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`, `out` oder `in`), 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 abgelegt werden. Jeder hat bezüglich der Lebensspanne der Objekte eine andere Semantik.

### Stack

Der Stack ist der Speicher, in dem lokale Variablen und Parameter gespeichert werden. Der Stack wächst und schrumpft, sobald Methoden oder 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 der Speicher, 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 43).
- 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 kann eine Reihe von Parametern haben. 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, in 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* |
| in | Referenz (schreibgeschützt) | beim *Betreten* |

### 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 String-

Builder-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

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 in

Seit C# 7.2 können Sie einem Parameter den Modifikator `in` voranstellen, um zu verhindern, dass er in der Methode verändert wird.

Damit spart sich der Compiler den Overhead, der beim Kopieren des Arguments vor der Übergabe entsteht – was gerade bei großen, eigenen Werttypen ins Gewicht fallen kann (siehe Abschnitt »Structs« auf Seite 96).

### 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

Methoden, Konstruktoren und Indexer können *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 159 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
```