O'REILLY®

7. Auflage

C# 10

kurz & gut

O'Reillys Taschenbibliothek

Joseph Albahari
Ben Albahari
Übersetzung von Thomas Demmig

7. AUFLAGE

C# 10

kurz & gut

Joseph Albahari & Ben Albahari

Deutsche Übersetzung von
Thomas Demmig

O'REILLY®

Joseph Albahari, Ben Albahari

Lektorat: Alexandra Follenius
Übersetzung: Thomas Demmig
Copy-Editing: Sibylle Feldmann, *www.richtiger-text.de*
Satz: III-Satz, *www.drei-satz.de*
Herstellung: Stefanie Weidner
Umschlaggestaltung: Karen Montgomery, Michael Oréal, *www.oreal.de*
Druck und Bindung: mediaprint solutions 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 978-3-96009-200-1
PDF 978-3-96010-683-8
ePub 978-3-96010-684-5
mobi 978-3-96010-685-2

7., aktualisierte Auflage 2022

Wieblinger Weg 17
69123 Heidelberg

Hinweis:
Dieses Buch wurde auf PEFC-zertifiziertem Papier aus nachhaltiger Waldwirtschaft gedruckt. Der Umwelt zuliebe verzichten wir auf die Einschweißfolie.

Schreiben Sie uns:
Falls Sie Anregungen, Wünsche und Kommentare haben, lassen Sie es uns wissen: *kommentar@oreilly.de*.

5 4 3 2 1 0

Inhalt

C# 10 – kurz & gut

C# ist eine allgemein anwendbare, typsichere, vorwiegend 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# 10 ist darauf ausgerichtet, mit der *.NET Core 6*-Runtime von Microsoft zusammenzuarbeiten (während C# 9 für *.NET 5*, C# 8 für *.NET Core 3* und C# 7 für *.NET Core 2* sowie Microsoft *.NET Framework* 4.6/4.7/4.8 gedacht war).

Die Programme und Codefragmente in diesem Buch entsprechen denen aus den Kapiteln 2 und 4 von *C# 10 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.

```
int x = 12 * 30;                // Anweisung 1
System.Console.WriteLine (x);   // Anweisung 2
```

Unser Programm besteht aus zwei *Anweisungen*. Anweisungen werden in C# sequenziell ausgeführt und durch ein Semikolon abgeschlossen. Die erste Anweisung berechnet den *Ausdruck* `12 * 30` und speichert das Ergebnis in einer *Variablen* namens `x`, deren Typ ein 32-Bit-Integer ist (`int`). Die zweite Anweisung ruft die *Methode* `WriteLine` einer *Klasse* namens `Console` auf, die in einem *Namensraum* `System` definiert ist. Damit wird die Variable `x` in einem Textfenster auf dem Bildschirm ausgegeben.

Eine Methode führt eine Funktion aus – eine Klasse führt Funktions-Member und Daten-Member zusammen, um einen objektorientierten Baustein zu bilden. Die Klasse `Console` fasst Member zusammen, die sich um die Ein- und Ausgabe (*Input/Output*, I/O) an der Befehlszeile kümmern, wie zum Beispiel die Methode `WriteLine`. Eine Klasse ist die Art eines Typs, den wir im Abschnitt »Typgrundlagen« auf Seite 14 behandeln.

Auf der obersten Ebene sind Typen in Namensräumen organisiert. Viele häufig genutzte Typen – einschließlich der Klasse `Console` – befinden sich im Namensraum `System`. Die .NET-Bibliotheken sind in verschachtelten Namensräumen organisiert. So enthält beispielsweise der Namensraum `System.Text` Typen für den Umgang mit Text, und `System.IO` enthält Typen für die Ein- und Ausgabe.

Muss man die Klasse `Console` bei jeder Verwendung durch den Namensraum `System` qualifizieren, wird das schnell unübersichtlich. Die `using`-Direktive ermöglicht Ihnen, das Ganze besser lesbar zu machen, indem ein Namensraum *importiert* wird:

```
using System;               // Namensraum System importieren

int x = 12 * 30;
Console.WriteLine (x);      // System muss nicht angegeben werden
```

Eine einfache Form der Wiederverwendung von Code ist das Schreiben von High-Level-Funktionen, die wiederum Low-Level-Funktionen aufrufen. Wir können unser Programm mit einer wiederverwendbaren Methode namens `FeetToInches` *refaktorieren*, die eine Ganzzahl mit 12 multipliziert:

```
using System;

Console.WriteLine (FeetToInches (30));      // 360
Console.WriteLine (FeetToInches (100));     // 1200

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

Unsere Methode enthält eine Folge von Anweisungen, die durch ein Paar geschweifter Klammern umschlossen sind. Das wird als *Anweisungsblock* bezeichnet.

Eine Methode kann *Eingabedaten* vom Aufrufer erhalten, indem sie *Parameter* spezifiziert, und Daten an den Aufrufer zurückgeben, indem sie einen *Rückgabetyp* festlegt. Unsere Methode `FeetToInches` besitzt einen Parameter für die Übergabe von Feet und einen Rückgabetyp für die Ausgabe der Inches:

```
int FeetToInches (int feet)
...
```

Die *Literale* `30` und `100` sind die *Argumente*, die an die Methode `FeetToInches` übergeben werden.

Erhält eine Methode keine Eingabewerte, nutzen Sie ein leeres Klammernpaar. Gibt sie nichts zurück, verwenden Sie das Schlüsselwort `void`:

```
using System;
SayHello();

void SayHello()
{
  Console.WriteLine ("Hello, world");
}
```

Methoden sind eine von vielen Arten von Funktionen in C#. Eine andere Art von Funktionen, die wir in unserem Beispielprogramm genutzt haben, war der *Operator* *, der eine Multiplikation durchführt. Es gibt zudem *Konstruktoren*, *Eigenschaften*, *Events*, *Indexer* und *Finalizer*.

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, wobei eine Konsolen- oder Windows-Anwendung einen *Einsprungpunkt* besitzt, während das bei einer Bibliothek nicht der Fall ist. Der Zweck einer Bibliothek ist es, von einer Anwendung oder anderen Bibliotheken aufgerufen (*referenziert*) zu werden. .NET selbst ist ein Satz von Bibliotheken (und eine Laufzeitumgebung).

Jedes der Programme im vorherigen Abschnitt begann direkt mit einer Folge von Anweisungen (auch als *Anweisungen auf oberster Ebene* bezeichnet). Gibt es solche Anweisungen, wird implizit ein Einstiegspunkt für eine Konsolen- oder Windows-Anwendung erzeugt. (Ohne Anweisungen auf oberster Ebene stellt eine *Main-Methode* den Einstiegspunkt dar – siehe den Abschnitt »Symmetrie vordefinierter und benutzerdefinierter Typen« auf Seite 16.)

Um den Compiler aufzurufen, können Sie entweder eine integrierte Entwicklungsumgebung (*Integrated Development Environment*, IDE) wie Visual Studio oder Visual Studio Code nutzen oder ihn selbst per Hand über die Befehlszeile aufrufen. Um eine Konsolenanwendung manuell mit .NET zu kompilieren, laden Sie zunächst das .NET 6 SDK herunter und erzeugen dann ein neues Projekt:

```
dotnet new console -o MyFirstProgram
cd MyFirstProgram
```

Das erstellt einen Ordner namens *MyFirstProgram* mit der C#-Datei Program.cs, die Sie dann bearbeiten können. Zum Aufruf des Compilers rufen Sie `dotnet build` (oder `dotnet run`) auf, wodurch das Programm ausgeführt und dann gestartet wird. Die Ausgabe wird in ein Unterverzeichnis unter *bin\debug* geschrieben. Dort finden sich *MyFirstProgram.dll* (die Ausgabe-Assembly) und *MyFirstProgram.exe* (was das kompilierte Programm direkt aufruft).

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;

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   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. Die Schlüsselwörter in unserem Beispielprogramm sind `using` und `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	catch	default	explicit
as	char	delegate	extern
base	checked	do	false
bool	class	double	finally
break	const	else	fixed
byte	continue	enum	float
case	decimal	event	for

foreach	null	sbyte	typeof
goto	object	sealed	uint
if	operator	short	ulong
implicit	out	sizeof	unchecked
in	override	stackalloc	unsafe
int	params	static	ushort
interface	private	string	using
internal	protected	struct	virtual
is	public	switch	volatile
lock	readonly	this	void
long	record	throw	while
namespace	ref	true	
new	return	try	

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	from	nameof	set
alias	get	nint	unmanaged
alias	global	not	value
and	group	notnull	var
async	into	nuint	when
await	init	or	where
by	join	orderby	with
descending	let	partial	yield
dynamic	managed	remove	
equals	nameof	select	

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. Ein Beispiel ist das Semikolon, das eine Anweisung beendet. 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 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 248) 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 = 2022;
```

```
message = message + x.ToString();
Console.WriteLine (message);             // Hallo Welt2022
```

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` in .NET Core 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:

```
UnitConverter feetToInches = new UnitConverter (12);
UnitConverter milesToFeet = new UnitConverter (5280);

Console.WriteLine (feetToInches.Convert(30));   // 360
Console.WriteLine (feetToInches.Convert(100));  // 1200
Console.WriteLine (feetToInches.Convert
                   (milesToFeet.Convert(1)));  // 63360


public class UnitConverter
{
  int ratio;                              // Feld

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

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

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 unser Programm 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 Konstruktor 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, können als `static` gekennzeichnet werden. Um sich außerhalb eines Typs auf einen statischen Member von ihm zu beziehen, geben Sie statt einer *Instanz seinen Typnamen* an. Ein Beispiel ist die Methode `WriteLine` der Klasse `Console`. Da sie statisch ist, rufen wir `Console.WriteLine()` und nicht `new Console().WriteLine()` auf.

Der Unterschied zwischen Instanz- und statischen Membern ist dieser: Im folgenden Beispielcode gehört das Instanzfeld `Name` zu einer Instanz eines bestimmten `Panda`, während `Population` zur Menge aller `Panda`-Instanzen gehört. Wir erzeugen zwei Instanzen von `Panda`, geben ihre Namen und dann die gesamte `Population` 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

public class Panda
{
  public string Name;             // Instanzfeld
  public static int Population;   // statisches Feld

  public Panda (string n)         // Konstruktor
  {
    Name = n;                     // Instanzfeld
    Population = Population + 1   // statisches Feld
  }
}
```

Versuchen Sie, `p1.Population` oder `Panda.Name` auszuwerten, wird das zu einem Fehler beim Kompilieren führen.

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 könnte nicht von außerhalb der Klasse angesprochen werden. 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*.

Einen Namensraum erstellen

Insbesondere bei größeren Programmen ist es sinnvoll, Typen in Namensräumen zu organisieren. So definieren Sie die Klasse `Panda` innerhalb eines Namensraums `Animals`:

```
namespace Animals
{
  public class Panda
  {
    ...
  }
}
```

Wir behandeln Namensräume detaillierter im Abschnitt »Namensräume« auf Seite 71.

Eine Main-Methode definieren

Alle unsere Beispiele haben bisher Anweisungen auf oberster Ebene genutzt – ein Feature, das in C# 9 eingeführt wurde. Ohne solche Anweisungen sieht eine einfache Konsolen- oder Windows-Anwendung wie folgt aus:

```
using System;

class Program
{
  static void Main()   // Programm-Einstiegspunkt
  {
    int x = 12 * 30;
```

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

Gibt es keine Anweisungen auf oberster Ebene, sucht C# nach einer statischen Methode namens `Main`, die dann zum Einstiegspunkt wird. Die Methode `Main` kann innerhalb einer beliebigen Klasse definiert sein (und es kann nur eine Methode `Main` geben). Muss Ihre `Main`-Methode auf private Member einer bestimmten Klasse zugreifen können, kann es einfacher sein, eine `Main`-Methode innerhalb dieser Klasse zu definieren, statt Anweisungen auf oberster Ebene zu verwenden.

Die `Main`-Methode kann optional einen Integer-Wert zurückgeben (statt `void`), um der ausführenden Umgebung einen Wert mitzuteilen (wobei ein Wert ungleich null normalerweise für einen Fehler steht). Sie kann optional auch ein Array mit Strings als Parameter übernehmen (das mit allen Argumenten gefüllt wird, die an die ausführbare Datei übergeben werden) – zum Beispiel:

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

Ein Array (wie `string[]`) steht für eine feste Zahl an Elementen eines bestimmten Typs. Arrays werden spezifiziert, indem man eckige Klammern hinter den Typ des Elements schreibt. Wir beschreiben sie im Abschnitt »Arrays« auf Seite 38.

(Die Methode `Main` kann auch `async` deklariert werden und einen `Task` oder `Task<int>` zurückgeben, um asynchrone Programmierung zu unterstützen – siehe den Abschnitt »Asynchrone Funktionen« auf Seite 229.)

Anweisungen auf oberster Ebene

Anweisungen auf oberster Ebene ermöglichen Ihnen (seit C# 9), den Aufwand mit einer statischen `Main`-Methode und einer Klasse, die diese Methode enthält, zu vermeiden. Eine Datei mit Anweisungen auf oberster Ebene besteht aus drei Abschnitten in dieser Reihenfolge:

1. (optional) `using`-Direktiven
2. eine Folge von Anweisungen, optional vermischt mit Methodendeklarationen
3. (optional) Typ- und Namensraumdeklarationen

Alles in Abschnitt 2 landet in einer vom Compiler generierten `Main`-Methode innerhalb einer ebenso generierten Klasse. Das heißt, dass die Methoden in Ihren Anweisungen auf oberster Ebene zu *lokalen Methoden* werden (deren Feinheiten wir im Abschnitt »Lokale Methoden« auf Seite 78 beschreiben). Anweisungen auf oberster Ebene können optional einen Integer-Wert an den Aufrufenden zurückgeben und auf eine »magische« Variable namens `args` vom Typ `string[]` zugreifen, in der die vom Aufrufenden übergebenen Befehlszeilenargumente enthalten sind.

Da ein Programm nur einen Einstiegspunkt haben kann, kann es in einem C#-Projekt auch nur höchstens eine Datei mit Anweisungen auf oberster Ebene geben.

Typen und 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; }
```

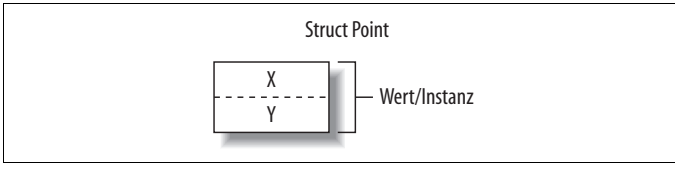

Abbildung 1: Eine Werttyp-Instanz im Speicher

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.

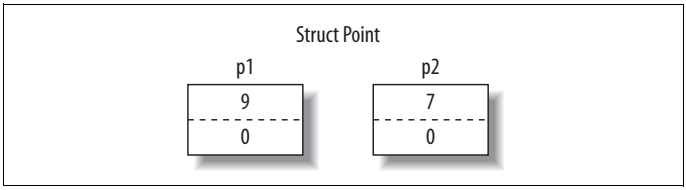

Abbildung 2: Eine Zuweisung kopiert eine Werttyp-Instanz.

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; }
```

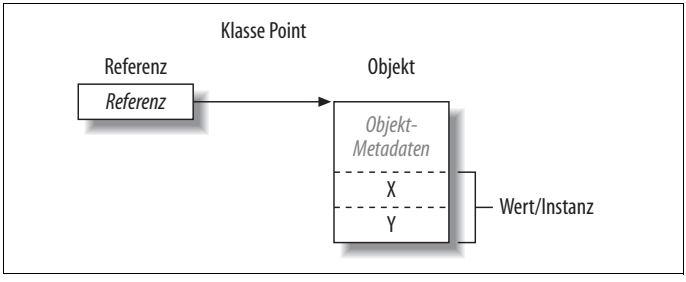

Abbildung 3: Ein Referenztyp im Speicher

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.

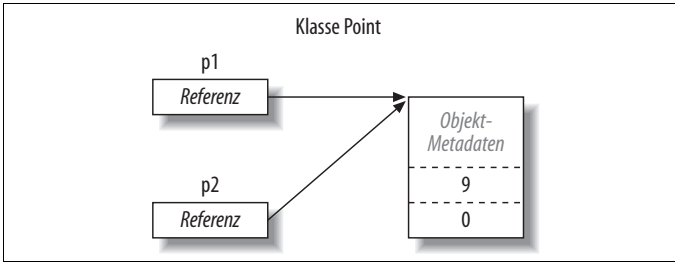

Abbildung 4: Eine Zuweisung kopiert eine Referenz.

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 Abschnitt »Nullbare Referenztypen« auf Seite 167 beschreiben wir ein Feature von C#, mit dem sich ungewollte `NullReferenceException`-Fehler verhindern lassen.

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 Werttypen« auf Seite 162).

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-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 unmit-

telbare 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 $2^{63} - 1$
`nint`	`IntPtr`		32/64 Bit	
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 $2^{64} - 1$
`unint`	`UIntPtr`		32/64 Bit	
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. Die ganzzahligen Typen `nint` und `nuint` (eingeführt in C# 9) haben eine Größe, die zum Adressraum des Laufzeitprozesses passt. Sie können bei der Arbeit mit Zeiger-Arithmetik hilfreich sein. Wir beschreiben sie detailliert in Kapitel 4 von *C# 10 in a Nutshell*.

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-, Hexadezimal- oder Binärnotation dargestellt werden; die Hexadezimalnotation wird mit dem Präfix `0x` angezeigt (z. B. entspricht `0x7f` dem Dezimalwert `127`), die Binärnotation mit dem Präfix `0b`. *Reelle Literale* können die Dezimal- oder die Exponentialnotation nutzen, z. B. `1E06`. Numerische Literale können 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 Exponentialsymbol (`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.