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

O'REILLY®

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 **1**
Ein erstes C#-Programm 1
Syntax 5
Typgrundlagen 8
Numerische Typen 17
Der Typ bool und die booleschen Operatoren 25
Strings und Zeichen 27
Arrays 31
Variablen und Parameter 36
Ausdrücke und Operatoren 44
Null-Operatoren 49
Anweisungen 51
Namensräume 60
Klassen 64
Vererbung 79
Der Typ object 88
Structs 93
Zugriffsmodifikatoren 94
Interfaces 95
Enums 99
Eingebettete Typen 102
Generics 102
Delegates 111
Events 118
Lambda-Ausdrücke 124
Anonyme Methoden 128

try-Anweisungen und Exceptions 129
Enumeration und Iteratoren 138
Nullbare Typen 144
Erweiterungsmethoden 149
Anonyme Typen 151
Tupel (C# 7) 152
LINQ 154
Die dynamische Bindung 179
Überladen von Operatoren 188
Attribute 191
Aufrufer-Info-Attribute 195
Asynchrone Funktionen 196
Unsicherer Code und Zeiger 206
Präprozessordirektiven 210
XML-Dokumentation 213

Index 217

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 LINQPad 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` auto-

matisch 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
```

Eigenartigerweise bringen die .NET Frameworks 4.6 und 4.7 den C# 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 CamelCase 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	enum	long	stackalloc
as	event	namespace	static
base	explicit	new	string
bool	extern	null	struct
break	false	object	switch
byte	finally	operator	this
case	fixed	out	throw
catch	float	override	true
char	for	params	try
checked	foreach	private	typeof
class	goto	protected	uint
const	if	public	ulong
continue	implicit	readonly	unchecked
decimal	in	ref	unsafe
default	int	return	ushort
delegate	interface	sbyte	using
do	internal	sealed	virtual
double	is	short	void
else	lock	sizeof	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`	`equals`	`join`	`select`
`ascending`	`from`	`let`	`set`
`async`	`get`	`nameof`	`value`
`await`	`global`	`on`	`var`
`by`	`group`	`orderby`	`when`
`descending`	`in`	`partial`	`where`
`dynamic`	`into`	`remove`	`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, beispiels-

weise 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 Methodenname 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; }
```

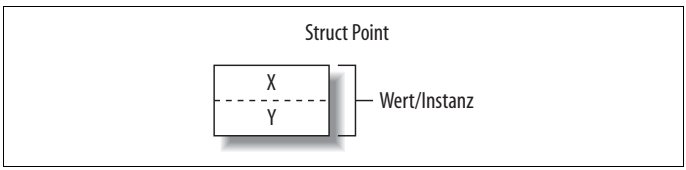

Abbildung 1: Eine Werttyp-Instanz im Speicher

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.

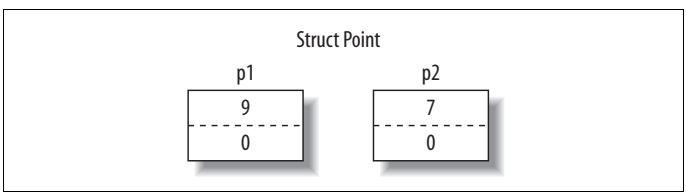

Abbildung 2: Eine Zuweisung kopiert eine Werttyp-Instanz.

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; }
```

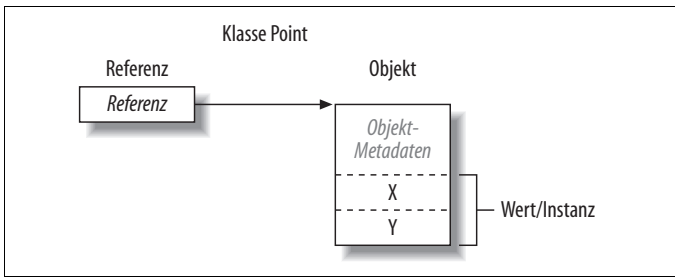

Abbildung 3: Ein Referenztyp im Speicher

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.

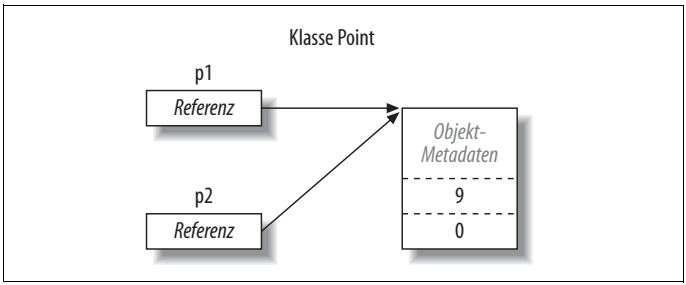

Abbildung 4: Eine Zuweisung kopiert eine Referenz.

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 $2^{15} - 1$

C#-Typ	Systemtyp	Suffix	Breite	Bereich
`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	$\pm$ ($\sim 10^{-45}$ bis 10^{38})
`double`	`Double`	`D`	64 Bit	$\pm$ ($\sim 10^{-324}$ bis 10^{308})
`decimal`	`Decimal`	`M`	128 Bit	$\pm$ ($\sim 10^{-28}$ 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 ein-