O'REILLY®

5. Auflage

C# 7.0

kurz & gut

O'REILLYS TASCHEN-BIBLIOTHEK

Joseph Albahari
Ben Albahari

Übersetzung von
Lars Schulten und Thomas Demmig

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; }
```

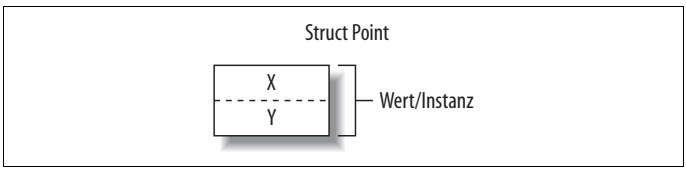

Abbildung 1: Eine Werttyp-Instanz im Speicher

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.

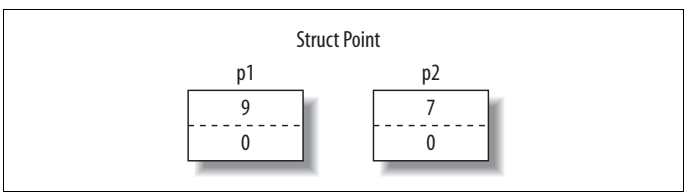

Abbildung 2: Eine Zuweisung kopiert eine Werttyp-Instanz.