von Inc einen Wert haben, der an die Methode übergeben wird. Die Methode ändert diesen Wert und gibt ihn wieder an den Rufer zurück. Bei Referenzparametern müssen der aktuelle und der formale Parameter den gleichen Typ haben, da sie ja dieselbe Variable bezeichnen.

Out-Parameter. Ein Out-Parameter ist ein *Ausgangsparameter*, der von der Methode an ihren Rufer zurückgegeben wird. Die Parameterübergabe erfolgt wie bei Referenzparametern durch *call by reference*. Allerdings muss der aktuelle Parameter vor dem Aufruf noch keinen Wert haben. Dafür prüft der Compiler, dass der formale Parameter in der Methode einen Wert bekommt und liefert anderenfalls eine Fehlermeldung. Wie bei Referenzparametern muss ein aktueller Out-Parameter eine Variable sein, deren Typ gleich ist wie der Typ des entsprechenden formalen Parameters. Out-Parameter werden sowohl bei der Deklaration als auch beim Aufruf mit out gekennzeichnet.

```
void Read(out int first, out int next) {
    first = Console.Read();
    next = Console.Read();
}

void F() {
    int a, b;
    Read(out a, out b);
}
```

In diesem Beispiel sind a und first wieder zwei Namen für dieselbe Variable, ebenso b und next. Out-Parameter werden vor allem dann verwendet, wenn eine Methode mehr als einen Wert liefern soll. Andernfalls kann man das Ergebnis auch als Funktionswert zurückgeben.

Variable Anzahl von Parametern

In C# kann man Methoden deklarieren, die eine variable Anzahl von Parametern haben. Genau genommen darf der jeweils letzte Parameter als Array deklariert werden, für das dann beim Aufruf eine Folge von Einzelwerten angegeben werden kann. Dieser Parameter muss bei der Deklaration mit dem Schlüsselwort params gekennzeichnet sein, wie im folgenden Beispiel:

```
void Add(out int sum, params int[] val) {
    sum = 0;
    foreach (int i in val) sum += i;
}
```

Wenn man die Methode folgendermaßen aufruft

```
Add(out sum, 3, 4, 2, 9);
```

wird die Folge 3, 4, 2, 9 zum int-Array val zusammengefasst, und in sum wird der Wert 18 geliefert.

Andere bekannte Beispiele für eine variable Anzahl von Parametern sind die Methoden Write und WriteLine aus der Klasse Console. Sie sind wie folgt deklariert

```
public void WriteLine(string format, params object[] arg) {...}
```

und können nach dem Formatstring eine beliebige Anzahl von Argumenten aufweisen.

Als Einschränkung muss noch gesagt werden, dass das Schlüsselwort params nicht mit ref oder out kombiniert werden darf. Man kann also nur eine variable Anzahl von Eingangsparametern benutzen, nicht aber eine variable Anzahl von Ausgangs- oder Übergangsparametern.

Überladen von Methoden

Methoden einer Klasse dürfen gleich heißen, wenn sie sich in ihren Parametern unterscheiden, d.h., wenn sie

- unterschiedliche *Anzahl* von Parametern oder
- unterschiedliche Parameter*typen* oder
- unterschiedliche Parameter*arten* (value, ref/out) haben.

Im folgenden Beispiel werden fünf Methoden mit dem Namen P deklariert, die sich nach den oben angegebenen Kriterien unterscheiden.

```
void P(int x) {...}
void P(char x) {...}
void P(int x, long y) {...}
void P(long x, int y) {...}
void P(ref int x) {...}
```

Beim Aufruf der Methode wird vom Compiler die richtige Variante anhand der Typen der aktuellen Parameter ausgewählt, also:

```
int i; long n; short s;
P(i);      // P(int x)
P(ref i);  // P(ref int x)
P('a');    // P(char x)
P(i, n);   // P(int x, long y)
P(n, s);   // P(long x, int y)
P(i, s);   // mehrdeutig zwischen P(int x, long y) und P(long x, int y) => Fehler
P(i, i);   // mehrdeutig zwischen P(int x, long y) und P(long x, int y) => Fehler
```

Wenn die Auswahl nicht eindeutig ist, wie in den letzten beiden Zeilen des Beispiels, meldet der Compiler einen Fehler.

Überladene Methoden dürfen sich nicht bloß im Funktionstyp unterscheiden, wie im folgenden Beispiel:

```
int F() {...}
string F() {...}
```

Da sich jede Funktion auch als Prozedur aufrufen lässt

```
F();  // Rückgabewert wird ignoriert
```

kann der Compiler aus diesem Aufruf nicht erkennen, welche der beiden Funktionen gemeint ist. Daher ist dieser Fall verboten.

Auch die Schlüsselwörter params, ref und out dürfen nicht als Unterscheidungskriterien beim Überladen von Methoden verwendet werden. Die folgenden Deklarationen sind also falsch:

```
void P(int[] a) {...}
void P(params int[] a) {...}      // Fehler: Mehrdeutigkeit
```

```
void Q(ref int x) {...}
void Q(out int x) {...}          // Fehler: Mehrdeutigkeit
```

Der Grund für dieses Verbot ist nicht ganz offensichtlich, denn Q muss ja entweder als Q(ref x); oder als Q(out x); aufgerufen werden, womit eine Unterscheidung möglich sein sollte. Das Verbot kommt aber daher, dass der C#-Compiler diese Konstrukte auf denselben CIL-Code abbildet und der JIT-Compiler dann die beiden Aufrufe nicht mehr unterscheiden kann.

2.8.4 Konstruktoren

Konstruktoren sind spezielle Methoden zur Initialisierung von Objekten. Wir behandeln zuerst Konstruktoren in Klassen, anschließend Konstruktoren in Structs und schließlich statische Konstruktoren.

Konstruktoren in Klassen

Ein Konstruktor ist eine Methode, die bei der Erzeugung eines Objekts automatisch aufgerufen wird, um das Objekt zu initialisieren. Konstruktoren tragen denselben Namen wie die Klasse, in der sie deklariert sind, und haben keinen Funktionstyp. Folgende Klasse Rectangle hat zum Beispiel drei überladene Konstruktoren:

```
class Rectangle {
    int x, y, width, height;
    public Rectangle(int x, int y, int w, int h) { this.x = x; this.y = y; width = w; height = h; }
    public Rectangle(int w, int h) : this(0, 0, w, h) {}
    public Rectangle() : this(0, 0, 0, 0) {}
    ...
}
```

Die Anweisung

```
Rectangle r1 = new Rectangle(2, 2, 10, 5);
```

erzeugt ein neues Rectangle-Objekt und ruft den passenden Konstruktor auf (in diesem Fall den mit vier Parametern), der die Felder x, y, width und height mit den Werten 2, 2, 10 und 5 initialisiert. Erzeugung und Initialisierung eines Objekts werden somit zu einer Einheit verbunden. Erzeugt man ein Objekt mittels

```
Rectangle r2 = new Rectangle(10, 5);
```

so wird der Konstruktor mit zwei Parametern aufgerufen. Im Kopf seiner Deklaration sieht man den Aufruf

```
... : this(0, 0, w, h)
```

was bedeutet, dass vor seiner ersten Anweisung ein anderer Konstruktor dieser Klasse aufgerufen wird, und zwar der mit vier Parametern. Die Implementierung eines Konstruktors kann so auf die Implementierung anderer Konstruktoren zurückgeführt werden. Gleiches gilt für den dritten Konstruktor.

Nach der Erzeugung eines Objekts und vor dem Aufruf des Konstruktors werden noch jene Initialisierungen durchgeführt, die bei der Deklaration der Felder angegeben wurden.

Standardkonstruktor. Hat eine Klasse keinen Konstruktor, wird automatisch ein parameterloser Standardkonstruktor erzeugt. Sieht eine Klasse also wie folgt aus

```
class C {
    int x;
    bool y;
}
```

kann man wie gewohnt ein Objekt davon erzeugen:

```
C obj = new C();
```

Dabei wird der parameterlose Standardkonstruktor aufgerufen, der die Felder je nach Typ folgendermaßen initialisiert:

numerischer Typ:	0
Enumeration:	Konstante, die dem Wert 0 entspricht
bool:	false
char:	'\0'
Referenztyp:	null

Hat eine Klasse zumindest *einen* Konstruktor, so wird *kein* Standardkonstruktor erzeugt. Sieht eine Klasse also wie folgt aus

```
class C {
    int x;
    public C(int y) { x = y; }
}
```

muss man bei der Erzeugung von Objekten einen Parameter angeben:

```
C c1 = new C(3);        // o.k.
C c2 = new C();         // Compiler meldet einen Fehler
```

Konstruktoren in Structs

Grundsätzlich haben Konstruktoren bei Structs dieselbe Form und Bedeutung wie bei Klassen. In Structs dürfen aber keine parameterlosen Konstruktoren deklariert werden (der Grund liegt in der Implementierung der Common Language Runtime). Dafür erzeugt der Compiler für jeden Struct-Typ automatisch einen para-

meterlosen Standardkonstruktor, der die Felder wie oben beschrieben mit Nullwerten initialisiert. Hier ist ein Beispiel:

```
struct Complex {
    double re, im;
    public Complex(double re, double im) { this.re = re; this.im = im; }
    public Complex(double re) : this(re, 0) {}
    ...
}
...
Complex c0;                          // c0.re und c0.im sind uninitialisiert
Complex c1 = new Complex();          // c1.re == 0, c1.im == 0
Complex c2 = new Complex(5);         // c2.re == 5, c2.im == 0
Complex c3 = new Complex(10, 3);     // c3.re == 10, c3.im == 3
```

Statische Konstruktoren

Wie Methoden können auch Konstruktoren mit dem Schlüsselwort static versehen werden und sind dann statisch. Ein statischer Konstruktor dient vor allem zur Initialisierung einer Klasse (d.h. ihrer statischen Felder) und wird automatisch aufgerufen, bevor die Klasse das erste Mal benutzt wird. Hier sind zwei Beispiele:

```
class Rectangle {
    static Color defaultColor;
    static Rectangle() { defaultColor = Color.black; }
    ...
}
struct Point {
    ...
    static Point() { Console.WriteLine("Point initialized"); }
}
```

Statische Konstruktoren müssen parameterlos sein (auch bei Structs) und werden weder als public noch als private deklariert. Es darf nur einen einzigen statischen Konstruktor pro Typ geben.

Bevor ein statischer Konstruktor aufgerufen wird, werden die bei der Deklaration der statischen Felder angegebenen Initialisierungen ausgeführt. Hat eine Klasse keinen statischen Konstruktor, werden alle ihre statischen Felder, die nicht explizit initialisiert wurden, mit 0, false, '\0' etc. initialisiert.

Die Reihenfolge, in der die statischen Konstruktoren der einzelnen Typen aufgerufen werden, ist nicht definiert. Es wird nur garantiert, dass sie aufgerufen werden, bevor die Typen benutzt werden.