O'REILLY®

5. Auflage

C# 7.0

kurz & gut

O'REILLYS TASCHEN-BIBLIOTHEK

Joseph Albahari
Ben Albahari

Übersetzung von
Lars Schulten und Thomas Demmig

C# 7.0

kurz & gut

Joseph Albahari & Ben Albahari

Deutsche Übersetzung von Lars Schulten & Thomas Demmig

Joseph Albahari und Ben Albahari

Lektorat: Alexandra Follenius
Übersetzung: Lars Schulten und Thomas Demmig
Korrektorat: Sibylle Feldmann, www.richtiger-text.de
Herstellung: Susanne Bröckelmann
Umschlaggestaltung: Karen Montgomery, Michael Oréal, www.oreal.de
Satz: III-Satz, www.drei-satz.de
Druck und Bindung: M.P. Media-Print Informationstechnologie GmbH, 33100 Paderborn

Bibliografische Information Der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.d-nb.de* abrufbar.

ISBN:

Print	978-3-96009-072-4
PDF	978-3-96010-174-1
ePub	978-3-96010-175-8
mobi	978-3-96010-176-5

Dieses Buch erscheint in Kooperation mit O'Reilly Media, Inc. unter dem Imprint »O'REILLY«. O'REILLY ist ein Markenzeichen und eine eingetragene Marke von O'Reilly Media, Inc. und wird mit Einwilligung des Eigentümers verwendet.

5. Auflage

Wieblinger Weg 17
69123 Heidelberg

Authorized German translation of the English edition of *C# 7.0 Pocket Reference: Instant Help for C# 7.0 Programmers*, ISBN 978-1-491-98853-4 © 2017 Joseph Albahari, Ben Albahari. This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

5 4 3 2 1 0

Inhalt

C# 7.0 – kurz & gut

C# ist eine allgemein anwendbare, typsichere, objektorientierte Programmiersprache, die die Produktivität des Programmierers erhöhen soll. Zu diesem Zweck versucht die Sprache, die Balance zwischen Einfachheit, Ausdrucksfähigkeit und Performance zu finden. Die Sprache C# ist plattformneutral, wurde aber geschrieben, um gut mit dem *.NET Framework* von Microsoft zusammenzuarbeiten. C# 7.0 ist auf das .NET Framework 4.6/4.7 ausgerichtet.

Die Programme und Codefragmente in diesem Buch entsprechen denen aus den Kapiteln 2 und 4 von *C# 7.0 in a Nutshell* und sind alle als interaktive Beispiele in LINQPad verfügbar. Das Durcharbeiten der Beispiele im Zusammenhang mit diesem Buch fördert den Lernvorgang, da Sie bei der Bearbeitung der Beispiele unmittelbar die Ergebnisse sehen können, ohne dass Sie in Visual Studio dazu Projekte und Projektmappen einrichten müssten.

Um die Beispiele herunterzuladen, klicken Sie in LINQ-Pad auf den Samples-Tab und wählen dort *Download more samples*. LINQPad ist kostenlos – Sie finden es unter *http://www.linqpad.net*.

Ein erstes C#-Programm

Das hier ist ein Programm, das 12 mit 30 multipliziert und das Ergebnis ausgibt (360). Der doppelte Schrägstrich (Slash) gibt an, dass der Rest einer Zeile ein *Kommentar* ist.

```
using System;                   // Importiert den Namensraum

class Test                      // Klassendeklaration
{
    static void Main()          // Methodendeklaration
    {
        int x = 12 * 30;        // Anweisung 1
        Console.WriteLine (x);  // Anweisung 2
    }                           // Ende der Methode
}                               // Ende der Klasse
```

Im Kern dieses Programms gibt es zwei *Anweisungen*. In C# werden Anweisungen nacheinander ausgeführt und jeweils durch ein Semikolon abgeschlossen. Die erste Anweisung berechnet den *Ausdruck* 12 * 30 und speichert das Ergebnis in einer *lokalen Variablen* namens x, die einen ganzzahligen Wert repräsentiert. Die zweite Anweisung ruft die *Methode* WriteLine der *Klasse* Console auf, um die Variable x in einem Textfenster auf dem Bildschirm auszugeben.

Eine Methode führt eine Aktion als Abfolge von Anweisungen aus, die als *Anweisungsblock* bezeichnet wird – ein (geschweiftes) Klammernpaar mit null oder mehr Anweisungen. Wir haben eine einzelne Methode mit dem Namen Main definiert.

Das Schreiben von High-Level-Funktionen, die Low-Level-Funktionen aufrufen, vereinfacht ein Programm. Wir können unser Programm *refaktorieren*, indem wir eine wiederverwendbare Methode schreiben, die einen Integer-Wert mit 12 multipliziert:

```
using System;

class Test
{
    static void Main()
    {
        Console.WriteLine (FeetToInches (30));    // 360
        Console.WriteLine (FeetToInches (100));   // 1200
    }

    static int FeetToInches (int feet)
    {
        int inches = feet * 12;
        return inches;
    }
}
```

Eine Methode kann *Eingabedaten* vom Aufrufenden erhalten, indem sie *Parameter* spezifiziert, und Daten zurück an den Aufrufenden geben, indem sie einen *Rückgabetyp* festlegt. Wir haben eine Methode FeetToInches definiert, die einen Parameter für die Übergabe der Feet und einen Rückgabetyp für die berechneten Inches hat.

Die *Literale* 30 und 100 sind die *Argumente*, die an die Methode FeetToInches übergeben wurden. Die Methode Main hat in unserem Beispiel leere Klammern, da sie keine Parameter besitzt, und sie ist void, weil sie keinen Wert an den Aufrufenden zurückliefert. C# erkennt eine Methode mit dem Namen Main als Angabe des Standardeinstiegspunkts für die Ausführung. Die Methode Main kann optional einen Integer-Wert zurückgeben (statt void), um der Ausführungsumgebung einen Wert zu

übermitteln. Sie kann auch optional ein Array mit Strings als Parameter erwarten (das dann durch die Argumente gefüllt wird, die an die ausführbare Datei übergeben werden). Hier sehen Sie ein Beispiel:

```
static int Main (string[] args) {...}
```

Ein Array (wie zum Beispiel `string[]`) steht für eine feste Zahl an Elementen eines bestimmten Typs (siehe den Abschnitt »Arrays« auf Seite 31).

Methoden sind eine der vielen Arten von Funktionen in C#. Eine andere Art von Funktionen, die wir verwenden, ist der **-Operator*, der dazu dient, Multiplikationen auszuführen. Des Weiteren gibt es noch *Konstruktoren*, *Eigenschaften*, *Events*, *Indexer* und *Finalizer*.

In unserem Beispiel sind die beiden Methoden in einer Klasse zusammengefasst. Eine *Klasse* gruppiert Funktions-Member und Daten-Member zu einem objektorientierten Building-Block. Die Klasse `Console` fasst Member zusammen, die Funktionalität zur Ein- und Ausgabe an der Befehlszeile bieten, zum Beispiel die Methode `WriteLine`. Unsere Klasse `Test` fasst zwei Methoden zusammen – `Main` und `FeetToInches`. Eine Klasse ist eine Art von *Typ*; das wird später im Abschnitt »Typgrundlagen« auf Seite 8 genauer erläutert.

Auf der obersten Ebene eines Programms werden Typen in *Namensräume* eingeteilt. Die `using`-Direktive wird genutzt, um unserer Anwendung den Namensraum `System` verfügbar zu machen, damit sie die Klasse `Console` nutzen kann. Wir können alle von uns bislang definierten Klassen folgendermaßen im `TestPrograms`-Namensraum zusammenfassen:

```
using System;

namespace TestPrograms
{
    class Test {...}

    class Test2 {...}
```

```
}
```

Das .NET Framework ist in hierarchischen Namensräumen organisiert. Dazu gehört zum Beispiel der Namensraum, der die Typen für den Umgang mit Text enthält:

```
using System.Text;
```

Die Direktive `using` dient der Bequemlichkeit – Sie können einen Typ auch über seinen vollständig qualifizierten Namen ansprechen. Das ist der Name des Typs, dem sein Namensraum vorangestellt ist, zum Beispiel `System.Text.StringBuilder`.

Kompilation

Der C#-Compiler führt Quellcode, der in einer Reihe von Dateien mit der Endung *.cs* untergebracht ist, in einer *Assembly* zusammen. Eine Assembly ist die Verpackungs- und Auslieferungseinheit in .NET und kann entweder eine *Anwendung* oder eine *Bibliothek* sein. Eine normale Konsolen- oder Windows-Anwendung hat eine `Main`-Methode und ist eine *.exe*-Datei. Eine Bibliothek ist eine *.dll*-Datei – im Prinzip eine *.exe*-Datei ohne Einsprungpunkt. Ihr Zweck ist es, von einer Anwendung oder anderen Bibliotheken aufgerufen (*referenziert*) zu werden. Das .NET Framework ist eine Sammlung von Bibliotheken.

Der Name des C#-Compilers ist *csc.exe*. Sie können entweder eine integrierte Entwicklungsumgebung (*Integrated Development Environment*, IDE) wie Visual Studio .NET nutzen, damit `csc` automatisch aufgerufen wird, oder den Compiler selbst per Hand über die Befehlszeile aufrufen. Um manuell zu kompilieren, speichern Sie ein Programm zunächst in einer Datei wie *MyFirstProgram.cs* und rufen dann `csc` auf (zu finden unter *%ProgramFiles(X86)%\msbuild\14.0\bin*):

```
csc MyFirstProgram.cs
```

Das erstellt eine Anwendung namens *MyFirstProgram.exe*.

Eine Bibliothek (*.dll*) erstellen Sie mit der folgenden Anweisung:

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

5-Compiler mit. Um den C# 7-Befehlszeilencompiler zu erhalten, müssen Sie Visual Studio oder MSBuild 15 installieren.

Syntax

Die Syntax von C# ist von der Syntax von C und C++ inspiriert. In diesem Abschnitt beschreiben wir die C#-Elemente der Syntax anhand des folgenden Programms:

```
using System;

class Test
{
    static void Main()
    {
        int x = 12 * 30;
        Console.WriteLine (x);
    }
}
```

Bezeichner und Schlüsselwörter

Bezeichner sind Namen, die Programmierer für ihre Klassen, Methoden, Variablen und so weiter wählen. Das hier sind die Bezeichner in unserem Beispielprogramm in der Reihenfolge ihres Auftretens:

```
System  Test  Main  x  Console  WriteLine
```

Ein Bezeichner muss ein ganzes Wort sein und aus Unicode-Zeichen bestehen, wobei den Anfang ein Buchstabe oder der Unterstrich bildet. C#-Bezeichner unterscheiden Groß- und Kleinschreibung. Es ist üblich, Argumente, lokale Variablen und private Felder in Camel-Case zu schreiben (zum Beispiel myVariable) und alle anderen Bezeichner in Pascal-Schreibweise (zum Beispiel MyMethod).

Schlüsselwörter sind Namen, die für den Compiler eine bestimmte Bedeutung haben. Dies sind die Schlüsselwörter in unserem Beispielprogramm:

using class static void int

Die meisten Schlüsselwörter sind für den Compiler *reserviert*, Sie können sie nicht als Bezeichner verwenden. Hier ist eine vollständige Liste aller C#-Schlüsselwörter:

---

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
while

Konflikte vermeiden

Wenn Sie wirklich einen Bezeichner nutzen wollen, der mit einem reservierten Schlüsselwort in Konflikt geraten würde, müssen Sie ihn mit dem Präfix @ auszeichnen:

```
class class  {...}     // illegal

class @class {...}     // legal
```

Das Zeichen @ gehört nicht zum Bezeichner selbst, daher ist @myVariable das Gleiche wie myVariable.

Kontextuelle Schlüsselwörter

Einige Schlüsselwörter sind *kontextbezogen*. Das heißt, sie können – auch ohne ein

vorangestelltes @-Zeichen – als Bezeichner eingesetzt werden, und zwar folgende:

---

add
ascending
async
await
by
descending
dynamic
equals
from
get
global
group
in
into
join
let
nameof
on
orderby
partial
remove
select
set
value
var
when
where
yield

Bei den kontextabhängigen Schlüsselwörtern kann es innerhalb des verwendeten Kontexts keine Mehrdeutigkeit geben.

Literale, Satzzeichen und Operatoren

Literale sind einfache Daten, die statisch im Programm verwendet werden. Die Literale in unserem Beispielprogramm sind 12 und 30. *Satzzeichen* helfen dabei, die Struktur des Programms abzugrenzen. Das hier sind die Satzzeichen in unserem Beispielprogramm: {, } und ;.

Die geschweiften Klammer gruppieren mehrere Anweisungen zu einem *Anweisungsblock*. Das Semikolon beendet eine Anweisung (die kein Block ist). Anweisungen können mehrere Zeilen übergreifen:

```
Console.WriteLine

    (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10);
```

Ein *Operator* verwandelt und kombiniert Ausdrücke. In C# werden die meisten Operatoren mithilfe von Symbolen angezeigt, beispielsweise dem Multiplikationsoperator *. Die Operatoren in unserem Programm sind folgende:

```
. () * =
```

Ein Punkt zeigt ein Member von etwas an (oder, in numerischen Literalen, den Dezimaltrenner). Die Klammern werden in unserem Beispiel genutzt, wenn eine Methode aufgerufen oder deklariert wird; leere Klammern werden verwendet, wenn eine Methode keine Argumente akzeptiert. Das Gleichheitszeichen führt eine *Zuweisung* aus (ein doppeltes Gleichheitszeichen, ==, führt einen Vergleich auf Gleichheit durch).

Kommentare

C# bietet zwei verschiedene Arten von Quellcodekommentaren: *einzeilige* und *mehrzeilige Kommentare*. Ein einzeiliger Kommentar beginnt mit zwei Schrägstrichen und geht bis zum Ende der aktuellen Zeile, zum Beispiel so:

```
int x = 3;  // Kommentar zur Zuweisung von 3 an x
```

Ein mehrzeiliger Kommentar beginnt mit /* und endet mit */, zum Beispiel so:

```
int x = 3;  /* Das ist ein Kommentar, der
```

zwei Zeilen umspannt. */

Kommentare können in XML-Dokumentations-Tags (siehe »XML-Dokumentation« auf Seite 213) eingebettet sein.

Typgrundlagen

Ein *Typ* definiert die Blaupause für einen Wert. In unserem Beispiel haben wir zwei Literale des Typs int mit den Werten 12 und 30 genutzt. Wir haben außerdem eine *Variable* des Typs int deklariert, deren Name x lautete.

Eine *Variable* zeigt einen Speicherort an, der mit der Zeit unterschiedliche Werte annehmen kann. Im Unterschied dazu repräsentiert eine *Konstante* immer den gleichen Wert (mehr dazu später).

Alle Werte sind in C# *Instanzen* eines spezifischen Typs. Die Bedeutung eines Werts und die Menge der möglichen Werte, die eine Variable aufnehmen kann, wird durch seinen bzw. ihren Typ bestimmt.

Vordefinierte Typen

Vordefinierte Typen (die auch als »eingebaute Typen« bezeichnet werden), sind solche, die besonders vom Compiler unterstützt werden. Der Typ int ist ein vordefinierter Typ, der die Menge der Ganzzahlen darstellen kann, die in einen 32-Bit-Speicher passen – von -2^{31} bis $2^{31}-1$. Wir können zum Beispiel arithmetische Funktionen mit Instanzen des Typs int durchführen:

```
int x = 12 * 30;
```

Ein weiterer vordefinierter Typ in C# ist string. Der Typ string repräsentiert eine Folge von Zeichen, zum Beispiel ».NET« oder »http://oreilly.com«. Wir können Strings bearbeiten, indem wir ihre Funktionen aufrufen:

```
string message = "Hallo Welt";

string upperMessage = message.ToUpper();

Console.WriteLine (upperMessage);        // HALLO WELT


int x = 2018;

message = message + x.ToString();
```

```
Console.WriteLine (message);           // Hallo Welt2018
```

Der vordefinierte Typ bool hat genau zwei mögliche Werte: true und false. bool wird häufig verwendet, um zusammen mit der if-Anweisung Befehle nur bedingt ausführen zu lassen:

```
bool simpleVar = false;

if (simpleVar)

    Console.WriteLine ("Das wird nicht ausgegeben");


int x = 5000;

bool lessThanAMile = x < 5280;

if (lessThanAMile)

    Console.WriteLine ("Das wird ausgegeben");
```

Der Namensraum System im .NET Framework enthält viele wichtige Typen, die C# nicht vordefiniert (zum Beispiel DateTime).

Benutzerdefinierte Typen

So, wie wir komplexe Funktionen aus einfachen Funktionen aufbauen können, können wir auch komplexe Typen aus primitiven Typen aufbauen. In diesem Beispiel werden wir einen eigenen Typ namens UnitConverter definieren – eine Klasse, die als Vorlage für die Umwandlung von Einheiten dient:

```
using System;


public class UnitConverter
```

```
{
    int ratio;                          // Feld

    public UnitConverter (int unitRatio)   // Konstruktor
    {
        ratio = unitRatio;
    }

    public int Convert (int unit)          // Methode
    {
        return unit * ratio;
    }
}

class Test
{
    static void Main( )
    {
        UnitConverter feetToInches = new UnitConverter(12);
        UnitConverter milesToFeet = new UnitConverter(5280);

        Console.Write (feetToInches.Convert(30));   // 360
        Console.Write (feetToInches.Convert(100));  // 1200
```

```
        Console.Write (feetToInches.Convert
                                (milesToFeet.Convert(1)));  // 63360
    }
}
```

Member eines Typs

Ein Typ enthält *Daten-Member* und *Funktions-Member*. Das Daten-Member von UnitConverter ist das *Feld* mit dem Namen ratio. Die Funktions-Member von UnitConverter sind die Methode Convert und der *Konstruktor* von UnitConverter.

Symmetrie vordefinierter und benutzerdefinierter Typen

Das Schöne an C# ist, dass vordefinierte und selbst definierte Typen nur wenige Unterschiede aufweisen. Der primitive Typ int dient als Vorlage für Ganzzahlen (Integer). Er speichert Daten – 32 Bit – und stellt Funktions-Member bereit, die diese Daten verwenden, zum Beispiel ToString. Genauso dient unser selbst definierter Typ UnitConverter als Vorlage für die Einheitenumrechnung. Er enthält Daten – das Verhältnis zwischen den Einheiten – und stellt Funktions-Member bereit, die diese Daten nutzen.

Konstruktoren und Instanziierung

Daten werden erstellt, indem ein Typ *instanziiert* wird. Vordefinierte Typen können einfach mit einem Literal wie 12 oder "Hallo Welt" definiert werden.

Der new-Operator erstellt Instanzen von benutzerdefinierten Typen. Wir haben unsere Main-Methode damit begonnen, dass wir zwei Instanzen des Typs UnitConverter erstellten. Unmittelbar nachdem der new-Operator ein Objekt instanziiert hat, wird der *Konstruktor* des Objekts aufgerufen, um die Initialisierung durchzuführen. Ein Konstruktur wird wie eine Methode definiert, aber der Methoden-name und der Rückgabetyp werden auf den Namen des einschließenden Typen reduziert:

```
public UnitConverter (int unitRatio) // Konstruktor
{
    ratio = unitRatio;
```

```
}
```

Instanz-Member versus statische Member

Die Daten-Member und die Funktions-Member, die mit der *Instanz* des Typs arbeiten, werden als Instanz-Member bezeichnet. Die Methode Convert von UnitConverter und die Methode ToString von int sind Beispiele für solche Instanz-Member. Standardmäßig sind Member Instanz-Member.

Daten-Member und Funktions-Member, die nicht mit der Instanz des Typs arbeiten, sondern mit dem Typ selbst, müssen als static gekennzeichnet werden. Die Methoden Test.Main und Console.WriteLine sind statische Methoden. Die Klasse Console ist sogar eine *statische Klasse*, bei der *alle* Member statisch sind. Man erzeugt nie tatsächlich Instanzen von Console – eine einzige Konsole wird in der gesamten Anwendung verwendet.

Der Unterschied zwischen Instanz- und statischen Membern ist dieser: Im folgenden Beispielcode gehört das Instanz-Feld Name zu einer Instanz eines bestimmten Panda, während Population zur Menge aller Panda-Instanzen gehört:

```
public class Panda

{

    public string Name;            // Instanz-Feld

    public static int Population;   // statisches Feld


    public Panda (string n)        // Konstruktor

    {

        Name = n;                  // Instanz-Feld zuweisen

        Population = Population+1;    // statisches Feld erhöhen

    }

}
```

Der nächste Code erzeugt zwei Instanzen von Panda und gibt ihre Namen und dann die Gesamtpopulation aus:

```
Panda p1 = new Panda ("Pan Dee");

Panda p2 = new Panda ("Pan Dah");

Console.WriteLine (p1.Name);          // Pan Dee

Console.WriteLine (p2.Name);          // Pan Dah


Console.WriteLine (Panda.Population);  // 2
```

Das Schlüsselwort public

Das Schlüsselwort public macht Member für andere Klassen zugänglich. Wenn in diesem Beispiel das Feld Name in Panda nicht als öffentlich markiert gewesen wäre, würde es sich um ein privates Feld handeln, und die Klasse Test hätte es nicht ansprechen können. Das »Öffentlichmachen« eines Members mit public lässt einen Typ sagen: »Das hier will ich andere Typen sehen lassen – alles andere sind meine privaten Implementierungsdetails.« In objektorientierten Begriffen sagen wir, dass die öffentlichen Member die privaten Member der Klasse *kapseln*.

Umwandlungen

C# kann Instanzen kompatibler Typen umwandeln. Eine Umwandlung erstellt immer einen neuen Wert für einen bestehenden Wert. Umwandlungen können entweder *implizit* oder *explizit* sein. Implizite Umwandlungen erfolgen automatisch, während explizite Umwandlungen einen *Cast* erfordern. Im folgenden Beispiel konvertieren wir *implizit* einen int in einen long (der doppelt so viel Kapazität an Bits wie ein int bietet) und casten *explizit* einen int auf einen short (der nur die halbe Bit-Kapazität eines int bietet):

```
int x = 12345;       // int ist ein 32-Bit-Integer

long y = x;          // implizite Umwandlung in einen 64-Bit-int

short z = (short)x;  // explizite Umwandlung in einen 16-Bit-int
```

In der Regel sind implizite Umwandlungen dann zulässig, wenn der Compiler garantieren kann, dass sie immer gelingen werden, ohne dass dabei Informationen verloren gehen. Andernfalls müssen Sie einen expliziten Cast nutzen, um die Umwandlung zwischen kompatiblen Typen durchzuführen.

Werttypen vs. Referenztypen

C#-Typen können in *Werttypen* und *Referenztypen* eingeteilt werden.

Werttypen enthalten die meisten eingebauten Typen (genauer gesagt, alle numerischen Typen sowie die Typen char und bool), aber auch selbst definierte struct- und enum-Typen. *Referenztypen* enthalten alle Klassen-, Array-, Delegate- und Interface-Typen.

Der prinzipielle Unterschied zwischen Werttypen und Referenztypen ist ihre Behandlung im Arbeitsspeicher.

Werttypen

Der Inhalt einer *Werttyp*-Variablen oder -Konstanten ist einfach ein Wert. So besteht zum Beispiel der Inhalt des eingebauten Werttyps int aus 32 Bit mit Daten.

Sie können einen selbst definierten Werttyp mithilfe des Schlüsselworts struct definieren (siehe Abbildung 1):

```
public struct Point { public int X, Y; }
```

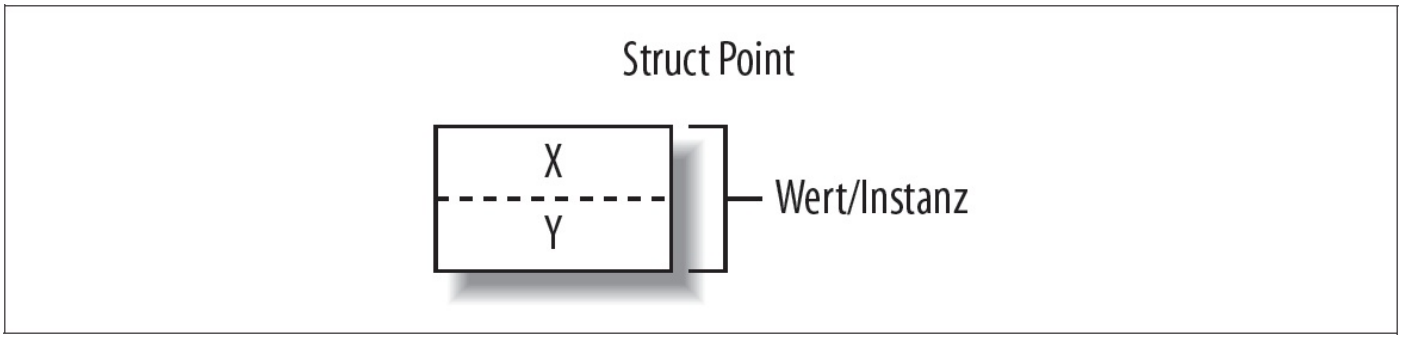

Abbildung 1: Eine Werttyp-Instanz im Speicher

Das Zuweisen einer Werttyp-Instanz *kopiert* immer die Instanz:

```
Point p1 = new Point();

p1.X = 7;
```

```
Point p2 = p1;          // Zuweisung führt zum Kopieren

Console.WriteLine (p1.X);  // 7

Console.WriteLine (p2.X);  // 7

p1.X = 9;               // ändert p1.X

Console.WriteLine (p1.X);  // 9

Console.WriteLine (p2.X);  // 7
```

Abbildung 2 zeigt, dass p1 und p2 unabhängig voneinander gespeichert werden.

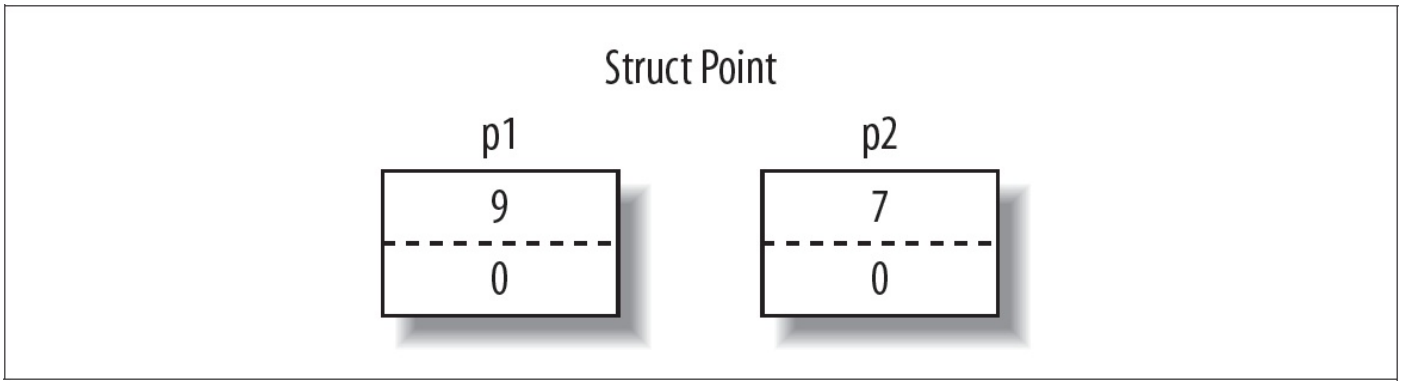

Abbildung 2: Eine Zuweisung kopiert eine Werttyp-Instanz.

Referenztypen

Ein *Referenztyp* ist komplexer als ein Werttyp. Er besteht aus zwei Teilen: einem *Objekt* und der *Referenz* auf dieses Objekt. Der Inhalt einer Referenztyp-Variablen oder -Konstanten ist eine Referenz auf ein Objekt, das den Wert enthält. Hier ist der Typ Point aus unserem vorigen Beispiel als Klasse umgeschrieben worden (siehe Abbildung 3):

```
public class Point { public int X, Y; }
```

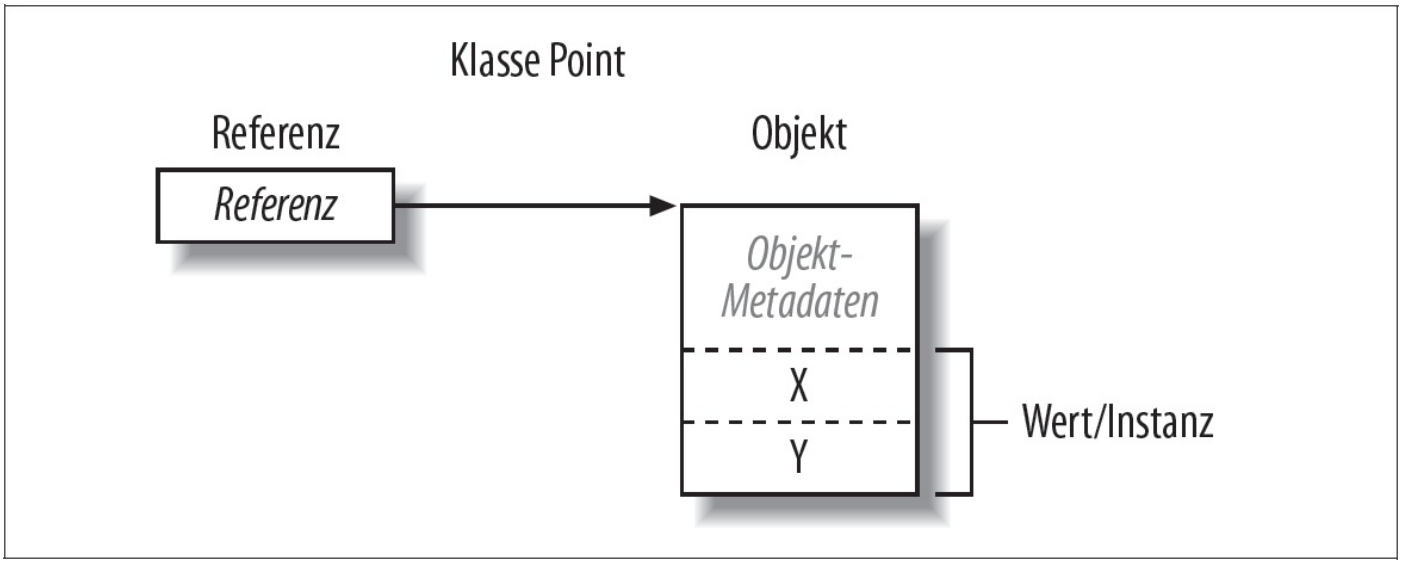

Abbildung 3: Ein Referenztyp im Speicher

Durch das Zuweisen einer Referenztyp-Variablen wird die Referenz kopiert, nicht die Objektinstanz. Damit ist es möglich, mit mehreren Variablen auf dasselbe Objekt zu verweisen – etwas, das mit Werttypen normalerweise nicht geht. Wenn wir das vorige Beispiel wiederholen, diesmal aber mit Point als Klasse, beeinflusst eine Operation auf p1 auch p2:

```
Point p1 = new Point();

p1.X = 7;


Point p2 = p1;            // kopiert Referenz von p1


Console.WriteLine (p1.X);  // 7

Console.WriteLine (p2.X);  // 7


p1.X = 9;                 // ändert p1.X

Console.WriteLine (p1.X);  // 9

Console.WriteLine (p2.X);  // 9
```

Abbildung 4 zeigt, dass p1 und p2 zwei Referenzen sind, die auf dasselbe Objekt verweisen.

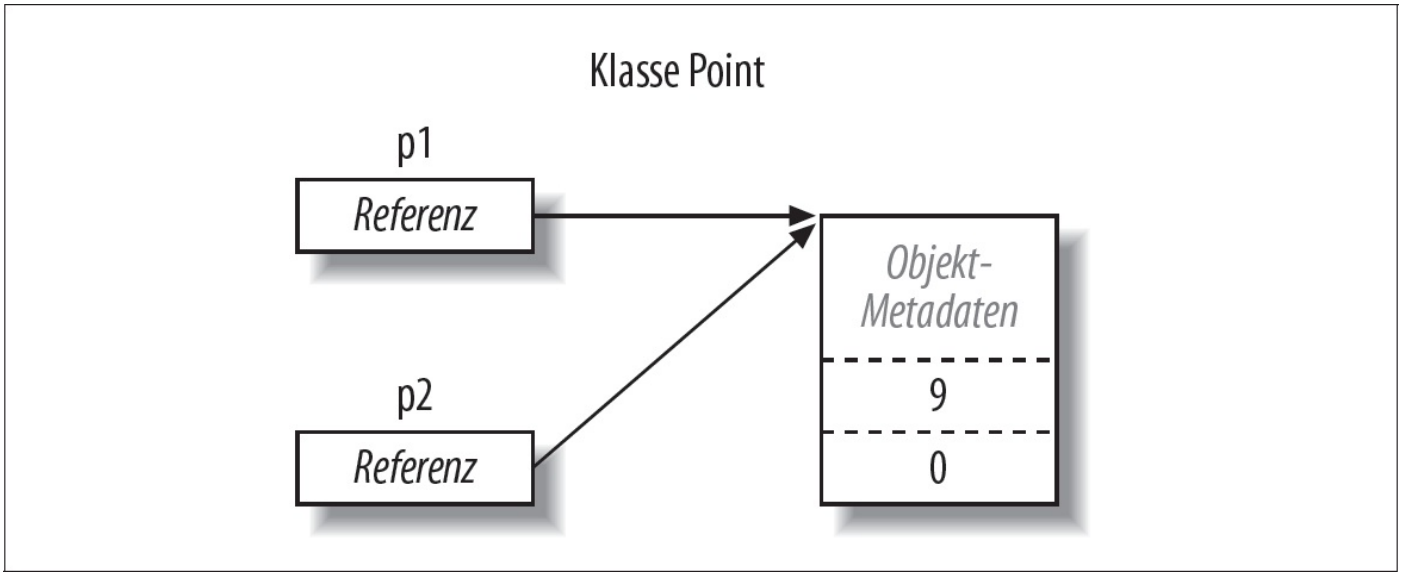

Abbildung 4: Eine Zuweisung kopiert eine Referenz.

Null

Einer Referenz kann das Literal null zugewiesen werden, wodurch ausgesagt wird, dass die Referenz auf kein Objekt zeigt – vorausgesetzt, Point ist eine Klasse:

```
Point p = null;

Console.WriteLine (p == null);   // True
```

Der Versuch, auf ein Member einer Null-Referenz zuzugreifen, führt zu einem Laufzeitfehler:

```
Console.WriteLine (p.X); // NullReferenceException
```

Im Gegensatz dazu kann einem Werttyp auf normalem Weg kein Null-Wert zugewiesen werden:

```
struct Point {...}

...

Point p = null;  // Compilerfehler

int x = null;    // Compilerfehler
```

C# bietet *nullbare Typen* an, mit denen Werttypen auch Null-Werte repräsentieren können (siehe den Abschnitt »Nullbare Typen« auf Seite 144).

Die Einteilung der vordefinierten Typen

Die vordefinierten Typen in C# sind folgende:

Werttypen

- Numerisch
  - – Ganzzahl mit Vorzeichen (sbyte, short, int, long)
  - – Ganzzahl ohne Vorzeichen (byte, ushort, uint, ulong)
  - – Reelle Zahl (float, double, decimal)
- Logisch (bool)
- Zeichen (char)

Referenztypen

- String (string)
- Objekt (object)

Die vordefinierten Typen in C# sind Aliase für .NET Framework-Typen aus dem Namensraum System. Zwischen den beiden folgenden Anweisungen gibt es nur syntaktische Unterschiede:

```
int i = 5;

System.Int32 i = 5;
```

Die vordefinierten Werttypen (mit Ausnahme von decimal) werden in der *Common Language Runtime* (CLR) als *elementare Typen* bezeichnet. Sie heißen so, weil sie im kompilierten Code direkt über Anweisungen unterstützt werden, die üblicherweise auf eine unmittelbare Unterstützung durch den zugrunde liegenden Prozessor zurückgehen.

Numerische Typen

C# bietet die folgenden vordefinierten numerischen Typen:

C#-Typ	Systemtyp	Suffix	Breite	Bereich
Ganzzahlig mit Vorzeichen				
sbyte	SByte		8 Bit	-2^7 bis $2^7 - 1$
short	Int16		16 Bit	-2^{15} bis $2^{15} - 1$
int	Int32		32 Bit	-2^{31} bis $2^{31} - 1$
long	Int64	L	64 Bit	-2^{63} bis $2^{63} - 1$
Ganzzahlig ohne Vorzeichen				
byte	Byte		8 Bit	0 bis $2^8 - 1$
ushort	UInt16		16 Bit	0 bis $2^{16} - 1$
uint	UInt32	U	32 Bit	0 bis $2^{32} - 1$
ulong	UInt64	UL	64 Bit	0 bis $2^{64} - 1$
Reell				
float	Single	F	32 Bit	$\pm$ (~10^{-45} bis 10^{38})
double	Double	D	64 Bit	$\pm$ (~10^{-324} bis 10^{308})
decimal	Decimal	M	128 Bit	$\pm$ (~10^{-28} bis 10^{28})

Von den *ganzzahligen* Typen sind int und long Bürger erster Klasse und werden von C# und der Runtime bevorzugt. Die anderen ganzzahligen Typen werden üblicherweise im Dienste der Interoperabilität eingesetzt oder wenn eine effiziente Speicherplatznutzung wichtig ist.

Von den *reellen* Zahltypen werden float und double auch als *Gleitkommatypen* bezeichnet und üblicherweise für wissenschaftliche Berechnungen sowie im Grafikumfeld genutzt. Der Typ decimal wird in der Regel für finanzmathematische Berechnungen verwendet, bei denen eine exakte Basis-10-Arithmetik und hohe Genauigkeit erforderlich sind. (Technisch betrachtet, ist decimal ebenfalls ein Gleitkommatyp, wird normalerweise aber nicht als solcher bezeichnet.)

Numerische Literale

Ganzzahlliterale können mit der Dezimal- oder der Hexadezimalnotation dargestellt werden; die Hexadezimalnotation wird mit dem Präfix 0x angezeigt (z. B. entspricht 0x7f dem Dezimalwert 127). Seit C# 7.0 können Sie auch das Präfix 0b für Binärliterale einsetzen. *Reelle Literale* können die Dezimal- oder die Exponentialnotation nutzen, z. B. 1E06.

Ab C# 7.0 können numerische Literale für eine bessere Lesbarkeit durch Unterstriche ergänzt werden (zum Beispiel 1_000_000).

Typableitung bei numerischen Literalen

Standardmäßig geht der Compiler davon aus, dass ein numerisches Literal entweder einen double-Wert oder einen ganzzahligen Wert darstellt:

- Wenn das Literal einen Dezimaltrenner oder das Exponential-symbol (E) enthält, ist der Typ double.
- Andernfalls ist der Typ des Literals der erste Typ aus der folgenden Liste, in den der Wert des Literals passt: int, uint, long und ulong.

Hier sehen Sie Beispiele dafür:

```
Console.Write (       1.0.GetType());  // Double (double)

Console.Write (      1E06.GetType());  // Double (double)

Console.Write (        1.GetType());  // Int32  (int)

Console.Write ( 0xF0000000.GetType());  // UInt32 (uint)

Console.Write (0x100000000.GetType());  // Int64  (long)
```

Numerische Suffixe

Die *numerischen Suffixe*, die in der vorangegangenen Tabelle aufgeführt sind, definieren den Typ eines Literals:

```
decimal d = 3.5M;  // M = decimal (ignoriert Groß-/Kleinschrei-

                   // bung)
```

Die Suffixe U und L werden nur selten benötigt, weil die Typen uint, long und ulong fast immer *erschlossen* werden können oder int *implizit* in sie umgewandelt werden kann.

```
long i = 5;       // implizite Umwandlung von int in long
```

Das Suffix D ist technisch gesehen redundant, da bei allen Literalen, die einen Dezimaltrenner enthalten, geschlossen wird, dass es double-Werte sind (und man einem numerischen Literal immer einen Dezimaltrenner anhängen kann). Die Suffixe F und M sind am nützlichsten und außerdem unumgänglich, wenn man float- oder decimal-Literale angeben will. Ohne Suffixe ließen sich die folgenden Anweisungen nicht kompilieren, da geschlossen würde, dass 4.5 den Typ double hat, der sich nicht implizit in float oder decimal umwandeln lässt.

```
float f = 4.5F;       // kompiliert ohne Suffix nicht

decimal d = 4.5M;     // kompiliert ohne Suffix nicht
```

Numerische Umwandlung

Ganzzahlige Umwandlungen

Ganzzahlige Umwandlungen sind *implizit*, wenn der Zieltyp jeden möglichen Wert des Ausgangstyps darstellen kann. Andernfalls ist eine *explizite* Umwandlung erforderlich, zum Beispiel so:

```
int x = 12345;       // int ist ein 32-Bit-Wert

long y = x;          // implizite Umwandlung in einen 64-Bit-int

short z = (short)x;  // explizite Umwandlung in einen 16-Bit-int
```

Reelle Umwandlungen

Ein float kann implizit in einen double umgewandelt werden, weil ein double jeden möglichen float-Wert darstellen kann. Die umgekehrte Umwandlung muss explizit sein.

Die Umwandlung zwischen decimal und den anderen reellen Typen muss explizit sein.

Reelle Typen in ganzzahlige Typen umwandeln

Die Umwandlung eines ganzzahligen Typs in einen reellen Typ ist immer implizit, während die umgekehrte Umwandlung immer explizit angegeben werden muss. Bei der Umwandlung eines Gleitkommatyps in einen ganzzahligen Typ wird immer der möglicherweise vorhandene Nachkommateil der Zahl abgeschnitten. Rundende Umwandlungen können Sie mit der statischen Klasse System.Convert durchführen.

Ein Nachteil der impliziten Umwandlung ist, dass bei der Umwandlung großer ganzzahlige Werte in Gleitkommawerte zwar die *Größenordnung* erhalten bleibt, gelegentlich aber *Genauigkeit* verloren geht.

```
int i1 = 100000001;

float f = i1;      // Größe bleibt erhalten, Genauigkeit ver-

                                  // schwindet

int i2 = (int)f;   // 100000000
```

Arithmetische Operatoren

Die arithmetischen Operatoren (+, -, *, /, %) sind für alle numerischen Typen außer den 8- und 16-Bit-Ganzzahltypen definiert. Der %-Operator wird zum Rest nach der Division ausgewertet.

Inkrement- und Dekrementoperatoren

Die Inkrement- und Dekrementoperatoren (++, –) erhöhen bzw. verringern numerische Typen um eins. Der Operator kann entweder vor oder hinter der Variablen stehen – je nachdem, ob Sie die Variable *vor* oder *nach* dem Auswerten des Ausdrucks verändern wollen. Hier sehen Sie ein Beispiel:

```
int x = 0;

Console.WriteLine (x++);   // Ausgabe 0; x ist jetzt 1
```

```
Console.WriteLine (++x);   // Ausgabe 2; x ist jetzt 2

Console.WriteLine (--x);   // Ausgabe 1; x ist jetzt 1
```

Besondere integrale Operationen

Division

Divisionsoperationen mit ganzzahligen Operanden schneiden immer den Rest ab (runden auf null). Eine Division durch eine Variable, deren Wert null ist, führt zu einem Laufzeitfehler (einer DivideByZeroException). Eine Division durch das *Literal* 0 oder eine *Konstante* mit dem Wert 0 führt zu einem Kompiliationsfehler.

Überlauf

Arithmetische Operationen auf Integer-Typen können zur Laufzeit zu einem Überlauf führen. Standardmäßig geschieht das still und leise – es wird keine Exception ausgelöst, und das Ergebnis weist ein Umschlagverhalten auf, als ob die Berechnung mit einem größeren Ganzzahltypen durchgeführt und die zusätzlichen signifikanten Bits verworfen worden wären. Verringert man zum Beispiel den minimalen möglichen int-Wert um eins, erhält man den maximal möglichen int-Wert:

```
int a = int.MinValue; a--;

Console.WriteLine (a == int.MaxValue); // True
```

Die Operatoren checked und unchecked

Der Operator checked teilt der Laufzeitumgebung mit, eine OverflowException zu erzeugen, statt still und heimlich fehlzuschlagen, wenn ein ganzzahliger Ausdruck oder eine Anweisung die arithmetischen Grenzen dieses Typs überschreitet. Der Operator checked ist für Ausdrücke mit ++, – (unär), +, -, *, / und expliziten Konvertierungsoperatoren zwischen ganzzahligen Typen nutzbar.

Sie können checked um einen Ausdruck oder einen Anweisungsblock herum verwenden, zum Beispiel so:

```
int a = 1000000, b = 1000000;

int c = checked (a * b);   // Prüft nur den Ausdruck
```

```
checked                 // Prüft alle Ausdrücke
{                       // im Anweisungsblock
    c = a * b;
    ...
}
```

Sie können die Prüfung auf arithmetische Überläufe zum Standard für alle Ausdrücke in einem Programm machen, indem Sie sie mit dem Schalter /checked+ an der Befehlszeile kompilieren (in Visual Studio gehen Sie zu den Advanced Build Settings). Wenn Sie die Überlaufprüfung nur für bestimmte Ausdrücke oder Anweisungen abschalten wollen, können Sie das mit dem Operator unchecked erreichen.

Bitoperatoren

C# unterstützt die folgenden Bitoperatoren :

Operator	Bedeutung	Beispiel	Ergebnis
~	Komplement	~0xfU	0xfffffff0U
&	Und	0xf0 & 0x33	0x30
|	Oder	0xf0 | 0x33	0xf3
^	Exklusives Oder	0xff00 ^ 0x0ff0	0xf0f0
<<	Verschiebung nach links	0x20 << 2	0x80
>>	Verschiebung nach rechts	0x20 >> 1	0x10

8- und 16-Bit-Ganzzahlwerte

Die 8- und 16-Bit-Integer-Typen sind byte, sbyte, short und ushort. Sie haben keine eigenen arithmetischen Operatoren, daher konvertiert C# sie bei Bedarf implizit in größere Typen. Das kann zu einem Kompilierungsfehler führen, wenn man versucht, das Ergebnis wieder einem kleinen Integer-Typ zuzuweisen:

```
short x = 1, y = 1;

short z = x + y;          // Compilerfehler
```

In diesem Fall werden x und y implizit nach int konvertiert, damit die Addition durchgeführt werden kann. Das bedeutet aber, dass das Ergebnis ebenfalls ein int ist, der nicht implizit in einen short konvertiert werden kann (da es dabei zu Datenverlust kommen könnte). Damit sich diese Anweisung kompilieren lässt, müssen wir einen expliziten Cast hinzufügen:

```
short z = (short) (x + y);   // okay
```

Spezielle Float- und Double-Werte

Im Unterschied zu Integer-Typen haben Gleitkommatypen Werte, die bestimmte Operationen besonders behandeln. Diese speziellen Werte sind NaN (*Not a Number*, keine Zahl), +∞, –∞ und –0. Die Float- und Double-Klassen haben Konstanten für NaN, +∞ und –∞ und auch für andere Werte (MaxValue, MinValue und Epsilon). Hier sehen Sie ein Beispiel:

```
Console.Write (double.NegativeInfinity);   // -Infinity
```

Die Division einer Zahl ungleich null durch null ergibt einen der Unendlich-Werte:

```
Console.WriteLine ( 1.0 /  0.0);   //  Infinity

Console.WriteLine (–1.0 /  0.0);   // -Infinity

Console.WriteLine ( 1.0 / –0.0);   // -Infinity

Console.WriteLine (–1.0 / –0.0);   //  Infinity
```

Null geteilt durch null oder unendlich minus unendlich ergibt NaN:

```
Console.Write ( 0.0 / 0.0);              // NaN
```

```
Console.Write ((1.0 / 0.0) – (1.0 / 0.0));  // NaN
```

Wenn == genutzt wird, ist ein NaN-Wert niemals gleich einem anderen Wert, auch nicht einem anderen NaN-Wert. Um zu prüfen, ob ein Wert NaN ist, müssen Sie die Methoden float.IsNaN oder double.IsNaN einsetzen:

```
Console.WriteLine (0.0 / 0.0 == double.NaN);   // False

Console.WriteLine (double.IsNaN (0.0 / 0.0));  // True
```

Verwenden Sie allerdings object.Equals, sind zwei NaN-Werte gleich:

```
bool isTrue = object.Equals (0.0/0.0, double.NaN);
```

double vs. decimal

Für wissenschaftliche Berechnungen (wie die Berechnung von Raumkoordinaten) eignet sich double. decimal ist für finanzmathematische Berechnungen geeignet und für Werte, die »menschengeschaffen« sind und nicht das Ergebnis echter Messungen. Hier sehen Sie eine Zusammenfassung der Unterschiede:

Eigenschaft	double	decimal
Interne Darstellung	Basis 2	Basis 10
Genauigkeit	15 bis 16 signifikante Ziffern	28 bis 29 signifikante Ziffern
Wertbereich	±(~10^{-324} bis ~10^{308})	±(~10^{-28} bis ~10^{28})
Besondere Werte	+0, –0, +∞, –∞ und NaN	keine
Geschwindigkeit	native CPU-Verarbeitung	keine native Verarbeitung auf der CPU (ungefähr 10-mal langsamer als double)

Rundungsfehler bei reellen Zahlen

float und double werden intern durch Zahlen zur Basis 2 repräsentiert. Aus diesem Grund werden die meisten Literale mit einem Nachkommateil (die zur Basis 10 sind) nicht genau abgebildet:

```
float tenth = 0.1f;                 // nicht ganz 0.1

float one = 1f;

Console.WriteLine (one - tenth * 10f);  // -1.490116E-08
```

Das ist der Grund dafür, dass float und double für finanzmathematische Berechnungen nicht geeignet sind. Im Gegensatz dazu nutzt decimal die Basis 10 und kann Nachkommaanteile wie 0.1 (deren Basis-10-Darstellung nicht unendlich periodisch ist) deswegen genau darstellen.

Der Typ bool und die booleschen Operatoren

Der C#-Typ bool (der ein Alias für den Typ System.Boolean ist) ist ein logischer Wert, dem eines der Literale true und false zugewiesen werden kann.

Auch wenn ein boolescher Wert beim Speichern nur ein Bit benötigt, nutzt die Laufzeitumgebung ein Byte im Speicher, da dies die kleinste Einheit ist, mit der die Laufzeitumgebung und der Prozessor sinnvoll umgehen können. Um bei Arrays den Speicherplatz nicht ineffizient zu nutzen, stellt das Framework im Namensraum System.Collections eine Klasse namens BitArray bereit, die nur ein Bit pro booleschen Wert verwendet.

Gleichheits- und Vergleichsoperatoren

== und != prüfen auf Gleichheit bzw. Ungleichheit zwischen beliebigen Typen, liefern aber immer einen bool-Wert zurück. Werttypen haben normalerweise eine sehr einfache Vorstellung von Gleichheit:

```
int x = 1, y = 2, z = 1;

Console.WriteLine (x == y);     // False

Console.WriteLine (x == z);     // True
```

Bei Referenztypen basiert die Gleichheit standardmäßig auf der *Referenz* und nicht auf dem eigentlichen *Wert* des zugrunde liegenden Objekts. Zwei Instanzen eines Objekts mit identischen Daten werden nur dann als gleich betrachtet, wenn der ==-Operator für diesen Typ explizit entsprechend überladen wurde (mehr Informationen finden Sie in den Abschnitten »Der Typ object« auf Seite 88 und »Überladen von Operatoren« auf Seite 188).

Die Gleichheits- und Vergleichsoperatoren ==, !=, <, >, >= und <= funktionieren bei allen numerischen Typen, sollten bei reellen Zahlen aber mit Vorsicht eingesetzt werden (siehe den Abschnitt »Rundungsfehler bei reellen Zahlen« auf Seite 25). Die Vergleichsoperatoren funktionieren auch bei Membern von enum-Typen, indem sie die zugrunde liegenden ganzzahligen Werte vergleichen.

Bedingungsoperatoren

Die Operatoren && und || testen auf die Bedingungen *und* und *oder*. Sie werden häufig zusammen mit dem Operator ! verwendet, der für *nicht* steht. Im folgenden Beispiel liefert die Methode UseUmbrella den Wert true zurück, wenn es regnet oder die Sonne scheint (um uns vor dem Regen oder der Sonne zu schützen), solange es nicht auch noch windig ist (da Regenschirme bei Wind nicht viel helfen):

```
static bool UseUmbrella (bool rainy, bool sunny,
                                        bool windy)
{
    return !windy && (rainy || sunny);
}
```

Die Operatoren && und || *kürzen* die Auswertung wenn möglich *ab*. Im vorigen Beispiel wird der Ausdruck (rainy || sunny) gar nicht ausgewertet, wenn es nicht windig ist. Das Abkürzen spielt eine wesentliche Rolle dabei, dass Ausdrücke wie der folgende ausgewertet werden können, ohne dass eine NullReferenceException ausgelöst wird:

```
if (sb != null && sb.Length > 0) ...
```

Die Operatoren & und | können ähnlich verwendet werden:

```
return !windy & (rainy | sunny);
```

Der Unterschied liegt darin, dass sie *keine abgekürzte Auswertung* vornehmen. Aus diesem Grund werden sie nur selten als bedingte Operatoren eingesetzt.

Der ternäre Bedingungsoperator (der einfach als *Bedingungsoperator* bezeichnet wird) hat die Form q ? a : b. Dabei wird a ausgewertet, falls die Bedingung q wahr ist, ansonsten b. Hier sehen Sie ein Beispiel:

```
static int Max (int a, int b)
{
```

```
        return (a > b) ? a : b;

    }
```

Der Bedingungsoperator ist bei LINQ-Abfragen(*Language Integrated Query*) besonders nützlich.

Strings und Zeichen

Der C#-Typ char (ein Alias des Typs System.Char) repräsentiert ein Unicode-Zeichen und nimmt zwei Bytes ein. Ein char-Literal wird innerhalb von einfachen Anführungszeichen angegeben:

```
char c = 'A';       // einzelnes Zeichen
```

Escape-Sequenzen geben Zeichen an, die nicht direkt ausgedrückt oder interpretiert werden können. Eine Escape-Sequenz ist ein Backslash, gefolgt von einem Zeichen mit einer bestimmten Bedeutung:

```
char newLine = '\n';

char backSlash = '\\';
```

Die Escape-Sequenz-Zeichen sind folgende:

Char	Bedeutung	Wert
\'	einfache Anführungszeichen	0x0027
\"	doppelte Anführungszeichen	0x0022
\\	Backslash	0x005C
\0	Null	0x0000
\a	Alert	0x0007
\b	Backspace	0x0008
\f	Seitenvorschub	0x000C
\n	Zeilenumbruch	0x000A
\r	Carriage-Return	0x000D
\t	horizontaler Tabulator	0x0009
\v	vertikaler Tabulator	0x000B

Die Escape-Sequenz \u (bzw. \x) ermöglicht Ihnen, ein beliebiges Unicode-Zeichen über seinen vierstelligen Hexadezimalcode anzugeben:

```
char copyrightSymbol = '\u00A9';

char omegaSymbol     = '\u03A9';

char newLine         = '\u000A';
```

Eine implizite Konvertierung von einem char in einen numerischen Typ funktioniert für die numerischen Typen, die ein vorzeichenloses short aufnehmen können. Bei anderen numerischen Typen ist eine explizite Konvertierung notwendig.

String-Typ

Der string-Typ von C# (ein Alias für den Typ System.String) repräsentiert eine nicht veränderbare Kette von Unicode-Zeichen. Ein String-Literal wird innerhalb von doppelten Anführungszeichen angegeben:

```
string a = "Heat";
```

string ist ein Referenztyp, kein Werttyp. Seine Vergleichsoperatoren folgen jedoch der Werttyp-Semantik.

```
string a = "test", b = "test";
Console.Write (a == b); // True
```

Die für char-Literale möglichen Escape-Sequenzen lassen sich auch bei Strings nutzen:

```
string a = "Hier ist ein Tabulator:\t";
```

Wenn Sie allerdings einen Backslash als Zeichen nutzen wollen, müssen Sie ihn doppelt schreiben:

```
string a1 = "\\\\server\\fileshare\\helloworld.cs";
```

Um dieses Problem zu vermeiden, bietet C# *Verbatim-Strings* an. Ein Verbatim-string-Literal wird durch ein @ eingeleitet und unterstützt keine Escape-Sequenzen. Der folgende Verbatim-String ist mit dem String oben identisch:

```
string a2 = @"\\server\fileshare\helloworld.cs";
```

Ein Verbatim-String-Literal kann auch über mehrere Zeilen gehen. Sie können das doppelte Anführungszeichen in einem Verbatim-Literal nutzen, indem Sie es doppelt schreiben.

String-Verkettung

Der +-Operator verkettet zwei Strings:

```
string s = "a" + "b";
```

Einer der Operanden kann ein Nicht-String-Wert sein. Dann wird ToString auf diesem Wert aufgerufen, zum Beispiel so:

```
string s = "a" + 5; // a5
```

Es kann ineffizient sein, einen String unter wiederholter Verwendung des +-Operators aufzubauen. Eine bessere Lösung ist der Einsatz des Typs System.Text.StringBuilder, der einen veränderlichen (bearbeitbaren) String repräsentiert und Methoden bietet, über die Substrings effizient *angehängt*, *eingefügt*, *entfernt* oder *ersetzt* werden können.

String-Interpolation

Ein String, dem ein $ (Dollarzeichen) vorangestellt ist, wird als *interpolierter String* bezeichnet. Interpolierte Strings können in geschweiften Klammern Ausdrücke enthalten:

```
int x = 4;

Console.Write ($"Ein Quadrat hat {x} Seiten");

// Gibt aus: Ein Quadrat hat 4 Seiten
```

Jeder gültige C#-Ausdruck beliebigen Typs kann in den geschweiften Klammern stehen, und C# wird ihn durch den Aufruf seiner ToString-Methode oder eines Äquivalents umwandeln. Sie können die Formatierung anpassen, indem Sie auf den Ausdruck einen Doppelpunkt und einen *Formatstring* folgen lassen:

```
string s = $"255 in Hex ist {byte.MaxValue:X2}"

// Wird zu "255 in Hex ist FF"
```

Interpolierte Strings dürfen nicht über eine Zeile hinausgehen, sofern Sie nicht auch noch den Verbatim-String-Operator verwenden. Dabei muss das $ vor dem @ stehen:

```
int x = 2;

string s = $@"Dies geht über {

x} Zeilen.";
```

Um in einem interpolierten String eine geschweifte öffnende oder schließende Klammer auszugeben, geben Sie sie doppelt an.

String-Vergleiche

string unterstützt nicht die Vergleichsoperatoren < und >. Sie müssen stattdessen die Methode CompareTo von string nutzen, die eine positive Zahl, eine negative Zahl oder null zurückgibt – abhängig davon, ob der erste Wert nach oder vor dem zweiten Wert kommt bzw. ob er gleich ist:

```
Console.Write ("Boston".CompareTo ("Austin"));    // 1

Console.Write ("Boston".CompareTo ("Boston"));    // 0

Console.Write ("Boston".CompareTo ("Chicago"));   // -1
```

In Strings suchen

Der Indexer von string liefert ein Zeichen an einer bestimmten Position zurück:

```
Console.Write ("Wort"[2]);  // r
```

Die Methoden IndexOf und LastIndexOf suchen nach einem Zeichen innerhalb eines String. Die Methoden Contains, StartsWith und EndsWith suchen nach einem Substring in einem String.

Strings verändern

Da string nicht veränderbar ist, liefern alle Methoden, die einen String »bearbeiten«, einen neuen zurück und lassen das Original unangetastet:

- Substring liefert einen Teil eines Strings zurück.
- Insert und Remove fügen Zeichen an einer bestimmten Position ein bzw. entfernen sie.
- PadLeft und PadRight fügen Leerzeichen ein.
- TrimStart, TrimEnd und Trim entfernen Leerraum.

Die Klasse string definiert zudem die Methoden ToUpper und ToLower, um die Groß- und Kleinschreibung zu ändern, eine Methode Split, um einen String in Substrings aufzuteilen (basierend auf einem angegebenen Trennzeichen), und eine statische Methode Join, um Substrings wieder in einen String zusammenzuführen.

Arrays

Ein Array repräsentiert eine feste Zahl von Elementen eines bestimmten Typs. Die Elemente in einem Array werden immer in einem zusammenhängenden Speicherabschnitt gespeichert, wodurch man sehr schnell auf sie zugreifen kann.

Ein Array wird durch eckige Klammern nach dem Typ des Elements definiert. Die folgende Zeile deklariert ein Array mit fünf Zeichen:

```
char[] vowels = new char[5];
```

Eckige Klammern *indexieren* das Array auch, wodurch man auf ein bestimmtes Element über die Position zugreifen kann:

```
vowels[0] = 'a'; vowels[1] = 'e'; vowels[2] = 'i';

vowels[3] = 'o'; vowels[4] = 'u';


Console.WriteLine (vowels [1]);      // e
```

Damit wird »e« ausgegeben, weil Array-Indizes bei null beginnen. Wir können eine for-Schleife nutzen, um über jedes Element im Array zu iterieren. Die for-Schleife lässt in diesem Beispiel den Integer-Wert i von 0 bis 4 laufen:

```
for (int i = 0; i < vowels.Length; i++)

    Console.Write (vowels [i]);          // aeiou
```

Arrays implementieren zudem IEnumerable<T> (siehe »Enumeration und Iteratoren« auf Seite 138), sodass Sie Elemente über die Anweisung foreach aufzählen können:

```
foreach (char c in vowels) Console.Write (c);  // aeiou
```

Bei allen Array-Indizierungen prüft die Laufzeit die Grenzen. Wenn Sie einen

ungültigen Index nutzen, wird eine IndexOutOfRange-Exception ausgelöst:

```
vowels[5] = 'y';  // Laufzeitfehler
```

Die Eigenschaft Length eines Arrays liefert die Anzahl der Elemente im Array zurück. Ist ein Array einmal angelegt, kann seine Länge nicht mehr geändert werden. Der Namensraum System.Collection und die darunterliegenden Namensräume stellen weiterentwickelte Datenstrukturen bereit, zum Beispiel dynamisch in der Größe veränderbare Arrays und Dictionaries.

Mit einem *Array-Initialisierungsausdruck* können Sie ein Array in einem Schritt deklarieren und füllen, zum Beispiel so:

```
char[] vowels = new char[] {'a','e','i','o','u'};
```

Oder einfach so:

```
char[] vowels = {'a','e','i','o','u'};
```

Alle Arrays erben von der Klasse System.Array, in der Methoden und Eigenschaften für alle Arrays definiert sind. Dazu gehören Instanzeigenschaften wie Length und Rank, aber auch statische Methoden, um

- ein Array dynamisch anzulegen (CreateInstance),
- Elemente unabhängig vom Array-Typ zu lesen und zu schreiben (GetValue/SetValue),
- ein sortiertes Array (BinarySearch) oder ein unsortiertes Array (IndexOf, LastIndexOf, Find, FindIndex, FindLastIndex) zu durchsuchen,
- ein Array zu sortieren (Sort) oder
- ein Array zu kopieren (Copy).

Standard-Elementinitialisierung

Beim Erstellen eines Arrays werden dessen Elemente immer durch Standardwerte vorinitialisiert. Der Standardwert eines Typs ist das Ergebnis eines bitweisen Löschens seines Speicherbereichs. Stellen Sie sich zum Beispiel das Erstellen eines Arrays mit Integer-Werten vor. Da int ein Werttyp ist, werden 1.000 Integer-Zahlen in einem zusammenhängenden Speicherabschnitt zugewiesen. Der Standardwert für jedes

Element ist 0:

```
int[] a = new int[1000];

Console.Write (a[123]);        // 0
```

Bei Referenztyp-Elementen ist der Vorgabewert null.

Ein Array *selbst* ist immer ein Referenztyp-Objekt – unabhängig davon, welchen Typ seine Elemente haben. Beispielsweise ist Folgendes zulässig:

```
int[] a = null;
```

Mehrdimensionale Arrays

Mehrdimensionale Arrays gibt es in zwei Varianten: *rechteckig* und *ungleichförmig*. Rechteckige Arrays stellen einen *n*-dimensionalen Speicherblock dar, während es sich bei ungleichförmigen Arrays um Arrays aus Arrays handelt.

Rechteckige Arrays

Rechteckige Arrays werden deklariert, indem die einzelnen Dimensionen durch Kommata getrennt werden. Mit der folgenden Anweisung wird ein rechteckiges, zweidimensionales Array deklariert, bei dem die Dimensionen 3 × 3 sind:

```
int[,] matrix = new int [3, 3];
```

Die Methode GetLength eines Arrays liefert die Länge für eine gegebene Dimension zurück (beginnend bei 0):

```
for (int i = 0; i < matrix.GetLength(0); i++)
    for (int j = 0; j < matrix.GetLength(1); j++)
        matrix [i, j] = i * 3 + j;
```

Ein rechteckiges Array kann wie folgt initialisiert werden (jedes Element wird in diesem Beispiel so initialisiert wie im vorigen Beispiel):

```
int[,] matrix = new int[,]
{
    {0,1,2},
    {3,4,5},
    {6,7,8}
};
```

(Der fett gedruckte Code kann in Deklarationsanweisungen wie dieser weggelassen werden.)

Ungleichförmige Arrays

Ungleichförmige Arrays werden mit aufeinanderfolgenden eckigen Klammern deklariert, mit denen jede Dimension repräsentiert wird. Hier sehen Sie ein Beispiel für das Deklarieren eines unregelmäßigen zweidimensionalen Arrays, bei dem die äußerste Dimension 3 ist:

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

Die inneren Dimensionen werden in der Deklaration nicht spezifiziert, da anders als bei einem rechteckigen Array jedes innere Array eine eigene Länge haben kann. Jedes innere Array wird implizit nicht auf ein leeres Array initialisiert, sondern auf null. Jedes innere Array muss per Hand angelegt werden:

```
for (int i = 0; i < matrix.Length; i++)
{
    matrix[i] = new int [3];       // inneres Array erzeugen
    for (int j = 0; j < matrix[i].Length; j++)
        matrix[i][j] = i * 3 + j;
}
```

Ein ungleichförmiges Array kann folgendermaßen initialisiert werden (um ein Array zu erstellen, das dem aus dem letzten Beispiel entspricht und nur am Ende ein zusätzliches Element enthält):

```
int[][] matrix = new int[][]
{
    new int[] {0,1,2},
    new int[] {3,4,5},
    new int[] {6,7,8,9}
};
```

(Der fett gedruckte Code kann in Deklarationsanweisungen wie dieser weggelassen werden.)

Vereinfachter Array-Initialisierungsausdruck

Wir haben bereits gesehen, dass man den Array-Initialisierungsausdruck vereinfachen kann, indem man das Schlüsselwort new und die Typdeklaration weglässt:

```
char[] vowels = new char[] {'a','e','i','o','u'};
char[] vowels =          {'a','e','i','o','u'};
```

Ein weiterer Ansatz ist, den Typnamen nach dem Schlüsselwort new wegzulassen und den Compiler den Array-Typ *erschließen* zu lassen. Das ist eine praktische Kurzform, wenn man Arrays als Argumente übergibt. Betrachten Sie beispielsweise die folgende Methode:

```
void Foo (char[] data) { ... }
```

Wir können diese Methode mit einem Array aufrufen, das wir im Vorübergehen erstellen:

Foo (**new char[] {'a','e','i','o','u'}**); // Langform

Foo (**new[] {'a','e','i','o','u'}**); // Kurzform

Diese Kurzform ist notwendig, wenn Sie Arrays *anonymer Typen* erstellen, wie Sie später noch sehen werden .

Variablen und Parameter

Eine Variable repräsentiert einen Speicherbereich, der einen veränderbaren Wert enthält. Eine Variable kann eine *lokale Variable*, ein *Parameter* (value, ref oder out), ein *Feld* (*Instanz* oder *statisch*) oder ein *Array-Element* sein.

Der Stack und der Heap

Der Stack und der Heap sind die Orte, an denen Variablen und Konstanten abgelegt werden. Jeder hat bezüglich der Lebensspanne der Objekte eine andere Semantik.

Stack

Der Stack ist ein Speicherbereich, in dem lokale Variablen und Parameter gespeichert werden. Der Stack wächst und schrumpft, sobald Funktionen betreten und wieder verlassen werden. Schauen Sie sich die folgende Methode an (um es nicht zu kompliziert zu machen, wird die Prüfung des übergebenen Arguments ignoriert):

```
static int Factorial (int x)
{
    if (x == 0) return 1;
    return x * Factorial (x-1);
}
```

Diese Methode ist rekursiv, ruft sich also selbst auf. Bei jedem Aufruf der Methode wird Platz für ein neues int auf dem Stack reserviert, und bei jedem Verlassen der Methode wird int wieder freigegeben.

Heap

Der Heap ist ein Speicherabschnitt, in dem *Objekte* (also Instanzen von Referenztypen) liegen. Immer dann, wenn ein neues Objekt erzeugt wird, wird auf dem Heap dafür Platz reserviert und eine Referenz auf dieses Objekt zurückgegeben. Während der Ausführung eines Programms füllt sich der Heap, während neue Objekte

angelegt werden. Die Laufzeitumgebung hat einen Garbage Collector, der regelmäßig Objekte vom Heap entfernt, sodass Ihrem Programm nicht der Speicher ausgeht. Ein Objekt kann dann vom Heap entfernt werden, wenn es von keinem Objekt mehr referenziert wird, das selbst noch lebt.

Werttyp-Instanzen (und Objektreferenzen) leben dort, wo die Variable deklariert wurde. Wenn die Instanz als Feld innerhalb eines Klassentyps oder als Array-Element deklariert wurde, lebt diese Instanz auf dem Heap.

Sie können Objekte in C# nicht explizit löschen, so wie es in C++ möglich ist. Ein nicht referenziertes Objekt wird irgendwann vom Garbage Collector abgeräumt.

Der Heap wird auch genutzt, um statische Felder und Konstanten zu speichern. Anders als Objekte, die auf dem Heap untergebracht sind (und von der Garbage Collection gelöscht werden können), leben diese Daten, bis die Anwendungsdomäne ihrem Ende entgegengeht.

Sichere Zuweisung

C# nutzt ausschließlich sichere Zuweisungen. In der Praxis bedeutet dies, dass es außerhalb eines unsafe-Kontexts unmöglich ist, auf nicht initialisierten Speicher zuzugreifen. Die sichere Zuweisung hat drei Folgen:

- Lokalen Variablen muss ein Wert zugewiesen werden, bevor man lesend auf sie zugreifen kann.
- Beim Aufruf einer Methode müssen die Funktionsargumente gefüllt sein (es sei denn, sie sind als optional markiert – mehr Informationen finden Sie unter »Optionale Parameter« auf Seite 42).
- Alle anderen Variablen (wie Felder und Array-Elemente) werden zur Laufzeit automatisch initialisiert.

Der folgende Code führt zum Beispiel zu einem Kompilierungsfehler:

```
static void Main()
{
    int x;
```

```
    Console.WriteLine (x);      // Kompilierungsfehler
}
```

Wäre x stattdessen aber ein *Feld* der umschließenden Klasse, wäre das vollkommen zulässig und würde zur Ausgabe 0 führen.

Vorgabewerte

Alle Typinstanzen haben einen Vorgabewert. Dieser Vorgabewert vordefinierter Typen ist das Ergebnis der Nullsetzung der Speicherbits und ist bei Referenztypen null, bei numerischen Typen und Enums 0 , beim Typ char '\0' und beim Typ bool false.

Sie können den Vorgabewert für jeden Typ mit dem Schlüsselwort default anfordern (in der Praxis ist das, wie wir später sehen werden, bei Generics nützlich). Der Vorgabewert für einen selbst definierten Werttyp (also ein Struct) entspricht dem Vorgabewert für jedes Feld, das vom selbst definierten Typ angegeben wird.

Parameter

Eine Methode hat eine Reihe von Parametern. Parameter definieren die Argumente, die für diese Methode angegeben werden müssen. In diesem Beispiel hat die Methode Foo einen einzelnen Parameter p vom Typ int:

```
static void Foo (int p)         // p ist ein Parameter
{
    ...
}

static void Main() { Foo (8); }  // 8 ist ein Argument
```

Sie können mit den Modifikatoren ref und out beeinflussen, wie Parameter übergeben werden.

Parameter-Modifikator	Übergeben als	Variable muss zugewiesen sein
keiner	Wert	beim *Betreten*

ref	Referenz	beim *Betreten*
out	Referenz	beim *Verlassen*

Argumente als Wert übergeben

Standardmäßig werden Argumente in C# als Wert übergeben (*by Value*), was mit Abstand der am häufigsten vorkommende Fall ist. Das bedeutet, dass eine Kopie des Werts erstellt wird, wenn man ihn an die Methode übergibt:

```
static void Foo (int p)
{
    p = p + 1;              // p um 1 erhöhen
    Console.WriteLine (p);  // p auf dem Bildschirm ausgeben
}
static void Main()
{
    int x = 8;
    Foo (x);                // eine Kopie von x erstellen
    Console.WriteLine (x);  // x ist immer noch 8
}
```

Weist man p einen neuen Wert zu, ändert das nicht den Inhalt von x, weil p und x an unterschiedlichen Stellen im Speicher liegen.

Wenn ein Referenztyp-Argument als Wert übergeben wird, wird die Referenz kopiert, nicht das Objekt. Foo sieht dasselbe StringBuilder-Objekt, das Main instanziiert hat, besitzt aber eine eigene Referenz darauf. Das bedeutet also, dass sb und fooSB unterschiedliche Variablen sind, die auf dasselbe StringBuilder-Objekt verweisen:

```
static void Foo (StringBuilder fooSB)
```

```
{
    fooSB.Append ("Test");
    fooSB = null;
}
static void Main()
{
    StringBuilder sb = new StringBuilder();
    Foo (sb);
    Console.WriteLine (sb.ToString());    // Test
}
```

Da fooSB eine Kopie einer Referenz ist, wird sb nicht zu null, wenn man fooSB auf null setzt. (Wenn allerdings fooSB mit dem Modifikator ref deklariert und aufgerufen wurde, würde sb null werden.)

Der Modifikator ref

Um eine Referenz zu übergeben (*by Reference*), stellt C# den Modifikator ref bereit. Im folgenden Beispiel verweisen p und x auf denselben Speicherbereich:

```
static void Foo (ref int p)
{
    p = p + 1;
    Console.WriteLine (p);
}
static void Main()
```

```
{
    int x = 8;
    Foo (ref x);          // x by Reference übergeben
    Console.WriteLine (x);   // x ist jetzt 9
}
```

Weist man p nun einen neuen Wert zu, ändert sich auch der Inhalt von x. Beachten Sie, dass der Modifikator ref sowohl beim Schreiben als auch beim Aufrufen der Methode notwendig ist. Damit wird sehr deutlich, was hier passiert.

Ein Parameter kann als Referenz oder als Wert übergeben werden – unabhängig davon, ob der Parametertyp ein Referenztyp oder ein Werttyp ist.

Der Modifikator out

Ein out-Argument ist wie ein ref-Argument, jedoch mit folgenden Ausnahmen:

- Es muss nicht zugewiesen sein, bevor es der Funktion *übergeben* wird.
- Es muss zugewiesen sein, bevor die Funktion *verlassen* wird.

Der Modifikator out wird meist genutzt, um mehrere Rückgabewerte in einer Methode zu haben.

out-Variablen und Discards (C# 7)

Seit C# 7 können Sie Variablen beim Aufrufen von Methoden mit out-Parametern spontan deklarieren:

```
int.TrypParse ("123", out int x);
Console.WriteLine (x);
```

Das entspricht:

```
int x;

int.TryParse ("123", out x);

Console.WriteLine (x);
```

Bei Aufruf von Methoden mit mehreren out-Parametern können Sie alle für Sie uninteressanten Parameter durch einen Unterstrich »verwerfen«. Angenommen, SomeBigMethod wurde mit fünf out-Parametern definiert. Dann können alle bis auf den dritten wie folgt ignoriert werden:

```
SomeBigMethod (out _, out _, out int x, out _, out _);

Console.WriteLine (x);
```

Der Modifikator params

Der Modifikator params kann für den letzten Parameter einer Methode angegeben werden. Dann akzeptiert die Methode eine beliebige Zahl an Argumenten eines bestimmten Typs. Der Parametertyp muss als Array deklariert werden:

```
static int Sum (params int[] ints)
{
    int sum = 0;
    for (int i = 0; i < ints.Length; i++) sum += ints[i];
    return sum;
}
```

Diese Methode können wir folgendermaßen aufrufen:

```
Console.WriteLine (Sum (1, 2, 3, 4));   // 10
```

Sie können ein params-Argument auch als normales Array angeben. Der vorangegangene Aufruf entspricht dem folgenden:

```
Console.WriteLine (Sum(new int[] { 1, 2, 3, 4 } ));
```

Optionale Parameter

Seit C# 4.0 können Methoden, Konstruktoren und Indexer *optionale Parameter* deklarieren. Ein Parameter ist optional, wenn er in seiner Deklaration einen *Vorgabewert* definiert:

```
void Foo (int x = 23) { Console.WriteLine (x); }
```

Optionale Parameter können ausgespart werden, wenn die Methode aufgerufen wird:

```
Foo();    // 23
```

Das Vorgabeargument 23 wird tatsächlich an den optionalen Parameter x übergeben – der Compiler schreibt den Wert 23 auf der aufrufenden Seite in den kompilierten Code. Der vorangegangene Aufruf von Foo entspricht semantisch dem Aufruf

```
Foo (23);
```

weil der Compiler einfach den Vorgabewert einsetzt, wenn ein optionaler Parameter genutzt wird.

Wird einer öffentlichen Methode, die von einer anderen Assembly aufgerufen wird, ein optionaler Parameter hinzugefügt, müssen beide Assemblies neu kompiliert werden – also genau wie bei einem obligatorischen Parameter.

Der Vorgabewert für einen optionalen Parameter muss durch einen konstanten Ausdruck oder den parameterlosen Konstruktor eines Werttyps angegeben werden. Optionale Parameter dürfen nicht mit den Modifikatoren ref oder out markiert werden.

Notwendige Parameter müssen in der Methodendeklaration und im Methodenaufruf *vor* optionalen Parametern stehen (ausgenommen params-Argumente, die immer noch

stets an letzter Stelle kommen). Im folgenden Beispiel wird x der explizit angegebene Wert 1 übergeben und y der Vorgabewert 0:

```
void Foo (int x = 0, int y = 0)
{
    Console.WriteLine (x + ", " + y);
}
void Test()
{
    Foo(1);    // 1, 0
}
```

Wenn Sie das Gegenteil tun wollen (x den Vorgabewert und y einen expliziten Wert übergeben), müssen Sie optionale Parameter mit benannten Argumenten kombinieren.

Benannte Argumente

Statt über die Position können Sie Argumente auch über den Namen identifizieren, zum Beispiel so:

```
void Foo (int x, int y)
{
    Console.WriteLine (x + ", " + y);
}
void Test()
{
    Foo (x:1, y:2);  // 1, 2
```

```
}
```

Benannte Argumente können in beliebiger Abfolge auftreten. Die beiden folgenden Aufrufe von Foo haben die gleiche Bedeutung:

```
Foo (x:1, y:2);

Foo (y:2, x:1);
```

Sie können benannte und positionelle Argumente mischen, die benannten Parameter müssen aber an letzter Stelle erscheinen:

```
Foo (1, y:2);
```

Benannte Parameter sind insbesondere in Kombination mit optionalen Parametern nützlich. Betrachten Sie beispielsweise die folgende Methode:

```
void Bar (int a=0, int b=0, int c=0, int d=0) { ... }
```

Diese Methode können wir folgendermaßen nur mit einem Wert für d aufrufen:

```
Bar (d:3);
```

Das ist besonders hilfreich, wenn Sie COM-APIs aufrufen.

var – implizit typisierte lokale Variablen

Häufig deklarieren und initialisieren Sie eine Variable in einem Schritt. Wenn der Compiler den Typ aus dem Initialisierungsausdruck ermitteln kann, können Sie das Wort var statt der Typdeklaration nutzen, zum Beispiel so:

```
var x = "Hallo";

var y = new System.Text.StringBuilder();

var z = (float)Math.PI;
```

Das ist vollkommen äquivalent zu Folgendem:

```
string x = "Hallo";
System.Text.StringBuilder y =
    new System.Text.StringBuilder();
float z = (float)Math.PI;
```

Aufgrund dieser direkten Äquivalenz sind implizit typisierte Variablen statisch typisiert. Der folgende Code würde zum Beispiel zu einem Kompilierungsfehler führen:

```
var x = 5;
x = "Hallo";   // Kompilierungsfehler; x hat den Typ int.
```

Im Abschnitt »Anonyme Typen« auf Seite 151 beschreiben wir ein Szenario, bei dem die Verwendung von`var` unumgänglich ist.

Ausdrücke und Operatoren

Ein *Ausdruck* entspricht prinzipiell einem Wert. Die einfachste Form eines Ausdrucks ist eine Konstante (wie 123) oder eine Variable (wie x). Ausdrücke können mithilfe von Operatoren umgewandelt und kombiniert werden. Ein *Operator* erwartet einen oder mehrere *Operanden* als Eingabe, um einen neuen Ausdruck auszugeben.

```
12 * 30   // * ist ein Operator; 12 und 30 sind Operanden.
```

Komplexe Ausdrücke können gebaut werden, da ein Operand selbst ein Ausdruck sein kann, zum Beispiel der Operand (12 * 30) im folgenden Beispiel:

```
1 + (12 * 30)
```

Operatoren sind in C# als *unär*, *binär* oder *ternär* klassifiziert, abhängig von der Zahl der Operanden, mit denen sie arbeiten (einem, zwei oder drei). Die binären Operatoren verwenden immer die *Infix*-Notation, bei der der Operator zwischen den beiden Operanden platziert wird.

Operatoren , die wesentliche Bestandteile des Grundgerüsts der Sprache sind, werden als *primäre Operatoren* bezeichnet; ein Beispiel dafür ist der Methodenaufrufoperator. Ein Ausdruck, der keinen Wert hat, wird als *leerer Ausdruck* bezeichnet:

```
Console.WriteLine (1)
```

Da ein leerer Ausdruck keinen Wert hat, kann er nicht als Operand genutzt werden, um einen komplexeren Ausdruck aufzubauen:

```
1 + Console.WriteLine (1)     // Kompilierungsfehler
```

Zuweisungsausdrücke

Ein Zuweisungsausdruck nutzt den Operator =, um einer Variablen das Ergebnis eines anderen Ausdrucks zuzuweisen:

```
x = x * 5
```

Ein Zuweisungsausdruck ist kein void-Ausdruck. Er gibt den zugewiesenen Wert aus und kann daher in einem anderen Ausdruck verwendet werden. Im folgenden Beispiel wird x der Wert 2 und y der Wert 10 zugewiesen:

```
y = 5 * (x = 2)
```

Auf diese Weise können mehrere Variablen initialisiert werden:

```
a = b = c = d = 0
```

Die *zusammengesetzten Zuweisungsoperatoren* sind syntaktische Abkürzungen, die eine Zuweisung mit einem anderen Operator kombinieren:

```
x *= 2    // entspricht x = x * 2

x <<= 1   // entspricht x = x << 1
```

(Eine feine Ausnahme von dieser Regel finden Sie bei *Events*, die wir später beschreiben werden: Die Operatoren += und -= werden bei diesen besonders behandelt und auf die Zugriffsmethoden add und remove abgebildet.)

Priorität und Assoziativität von Operatoren

Wenn ein Ausdruck mehrere Operatoren enthält, bestimmen *Priorität* und *Assoziativität* die Auswertungsreihenfolge. Operatoren mit einer höheren Priorität werden vor Operatoren mit einer niedrigeren Priorität ausgeführt. Wenn die Operatoren die gleiche Priorität haben, bestimmt die Assoziativität die Reihenfolge der Auswertung.

Priorität

Der Ausdruck 1 + 2 * 3 wird als 1 + (2 * 3) ausgewertet, weil * eine höhere Priorität hat als +.

Linksassoziative Operatoren

Binäre Operatoren (mit Ausnahme von Zuweisungen, dem Lambda-Operator und dem

Null-Verbindungsoperator) sind *linksassoziativ*. Anders gesagt, werden sie von links nach rechts ausgewertet. Beispielsweise wird der Ausdruck 8/4/2 aufgrund der Linksassoziativität als (8/4)/2 ausgewertet. Natürlich können Sie Ihre eigenen Klammern einsetzen, um die Auswertungsreihenfolge zu ändern.

Rechtsassoziative Operatoren

Die Zuweisungs- und Lambda-Operatoren, der Null-Verbindungsoperator und der (ternäre) Bedingungsoperator sind *rechtsassoziativ*. Anders gesagt, werden sie von rechts nach links ausgewertet. Die Rechtsassoziativität ermöglicht die Kompilation von Mehrfachzuweisungen wie x=y=3: Das funktioniert, weil erst y 3 zugewiesen wird und dann das Ergebnis dieses Ausdrucks (3) x.

Operatorentabelle

Die folgende Tabelle führt die Operatoren von C# der Priorität nach auf. Operatoren, die unter ein und derselben Unterüberschrift aufgeführt werden, haben die gleiche Priorität. Von Benutzern überladbare Operatoren erläutern wir im Abschnitt »Überladen von Operatoren« auf Seite 188.

Operatorsymbol	Operatorenname	Beispiel	Überladbar
Primär (höchste Priorität)			
.	Member-Zugriff	x.y	Nein
->	Zeiger auf Struct (unsicher)	x->y	Nein
()	Funktionsaufruf	x()	Nein
[]	Array/Index	a[x]	Via Indexer
++	Post-Inkrement	x++	Ja
--	Post-Dekrement	x--	Ja
new	Instanz erstellen	new Foo()	Nein
stackalloc	Unsichere Stack-Allozierung	stackalloc(10)	Nein
typeof	Typ eines Bezeichners ermitteln	typeof(int)	Nein
nameof	Namen eines Bezeichners ermitteln	nameof(x)	Nein
checked	Ganzzahlüberlaufprüfung an	checked(x)	Nein

unchecked	Ganzzahlüberlaufprüfung aus	unchecked(x)	Nein
default	Vorgabewert	default(char)	Nein
Unär			
await	Warten auf	await mytask	Nein
sizeof	Struct-Größe ermitteln	sizeof(int)	Nein
+	Positiver Wert von	+x	Ja
-	Negativer Wert von	-x	Ja
!	Nicht	!x	Ja
~	Bit-Komplement	~x	Ja
++	Prä-Inkrement	++x	Ja
--	Dekrement	--x	Ja
()	Cast	(int)x	Nein
*	Wert an Adresse (unsicher)	*x	Nein
&	Adresse von Wert (unsicher)	&x	Nein
Multiplikativ			
*	Multiplikation	x * y	Ja
/	Division	x / y	Ja
%	Rest	x % y	Ja
Additiv			
+	Addition	x + y	Ja
-	Subtraktion	x - y	Ja
Verschiebung			
<<	Verschiebung nach links	x << 1	Ja
>>	Verschiebung nach rechts	x >> 1	Ja
Relational			
<	Kleiner	x < y	Ja
>	Größer	x > y	Ja

<=	Kleiner-gleich	x <= y	Ja
>=	Größer-gleich	x >= y	Ja
is	Typ entspricht Klasse oder Subklasse von	x is y	Nein
as	Typumwandlung	x as y	Nein
Gleichheit			
==	Gleich	x == y	Ja
!=	Ungleich	x != y	Ja
Logisches Und			
&	UND	x & y	Ja
Logisches Xoder			
^	Exklusives ODER	x ^ y	Ja
Logisches Oder			
|	ODER	x | y	Ja
Bedingungs-Und			
&&	Bedingungs-UND	x && y	Via &
Bedingungs-Oder			
||	Bedingungs-ODER	x || y	Via |
Bedingung (ternär)			
? :	Bedingung	isTrue ? thenThis : elseThis	Nein
Zuweisung und Lambda (geringste Priorität)			
=	Zuweisung	x = y	Nein
*=	Multiplikation und Zuweisung	x *= 2	Via *
/=	Division und Zuweisung	x /= 2	Via /
+=	Addition und Zuweisung	x += 2	Via +
-=	Subtraktion und Zuweisung	x -= 2	Via -
<<=	Linksverschiebung und	x <<= 2	Via <<

	Zuweisung		
>>=	Rechtsverschiebung und Zuweisung	x >>= 2	Via >>
&=	UND und Zuweisung	x &= 2	Via &
^=	XOR und Zuweisung	x ^= 2	Via ^
|=	ODER und Zuweisung	x |= 2	Via |
=>	Lambda	x => x + 1	Nein

Null-Operatoren

C# bietet zwei Operatoren, die die Arbeit mit Null-Werten vereinfachen: den Null-Verbindungsoperator und den Null-Bedingungsoperator.

Null-Verbindungsoperator

Der Operator ?? ist der *Null-Verbindungsoperator*. Er sagt: »Wenn der Operand nicht null ist, gib ihn mir – ansonsten gib mir einen Standardwert.« Hier sehen Sie ein Beispiel:

```
string s1 = null;

string s2 = s1 ?? "nichts"; // s2 enthält "nichts"
```

Ist der linksseitige Ausdruck nicht null, wird der rechtsseitige Ausdruck nie ausgewertet. Der Null-Verbindungsoperator funktioniert auch mit nullbaren Typen (siehe »Nullbare Typen« auf Seite 144).

Null-Bedingungsoperator

Der Operator ?. ist der *Null-Bedingungsoperator* (oder *Elvis-Operator*), der in C# 6 neu aufgenommen wurde. Er ermöglicht Ihnen, wie mit dem normalen Punktoperator Methoden aufzurufen und Member anzusprechen, jedoch wird mit diesem Operator bei einem linksseitigen Operanden der Ausdruck zu null, es wird also keine NullReferenceException geworfen:

```
System.Text.StringBuilder sb = null;

string s = sb?.ToString();   // kein Fehler, s ist null
```

Die zweite Zeile entspricht:

```
string s = (sb == null ? null : sb.ToString());
```

Trifft der Elvis-Operator auf null, verwirft er den Rest des Ausdrucks. Im folgenden

Beispiel wird s zu null ausgewertet, obwohl zwischen ToString() und ToUpper() ein Standard-Punktoperator genutzt wird:

```
System.Text.StringBuilder sb = null;

string s = sb?.ToString().ToUpper();   // kein Fehler
```

Ein wiederholter Einsatz des Elvis-Operators ist nur dann notwendig, wenn der direkt links danebenstehende Operand null sein kann. Der folgende Ausdruck reagiert robust, wenn x oder x.y null ist:

```
x?.y?.z
```

Er entspricht dem folgenden Code (abgesehen davon, dass x.y nur einmal ausgewertet wird):

```
x == null ? null
          : (x.y == null ? null : x.y.z)
```

Der endgültige Ausdruck muss eine Null aufnehmen können. Folgender Code ist ungültig, weil int keine Null akzeptiert:

```
System.Text.StringBuilder sb = null;

int length = sb?.ToString().Length;   // ungültig
```

Wir können das beheben, indem wir einen nullbaren Werttyp einsetzen (siehe »Nullbare Typen« auf Seite 144):

```
int? length = sb?.ToString().Length;  // okay:
                                       // int? kann null sein
```

Sie können den Null-Bedingungsoperator auch einsetzen, um eine void-Methode aufzurufen:

```
someObject?.SomeVoidMethod();
```

Ist someObject null, wird dies zu einer »Nicht-Operation«, statt eine NullReferenceException zu werfen.

Der Null-Bedingungsoperator kann mit den üblichen Typ-Membern genutzt werden, die wir in »Klassen« auf Seite 64 beschreiben – einschließlich Methoden, Feldern, Eigenschaften und Indexern. Er lässt sich auch gut mit dem Null-Verbindungsoperator kombinieren:

```
System.Text.StringBuilder sb = null;

string s = sb?.ToString() ?? "nichts";   // s wird zu "nichts"
```

Anweisungen

Funktionen enthalten *Anweisungen*, die in der Reihenfolge abgearbeitet werden, in der sie im Quelltext erscheinen. Ein *Anweisungsblock* ist eine Abfolge von Anweisungen, die zwischen geschweiften Klammern stehen (den Token {}).

Deklarationsanweisungen

Eine *Deklarationsanweisung* deklariert eine neue Variable und initialisiert sie optional mit einem Ausdruck. Eine Deklarationsanweisung endet mit einem Semikolon. Sie können mehrere Variablen desselben Typs in einer kommaseparierten Liste angeben:

```
bool rich = true, famous = false;
```

Eine *Deklaration einer Konstanten* ähnelt der Variablendeklaration, nur dass sie nach der Deklaration nicht mehr geändert werden kann und die Initialisierung zusammen mit der Deklaration geschehen muss (siehe »Konstanten« auf Seite 75):

```
const double c = 2.99792458E08;
```

Variablen mit lokaler Geltung

Die Geltung lokaler Variablen und lokaler Konstanten erstreckt sich über den aktuellen Block. Sie können keine weitere lokale Variable mit dem gleichen Namen im aktuellen Block oder eingebetteten Blöcken deklarieren.

Ausdrucksanweisungen

Ausdrucksanweisungen sind Ausdrücke, die auch gültige Anweisungen sind. In der Praxis sind das Ausdrücke, die etwas »tun«; anders gesagt, sind es Ausdrücke, die

- eine Variable zuweisen oder modifizieren,
- ein Objekt instanziieren oder
- eine Methode aufrufen.

Ausdrücke, die nichts davon tun, sind keine gültigen Anweisungen:

```
string s = "foo";

s.Length;          // Illegale Anweisung: Tut nichts!
```

Wenn Sie einen Konstruktor oder eine Methode aufrufen, der/die einen Wert zurückgibt, sind Sie nicht gezwungen, das Ergebnis zu nutzen. Wenn allerdings der Konstruktor oder die Methode keinen Zustand ändert, ist die Anweisung komplett nutzlos:

```
new StringBuilder();    // Legal, aber nutzlos.

x.Equals (y);           // Legal, aber nutzlos.
```

Auswahlanweisungen

Auswahlanweisungen steuern den Fluss der Programmausführung mithilfe von Bedingungen.

Die if-Anweisung

Eine if-Anweisung führt eine Anweisung aus, wenn ein bool-Ausdruck wahr ist, zum Beispiel so:

```
if (5 < 2 * 3)
    Console.WriteLine ("true");     // true
```

Die Anweisung kann ein Anweisungsblock sein:

```
if (5 < 2 * 3)
{
    Console.WriteLine ("true");     // true
    Console.WriteLine ("...")
}
```

Die else-Klausel

Eine if-Anweisung kann optional eine else-Klausel haben:

```
if (2 + 2 == 5)
    Console.WriteLine ("Wird nicht ausgeführt");
else
    Console.WriteLine ("False");        // False
```

In eine else-Klausel können Sie weitere if-Anweisungen einbetten:

```
if (2 + 2 == 5)
    Console.WriteLine ("Wird nicht ausgeführt");
else
    if (2 + 2 == 4)
        Console.WriteLine ("Wird ausgeführt"); // Wird ausgeführt.
```

Änderung des Programmflusses durch geschweifte Klammern

Eine else-Klausel gehört immer zur direkt davor befindlichen if-Anweisung im Anweisungsblock, zum Beispiel hier:

```
if (true)
    if (false)
        Console.WriteLine();
    else
        Console.WriteLine ("Wird ausgeführt");
```

Das ist semantisch identisch mit dem hier:

```
if (true)
{
    if (false)
        Console.WriteLine();
    else
        Console.WriteLine ("Wird ausgeführt");
}
```

Wir können den Programmablauf ändern, indem wir die Klammern verschieben:

```
if (true)
{
    if (false)
        Console.WriteLine();
}
else
    Console.WriteLine ("Wird nicht ausgeführt");
```

C# hat kein »elseif«-Schlüsselwort, aber mit dem folgenden Muster erzielt man das gleiche Ergebnis :

```
static void TellMeWhatICanDo (int age)
{
    if (age >= 40)
```

```
        Console.WriteLine ("Sie können Bundespräsident werden!");
    else if (age >= 18)
        Console.WriteLine ("Sie dürfen wählen!");
    else if (age >= 16)
        Console.WriteLine ("Sie dürfen Bier trinken!");
    else
        Console.WriteLine ("Sie müssen warten!");
}
```

Die switch-Anweisung

switch-Anweisungen ermöglichen Ihnen, den Programmablauf basierend auf einer Auswahl möglicher Werte einer Variablen zu verzweigen. switch-Anweisungen können zu saubererem Code als mit mehreren aufeinanderfolgenden if-Anweisungen führen, da ein Ausdruck bei switch-Anweisungen nur einmal ausgewertet werden muss:

```
static void ShowCard (int cardNumber)
{
    switch (cardNumber)
    {
        case 13:
            Console.WriteLine ("König");
            break;
        case 12:
```

```
            Console.WriteLine ("Königin");
            break;
        case 11:
            Console.WriteLine ("Bauer");
            break;
        default:    // ein anderer Kartenwert
            Console.WriteLine (cardNumber);
            break;
    }
}
```

Die Werte in jedem case-Ausdruck müssen Konstanten sein, was die erlaubten Typen auf die eingebauten Integraltypen, auf bool, char, enum-Typen und string beschränkt. Am Ende jeder case-Klausel müssen Sie explizit mit einer Art Sprunganweisung angeben, wo die Ausführung fortgesetzt werden soll . Hier sind die Optionen:

- break (springt an das Ende der switch-Anweisung)
- goto case x (springt zu einer anderen case-Klausel)
- goto default (springt zur default-Klausel)
- jede beliebige andere Sprunganweisung – also return, throw, continue oder goto *label*

Wenn für mehr als einen Wert der gleiche Code ausgeführt werden soll, können Sie die gemeinsamen case-Klauseln hintereinander auflisten:

```
switch (cardNumber)
{
    case 13:
```

```
        case 12:
        case 11:
            Console.WriteLine ("Bildkarte");
            break;
        default:
            Console.WriteLine ("Gewöhnliche Karte");
            break;
    }
```

Dieses Feature einer switch-Anweisung kann in Bezug auf das Schreiben saubereren Codes im Vergleich zu mehreren if-else-Anweisungen entscheidend sein.

Die switch-Anweisung mit Mustern (C# 7)

Seit C# 7 können Sie bei switch abhängig vom *Typ* vorgehen:

```
    static void TellMeTheType (object x)
    {
        switch (x)
        {
            case int i:
                Console.WriteLine ("Es ist ein int!");
                break;
            case string s:
                Console.WriteLine (s.Length);      // Wir können s nutzen
```

```
            break;
        case bool b when b == true          // Wenn b true ist
            Console.WriteLine ("True");
            break;
        case null:   // In C# 7 können Sie auch auf null prüfen.
            Console.WriteLine ("null");
            break;
    }
}
```

(Der object-Typ erlaubt eine Variable beliebigen Typs – siehe »Vererbung« auf Seite 79 und »Der Typ object« auf Seite 88.)

Jeder case-Fall gibt einen Typ an, der passen muss, und eine Variable, der der typisierte Wert zugewiesen werden soll, wenn die Prüfung erfolgreich ist. Anders als bei Konstanten gibt es keine Beschränkung der möglichen Typen. Die optionale when-Klausel legt eine Bedingung fest, die erfüllt sein muss.

Die Reihenfolge der case-Klauseln hat bei der Auswahl nach dem Typ eine Bedeutung – anders als bei der Auswahl nach Konstanten. Eine Ausnahme zu dieser Regel bildet die default-Klausel, die unabhängig von ihrer Position in der switch-Anweisung als Letztes ausgeführt wird.

Sie können mehrere case-Klauseln aufeinanderfolgen lassen. Im nächsten Code wird Console.WriteLine für einen beliebigen Gleitkommatyp ausgeführt, dessen Wert größer als 1000 ist:

```
switch (x)
{
    case float f when f > 1000:
    case double d when d > 1000:
```

```
case decimal m when m > 1000:

    Console.WriteLine ("f, d und m sind außerhalb des Scopes");

    break;
```

In diesem Beispiel können wir die Variablen f, d und m nur innerhalb der when-Klauseln einsetzen. Rufen wir Console.WriteLine auf, weiß dieses nicht, welche der drei Variablen zugewiesen sein werden, daher sorgt der Compiler dafür, dass keine im Scope ist.

Iterationsanweisungen

C# ermöglicht es, eine Abfolge von Anweisungen mit den Anweisungen while, do-while, for und foreach wiederholt auszuführen.

while- und do-while-Schleifen

while-Schleifen wiederholen Code, solange ein bool-Ausdruck wahr ist. Der Ausdruck wird geprüft, *bevor* die Schleife ausgeführt wird. Beispielsweise gibt Folgendes 012 aus:

```
int i = 0;

while (i < 3)

{                   // Die geschweiften Klammen

                                    // sind hier optional.

    Console.Write (i++);

}
```

do-while-Schleifen unterscheiden sich von while-Schleifen dadurch, dass sie den Ausdruck prüfen, *nachdem* der Anweisungsblock durchlaufen wurde (was sicherstellt, dass der Block immer zumindest einmal ausgeführt wird). So sieht das obige Beispiel mit einer do-while-Schleife aus:

```
int i = 0;

do

{

    Console.WriteLine (i++);

}

while (i < 3);
```

for-Schleifen

for-Schleifen sind wie while-Schleifen mit besonderen Klauseln für die *Initialisierung* und das *Iterieren* einer Schleifenvariablen. Eine for-Schleife besteht aus drei Klauseln:

```
for (Initialisierung; Bedingung; Iteration)

    Anweisung-oder-Anweisungsblock
```

Die *Initialisierungsklausel* wird vor dem Eintritt in die Schleife ausgeführt und initialisiert eine oder mehrere *Iterationsvariablen*.

Die *Bedingungsklausel* ist ein bool-Ausdruck, der *vor* jedem Schleifendurchlauf ausgeführt wird. Der Schleifeninhalt wird ausgeführt, wenn die Bedingung wahr ist.

Die *Iterationsklausel* wird *nach* jeder Ausführung des Schleifeninhalts ausgeführt. Sie wird üblicherweise genutzt, um die Iterationsvariable zu aktualisieren.

Folgendes gibt beispielsweise die Zahlen 0 bis 2 aus:

```
for (int i = 0; i < 3; i++)

    Console.WriteLine (i);
```

Die folgende Schleife gibt die ersten zehn Fibonacci-Zahlen aus (bei denen jede Zahl die Summe der beiden vorangegangenen Zahlen ist):

```
for (int i = 0, prevFib = 1, curFib = 1; i < 10; i++)
```

```
{
    Console.WriteLine (prevFib);
    int newFib = prevFib + curFib;
    prevFib = curFib; curFib = newFib;
}
```

Von den drei Teilen der for-Anweisung können beliebige weggelassen werden. Sie können eine unendliche Schleife wie folgt erzeugen (auch wenn man stattdessen while(true) nutzen kann):

```
for (;;) Console.WriteLine ("Unterbrich mich");
```

foreach-Schleifen

Die foreach-Schleife iteriert über jedes Element eines aufzählbaren Objekts. Die meisten Typen in C# und dem .NET Framework, die eine Menge oder Liste von Elementen repräsentieren, sind aufzählbar (*enumerable*). So sind zum Beispiel sowohl ein Array als auch ein String aufzählbar. Hier ein Beispiel für das Enumerieren der Zeichen eines Strings – vom ersten bis zum letzten Zeichen:

```
foreach (char c in "Bier")
    Console.WriteLine (c + " ");   // B i e r
```

Enumerierbare Objekte definieren wir in »Enumeration und Iteratoren« auf Seite 138.

Sprunganweisungen

Die C#-Srunganweisungen sind break, continue, goto, return und throw. Das Schlüsselwort throw behandeln wir in »try-Anweisungen und Exceptions« auf Seite 129.

Die break-Anweisung

Die break-Anweisung beendet die Ausführung des Bodys einer Iterations- oder

switch-Anweisung:

```
int x = 0;
while (true)
{
    if (x++ > 5) break;      // Aus der Schleife ausbrechen.
}
// Hier wird die Ausführung nach dem break fortgesetzt
...
```

Die continue-Anweisung

Die continue-Anweisung übergeht die restlichen Anweisungen einer Schleife und geht zur nächsten Iteration weiter. Die folgende Schleife überspringt alle geraden Zahlen:

```
for (int i = 0; i < 10; i++)
{
    if ((i % 2) == 0) continue;
    Console.Write (i + " ");    // 1 3 5 7 9
}
```

Die goto-Anweisung

Die goto-Anweisung übergibt die Ausführung an eine Marke (die mit einem vorangestellten Doppelpunkt angezeigt wird) in einem Anweisungsblock. Der folgende Code iteriert über die Zahlen 1 bis 5 und imitiert eine for-Schleife:

```
int i = 1;
```

```
startLoop:

if (i <= 5)

{

    Console.Write (i + " "); // 1 2 3 4 5

    i++;

    goto startLoop;

}
```

Die return-Anweisung

Die return-Anweisung beendet die Methode und muss einen Ausdruck liefern, dessen Typ dem Rückgabetyp der Methode entspricht, wenn die Methode nicht void ist:

```
static decimal AsPercentage (decimal d)

{

    decimal p = d * 100m;

    return p;     // Folgenden Wert an die aufrufende

                              // Methode liefern.

}
```

Eine return-Anweisung kann überall in einer Methode vorkommen (außer in einem finally-Block).

Namensräume

Ein *Namensraum* ist eine Domäne, in der Namen von Typen eindeutig sein müssen. Typen werden normalerweise innerhalb von hierarchischen Namensräumen organisiert – um Namenskonflikte zu vermeiden und die Typen leichter zu finden. So ist zum Beispiel der RSA-Typ, der die Verschlüsselung mit öffentlichen Schlüsseln behandelt, in folgendem Namensraum definiert:

```
System.Security.Cryptography
```

Ein Namensraum gehört direkt zum Namen eines Typs. Dieser Code ruft die Methode Create von RSA auf:

```
System.Security.Cryptography.RSA rsa =
    System.Security.Cryptography.RSA.Create();
```

Namensräume sind unabhängig von Assemblies, bei denen es sich um Auslieferungseinheiten wie *.exe* oder *.dll* handelt. Namensräume haben auch keinen Einfluss auf die Sichtbarkeit von Membern – public, internal, private und so weiter.

Das Schlüsselwort namespace definiert einen Namensraum für die Typen in diesem Block, zum Beispiel so:

```
namespace Outer.Middle.Inner
{
    class Class1 {}
    class Class2 {}
}
```

Die Punkte im Namensraum stehen für eine Hierarchie verschachtelter Namensräume. Der folgende Code ist mit dem vorigen Beispiel semantisch identisch:

```
namespace Outer
{
    namespace Middle
    {
        namespace Inner
        {
            class Class1 {}
            class Class2 {}
        }
    }
}
```

Sie können einen Typ über seinen *vollständig qualifizierten Namen* ansprechen, zu dem alle Namensräume gehören – vom äußersten bis zum innersten. So könnten wir zum Beispiel auf Class1 aus dem vorigen Beispiel über Outer.Middle.Inner.Class1 zugreifen.

Typen, die in keinem Namensraum definiert sind, liegen im *globalen Namensraum*. Zum globalen Namensraum gehören auch die obersten Namensräume wie Outer in unserem Beispiel.

Die using-Direktive

Die using-Direktive importiert einen Namensraum und ist eine praktische Möglichkeit, auf Typen zuzugreifen, ohne ihre vollständig qualifizierten Namen angeben zu müssen:

```
using Outer.Middle.Inner;
```

```
class Test     // Test ist im globalen Namensraum
{
    static void Main()
    {
        Class1 c;  // vollständig qualifizierter Name
                           // nicht erforderlich ...
    }
}
```

Eine using-Direktive kann innerhalb eines Namensraums selbst angegeben werden, um den Geltungsbereich der Direktive einzuschränken.

using static

Seit C# 6 können Sie nicht nur einen Namensraum, sondern mithilfe der Direktive using static auch einen bestimmten Typ importieren. Alle statischen Member dieses Typs können dann genutzt werden, ohne sie über den Typnamen qualifizieren zu müssen. Im folgenden Beispiel rufen wir die statische Methode WriteLine der Klasse Console auf:

```
using static System.Console;

class Test
{
    static void Main() { WriteLine ("Hallo"); }
}
```

Die using static-Direktive importiert alle erreichbaren statischen Member des Typs – einschließlich Feldern, Eigenschaften und eingebetteten Typen. Sie können diese Direktive auch auf EnumTypen anwenden (siehe »Enums« auf Seite 99), bei denen dann deren Member importiert werden. Sollte es bei mehreren statischen Importen Mehrdeutigkeiten geben, kann der C#-Compiler den richtigen Typ nicht aus dem Kontext ermitteln und meldet einen Fehler.

Regeln in einem Namensraum

Geltung von Namen

Namen, die in einem äußeren Namensraum deklariert sind, können unqualifiziert in inneren Namensräumen verwendet werden. In diesem Beispiel muss Class1 in Inner nicht qualifiziert werden:

```
namespace Outer
{
    class Class1 {}

    namespace Inner
    {
        class Class2 : Class1 {}
    }
}
```

Wenn Sie auf einen Typ in einem anderen Zweig Ihrer Namensraumhierarchie zugreifen wollen, können Sie einen teilweise qualifizierten Namen nutzen. Im folgenden Beispiel lassen wir SalesReport auf Common.ReportBase basieren:

```
namespace MyTradingCompany
{
    namespace Common
```

```
    {
        class ReportBase {}
    }
    namespace ManagementReporting
    {
        class SalesReport : Common.ReportBase {}
    }
}
```

Namen verdecken

Wenn der gleiche Typname sowohl in einem inneren als auch in einem äußeren Namensraum auftritt, gewinnt der innere Name. Um auf den äußeren Namen zugreifen zu können, müssen Sie seinen vollständig qualifizierten Namen verwenden.

Alle Typnamen werden zur Kompilierungszeit in vollständig qualifizierte Namen konvertiert. Der Intermediate-Language-(IL-)Code enthält keine unqualifizierten oder teilweise qualifizierten Namen.

Wiederholte Namensräume

Sie können eine Namensraumdeklaration wiederholen, solange die Typnamen im Namensraum nicht für Konflikte sorgen:

```
namespace Outer.Middle.Inner { class Class1 {} }

namespace Outer.Middle.Inner { class Class2 {} }
```

Die Klassen können sogar Quellcodedateien und Assemblies übergreifen.

Der Qualifizierer global::

Gelegentlich kann ein vollständig qualifizierter Typname mit einem inneren Namen in Konflikt geraten. Sie können C# dazu zwingen, den vollständig qualifizierten Typnamen zu nutzen, indem Sie ihm das Präfix global:: voranstellen:

```
global::System.Text.StringBuilder sb;
```

Aliase für Typen und Namensräume

Das Importieren eines Namensraums kann zu Kollisionen bei Typnamen führen. Anstatt den gesamten Namensraum zu importieren, können Sie nur die für Sie notwendigen Typen importieren und jedem davon einen Aliasnamen geben:

```
using PropertyInfo2 = System.Reflection.PropertyInfo;

class Program { PropertyInfo2 p; }
```

Es kann auch ein gesamter Namensraum mit einem Alias versehen werden:

```
using R = System.Reflection;

class Program { R.PropertyInfo p; }
```

Klassen

Die *Klasse* ist die am häufigsten verwendete Form eines Referenztyps. Die einfachste mögliche Klassendeklaration sieht so aus:

```
class Foo
{
}
```

Eine komplexere Klassendeklaration kann folgende Elemente enthalten:

Vor dem Schlüsselwort class	*Attribute* und *Klassenmodifikatoren*. Die nicht geschachtelten Klassenmodifikatoren sind public, internal, abstract, sealed, static, unsafe und partial.
Hinter *IhrKlassenname*	*Generische Typparameter*, eine *Basisklasse* und *Schnittstellen*.
Innerhalb der geschweiften Klammern	*Klassen-Member* (das sind *Methoden, Eigenschaften, Indexer, Events, Felder, Konstruktoren, überladene Operatoren, eingebettete Typen* und ein *Finalisierer*).

Felder

Ein *Feld* ist eine Variable, die ein Member einer Klasse oder eines Struct ist:

```
class Octopus
{
    string name;
    public int age = 10;
}
```

Ein Feld kann den Modifikator readonly haben, um zu verhindern, dass es nach dem Erstellen verändert wird. Einem nur lesbaren Feld kann lediglich während seiner Deklaration oder im Konstruktor des Typs, der es enthält, ein Wert zugewiesen werden.

Die Initialisierung eines Felds ist optional. Ein nicht initialisiertes Feld hat einen Standardwert (0, \0, null, false). Feldinitialisierer werden vor den Konstruktoren in der Reihenfolge ausgeführt, in der sie erscheinen.

Aus Gründen der Bequemlichkeit können Sie mehrere Felder desselben Typs in einer kommaseparierten Liste deklarieren. Das ist praktisch für Felder, die die gleichen Attribute und Feldmodifikatoren haben :

```
static readonly int legs = 8, eyes = 2;
```

Methoden

Eine Methode führt eine Aktivität als Abfolge von Anweisungen aus. Sie kann *Eingabedaten* vom aufrufenden Code bekommen, indem *Parameter* spezifiziert werden, und *Ausgabedaten* an den Aufrufenden zurückgeben, indem ein *Rückgabetyp* angegeben wird.

Eine Methode kann als Rückgabetyp void spezifizieren und damit zeigen, dass sie keinen Wert zurückgeben wird. Sie kann zudem über ref- und out-Parameter Daten an den Aufrufenden übermitteln.

Die *Signatur* einer Methode muss innerhalb des Typs eindeutig sein. Sie besteht aus dem Namen der Methode und der Reihenfolge der Parametertypen (aber nicht den Parameternamen oder dem Rückgabetyp).

Expression-bodied Methoden

Eine Methode, die aus einem einzelnen Ausdruck besteht, zum Beispiel:

```
int Foo (int x) { return x * 2; }
```

kann (seit C# 6) kompakter als *Expression-bodied Methode* geschrieben werden. Ein »dicker Pfeil« ersetzt die geschweiften Klammern und das Schlüsselwort return:

```
int Foo (int x) => x * 2;
```

Expression-bodied Methoden können auch einen void-Rückgabetyp besitzen:

```
void Foo (int x) => Console.WriteLine (x);
```

Methoden überladen

Ein Typ kann Methoden überladen (also mehrere Methoden mit demselben Namen haben), solange sich die Parametertypen unterscheiden. So können zum Beispiel die folgenden Methoden alle gemeinsam im selben Typ existieren:

```
void Foo (int x);

void Foo (double x);

void Foo (int x, float y);

void Foo (float x, int y);
```

Lokale Methoden (C# 7)

Mit C# 7 können Sie eine Methode innerhalb einer anderen Methode definieren:

```
void WriteCubes()
{
    Console.WriteLine (Cube (3));

    int Cube (int value) => value * value * value;
}
```

Die lokale Methode (in diesem Fall Cube) ist nur für die umschließende Methode (WriteCubes) sichtbar. Das vereinfacht den enthaltenden Typ und macht jedem, der sich den Code anschaut, direkt deutlich, dass Cube nirgendwo anders verwendet wird. Lokale Methoden können auf die lokalen Variablen und Parameter der umschließenden Methode zugreifen. Das hat eine Reihe von Konsequenzen, die wir in »Äußere Variablen übernehmen« auf Seite 126 beschreiben.

Instanzkonstruktoren

Konstruktoren führen Initialisierungscode für eine Klasse oder ein Struct aus. Ein Konstruktor wird wie eine Methode definiert, nur dass der Methodenname und der Rückgabewert auf den Namen des Typs reduziert werden, in dem sich der Konstruktor befindet:

```
public class Panda
{
    string name;            // Feld definieren
    public Panda (string n)   // Konstruktor definieren
    {
        name = n;             // Initialisierungscode
    }
}
...
Panda p = new Panda ("Petey");  // Konstruktor aufrufen
```

Seit C# 7 können aus einer Anweisung bestehende Konstruktoren als Expression-bodied Methode geschrieben werden:

```
public Panda (string n) => name = n;
```

Klassen und Structs können Konstruktoren überladen. Eine Überladung ruft über das Schlüsselwort this eine andere auf:

```
public class Wine
{
    public Wine (decimal price) {...}
```

```
    public Wine (decimal price, int year)
                        : this (price) {...}
}
```

Wenn ein Konstruktor einen anderen aufruft, wird der aufgerufene Konstruktor zuerst ausgeführt.

Sie können einem anderen Konstruktor einen *Ausdruck* wie folgt übermitteln:

```
public Wine (decimal price, DateTime year)
                    : this (price, year.Year) {...}
```

Der Ausdruck selbst kann nicht auf die Referenz this zurückgreifen, um zum Beispiel eine Instanzmethode aufzurufen. Allerdings können dabei statische Methoden genutzt werden.

Implizite parameterlose Konstruktoren

Für Klassen generiert der C#-Compiler automatisch einen parameterlosen Konstruktor, wenn (und nur wenn) Sie keinerlei eigene Konstruktoren definiert haben. Sobald Sie aber mindestens einen Konstruktor definiert haben, wird der parameterlose Konstruktor nicht mehr automatisch generiert.

Nicht öffentliche Konstruktoren

Konstruktoren müssen nicht öffentlich sein. Ein guter Grund für einen nicht öffentlichen Konstruktor ist, die Instanzerzeugung über einen statischen Methodenaufruf zu kontrollieren. Mit diesem Aufruf kann dann zum Beispiel ein Objekt aus einem Pool zurückgegeben werden, anstatt ein neues Objekt zu erzeugen, oder es wird abhängig von den Eingabewerten eine Instanz einer Subklasse zurückgegeben.

Dekonstruktoren (C# 7)

Während ein Konstruktor im Allgemeinen (als Parameter) einen Satz Werte mitbekommt und diese Feldern zuweist, geht ein Dekonstruktor umgekehrt vor – die Feldwerte werden an Variablen übertragen. Eine Dekonstruktormethode muss den Namen Deconstruct tragen und einen oder mehrere out-Parameter besitzen:

```
class Rectangle
{
    public readonly float Width, Height;

    public Rectangle (float width, float height)
    {
        Width = width; Height = height;
    }

    public void Deconstruct (out float width,
                             out float height)
    { width = Width; height = Height; }
}
```

Um den Dekonstruktor aufzurufen, verwenden wir die folgende spezielle Syntax:

```
var rect = new Rectangle (3, 4);
(float width, float height) = rect;
Console.WriteLine (width + " " + height) // 3 4
```

Die zweite Zeile enthält den Dekonstruktoraufruf. Sie erstellt zwei lokale Variablen und ruft dann die Deconstruct-Methode auf. Dies ist äquivalent zu:

```
rect.Deconstruct (out var width, out var height);
```

Dekonstruktoraufrufe erlauben eine implizite Typisierung, daher können wir unseren Aufruf wie folgt kürzer fassen:

```
(var width, var height) = rect;
```

oder noch einfacher:

```
var (width, height) = rect;
```

Sie können dem Aufrufer mehrere Dekonstruktionsoptionen bieten, indem Sie die Methode Deconstruct überladen.

Bei der Deconstruct-Methode kann es sich um eine Erweiterungsmethode handeln (siehe »Erweiterungsmethoden« auf Seite 149). Das ist ein nützlicher Trick, wenn Sie Typen dekonstruieren wollen, von denen Sie nicht der Autor sind.

Objektinitialisierer

Um die Objektinitialisierung zu vereinfachen, können die von außen erreichbaren Felder bzw. Eigenschaften eines Objekts über einen *Objektinitialisierer* direkt nach dem Erzeugen initialisiert werden. Schauen Sie sich zum Beispiel diese Klasse an:

```
public class Bunny
{
    public string Name;
    public bool LikesCarrots, LikesHumans;

public Bunny () {}
    public Bunny (string n) { Name = n; }
}
```

Mithilfe von Objektinitialisierern können Sie Bunny-Objekte wie folgt initialisieren:

```
Bunny b1 = new Bunny {
                                    Name="Bo",
                                    LikesCarrots = true,
                                    LikesHumans = false
                                };

Bunny b2 = new Bunny ("Bo") {
                                    LikesCarrots = true,
                                    LikesHumans = false
                                    };
```

Die Referenz this

Die Referenz this verweist auf die Instanz selbst. Im nächsten Beispiel nutzt die Methode Marry dieses this, um das Feld Mate von partner zu setzen:

```
public class Panda
{
    public Panda Mate;

    public void Marry (Panda partner)
    {
        Mate = partner;
        partner.Mate = this;
```

```
    }
}
```

Die Referenz this sorgt zudem für das Auflösen möglicher Mehrdeutigkeiten zwischen lokalen Variablen bzw. Parametern und einem Feld:

```
public class Test
{
    string name;
    public Test (string name) { this.name = name; }
}
```

Die Referenz this ist nur bei nicht statischen Membern einer Klasse oder eines Struct gültig.

Eigenschaften

Eigenschaften sehen von außen wie Felder aus, enthalten intern aber wie Methoden eine Logik. Beispielsweise können Sie mit einem Blick auf den folgenden Code nicht erkennen, ob CurrentPrice ein Feld oder eine Eigenschaft ist:

```
Stock msft = new Stock();
msft.CurrentPrice = 30;
msft.CurrentPrice -= 3;
Console.WriteLine (msft.CurrentPrice);
```

Eine Eigenschaft wird wie ein Feld deklariert, besitzt aber zusätzlich einen get/set-Block. So wird CurrentPrice als Eigenschaft implementiert:

```
public class Stock
```

```
{
    decimal currentPrice;  // das private "Hintergrundfeld"

    public decimal CurrentPrice    // die öffentliche Eigenschaft
    {
        get { return currentPrice; }
        set { currentPrice = value; }
    }
}
```

get und set sind *Eigenschafts-Accessors*. Der get-Accessor wird ausgeführt, wenn die Eigenschaft gelesen wird. Sie müssen einen Wert mit dem Typ der Eigenschaft zurückgeben. Der set-Accessor wird ausgeführt, wenn der Eigenschaft ein Wert zugewiesen wird. Er hat einen impliziten Parameter namens value mit dem Typ der Eigenschaft, den Sie meist einem privaten Feld zuweisen (in diesem Fall currentPrice).

Auch wenn Eigenschaften auf die gleiche Art und Weise angesprochen werden wie Felder, unterscheiden sie sich von ihnen darin, dass der Implementierer bei ihnen die vollständige Kontrolle über das Lesen und Schreiben der Werte erhält. Diese Kontrolle ermöglicht ihm, eine für ihn bevorzugte interne Repräsentation zu wählen, ohne die internen Details dem Anwender der Eigenschaft preisgeben zu müssen. In diesem Beispiel könnte die set-Methode eine Exception werfen, wenn sich value außerhalb eines gewissen Wertebereichs befände.

In diesem Buch nutzen wir öffentliche Felder, um die Beispiele einfach zu halten. In einer echten Anwendung werden Sie meist öffentliche Eigenschaften bevorzugen, um die Kapselung zu fördern.

Auf eine Eigenschaft lässt sich nur lesend zugreifen, wenn lediglich ein get-Accessor angegeben ist, während sie nur beschreibbar ist, wenn es lediglich einen set-Accessor gibt. Allerdings werden Eigenschaften, die nur beschreibbar sind, selten verwendet.

Eine Eigenschaft hat meist ein eindeutig zugehöriges privates Feld »im Hintergrund«, in dem die zugrunde liegenden Daten abgelegt werden. Das muss allerdings nicht der Fall sind – eine Eigenschaft kann auch aus anderen Daten berechnet werden.

```
decimal currentPrice, sharesOwned;

public decimal Worth
{
    get { return currentPrice * sharesOwned; }
}
```

Expression-bodied Eigenschaften

Seit C# 6 können Sie eine schreibgeschützte Eigenschaft wie die im vorigen Abschnitt kompakter als *Expression-bodied Eigenschaft* deklarieren. Ein »dicker Pfeil« ersetzt alle geschweiften Klammern sowie die Schlüsselwörter get und return:

```
public decimal Worth => currentPrice * sharesOwned;
```

Mit C# 7 wird dies noch weiter fortgesetzt, weil nun auch set-Accessoren Expression-bodied sein können:

```
public decimal Worth
{
    get => currentPrice * sharesOwned;
    set => sharesOwned = value / currentPrice;
}
```

Automatische Eigenschaften

Die häufigste Implementierung einer Eigenschaft ist ein Getter und/oder Setter, der

einfach ein privates Feld des gleichen Typs wie die Eigenschaft liest und schreibt. Eine Deklaration einer *automatischen Eigenschaft* weist den Compiler an, diese Implementierung bereitzustellen. Wir können das erste Beispiel dieses Abschnitts verbessern, indem wir CurrentPrice als automatische Eigenschaft deklarieren:

```
public class Stock
{
    public decimal CurrentPrice { get; set; }
}
```

Der Compiler generiert automatisch ein privates Feld mit vom Compiler generierten Namen im Hintergrund, auf das nicht zugegriffen werden kann. Der set-Accessor kann als private oder als protected gekennzeichnet werden, wenn Sie die Eigenschaft als für andere Typen nur lesbar bereitstellen wollen.

Eigenschaftsinitialisierer

Seit C# 6 können Sie automatische Eigenschaften wie schon Felder mit einem *Eigenschaftsinitialisierer* ausstatten:

```
public decimal CurrentPrice { get; set; } = 123;
```

Damit erhält CurrentPrice den Initialwert 123. Eigenschaften mit einem Initialisierer können schreibgeschützt sein:

```
public int Maximum { get; } = 999;
```

Wie schreibgeschützte Felder können auch schreibgeschützte automatische Eigenschaften im Konstruktor des Typs zugewiesen werden. Das ist nützlich, wenn man *unveränderliche* (schreibgeschützte) Typen erstellt.

Sichtbarkeit von get und set

Die get- und set-Accessors können unterschiedliche Sichtbarkeit haben. Oft hat man eine public Eigenschaft mit einem internal oder private Zugriffsmodifikator für den Setter :

```
private decimal x;

public decimal X

{

    get        { return x; }

    private set { x = Math.Round (value, 2); }

}
```

Beachten Sie, dass Sie die Eigenschaft selbst mit der weiteren Sichtbarkeit (hier public) deklarieren und den Modifikator dem Accessor hinzufügen, der *weniger* sichtbar sein soll.

Indexer

Indexer stellen eine natürliche Syntax für den Zugriff auf die Elemente einer Klasse oder eines Struct bereit, die eine Liste oder ein Dictionary kapselt. Indexer ähneln Eigenschaften, werden aber über ein Indexargument statt über einen Eigenschaftsnamen angesprochen. Die Klasse string hat einen Indexer, mit dem Sie auf jeden einzelnen char-Wert über einen int-Index zugreifen können:

```
string s = "hello";

Console.WriteLine (s[0]); // 'h'

Console.WriteLine (s[3]); // 'l'
```

Die Syntax für die Verwendung von Indexern entspricht der für die Verwendung von Arrays, aber die Indexargumente können einen beliebigen Typ haben. Indexer lassen sich null-bedingt aufrufen, indem Sie vor den eckigen Klammern ein Fragezeichen einfügen (siehe »Null-Operatoren« auf Seite 49):

```
string s = null;

Console.WriteLine (s?[0]); // Schreibt nichts; kein Fehler.
```

Implementieren eines Indexers

Um einen Indexer zu schreiben, definieren Sie eine Eigenschaft mit dem Namen this und geben die Argumente in eckigen Klammern an:

```
class Sentence
{
    string[] words = "The quick brown fox".Split( );

    public string this [int wordNum] // Indexer
    {
    get { return words [wordNum]; }
    set { words [wordNum] = value; }
    }
}
```

Und so könnten wir den Indexer nutzen:

```
Sentence s = new Sentence( );
Console.WriteLine (s[3]);      // fox
s[3] = "kangaroo";
Console.WriteLine (s[3]);      // kangaroo
```

Ein Typ kann mehrere Indexer deklarieren, bei denen jeder Parameter einen anderen Typ hat. Ein Indexer kann auch mehr als einen Parameter erwarten:

```
public string this [int arg1, string arg2]
{
```

```
    get { ... } set { ... }

}
```

Wenn Sie den set-Accessor weglassen, wird ein Indexer schreibgeschützt. Mit der Expression-bodied Syntax (seit C# 6) lässt sich die Definition kompakter halten:

```
public string this [int wordNum] => words [wordNum];
```

Konstanten

Eine *Konstante* ist ein Feld, dessen Wert sich nicht ändern kann. Eine Konstante wird statisch beim Kompilieren ausgewertet, und ihr Wert wird bei jeder Verwendung vom Compiler direkt eingesetzt (so ähnlich wie ein Makro in C++). Eine Konstante kann einen der eingebauten numerischen Typen bool, char und string oder einen enum-Typ nutzen.

Eine Konstante wird mit dem Schlüsselwort const deklariert und muss mit einem Wert initialisiert werden:

```
public class Test

{

    public const string Message = "Hello World";

}
```

Eine Konstante ist deutlich restriktiver als ein static readonly-Feld, und zwar sowohl bezüglich der nutzbaren Typen als auch bei der Semantik der Feldinitialisierung. Eine Konstante unterscheidet sich von einem static readonly-Feld zudem darin, dass die Auswertung der Konstanten schon beim Kompilieren stattfindet. Konstanten können auch lokal in einer Methode deklariert werden :

```
static void Main()

{

    const double twoPI = 2 * System.Math.PI;
```

```
    ...
}
```

Statische Konstruktoren

Ein statischer Konstruktor wird einmal pro *Typ* statt einmal pro *Instanz* ausgeführt. Ein Typ kann nur einen statischen Konstruktor definieren; dieser muss parameterlos sein und den gleichen Namen wie der Typ haben:

```
class Test
{
    static Test() { Console.Write ("Typ initialisiert"); }
}
```

Die Laufzeitumgebung ruft einen statischen Konstruktor automatisch auf, und zwar unmittelbar bevor der Typ genutzt wird. Das wird von zwei Dingen ausgelöst: einer Instanziierung des Typs und dem Zugriff auf einen der statischen Member des Typs.

Wenn ein statischer Konstruktor eine unbehandelte Ausnahme auslöst, wird dieser Typ für die gesamte Lebensdauer der Anwendung unbrauchbar.

Statische Feldinitialisierer werden unmittelbar vor dem Aufruf des statischen Konstruktors ausgeführt. Wenn ein Typ keinen statischen Konstruktor hat, werden Feldinitialisierer direkt vor der Verwendung des Typs ausgeführt – oder irgendwann vorher, wenn es der Laufzeit beliebt.

Statische Klassen

Eine Klasse kann als static gekennzeichnet werden, was besagt, dass sie vollständig aus statischen Membern besteht und man nicht von ihr ableiten kann. Die Klassen System.Console und System.Math sind gute Beispiele für statische Klassen.

Finalizer

Finalizer sind Klassenmethoden, die ausgeführt werden, bevor der Garbage Collector den Speicher eines nicht mehr referenzierten Objekts freigibt. Die Syntax für einen Finalizer ist der Name der Klasse, dem das Zeichen ~ vorangestellt ist:

```
class Class1
{
    ~Class1() { ... }
}
```

C# übersetzt einen Finalizer in eine Methode, die die Finalize-Methode der Klasse object überschreibt.

Seit C# 7 können Finalizer, die aus einer einzelnen Anweisung bestehen, in der Expression-bodied Syntax geschrieben werden.

Partielle Typen und Methoden

Partielle Typen ermöglichen es, eine Typdefinition aufzuteilen – meist auf mehrere Dateien. Häufig wird das für automatisch generierte Klassen aus anderen Quellen (zum Beispiel aus einer Visual-Studio-Vorlage) genutzt, bei denen die Klasse dann um eigene Methoden ergänzt werden kann, zum Beispiel so:

```
// PaymentFormGen.cs - auto-generated
partial class PaymentForm { ... }
// PaymentForm.cs - handgeschrieben
partial class PaymentForm { ... }
```

Jeder Teil muss die Deklaration partial einschließen.

Die Teilnehmer dürfen keine Member haben, die zu Konflikten führen. So kann zum Beispiel ein Konstruktor mit den gleichen Parametern nicht wiederholt werden. Partielle Typen werden komplett vom Compiler aufgelöst, daher muss jeder Teilnehmer beim Kompilieren verfügbar sein und sich in der gleichen Assembly

befinden.

Eine Basisklasse kann auf einem oder mehreren Teilen angegeben werden (solange die spezifizierte Basisklasse die gleiche ist). Außerdem kann jeder Teilnehmer unabhängig zu implementierende Schnittstellen angeben. Basisklassen und Schnittstellen behandeln wir in den Abschnitten »Vererbung« auf Seite 79 und »Interfaces« auf Seite 95 ausführlicher.

Partielle Methoden

Ein partieller Typ kann *partielle Methoden* enthalten. Über diese kann ein automatisch generierter partieller Typ konfigurierbare Hooks für manuell geschriebene Teile anbieten, zum Beispiel so:

```
partial class PaymentForm    // in automatisch generierter
                             // Datei
{
    partial void ValidatePayment (decimal amount);
}

partial class PaymentForm    // in handgeschriebener Datei
{
    partial void ValidatePayment (decimal amount)
    {
        if (amount > 100) Console.Write ("Teuer!");
    }
}
```

Eine partielle Methode besteht aus zwei Teilen: einer *Definition* und einer *Implementierung*. Die Definition wird typischerweise von einem Codegenerator geschrieben, während die Implementierung meist per Hand erstellt wird. Wird keine Implementierung bereitgestellt, wird die Definition der partiellen Methode beim

Kompilieren entfernt. Damit kann automatisch generierter Code beim Bereitstellen von Hooks großzügig sein, ohne dass man sich um explodierende Codemengen sorgen müsste. Partielle Methoden müssen void sein und sind implizit private.

Der Operator nameof

Der Operator nameof (eingeführt in C# 6) gibt den Namen eines beliebigen Symbols (Typ, Member, Variable und so weiter) als String zurück:

```
int count = 123;

string name = nameof (count);   // name ist "count"
```

Sein Vorteil gegenüber dem schlichten Definieren eines Strings ist die statische Typprüfung. Tools wie Visual Studio können die Symbolreferenz verstehen, und wenn Sie das fragliche Symbol umbenennen, werden alle Referenzen ebenfalls umbenannt.

Um den Namen eines Typ-Members wie zum Beispiel eines Felds oder einer Eigenschaft zu erhalten, geben Sie auch den Typ selbst an. Das funktioniert sowohl mit statischen als auch mit Instanz-Membern:

```
string name = nameof (StringBuilder.Length);
```

Das wird zu Length ausgewertet. Um StringBuilder.Length zu erhalten, würden Sie so vorgehen müssen:

```
nameof(StringBuilder)+"."+nameof(StringBuilder.Length);
```

Vererbung

Eine Klasse kann von einer Klasse *erben*, damit die ursprüngliche Klasse erweitert oder angepasst wird. Durch das Erben von einer Klasse können Sie die Funktionalität in dieser Klasse nutzen, statt alles von Grund auf wieder selbst aufbauen zu müssen. Eine Klasse kann nur von einer einzigen Klasse erben, kann aber selbst an viele andere Klassen weitervererben, wodurch eine Klassenhierarchie entsteht. In diesem Beispiel wollen wir damit beginnen, eine Klasse namens Asset zu erstellen:

```
public class Asset { public string Name; }
```

Als Nächstes definieren wir Klassen mit den Namen Stock und House, die von Asset erben werden. Stock und House erhalten alles, was Asset hat, und dazu noch alle Member, die sie selbst definieren:

```
public class Stock : Asset   // erbt von Asset
{
    public long SharesOwned;
}

public class House : Asset   // erbt von Asset
{
    public decimal Mortgage;
}
```

Und so können wir diese Klassen nutzen:

```
Stock msft = new Stock { Name="MSFT",
                                        SharesOwned= 1000 };
```

```
Console.WriteLine (msft.Name);          // MSFT

Console.WriteLine (msft.SharesOwned);   // 1000


House mansion = new House { Name="Mansion",

                                        Mortgage=250000 };


Console.WriteLine (mansion.Name);       // Mansion

Console.WriteLine (mansion.Mortgage);  // 250000
```

Die *Subklassen* Stock und House erben die Eigenschaft Name von der *Basisklasse* Asset.

Subklassen werden auch als *abgeleitete Klassen* bezeichnet.

Polymorphie

Referenzen sind polymorph. Das bedeutet, dass eine Variable des Typs *x* auf Instanzen von Typen verweisen kann, die Subklassen von *x* sind. Betrachten Sie beispielsweise die folgende Methode:

```
public static void Display (Asset asset)

{

    System.Console.WriteLine (asset.Name);

}
```

Diese Methode kann ein Stock- und ein House-Objekt anzeigen, da beides Assets sind. Polymorphie basiert darauf, dass die Subklassen (Stock und House) alle Features ihrer Basisklasse (Asset) besitzen. Das Gegenteil ist aber nicht der Fall. Wenn Display so geändert würde, dass es ein House abzeptiert, könnten Sie kein Asset übergeben.

Casting und Referenzumwandlungen

Eine Objektreferenz kann

- implizit zu einer Basisklassenreferenz nach oben gecastet werden (*Upcast*) oder
- explizit zu einer Subklassenreferenz nach unten gecastet werden (*Downcast*).

Ein Upcasting oder Downcasting zwischen kompatiblen Referenztypen führt zu einer *Referenzumwandlung*: Es wird eine neue Referenz erstellt, die auf dasselbe Objekt zeigt. Ein Upcast ist immer erfolgreich; ein Downcast ist nur erfolgreich, wenn das Objekt den richtigen Typ hat.

Upcasting

Eine Upcast-Operation erzeugt aus einer Subklassenreferenz eine Basisklassenreferenz:

```
Stock msft = new Stock();    // aus letztem Beispiel

Asset a = msft;              // Upcast
```

Nach dem Upcast verweist Variable a immer noch auf dasselbe Stock-Objekt wie die Variable msft. Das referenzierte Objekt ist nicht verändert oder umgewandelt worden:

```
Console.WriteLine (a == msft);      // True
```

Auch wenn a und msft auf dasselbe Objekt verweisen, hat a eine eingeschränktere Sicht darauf:

```
Console.WriteLine (a.Name);         // Okay

Console.WriteLine (a.SharesOwned);  // Fehler
```

Die letzte Zeile führt zu einem Kompilierungsfehler, da die Variable a vom Typ Asset ist, obwohl sie auf ein Objekt vom Typ Stock verweist. Um an dessen Feld SharesOwned zu gelangen, müssen Sie das Asset zu einem Stock »downcasten«.

Downcasting

Eine Downcast-Operation erzeugt eine Subklassenreferenz aus einer

Basisklassenreferenz:

```
Stock msft = new Stock();
Asset a = msft;                  // Upcast
Stock s = (Stock)a;              // Downcast
Console.WriteLine (s.SharesOwned);    // <Kein Fehler>
Console.WriteLine (s == a);           // True
Console.WriteLine (s == msft);        // True
```

Wie bei einem Upcast sind nur die Referenzen betroffen, nicht das zugrunde liegende Objekt. Ein Downcast benötigt einen expliziten Cast, da er zur Laufzeit potenziell fehlschlagen kann:

```
House h = new House( );
Asset a = h;        // Upcast ist immer erfolgreich
Stock s = (Stock)a; // Downcast schlägt fehl: a ist kein Stock
```

Wenn ein Downcast fehlschlägt, wird eine InvalidCastException geworfen. Dies ist ein Beispiel für *Typprüfung zur Laufzeit* (siehe »Statische und Laufzeit-Typprüfung« auf Seite 90).

Der as-Operator

Der Operator as führt einen Downcast durch, der null als Ergebnis hat, wenn der Downcast fehlschlägt (anstatt eine Exception zu werfen):

```
Asset a = new Asset();
Stock s = a as Stock;   // s ist null; keine Ausnahme
```

Das ist praktisch, wenn Sie später prüfen werden, ob das Ergebnis null ist:

```
if (s != null) Console.WriteLine (s.SharesOwned);
```

Der as-Operator kann keine angepassten Umwandlungen (siehe »Überladen von Operatoren« auf Seite 188) und keine numerischen Umwandlungen durchführen.

Der is-Operator

Der is-Operator prüft, ob eine Referenzumwandlung erfolgreich wäre – mit anderen Worten, ob ein Objekt von einer bestimmten Klasse abgeleitet ist (oder ein Interface implementiert). Er wird häufig genutzt, um vor einem Downcast eine Prüfung vorzunehmen:

```
if (a is Stock) Console.Write (((Stock)a).SharesOwned);
```

Der is-Operator gibt auch true zurück, wenn eine Unboxing-Umwandlung erfolgreich wäre (siehe »Der Typ object« auf Seite 88). Benutzerdefinierte und numerische Umwandlungen werden allerdings nicht berücksichtigt.

Seit C# 7 können Sie im Rahmen des is-Operators eine Variable einführen:

```
if (a is Stock s)
    Console.WriteLine (s.SharesOwned);
```

Die so geschaffene Variable steht zur sofortigen Verwendung zur Verfügung und bleibt auch außerhalb des is-Ausdrucks im Scope:

```
f (a is Stock s && s.SharesOwned > 100000)
    Console.WriteLine ("Reich");
else
    s = new Stock(); // s ist im Scope

Console.WriteLine (s.SharesOwned); // immer noch im Scope
```

Virtuelle Funktions-Member

Eine Funktion, die als virtual gekennzeichnet ist, kann von Subklassen *überschrieben* werden, die eine speziellere Implementierung bereitstellen wollen. Methoden, Eigenschaften, Indexer und Events können als virtual deklariert werden:

```
public class Asset
{
    public string Name;
    public virtual decimal Liability => 0; }
}
```

Liability => 0 ist eine Kurzform für { get { return 0; } } (siehe dazu »Expression-bodied Eigenschaften« auf Seite 72). Eine Subklasse überschreibt eine virtuelle Methode, indem sie den Modifikator override angibt:

```
public class House : Asset
{
    public decimal Mortgage;

    public override decimal Liability => Mortgage;
}
```

Standardmäßig liefert die Liability eines Asset den Wert 0. Ein Stock muss hier nicht spezieller tätig werden. Aber das House spezialisiert die Eigenschaft Liability so, dass der Wert der Mortgage zurückgegeben wird:

```
House mansion = new House { Name="Mansion",
                            Mortgage=250000 };
Asset a = mansion;
```

```
Console.WriteLine (mansion.Liability);  // 250000

Console.WriteLine (a.Liability);        // 250000
```

Signaturen, Rückgabewerte und Sichtbarkeit der virtuellen und überschriebenen Methoden müssen identisch sein. Eine überschriebene Methode kann ihre Implementierung in der Basisklasse mithilfe des Schlüsselworts base ansprechen (siehe »Das Schlüsselwort base« auf Seite 86).

Abstrakte Klassen und abstrakte Member

Eine Klasse, die als abstrakt deklariert wurde, kann man niemals instanziieren. Stattdessen können nur ihre konkreten Subklassen instanziiert werden.

Abstrakte Klassen können *abstrakte Member* deklarieren, die sich wie virtuelle Member verhalten, aber keine Standardimplementierung bereitstellen. Diese Implementierung muss in der Subklasse vorgenommen werden, sofern die Subklasse nicht auch als abstrakt deklariert wurde:

```
public abstract class Asset
{
    // Beachten Sie die leere Implementierung. public
    abstract decimal NetValue { get; }
}
```

Subklassen überschreiben abstrakte Member ebenso, wie sie virtuelle Member überschreiben.

Verbergen geerbter Member

Eine Basisklasse und eine Subklasse können identische Member definieren:

```
public class A      { public int Counter = 1; }

public class B : A  { public int Counter = 2; }
```

Das Feld Counter in Klasse B verbirgt das Feld Counter aus Klasse A. Normalerweise geschieht das unabsichtlich, wenn der Basisklasse ein Member hinzugefügt wird, *nachdem* dem Subtyp ein identisches Member hinzugefügt wurde. Aus diesem Grund erzeugt der Compiler eine Warnung und löst die Mehrdeutigkeit wie folgt auf:

- Referenzen auf A (beim Kompilieren) werden mit A.Counter verbunden.
- Referenzen auf B (beim Kompilieren) werden mit B.Counter verbunden.

Gelegentlich wollen Sie ein Member mit Absicht verbergen. In solchen Fällen können Sie dem Member in der Subklasse den Modifikator new mitgeben. Dieser sorgt lediglich dafür, dass die Compilerwarnung unterdrückt wird, die es sonst hier gäbe:

```
public class A     { public     int Counter = 1; }

public class B : A { public new int Counter = 2; }
```

Der Modifikator new teilt dem Compiler (und den weiteren Programmierern) mit, dass das doppelte Member kein Versehen von Ihnen ist.

Funktionen und Klassen versiegeln

Ein überschriebenes Funktions-Member kann seine Implementierung mithilfe des Schlüsselworts sealed *versiegeln*, um zu verhindern, dass es durch weitere Subklassen überschrieben wird. In unserem Beispiel für virtuelle Funktions-Member könnten wir die Implementierung von Liability in House versiegeln und damit verhindern, dass eine Klasse, die von House abgeleitet wird, Liability überschreibt:

```
public sealed override decimal Liability { get { ... } }
```

Sie können auch die Klasse selbst versiegeln, wodurch Sie implizit alle virtuellen Funktionen versiegeln, indem Sie den Modifikator sealed für die Klasse nutzen.

Das Schlüsselwort base

Das Schlüsselwort base entspricht dem Schlüsselwort this. Dabei dient es zwei Zwecken: dem Zugriff auf ein überschriebenes Funktions-Member aus der Subklasse und dem Aufruf eines Basisklassenkonstruktors (siehe nächsten Abschnitt).

In diesem Beispiel nutzt House das Schlüsselwort base, um auf die Asset-Implementierung von Liability zuzugreifen:

```
public class House : Asset
{
    ...
    public override decimal Liability
        => base.Liability + Mortgage;
    }
}
```

Wir greifen mit dem Schlüsselwort base *nichtvirtuell* auf die Liability-Eigenschaft von Asset zu. Das bedeutet, dass wir immer auf die Asset-Version dieser Eigenschaft zugreifen – unabhängig davon, was der tatsächliche Laufzeittyp der Instanz ist.

Der gleiche Ansatz funktioniert, wenn Liability *verborgen* ist und nicht *überschrieben*. (Sie können auch auf verborgene Member zugreifen, indem Sie auf die Basisklasse casten, bevor Sie die Funktion aufrufen.)

Konstruktoren und Vererbung

Eine Subklasse muss ihre eigenen Konstruktoren definieren. Würden wir zum Beispiel Baseclass und Subclass so definieren:

```
public class Baseclass
{
    public int X;
    public Baseclass () { }
    public Baseclass (int x) { this.X = x; }
```

```
}

public class Subclass : Baseclass { }
```

wäre das Folgende ungültig:

```
Subclass s = new Subclass (123);
```

Subclass muss daher alle Konstruktoren neu deklarieren, die sie bereitstellen will. Dabei kann sie beliebige Basisklassenkonstruktoren mit dem Schlüsselwort base aufrufen:

```
public class Subclass : Baseclass
{
    public Subclass (int x) : base (x) { ... }
}
```

Das Schlüsselwort base funktioniert wie das Schlüsselwort this, nur dass es einen Konstruktor in der Basisklasse aufruft. Basisklassenkonstruktoren werden immer zuerst ausgeführt – damit ist sichergestellt, dass die *Basisinitialisierung* vor der *spezialisiertenInitialisierung* ausgeführt wird.

Verwendet ein Konstruktor in einer Subklasse das Schlüsselwort base nicht, wird der *parameterlose* Konstruktor der Basisklasse implizit aufgerufen (wenn die Basisklasse keinen erreichbaren parameterlosen Konstruktor hat, meldet der Compiler einen Fehler).

Reihenfolge von Konstruktor- und Feldinitialisierung

Wenn ein Objekt instanziiert wird, geschieht die Initialisierung in dieser Reihenfolge:

1. Von der Subklasse zur Basisklasse:
    a. Felder werden initialisiert.
    b. Argumente für den Basisklassenkonstruktor werden ausgewertet.
2. Von der Basisklasse zur Subklasse:
    a. Konstruktorcode wird ausgeführt.

Auflösen beim Überladen

Die Vererbung hat einen interessanten Einfluss auf das Überladen von Methoden. Schauen Sie sich die folgende überladene Methode an:

```
static void Foo (Asset a) { }

static void Foo (House h) { }
```

Wird eine überladene Methode aufgerufen, hat der spezifischste Typ Vorrang:

```
House h = new House (...);

Foo(h);              // Ruft Foo(House) auf.
```

Die aufzurufende Version der überladenen Methode wird statisch bestimmt (beim Kompilieren) und nicht zur Laufzeit. Der folgende Code ruft Foo(Asset) auf, auch wenn der Typ zur Laufzeit ein House ist:

```
Asset a = new House (...);

Foo(a);              // Ruft Foo(Asset) auf.
```

Wenn Sie Asset auf dynamic casten (siehe »Die dynamische Bindung« auf Seite 179), wird die Entscheidung darüber, welche Überladung aufgerufen wird, bis zur Laufzeit aufgeschoben und basiert dann auf dem tatsächlichen Typ des Objekts.

Der Typ object

object (System.Object) ist die Ausgangsbasisklasse für alle Typen. Jeder Typ kann per Upcast in ein object umgewandelt werden.

Um zu zeigen, wie nützlich das sein kann, stellen Sie sich einen allgemeinen *Stack* vor. Ein Stack ist eine Datenstruktur, die auf dem Prinzip »Last in, first out« (kurz LIFO) basiert. Ein Stack hat zwei Operationen: per Push ein Objekt auf den Stack schieben und per Pop eines wieder herunterholen. So sieht eine einfache Implementierung aus, die bis zu zehn Objekte verwalten kann:

```
public class Stack
{
    int position;
    object[] data = new object[10];
    public void Push (object o) { data[position++] = o; }
    public object Pop() { return data[--position]; }
}
```

Da Stack mit dem Typ object arbeitet, können wir Instanzen von *jedem beliebigen Typ* per Push und Pop auf den Stack bringen und wieder von ihm herunterbringen:

```
Stack stack = new Stack();
stack.Push ("Würstchen");
string s = (string) stack.Pop();   // Cast nötig
Console.WriteLine (s);             // Würstchen
```

object ist ein Referenztyp, da es eine Klasse ist. Trotzdem können Werttypen wie int auf object gecastet werden. Das ermöglicht die CLR mit einigen speziellen

Operationen, die die grundlegenden Unterschiede zwischen Wert- und Referenztypen überbrücken. Man bezeichnet diese Schritte als *Boxing* und *Unboxing*.

Im Abschnitt »Generics« auf Seite 102 werden wir beschreiben, wie wir unsere Stack-Klasse verbessern können, damit sie besser mit Stacks von Elementen desselben Typs umgeht.

Boxing und Unboxing

Boxing ist das Casten einer Instanz eines Werttyps auf eine Instanz eines Referenztyps. Der Referenztyp kann entweder von der Klasse object oder ein Interface sein (siehe »Interfaces« auf Seite 95). In diesem Beispiel *boxen* wir ein int in ein object:

```
int x = 9;

object obj = x;        // int verpacken
```

Beim Unboxing wird die Operation umgekehrt, indem das Objekt zurück in den ursprünglichen Werttyp gewandelt wird:

```
int y = (int)obj;      // int auspacken
```

Unboxing erfordert einen expliziten Cast. Die Laufzeitumgebung prüft, ob der angegebene Werttyp dem tatsächlichen Objekttyp entspricht, und wirft bei Nichtübereinstimmung eine InvalidCastException. So wird zum Beispiel hier eine Exception geworfen, weil long nicht genau zu int passt:

```
object obj = 9;       // 9 wird als int erkannt

long x = (long) obj;  // InvalidCastException
```

Der folgende Code wird aber erfolgreich sein:

```
object obj = 9;
```

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

Und dieser auch:

```
object obj = 3.5;       // erschlossener Typ von 3.5 ist double

int x = (int) (double) obj;     // x ist jetzt 3
```

Im letzten Beispiel führt (double) ein Unboxing durch und (int) eine numerische Umwandlung.

Beim Boxing wird die Werttyp-Instanz in das neue Objekt kopiert, und beim Unboxing wird der Inhalt des Objekts wieder in eine Werttyp-Instanz kopiert:

```
int i = 3;

object boxed = i;

i = 5;

Console.WriteLine (boxed);    // 3
```

Statische und Laufzeit-Typprüfung

C# prüft Typen sowohl statisch (beim Kompilieren) als auch zur Laufzeit.

Die statische Typprüfung ermöglicht dem Compiler, die Korrektheit Ihres Programms zu prüfen, ohne es auszuführen. Der folgende Code wird fehlschlagen, da der Compiler die statische Typprüfung durchführt:

```
int x = "5";
```

Die Laufzeit-Typprüfung wird von der CLR durchgeführt, wenn Sie über eine Referenzumwandlung oder Unboxing einen Downcast vornehmen:

```
object y = "5";

int z = (int) y;       // Laufzeitfehler, Downcast
```

```
                // fehlgeschlagen
```

Die Laufzeit-Typprüfung ist möglich, weil jedes Objekt auf dem Heap intern ein kleines Typ-Token mit abspeichert. Dieses Token kann durch den Aufruf der Methode GetType von object genutzt werden.

Die GetType-Methode und der typeof-Operator

Alle Typen in C# werden zur Laufzeit durch eine Instanz von System.Type repräsentiert. Ein Objekt vom Typ System.Type kann man auf zweierlei Weise erhalten: durch einen Aufruf von GetType für die Instanz oder durch den Einsatz des Operators typeof für einen Typnamen. GetType wird zur Laufzeit ausgewertet, während typeof statisch beim Kompilieren ermittelt wird.

System.Type hat Eigenschaften für den Namen des Typs, die Assembly, den Basistyp und so weiter:

```
int x = 3;

Console.Write (x.GetType().Name);          // Int32
Console.Write (typeof(int).Name);          // Int32
Console.Write (x.GetType().FullName);  // System.Int32
Console.Write (x.GetType() == typeof(int));   // True
```

System.Type stellt auch Methoden zur Verfügung, die als Brücke zum Reflection-Modell der Laufzeitumgebung dienen.

Die Member von object

Das hier sind alle Member von object:

```
public extern Type GetType();
public virtual bool Equals (object obj);
```

```
public static bool Equals (object objA, object objB);

public static bool ReferenceEquals (object objA,
                                    object objB);

public virtual int GetHashCode();

public virtual string ToString();

protected override void Finalize();

protected extern object MemberwiseClone();
```

Equals, ReferenceEquals und GetHashCode

Die Methode Equals entspricht dem Operator ==, nur dass Equals virtuell, == aber statisch ist. Das folgende Beispiel zeigt den Unterschied:

```
object x = 3;

object y = 3;

Console.WriteLine (x == y);      // False

Console.WriteLine (x.Equals (y));  // True
```

Da x und y auf den Typ object gecastet wurden, bindet der Compiler statisch gegen den Operator == von object, der die *Referenztyp*-Semantik nutzt, um zwei Instanzen zu vergleichen. (Da x und y geboxt sind, handelt es sich um zwei verschiedene Speicherplätze, die damit nicht gleich sind.) Die virtuelle Methode Equals greift auf Equals des Typs Int32 zurück, das eine *Werttyp*-Semantik beim Vergleich zweier Werte nutzt.

Die statische Methode object.Equals ruft einfach die virtuelle Methode Equals auf – nachdem sie sichergestellt hat, dass die Argumente nicht null sind.

```
object x = null, y = 3;
```

```
bool error = x.Equals (y);          // Laufzeitfehler!

bool ok = object.Equals (x, y);   // Okay (false)
```

ReferenceEquals erzwingt einen Test auf Referenztyp-Gleichheit (das ist gelegentlich bei Referenztypen nützlich, wenn der Operator == überladen wurde, um etwas anderes zu bewirken).

GetHashCode gibt einen Hashcode aus, der für den Einsatz in Hashtable-basierten Dictionaries geeignet ist, d. h. System.Collections.Generic.Dictionary und System.Collections.Hashtable.

Um die Gleichheitssemantik eines Typs anzupassen, müssen Sie mindestens Equals und GetHashCode überschreiben. Sie werden meist aber auch die Operatoren == und != überladen. Ein Beispiel dazu finden Sie in »Überladen von Operatoren« auf Seite 188.

Die ToString-Methode

Die Methode ToString liefert die Standardtextdarstellung einer Typinstanz zurück. Die Methode wird von allen eingebauten Typen überschrieben. Hier sehen Sie ein Beispiel für die Verwendung der Methode ToString des Typs int:

```
string s1 = 1.ToString();     // s1 ist "1"

string s2 = true.ToString();  // s2 ist "True"
```

Sie können die Methode ToString für selbst definierte Typen wie folgt überschreiben:

```
public override string ToString() => "Foo";
```

Structs

Ein *Struct* ähnelt einer Klasse, es gibt aber auch entscheidende Unterschiede:

- Ein Struct ist ein Werttyp, während eine Klasse ein Referenztyp ist.
- Ein Struct unterstützt keine Vererbung (außer dass es implizit von object oder genauer von System.ValueType abgeleitet ist).

Ein Struct kann alle Member nutzen, die eine Klasse haben kann, mit Ausnahme von parameterlosen Konstruktoren, Feldinitialisierern, Finalizern und virtuellen oder geschützten (protected) Membern.

Ein Struct ist passend, wenn die Werttyp-Semantik gewünscht wird. Gute Beispiele für Structs sind numerische Typen, bei denen es natürlicher ist, in einer Zuweisung einen Wert zu kopieren als eine Referenz. Da ein Struct ein Werttyp ist, muss nicht bei jeder Instanziierung ein Objekt auf dem Heap erstellt werden. Das kann sich zu einer nützlichen Einsparung summieren, wenn man viele Instanzen eines Typs erstellen muss. Beispielsweise erfordert die Erstellung eines Arrays mit einem Werttyp nur eine einzige Heap-Allozierung.

Semantik beim Erzeugen eines Struct

Beim Erzeugen eines Struct wird wie folgt vorgegangen:

- Es existiert ein parameterloser Konstruktor, den Sie nicht überschreiben können. Dieser nullt die Felder bitweise.
- Wenn Sie einen Konstruktor für ein Struct (mit Parametern) definieren, müssen Sie explizit jedes Feld zuweisen.
- Sie können keine Feldinitialisierer in einem Struct verwenden.

Zugriffsmodifikatoren

Um die Kapselung zu unterstützen, kann ein Typ oder Typ-Member seine *Sichtbarkeit* gegenüber anderen Typen und Assemblies einschränken, indem er seiner Deklaration einen von fünf *Zugriffsmodifikatoren* hinzufügt:

public
: Vollständig sichtbar. Der implizite Wert für Member eines Enum oder eines Interface.

internal
: Sichtbar nur innerhalb der Assembly, in der man sich befindet, und in befreundeten Assemblies. Standardwert für nicht verschachtelte Typen.

private
: Sichtbar nur für den umschließenden Typ. Standardwert für Member einer Klasse oder eines Struct.

protected
: Sichtbar nur für enthaltende Typen oder Subklassen.

protected internal
: Die *Vereinigung* von protected und internal. (Das ist weniger restriktiv als protected oder internal allein, da es ein Member auf zweierlei Weise besser sichtbar macht).

Im folgenden Beispiel ist Class2 auch außerhalb seiner Assembly erreichbar, Class1 nicht:

```
class Class1 {}        // Class1 ist internal (default)

public class Class2 {}
```

ClassB stellt das Feld x anderen Typen in der Assembly bereit, ClassA nicht:

```
class ClassA { int x;          } // x ist private

class ClassB { internal int x; }
```

Wenn Sie eine Basisklassenfunktion überschreiben, muss die Sichtbarkeit derjenigen

der überschriebenen Funktion entsprechen. Der Compiler verhindert jegliche inkonsistente Verwendung von Zugriffsmodifikatoren – beispielsweise kann eine Subklasse eine geringere Sichtbarkeit haben als eine Elternklasse, eine bessere Sichtbarkeit hingegen nicht.

Friend Assemblies

In komplexeren Situationen können Sie internal-Member für andere *befreundete* Assemblies sichtbar machen, indem Sie das Attribut System.Runtime.CompilerServices.InternalsVisibleTo setzen und dabei folgendermaßen den Namen der Friend Assembly angeben:

```
[assembly: InternalsVisibleTo ("Friend")]
```

Wenn die Friend Assembly mit einem Strong Name signiert ist, müssen Sie ihren vollständigen öffentlichen 160-Byte-Schlüssel angeben. Sie können diesen Schlüssel über eine LINQ-Abfrage heranziehen – ein interaktives Beispiel dafür finden Sie in LINQPads freier Beispielbibliothek für *C# 7.0 in a Nutshell*.

Beschneiden der Sichtbarkeit

Ein Typ beschneidet die Sichtbarkeit seiner deklarierten Member. Der häufigste Fall ist ein internal Typ mit public Membern:

```
class C { public void Foo() {} }
```

Die (Standard-)Sichtbarkeit internal von C beschneidet die Sichtbarkeit von Foo, wodurch Foo im Endeffekt internal wird. Ein Grund dafür, dass man Foo als public kennzeichnet, könnte im einfacheren Refactoring in dem Fall liegen, dass man C später in public ändern möchte.

Interfaces

Ein *Interface* ähnelt einer Klasse, stellt aber für ihre Member nur eine Spezifikation und keine Implementierung bereit. Ein Interface hat folgende Besonderheiten:

- Interface-Member sind *alle implizit abstrakt*. Im Gegensatz dazu kann eine Klasse sowohl abstrakte als auch konkrete Member mit Implementierungen bereitstellen.
- Eine Klasse kann *mehrere* Interfaces implementieren. Im Gegensatz dazu kann sie nur von einer *einzigen* Klasse erben. Structs können Interfaces implementieren, aber nicht von einer Klasse erben (davon abgesehen, dass sie von System.ValueType abgeleitet sind).

Eine Deklaration eines Interface ist wie eine Klassendeklaration, stellt aber für seine Member keine Implementierungen bereit, da alle seine Member implizit abstrakt sind. Diese Member werden von den Klassen und Structs implementiert, die das Interface implementieren. Ein Interface kann nur Methoden, Eigenschaften, Events und Indexer enthalten, was durchaus nicht ohne Absicht genau die Member einer Klasse sind, die abstrakt sein können.

Hier sehen Sie eine etwas vereinfachte Version des Interface IEnumerator, das in System.Collections definiert ist:

```
public interface IEnumerator
{
    bool MoveNext();
    object Current { get; }
}
```

Interface-Member sind implizit public und dürfen keine Zugriffsmodifikatoren deklarieren. Implementiert man ein Interface, muss man eine public Implementierung für alle seine Member bereitstellen:

```
internal class Countdown : IEnumerator
{
```

```
    int count = 11;

    public bool MoveNext() => count-- > 0 ;

    public object Current  => count;

}
```

Sie können ein Objekt implizit auf jedes Interface casten, das es implementiert:

```
IEnumerator e = new Countdown();

while (e.MoveNext())

    Console.Write (e.Current);     // 109876543210
```

Erweitern eines Interface

Interfaces können von anderen Interfaces abgeleitet werden:

```
public interface IUndoable            { void Undo(); }

public interface IRedoable : IUndoable { void Redo(); }
```

IRedoable »erbt« alle Member von IUndoable.

Explizite Implementierung eines Interface

Das Implementieren mehrerer Interfaces kann manchmal zu einem Konflikt zwischen den Member-Signaturen führen. Sie können solche Konflikte durch die *explizite Implementierung* eines Interface-Members auflösen, zum Beispiel so:

```
interface I1 { void Foo(); }

interface I2 { int Foo();  }


public class Widget : I1, I2
```

```
{
    public void Foo()   // implizite Implementierung
    {
        Console.Write ("Widget-Implementierung von I1.Foo");
    }

    int I2.Foo()   // explizite Implementierung von I2.Foo
    {
        Console.Write ("Widget-Implementierung von I2.Foo");
        return 42;
    }
}
```

Da sowohl I1 als auch I2 sich einander ausschließende Foo-Signaturen besitzen, implementiert Widget explizit die Methode Foo von I2. Damit können die beiden Methoden in einer Klasse gemeinsam existieren. Die einzige Möglichkeit, ein explizit implementiertes Member aufzurufen, ist ein Cast auf sein Interface:

```
Widget w = new Widget();
w.Foo();            // Widget-Implementierung von I1.Foo
((I1)w).Foo();      // Widget-Implementierung von I1.Foo
((I2)w).Foo();      // Widget-Implementierung von I2.Foo
```

Ein anderer Grund für explizit implementierte Interface-Member ist das Verbergen von Membern, die sehr spezialisiert sind und den normalen Anwender eher ablenken. So würde zum Beispiel ein Typ, der ISerializable implementiert, normalerweise vermeiden wollen, seine ISerializable-Member zu präsentieren, sofern nicht explizit

auf dieses Interface gecastet wurde.

Virtuelles Implementieren von Interface-Membern

Ein implizit implementiertes Interface-Member ist standardmäßig versiegelt. Es muss in der Basisklasse als virtual oder abstract gekennzeichnet werden, damit man es überschreiben kann. Ein Aufruf des Interface-Members sowohl über die Basisklasse als auch über das Interface ruft die Subklassenimplementierung auf.

Ein explizit implementiertes Interface-Member kann nicht als virtual gekennzeichnet und auch nicht auf die übliche Art und Weise überschrieben werden. Es kann aber *reimplementiert* werden.

Reimplementieren eines Interface in einer Subklasse

Eine Subklasse kann jedes Interface-Member reimplementieren, das schon von einer Basisklasse implementiert wurde. Durch das Reimplementieren wird eine Member-Implementierung gekapert (wenn sie über das Interface aufgerufen wurde), was unabhängig davon funktioniert, ob das Member in der Basisklasse virtual ist oder nicht.

Im folgenden Beispiel implementiert TextBox das Member IUndo.Undo explizit, daher kann es nicht als virtual gekennzeichnet werden. Um es zu »überschreiben«, muss RichTextBox die Methode Undo von IUndo reimplementieren:

```
public interface IUndoable { void Undo(); }

public class TextBox : IUndoable
{
    void IUndoable.Undo()
        => Console.WriteLine ("TextBox.Undo");
}

public class RichTextBox : TextBox, IUndoable
{
```

```
    public new void Undo()
        => Console.WriteLine ("RichTextBox.Undo");
}
```

Ein Aufruf des reimplementierten Members über das Interface sorgt für einen Aufruf der Implementierung in der Subklasse:

```
RichTextBox r = new RichTextBox();
r.Undo();                // RichTextBox.Undo
((IUndoable)r).Undo();   // RichTextBox.Undo
```

In diesem Fall wird Undo explizit implementiert. Implizit implementierte Member können auch reimplementiert werden, aber das wirkt sich nicht durchgängig aus, da ein Aufruf des Members über eine Basisklassenreferenz die Basisklassenimplementierung aufruft .

Enums

Ein Enum ist ein spezieller Werttyp, der es Ihnen ermöglicht, eine Gruppe von benannten numerischen Konstanten zu definieren:

```
public enum BorderSide { Left, Right, Top, Bottom }
```

Sie können diesen Enum-Typ wie folgt verwenden:

```
BorderSide topSide = BorderSide.Top;

bool isTop = (topSide == BorderSide.Top);   // true
```

Jedes Enum-Member hat einen zugrunde liegenden ganzzahligen Wert. Standardmäßig haben die zugrunde liegenden Werte den Typ int, und den Enum-Membern werden die Konstanten 0, 1, 2 ... (in der Deklarationsabfolge) zugewiesen. Folgendermaßen können Sie einen anderen Ganzzahltypen angeben:

```
public enum BorderSide : byte { Left,Right,Top,Bottom }
```

Sie können auch für jedes Member einen expliziten ganzzahligen Wert festlegen:

```
public enum BorderSide : byte

  { Left=1, Right=2, Top=10, Bottom=11 }
```

Der Compiler ermöglicht Ihnen ebenfalls, nur ein paar der Enum-Member explizit mit einer Zahl zu versehen. Die so nicht ausgezeichneten Member werden weiterhin ausgehend vom letzten expliziten Wert durchnummeriert. Das vorige Beispiel ist also äquivalent zu dem hier:

```
public enum BorderSide : byte

  { Left=1, Right, Top=10, Bottom }
```

Enum-Konvertierungen

Sie können eine enum-Instanz mit einem expliziten Cast auf ihren zugrunde liegenden ganzzahligen Wert und wieder zurück umwandeln:

```
int i = (int) BorderSide.Left;

BorderSide side = (BorderSide) i;

bool leftOrRight = (int) side <= 2;
```

Auch ist es möglich, explizit einen Enum-Typ in einen anderen zu casten. Die Übersetzung nutzt dann die zugrunde liegenden ganzzahligen Werte für das Member.

Das numerische Literal 0 erhält eine spezielle Behandlung und erfordert keinen expliziten Cast:

```
BorderSide b = 0;   // kein Cast erforderlich

if (b == 0) ...
```

In diesem speziellen Beispiel hat BorderSide kein Member mit dem ganzzahligen Wert 0. Das führt nicht zu einem Fehler: Einer der Haken an Enums ist, dass der Compiler und die CLR die Zuweisung von Werten, die außerhalb des Bereichs der Member liegen, nicht verhindern:

```
BorderSide b = (BorderSide) 12345;

Console.WriteLine (b);              // 12345
```

Flag-Enums

Sie können Enum-Member kombinieren. Um Mehrdeutigkeiten zu vermeiden, müssen den Membern eines kombinierbaren enum explizit Werte zugewiesen werden, üblicherweise Potenzen von zwei:

```
[Flags]
```

```
public enum BorderSides

  { None=0, Left=1, Right=2, Top=4, Bottom=8 }
```

Konventionsgemäß erhalten kombinierbare Enums einen Namen im Plural statt im Singular. Um mit kombinierten Enum-Werten zu arbeiten, verwenden Sie Operatoren, die bitweise arbeiten, zum Beispiel | und &. Diese Operatoren arbeiten mit den zugrunde liegenden ganzzahligen Werten:

```
BorderSides leftRight =

    BorderSides.Left | BorderSides.Right;


if ((leftRight & BorderSides.Left) != 0)

    Console.WriteLine ("Enthält Left");    // Enthält Left


string formatted = leftRight.ToString(); // "Left, Right"


BorderSides s = BorderSides.Left;

s |= BorderSides.Right;

Console.WriteLine (s == leftRight);      // True
```

Das Attribut Flags sollte auf kombinierbare Enum-Typen angewendet werden. Tun Sie das nicht, liefert der Aufruf von ToString auf einer enum-Instanz eine Zahl statt einer Folge von Namen.

Praktisch ist es, in eine Enum-Deklaration auch Kombinations-Member aufzunehmen:

```
[Flags] public enum BorderSides

{

    None=0,

    Left=1, Right=2, Top=4, Bottom=8,
```

```
    LeftRight = Left | Right,

    TopBottom = Top | Bottom,

    All = LeftRight | TopBottom

}
```

Enum-Operatoren

Folgende Operatoren arbeiten mit Enums:

```
=  == != < >  <= >= + - ^ & | ~

+= -= ++ - sizeof
```

Die bitweise arbeitenden, arithmetischen und Vergleichsoperatoren liefern das Ergebnis einer Verarbeitung der zugrunde liegenden ganzzahligen Werte zurück. Addition und Subtraktion sind zwischen einem enum und einem Integer-Typ erlaubt, aber nicht zwischen zwei enums .

Eingebettete Typen

Ein *eingebetteter Typ* wird im Geltungsbereich eines anderen Typs deklariert:

```
public class TopLevel
{
    public class Nested { }             // eingebettete Klasse
    public enum Color { Red, Blue, Tan }  // eingebettetes Enum
}
```

Ein eingebetteter Typ hat folgende Merkmale:

- Er kann auf die privaten Member des umhüllenden Typs und alles andere zugreifen, auf das der umhüllende Typ Zugriff hat.
- Er kann aus allen möglichen Zugriffsmodifikatoren auswählen und muss sich nicht auf public und internal beschränken.
- Die Standardsichtbarkeit für einen eingebetteten Typ ist private und nicht internal.
- Der Zugriff auf einen eingebetteten Typ von außerhalb des umhüllenden Typs erfordert eine Qualifikation durch den Namen des umhüllenden Typs (wie beim Zugriff auf statische Member).

Um zum Beispiel von außerhalb unserer Klasse TopLevel auf Color. Red zugreifen zu können, müssten wir Folgendes schreiben:

```
TopLevel.Color color = TopLevel.Color.Red;
```

Alle Typen können eingebettet werden, allerdings kann nur in Klassen und Structs eingebettet werden.

Generics

C# verfügt über zwei separate Mechanismen, um Code zu schreiben, der mit verschiedenen Typen verwendbar ist: *Vererbung* und *Generics*. Während bei der Vererbung die Wiederverwendbarkeit durch einen Basistyp ausgedrückt wird, geschieht das bei Generics durch ein »Template«, das Typen als »Platzhalter« enthält. Generics können im Vergleich zur Vererbung die *Typsicherheit erhöhen* und *für weniger Casting und Boxing sorgen*.

Generische Typen

Ein *generischer Typ* deklariert *Typparameter* – Platzhaltertypen, die vom Anwender des generischen Typs gefüllt werden, indem er die *Typargumente* bereitstellt. Hier wurde ein generischer Typ Stack<T> entworfen, der Instanzen vom Typ T auf einem Stack verwalten soll. Stack<T> deklariert einen einzelnen Typparameter T:

```
public class Stack<T>
{
    int position;
    T[] data = new T[100];
    public void Push (T obj) => data[position++] = obj;
    public T Pop()          => data[--position];
}
```

Wir können Stack<T> so nutzen:

```
var stack = new Stack<int>();
stack.Push(5);
stack.Push(10);
```

```
int x = stack.Pop();       // x ist 10

int y = stack.Pop();       // y ist 5
```

Beachten Sie, dass in den beiden letzten Zeilen keine Downcasts erforderlich sind, was das Risiko eines Laufzeitfehlers und den Overhead der Boxing/Unboxing-Operationen vermeidet. Das macht unseren generischen Stack einem nichtgenerischen Stack überlegen, der object anstelle von T nutzt (ein Beispiel finden Sie unter »Der Typ object« auf Seite 88).

Stack<int> gibt für den Typparameter T das Typargument int vor, wodurch implizit ein Typ erstellt wird (die Synthese geschieht zur Laufzeit). Stack<int> hat im Endeffekt folgende Definition (die Ersetzungen sind hervorgehoben, und der Klassenname wurde durch ### ersetzt, um Verwirrung zu vermeiden):

```
public class ###
{
   int position;
   int[] data;
   public void Push(int obj) => data[position++] = obj;
   public int Pop()          => data[--position];
}
```

Technisch ausgedrückt, ist Stack<T> ein *offener Typ*, Stack<int> dagegen ein *geschlossener Typ*. Zur Laufzeit sind alle Instanzen generischer Typen geschlossen – ihre Typplatzhalter sind gefüllt.

Generische Methoden

Eine *generische Methode* deklariert Typparameter innerhalb ihrer Signatur. Mit generischen Methoden können viele grundlegende Algorithmen sehr allgemein implementiert werden. Hier sehen Sie eine generische Methode, die zwei Werte eines beliebigen Typs T vertauscht:

```
static void Swap<T> (ref T a, ref T b)
{
    T temp = a; a = b; b = temp;
}
```

Swap<T> kann wie folgt verwendet werden:

```
int x = 5, y = 10;
Swap (ref x, ref y);
```

Im Allgemeinen ist es nicht notwendig, Typargumente an eine generische Methode zu übergeben, da der Compiler den Typ implizit ermitteln kann. Würde das zu einer Mehrdeutigkeit führen, können generische Methoden mit dem Typargument aufgerufen werden:

```
Swap<int> (ref x, ref y);
```

Innerhalb eines generischen Typs ist eine Methode so lange nichtgenerisch, wie sie nicht selbst Typparameter *einführt* (mit den spitzen Klammern). In unserem generischen Stack nutzt die Methode Pop nur den schon im Typ bestehenden Typparameter T und ist daher nicht als generische Methode klassifiziert.

Methoden und Typen sind die einzigen Konstrukte, die Typparameter einführen können. Eigenschaften, Indexer, Events, Felder, Operatoren und so weiter können keine Typparameter deklarieren, auch wenn sie die Typparameter verwenden können, die vom umhüllenden Typ deklariert wurden. In unserem Beispiel mit dem generischen Stack könnten wir zum Beispiel einen Indexer schreiben, der ein generisches Element zurückgibt:

```
public T this [int index] { get { return data[index]; } }
```

Auch Konstruktoren können nur die vorhandenen Typparameter nutzen, aber nicht selbst welche einführen.

Deklarieren generischer Parameter

Typparameter können bei der Deklaration von Klassen, Structs, Interfaces, Delegates (siehe »Delegates« auf Seite 111) und Methoden eingeführt werden. Ein generischer Typ bzw. eine generische Methode kann mehrere Typparameter haben:

```
class Dictionary<TKey, TValue> {...}
```

Die Instanziierung erfolgt folgendermaßen:

```
var myDic = new Dictionary<int,string>();
```

Generische Typnamen und Methodennamen können überladen werden, solange sich die Anzahl der Typparameter unterscheidet. So kommen zum Beispiel die folgenden drei Typnamen nicht miteinander in Konflikt:

```
class A {}

class A<T> {}

class A<T1,T2> {}
```

Es ist üblich, bei generischen Typen und Methoden mit einem einzelnen Typparameter diesen Parameter `T` zu nennen, solange der Sinn des Parameters klar ist. Bei mehreren Typparametern beginnt jeder Parameter mit `T`, hat aber einen stärker beschreibenden Namen.

typeof und ungebundene generische Typen

Zur Laufzeit gibt es keine offenen generischen Typen: Offene generische Typen werden bei der Kompilation geschlossen. Aber es kann zur Laufzeit *ungebundene* generische Typen geben – nur als `Type`-Objekt. Einen ungebundenen generischen Typ können Sie in C# nur mit dem `typeof`-Operator angeben:

```
class A<T> {}
```

```
class A<T1,T2> {}

...

Type a1 = typeof (A<>);   // ungebundener Typ

Type a2 = typeof (A<,>);  // zeigt 2 Typargumente an

Console.Write (a2.GetGenericArguments().Count());  // 2
```

Sie können den typeof-Operator nutzen, um einen geschlossenen Typ anzugeben:

```
Type a3 = typeof (A<int,int>);
```

oder einen offenen Typ (der zur Laufzeit geschlossen ist):

```
class B<T> { void X() { Type t = typeof (T); } }
```

Der generische Wert default

Das Schlüsselwort default kann genutzt werden, um den Standardwert für einen angegebenen generischen Typparameter zu erhalten. Der Standardwert für einen Referenztyp ist null, der für Werttypen ist das Ergebnis eines bitweisen Löschens der Felder des Typs:

```
static void Zap<T> (T[] array)

{

    for (int i = 0; i < array.Length; i++)

        array[i] = default(T);

}
```

Generische Constraints

Standardmäßig kann ein Typparameter durch jeden beliebigen Typ ersetzt werden. *Constraints* können auf einen Typparameter angewandt werden, um spezifischere Typargumente zu verlangen. Es gibt sechs Arten von Constraints:

```
where T : Basisklasse  // Basisklassen-Constraint

where T : Interface    // Interface-Constraint

where T : class        // Referenztyp-Constraint

where T : struct       // Werttyp-Constraint

where T : new()        // parameterloser Konstruktor-
                                    // Constraint

where U : T            // Typ-Constraint
```

Im folgenden Beispiel verlangt GenericClass<T,U>, dass T von SomeClass abgeleitet ist (oder der Klasse selbst entspricht) und Interface1 implementiert, und verlangt außerdem, dass U einen parameterlosen Konstruktor anbietet:

```
class     SomeClass {}

interface Interface1 {}


class GenericClass<T,U> where T : SomeClass, Interface1

                                    where U : new()

{ ... }
```

Constraints können überall dort angewendet werden, wo generische Parameter definiert sind, sowohl in Methoden als auch in Typdefinitionen.

Ein *Basisklassen-Constraint* verlangt, dass der Typparameter eine Subklasse einer bestimmten Klasse sein muss (oder die Klasse selbst); ein *Interface-Constraint* fordert, dass der Typparameter dieses Interface implementieren muss. Diese Constraints

gestatten es, dass Instanzen des Typparameters implizit in diese Klasse oder dieses Interface umgewandelt werden.

Der *class-Constraint* und der *struct-Constraint* erfordern, dass T ein Referenztyp bzw. ein (nicht nullbarer) Werttyp ist. Der *parameterlose Konstruktor-Constraint* erfordert, dass T einen öffentlichen parameterlosen Konstruktor hat, und erlaubt Ihnen, new() auf T aufzurufen:

```
static void Initialize<T> (T[] array) where T : new()
{
    for (int i = 0; i < array.Length; i++)
        array[i] = new T();
}
```

Der *Typ-Constraint* verlangt, dass der eine Typparameter von einem anderen Typparameter abgeleitet ist bzw. ihm entspricht.

Erstellen abgeleiteter Klassen von generischen Typen

Von einer generischen Klasse können Klassen genauso abgeleitet werden wie von einer nichtgenerischen Klasse. Die Subklasse kann den Typparameter der Basisklasse offen lassen:

```
class Stack<T>              {...}
class SpecialStack<T> : Stack<T> {...}
```

Es ist aber auch möglich, den generischen Typparameter durch einen konkreten Typ zu schließen:

```
class IntStack : Stack<int>  {...}
```

Ein Subtyp kann auch eigene, neue generische Argumente mitbringen:

```
class List<T>               {...}
```

```
class KeyedList<T,TKey> : List<T> {...}
```

Selbstreferenzielle generische Deklarationen

Ein Typ kann *sich selbst* als konkreten Typ nennen, wenn er ein Typargument schließt:

```
public interface IEquatable<T> { bool Equals (T obj); }

public class Balloon : IEquatable<Balloon>
{
    public bool Equals (Balloon b) { ... }
}
```

Folgendes ist ebenfalls zulässig:

```
class Foo<T> where T : IComparable<T> { ... }
class Bar<T> where T : Bar<T> { ... }
```

Statische Daten

Statische Daten sind für jeden geschlossenen Typ getrennt vorhanden:

```
class Bob<T> { public static int Count; }
...
Console.WriteLine (++Bob<int>.Count);     // 1
Console.WriteLine (++Bob<int>.Count);     // 2
Console.WriteLine (++Bob<string>.Count);  // 1
```

```
Console.WriteLine (++Bob<object>.Count);  // 1
```

Kovarianz

Kovarianz und Kontravarianz sind fortgeschrittene Konzepte. Sie wurden in C# eingeführt, damit sich generische Interfaces und Generics (insbesondere die, die im .NET Framework definiert werden, wie IEnumerable<T>) eher so verhalten, wie Sie es erwarten würden. Das können Sie nutzen, ohne dass Sie die Einzelheiten hinter Kovarianz und Kontravarianz verstehen müssen.

Angenommen, A kann in B umgewandelt werden, dann hat X einen kovarianten Typparameter, wenn X<A> in X<B> umgewandelt werden kann.

(Im Sinne dessen, was C# unter Varianz versteht, bedeutet »umwandelbar« umwandelbar über eine *implizite Referenzumwandlung* – d. h., A *ist eine Subklasse von* B oder A *implementiert* B. Numerische Konvertierungen, Boxing und benutzerdefinierte Konvertierungen sind nicht eingeschlossen.)

Beispielsweise hat Typ IFoo<T> eine Kovariante T, wenn Folgendes zulässig ist:

```
IFoo<string> s = ...;
IFoo<object> b = s;
```

Seit C# 4.0 gestatten Interfaces (und Delegates) Kovarianz für Typparameter. Nehmen Sie zu Illustrationszwecken an, dass die Stack<T>-Klasse, die wir am Anfang dieses Abschnitts geschrieben haben, das folgende Interface implementiert:

```
public interface IPoppable<out T> { T Pop(); }
```

Der out-Modifikator auf T zeigt an, dass T nur auf Ausgabeseite genutzt wird (also als Rückgabetyp von Methoden). Er markiert das Interface als *kovariant* und gestattet uns dieses:

```
// Vorausgesetzt, Bear ist eine Subklasse von Animal:
var bears = new Stack<Bear>();
```

```
bears.Push (new Bear());

// Da bears IPoppable<Bear> implementiert,
// können wir es in IPoppable<Animal>: umwandeln.
IPoppable<Animal> animals = bears; // Erlaubt
Animal a = animals.Pop();
```

Der Cast von bears auf animals wird vom Compiler gestattet, weil der Typparameter des Interface kovariant ist.

Die Interfaces IEnumerator<T> und IEnumerable<T> (siehe »Enumeration und Iteratoren« auf Seite 138) sind seit .NET Framework 4.0 mit einem kovarianten T markiert. Das gestattet Ihnen beispielsweise, IEnumerable<string> auf IEnumerable<object> zu casten.

Der Compiler meldet einen Fehler, wenn Sie einen kovarianten Typparameter auf Eingabeseite nutzen (z. B. als Parameter für eine Methode oder für eine schreibbare Eigenschaft). Der Grund für diese Einschränkung liegt darin, dass nur so zur Kompilierzeit die Typsicherheit garantiert werden kann. Beispielsweise hindert sie uns daran, diesem Interface eine Push(T)-Methode hinzuzufügen, die Benutzer missbrauchen könnten, um ein Camel-Objekt auf unser IPoppable<Animal> zu schieben (denken Sie daran, dass der zugrunde liegende Typ in unserem Beispiel ein Stack mit Bären ist). Wenn Sie eine Push(T)-Methode definieren wollen, muss T *kontravariant* sein.

C# unterstützt Kovarianz (und Kontravarianz) nur für Elemente mit *Referenzumwandlungen* – nicht für *Boxing-Umwandlungen*. Hätten Sie eine Methode geschrieben, die einen Parameter des Typs IPoppable<object> akzeptiert, könnten Sie diese mit IPoppable<string> aufrufen, aber nicht mit IPoppable<int>.

Kontravarianz

Wir haben zuvor gesehen, dass, die Möglichkeit einer impliziten Referenzumwandlung

von A in B vorausgesetzt, ein Typ X einen kovarianten Typparameter hat, wenn X<A> eine Referenzumwandlung in X<B> zulässt. Ein Typ ist *kontravariant*, wenn eine Konvertierung in die entgegengesetzte Richtung möglich ist – von X<B> in X<A>. Das wird auf Interfaces und Delegates unterstützt, wenn der Typparameter nur auf Eingabeseite erscheint, was mit dem Modifikator in angezeigt wird. Erweitern wir unser letztes Beispiel: Wenn die Klasse Stack<T> das folgende Interface implementiert:

```
public interface IPushable<in T> { void Push (T obj); }
```

können wir Folgendes tun:

```
IPushable<Animal> animals = new Stack<Animal>();
IPushable<Bear> bears = animals; // zulässig
bears.Push (new Bear());
```

Wie bei der Kovarianz meldet der Compiler einen Fehler, wenn Sie einen kontravarianten Typparameter auf Ausgabeseite nutzen (z. B. als Rückgabetyp oder als Typ einer lesbaren Eigenschaft).

Delegates

Ein Delegate verbindet einen Aufrufer einer Methode zur Laufzeit mit seiner Zielmethode. Es gibt zwei Ausprägungen bei einem Delegate: *Typ* und *Instanz*. Ein *Delegate-Typ* definiert ein Protokoll, an das sich Aufrufender und Ziel halten und das aus einer Liste von Parametertypen und einem Rückgabetyp besteht. Eine *Delegate-Instanz* ist ein Objekt, das sich auf eine oder mehrere Zielmethoden bezieht, die diesem Protokoll entsprechen.

Eine Delegate-Instanz funktioniert als Delegierter für den Aufrufenden: Der Aufrufende wendet sich an das Delegate, und das Delegate ruft die Zielmethode auf. Diese Indirektion entkoppelt den Aufrufenden von der Zielmethode.

Die Deklaration eines Delegate-Typs beginnt mit dem Schlüsselwort delegate, sieht aber ansonsten wie eine (abstrakte) Methodendeklaration aus:

```
delegate int Transformer (int x);
```

Um eine Delegate-Instanz zu erzeugen, können Sie einer Delegate-Variablen eine Methode zuweisen:

```
class Test
{
    static void Main()
    {
        Transformer t = Square;  // erstellt die Delegate-Instanz
        int result = t(3);       // ruft Delegate auf
        Console.Write (result);  // 9
    }
    static int Square (int x) => x * x;
}
```

Das Aufrufen eines Delegate ist wie das Aufrufen einer Methode (da der Zweck des Delegate ist, nur eine Indirektionsschicht einzuschieben):

```
t(3);
```

Die Anweisung Transformer t = Square ist eine Kurzform für:

```
Transformer t = new Transformer (Square);
```

Und t(3) ist eine Kurzform für:

```
t.Invoke (3);
```

Ein Delegate entspricht einem *Callback*, einem allgemeinen Begriff, der solche Konstrukte wie C-Funktionszeiger beschreibt.

Schreiben von Plug-in-Methoden mit Delegates

Einer Delegate-Variablen wird eine Methode *zur Laufzeit* zugewiesen. Das ist nützlich, wenn man Plug-in-Methoden schreiben will. In diesem Beispiel haben wir eine Hilfsmethode namens Transform, die eine Transformation auf jedes Element in einem Integer-Array anwendet. Die Methode Transform bietet einen Delegate-Parameter, um eine Plug-in-Transformation anzugeben.

```
public delegate int Transformer (int x);


class Test
{
    static void Main()
    {
        int[] values = { 1, 2, 3 };
```

```
        Transform (values, Square);
        foreach (int i in values)
            Console.Write (i + " ");    // 1 4 9
    }

    static void Transform (int[] values, Transformer t)
    {
        for (int i = 0; i < values.Length; i++)
            values[i] = t (values[i]);
    }

    static int Square (int x) => x * x;
}
```

Multicast-Delegates

Alle Delegate-Instanzen haben eine *Multicast-Fähigkeit*. Das bedeutet, dass eine Delegate-Instanz nicht nur auf eine einzelne Zielmethode verweisen kann, sondern auf eine ganze Liste davon. Die Operatoren + und += verbinden Delegate-Instanzen:

```
SomeDelegate d = SomeMethod1;
d += SomeMethod2;
```

Funktionell entspricht diese Zeile Folgendem:

```
d = d + SomeMethod2;
```

Ruft man d auf, werden nun sowohl SomeMethod1 als auch SomeMethod2

aufgerufen. Delegates werden in der Reihenfolge ausgeführt, in der sie hinzugefügt wurden.

Die Operatoren - und -= entfernen den rechten Delegate-Operanden vom linken Delegate-Operanden:

```
d -= SomeMethod1;
```

Ruft man d erneut auf, wird nur noch SomeMethod2 ausgeführt.

Ein Aufruf von + oder += auf einer Delegate-Variablen mit dem Wert null ist ebenso zulässig wie der Aufruf von -= auf einer Delegate-Variablen mit nur einem Ziel (was dazu führt, dass die Delegate-Instanz null wird).

Delegates sind *unveränderlich*. Wenn Sie += oder -= nutzen, erstellen Sie eigentlich also eine *neue* Delegate-Instanz und weisen diese der vorhandenen Variablen zu.

Wenn ein Multicast-Delegate einen Rückgabewert hat, der nicht void ist, erhält der Aufrufende den Rückgabewert der letzten ausgeführten Methode. Die vorigen Methoden werden zwar auch aufgerufen, ihre Rückgabewerte aber verworfen. In den meisten Szenarien, in denen Multicast-Delegates verwendet werden, haben diese den Rückgabetyp void, daher spielt dieses Detail dort keine Rolle.

Alle Delegate-Typen erben implizit von System.MulticastDelegate, das von System.Delegate erbt. C# wandelt die Operationen +, -, += und -= für ein Delegate in die statischen Methoden Combine und Remove der Klasse System.Delegate um.

Instanzmethoden versus statische Methoden als Ziele

Wenn einem Delegate-Objekt eine *Instanzmethode* zugewiesen wird, muss es nicht nur die Referenz auf die *Methode*, sondern auch die auf die *Instanz* festhalten, zu der die Methode gehört. Die Target-Eigenschaft der Klasse System.Delegate repräsentiert diese Instanz (und ist null, wenn das Delegate eine statische Methode referenziert).

Generische Delegate-Typen

Ein Delegate-Typ kann generische Typparameter enthalten:

```
public delegate T Transformer<T> (T arg);
```

So könnten wir diesen Delegate-Typ verwenden:

```
static double Square (double x) => x * x;

static void Main()
{
    Transformer<double> s = Square;
    Console.WriteLine (s (3.3));        // 10.89
}
```

Die Func- und Action-Delegates

Generische Delegates ermöglichen es, einen kleinen Satz von Delegates zu schreiben, die so allgemein sind, dass sie mit Methoden beliebigen Rückgabetyps und einer bliebigen Anzahl (einer vernünftigen Anzahl) von Argumenten funktionieren. Diese Delegates sind die Func- und Action-Delegates, die im Namensraum System definiert werden (die in- und out-Modifikatoren zeigen die Varianz an, die wir weiter oben betrachtet haben):

```
delegate TResult Func <out TResult> ();
delegate TResult Func <in T, out TResult> (T arg);
delegate TResult Func <in T1, in T2, out TResult>
  (T1 arg1, T2 arg2);
... und so weiter bis T16

delegate void Action ();
```

```
delegate void Action <in T> (T arg);

delegate void Action <in T1, in T2> (T1 arg1, T2 arg2);
```

... und so weiter bis T16

Diese Delegates sind äußerst allgemein. Das Transformer-Delegate in unserem letzten Beispiel kann durch ein Func-Delegate ersetzt werden, das ein Argument des Typs T akzeptiert und einen Wert des gleichen Typs liefert:

```
public static void Transform<T> (
    T[] values, Func<T,T> transformer)
{
    for (int i = 0; i < values.Length; i++)
        values[i] = transformer (values[i]);
}
```

Die einzigen praktischen Szenarien, die von diesen Delegates nicht abgedeckt werden, sind ref/out und Zeigerparameter.

Delegate-Kompatibilität

Delegate -Typen sind untereinander alle inkompatibel, selbst wenn ihre Signaturen gleich sind:

```
delegate void D1(); delegate void D2();
...
D1 d1 = Method1;
D2 d2 = d1;          // Kompilierungsfehler
```

Folgendes ist jedoch zulässig:

```
D2 d2 = new D2 (d1);
```

Delegate-Instanzen werden als gleich betrachtet, wenn sie den gleichen Typ und die gleichen Methodenziele haben. Bei Multicast-Delegates ist die Abfolge der Methodenziele relevant.

Rückgabetypvarianz

Wenn Sie eine Methode aufrufen, erhalten Sie eventuell einen Rückgabetyp, der spezifischer ist als das, wonach Sie gefragt haben. Dementsprechend kann eine Delegate-Zielmethode einen spezifischeren Typ liefern als den, der vom Delegate beschrieben wird. Das ist eine Form der *Kovarianz*, die seit C# 2.0 unterstützt wird:

```
delegate object ObjectRetriever();

...

static void Main()

{

    ObjectRetriever o = new ObjectRetriever (GetString);

    object result = o();

    Console.WriteLine (result);      // Hallo

}

static string GetString() => "Hallo";
```

Der ObjectRetriever erwartet als Rückgabewert object, aber eine object-*Subklasse* tut es auch, weil Delegate-Rückgabetypen kovariant sind.

Parametervarianz

Wenn Sie eine Methode aufrufen, können Sie Argumente mitgeben, die spezifischere Typen haben als die Parameter dieser Methode. Das ist ganz normales polymorphes Verhalten. Dementsprechend kann eine Delegate-Zielmethode weniger spezifische Parameter haben, als vom Delegate beschrieben werden. Das nennt man *Kontravarianz*:

```
delegate void StringAction (string s);

...

static void Main()

{

    StringAction sa = new StringAction (ActOnObject);

    sa ("hallo"); // gibt "hallo" aus

}

static void ActOnObject (object o) => Console.WriteLine (o);
```

Das Standard-Event-Muster ist dazu gedacht, Ihnen beim Hervorheben der Kontravarianz durch die Verwendung der gebräuchlichen Basisklasse EventArgs zu helfen. Sie können zum Beispiel eine einzelne Methode durch zwei verschiedene Delegates aufrufen lassen, wobei das eine ein MouseEventArgs und das andere ein KeyEventArgs übergibt.

Typparametervarianz bei generischen Delegates

Sie haben in »Generics« auf Seite 102 gesehen, dass Typparameter für generische Interfaces kovariant und kontravariant sein können. Seit C# 4.0 haben generische Delegates die gleichen Eigenschaften. Wenn Sie einen generischen Delegate-Typ definieren, sollten Sie deswegen Folgendes tun:

- Markieren Sie Typparameter, die nur für Rückgabewerte genutzt werden, als kovariant (out).
- Markieren Sie Typparameter, die nur für Parameter genutzt werden, als kontravariant (in).

Diese Schritte ermöglichen es, dass sich Umwandlungen natürlich verhalten, weil Vererbungsbeziehungen zwischen Typen berücksichtigt werden. Das folgende Delegate (das im Namensraum System definiert wird) ist kovariant für TResult:

```
delegate TResult Func<out TResult>();
```

Es macht das hier möglich:

```
Func<string> x = ...;

Func<object> y = x;
```

Das folgende Delegate (das im Namensraum System definiert wird) ist kontravariant für T:

```
delegate void Action<in T> (T arg);
```

Es ermöglicht das hier :

```
Action<object> x = ...;

Action<string> y = x;
```

Events

Wenn man Delegates nutzt, gibt es zwei Rollen, die sehr häufig im Spiel sind: *Broadcaster* und *Subscriber*. Der Broadcaster ist ein Typ, der ein Delegate-Feld enthält. Er entscheidet, wann ein Broadcast verschickt werden soll, indem er das Delegate aufruft. Die Subscriber sind die Zielmethodenempfänger. Ein Subscriber entscheidet, von wann bis wann er zuhört, indem er += und -= für das Delegate des Broadcasters verwendet. Ein Subscriber weiß nichts von anderen Subscribern, kommt mit ihnen aber auch nicht in Konflikt.

Events sind ein Sprachfeature, das dieses Muster formalisiert. Ein event ist ein Konstrukt, das nur genau die Untermenge an Delegate-Features bereitstellt, die für das Broadcaster/Subscriber-Modell notwendig sind. Der Hauptzweck von Events ist, Subscriber davon abzuhalten, sich untereinander ins Gehege zu kommen.

Der einfachste Weg, ein Event zu deklarieren, ist, einem Delegate-Member das Schlüsselwort event voranzustellen:

```
public class Broadcaster

{

    public event ProgressReporter Progress;

}
```

Code innerhalb des Typs Broadcaster hat vollständigen Zugriff auf Progress und kann es wie ein Delegate behandeln. Code außerhalb von Broadcaster kann nur die Operationen += und -= auf Progress anwenden.

Im folgenden Beispiel löst die Klasse Stock ihr PriceChanged-Event jedes Mal aus, wenn sich der Price von Stock ändert:

```
public delegate void PriceChangedHandler

  (decimal oldPrice, decimal newPrice);


public class Stock
```

```
{
    string symbol; decimal price;

    public Stock (string symbol) { this.symbol = symbol; }

    public event PriceChangedHandler PriceChanged;

    public decimal Price
    {
        get { return price; }
        set
        {
            if (price == value) return;
            // Event absetzen, wenn Aufrufliste nicht leer:
            if (PriceChanged != null)
                PriceChanged (price, value);
            price = value;
        }
    }
}
```

Hätten wir das Schlüsselwort event aus unserem Beispiel entfernt, sodass PriceChanged ein normales Delegate-Feld geworden wäre, würde unser Beispiel zum gleichen Ergebnis führen. Aber Stock wäre dann weniger robust, denn Subscriber könnten Folgendes tun, um sich gegenseitig zu beeinflussen:

- Ersetzen anderer Subscriber durch erneutes Zuweisen an PriceChanged (statt

den Operator += zu nutzen)

- Entfernen aller Subscriber (indem PriceChanged auf null gesetzt wird)
- Absetzen eines Broadcast an andere Subscriber, indem das Delegate aufgerufen wird

Events können virtuell, überschrieben, abstrakt oder versiegelt sein. Sie können auch statisch sein.

Standard-Event-Muster

Das .NET Framework definiert ein Standardmuster zum Schreiben von Events. Sein Zweck ist, im Framework und im Benutzercode für Konsistenz zu sorgen. So sieht das vorige Beispiel aus, wenn man es auf dieses Muster umbaut:

```
public class PriceChangedEventArgs : EventArgs
{
    public readonly decimal LastPrice, NewPrice;

    public PriceChangedEventArgs (decimal lastPrice,
                                  decimal newPrice)
    {
        LastPrice = lastPrice; NewPrice = newPrice;
    }
}

public class Stock
{
    string symbol; decimal price;

    public Stock (string symbol) { this.symbol = symbol; }
```

```
    public event EventHandler<PriceChangedEventArgs>
                            PriceChanged;

    protected virtual void OnPriceChanged(PriceChangedEventArgs e)
    {
        if (PriceChanged != null) PriceChanged (this, e);
    }

    public decimal Price
    {
        get { return price; }
        set
        {
            if (price == value) return;
            OnPriceChanged (new PriceChangedEventArgs (price,value));
            price = value;
        }
    }
}
```

Das Herzstück des Standard-Event-Musters ist System.EventArgs, eine vordefinierte .NET-Framework-Klasse ohne Member (abgesehen von der statischen Eigenschaft Empty). EventArgs ist eine Basisklasse für das Übermitteln von Informationen an ein Event. In diesem Beispiel bilden wir eine Subklasse von EventArgs, um den alten und

den neuen Preis weiterzugeben, wenn das Event PriceChanged ausgelöst wird.

Das generische Delegate System.EventHandler ist ebenfalls Teil des .NET Framework und wie folgt definiert:

```
public delegate void EventHandler<TEventArgs>
    (object source, TEventArgs e)
      where TEventArgs : EventArgs;
```

Vor C# 2.0 (als Generics hinzukamen) schrieb man stattdessen für jeden EventArgs-Typ ein eigenes Delegate zum Event-Handling:

```
delegate void PriceChangedHandler
  (object sender,
     PriceChangedEventArgs e);
```

Aus historischen Gründen verwenden die meisten Events im Framework Delegates, die auf diese Art und Weise definiert wurden.

Eine geschützte (protected) virtuelle Methode namens On-*Eventname* ist zentral für das Auslösen des Events verantwortlich. Damit können Subklassen das Event feuern (was meist erwünscht ist) und vor oder nach dem Auslösen des Events Code einfügen.

So könnten wir unsere Klasse Stock nutzen:

```
static void Main()
{
    Stock stock = new Stock ("THPW");
    stock.Price = 27.10M;
    stock.PriceChanged += stock_PriceChanged;
    stock.Price = 31.59M;
}
```

```
static void stock_PriceChanged
    (object sender, PriceChangedEventArgs e)
{
    if ((e.NewPrice - e.LastPrice) / e.LastPrice > 0.1M)
        Console.WriteLine ("Achtung, Preissteigerung von 10%!");
}
```

Bei Events, die keine zusätzlichen Informationen übermitteln, stellt das .NET Framework noch ein nichtgenerisches Delegate namens EventHandler bereit. Wir können das demonstrieren, indem wir die Klasse Stock so umschreiben, dass das Event PriceChanged ausgelöst wird, *nachdem* sich der Preis geändert hat. Das bedeutet, dass keine zusätzlichen Informationen mit dem Event übermittelt werden müssen:

```
public class Stock
{
    string symbol; decimal price;

    public Stock (string symbol) {this.symbol = symbol;}

    public event EventHandler PriceChanged;

    protected virtual void OnPriceChanged (EventArgs e)
    {
        if (PriceChanged != null) PriceChanged (this, e);
    }

    public decimal Price
```

```
    {
        get { return price; }
        set
        {
            if (price == value) return;
            price = value;
            OnPriceChanged (EventArgs.Empty);
        }
    }
}
```

Beachten Sie, dass wir hier auch die Eigenschaft EventArgs.Empty verwendet haben – dadurch ersparen wir uns das Erzeugen einer Instanz von EventArgs.

Event-Accessors

Die *Accessors* eines Events sind die Implementierungen seiner Funktionen += und -=. Standardmäßig werden Accessors *implizit* vom Compiler implementiert. Die Eventdeklaration

```
public event EventHandler PriceChanged;
```

wandelt der Compiler um in

- ein privates Delegate-Feld und
- ein öffentliches Paar Event-Accessor-Funktionen, deren Implementierungen die Operationen += und -= an das private Delegate-Feld weiterleiten.

Sie können diesen Prozess selbst in die Hand nehmen, indem Sie *explizit* Event-Accessors definieren. So sieht eine manuelle Implementierung des Events PriceChanged aus unserem vorigen Beispiel aus:

```
EventHandler priceChanged;   // privates Delegate

public event EventHandler PriceChanged

{

    add { priceChanged += value; }

    remove { priceChanged -= value; }

}
```

Funktionell entspricht dieses Beispiel der Standard-Accessor-Implementierung von C# (davon abgesehen, dass C# zusätzlich die Threadsicherheit um die Delegate-Aktualisierung herum sicherstellt). Wenn wir unsere Event-Accessors selbst definieren, weisen wir C# an, die standardmäßige Feld- und Accessor-Logik nicht zu generieren.

Bei expliziten Event-Accessors können Sie komplexere Strategien zum Speichern und Ansprechen des zugrunde liegenden Delegate anwenden. Das ist nützlich, wenn die Event-Accessors bloße Weiterleitungen für eine andere Klasse sind, die das Event ausstrahlt, oder wenn man explizit ein Interface implementiert, das ein Event deklariert :

```
public interface IFoo { event EventHandler Ev; }

class Foo : IFoo

{

    EventHandler ev;

    event EventHandler IFoo.Ev

    {

        add { ev += value; } remove { ev -= value; }

    }

}
```

Lambda-Ausdrücke

Ein *Lambda-Ausdruck* ist eine Methode ohne Namen, die anstelle einer Delegate-Instanz geschrieben wird. Der Compiler wandelt den Lambda-Ausdruck direkt um, und zwar

- in eine Delegate-Instanz oder
- in einen *Expression Tree* vom Typ Expression<TDelegate>, der den Code im Lambda-Ausdruck in einem traversierbaren Objektmodell repräsentiert. Damit können Lambda-Ausdrücke zur Laufzeit interpretiert werden.

Wenn wir den Delegate-Typ

```
delegate int Transformer (int i);
```

haben, können wir den Lambda-Ausdruck x => x * x wie folgt zuweisen und ausführen:

```
Transformer sqr = x => x * x;

Console.WriteLine (sqr(3));    // 9
```

Intern löst der Compiler Lambda-Ausdrücke dieses Typs auf, indem er eine private Methode erstellt und den Code des Ausdrucks in diese Methode kopiert.

Ein Lambda-Ausdruck hat die folgende Form:

(**Parameter**) => **Ausdruck-oder-Anweisungsblock**

Aus Gründen der Bequemlichkeit können Sie die Klammern weglassen, wenn es genau einen Parameter eines ableitbaren Typs gibt.

In unserem Beispiel gibt es einen einzelnen Parameter x, und der Ausdruck ist x * x:

```
x => x * x;
```

Jeder Parameter des Lambda-Ausdrucks entspricht einem Delegate-Parameter, und der Typ des Ausdrucks (der void sein kann) entspricht dem Rückgabetyp des Delegate.

In unserem Beispiel entspricht x dem Parameter i und der Ausdruck x * x dem Rückgabetyp int, was kompatibel zum Delegate Transformer ist.

Der Code eines Lambda-Ausdrucks kann statt eines Ausdrucks auch einen Anweisungsblock enthalten. Wir können unser Beispiel wie folgt umschreiben:

```
x => { return x * x; };
```

Lambda-Ausdrücke werden am häufigsten mit den Func- und Action-Delegates eingesetzt – meist werden Ihnen die oben formulierten Ausdrücke also in folgender Form begegnen:

```
Func<int,int> sqr = x => x * x;
```

In der Regel kann der Compiler den Typ der Lambda-Parameter aus dem Kontext *erschließen*. Wenn das nicht der Fall ist, können Sie die Typen der Parameter explizit angeben:

```
Func<int,int> sqr = (int x) => x * x;
```

Hier ist ein Beispiel für einen Ausdruck, der zwei Parameter erwartet:

```
Func<string,string,int> totalLength =
  (s1, s2) => s1.Length + s2.Length;

int total = totalLength ("hallo", "welt");  // total=10;
```

Wenn Clicked ein Event des Typs EventHandler ist, bindet folgender Code über einen Lambda-Ausdruck einen Event-Handler daran:

```
obj.Clicked += (sender,args) => Console.Write ("Click");
```

Äußere Variablen übernehmen

Ein Lambda-Ausdruck kann die lokalen Variablen und Parameter der Methode referenzieren, in der er definiert wird (*äußere Variablen*), zum Beispiel so:

```
static void Main()
{
   int factor = 2;
   Func<int, int> multiplier = n => n * factor;
   Console.WriteLine (multiplier (3));          // 6
}
```

Äußere Variablen, die von einem Lambda-Ausdruck referenziert werden, bezeichnet man als *übernommene Variablen*. Einen Lambda-Ausdruck, der Variablen einfängt, nennt man eine *Closure*. Übernommene Variablen werden ausgewertet, wenn das Delegate tatsächlich *aufgerufen* wird, nicht wenn die Variablen *übernommen* werden:

```
int factor = 2;
Func<int, int> multiplier = n => n * factor;
factor = 10;
Console.WriteLine (multiplier (3));          // 30
```

Lambda-Ausdrücke können übernommene Variablen verändern:

```
int seed = 0;
Func<int> natural = () => seed++;
Console.WriteLine (natural());          // 0
```

```
Console.WriteLine (natural());          // 1

Console.WriteLine (seed);               // 2
```

Die Lebenszeit übernommener Variablen wird auf die des Delegate ausgeweitet. Im folgenden Beispiel würde die lokale Variable seed normalerweise die Geltung verlieren, wenn die Ausführung von Natural beendet würde. Aber da seed übernommen wurde, wurde ihre Lebenszeit auf die des übernehmenden Delegate natural ausgedehnt:

```
static Func<int> Natural()
{
   int seed = 0;
   return () => seed++;      // liefert eine Closure
}

static void Main()
{
   Func<int> natural = Natural();
   Console.WriteLine (natural());      // 0
   Console.WriteLine (natural());      // 1
}
```

Variablen können auch von anonymen Methoden oder lokalen Methoden übernommen werden. Die Regeln für die übernommenen Variablen sind gleich.

Iterationsvariablen übernehmen

Wenn Sie die Iterationsvariable einer for-Schleife übernehmen, behandelt C# die Iterationsvariable so, als wäre sie außerhalb der Schleife deklariert worden. Das bedeutet, dass bei jeder Iteration dieselbe Variable übernommen wird. Das folgende Programm gibt 333 statt 012 aus:

```
Action[] actions = new Action[3];

for (int i = 0; i < 3; i++)
    actions [i] = () => Console.Write (i);

foreach (Action a in actions) a();     // 333
```

Alle Closures (fett gedruckt) übernehmen dieselbe Variable, nämlich i. (Das ist sinnvoll, wenn Sie bedenken, dass i eine Variable ist, deren Wert über die Schleifeniterationen erhalten bleibt; Sie können den Wert von i im Schleifenrumpf sogar explizit ändern, wenn Sie wollen.) Die Folge ist, dass alle Delegates, wenn sie später aufgerufen werden, den Wert von i zur Zeit des Aufrufs sehen – und der ist 3. Dass stattdessen 012 ausgegeben wird, erreichen Sie, indem Sie die Iterationsvariable einer lokalen Variablen zuweisen, die nur innerhalb der Schleife gültig ist:

```
Action[] actions = new Action[3];

for (int i = 0; i < 3; i++)
{
    int loopScopedi = i;
    actions [i] = () => Console.Write (loopScopedi);
}

foreach (Action a in actions) a();     // 012
```

Das bewirkt, dass die Closure bei jeder Iteration eine andere Variable übernimmt.

foreach-Schleifen verhielten sich früher auf gleiche Weise, aber diese Regeln haben sich inzwischen geändert. Seit C# 5.0 können Sie problemlos die Iterationsvariable einer foreach-Schleife übernehmen, ohne dass Sie dazu eine temporäre Variable benötigen .

Lambda-Ausdrücke versus lokale Methoden

Die Funktionalität der lokalen Methoden aus C# 7 (siehe »Lokale Methoden (C# 7)« auf Seite 66) und von Lambda-Ausdrücken ist teilweise gleich. Lokale Methoden sind im Vorteil, da sie Rekursion zulassen und durch sie kein Delegate spezifiziert werden muss. Durch Letzteres wird zudem eine Indirektion vermieden, und sie werden so ein kleines bisschen effizienter. Darüber hinaus kann man in ihnen auf die lokalen Variablen der umgebenden Methode zugreifen, ohne dass der Compiler diese in einer verborgenen Klasse vorhalten muss.

Aber in vielen Fällen brauchen Sie ein Delegate, vor allem beim Aufruf einer Funktion höherer Ordnung (also einer Methode mit einem Delegate-typisierten Parameter):

```
public void Foo (Func<int,bool> predicate) { ... }
```

In solchen Fällen kommen Sie um ein Delegate nicht herum, und genau dort sind Lambda-Ausdrücke im Allgemeinen auch knapper und sauberer.

Anonyme Methoden

Anonyme Methoden sind eine C# 2.0-Einrichtung, die im Wesentlichen von Lambda-Ausdrücken ersetzt wurde. Eine anonyme Methode ähnelt einem Lambda-Ausdruck, dem implizit typisierte Parameter, die Ausdruckssyntax (anonyme Methoden müssen immer einen Anweisungsblock haben) und die Fähigkeit fehlen, zu einem Ausdrucksbaum kompiliert zu werden. Um eine anonyme Methode zu schreiben, nutzen Sie das Schlüsselwort delegate, gefolgt von einer (optionalen) Parameterdeklaration und einem Methodenrumpf. So könnten wir zum Beispiel für das Delegate

```
delegate int Transformer (int i);
```

eine anonyme Funktion wie folgt schreiben und aufrufen:

```
Transformer sqr = delegate (int x) {return x * x;};
Console.WriteLine (sqr(3));         // 9
```

Die erste Zeile ist semantisch äquivalent zum folgenden Lambda-Ausdruck:

```
Transformer sqr =       (int x) => {return x * x;};
```

Oder einfacher:

```
Transformer sqr =           x => x * x;
```

Eine einzigartige Eigenschaft anonymer Methoden ist, dass Sie die Parameterdeklaration vollständig weglassen können – selbst wenn das Delegate sie erwartet. Das kann bei der Deklaration von Events mit einem leeren Standard-Handler nützlich sein:

```
public event EventHandler Clicked = delegate { };
```

Das hebt die Notwendigkeit der Nullprüfung vor dem Absetzen des Events auf.

Folgendes ist ebenfalls zulässig (beachten Sie das Fehlen der Parameter):

```
Clicked += delegate { Console.Write ("clicked"); };
```

Anonyme Methoden übernehmen äußere Variablen genau so, wie es Lambda-Ausdrücke tun .

try-Anweisungen und Exceptions

Eine try-Anweisung definiert einen Codeblock, der einer Fehlerbehandlung untersteht oder nach dem auf jeden Fall aufgeräumt werden soll. Auf den try-Block muss ein catch-Block, ein abschließender finally-Block oder beides folgen. Der finally-Block wird ausgeführt, wenn der try-Block verlassen wird (oder – falls vorhanden – der catch-Block), um Code auszuführen, der auf jeden Fall am Ende laufen soll, egal ob ein Fehler aufgetreten ist oder nicht.

Ein catch-Block hat Zugriff auf ein Exception-Objekt, das Informationen über den Fehler enthält. Sie verwenden einen catch-Block, um entweder den Fehler zu beheben oder die Exception *weiterzuwerfen*. Sie werfen dann eine Exception weiter, wenn Sie das Problem nur protokollieren oder einen neuen, übergeordneteren Exception-Typ werfen wollen.

Ein finally-Block sorgt für mehr Determinismus in Ihrem Programm, indem er immer ausgeführt wird – egal was passiert ist. Das ist nützlich, wenn Sie Aufräumarbeiten zu erledigen haben, zum Beispiel das Schließen von Netzwerkverbindungen.

Eine try-Anweisung sieht so aus:

```
try
{
    ... // Exception wird eventuell beim Ausführen dieses
            // Blocks geworfen
}
catch (ExceptionA ex)
{
    ... // kümmere dich um Exception vom Typ ExceptionA
}
catch (ExceptionB ex)
```

```
{
    ... // kümmere dich um Exception vom Typ ExceptionB
}
finally
{
    ... // Code zum Aufräumen
}
```

Betrachten Sie den folgenden Code:

```
int x = 3, y = 0;
Console.WriteLine (x / y);
```

Da y null ist, löst die Runtime eine DivideByZeroException aus und bricht damit die Ausführung des Codes ab. Das können wir verhindern, indem wir diese Exception folgendermaßen abfangen:

```
try
{
    int x = 3, y = 0;
    Console.WriteLine (x / y);
}
catch (DivideByZeroException ex)
{
    Console.Write ("y darf nicht null sein. ");
```

```
}

// Hier wird die Ausführung nach der Exception wieder

// aufgenommen ...
```

Das ist ein einfaches Beispiel zur Erläuterung des Exception-Handling. Dieses spezielle Szenario könnten wir in der Praxis erheblich besser bewältigen, indem wir explizit den Teiler prüfen, bevor wir Calc aufrufen.

Die Verarbeitung von Exceptions ist recht teuer; sie nimmt Hunderte von CPU-Zyklen in Anspruch.

Wenn eine Exception geworfen wird, führt die CLR eine Prüfung durch:

Befindet sich das Programm gerade in einer try*-Anweisung, die die Exception fangen kann?*

- Wenn das der Fall ist, wird die Ausführung an den passenden catch-Block übergeben. Konnte der catch-Block erfolgreich abgearbeitet werden, wird mit der Ausführung der nächsten Anweisung nach der try-Anweisung fortgefahren (falls vorhanden, wird vorher noch der finally-Block ausgeführt).
- Wenn das nicht der Fall ist, springt die Ausführung zum aufrufenden Code der Funktion zurück, und die Prüfung wird wiederholt (nachdem eventuell vorhandene finally-Blöcke ausgeführt wurden, die die Anweisung umschließen).

Übernimmt keine Funktion die Verantwortung für die Ausführung, wird dem Benutzer eine Fehlermeldung angezeigt, und das Programm wird beendet.

Die catch-Klausel

Eine catch-Klausel legt fest, welche Typen von Exceptions gefangen werden sollen. Diese müssen entweder vom Typ System.Exception oder einer Subklasse davon sein. Mit dem Fangen von System.Exception fängt man alle möglichen Fehler. Das ist nützlich, wenn

- Ihr Programm möglicherweise weiterlaufen kann – unabhängig vom speziellen Exception-Typ –,
- Sie vorhaben, die Exception weiterzuwerfen (eventuell nach dem Protokollieren), oder
- Ihre Fehlerbehandlung der letzte Schritt vor dem Abbrechen des Programms ist.

Normalerweise aber fangen Sie *bestimmte Exception-Typen*, um zu vermeiden, sich mit Umständen herumschlagen zu müssen, für die Ihre Fehlerbehandlung nicht ausgelegt ist (zum Beispiel eine OutOfMemoryException).

Sie können mehrere Exception-Typen mit mehreren catch-Klauseln behandeln:

```
try
{
    DoSomething();
}
catch (IndexOutOfRangeException ex) { ... }
catch (FormatException ex)          { ... }
catch (OverflowException ex)        { ... }
```

Nur eine catch-Klausel wird für eine gegebene Exception ausgeführt. Wenn Sie ein Sicherheitsnetz spannen wollen, in dem Sie auch allgemeinere Exceptions fangen (zum Beispiel System.Exception), müssen Sie die spezifischeren Handler an den Anfang stellen.

Eine Exception kann gefangen werden, ohne eine Variable anzugeben, wenn Sie nicht auf ihre Eigenschaften zugreifen müssen:

```
catch (OverflowException)  // keine Variable
  { ... }
```

Zudem können Sie sowohl die Variable als auch den Typ weglassen (und damit alle Exceptions fangen):

```
catch { ... }
```

Exception-Filter

Seit C# 6.0 können Sie in einer catch-Klausel einen *Exception-Filter* definieren,

indem Sie eine when-Klausel hinzufügen:

```
catch (WebException ex)
    when (ex.Status == WebExceptionStatus.Timeout)
{
    ...
}
```

Wird in diesem Beispiel eine WebException geworfen, wird der boolesche Ausdruck nach dem Schlüsselwort when ausgewertet. Ist das Ergebnis false, wird der fragliche catch-Block ignoriert, und die folgenden catch-Klauseln werden berücksichtigt. Mit Exception-Filtern kann es sinnvoll sein, den gleichen Exception-Typ mehrfach zu fangen:

```
catch (WebException ex) when (ex.Status == etwas)
{ ... }
catch (WebException ex) when (ex.Status == etwas_anderes)
{ ... }
```

Der boolesche Ausdruck in der when-Klausel kann Nebeneffekte haben, wie zum Beispiel eine Methode, die die Exception zu Diagnosezwecken protokolliert.

Der finally-Block

Ein finally-Block wird immer ausgeführt – egal, ob eine Exception geworfen wurde oder nicht und ob der try-Block bis zum Ende lief oder nicht. finally-Blöcke werden meist genutzt, um dort Code unterzubringen, der dem Aufräumen dienen soll.

Ein finally-Block wird ausgeführt,

- wenn ein catch-Block fertig ist,
- nachdem der try-Block durch einen Sprungbefehl verlassen wurde (zum Beispiel durch return oder goto) oder

- wenn der try-Block beendet ist.

Durch einen finally-Block kann ein Programm zuverlässiger werden. Im folgenden Beispiel wird die Datei, die wir geöffnet haben, *immer* geschlossen, egal ob

- der try-Block normal beendet wurde,
- die Ausführung vorzeitig endet, weil die Datei leer ist (EndOfStream), oder
- eine IOException beim Lesen der Datei geworfen wird.

Das kann zum Beispiel so aussehen:

```
static void ReadFile()
{
    StreamReader reader = null;  // im Namensraum System.IO
    try
    {
        reader = File.OpenText ("file.txt");
        if (reader.EndOfStream) return;
        Console.WriteLine (reader.ReadToEnd());
    }
    finally
    {
        if (reader != null) reader.Dispose();
    }
}
```

In diesem Beispiel haben wir die Datei durch den Aufruf von Dispose für den StreamReader geschlossen. Der Aufruf von Dispose für ein Objekt in einem finally-Block ist eine Standardvorgehensweise im .NET Framework und wird in C# explizit

durch die using-Anweisung unterstützt.

Die using-Anweisung

Viele Klassen kapseln nicht verwaltete (unmanaged) Ressourcen, zum Beispiel Datei-Handles, Grafik-Handles oder Datenbankverbindungen. Diese Klassen implementieren System.IDisposable, das eine einzige parameterlose Methode namens Dispose definiert, um diese Ressourcen aufzuräumen. Die Anweisung using stellt eine elegante Syntax für das Aufrufen der Dispose-Methode eines IDisposable-Objekts in einem finally-Block bereit.

Das hier

```
using (StreamReader reader = File.OpenText ("file.txt"))
{
    ...
}
```

ist exakt äquivalent zu dem:

```
{
    StreamReader reader = File.OpenText ("file.txt");
    try
    {
        ...
    }
    finally
    {
        if (reader != null) ((IDisposable)reader).Dispose();
```

```
    }
}
```

Werfen von Exceptions

Exceptions können entweder von der Laufzeitumgebung oder aus dem Anwendungscode heraus geworfen werden. Hier wirft Display eine System.ArgumentNullException:

```
static void Display (string name)
{
    if (name == null)
        throw new ArgumentNullException (nameof (name));

    Console.WriteLine (name);
}
```

Ausdrücke werfen (C# 7)

Vor C# 7 handelte es sich bei throw immer um eine Anweisung. Jetzt kann es auch als Ausdruck in Expression-bodied Funktionen auftauchen:

```
public string Foo() => throw new NotImplementedException();
```

Ein throw-Ausdruck kann ebenfalls in einem ternären bedingten Ausdruck erscheinen:

```
string ProperCase (string value) =>
    value == null ? throw new ArgumentException ("value") :
    value == "" ? "" :
```

```
char.ToUpper (value[0]) + value.Substring (1);
```

Eine Exception weiterwerfen

Sie können eine Exception fangen und sie dann weiterwerfen:

```
try {  ...  }
catch (Exception ex)
{
   // Fehler protokollieren
   ...
   throw; // gleiche Exception erneut werfen
}
```

Wenn Sie eine Exception auf diesem Weg weiterwerfen, können Sie einen Fehler protokollieren, ohne ihn zu »verschlucken«. Zudem können Sie sich so auch aus dem Verarbeiten einer Exception zurückziehen, wenn die Umstände anders sind, als von Ihnen erwartet.

Würden wir `throw` durch `throw ex` ersetzen, würde das Beispiel immer noch funktionieren, aber die `StackTrace`-Eigenschaft der Exception würde den ursprünglichen Fehler nicht mehr widerspiegeln.

Ein weiteres verbreitetes Szenario ist das Auslösen eines spezifischeren oder aussagekräftigeren Exception-Typs:

```
try
{
   ... // ein Geburtsdatum aus einem XML-Element parsen
```

```
}

catch (FormatException ex)

{

    throw new XmlException ("Falsches Geburtsdatum", ex);

}
```

Wenn eine andere Exception ausgelöst wird, können Sie die InnerException-Eigenschaft mit der ursprünglichen Exception füllen, um das Debugging zu vereinfachen. Fast alle Exception-Typen bieten einen Konstruktor zu diesem Zweck (wie in unserem Beispiel).

Wichtige Eigenschaften von System.Exception

Die wichtigsten Eigenschaften von System.Exception sind folgende:

StackTrace
: Ein String, der alle Methoden repräsentiert, die aufgerufen sind – vom Ursprung der Exception bis zum catch-Block.

Message
: Ein String mit einer Beschreibung des Fehlers.

InnerException
: Die innere Exception (wenn vorhanden), die die äußere Exception verursacht hat. Diese innere Exception kann wieder eine weitere InnerException haben.

Die wichtigsten Exception-Typen

Die folgenden Exception-Typen werden von der CLR und dem .NET Framework immer wieder genutzt. Sie können sie selbst werfen oder als Basisklassen für das Ableiten eigener Exception-Typen verwenden.

System.ArgumentException
: Wird geworfen, wenn eine Funktion mit einem ungültigen Argument aufgerufen wird. Das deutet meist auf einen Programmfehler hin.

System.ArgumentNullException

Eine Subklasse von ArgumentException, die genutzt wird, wenn ein Funktionsargument (unerwarteterweise) null ist.

System.ArgumentOutOfRangeException

Ein Subklasse von ArgumentException, die genutzt wird, wenn ein (meist numerisches) Argument zu groß oder zu klein ist. So wird diese Exception zum Beispiel ausgelöst, wenn man einer Funktion eine negative Zahl übergibt, die nur positive Zahlen akzeptiert.

System.InvalidOperationException

Wird geworfen, wenn der Zustand eines Objekts für das erfolgreiche Ausführen einer Methode unpassend ist, unabhängig von irgendwelchen Argumentwerten. Das ist zum Beispiel dann der Fall, wenn man aus einer nicht geöffneten Datei lesen oder das nächste Element eines Enumerators holen will, die zugrunde liegende Liste aber während der Iteration verändert wurde.

System.NotSupportedException

Wird geworfen, um anzuzeigen, dass eine bestimmte Funktion nicht unterstützt wird. Ein gutes Beispiel ist der Aufruf der Methode Add für eine Collection, deren Eigenschaft IsReadOnly den Wert true hat.

System.NotImplementedException

Wird geworfen, um zu zeigen, dass eine Funktion noch nicht implementiert wurde.

System.ObjectDisposedException

Wird geworfen, wenn das Objekt, für das die Funktion aufgerufen wurde, schon »abgelegt« wurde.

Enumeration und Iteratoren

Enumeration

Ein *Enumerator* ist ein schreibgeschützter Cursor über eine *Folge von Werten*, der sich nur vorwärts bewegen kann; es ist ein Objekt, das System.Collections.IEnumerator oder System.Collections.Generic.IEnumerator<T> implementiert.

Die foreach-Anweisung iteriert über ein *enumerierbares* Objekt. Ein enumerierbares Objekt ist eine logische Repräsentation einer Folge. Es ist nicht selbst ein Cursor, sondern ein Objekt, das einen Cursor über sich selbst stellen kann. Ein enumerierbares Objekt implementiert entweder IEnumerable/IEnumerable<T> oder bietet eine Methode namens GetEnumerator, die einen Enumerator liefert.

Das Enumerierungsmuster sieht wie folgt aus:

```
class Enumerator   // implementiert üblicherweise
                                // IEnumerator<T>
{
    public IteratorVariableType Current { get {...} }
    public bool MoveNext() {...}
}
class Enumerable   // implementiert in der Regel IEnumerable<T>
{
    public Enumerator GetEnumerator() {...}
}
```

So sieht die High-Level-Variante des Iterierens über die Zeichen im Wort *beer* mithilfe einer foreach-Anweisung aus:

```
foreach (char c in "beer") Console.WriteLine (c);
```

So sieht die Low-Level-Variante des Iterierens über die Zeichen im Wort *beer* ohne Verwendung der foreach-Anweisung aus:

```
using (var enumerator = "beer".GetEnumerator())
  while (enumerator.MoveNext())
  {
    var element = enumerator.Current;
    Console.WriteLine (element);
  }
```

Wenn der Enumerator IDisposable implementiert, fungiert die foreach-Anweisung auch als using-Anweisung und entsorgt implizit das Enumerator-Objekt.

Collection-Initialisierer

Sie können ein enumerierbares Objekt in einem Schritt erstellen und füllen, zum Beispiel so:

```
using System.Collections.Generic;
...

List<int> list = new List<int> {1, 2, 3};
```

Der Compiler übersetzt die letzte Zeile in Folgendes:

```
List<int> list = new List<int>();
list.Add (1); list.Add (2); list.Add (3);
```

Das verlangt, dass das enumerierbare Objekt das Interface System.Collections.IEnumerable implementiert und über eine Add-Methode verfügt, die die erforderliche Anzahl an Parametern für den Aufruf bietet. Dictionaries (also Typen, die System.Collections.IDictionary implementieren) können Sie ähnlich initialisieren:

```
var dict = new Dictionary<int, string>()
{
    { 5, "fünf" },
    { 10, "zehn" }
};
```

Oder neuer:

```
var dict = new Dictionary<int, string>()
{
    [5] = "fünf",
    [10] = "zehn"
};
```

Diese zweite Form ist nicht nur bei Dictionaries möglich, sondern bei jedem Typ, für den ein Indexer existiert.

Iteratoren

Während die foreach-Anweisung ein *Konsument* eines Enumerators ist, ist ein Iterator ein *Produzent* davon. In diesem Beispiel nutzen wir einen Iterator, um eine Folge von Fibonacci-Zahlen zurückzugeben (bei der jede Zahl der Summe ihrer beiden Vorgänger entspricht):

```
using System;
```

```
using System.Collections.Generic;

class Test
{
    static void Main()
    {
        foreach (int fib in Fibs(6))
            Console.Write (fib + "  ");
    }

    static IEnumerable<int> Fibs(int fibCount)
    {
        for (int i = 0, prevFib = 1, curFib = 1;
                i < fibCount;
                i++)
        {
            yield return prevFib;
            int newFib = prevFib+curFib;
            prevFib = curFib;
            curFib = newFib;
        }
    }
```

}

AUSGABE: 1 1 2 3 5 8

Während eine return-Anweisung »Hier hast du den Wert, den diese Methode zurückgeben soll« ausdrückt, besagt eine yield return-Anweisung: »Hier ist das nächste Element, das dieser Enumerator erzeugen soll.« Bei jeder yield-Anweisung wird die Kontrolle an den Aufrufenden zurückgegeben, der Zustand des Aufgerufenen bleibt aber bestehen, sodass die Methode an dieser Stelle fortfahren kann, sobald der Aufrufende nach dem nächsten Element fragt. Die Lebenszeit dieses Zustands ist an den Enumerator gebunden, sodass er freigegeben werden kann, wenn der Aufrufende mit dem Enumerieren fertig ist.

Der Compiler wandelt Iteratormethoden in private Klassen um, die IEnumerable<T> und/oder IEnumerator<T> implementieren. Die Logik im Iteratorblock wird »umgekehrt« und auf die Methode MoveNext und die Eigenschaft Current der vom Compiler erzeugten Enumerator-Klasse verteilt, die damit im Prinzip zu einer Zustandsmaschine wird. Das bedeutet, dass Sie beim Aufruf einer Iteratormethode nur die generierte Klasse instanziieren – Ihr Code wird noch nicht ausgeführt! Das geschieht erst, wenn Sie damit beginnen, über die Ergebnissequenz zu enumerieren, im Allgemeinen mit einer foreach-Anweisung.

Iteratorsemantik

Ein Iterator ist eine Methode, eine Eigenschaft oder ein Indexer, die bzw. der eine oder mehrere yield-Anweisungen enthält. Ein Iterator muss eines der folgenden vier Interfaces zurückgeben (ansonsten wird der Compiler einen Fehler melden):

System.Collections.IEnumerable

System.Collections.IEnumerator

System.Collections.Generic.IEnumerable<T>

System.Collections.Generic.IEnumerator<T>

Iteratoren , die ein *Enumerator*-Interface liefern, werden in der Regel nicht so oft verwendet. Sie sind nützlich, wenn man eine eigene Collection-Klasse schreibt: Im Allgemeinen nennen Sie den Iterator GetEnumerator und lassen Ihre Klasse

IEnumerable<T> implementieren.

Iteratoren , die ein *Enumerable*-Interface liefern, sind verbreiteter und einfacher zu nutzen, weil man keine Collection-Klasse schreiben muss. Der Compiler erzeugt hinter den Kulissen eine private Klasse, die IEnumerable<T> (und IEnumerator<T>) implementiert.

Mehrere yield-Anweisungen

Ein Iterator kann mehrere yield-Anweisungen haben:

```
static void Main()
{
    foreach (string s in Foo())
        Console.Write(s + " ");     // Eins Zwei Drei
}

static IEnumerable<string> Foo()
{
    yield return "Eins";
    yield return "Zwei";
    yield return "Drei";
}
```

yield break

Die Anweisung yield break legt fest, dass der Iteratorblock vorzeitig verlassen werden soll, ohne weitere Elemente zurückzugeben. Wir können die Methode Foo anpassen, um das zu demonstrieren:

```
static IEnumerable<string> Foo (bool breakEarly)
```

```
{
    yield return "Eins";
    yield return "Zwei";
    if (breakEarly) yield break;
    yield return "Drei";
}
```

return-Anweisungen sind in einem Iteratorblock nicht erlaubt – Sie müssen stattdessen yield break verwenden.

Sequenzen kombinieren

Iteratoren sind sehr gut kombinierbar. Wir können unser Fibonacci-Beispiel erweitern, indem wir der Klasse folgende Methode hinzufügen:

```
static IEnumerable<int> EvenNumbersOnly (
    IEnumerable<int> sequence)
    {
        foreach (int x in sequence)
            if ((x % 2) == 0)
                yield return x;
    }
}
```

Dann können wir gerade Fibonacci-Zahlen folgendermaßen ausgeben:

```
foreach (int fib in EvenNumbersOnly (Fibs (6)))

    Console.Write (fib + " ");   // 2 8
```

Jedes Element wird erst im letzten Moment berechnet – dann, wenn es von einer MoveNext( )-Operation angefordert wird. Abbildung 5 zeigt die Datenanforderung und die Datenausgabe über die Zeit.

Die Kombinierbarkeit des Iteratormusters ist ein notwendiger Baustein beim Erstellen von LINQ-Abfragen.

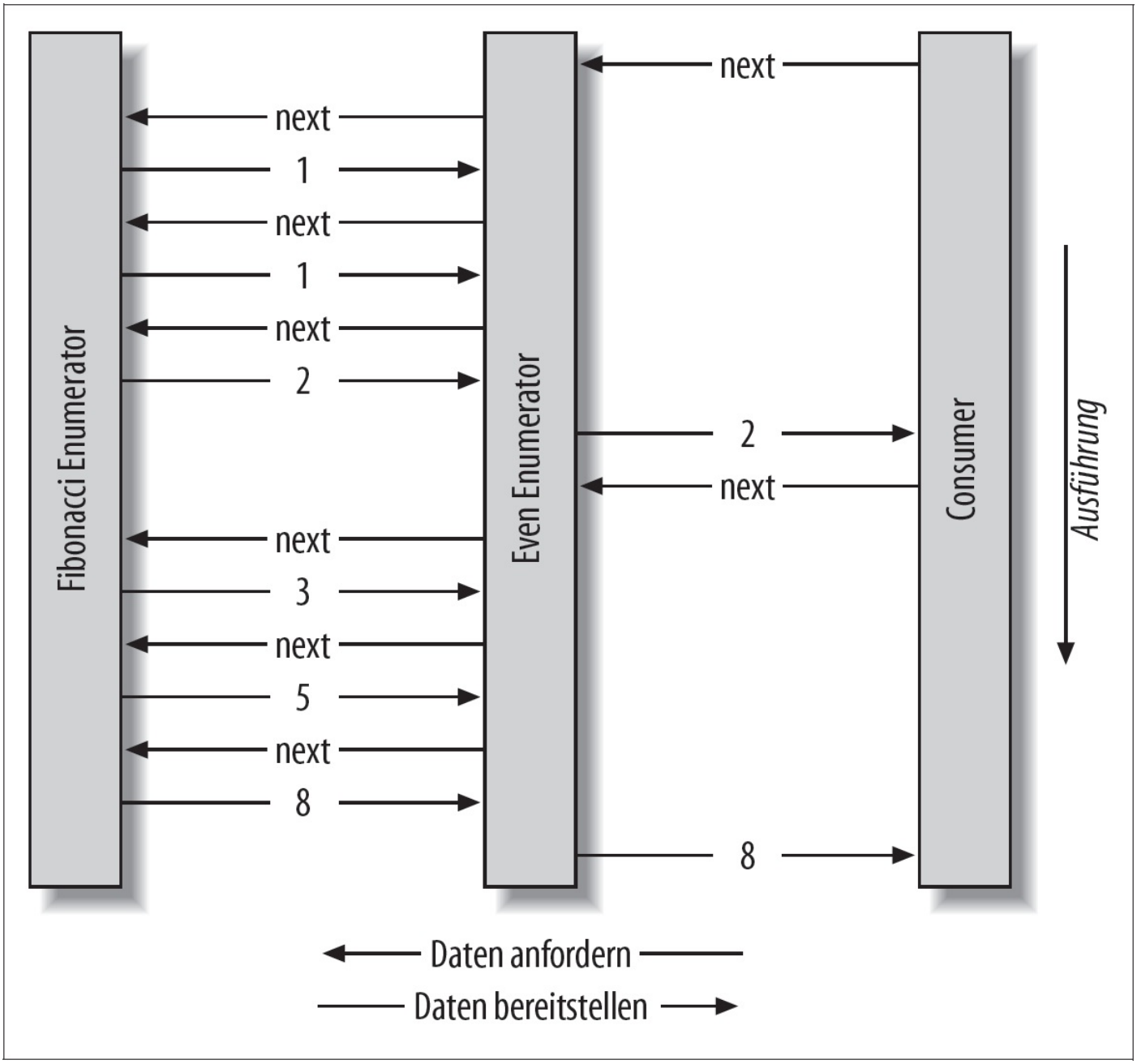

Abbildung 5: Sequenzen kombinieren

Nullbare Typen

Referenztypen können einen nicht existierenden Wert durch eine Nullreferenz darstellen. Bei Werttypen ist das allerdings normalerweise nicht möglich:

```
string s = null;   // Okay - Referenztyp

int i = null;      // Kompilierungsfehler -

                                    // int kann nicht null sein
```

Um null (nicht 0) in einem Werttyp zu repräsentieren, müssen Sie ein spezielles Konstrukt namens *nullbarer Typ* nutzen. Ein nullbarer Typ wird durch einen Werttyp mit einem Fragezeichen ? dahinter kenntlich gemacht:

```
int? i = null;                  // Okay - nullbarer Typ

Console.WriteLine (i == null);     // True
```

Das Struct Nullable<T>

T? wird in System.Nullable<T> umgewandelt. Nullable<T> ist ein leichtgewichtiges, unveränderliches Struct, das nur zwei Felder für Value und HasValue besitzt. Leicht gekürzt, ist System.Nullable<T> sehr einfach:

```
public struct Nullable<T> where T : struct
{
    public T Value {get;}
    public bool HasValue {get;}
    public T GetValueOrDefault();
    public T GetValueOrDefault (T defaultValue);
```

```
    ...
}
```

Der Code

```
int? i = null;
Console.WriteLine (i == null);          // True
```

wird vom Compiler umgewandelt in

```
Nullable<int> i = new Nullable<int>();
Console.WriteLine (! i.HasValue);       // True
```

Wenn man versucht, Value auszulesen, wenn HasValue false ist, wird eine InvalidOperationException geworfen. GetValueOrDefault( ) liefert Value zurück, wenn HasValue true ist – ansonsten gibt es new T( ) oder einen angegebenen Standardwert zurück.

Der Standardwert von T? ist null.

Nullbare Konvertierungen

Die Konvertierung von T nach T? ist implizit, von T? nach T dagegen explizit:

```
int? x = 5;       // implizit
int y = (int)x;   // explizit
```

Der explizite Cast entspricht einem Aufruf der Eigenschaft Value des nullbaren Objekts. Wenn daher HasValue den Wert false hat, wird eine InvalidOperationException geworfen.

Boxing/Unboxing nullbarer Werte

Wenn T? geboxt ist, enthält der geboxte Wert auf dem Heap T, nicht T?. Diese Optimierung ist möglich, weil ein geboxter Wert ein Referenztyp ist, der schon null darstellen kann.

C# gestattet außerdem das Unboxing nullbarer Typen mit dem as-Operator. Das Ergebnis ist null, wenn der Cast fehlschlägt:

```
object o = "string";

int? x = o as int?;

Console.WriteLine (x.HasValue);   // False
```

Übernommene Operatoren

Das Struct Nullable<T> definiert keine Operatoren wie <, > oder sogar ==. Trotzdem lässt sich der folgende Code kompilieren und korrekt ausführen:

```
int? x = 5;

int? y = 10;

bool b = x < y;      // True
```

Das funktioniert, weil der Compiler den Kleiner-als-Operator vom zugrunde liegenden Werttyp ausleiht oder »übernimmt«. Semantisch gesehen, wird der vorige Vergleichsausdruck in das hier umgewandelt:

```
bool b = (x.HasValue && y.HasValue)

              ? (x.Value < y.Value)

              : false;
```

Anders ausgedrückt: Wenn sowohl x als auch y Werte haben, werden sie über den Kleiner-als-Operator von int verglichen, ansonsten wird false zurückgegeben.

Das Übernehmen von Operatoren bedeutet, dass Sie implizit die Operatoren von T für T? nutzen können. Sie können Operatoren für T? definieren, um ein besonderes Null-

Verhalten zu bieten, aber in einem Großteil der Fälle ist es am besten, sich darauf zu verlassen, dass der Compiler automatisch eine systematische nullbare Logik für Sie anwendet.

Wie der Compiler die Null-Logik ausführt, ist von der Kategorie des Operators abhängig.

Gleichheitsoperatoren (==, !=)

Übernommene Gleichheitsoperatoren behandeln null, wie es Referenztypen tun. Das bedeutet, dass zwei Null-Werte gleich sind:

```
Console.WriteLine (      null ==       null);  // True

Console.WriteLine ((bool?)null == (bool?)null);  // True
```

Außerdem:

- Wenn genau ein Operand null ist, sind alle Operanden ungleich.
- Wenn beide Operanden nicht null sind, werden ihre Values verglichen.

Relationale Operatoren (<, <=, >=, >)

Die relationalen Operatoren basieren auf dem Prinzip, dass es unsinnig ist, Null-Operanden zu vergleichen. Das bedeutet, dass ein Vergleich eines Null-Werts mit einem anderen Null- oder Nicht-Null-Wert false zurückliefert.

```
bool b = x < y;    // Übersetzung:


bool b = (x == null || y == null)
    ? false
    : (x.Value < y.Value);


// b ist False (wenn x 5 und y null)
```

Alle anderen Operatoren (+, –, *, /, %, &, |, ^, <<, >>, +, ++, --, !, ~)

Diese Operatoren liefern null zurück, wenn einer der Operanden null ist. Das Muster sollte SQL-Anwendern vertraut sein.

```
int? c = x + y;   // Translation:

int? c = (x == null || y == null)
              ? null
              : (int?) (x.Value + y.Value);

// c ist null (wenn x 5 und y null ist)
```

Eine Ausnahme tritt auf, wenn die Operatoren & und | auf bool? angewandt werden. Diese werden wir uns gleich ansehen.

Mischen von nullbaren und nicht nullbaren Typen

Sie können nullbare und nicht nullbare Typen kombinieren und vergleichen (das funktioniert, weil es eine implizite Konvertierung von T nach T? gibt):

```
int? a = null;

int b = 2;

int? c = a + b;   // c ist null - entspricht a + (int?)b
```

bool? mit den Operatoren & und |

Wenn Operanden den Typ bool? haben, behandeln die Operatoren & und | null als unbekannten Wert. Daher ist null | true true, denn:

- Wenn der unbekannte Wert false wäre, würde das Ergebnis true sein.
- Wenn der unbekannte Wert true wäre, würde das Ergebnis true sein.

Genauso ist null & false false. Dieses Verhalten wird SQL-Anwendern vertraut vorkommen. Das folgende Beispiel zeigt andere Kombinationen:

```
bool? n = null, f = false, t = true;
```

```
Console.WriteLine (n | n);    // (null)

Console.WriteLine (n | f);    // (null)

Console.WriteLine (n | t);    // True

Console.WriteLine (n & n);    // (null)

Console.WriteLine (n & f);    // False

Console.WriteLine (n & t);    // (null)
```

Nullbare Typen und Null-Operatoren

Nullbare Typen arbeiten besonders gut mit dem Operator ?? zusammen (siehe »Null-Verbindungsoperator« auf Seite 49), zum Beispiel:

```
int? x = null;

int y = x ?? 5;   // y ist 5


int? a = null, b = null, c = 123;

Console.WriteLine (a ?? b ?? c);   // 123
```

Die Verwendung von ?? auf einen nullbaren Werttyp entspricht einem Aufruf von GetValueOrDefault mit einem expliziten Standardwert, nur dass der Ausdruck für den Standardwert niemals ausgewertet wird, wenn die Variablen nicht null ist.

Nullbare Typen sind auch gut für den Einsatz mit dem Null-Bedingungsoperator (siehe »Null-Bedingungsoperator« auf Seite 50) zu gebrauchen. Im folgenden Beispiel wird length zu null ausgewertet:

```
System.Text.StringBuilder sb = null;

int? length = sb?.ToString().Length;
```

Darüber hinaus können wir das mit dem Null-Verbindungsoperator kombinieren, um statt null den Wert 0 zu erhalten:

```
int length = sb?.ToString().Length ?? 0;
```

Erweiterungsmethoden

Erweiterungsmethoden ermöglichen es, bestehende Typen durch neue Methoden zu erweitern, ohne die Definition des Ursprungstyps verändern zu müssen. Eine Erweiterungsmethode ist eine statische Methode einer statischen Klasse, bei der der Modifikator this dem ersten Parameter mitgegeben wird. Der Typ des ersten Parameters ist dann der zu erweiternde Typ:

```
public static class StringHelper
{
    public static bool IsCapitalized (this string s)
    {
        if (string.IsNullOrEmpty (s)) return false;
        return char.IsUpper (s[0]);
    }
}
```

Die Erweiterungsmethode IsCapitalized kann so aufgerufen werden, als würde es sich um eine Instanzmethode eines Strings handeln:

```
Console.Write ("Perth".IsCapitalized());
```

Der Aufruf einer Erweiterungsmethode wird beim Kompilieren in einen normalen statischen Methodenaufruf umgewandelt:

```
Console.Write (StringHelper.IsCapitalized ("Perth"));
```

Interfaces können ebenfalls erweitert werden:

```
public static T First<T> (this IEnumerable<T> sequence)
{
    foreach (T element in sequence)
        return element;
    throw new InvalidOperationException ("Keine Elemente!");
}
...
Console.WriteLine ("Seattle".First());   // S
```

Verketten von Erweiterungsmethoden

Erweiterungsmethoden können wie Instanzmethoden verkettet werden. Schauen Sie sich die folgenden beiden Funktionen an:

```
public static class StringHelper
{
    public static string Pluralize (this string s) {...}
    public static string Capitalize (this string s) {...}
}
```

x und y sind gleich und werden beide zu "Sausages", aber x verwendet Erweiterungsmethoden, während y statische Methoden nutzt:

```
string x = "sausage".Pluralize().Capitalize();

string y = StringHelper.Capitalize
```

```
(StringHelper.Pluralize ("sausage"));
```

Mehrdeutigkeit und Auflösung

Namensräume

Eine Erweiterungsmethode kann nicht angesprochen werden, wenn der Namensraum nicht im Geltungsbereich liegt (normalerweise wird er dafür durch eine using-Anweisung importiert).

Erweiterungsmethoden vs. Instanzmethoden

Eine kompatible Instanzmethode hat immer Vorrang vor einer Erweiterungsmethode, selbst wenn die Parameter der Erweiterungsmethode auf spezifischere Typen passen.

Erweiterungsmethoden vs. Erweiterungsmethoden

Wenn zwei Erweiterungsmethoden die gleiche Signatur haben, muss die Erweiterungsmethode als normale statische Methode aufgerufen werden, um eindeutig klarzustellen, welche Methode aufgerufen werden soll. Wenn eine Erweiterungsmethode allerdings spezifischere (»passendere«) Argumente hat, bekommt diese den Vorzug.

Anonyme Typen

Ein anonymer Typ ist eine einfache Klasse, die bei Bedarf erzeugt wird, um ein paar Werte zu speichern. Um einen anonymen Typ zu erzeugen, verwenden Sie das Schlüsselwort new, gefolgt von einem Objektinitialisierer, in dem Sie die Eigenschaften und Werte angeben, die der Typ enthalten wird. Hier ist ein Beispiel:

```
var dude = new { Name = "Bob", Age = 1 };
```

Der Compiler löst das auf, indem er einen privaten eingebetteten Typ erstellt, der nur lesbare Eigenschaften für Name (Typ string) und Age (Typ int) besitzt. Sie müssen das Schlüsselwort var verwenden, um sich auf einen anonymen Typ zu beziehen, da der Name des Typs vom Compiler erzeugt wird.

Der Eigenschaftsname eines anonymen Typs kann aus einem Ausdruck ermittelt werden, der selbst ein Bezeichner ist:

```
int Age = 1;

var dude = new { Name = "Bob", Age };
```

Das ist äquivalent zu

```
var dude = new { Name = "Bob", Age = Age };
```

Ein Array anonymer Typen können Sie folgendermaßen erstellen:

```
var dudes = new[]
{
    new { Name = "Bob", Age = 30 },
    new { Name = "Mary", Age = 40 }
};
```

Anonyme Typen werden vor allem beim Schreiben von LINQ-Abfragen genutzt.

Tupel (C# 7)

Tupel bieten wie anonyme Typen eine einfache Möglichkeit, einen Satz von Variablen abzulegen. Haupteinsatzgebiet von Tupeln ist die sichere Rückgabe mehrerer Werte aus einer Methode, ohne out-Parameter nutzen zu müssen (etwas, das mit anonymen Typen nicht möglich ist). Ein *Tupel-Literal* erzeugen Sie am einfachsten, indem Sie die gewünschten Werte in Klammern aufführen. Damit wird ein Tupel mit unbenannten Elementen geschaffen:

```
var bob = ("Bob", 23);

Console.WriteLine (bob.Item1);  // Bob

Console.WriteLine (bob.Item2);  // 23
```

Die Tupel-Funktionalität von C# 7 baut auf einer Reihe generischer Structs namens System.ValueTupel <…> auf. Diese sind nicht Teil des .NET Framework 4.6, sie befinden sich in einer Assembly namens System.ValueTuple, die in einem NuGet-Paket gleichen Namens verfügbar ist. Nutzen Sie Visual Studio mit dem .NET Framework 4.6, müssen Sie dieses Paket explizit herunterladen. (Verwenden Sie LINQPad, wird die erforderliche Assembly automatisch eingebunden.)

System.ValueTuple ist in das .NET Framework 4.7 in *mscorlib.dll* eingebaut.

Anders als bei anonymen Typen ist var optional, und Sie können einen *Tupel-Typ* explizit festlegen:

```
(string,int) bob = ("Bob", 23);
```

Damit können Sie ein Tupel aus einer Methode zurückgeben:

```
static (string,int) GetPerson() => ("Bob", 23);

static void Main()
```

```
{
    (string,int) person = GetPerson();
    Console.WriteLine (person.Item1);  // Bob
    Console.WriteLine (person.Item2);  // 23
}
```

Tupel arbeiten sehr gut mit Generics zusammen, daher sind alle folgenden Typen korrekt:

```
Task<(string,int)>
Dictionary<(string,int),Uri>
IEnumerable<(int ID, string Name)>  // siehe unten ...
```

Tupel sind Werttypen mit *mutablen* Elementen (les- und schreibbar). Das heißt, Sie können Item1, Item2 und so weiter nach dem Erstellen eines Tupels verändern.

Tupel-Elemente benennen

Sie können den Elementen beim Erstellen von Tupel-Literalen sinnvolle Namen verpassen:

```
var tuple = (Name:"Bob", Age:23);
Console.WriteLine (tuple.Name);  // Bob
Console.WriteLine (tuple.Age);   // 23
```

Das Gleiche erreichen Sie beim Spezifizieren von Tupel-Typen:

```
static (string Name, int Age) GetPerson() => ("Bob",23);
```

Tupel sind eine syntaktische Vereinfachung für den Einsatz einer Familie von generischen Structs namens ValueTuple<T1> und ValueTuple<T1,T2>, die Felder mit den Namen Item1, Item2 und so weiter enthalten. Daher ist (string,int) ein Alias für ValueTuple<string,int>. »Benannte Elemente« existieren daher nur im Quellcode – und in der Vorstellung des Compilers –, zur Laufzeit sind sie größtenteils wieder verschwunden.

Tupel dekonstruieren

Tupel unterstützen implizit das Dekonstruktorenmuster (siehe »Dekonstruktoren (C# 7)« auf Seite 68), daher können Sie ein Tupel problemlos in seine einzelnen Variablen *dekonstruieren*. Statt also diesen Code einzusetzen:

```
var bob = ("Bob", 23);
string name = bob.Item1;
int age = bob.Item2;
```

können Sie auch Folgendes schreiben:

```
var bob = ("Bob", 23);
(string name, int age) = bob; // bob in name und
                              // age auflösen.
Console.WriteLine (name);
Console.WriteLine (age);
```

Die Syntax für das Dekonstruieren ist der zum Deklarieren eines Tupels mit benannten Elementen verwirrend ähnlich. Im Folgenden sehen Sie den Unterschied:

```
(string name, int age) = bob;      // dekonstruieren
(string name, int age) bob2 = bob; // Tupel deklarieren
```

LINQ

LINQ (*Language Integrated Query*) ermöglicht Ihnen, strukturierte, typsichere Abfragen für lokale Objekt-Collections und Datenquellen auf anderen Rechnern zu schreiben.

LINQ lässt Sie beliebige Collections durchsuchen, die IEnumerable<> implementieren – sei es ein Array, eine Liste, ein XML-DOM oder eine entfernte Datenquelle (wie eine Tabelle im SQL-Server). LINQ kombiniert die Vorteile einer Typprüfung beim Kompilieren und einer dynamischen Abfrageerzeugung.

Eine gute Möglichkeit, mit LINQ zu experimentieren, ist das Herunterladen von LINQPad unter *http://www.linqpad.net*. LINQPad ermöglicht Ihnen, interaktiv lokale Collections und SQL-Datenbanken in LINQ ohne ein weiteres Setup abzufragen, und bringt von Haus aus zahllose Beispiele mit.

Grundlagen von LINQ

Die grundlegenden Dateneinheiten in LINQ sind *Sequenzen* und *Elemente*. Eine Sequenz ist ein beliebiges Objekt, das das generische Interface IEnumerable implementiert, ein Element ist jedes Teil in der Sequenz. Im folgenden Beispiel ist names eine Sequenz, und Tom, Dick und Harry sind Elemente:

```
string[] names = { "Tom", "Dick", "Harry" };
```

Eine Sequenz wie diese bezeichnen wir als *lokale Sequenz*, da sie eine lokale Collection mit Objekten im Speicher repräsentiert.

Ein *Abfrageoperator* ist eine Methode, die eine Sequenz transformiert. Ein typischer Abfrageoperator erwartet eine *Eingabesequenz* und gibt eine transformierte *Ausgabesequenz* zurück. In der Klasse Enumerable in System.Linq gibt es etwa 40 Abfrageoperatoren, die alle als statische Erweiterungsmethoden implementiert sind. Diese werden *Standard-Abfrageoperatoren* genannt.

LINQ unterstützt auch Sequenzen, die dynamisch aus einer Remote-Datenquelle gefüllt werden können, zum Beispiel vom SQL-Server. Diese Sequenzen implementieren zusätzlich das Interface IQueryable<> und erhalten Hilfe von passenden Standard-Abfrageoperatoren in der Klasse

Queryable.

Eine einfache Abfrage

Eine Abfrage ist ein Ausdruck, der Sequenzen mit einem oder mehreren Abfrageoperatoren transformiert. Die einfachste Abfrage besteht aus einer Eingabesequenz und einem Operator. So können wir zum Beispiel den Operator Where auf ein einfaches Array anwenden, um die Elemente zu extrahieren, die mindestens vier Zeichen lang sind:

```
string[] names = { "Tom", "Dick", "Harry" };

IEnumerable<string> filteredNames =
System.Linq.Enumerable.Where (
names, n => n.Length >= 4);
foreach (string n in filteredNames)
    Console.Write (n + "|");          // Dick|Harry|
```

Da die Standard-Abfrageoperatoren als Erweiterungsmethoden implementiert sind, können wir Where direkt auf names aufrufen, als wäre es eine Instanzmethode:

```
IEnumerable<string> filteredNames =
    names.Where (n => n.Length >= 4);
```

(Damit sich dieser Code kompilieren lässt, müssen Sie den Namensraum System.Linq mit einer using-Direktive importieren.) Die Methode Where in System.Linq.Enumerable hat folgende Signatur:

```
static IEnumerable<TSource> Where<TSource> (
```

```
    this IEnumerable<TSource> source,
    Func<TSource,bool> predicate)
```

source ist die *Eingabesequenz*, und predicate ist ein Delegate, das für jedes *Eingabeelement* aufgerufen wird. Die Methode Where nimmt alle Elemente mit in die *Ausgabesequenz* auf, für die das Delegate true zurückliefert. Intern wird sie mit einem Iterator implementiert – so sieht der Quellcode aus:

```
foreach (TSource element in source)
    if (predicate (element))
        yield return element;
```

Projizieren

Ein weiterer grundlegender Abfrageoperator ist die Methode Select. Diese transformiert (*projiziert*) jedes Element in der Eingabesequenz mit einem angegebenen Lambda-Ausdruck:

```
string[] names = { "Tom", "Dick", "Harry" };

IEnumerable<string> upperNames =
    names.Select (n => n.ToUpper());

foreach (string n in upperNames)
    Console.Write (n + "|");       // TOM|DICK|HARRY|
```

Eine Abfrage kann in einen anonymen Typ projizieren:

```
var query = names.Select (n => new {
                                    Name = n,
```

```
                                        Length = n.Length

                                    });

    foreach (var row in query)

        Console.WriteLine (row);
```

Das Ergebnis sieht so aus:

```
{ Name = Tom, Length = 3 }

{ Name = Dick, Length = 4 }

{ Name = Harry, Length = 5 }
```

Take und Skip

Die ursprüngliche Reihenfolge der Elemente in einer Eingabesequenz ist für LINQ von Bedeutung. Manche Abfrageoperatoren verlassen sich auf dieses Verhalten, zum Beispiel Take, Skip und Reverse. Der Operator Take gibt die ersten x Elemente aus und verwirft den Rest:

```
int[] numbers  = { 10, 9, 8, 7, 6 };

IEnumerable<int> firstThree = numbers.Take (3);

// firstThree ist { 10, 9, 8 }
```

Der Operator Skip ignoriert die ersten x Elemente und gibt den Rest aus:

```
IEnumerable<int> lastTwo = numbers.Skip (3);
```

Elementoperatoren

Nicht alle Abfrageoperatoren liefern eine Sequenz zurück. Die *Elementoperatoren* extrahieren genau ein Element aus der Eingabesequenz – so zum Beispiel First, Last,

Single und ElementAt:

```
int[] numbers    = { 10, 9, 8, 7, 6 };

int firstNumber  = numbers.First();              // 10

int lastNumber   = numbers.Last();               // 6

int secondNumber = numbers.ElementAt (2);        // 8

int firstOddNum  = numbers.First (n => n%2 == 1);  // 9
```

Alle diese Operatoren werfen eine Exception, wenn keine Elemente vorhanden sind. Um statt einer Exception Null bzw. einen leeren Rückgabewert zu erhalten, müssen Sie FirstOrDefault, LastOr-Default, SingleOrDefault oder ElementAtOrDefault nutzen.

Die Methoden Single und SingleOrDefault entsprechen First und FirstOrDefault, nur dass sie eine Exception werfen, wenn es mehr als einen Eintrag gibt. Dieses Verhalten ist nützlich, wenn man in einer Datenbankabfrage eine Zeile über einen Primärschlüssel erhalten will.

Aggregationsoperatoren

Die *Aggregationsoperatoren* liefern einen skalaren Wert zurück, im Allgemeinen einen numerischen. Die am häufigsten genutzten Aggregationsoperatoren sind Count, Min, Max und Average:

```
int[] numbers = { 10, 9, 8, 7, 6 };

int count     = numbers.Count();          // 5

int min       = numbers.Min();            // 6

int max       = numbers.Max();            // 10

double avg    = numbers.Average();        // 8
```

Count kann ein optionales Prädikat übergeben werden, mit dem bestimmt wird, welche Elemente zu zählen sind. Mit der folgenden Zeile werden alle geraden Zahlen gezählt:

```
int evenNums = numbers.Count (n => n % 2 == 0);   // 3
```

Den Operatoren Min, Max und Average kann ein optionales Argument übergeben werden, das jedes Element transformiert, bevor es aggregiert wird:

```
int maxRemainderAfterDivBy5 = numbers.Max
                                    (n => n % 5);      // 4
```

Mit diesem Code wird der quadratische Mittelwert von numbers berechnet:

```
double rms = Math.Sqrt (numbers.Average (n => n * n));
```

Quantifizierer

Die *Quantifizierer* liefern einen bool-Wert zurück. Quantifizierer sind Contains, Any, All und SequenceEquals (das zwei Sequenzen vergleicht):

```
int[] numbers = { 10, 9, 8, 7, 6 };

bool hasTheNumberNine = numbers.Contains (9);   // true

bool hasMoreThanZeroElements = numbers.Any();   // true

bool hasOddNum = numbers.Any (n => n % 2 == 1); // true

bool allOddNums = numbers.All (n => n % 2 == 1); // false
```

Set-Operatoren

Die *Set*-Operatoren erwarten zwei Eingabesequenzen vom gleichen Typ. Concat hängt eine Sequenz an die andere an. Union tut das Gleiche, entfernt aber Duplikate:

```
int[] seq1 = { 1, 2, 3 }, seq2 = { 3, 4, 5 };
```

```
IEnumerable<int>
  concat = seq1.Concat (seq2),   // { 1, 2, 3, 3, 4, 5 }
  union  = seq1.Union  (seq2),   // { 1, 2, 3, 4, 5 }
```

Die anderen beiden Operatoren in dieser Kategorie sind Intersect und Except:

```
IEnumerable<int>
  commonality = seq1.Intersect (seq2),   // { 3 }
  difference1 = seq1.Except    (seq2),   // { 1, 2 }
  difference2 = seq2.Except    (seq1);   // { 4, 5 }
```

Verzögerte Ausführung

Ein wichtiges Feature vieler Abfrageoperatoren ist, dass sie nicht beim Erstellen, sondern erst beim Enumerieren ausgeführt werden (also dann, wenn MoveNext für den Enumerator aufgerufen wird). Schauen Sie sich die folgende Abfrage an:

```
var numbers = new List<int> { 1 };

IEnumerable<int> query = numbers.Select (n => n * 10);
numbers.Add (2);    // ein zusätzliches Element einschleusen

foreach (int n in query)
  Console.Write (n + "|");          // 10|20|
```

Die zusätzliche Zahl wurde erst nach dem Erstellen der Abfrage in die Liste eingeschmuggelt, sie ist aber trotzdem im Ergebnis enthalten, weil das Filtern oder Sortieren erst bei der foreach-Anweisung durchgeführt wird. Das wird als *verzögerte* Auswertung bezeichnet. Die verzögerte Ausführung entkoppelt das *Erstellen* der Abfrage von ihrer *Ausführung* und ermöglicht Ihnen damit, eine Abfrage in mehreren

Schritten aufzubauen und eine Datenbank abzufragen, ohne alle Zeilen auf den Client zu laden. Alle Standard-Abfrageoperatoren arbeiten mit der verzögerten Ausführung, jedoch mit folgenden Ausnahmen:

- Operatoren, die ein einzelnes Element oder einen skalaren Wert zurückgeben (*Elementoperatoren*, *Aggregationsoperatoren* und *Quantifizierer*)
- die *Konvertierungsoperatoren* ToArray, ToList, ToDictionary und ToLookup

Die Konvertierungsoperatoren sind manchmal gerade deshalb nützlich, weil sie eine sofortige Auswertung anstoßen. Das kann hilfreich sein, wenn Sie das Ergebnis zu einem bestimmten Zeitpunkt »einfrieren« oder cachen wollen oder um die erneute Ausführung einer berechnungsintensiven Abfrage oder eine Abfrage auf einer entfernten Quelle wie einer LINQ-to-SQL-Tabelle zu vermeiden. (Eine Nebenwirkung der verzögerten Ausführung ist, dass die Abfrage bei einer späteren erneuten Enumerierung neu ausgewertet wird.)

Das folgende Beispiel zeigt die Verwendung des Operators ToList:

```
var numbers = new List<int>() { 1, 2 };

List<int> timesTen = numbers
    .Select (n => n * 10)
    .ToList(); // wird direkt in eine List<int> herausgezogen

numbers.Clear();
Console.WriteLine (timesTen.Count);      // immer noch 2
```

Unterabfragen ermöglichen eine weitere Ebene der Indirektion. Alle Teile einer Unterabfrage werden verzögert ausgeführt, einschließlich der Aggregations- und Konvertierungsmethoden, weil die Unterabfrage selbst erst bei Bedarf ausgeführt wird. Ist names ein String-Array, sieht eine Unterabfrage so aus:

```
names.Where (
    n => n.Length ==
        names.Min (n2 => n2.Length))
```

Standard-Abfrageoperatoren

Man kann die Standard-Abfrageoperatoren (wie sie in der Klasse System.Linq.Enumerable implementiert sind) in zwölf Kategorien einteilen, die in Tabelle 1 beschrieben werden.

Tabelle 1: Abfrageoperator-Kategorien

Kategorie	Beschreibung	Verzögerte Ausführung?
Filtern	Liefert eine Untermenge der Elemente zurück, die eine bestimmte Bedingung erfüllen.	Ja
Projektion	Transformiert jedes Element (im Allgemeinen mithilfe einer Lambda-Funktion), wobei Untersequenzen optional expandiert werden.	Ja
Verknüpfung	Verbindet Elemente einer Collection mit einer anderen, wobei eine zeitsparende Lookup-Strategie gewählt wird.	Ja
Ordnen	Liefert eine umgeordnete Sequenz zurück.	Ja
Gruppieren	Gruppiert eine Sequenz in Untersequenzen.	Ja
Set	Akzeptiert zwei Sequenzen gleichen Typs und liefert ihre Gemeinsamkeiten, Summe oder Differenz zurück.	Ja
Element	Wählt ein einzelnes Element aus einer Sequenz.	Nein
Aggregation	Führt eine Berechnung für eine Sequenz durch und liefert einen skalaren Wert zurück (normalerweise eine Zahl).	Nein
Quantifizierung	Führt eine Berechnung für eine Sequenz durch und liefert true oder false zurück.	Nein
Konvertierung: Import	Wandelt eine nichtgenerische Sequenz in eine (abfragbare) generische Sequenz um.	Ja
Konvertierung: Export	Wandelt eine Sequenz in ein Array, eine List, ein Dictionary oder einen Lookup um, wobei sofort ausgewertet wird.	Nein
Generieren	Erstellt eine einfache Sequenz.	Ja

Tabelle 2 bis Tabelle 13 bieten einen Überblick über alle Abfrageoperatoren. Für die fett gedruckten Operatoren bietet C# eine spezielle Unterstützung (siehe »Abfrageausdrücke« auf Seite 166).

Tabelle 2: Filteroperatoren

Methode	Beschreibung
Where	Liefert eine Untermenge mit Elementen zurück, die eine bestimmte Bedingung erfüllen.
Take	Liefert die ersten x Elemente zurück und verwirft den Rest.
Skip	Ignoriert die ersten x Elemente und liefert den Rest zurück.
TakeWhile	Gibt Elemente aus der Eingabesequenz zurück, bis das angegebene Prädikat true ist.
SkipWhile	Ignoriert Elemente aus der Eingabesequenz, bis das angegebene Prädikat true ist, und gibt dann den Rest zurück.
Distinct	Liefert eine Collection ohne Duplikate zurück.

Tabelle 3: Projektionsoperatoren

Methode	Beschreibung
Select	Transformiert jedes Eingabeelement mit einem angegebenen Lambda-Ausdruck.
SelectMany	Transformiert jedes Eingabeelement, dann werden die Ergebnis-Untersequenzen flach geklopft und aneinandergehängt.

Tabelle 4: Verknüpfungsoperatoren

Methode	Beschreibung
Join	Wendet eine Lookup-Strategie an, um Elemente aus zwei Collections zu verbinden und eine flache Ergebnismenge zurückzugeben.
GroupJoin	Wie oben, aber es wird eine hierarchische Ergebnismenge zurückgegeben.
Zip	Enumeriert parallel zwei Sequenzen und liefert eine Sequenz, die eine Funktion auf jedes Elementpaar anwendet.

Tabelle 5: Ordnungsoperatoren

Methode	Beschreibung
OrderBy, **ThenBy**	Gibt die Elemente in aufsteigender Reihenfolge sortiert zurück.
OrderByDescending, ThenByDescending	Gibt die Elemente in absteigender Reihenfolge sortiert zurück.
Reverse	Liefert die Elemente in umgekehrter Reihenfolge zurück.

Tabelle 6: Gruppierungsoperatoren

Methode	Beschreibung
GroupBy	Gruppiert eine Sequenz in Untersequenzen.

Tabelle 7: Mengenoperatoren

Methode	Beschreibung
Concat	Verbindet zwei Sequenzen.
Union	Verbindet zwei Sequenzen und entfernt doppelte Einträge.
Intersect	Liefert die Elemente zurück, die in beiden Sequenzen vorhanden sind.
Except	Liefert Elemente zurück, die in der ersten, aber nicht in der zweiten Sequenz vorhanden sind.

Tabelle 8: Elementoperatoren

Methode	Beschreibung
First, FirstOrDefault	Liefert das erste Element in der Sequenz bzw. das erste Element zurück, das eine angegebene Bedingung erfüllt.
Last, LastOrDefault	Liefert das letzte Element in der Sequenz bzw. das letzte Element zurück, das eine angegebene Bedingung erfüllt.
Single, SingleOrDefault	Entspricht First/FirstOrDefault, wirft aber eine Exception, wenn es mehr als einen Treffer gibt.
ElementAt,	Liefert das Element an der angegebenen Position zurück.

ElementAtOrDefault	
DefaultIfEmpty	Liefert eine Sequenz mit nur einem Wert, der null oder default(TSource) ist, wenn die Sequenz keine Elemente enthält.

Tabelle 9: Aggregationsoperatoren

Methode	Beschreibung
Count, LongCount	Liefert die Anzahl an Elementen in der Eingabesequenz zurück bzw. die Anzahl an Elementen, die ein Prädikat erfüllen.
Min, Max	Liefert das kleinste bzw. größte Element in der Sequenz zurück.
Sum, Average	Berechnet die Summe bzw. den Mittelwert von Elementen in der Sequenz.
Aggregate	Führt eine benutzerdefinierte Zusammenfassungsoperation durch.

Tabelle 10: Qualifizierer

Methode	Beschreibung
Contains	Liefert true zurück, wenn die Sequenz das angegebene Element enthält.
Any	Liefert true zurück, wenn mindestens ein Element das angegebene Prädikat erfüllt.
All	Liefert true zurück, wenn alle Elemente das angegebene Prädikat erfüllen.
SequenceEqual	Liefert true zurück, wenn die zweite Sequenz die gleichen Elemente enthält wie die Eingabesequenz.

Tabelle 11: Konvertierungsoperatoren (Import)

Methode	Beschreibung
OfType	Konvertiert IEnumerable nach IEnumerable<T> und verwirft falsch typisierte Elemente.

Cast	Konvertiert IEnumerable nach IEnumerable<T> und wirft eine Exception, wenn es ein falsch typisiertes Element gibt.

Tabelle 12: Konvertierungsoperatoren (Export)

Methode	Beschreibung
ToArray	Konvertiert IEnumerable<T> nach T[].
ToList	Konvertiert IEnumerable<T> nach List<T>.
ToDictionary	Konvertiert IEnumerable<T> nach Dictionary<TKey, TValue>.
ToLookup	Konvertiert IEnumerable<T> nach ILookup<TKey, TElement>.
AsEnumerable	Downcastet nach IEnumerable<T>.
AsQueryable	Castet oder konvertiert nach IQueryable<T>.

Tabelle 13: Generierungsoperatoren

Methode	Beschreibung
Empty	Erzeugt eine leere Sequenz.
Repeat	Erstellt eine Sequenz mit sich wiederholenden Elementen.
Range	Erstellt eine Sequenz mit Integer-Zahlen.

Abfrageoperatoren verketten

Um komplexere Abfragen zu erstellen, verketten Sie Abfrageoperatoren. So extrahiert zum Beispiel die folgende Abfrage alle Strings, die den Buchstaben *a* enthalten, sortiert sie nach der Länge und wandelt dann die Ergebnisse in Großbuchstaben um:

```
string[] names = { "Tom","Dick","Harry","Mary","Jay" };

IEnumerable<string> query = names
    .Where   (n => n.Contains ("a"))
```

```
        .OrderBy (n => n.Length)
        .Select  (n => n.ToUpper());

    foreach (string name in query)
        Console.Write (name + "|");

    // RESULT: JAY|MARY|HARRY|
```

Where, OrderBy und Select sind Standard-Abfrageoperatoren, die in Erweiterungsmethoden der Klasse Enumerable aufgelöst werden. Der Operator Where liefert eine gefilterte Version der Eingabesequenz; OrderBy gibt eine sortierte Version seiner Eingabesequenz aus; Select liefert eine Sequenz, in der jedes Eingabeelement mit einem angegebenen Lambda-Ausdruck (in diesem Fall n.ToUpper( )) transformiert oder *projiziert* wird. Die Daten fließen von links nach rechts durch die Kette der Operatoren, sodass die Daten erst gefiltert, dann sortiert und dann projiziert werden. Das Endergebnis sieht aus wie eine Kette von Förderbändern (siehe Abbildung 6).

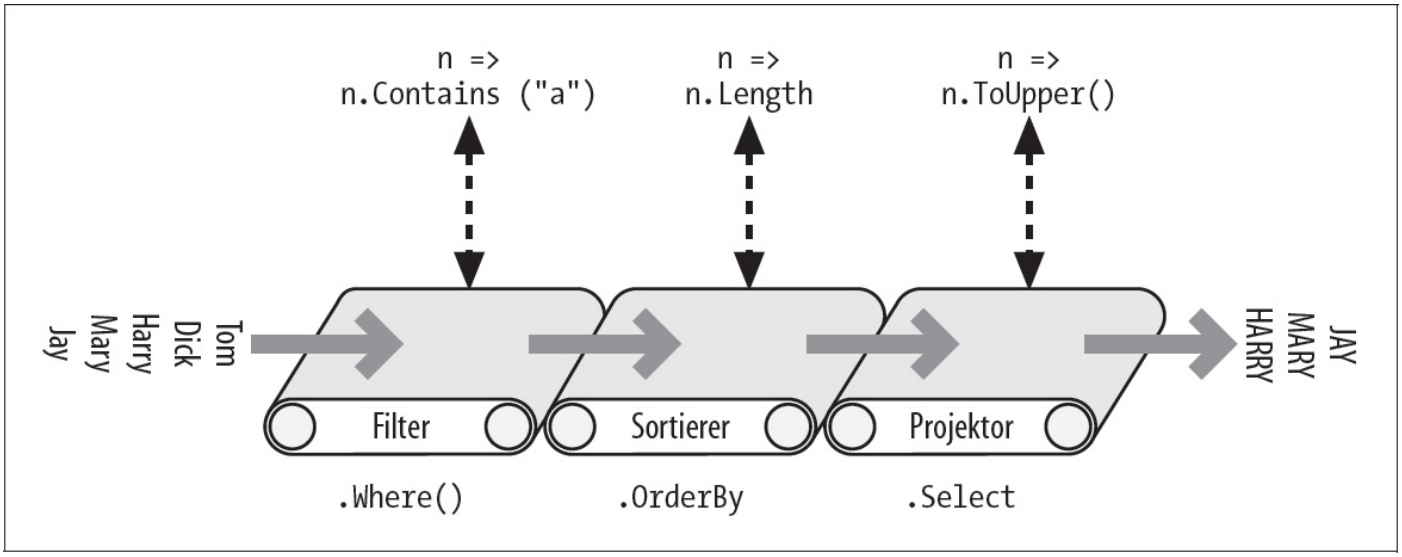

Abbildung 6: Verketten von Abfrageoperatoren

Die verzögerte Ausführung wird von allen Operatoren unterstützt, daher wird nichts gefiltert, sortiert oder projiziert, bevor die Abfrage tatsächlich enumeriert wird.

Abfrageausdrücke

Bislang haben wir Abfragen geschrieben, indem wir die Erweiterungsmethoden aus der Klasse Enumerable aufgerufen haben. In diesem Buch bezeichnen wir das als *fließende Syntax*. C# bietet außerdem eine spezielle Sprachunterstützung zum Schreiben von Abfragen, die den Titel *Abfrageausdrücke* trägt. Hier ist die vorangegangene Abfrage in Form eines Abfrageausdrucks:

```
IEnumerable<string> query =
    from n in names
    where n.Contains ("a")
    orderby n.Length
    select n.ToUpper();
```

Ein Abfrageausdruck beginnt immer mit einer from-Klausel und endet entweder mit einer select- oder einer group-Klausel. Die from-Klausel deklariert eine *Bereichsvariable* (in diesem Fall n), die die Eingabe-Collection traversiert – wie bei foreach. Abbildung 7 zeigt die vollständige Syntax.

Wenn Sie mit SQL vertraut sind, wird Ihnen die Query-Syntax von LINQ – mit der from-Klausel an erster und der select-Klausel an letzter Stelle – bizarr vorkommen. Die Query-Syntax ist jedoch tatsächlich logischer, da die Klauseln *in der Reihenfolge erscheinen, in der sie ausgeführt werden*. Damit kann Visual Studio Ihnen beim Tippen sinnvolle IntelliSense-Hinweise geben und auch die Scoping-Regeln für Unterabfragen vereinfachen.

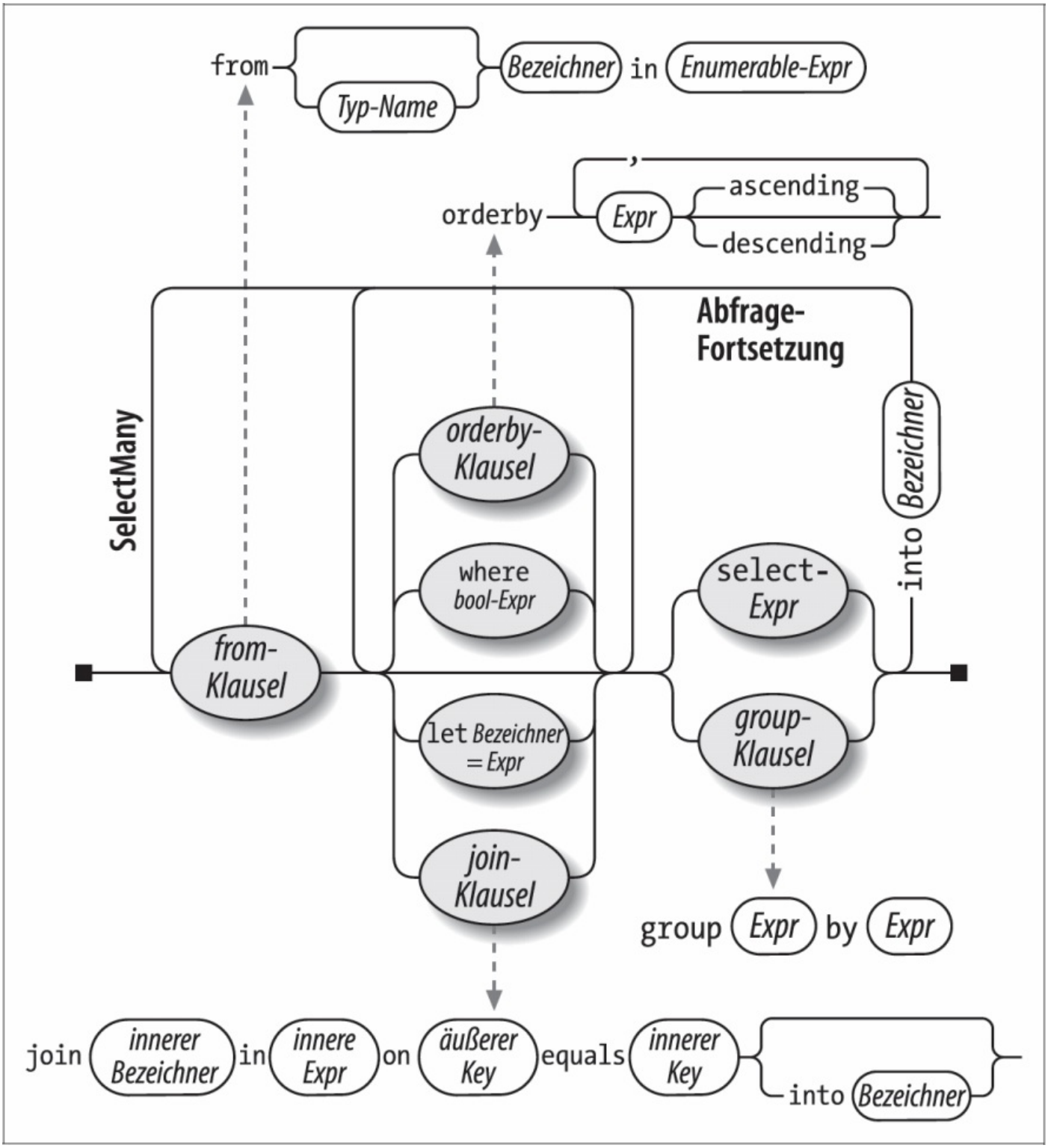

Abbildung 7: Syntax für Abfrageausdrücke

Der Compiler verarbeitet den Abfrageausdruck, indem er ihn in die fließende Syntax übersetzt. Das macht er auf eine recht mechanische Weise – ganz ähnlich, wie er foreach-Anweisungen in Aufrufe von GetEnumerator und MoveNext übersetzt:

```
IEnumerable<string> query = names

    .Where (n => n.Contains ("a"))
```

```
.OrderBy (n => n.Length)

.Select (n => n.ToUpper());
```

Die Operatoren Where, OrderBy und Select werden dann nach denselben Regeln aufgelöst, die gelten würden, wenn die Abfrage in Lambda-Syntax geschrieben wäre. In diesem Fall binden sie an Erweiterungsmethoden der Klasse Enumerable (vorausgesetzt, Sie haben den Namensraum System.Linq importiert), da names das Interface IEnumerable<string> implementiert. Der Compiler bevorzugt aber beim »Übersetzen« der Query-Syntax nicht unbedingt die Klasse Enumerable. Sie können es sich so vorstellen, dass der Compiler ganz mechanisch die Wörter Where, OrderBy und Select in die Anweisung einfügt und das Ganze dann so kompiliert, als hätten Sie die Methodennamen selbst eingetippt. Damit bleibt die Flexibilität beim Auflösen bestehen – die Operatoren in LINQ-to-SQL-Abfragen zum Beispiel binden stattdessen an die Erweiterungsmethoden der Klasse Queryable.

Abfrageausdrücke vs. fließende Syntax

Abfrageausdrücke und die fließende Syntax haben jeweils ihre Vorteile.

Abfrageausdrücke unterstützen nur eine kleine Untermenge der Abfrageoperatoren, und zwar folgende:

Where, Select, SelectMany

OrderBy, ThenBy, OrderByDescending, ThenByDescending

GroupBy, Join, GroupJoin

Bei Abfragen, die andere Operatoren nutzen, müssen Sie sie entweder komplett in fließender Syntax schreiben oder eine Abfrage mit gemischter Syntax erstellen, zum Beispiel so:

```
string[] names = { "Tom","Dick","Harry","Mary","Jay" };

IEnumerable<string> query =
    from   n in names
    where  n.Length == names.Min (n2 => n2.Length)
```

```
    select n;
```

Diese Abfrage liefert Namen zurück, deren Länge der der kürzesten Namen entspricht (»Tom« und »Jay«). Die Unterabfrage (in Fettdruck) ermittelt die minimale Länge der verschiedenen Namen, also 3. Wir müssen für die Unterabfrage fließende Syntax benutzen, weil der Operator Min in der Abfragesyntax nicht unterstützt wird. Wir können aber immer noch für die äußere Abfrage die Abfragesyntax verwenden.

Der Hauptvorteil der Abfragesyntax ist, dass sie Abfragen radikal vereinfachen kann, die folgende Elemente enthalten:

- Eine let-Klausel, mit der neben der Bereichsvariablen eine neue Variable eingeführt wird.
- Mehrere Generatoren (SelectMany), gefolgt von einer äußeren Bereichsvariablenreferenz.
- Ein Join- oder GroupJoin-Äquivalent, gefolgt von einer äußeren Bereichsvariablenreferenz.

Das Schlüsselwort let

Das Schlüsselwort let führt neben der Bereichsvariablen eine neue Variable ein. Stellen Sie sich zum Beispiel vor, dass wir alle Namen ausgeben wollen, deren Länge ohne Vokale größer als zwei Zeichen ist:

```
string[] names = { "Tom","Dick","Harry","Mary","Jay" };

IEnumerable<string> query =
    from n in names
    let vowelless = Regex.Replace (n, "[aeiou]", "")
    where vowelless.Length > 2
    orderby vowelless
    select n + " - " + vowelless;
```

Die Ausgabe beim Enumerieren dieser Abfrage sieht so aus:

```
Dick - Dck

Harry - Hrry

Mary - Mry
```

Die let-Klausel führt für jedes Element eine Berechnung durch, ohne das ursprüngliche Element zu verlieren. In unserer Abfrage haben die folgenden Klauseln (where, orderby und select) Zugriff sowohl auf n als auch auf vowelless. Abfragen können mehrere let-Klauseln nutzen und diese bei zusätzlichen where- und join-Klauseln einstreuen.

Der Compiler übersetzt das Schlüsselwort let, indem er in einen temporären, anonymen Typ projiziert, der das ursprüngliche und das transformierte Element enthält:

```
IEnumerable<string> query = names

 .Select (n => new

    {

        n = n,

        vowelless = Regex.Replace (n, "[aeiou]", "")

    }

 )

 .Where (temp0 => (temp0.vowelless.Length > 2))

 .OrderBy (temp0 => temp0.vowelless)

 .Select (temp0 => ((temp0.n + " - ") + temp0.vowelless))
```

Abfragefortsetzungen

Wenn Sie nach einer select- oder group-Klausel noch Klauseln anhängen wollen,

müssen Sie das Schlüsselwort into nutzen, um die Abfrage »fortzusetzen«. Hier sehen Sie ein Beispiel dafür:

```
from c in "The quick brown tiger".Split()
select c.ToUpper() into upper
where upper.StartsWith ("T")
select upper

// Ergebnis: "THE", "TIGER"
```

Nach einer into-Klausel ist die vorherige Bereichsvariable nicht mehr gültig.

Der Compiler übersetzt Abfragen mit einem into-Schlüsselwort einfach in eine längere Kette von Operatoren:

```
"The quick brown tiger".Split()
    .Select (c => c.ToUpper())
    .Where (upper => upper.StartsWith ("T"))
```

(Er lässt das letzte Select(upper=>upper) weg, da es redundant ist.)

Mehrere Generatoren

Eine Abfrage kann mehrere Generatoren (from-Klauseln) enthalten:

```
int[] numbers = { 1, 2, 3 };
string[] letters = { "a", "b" };

IEnumerable<string> query = from n in numbers
                            from l in letters
```

```
                                    select n.ToString() + l;
```

Das Ergebnis ist ein Kreuzprodukt, wie Sie es bei verschachtelten foreach-Schleifen erhalten würden:

```
"1a", "1b", "2a", "2b", "3a", "3b"
```

Wenn es in einer Abfrage mehr als eine from-Klausel gibt, erzeugt der Compiler einen Aufruf von SelectMany:

```
IEnumerable<string> query = numbers.SelectMany (
    n => letters,
    (n, l) => (n.ToString() + l));
```

SelectMany führt verschachtelte Schleifendurchläufe aus. Es enumeriert jedes Element in der Quell-Collection (numbers) und wandelt jedes Element mit dem ersten Ausdruck um (letters). Damit wird eine Sequenz aus *Subsequenzen* erzeugt, die dann enumeriert werden. Die endgültigen Ausgabeelemente werden durch den zweiten Ausdruck bestimmt (n.ToString( )+l).

Wenn Sie anschließend eine where-Klausel anwenden, können Sie das Kreuzprodukt filtern und ein Ergebnis projizieren, das einem *Join* ähnelt:

```
string[] players = { "Tom", "Jay", "Mary" };

IEnumerable<string> query =
    from name1 in players
    from name2 in players
    where name1.CompareTo (name2) < 0
    orderby name1, name2
    select name1 + " vs " + name2;
```

RESULT: { "Jay vs Mary", "Jay vs Tom", "Mary vs Tom" }

Das Übersetzen dieser Abfrage in die fließende Syntax ist natürlich deutlich komplexer, da eine temporäre anonyme Projektion erforderlich ist. Die Möglichkeit, diese Übersetzung automatisch durchführen zu lassen, ist einer der Hauptvorteile der Abfragesyntax.

Der Ausdruck im zweiten Generator darf die erste Bereichsvariable nutzen:

```
string[] fullNames =
    { "Anne Williams", "John Fred Smith", "Sue Green" };

IEnumerable<string> query =
    from fullName in fullNames
    from name in fullName.Split()
    select name + " kommt aus " + fullName;
```

Anne kommt aus Anne Williams

Williams kommt aus Anne Williams

John kommt aus John Fred Smith

Das funktioniert, weil der Ausdruck fullName.Split eine *Sequenz* ausgibt (ein Array mit Strings).

In Datenbankabfragen werden sehr häufig mehrere Generatoren eingesetzt, um Eltern-Kind-Beziehungen zu ebnen und manuelle Joins durchzuführen.

Verknüpfen

LINQ bietet drei *Verknüpfungsoperatoren*, von denen Join und GroupJoin am wichtigsten sind, die schlüsselbasierte Joins durchführen. Join und GroupJoin unterstützen nur eine Untermenge der Funktionalität, die Sie mit mehreren

Generatoren/SelectMany erhalten können, sind bei lokalen Abfragen aber leistungsfähiger, weil sie eine Hashtable-basierte Suchstrategie nutzen und auf geschachtelte Schleifen verzichten. (Bei LINQ-to-SQL- und Entity-Framework-Abfragen bieten die Verknüpfungsoperatoren keine Vorteile gegenüber mehreren Generatoren.)

Join und GroupJoin unterstützen nur *Equi-Joins* (das heißt, die Verknüpfungsbedingung muss den Gleichheitsoperator nutzen). Es gibt dabei zwei Methoden: Join und GroupJoin. Join erzeugt eine flache Ergebnismenge, während GroupJoin eine hierarchische Ergebnismenge liefert.

Die Abfrageausdruckssyntax für einen flachen Join sieht so aus:

```
from äußere-Var in äußere-Sequenz
join innere-Var in innere-Sequenz
    on äußerer-Schlüssel equals innerer-Schlüssel
```

Nehmen wir zum Beispiel die folgenden Collections:

```
var customers = new[]
{
    new { ID = 1, Name = "Tom" },
    new { ID = 2, Name = "Dick" },
    new { ID = 3, Name = "Harry" }
};
var purchases = new[]
{
        new { CustomerID = 1, Product = "Haus" },
        new { CustomerID = 2, Product = "Boot" },
        new { CustomerID = 2, Product = "Auto" },
```

```
        new { CustomerID = 3, Product = "Urlaub" }
    };
```

Damit können wir auf folgende Weise eine Verknüpfung durchführen:

```
    IEnumerable<string> query =
        from c in customers join p in purchases on c.ID equals p.
    CustomerID
        select c.Name + " hat gekauft: " + p.Product;
```

Der Compiler übersetzt das in

```
    customers.Join (              // äußere Collection
        purchases,                // innere Collection
        c => c.ID,                // äußerer Schlüssel
        p => p.CustomerID,        // innerer Schlüssel
        (c, p) =>                 // Ergebnisselektor
            c.Name + " hat gekauft: " + p.Product
     );
```

Und so sieht das Ergebnis aus:

```
    Tom hat gekauft: Haus
    Dick hat gekauft: Boot
    Dick hat gekauft: Auto
```

Harry hat gekauft: Urlaub

Bei lokalen Sequenzen sind Join und GroupJoin effizienter beim Verarbeiten großer Collections als SelectMany, weil sie zunächst die innere Sequenz in einen Schlüssel-Hashtable-basierten Lookup laden. Bei einer Datenbankabfrage können Sie das gleiche Ergebnis genauso effizient erreichen, wenn Sie so vorgehen:

```
from c in customers
from p in purchases
where c.ID == p.CustomerID
select c.Name + " bought a " + p.Product;
```

GroupJoin

GroupJoin macht dasselbe wie Join, liefert aber keine flache Ergebnismenge zurück, sondern ein hierarchisches Ergebnis, das nach den einzelnen äußeren Elementen gruppiert ist.

Die Syntax für GroupJoin ist die gleiche wie für Join, nur dass ihr das Schlüsselwort into folgt. Ein einfaches Beispiel mit den Collections customers und purchases, die wir im vorigen Abschnitt angelegt haben, sieht so aus:

```
IEnumerable<IEnumerable<Purchase>> query =
    from c in customers
    join p in purchases on c.ID equals p.CustomerID
    into custPurchases
    select custPurchases;    // custPurchases ist eine Sequenz
```

> Eine into-Klausel wird nur dann in einen GroupJoin umgewandelt, wenn sie direkt auf eine join-Klausel folgt. Nach einer select- oder group-Klausel bedeutet sie eine *Abfragefortsetzung*. Die zwei Anwendungsarten des Schlüsselworts into sind ziemlich verschieden, sie haben aber ein

Feature gemeinsam: Beide führen eine neue Abfragevariable ein.

Das Ergebnis ist eine Sequenz mit Sequenzen, die wir wie folgt enumerieren könnten:

```
foreach (IEnumerable<Purchase> purchaseSequence in query)
    foreach (Purchase p in purchaseSequence)
        Console.WriteLine (p.Description);
```

Das ist natürlich nicht sehr nützlich, da outerSeq keine Referenz auf den äußeren Kunden enthält. Im Allgemeinen werden Sie die äußere Bereichsvariable in der Projektion referenzieren:

```
from c in customers
join p in purchases on c.ID equals p.CustomerID
into custPurchases
select new { CustName = c.Name, custPurchases };
```

Wir könnten das gleiche Ergebnis (allerdings für lokale Abfragen weniger effektiv) erreichen, indem wir in einen anonymen Typ projizieren, der eine Unterabfrage enthält:

```
from c in customers
select new
{
    CustName = c.Name,
    custPurchases =
```

```
        purchases.Where (p => c.ID == p.CustomerID)
}
```

Zip

Zip ist der einfachste Verknüpfungsoperator. Er enumeriert zwei Sequenzen nebeneinander (wie ein Reißverschluss) und liefert eine Sequenz, die auf der Anwendung einer Funktion auf die jeweiligen Elementpaare basiert. Hier sehen Sie ein Beispiel dafür:

```
int[] numbers = { 3, 5, 7 };
string[] words = { "drei", "fünf", "sieben", "ignoriert" };
IEnumerable<string> zip =
    numbers.Zip (words, (n, w) => n + "=" + w);
```

Dieser Ausdruck erzeugt eine Sequenz mit den folgenden Elementen:

```
3=drei
5=fünf
7=sieben
```

Zusätzliche Elemente in einer der Eingabesequenzen werden ignoriert. Zip wird bei Abfragen von Datenbanken nicht unterstützt .

Ordnen

Das Schlüsselwort orderby sortiert eine Sequenz. Sie können beliebig viele Ausdrücke angeben, nach denen sortiert werden soll:

```
string[] names = { "Tom","Dick","Harry","Mary","Jay" };
IEnumerable<string> query = from n in names
```

```
                    orderby n.Length, n

                    select n;
```

Damit wird zuerst nach der Länge und dann nach dem Namen sortiert, sodass das Ergebnis so aussieht:

```
Jay, Tom, Dick, Mary, Harry
```

Der Compiler übersetzt den ersten orderby-Ausdruck in einen Aufruf von OrderBy, die folgenden werden dann zu ThenBy umgewandelt:

```
IEnumerable<string> query = names
  .OrderBy (n => n.Length)
  .ThenBy (n => n)
```

Der Operator ThenBy*ergänzt* die vorige Sortierung, *ersetzt* sie aber nicht.

Sie können das Schlüsselwort descending in den orderby-Ausdrücken nutzen:

```
orderby n.Length descending, n
```

Das wird umgewandelt in:

```
.OrderByDescending (n => n.Length).ThenBy (n => n)
```

Die Ordnungsoperatoren liefern einen erweiterten Typ von IEnumerable<T> namens IOrderedEnumerable<T> zurück. Dieses Interface definiert die zusätzliche Funktionalität, die vom ThenBy-Operator benötigt wird.

Gruppieren

GroupBy arrangiert eine flache Eingabesequenz in Sequenzen aus *Gruppen*. So gruppiert zum Beispiel der folgende Code eine Sequenz aus Namen nach ihrer Länge:

```
string[] names = { "Tom","Dick","Harry","Mary","Jay" };

var query = from name in names
                    group name by name.Length;
```

Der Compiler wandelt das um in:

```
IEnumerable<IGrouping<int,string>> query =
    names.GroupBy (name => name.Length);
```

So kann man das Ergebnis enumerieren:

```
foreach (IGrouping<int,string> grouping in query)
{
    Console.Write ("\r\n Length=" + grouping.Key + ":");
    foreach (string name in grouping)
        Console.Write (" " + name);
}

Length=3: Tom Jay
Length=4: Dick Mary
Length=5: Harry
```

Enumerable.GroupBy liest die Eingabeelemente in ein temporäres Dictionary mit Listen ein, sodass alle Elemente mit demselben Schlüssel in derselben Unterliste landen. Dann gibt die Methode eine Sequenz aus *Gruppierungen* aus. Eine Gruppierung ist eine Sequenz mit einer Eigenschaft Key:

```
public interface IGrouping <TKey,TElement>
    : IEnumerable<TElement>, IEnumerable
{
    // Key gilt für die ganze Untersequenz.
    TKey Key { get; }
}
```

Standardmäßig handelt es sich bei den Elementen in jeder Gruppierung um die nicht transformierten Eingabeelemente, sofern Sie kein Argument elementSelector angeben. Durch den folgenden Code wird jedes Eingabeelement in Großbuchstaben umgewandelt:

```
from name in names
group name.ToUpper() by name.Length
```

Das wird umgewandelt in

```
names.GroupBy (
    name => name.Length,
    name => name.ToUpper() )
```

Die Untercollections werden nicht in der Reihenfolge ihrer Schlüssel ausgegeben. GroupBy führt keine Sortierung durch (es behält sogar die ursprüngliche Reihenfolge bei). Um sortieren zu können, müssen Sie einen Operator OrderBy hinzufügen (was bedeutet, dass Sie zunächst eine into-Klausel anhängen müssen, da group by

normalerweise eine Abfrage abschließt):

```
from name in names
group name.ToUpper() by name.Length into grouping
orderby grouping.Key
select grouping
```

Abfragefortsetzungen werden häufig in einer Abfrage mit group by genutzt. Die nächste Abfrage filtert Gruppen heraus, die aus genau zwei Einträgen bestehen:

```
from name in names
group name.ToUpper() by name.Length into grouping
where grouping.Count() == 2
select grouping
```

Ein where nach einem group by entspricht HAVING in SQL. Es wird für jede Untersequenz bzw. Gruppierung als Ganzes angewendet, statt auf die einzelnen Elemente zu wirken.

OfType und Cast

OfType und Cast erwarten eine nichtgenerische Collection IEnumerable und geben eine generische Sequenz IEnumerable<T> zurück, die Sie dann im Weiteren abfragen können:

```
var classicList = new System.Collections.ArrayList();
classicList.AddRange ( new int[] { 3, 4, 5 } );
IEnumerable<int> sequence1 = classicList.Cast<int>();
```

Das ist nützlich, da Sie damit Collections abfragen können, die vor C# 2.0 geschrieben wurden (als IEnumerable<T> ins Spiel kam), zum Beispiel ControlCollection in System.Windows.Forms.

Cast und OfType unterscheiden sich in ihrem Verhalten, wenn sie auf ein Eingabeelement treffen, dessen Typ nicht kompatibel ist: Cast wirft eine Exception, während OfType das inkompatible Element ignoriert.

Die Regeln der Elementkompatibilität entsprechen denen des is-Operators von C#. So sieht die interne Implementierung von Cast aus:

```
public static IEnumerable<TSource> Cast <TSource>
                        (IEnumerable source)
{
    foreach (object element in source)
        yield return (TSource)element;
}
```

C# unterstützt den Operator Cast in Abfrageausdrücken. Fügen Sie einfach direkt nach dem Schlüsselwort from den Elementtyp ein:

```
from int x in classicList ...
```

Das wird in Folgendes übersetzt:

```
from x in classicList.Cast <int>() ...
```

Die dynamische Bindung

Dynamische Bindung schiebt die *Bindung* – den Vorgang der Auflösung von Typen, Membern und Operationen – von der Kompilierzeit bis zur Laufzeit hinaus. Dynamische Bindung wurde in C# 4.0 eingeführt und ist praktisch, wenn Sie zur Kompilierzeit wissen, dass eine bestimmte Funktion, ein bestimmtes Member oder eine bestimmte Operation existiert, während der Compiler es nicht weiß. Das tritt beispielsweise bei der Interaktion mit dynamischen Sprachen (wie IronPython), bei COM und in anderen Szenarien ein, bei denen Sie ansonsten Reflection nutzen würden.

Ein dynamischer Typ wird mit dem kontextabhängigen Schlüsselwort dynamic deklariert:

```
dynamic d = GetSomeObject();

d.Quack();
```

Ein dynamischer Typ sagt dem Compiler, dass er sich entspannen soll: »Wir gehen davon aus, dass der Laufzeittyp von d eine Quack-Methode bietet wird. Wir können das statisch einfach nur nicht belegen.« Da d dynamisch ist, schiebt der Compiler die Bindung von Quack an d bis zur Laufzeit hinaus. Um das zu begreifen, müssen Sie verstehen, welcher Unterschied zwischen *statischer* und *dynamischer* Bindung besteht.

Statische Bindung vs. dynamische Bindung

Das übliche Beispiel für die Bindung ist die Zuordnung eines Namens zu einer bestimmten Funktion bei der Kompilierung eines Ausdrucks. Wenn der Compiler den folgenden Ausdruck kompilieren soll, muss er die Implementierung einer Methode namens Quack finden:

```
d.Quack();
```

Nehmen wir an, der statische Typ von d ist Duck:

```
Duck d = ...
```

```
d.Quack();
```

Im einfachsten Fall führt der Compiler die Bindung durch, indem er nach einer parameterlosen Methode mit dem Namen Quack auf Duck sucht. Kann er keine finden, weitet er seine Suche auf Methoden aus, die optionale Parameter erwarten, Methoden in Basisklassen von Duck und Erweiterungsmethoden, die Duck als ersten Parameter erwarten. Wenn keine passende Methode gefunden wird, erhalten Sie einen Kompilierungsfehler. Ganz gleich, welche Methode gebunden wird – entscheidend ist, dass das Binden durch den Compiler erfolgt und vollständig von einer statischen Kenntnis der Typen (hier d) der Operanden abhängig ist. Das ist das, was diese Art der Bindung *statisch* macht.

Ändern wir nun den statischen Typ von d in object:

```
object d = ...

d.Quack();
```

Dieser Aufruf von Quack führt zu einem Kompilierungsfehler, da der in d gespeicherte Wert zwar eine Methode namens Quack besitzen kann, der Compiler das aber nicht wissen kann, weil seine einzige Information der Typ der Variablen ist, der in diesem Fall object ist. Ändern wir jetzt einmal den statischen Typ von d in dynamic:

```
dynamic d = ...

d.Quack();
```

Der Typ dynamic ist wie object – er sagt ebenso wenig über den (tatsächlichen) Typ aus. Der Unterschied ist, dass er Ihnen gestattet, ihn auf Weisen zu verwenden, die zur Kompilierungszeit nicht bekannt sind. Ein dynamisches Objekt bindet zur Laufzeit auf Basis der Laufzeittypen, nicht auf Basis des Kompilierzeittyps. Wenn der Compiler einen dynamisch gebundenen Ausdruck sieht (was in der Regel ein Ausdruck ist, der einen Wert des Typs dynamic einschließt), verpackt er diesen Ausdruck einfach so, dass die Bindung später zur Laufzeit erfolgen kann.

Wenn ein dynamisches Objekt IDynamicMetaObjectProvider implementiert, wird zur Laufzeit dieses Interface genutzt, um die Bindung durchzuführen. Ist das nicht der Fall, erfolgt die Bindung fast auf die gleiche Weise, auf die sie erfolgt wäre, wenn der Compiler den Laufzeittyp des dynamischen Objekts gekannt hätte. Diese beiden Optionen werden als *benutzerdefinierte Bindung* und *Sprachbindung* bezeichnet.

Benutzerdefinierte Bindung

Benutzerdefinierte Bindung erfolgt, wenn ein dynamisches Objekt IDynamicMetaObjectProvider (IDMOP) implementiert. Obwohl Sie IDMOP auf Typen implementieren können, die Sie in C# schreiben – und das auch sehr praktisch sein kann –, ist es dennoch am häufigsten der Fall, dass Sie ein IDMOP-Objekt von einer dynamischen Sprache erhalten haben, die in .NET auf der *Dynamic Language Runtime* (DLR) implementiert, ist wie IronPython oder IronRuby. Objekte von diesen Sprachen implementieren IDMOP implizit als Mittel zur unmittelbaren Steuerung der Operationen, die von ihnen ausgeführt werden. Hier ist ein einfaches Beispiel dafür:

```
using System;
using System.Dynamic;

public class Test
{
    static void Main()
    {
        dynamic d = new Duck();
        d.Quack();       // Quack wurde gerufen
        d.Waddle();      // Waddle wurde gerufen
    }
}
public class Duck : DynamicObject
{
    public override bool TryInvokeMember (
        InvokeMemberBinder binder, object[] args,
```

```
            out object result)
        {
            Console.WriteLine (binder.Name + "wurde gerufen");
            result = null;
            return true;
        }
    }
```

Die Klasse Duck hat eigentlich keine Quack-Methode. Stattdessen nutzt sie benutzerdefinierte Bindung, um alle Methodenaufrufe abzufangen und zu interpretieren.

Sprachbindung

Sprachbindung findet statt, wenn ein dynamisches Objekt IDynamicMetaObjectProvider nicht implementiert. Sprachbindung ist praktisch, wenn man mit schlecht entworfenen Typen arbeiten muss oder eine Möglichkeit sucht, die Grenzen des .NET-Typsystems auszuhebeln. Ein typisches Problem bei der Verwendung numerischer Typen ist, dass diese keine einheitliche Schnittstelle haben. Wir haben gesehen, dass Methoden dynamisch gebunden werden können – und das gilt auch für Operatoren:

```
static dynamic Mean (dynamic x, dynamic y) => (x + y) / 2;

static void Main()
{
    int x = 3, y = 4;
    Console.WriteLine (Mean (x, y));
}
```

Der Vorteil ist offensichtlich: Sie müssen diesen Code nicht mehr einzeln für jeden numerischen Typ schreiben. Aber Sie verlieren damit die statische Typsicherheit und riskieren Laufzeitfehler anstelle von Kompilierungsfehlern.

Dynamische Bindung umgeht die statische Typsicherheit, aber nicht die Laufzeit-Typsicherheit. Anders als bei Reflection können Sie mit dynamischer Bindung die Zugriffsregeln für Member nicht aushebeln.

Die Sprachbindung zur Laufzeit wurde so gestaltet, dass sie sich der statischen Bindung so ähnlich wie möglich verhält, wenn die Laufzeittypen dynamischer Objekte bei der Kompilierung bekannt gewesen wären. In unserem vorangegangenen Beispiel hätte sich unser Programm auf genau die gleiche Weise verhalten, wenn wir Mean so definiert hätten, dass es mit dem Typ int funktioniert. Die auffälligste Ausnahme bei der Gleichheit von statischer und dynamischer Bindung taucht bei Erweiterungsmethoden auf, mit denen wir uns in »Nicht aufrufbare Funktionen« auf Seite 187 befassen werden.

Dynamische Bindung bringt Leistungseinbußen mit sich. Aufgrund der Caching-Mechanismen der DLR werden wiederholte Aufrufe dynamischer Ausdrücke jedoch optimiert – das ermöglicht Ihnen, dynamische Ausdrücke effizient in Schleifen aufzurufen. Diese Optimierung reduziert den üblichen Nachteil für einen einfachen dynamischen Ausdruck auf moderner Hardware auf weniger als 100 Nanosekunden.

RuntimeBinderException

Wenn ein Member nicht gebunden werden kann, wird eine RuntimeBinderException ausgelöst. Diese können Sie als eine Art Compilerfehler zur Laufzeit betrachten:

```
dynamic d = 5;

d.Hello();      // löst RuntimeBinderException aus
```

Die Exception wird ausgelöst, weil der Typ int keine Hello-Methode hat.

Laufzeitdarstellung von dynamic

Es gibt eine tiefgehende Äquivalenz zwischen den Typen dynamic und object. Die Laufzeitumgebung behandelt den folgenden Ausdruck als true:

```
typeof (dynamic) == typeof (object)
```

Dieses Prinzip erstreckt sich auch auf Collection- und Array-Typen:

```
typeof (List<dynamic>) == typeof (List<object>)

typeof (dynamic[]) == typeof (object[])
```

Wie eine Objektreferenz kann eine dynamische Referenz auf ein Objekt eines beliebigen Typs (außer Zeigertypen) zeigen:

```
dynamic x = "Hallo";

Console.WriteLine (x.GetType().Name); // String


x = 123; // kein Fehler (obwohl die gleiche Variable

         // genutzt wird)

Console.WriteLine (x.GetType().Name); // Int32
```

Strukturell gibt es keinen Unterschied zwischen einer Objektreferenz und einer dynamischen Referenz. Eine dynamische Referenz macht einfach nur dynamische Operationen auf dem Objekt möglich, auf das sie zeigt. Sie können von object in dynamic umwandeln, um beliebige dynamische Operationen an einem object auszuführen:

```
object o = new System.Text.StringBuilder();

dynamic d = o;

d.Append ("Hallo");

Console.WriteLine (o);    // Hallo
```

Dynamische Konvertierungen

Der Typ dynamic bietet implizite Konvertierungen in alle anderen Typen und zurück. Eine Konvertierung ist dann erfolgreich, wenn der Laufzeittyp des dynamischen Objekts implizit in den statischen Zieltyp konvertierbar ist.

Das folgende Beispiel löst eine RuntimeBinderExceptionaus, weil ein int nicht implizit in einen short umgewandelt werden kann:

```
int i = 7;

dynamic d = i;

long l = d;      // Okay - implizite Konvertierung klappt

short j = d;     // RuntimeBinderException
```

var vs. dynamic

Die Typen var und dynamic weisen eine oberflächliche Ähnlichkeit auf, aber die Unterschiede gehen tief:

- var sagt: »Lass den *Compiler* den Typ herausfinden.«
- dynamic sagt: »Lass die *Laufzeit* den Typ herausfinden.«

Zur Illustration:

```
dynamic x = "Hallo";  // statischer Typ ist dynamic

var y = "Hallo";      // statischer Typ ist string

int i = x;            // Laufzeitfehler

int j = y;            // Kompilierungsfehler
```

Dynamische Ausdrücke

Felder, Eigenschaften, Methoden, Konstruktoren, Indexer, Operatoren und Konvertierungen können alle dynamisch aufgerufen werden.

Es ist verboten, das Ergebnis eines dynamischen Ausdrucks mit dem Rückgabetyp void weiterzuverarbeiten – genau wie bei statisch typisierten Ausdrücken. Der Unterschied ist, dass der Fehler erst zur Laufzeit auftritt.

Ausdrücke mit dynamischen Operanden sind üblicherweise selbst dynamisch, da die Wirkung der fehlenden Typinformationen im Ausdruck weitergereicht wird:

```
dynamic x = 2;

var y = x * 3;       // Der statische Typ von y ist dynamisch.
```

Es gibt einige offensichtliche Ausnahmen von dieser Regel. Erstens erhält man einen statischen Ausdruck, wenn man einen dynamischen Ausdruck auf einen statischen Typ castet, und zweitens liefern Konstruktoraufrufe immer statische Ausdrücke, selbst wenn sie mit dynamischen Argumenten aufgerufen wurden.
Außerdem gibt es ein paar Randfälle, bei denen ein Ausdruck, der einen dynamischen Operanden enthält, statisch ist, unter anderem die Übergabe eines Index an ein Array und Ausdrücke zur Delegate-Erstellung.

Die dynamische Auflösung überladener Member

Der übliche Anwendungsfall für dynamic schließt einen dynamischen Empfänger ein. Das heißt, dass ein dynamisches Objekt der Empfänger eines dynamischen Funktionsaufrufs ist:

```
dynamic x = ...;

x.Foo (123);         // x ist der Empfänger
```

Aber statische Bindung ist nicht auf Empfänger beschränkt: Methodenargumente können ebenfalls dynamisch gebunden werden. Der Aufruf einer Funktion mit dynamischen Argumenten bewirkt, dass die Auflösung von Überladungen von der Kompilierung bis zur Laufzeit aufgeschoben wird:

```
static void Foo (int x)    => Console.WriteLine ("1");

static void Foo (string x) => Console.WriteLine ("2");


static void Main()
```

```
{
    dynamic x = 5;
    dynamic y = "Wassermelone";

    Foo (x);    // 1
    Foo (y);    // 2
}
```

Die Überladungsauflösung zur Laufzeit wird auch als *Multiple Dispatch* bezeichnet und ist bei der Implementierung von Entwurfsmustern wie dem *Besuchermuster* hilfreich.

Wenn kein dynamischer Empfänger beteiligt ist, kann der Compiler statisch eine einfache Prüfung darauf durchführen, ob der dynamische Aufruf gelingen wird: Er prüft einfach, ob es eine Funktion mit dem richtigen Namen und der richtigen Anzahl von Argumenten gibt. Wird kein Kandidat gefunden, erhalten Sie einen Kompilierungsfehler.

Wenn eine Funktion mit einer Mischung aus dynamischen und statischen Argumenten aufgerufen wird, spiegelt die endgültige Entscheidung eine Mischung dynamischer und statischer Bindungsentscheidungen:

```
static void X(object x, object y) => Console.Write("oo");
static void X(object x, string y) => Console.Write("os");
static void X(string x, object y) => Console.Write("so");
static void X(string x, string y) => Console.Write("ss");

static void Main()
{
    object o = "Hello";
```

```
        dynamic d = "Goodbye";

        X (o, d);               // os

    }
```

Der Aufruf von X(o,d) wird dynamisch gebunden, weil eines seiner Argumente, d, dynamic ist. Aber da o statisch bekannt ist, nutzt der Bindungsvorgang – obwohl er dynamisch erfolgt – diese Informationen. Hier wählt die Überladungsauflösung aufgrund des statischen Typs von o und des Laufzeittyps von d die zweite Implementierung von X. Anders formuliert: Der Compiler verhält sich so statisch, wie es eben geht .

Nicht aufrufbare Funktionen

Manche Funktionen können dynamisch nicht aufgerufen werden:

- Erweiterungsmethoden (über die Erweiterungsmethodensyntax)
- beliebige Member eines Interface (über das Interface)
- von einer Subklasse verborgene Basisklassen-Member

Das liegt daran, dass die dynamische Bindung zwei Informationselemente benötigt: den Namen der aufzurufenden Funktion und das Objekt, auf dem die Funktion aufgerufen werden soll. Aber bei den drei nicht aufrufbaren Konstrukten ist jeweils ein *zusätzlicher Typ* beteiligt, der nur zur Kompilierungszeit bekannt ist. Und es gibt keine Möglichkeit, diese zusätzlichen Typ dynamisch anzugeben.

Wenn Sie Erweiterungsmethoden aufrufen, ist dieser zusätzliche Typ eine Erweiterungsklasse, die implizit auf Basis der using-Direktiven in Ihrem Quellcode gewählt wird (die nach der Kompilierung verschwinden). Wenn Sie Member über ein Interface aufrufen, wird der zusätzliche Typ über einen impliziten oder expliziten Cast kommuniziert. (Bei expliziter Implementierung ist es in der Tat unmöglich, ein Member ohne einen Cast auf das Interface aufzurufen.) Ähnlich verhält es sich, wenn Sie versuchen, ein verborgenes Basisklassen-Member aufzurufen: Sie müssen einen zusätzlichen Typ angeben, entweder über einen Cast oder über das Schlüsselwort base – und diese zusätzlichen Typinformationen sind zur Laufzeit verloren.

Überladen von Operatoren

Operatoren können überladen werden, um eine natürlichere Syntax für eigene Typen anzubieten. Das Überladen von Operatoren ist dann am sinnvollsten, wenn Sie eigene Structs implementieren, die halbwegs primitive Datentypen repräsentieren. So ist zum Beispiel ein eigener numerischer Typ ein exzellenter Kandidat für das Überladen von Operatoren.

Die symbolischen Operatoren, die sich überladen lassen, sind:

```
+  -  *  /  ++  --  !  ~  %  &  |  ^

==  !=  <  <<  >>  >
```

Implizite und explizite Konvertierungen können genauso überschrieben werden (mit den Schlüsselwörtern implicit und explicit) wie die Literale true und false und die unären Operatoren + und -.

Die zusammengesetzten Zuweisungsoperatoren (z. B. += oder /=) werden automatisch überschrieben, wenn Sie die nicht zusammengesetzten Operatoren überschreiben (z. B. + oder /).

Operatorfunktion

Ein Operator wird überladen, indem man eine *Operatorfunktion* deklariert. Eine Operatorfunktion muss statisch sein, und mindestens einer der Operanden muss dem Typ entsprechen, in dem die Operatorfunktion deklariert wird.

Im folgenden Beispiel definieren wir ein Struct namens Note, das eine Musiknote repräsentiert, und überladen dann den Operator +:

```
public struct Note
{
    int value;

    public Note (int semitonesFromA)
```

```
        => value = semitonesFromA;

    public static Note operator + (Note x, int semitones)
    {
        return new Note (x.value + semitones);
    }
}
```

Durch dieses Überladen können wir nun ein int zu einer Note addieren:

```
Note B = new Note (2);
Note CSharp = B + 2;
```

Da wir + überschrieben haben, können wir auch += nutzen:

```
CSharp += 2;
```

Wie bei Methoden und Eigenschaften ist es seit C# 6 erlaubt, Operatorfunktionen, die aus einem einzelnen Ausdruck bestehen, mit der Expression-bodied Syntax kompakter zu schreiben:

```
public static Note operator + (Note x, int semitones)
    => new Note (x.value + semitones);
```

Überladen von Gleichheits- und Vergleichsoperatoren

Gleichheits- und Vergleichsoperatoren werden häufig überschrieben, wenn man Structs erstellt, und in seltenen Fällen auch beim Schreiben von Klassen. Es gibt spezielle Regeln und Pflichten beim Überladen dieser Operatoren:

Paare

Der C#-Compiler erwartet, dass bei Operatoren, die logische Paare bilden, beide definiert sind. Dabei handelt es sich um (== !=), (< >) und (<= >=).

Equals *und* GetHashCode

Wenn Sie == und != überladen, werden Sie normalerweise auch die Methoden Equals und GetHashCode überschreiben müssen, sodass Collections und Hashtabellen zuverlässig mit dem Typ arbeiten können.

IComparable *und* IComparable<T>

Wenn Sie > und < überladen, werden Sie im Allgemeinen auch IComparable und IComparable<T> implementieren.

Wenn wir das vorige Beispiel erweitern, können wir den Gleichheitsoperator von Note überladen:

```
public static bool operator == (Note n1, Note n2)

    => n1.value == n2.value;

  public static bool operator != (Note n1, Note n2)

    => !(n1.value == n2.value);


public override bool Equals (object otherNote)

{

    if (!(otherNote is Note)) return false;

    return this == (Note)otherNote;

}

// Den Hashcode des Werts nutzen.

public override int GetHashCode()

    => value.GetHashCode();
```

Eigene implizite und explizite Konvertierungen

Implizite und explizite Konvertierungen sind überladbare Operatoren. Diese Konvertierungen werden normalerweise überladen, um das Umwandeln eng verwandter Typen (wie zum Beispiel numerischer Typen) exakt und natürlich zu ermöglichen.

Wie bei der Betrachtung von Typen erläutert, ist das Prinzip bei impliziten Konvertierungen, dass sie immer erfolgreich sein sollten und während der Konvertierung keine Informationen verloren gehen sollen. Andernfalls sollten explizite Konvertierungen definiert werden.

Im folgenden Beispiel definieren wir Konvertierungen zwischen unserem musikalischen Typ Note und einem double (der die Frequenz der Note in Hertz repräsentiert):

```
...

// Konvertieren in Hertz

public static implicit operator double (Note x)

    => 440 * Math.Pow (2,(double) x.value / 12 );


// Konvertieren aus Hertz (in den nächsten Halbton)

public static explicit operator Note (double x)

    => new Note ((int) (0.5 + 12 * (Math.Log(x/440)

                        / Math.Log(2)) ));

...


Note n =(Note)554.37;  // explizite Konvertierung

double x = n;          // implizite Konvertierung
```

> Das ist ein etwas künstliches Beispiel: In der Praxis würden diese Konvertierungen wahrscheinlich besser über eine ToFrequency-Methode und eine (statische) FromFrequency-Methode implementiert.

Benutzerdefinierte Konvertierungen werden von den Operatoren as und is ignoriert.

Attribute

Sie sind schon darin geübt, Codeelemente eines Programms mit Modifikatoren zu versehen, zum Beispiel virtual oder ref. Diese Konstrukte sind in die Sprache eingebaut. *Attribute* sind ein Erweiterungsmechanismus, um Codeelementen (Assemblies, Typen, Membern, Rückgabewerten und Parametern) eigene Informationen hinzuzufügen. Diese Erweiterbarkeit ist für Services nützlich, die tief in das Typsystem eingebunden sind, ohne besondere Schlüsselwörter oder Konstrukte in C# erforderlich zu machen.

Ein gutes Szenario für Attribute ist die *Serialisierung* – der Prozess, bei dem eigene Objekte in ein bestimmtes Format umgewandelt und wieder zurückgeholt werden. In diesem Szenario kann ein Attribut für ein Feld die Umsetzung zwischen der Repräsentation eines Felds in C# und der im Speicherformat definieren.

Attributklassen

Ein Attribut wird durch eine Klasse definiert, die (direkt oder indirekt) von der abstrakten Klasse System.Attribute erbt. Um einem Codeelement ein Attribut zuzuweisen, geben Sie den Typnamen des Attributs vor dem Codeelement in eckigen Klammern an. So wird zum Beispiel durch die folgende Zeile der Klasse Foo das ObsoleteAttribute zugewiesen:

```
[ObsoleteAttribute]

public class Foo {...}
```

Dieses Attribut wird vom Compiler erkannt und sorgt für Warnungen, wenn ein Typ oder Member, der oder das als obsolet gekennzeichnet ist, verwendet wird. Die Konvention ist, alle Attributtypen mit dem Wort Attribute enden zu lassen. Das weiß C# und erlaubt Ihnen daher, das Suffix beim Zuweisen eines Attributs wegzulassen:

```
[Obsolete]

public class Foo {...}
```

ObsoleteAttribute ist ein Typ, der im Namensraum System wie folgt definiert ist (aus Gründen der Übersichtlichkeit etwas vereinfacht):

```
public sealed class ObsoleteAttribute : Attribute {...}
```

Benannte und positionelle Attributparameter

Attribute können Parameter haben. Im folgenden Beispiel wenden wir XmlElementAttribute auf eine Klasse an. Dieses Attribut sagt XmlSerializer (in System.Xml.Serialization), wie ein Objekt in XML dargestellt wird, und akzeptiert verschiedene *Attributparameter*. Das folgende Attribut bildet die Klasse CustomerEntity auf ein XML-Element namens Customer ab, das zum Namensraum http://oreilly.com gehört:

```
[XmlElement ("Customer", Namespace="http://oreilly.com")]

public class CustomerEntity { ... }
```

Attributparameter lassen sich immer einer von zwei Kategorien zuordnen: *positioniert* und *benannt*. Im vorigen Beispiel ist das erste Argument ein Positionsparameter, das zweite ein benannter Parameter. Positionsparameter entsprechen den Parametern der öffentlichen Konstruktoren des Attributtyps, benannte Parameter entsprechen öffentlichen Feldern oder öffentlichen Eigenschaften des Attributtyps.

Wenn Sie ein Attribut angeben, müssen Sie Positionsparameter angeben, die zu einem der Konstruktoren des Attributs passen. Benannte Parameter sind optional.

Attributziele

Das Ziel eines Attributs ist das Codeelement, vor dem es steht – normalerweise ein Typ oder ein Typ-Member. Sie können Attribute aber auch einer Assembly zuweisen. Dafür müssen Sie das Attributziel explizit angeben. Hier sehen Sie ein Beispiel für die Verwendung des Attributs CLSCompliant, um eine CLS-Konformität einer gesamten Assembly festzulegen:

```
[assembly:CLSCompliant(true)]
```

Angeben mehrerer Attribute

Es können mehrere Attribute für ein einzelnes Codeelement angegeben werden. Die

Attribute können entweder in einem einzelnen Paar eckiger Klammern angegeben werden (getrennt durch Kommata) oder jeweils extra in eckigen Klammern (oder als Kombination aus beidem). Die folgenden drei Beispiele sind semantisch identisch:

```
[Serializable, Obsolete, CLSCompliant(false)]

public class Bar {...}
```

```
[Serializable] [Obsolete] [CLSCompliant(false)]

public class Bar {...}
```

Schreiben eigener Attribute

Sie definieren Ihre eigenen Attribute, indem Sie von System.Attribute ableiten. So könnten wir zum Beispiel das folgende eigene Attribut nutzen, um eine Methode zum Unit-Testing zu markieren:

```
[AttributeUsage (AttributeTargets.Method)]

public sealed class TestAttribute : Attribute

{

    public int    Repetitions;

    public string FailureMessage;


    public TestAttribute () : this (1) { }

    public TestAttribute (int repetitions)

        => Repetitions = repetitions;

}
```

Und so ließe sich das Attribut dann anwenden:

```
class Foo
{
    [Test]
    public void Method1() { ... }

    [Test(20)]
    public void Method2() { ... }

    [Test(20, FailureMessage="Debugging Time!")]
    public void Method3() { ... }
}
```

AttributeUsage ist selbst ein Attribut, das das Konstrukt (oder die Kombination von Konstrukten) kenntlich macht, auf das das eigene Attribut angewendet werden kann. Das Enum AttributeTargets enthält Member wie Class, Method, Parameter und Constructor (sowie All, das alle Ziele kombiniert).

Auslesen von Attributen zur Laufzeit

Es gibt zwei klassische Wege, Attribute zur Laufzeit auszulesen:

- Der Aufruf von GetCustomAttributes für ein Type- oder Member-Info-Objekt.
- Der Aufruf von Attribute.GetCustomAttribute oder Attribute.GetCustomAttributes.

Diese beiden letzten Methoden sind überladen, damit ein beliebiges Reflection-Objekt übergeben werden kann, das zu einem gültigen Attributziel passt (Type, Assembly, Module, MemberInfo oder ParameterInfo).

So können wir jede Methode in der Klasse Foo von gerade eben enumerieren, die ein TestAttribute hat:

```
foreach (MethodInfo mi in typeof (Foo).GetMethods())
{
    TestAttribute att = (TestAttribute)
        Attribute.GetCustomAttribute
         (mi, typeof (TestAttribute));

    if (att != null)
        Console.WriteLine (
            "{0} wird geprüft; reps={1}; msg={2}",
            mi.Name, att.Repetitions, att.FailureMessage);
}
```

Das ist die Ausgabe:

```
Method Method1 wird getestet; reps=1; msg=
Method Method2 wird getestet; reps=20; msg=
Method Method3 wird getestet; reps=20; msg=Debugging!
```

Aufrufer-Info-Attribute

Seit C# 5.0 können Sie optionale Parameter mit einem der drei *Aufrufer-Info-Attribute* versehen, die den Compiler anweisen, Daten, die aus dem Quellcode des Aufrufers entnommen werden, in den Standardwert für den Parameter aufzunehmen:

- [CallerMemberName] wendet den Member-Namen des Aufrufers an.
- [CallerFilePath] wendet den Pfad zur Quellcodedatei des Aufrufers an.
- [CallerLineNumber] wendet die Zeilennummer in der Quellcodedatei des Aufrufers an.

Die Foo-Methode im folgenden Programm stellt alle drei vor:

```
using System;

using System.Runtime.CompilerServices;


class Program

{

    static void Main() => Foo();

    static void Foo (

        [CallerMemberName] string memberName = null,

        [CallerFilePath] string filePath = null,

        [CallerLineNumber] int lineNumber = 0)

    {

        Console.WriteLine (memberName);

        Console.WriteLine (filePath);

        Console.WriteLine (lineNumber);
```

```
    }
}
```

Wenn unser Programm unter *c:\source\test\Program.cs* gespeichert wäre, würde die Ausgabe so aussehen:

```
Main
c:\source\test\Program.cs
6
```

Wie bei gewöhnlichen optionalen Parametern erfolgt die Substitution auf *Aufruferseite*. Unsere Main-Methode ist also nur syntaktischer Zucker für Folgendes:

```
static void Main()
    => Foo ("Main", @"c:\source\test\Program.cs", 6);
```

Aufrufer-Info-Attribute sind sowohl beim Schreiben von Logging-Funktionen als auch bei der Implementierung von Benachrichtigungsmustern für Veränderungen praktisch. Beispielsweise kann eine Methode wie die folgende aus dem set-Accessor einer Eigenschaft aufgerufen werden, ohne dass der Name der Eigenschaft angegeben werden muss:

```
void RaisePropertyChanged (
    [CallerMemberName] string propertyName = null)
    {
        ...
    }
```

Asynchrone Funktionen

Die Schlüsselwörter await und async unterstützen seit C# 5 die *asynchrone Programmierung* – ein Programmierstil, bei dem Funktionen, die lange laufen, einen Großteil ihrer Arbeit verrichten, *nachdem* sie die Ausführungskontrolle an den Aufrufer zurückgegeben haben. Das steht im Gegensatz zur normalen *synchronen* Programmierung, bei der Funktionen, die lange laufen, den Aufrufer *blockieren*, bis die Operation abgeschlossen ist. Asynchrone Programmierung impliziert *Nebenläufigkeit*, da die lange laufenden Operationen *parallel* zu den Operationen des Aufrufers ausgeführt werden. Der Implementierer einer asynchronen Funktion initiiert diese Nebenläufigkeit entweder durch Multithreading (für berechnungsgebundene Operationen) oder über einen Callback-Mechanismus (für Eingabe/Ausgabe-gebundene Operationen).

Multithreading, Nebenläufigkeit und asynchrone Programmierung sind umfangreiche Themen. Wir erörtern sie online unter *http://albahari.com/threading*.

Betrachten Sie zum Beispiel die folgende *synchrone* Methode, die lange läuft und berechnungsgebunden ist:

```
int ComplexCalculation()
{
    double x = 2;
    for (int i = 1; i < 100000000; i++)
        x += Math.Sqrt (x) / i;
    return (int)x;
}
```

Wenn diese Methode ausgeführt wird, blockiert sie den Aufrufer für einige Sekunden. Anschließend wird das Ergebnis der Berechnung an den Aufrufer zurückgeliefert:

```
int result = ComplexCalculation();

// Etwas später:

Console.WriteLine (result);    // 116
```

Die CLR definiert eine Klasse namens Task<TResult> (in System.Threading.Tasks), um das Konzept einer Operation zu kapseln, die irgendwann in der Zukunft abgeschlossen wird. Sie können einen Task<TResult> für eine berechnungsgebundene Operation erstellen, indem Sie Task.Run aufrufen, das die CLR anweist, das angegebene Delegate in einem separaten Thread auszuführen, der parallel zum Aufrufer ausgeführt wird:

```
Task<int> ComplexCalculationAsync()
{
    return Task.Run (() => ComplexCalculation());
}
```

Diese Methode ist *asynchron*, weil sie die Ausführung unmittelbar an den Aufrufer zurückgibt, während sie nebenläufig ausgeführt wird. Wir benötigen allerdings einen Mechanismus, über den der Aufrufer angeben kann, was passieren soll, wenn die Operation abgeschlossen und das Ergebnis verfügbar ist. Task<TResult> löst das, indem es eine GetAwaiter-Methode veröffentlicht, über die der Aufrufer eine *Continuation* (Fortsetzung) anbinden kann:

```
Task<int> task = ComplexCalculationAsync();
var awaiter = task.GetAwaiter();
awaiter.OnCompleted (() =>          // Continuation
{
    int result = awaiter.GetResult();
    Console.WriteLine (result);       // 116
```

```
});
```

Das sagt der Operation, dass sie, wenn sie fertig ist, das angegebene Delegate ausführen soll. Unsere Continuation ruft zuerst GetResult, das das Ergebnis der Berechnung liefert (wenn der Task *fehlgeschlagen* ist, also eine Exception ausgelöst hat, löst ein Aufruf von Get-Result diese Exception neu aus). Die Continuation gibt das Ergebnis dann über Console.WriteLine aus.

Die Schlüsselwörter await und async

Das Schlüsselwort await vereinfacht das Anbinden von Continuations. Nehmen wir einen einfachen Fall als Beispiel. Der Compiler expandiert

```
var Ergebnis = await

Ausdruck;

Anweisung(en);
```

in etwas, das funktionell Folgendem entspricht:

```
var awaiter = Ausdruck.GetAwaiter();

awaiter.OnCompleted (() =>

{

    var Ergebnis = awaiter.GetResult();

    Anweisung(en);

);
```

> Der Compiler gibt außerdem Code aus, um das Szenario zu optimieren, dass die Operation synchron (sofort) abgeschlossen wird. Dass eine asynchrone Operation sofort abgeschlossen wird, tritt häufig dann ein, wenn sie einen internen Caching-Mechanismus implementiert und das Ergebnis bereits gecacht ist.

Also rufen wir die zuvor definierte ComplexCalculationAsync-Methode folgendermaßen auf:

```
int result = await ComplexCalculationAsync();

Console.WriteLine (result);
```

Das lässt sich nur kompilieren, wenn wir die umschließende Methode mit dem Modifikator async versehen:

```
async void Test()

{

   int result = await ComplexCalculationAsync();

   Console.WriteLine (result);

}
```

Der Modifikator async sagt dem Compiler, dass er await als Schlüsselwort behandeln soll und nicht als Bezeichner, sollte in dieser Methode eine Mehrdeutigkeit auftauchen (das sorgt dafür, dass Code, der vor C# 5.0 geschrieben wurde und eventuell await als Bezeichner verwendet, weiterhin anstandslos kompiliert werden kann). Der Modifikator async kann nur auf Methoden (und Lambda-Ausdrücke) angewendet werden, die void oder (wie wir später sehen werden) ein Task- oder ein Task<TResult>-Objekt liefern.

Der Modifikator async ähnelt dem Modifikator unsafe darin, dass er keine Auswirkungen auf die Methodensignatur oder die öffentlichen Metadaten hat. Er wirkt sich nur auf das aus, was *in* der Methode passiert.

Methoden mit dem Modifikator async werden als *asynchrone Funktionen* bezeichnet, weil sie selbst üblicherweise asynchron sind. Schauen wir uns an, wie die Ausführung durch eine asynchrone Funktion verläuft, um zu sehen, warum.

Wenn die Ausführung auf einen await-Ausdruck stößt, gibt sie normalerweise an den Aufrufer zurück – ganz ähnlich wie ein yield return in einem Iterator. Aber bevor die Ausführung zurückkehrt, bindet die Runtime eine Continuation an den Task, dessen Abschluss erwartet wird, und sichert so, dass die Ausführung, wenn der Task fertig ist, wieder in die Methode springt und die Arbeit dort fortsetzt, wo sie ausgesetzt wurde. Wenn der Task fehlschlägt, wird seine Exception neu ausgelöst (was der Get-Result-Aufruf bewirkt); andernfalls wird ihr Rückgabewert dem await-Ausdruck zugewiesen.

Die Implementierung, die die CLR für die OnCompleted-Methode für die Continuation eines Tasks bereitstellt, sorgt dafür, dass Continuations standardmäßig im gesamten aktuellen *Synchronisationskontext* bekannt gemacht werden, wenn ein solcher vorhanden ist. In der Praxis bedeutet das, dass in Rich-Client-UI-Umgebungen (WPF, UWP und Windows Forms) Ihr Code auf demselben Thread fortgesetzt wird, wenn Sie await auf einem UI-Thread setzen. Das vereinfacht die Threadsicherheit.

Der Ausdruck, auf dem Sie await nutzen, ist üblicherweise ein Task; aber den Compiler stellt jedes Objekt zufrieden, das eine GetAwaiter-Methode bietet, die ein Objekt liefert, das await unterstützt, also INotifyCompletion.OnCompleted implementiert und eine GetResult-Methode mit einem entsprechenden Rückgabetyp bietet (und eine bool IsCompleted-Eigenschaft hat, die auf synchrone Vervollständigung prüft).

Beachten Sie, dass unser await-Ausdruck zu einem int ausgewertet wird. Das liegt daran, dass der Ausdruck, auf den gewartet wird, ein Task<int> ist (dessen GetAwaiter().GetResult()-Methode einen int liefert).

Das Warten auf einen nichtgenerischen Task ist zulässig und erzeugt einen leeren Ausdruck (void):

```
await Task.Delay (5000);

Console.WriteLine ("Fünf Sekunden verstrichen!");
```

Task.Delay ist eine statische Methode, die einen Task liefert, der in der angegebenen Anzahl von Millisekunden fertig ist. Das *synchrone* Äquivalent zu Task.Delay ist Thread.Sleep.

Task ist die nichtgenerische Basisklasse von Task<TResult> und entspricht

funktionell Task<TResult>, davon abgesehen, dass es kein Ergebnis gibt .

Lokale Zustände einfangen

Die wahre Macht von await-Ausdrücken liegt darin, dass sie praktisch überall im Code erscheinen können. Genauer gesagt, kann ein await-Ausdruck (in einer asynchronen Funktion) überall dort stehen, wo ein Ausdruck stehen darf, außer innerhalb eines catch- oder finally-Blocks, eines lock-Ausdrucks oder eines unsafe-Kontexts oder im Einstiegspunkt eines Programms (der Main-Methode).

Im folgenden Beispiel nutzen wir await in einer Schleife:

```
async void Test()
{
    for (int i = 0; i < 10; i++)
    {
        int result = await ComplexCalculationAsync();
        Console.WriteLine (result);
    }
}
```

Beim ersten Aufruf von ComplexCalculationAsync kehrt die Ausführung dank await-Ausdruck an den Aufrufer zurück. Wenn die Methode fertig ist (oder fehlschlägt), wird die Ausführung dort wieder aufgenommen, wo sie abgebrochen wurde, und die Werte der lokalen Variablen und Schleifenzähler bleiben dabei erhalten. Das erreicht der Compiler dadurch, dass er derartigen Code in eine Zustandsmaschine übersetzt, wie er es bei Iteratoren macht.

Ohne das Schlüsselwort await bringt es der manuelle Einsatz von Continuations mit sich, dass Sie etwas einer Zustandsmaschine Äquivalentes schreiben müssen. Das ist das, was die asynchrone Programmierung traditionell so kompliziert gemacht hat.

Asynchrone Funktionen schreiben

Bei jeder asynchronen Funktion können Sie den Rückgabetyp void durch Task ersetzen, um die Methode selbst *nützlich* asynchron (und await-fähig) zu machen. Es sind keine weiteren Änderungen erforderlich:

```
async Task PrintAnswerToLife()
{
    await Task.Delay (5000);
    int answer = 21 * 2;
    Console.WriteLine (answer);
}
```

Beachten Sie, dass wir aus dem Methodenrumpf nicht explizit ein Task-Objekt zurückliefern. Der Compiler erzeugt das Task-Objekt, das er benachrichtigt, wenn die Methode abgearbeitet ist (oder eine unbehandelte Exception ausgelöst hat). Das macht es leicht, asynchrone Aufrufketten zu erstellen:

```
async Task Go()
{
    await PrintAnswerToLife();
    Console.WriteLine ("Fertig");
}
```

(Und weil Go ein Task-Objekt liefert, ist Go selbst await-fähig.) Der Compiler expandiert asynchrone Funktionen, die Tasks liefern, zu Code, der (indirekt) TaskCompletionSource nutzt, um einen Task zu erstellen, der dann benachrichtigt wird oder fehlschlägt.

> TaskCompletionSource ist ein CLR-Typ, mit dem Sie Tasks erstellen können, die Sie manuell steuern und deren Abschluss Sie mit einem Ergebnis signalisieren (oder deren Fehlschlag mit einer Exception). Im Unterschied zuTask.Run bindet TaskCompletionSource keinen

Thread für die Dauer der Operation. Die Klasse ist auch für Methoden geeignet, die Eingabe/Ausgabe-gebundene Tasks liefern (wie Task.Delay).

Das soll sicherstellen, dass die Ausführung immer über eine Continuation zum Wartenden zurückspringen kann, wenn die einen Task liefernde asynchrone Funktion fertig ist.

Task<TResult> zurückliefern

Sie können Task<TResult> liefern, wenn der Methodenrumpf TResult liefert:

```
async Task<int> GetAnswerToLife()
{
    await Task.Delay (5000);
    int answer = 21 * 2;
    // answer ist int, also nutzt die Methode Task<int>
    return answer;
}
```

Wir probieren GetAnswerToLife aus, indem wir es aus PrintAnswerToLife aufrufen (das seinerseits von Go aufgerufen wird):

```
async Task Go()
{
    await PrintAnswerToLife();
    Console.WriteLine ("Fertig");
}
```

```
async Task PrintAnswerToLife()
{
    int answer = await GetAnswerToLife();
    Console.WriteLine (answer);
}
async Task<int> GetAnswerToLife()
{
    await Task.Delay (5000);
    int answer = 21 * 2;
    return answer;
}
```

Asynchrone Funktionen gleichen die asynchrone Programmierung der synchronen Programmierung an. Hier ist das synchrone Äquivalent unseres Aufrufgrafen, für das ein Aufruf von Go() das gleiche Ergebnis liefert, nachdem die Ausführung für fünf Sekunden blockiert war:

```
void Go()
{
    PrintAnswerToLife();
    Console.WriteLine ("Fertig");
}
void PrintAnswerToLife()
{
```

```
    int answer = GetAnswerToLife();
    Console.WriteLine (answer);
}
int GetAnswerToLife()
{
    Thread.Sleep (5000);
    int answer = 21 * 2;
    return answer;
}
```

Das illustriert auch eines der Grundprinzipien beim Schreiben asynchroner Funktionen in C#, nämlich dass Sie Ihre Methoden synchron schreiben und dann die *synchronen* Methodenaufrufe durch *asynchrone* Methodenaufrufe ersetzen sollten, mit denen Sie await nutzen.

Parallelität

Wir haben gerade das Muster vorgestellt, das am weitesten verbreitet ist: unmittelbar nach dem Aufruf mit await auf Funktionen zu warten, die einen Task liefern. Das führt zu einem sequenziellen Programmablauf, der logisch dem synchronen Äquivalent ähnelt.

Der Aufruf einer asynchronen Methode ohne await gestattet die parallele Ausführung des nachfolgenden Codes. Beispielsweise führt Folgendes PrintAnswerToLife zwei Mal nebenläufig aus:

```
var task1 = PrintAnswerToLife();
var task2 = PrintAnswerToLife();
await task1; await task2;
```

Dadurch, dass wir danach auf beide Operationen warten, »beenden« wir die Parallelität

an diesem Punkt (und lösen alle eventuellen Exceptions aus diesen Tasks neu aus). Die Klasse Task bietet eine statische Methode namens WhenAll, mit der sich das gleiche Ergebnis etwas effizienter erreichen lässt. WhenAll liefert einen Task, der fertig wird, wenn alle Tasks, die Sie ihm übergeben, fertig sind:

```
await Task.WhenAll (PrintAnswerToLife(),
                    PrintAnswerToLife());
```

WhenAll bezeichnet man als einen *Task-Verknüpfer*. (Die Klasse Task bietet außerdem einen Task-Verknüpfer namens WhenAny, der fertig ist, wenn *einer* der ihm übergebenen Tasks fertig ist.)

Asynchrone Lambda-Ausdrücke

Ebenso wie gewöhnliche *benannte* Methoden asynchron sein können:

```
async Task NamedMethod()
{
  await Task.Delay (1000);
  Console.WriteLine ("Foo");
}
```

können auch *namenlose* Methoden (Lambda-Ausdrücke und anonyme Methoden) asynchron sein, wenn ihnen das Schlüsselwort async vorangestellt wird:

```
Func<Task> unnamed = async () =>
{
  await Task.Delay (1000);
  Console.WriteLine ("Foo");
};
```

Wir rufen diese Methoden auf die gleiche Weise auf und warten auf sie:

```
await NamedMethod();

await unnamed();
```

Asynchrone Lambda-Ausdrücke können beim Anbinden von Event-Handlern genutzt werden:

```
myButton.Click += async (sender, args) =>
{
    await Task.Delay (1000);
    myButton.Content = "Fertig";
};
```

Das ist kompakter als folgende Formulierung, die die gleiche Wirkung hat:

```
myButton.Click += ButtonHandler;
...
async void ButtonHander (object sender, EventArgs args)
{
    await Task.Delay (1000);
    myButton.Content = "Done";
};
```

Asynchrone Lambda-Ausdrücke können auch Task<TResult> liefern:

```
Func<Task<int>> unnamed = async () =>
{
    await Task.Delay (1000);
    return 123;
};
int answer = await unnamed();
```

Unsicherer Code und Zeiger

C# unterstützt die direkte Veränderung von Speicherinhalten durch Zeiger, die in als unsicher gekennzeichneten Codeblöcken genutzt werden und mit der Compileroption /unsafe kompiliert wurden. Zeigertypen werden vor allem aus Gründen der Interoperabilität mit C-APIs genutzt, sie können aber auch für den Zugriff auf Speicher außerhalb des verwalteten Heap oder für performancekritische Hotspots verwendet werden.

Zeigergrundlagen

Zu jedem Werttyp bzw. Referenztyp *V* gibt es einen korrespondierenden Zeigertyp *V**. Eine Zeigerinstanz hält die Adresse einer Variablen. Zeigertypen können (unsicher) auf jeden anderen Zeigertyp gecastet werden. Die wichtigsten Zeigeroperatoren sind folgende:

Operator	Bedeutung
&	Der *Adressoperator* liefert einen Zeiger auf die Adresse des Werts zurück.
*	Der *Dereferenzierungsoperator* liefert den Wert an der Adresse eines Zeigers zurück.
->	Der *Zeiger-auf-Member-Operator* ist eine syntaktische Abkürzung: x->y ist äquivalent zu (*x).y.

Unsicherer Code

Dadurch, dass Sie einen Typ, ein Typ-Member oder einen Anweisungsblock mit dem Schlüsselwort unsafe kennzeichnen, dürfen Sie innerhalb dieses Geltungsbereichs Zeigertypen nutzen und Zeigeroperationen im C++-Stil auf Speicherbereichen durchführen. Hier sehen Sie ein Beispiel für die Verwendung von Zeigern, um eine Bitmap schnell zu verarbeiten:

```
unsafe void BlueFilter (int[,] bitmap)
{
    int length = bitmap.Length;
```

```
        fixed (int* b = bitmap)
        {
            int* p = b;
            for (int i = 0; i < length; i++)
                *p++ &= 0xFF;
        }
    }
```

Unsicherer Code kann schneller laufen als die entsprechende sichere Implementierung. In diesem Fall wäre eine verschachtelte Schleife mit Array-Indexierung und Prüfung der Array-Grenzen notwendig. Eine unsichere C#-Methode kann auch schneller sein als das Aufrufen einer externen C-Funktion, da es keinen Overhead gibt, der durch das Verlassen der verwalteten Ausführungsumgebung entsteht.

Die Anweisung fixed

Die Anweisung fixed ist notwendig, um ein verwaltetes Objekt »festzunageln«, zum Beispiel die Bitmap im vorigen Beispiel. Während der Ausführung eines Programms werden viele Objekte auf dem Heap angefordert und wieder freigegeben. Um eine unnötige Verschwendung und Fragmentierung von Speicher zu vermeiden, verschiebt der Garbage Collector Objekte. Das Zeigen auf ein Objekt ist aber nicht sinnvoll, wenn sich seine Adresse während des Referenzierens ändern kann, daher weist die Anweisung fixed den Garbage Collector an, ein Objekt nicht zu verschieben. Das kann einen Einfluss auf die Laufzeiteffizienz haben, daher sollten feste Blöcke nur verwendet werden, wenn es notwendig ist. Zudem sollte man das Anfordern von Objekten auf dem Heap innerhalb eines festen Blocks vermeiden.

Innerhalb einer fixed-Anweisung können Sie einen Zeiger auf einen beliebigen Werttyp, ein Array mit Werttypen oder einen String erhalten. Bei Arrays und Strings wird der Zeiger auf das erste Element zeigen, einen Werttyp.

Bei Werttypen, die innerhalb von Referenztypen deklariert wurden, muss der Referenztyp »festgenagelt« werden:

```
class Test
```

```
{
    int x;
    unsafe static void Main()
    {
        Test test = new Test();
        fixed (int* p = &test.x)   // fixiert test
        {
            *p = 9;
        }
        System.Console.WriteLine (test.x);
    }
}
```

Der Zeiger-auf-Member-Operator

Neben den Operatoren & und * bietet C# auch den Operator -> im C++-Stil, der für Structs genutzt werden kann:

```
struct Test
{
    int x;
    unsafe static void Main()
    {7c
```

```
        Test test = new Test();
        Test* p = &test;
        p->x = 9;
        System.Console.WriteLine (test.x);
    }
}
```

Arrays

Das Schlüsselwort stackalloc keyword

Speicher kann als Block auf dem Stack explizit mit dem Schlüsselwort stackalloc angefordert werden. Da er auf dem Stack angefordert wird, ist seine Lebenszeit auf die Ausführung der Methode beschränkt, so wie jede andere lokale Variable. Der Block kann den Operator [] nutzen, um auf Speicher indexiert zuzugreifen:

```
int* a = stackalloc int [10];
for (int i = 0; i < 10; ++i)
    Console.WriteLine (a[i]);   // in den nackten Speicher
                                // ausgeben
```

Puffer fester Größe

Speicher kann in einem Block innerhalb eines Struct mithilfe des Schlüsselworts fixed angefordert werden:

```
unsafe struct UnsafeUnicodeString
{
    public short Length;
```

```
    public fixed byte Buffer[30];
}

unsafe class UnsafeClass
{
    UnsafeUnicodeString uus;

    public UnsafeClass (string s)
    {
        uus.Length = (short)s.Length;
        fixed (byte* p = uus.Buffer)
            for (int i = 0; i < s.Length; i++)
                p[i] = (byte) s[i];
    }
}
```

Das Schlüsselwort fixed wird in diesem Beispiel auch genutzt, um das Objekt, das den Puffer enthält, auf dem Heap festzuhalten (dabei handelt es sich um die Instanz von UnsafeClass).

void*

Ein *void-Zeiger* (void*) macht keine Annahmen in Bezug auf den Typ der zugrunde liegenden Daten und ist für Funktionen geeignet, die direkt mit dem Speicher arbeiten. Es existiert eine implizite Konvertierung von jedem Zeigertyp auf void*. Ein void* kann nicht dereferenziert werden, und es sind keine arithmetischen Operationen möglich:

```
unsafe static void Main()
{
    short[] a = {1,1,2,3,5,8,13,21,34,55};
    fixed (short* p = a)
    {
        // sizeof liefert die Größe von Werttypen
        // in Byte zurück
        Zap (p, a.Length * sizeof (short));
    }
    foreach (short x in a)
        System.Console.WriteLine (x);  // gibt Nullen aus
}

unsafe static void Zap (void* memory, int byteCount)
{
    byte* b = (byte*) memory;
        for (int i = 0; i < byteCount; i++)
            *b++ = 0;
}
```

Präprozessordirektiven

Präprozessordirektiven versorgen den Compiler mit zusätzlichen Informationen über Codebereiche. Die am häufigsten verwendeten Präprozessordirektiven sind die bedingten Direktiven, die eine Möglichkeit bieten, Codebereiche beim Kompilieren auszuschließen oder mit zu berücksichtigen. Hier sehen Sie ein Beispiel dafür:

```
#define DEBUG

class MyClass

{

    int x;

    void Foo()

    {

        # if DEBUG

        Console.WriteLine ("Test: x = {0}", x);

        # endif}

    ...

}
```

In dieser Klasse wird die Anweisung in Foo in Abhängigkeit davon kompiliert, ob das Symbol DEBUG vorhanden ist. Wenn wir es entfernen, wird die Anweisung nicht berücksichtigt. Präprozessorsymbole können in einer Quelldatei definiert (wie wir es getan haben) oder dem Compiler über die Befehlszeilenoption /define:*symbol* mitgegeben werden.

Bei den Direktiven #if und #elif können Sie die Operatoren ||, && und ! nutzen, um *oder*-, *und*- bzw. *nicht*-Operationen mit mehreren Symbolen durchzuführen. Die folgende Direktive weist den Compiler an, den Code mit zu berücksichtigen, wenn das Symbol TESTMODE definiert ist, DEBUG aber nicht:

```
#if TESTMODE && !DEBUG

    ...
```

Denken Sie jedoch daran, dass Sie hier keinen normalen C#-Ausdruck verwenden und die Symbole, mit denen Sie umgehen, keinerlei Verbindung zu Variablen haben – weder zu statischen noch zu anderen.

Die Symbole #error und #warning verhindern unbeabsichtigte Fehlanwendungen von bedingten Direktiven, indem sie dafür sorgen, dass der Compiler eine Warnung oder einen Fehler ausgibt, falls es zu unerwünschten Kombinationen von Kompilierungssymbolen kommt.

Hier ist eine vollständige Liste der Präprozessordirektiven:

Präprozessordirektive	**Aktion**
#define ***Symbol***	Definiert *Symbol*.
#undef ***Symbol***	Hebt Definition von *Symbol* auf.
#if *Symbol* [*Operator* Symbol2]...	Bedingte Kompilierung (*Operator* kann ==, !=, && und || sein).
#else	Führt den Code bis zum nächsten #endif aus.
#elif *Symbol* [*Operator Symbol2*]	Kombiniert #else-Zweig und #if-Test.
#endif	Beendet Bedingungsdirektiven.
#warning ***Text***	*Text* der Warnung, die in der Compilerausgabe erscheinen soll.
#error ***Text***	*Text* des Fehlers, der in der Compilerausgabe erscheinen soll.
#line [*Zahl* [" *Datei* "] | hidden]	*Zahl* legt die Zeile im Quellcode fest; *Datei* ist der Dateiname, der bei der Ausgabe erscheinen soll; hidden weist Debugger an, den Code zwischen dieser Zeile und der nächsten #line-Direktive zu überspringen.
#region ***name***	Markiert den Beginn eines Abschnitts.
#endregion	Beendet einen Abschnitt.

#pragma warning	Siehe den nächsten Abschnitt.

Warnungen

Der Compiler erzeugt eine Warnung, wenn er in Ihrem Code auf etwas stößt, das nicht gewollt aussieht. Anders als bei Fehlern halten Warnungen Ihre Anwendung normalerweise nicht davon ab, kompiliert zu werden.

Compilerwarnungen können außerordentlich nützlich sein, wenn es darum geht, Fehler zu finden. Ihr Nutzen wird allerdings ausgehöhlt, wenn es *falsche* Warnungen gibt. In einer großen Anwendung ist es wichtig, ein gutes Signal-Rausch-Verhältnis zu haben, damit die *echten* Warnungen nicht übersehen werden.

Daher erlaubt der Compiler Ihnen, ausgewählte Warnungen mit der Direktive #pragma warning zu unterdrücken. In diesem Beispiel weisen wir den Compiler an, uns nicht davor zu warnen, dass das Feld Message nicht verwendet wird:

```
public class Foo
{
    static void Main() { }
    #pragma warning disable 414
    static string Message = "Hallo";
    #pragma warning restore 414
}
```

Lässt man die Zahl in der Direktive #pragma warning weg, werden alle Warnmeldungen deaktiviert bzw. wieder aktiviert.

Wenn Sie bei der Anwendung dieser Direktive sehr exakt vorgehen, können Sie mit dem Schalter /warnaserror kompilieren – damit wird dem Compiler mitgeteilt, dass er jede verbleibende Warnung als Fehler ansehen soll.

XML-Dokumentation

Ein *Dokumentationskommentar* ist ein Stück eingebettetes XML, das einen Typ oder ein Member dokumentiert. Ein Dokumentationskommentar steht direkt vor einer Deklaration des Typs oder Members und beginnt mit drei Schrägstrichen (Slashes):

```
/// <summary>Beendet eine laufende Abfrage.</summary>
public void Cancel() { ... }
```

Kommentare über mehrere Zeilen können entweder so geschrieben werden:

```
/// <summary>
/// Beendet eine laufende Abfrage.
/// </summary>
public void Cancel() { ... }
```

oder so (beachten Sie das zusätzliche Sternchen am Anfang):

```
/**
    <summary> Beendet eine laufende Abfrage. </summary>
*/
public void Cancel() { ... }
```

Wenn Sie mit der Direktive /doc kompilieren, extrahiert der Compiler die Dokumentationskommentare und sammelt sie in einer einzelnen XML-Datei. Dafür gibt es zwei Hauptanwendungsfälle:

- Wenn sich die Datei im selben Ordner wie die kompilierte Assembly befindet, liest Visual Studio die XML-Datei automatisch ein und verwendet die Informationen, um Nutzern der Assembly gleichen Namens per IntelliSense die Member anzuzeigen.

- Tools von Fremdherstellern (wie Sandcastle und NDoc) können die XML-Datei in eine HTML-Hilfsdatei umwandeln.

Standard-XML-Dokumentations-Tags

Das hier sind die Standard-XML-Tags, die Visual Studio und Dokumentationsgeneratoren erkennen:

<summary>

<summary>**...**</summary>

Definiert den Tooltipp, den IntelliSense für den Typ oder das Member anzeigen soll, meist ein einzelner Satz oder eine Phrase.

<remarks>

<remarks>**...**</remarks>

Zusätzlicher Text, der den Typ oder das Member beschreibt. Dokumentationsgeneratoren nehmen dieses Tag und nutzen den Text für die Beschreibung eines Typs oder Members.

<param>

<param name="Name">...</param>

Erläutert einen Parameter einer Methode.

<returns>

<returns>...</returns>

Erläutert den Rückgabewert einer Methode.

<exception>

<exception [cref="*Typ*"]>...</exception>

Führt eine Exception auf, die eine Methode eventuell wirft (cref bezieht sich auf den Exception-Typ).

<permission>

<permission [cref="*Typ*"]>...</permission>

Gibt einen IPermission-Typ an, der vom dokumentierten Typ oder Member gebraucht wird.

<example>

<example>...</example>

Beschreibt ein Beispiel (das von Dokumentationsgeneratoren verwendet wird). Dazu gehören normalerweise sowohl beschreibender Text als auch Quellcode

(der dann in den Tags <c> oder <code> steht).

<c>

<c>...</c>

Steht für einen Codeschnipsel innerhalb von Text. Dieses Tag wird meist in einem <example>-Block genutzt.

<code>

<code>...<code>

Steht für ein mehrzeiliges Codebeispiel. Dieses Tag wird meist in einem <example>-Block genutzt.

<see>

<see cref="*Member*">...</see>

Fügt einen Verweis auf einen anderen Typ oder ein anderes Member ein. HTML-Dokumentationsgeneratoren wandeln dies normalerweise in einen Hyperlink um. Der Compiler gibt eine Warnung aus, wenn der Name des Typs oder Members ungültig ist.

<seealso>

<seealso cref="*Member*">7c...</seealso>

Verweis auf einen anderen Typ oder ein anderes Member. Dokumentationsgeneratoren fügen den Text im Allgemeinen in einen separaten Abschnitt »See also« am Ende der Seite ein.

<paramref>

<paramref name="*Name*"/>

Referenziert einen Parameter innerhalb eines <summary>- oder <remarks>-Tags.

<list>

<list type=[bullet | number | table]>

<listheader>

<term>...</term>

<description>...</description>

</listheader>

```
    <item>
      <term>...</term>
      <description>...</description>
    </item>
  </list>
```

Weist Dokumentationsgeneratoren an, eine unnummerierte, nummerierte oder tabellenförmige Liste auszugeben.

<para>

<para>...</para>

Weist Dokumentationsgeneratoren an, den Inhalt in einem eigenen Absatz zu formatieren.

<include>

<include file='*Dateiname*' path='*Tag-Pfad*[@name="*id*"]'>

...

</include>

Bindet eine externe XML-Datei ein. Das path-Attribut definiert eine XPath-Abfrage, um auf ein bestimmtes Element in dieser Datei zu verweisen.

Index

Symbole

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Rezensieren

Sie dieses Buch

Senden

Sie uns Ihre Rezension
unter **www.dpunkt.de/rez**

Erhalten

Sie Ihr Wunschbuch aus
unserem Verlagsangebot

Table of Contents