3.5.6 **Referenz- und Wertparameter**

Parameter mit dem Modifizierer »ref«

Nehmen wir nun zwei kleine Änderungen am Listing 3.27 vor. Zuerst wird der Methodenaufruf in `Main` wie folgt codiert:

```
DoSomething(ref value);
```

Im zweiten Schritt ergänzen wir in ähnlicher Weise auch die Parameterliste von `DoSomething`:

```
public void DoSomething(ref int param) {...}
```

Wenn Sie jetzt das Beispiel erneut starten, wird das zu folgender Ausgabe führen:

```
value = 550
```

Die Ergänzung sowohl des Methodenaufrufs als auch der Parameterliste um das Schlüsselwort `ref` hat also bedeutende Konsequenzen für die lokale Variable *value* – sie hat nach dem Methodenaufruf genau den Inhalt angenommen, der dem Parameter *param* zugewiesen worden ist. Wie ist das zu erklären?

Beim Aufruf von `DoSomething` wird nicht mehr der Inhalt der Variablen *value* übergeben, sondern deren Speicheradresse, also 1000. Der empfangende Parameter *param* muss selbstverständlich wissen, was ihn erwartet (nämlich eine Speicheradresse), und wird daher ebenfalls mit `ref` definiert. Für *param* muss die Methode natürlich auch weiterhin Speicher allokieren – gehen wir auch in diesem Fall noch einmal von der Adresse 2000 aus. Alle Aufrufe an *param* werden nun jedoch an die Adresse *1000* umgeleitet. Die Methode `DoSomething` weist dem Parameter *param* die Zahl 550 zu, die in die Adresse 1000 geschrieben wird. Damit gilt:

Adresse param = Adresse value = 550

Nachdem der Programmablauf an die aufrufende Methode zurückgegeben worden ist, wird an der Konsole der Inhalt der Variablen *value* – also der Inhalt, der unter der Adresse 1000 zu finden ist – angezeigt: Es handelt sich um die Zahl 550. Diese Technik der Parameterübergabe wird als **Referenzübergabe** (engl.: *Call by Reference*) bezeichnet.

Parameter ohne zusätzlichen Modifizierer

Sehen Sie sich das folgende Beispiel an:

```
// Beispiel: ..\Kapitel 3\Wertuebergabe

class Program {

  static void Main(string[] args) {
    int value = 3;
    DoSomething(value);
    Console.WriteLine("value = {0}", value);
    Console.ReadLine();
  }

  static void DoSomething(int param) {
    param = 550;
  }
}
```

Listing 3.27 Parameterübergabe (Call by Value)

In `Main` wird die lokale Variable *value* deklariert und danach der Methode `DoSomething` als Argument übergeben. Die Methode `DoSomething` nimmt das Argument im Parameter *param* entgegen und ändert danach den Inhalt von *param* in 550. Nachdem der Methodenaufruf beendet ist, wird der Inhalt der lokalen Variablen *value* in die Konsole geschrieben. Wenn Sie das Programm starten, lautet die Ausgabe an der Konsole:

value = 3

Der Inhalt der lokalen Variablen *value* hat sich nach dem Aufruf der Methode `DoSomething` nicht verändert.

Um zu verstehen, was sich bei diesem Methodenaufruf abspielt, müssen wir einen Blick in den Teilbereich des Speichers werfen, in dem die Daten vorgehalten werden. Zunächst wird für die Variable *value* Speicher allokiert. Nehmen wir an, es sei die Speicheradresse 1000. In diese Speicherzelle (genau genommen sind es natürlich vier Byte, die ein Integer für sich beansprucht) wird die Zahl 3 geschrieben.

Ein Parameter unterscheidet sich nicht von einer lokalen Variablen. Genau das ist der entscheidende Punkt, denn folgerichtig ist ein Parameter ebenfalls ein Synonym für eine bestimmte Adresse im Speicher. Mit der Übergabe des Arguments *value* beim Methodenaufruf wird von `DoSomething` zunächst Speicher für den Parameter *param* allokiert – wir gehen von der Adresse 2000 aus. Danach wird der Inhalt des Arguments *value* – also der Wert 3 – in die Speicherzelle 2000 kopiert.

Ändert `DoSomething` den Inhalt von *param*, wird die Änderung in die Adresse 2000 geschrieben. Damit weisen die beiden in unserem Beispiel angenommenen Speicheradressen die folgenden Inhalte auf:

Adresse 1000 = 3

Adresse 2000 = 550

Nachdem der Programmablauf zu der aufrufenden Methode zurückgekehrt ist, wird der Inhalt der Variablen *value*, also der Inhalt der Speicheradresse 1000 an der Konsole ausgegeben: Es ist die Zahl 3. Diese Technik der Argumentübergabe wird als **Wertübergabe** (engl.: *Call by Value*) bezeichnet.

Parameter mit dem Modifizierer »out«

Parameter mit dem Modifizierer »out«

Zusätzlich zu diesen beiden Übergabetechniken kann ein Methodenparameter auch mit `out` spezifiziert werden, der in derselben Weise wie `ref` verwendet wird: Er muss sowohl als Modifizierer des Übergabearguments wie auch als Modifizierer des empfangenen Parameters in der Methodendefinition angegeben werden. Obwohl der Effekt, der mit `out` erzielt werden kann, derselbe wie bei `ref` ist, gibt es zwischen den beiden zwei Unterschiede:

- Während die Übergabe einer nicht initialisierten Variablen mit `ref` zu einem Kompilierfehler führt, ist dies bei `out` zulässig.
- Innerhalb der Methode muss einem `out`-Parameter ein Wert zugewiesen werden, während das bei einem `ref`-Parameter nicht zwingend notwendig ist.

3.5.7 **Besondere Aspekte einer Parameterliste**

Der Modifizierer »params«

Stellen Sie sich vor, Sie beabsichtigen, eine Methode zu entwickeln, um Zahlen zu addieren. Eine Addition ist nur dann sinnvoll, wenn aus wenigstens zwei Zahlen eine Summe gebildet wird. Daher definieren Sie die Methode wie folgt:

```
public long Add(int value1, int value2) {
  return value1 + value2;
}
```

Vielleicht haben Sie danach noch die geniale Idee, nicht nur zwei Zahlen, sondern drei bzw. vier zu addieren. Um dieser Forderung zu genügen, könnten Sie die Methode `Add` wie folgt überladen:

```
public long Add(int value1, int value2, int value3) {[...]}

public long Add(int value1, int value2, int value3, int value4) {[...]}
```

Wenn Ihnen dieser Ansatz kritiklos gefällt, sollten Sie sich mit der Frage auseinandersetzen, wie viele überladene Methoden Sie maximal zu schreiben bereit sind, wenn möglicherweise nicht nur vier, sondern 10 oder 25 oder beliebig viele Zahlen addiert werden sollen.

Es muss für diese Problemstellung eine bessere Lösung geben – und es gibt sie auch: Sie definieren einen Parameter mit dem Modifizierer `params`. Dieser gestattet es, einer Methode eine beliebige Anzahl von Argumenten zu übergeben. Die Übergabewerte werden der Reihe nach in ein Array geschrieben.

Nun kann die Methode `Add` diesen Feinschliff erhalten. Da eine Addition voraussetzt, dass zumindest zwei Summanden an der Operation beteiligt sind, werden zuerst zwei konkrete Parameter definiert und anschließend ein `params`-Parameter für alle weiteren Werte.

```
public long Add(int value1, int value2, params int[] list) {
  long sum = value1 + value2;
  foreach(int z in list)
    sum += z;
  return sum;
}
```

Listing 3.31 Der »params«-Parameter

Werden einem `params`-Parameter Werte zugewiesen, wird das Array anhand der Anzahl der übergebenen Argumente implizit dimensioniert. In unserem Beispiel werden alle Elemente des Arrays in einer Schleife addiert und in der lokalen Variablen `sum` zwischengespeichert. Nachdem für das letzte Element die Schleife durchlaufen ist, wird mit `return` das Ergebnis an den Aufrufer übermittelt.

Mit einem `params`-Parameter sind ein paar Regeln verbunden, die eingehalten werden müssen:

- In der Parameterliste darf nur ein Parameter mit `params` festgelegt werden.
- Ein `params`-Parameter steht immer an letzter Position in einer Parameterliste.
- Eine Kombination mit den Modifikatoren `out` oder `ref` ist unzulässig.
- Ein `params`-Parameter ist grundsätzlich eindimensional.

Wenn Sie eine Methode aufrufen, die einen `params`-Parameter enthält, haben Sie zwei Möglichkeiten, diesem Werte zuzuweisen:

Optionale Parameter

Optionale Parameter

Als optionale Parameter werden Methodenparameter bezeichnet, die beim Aufruf in der Parameterliste nicht übergeben werden müssen. Optionale Parameter sind daran zu erkennen, dass ihnen in der Methodendefinition ein Standardwert zugewiesen wird. Wird dem optionalen Parameter beim Methodenaufruf nicht ausdrücklich ein Wert übergeben, behält der optionale Parameter den Standardwert.

Folgendes Codefragment zeigt die Implementierung der Methode `DoSomething`, die mit `value` einen optionalen Parameter beschreibt, dessen Standardwert `-1` ist.

```
public void DoSomething(string name, int value = -1)
{
  [...]
}
```

Listing 3.32 Methode mit optionalem Parameter

Hat eine Methode sowohl feste als auch optionale Parameter, sind zuerst die festen und danach die optionalen anzugeben.

Methodenaufruf mittels benannter Argumente

Eine besondere Rolle kommt den benannten Argumenten im Zusammenhang mit Methoden zu, die mehrere optionale Parameter haben. Angenommen, eine Methode definiert vier optionale Parameter, beispielsweise

```
public void DoSomething(int a = 10, int b = 3, int c = -5, int d = 5) {[...]}
```

Ohne die syntaktische Fähigkeit benannter Argumente bliebe Ihnen nur übrig, allen Parametern ausdrücklich einen Wert zuzuweisen. Mit

```
@object.DoSomething(d: 4711);
```

wird aber die Zuweisung an den vierten optionalen Parameter zu einer sehr überschaubaren und auch gut lesbaren Angelegenheit.

3.5.9 **Die Trennung von Daten und Code**

Ein Objekt besteht im Wesentlichen aus Eigenschaften und Methoden. Eigenschaften sind im Grunde genommen nichts anderes als Elemente, die objektspezifische Daten enthalten. Objekte werden meist durch mehrere Eigenschaften beschrieben. Für jedes Feld wird entsprechender Speicher reserviert, für einen Integer beispielsweise vier Byte. Alle Eigenschaften eines Objekts sind natürlich nicht wild verstreut im Speicher zu finden, sondern in einem zusammenhängenden Block.

Typgleiche Objekte reservieren grundsätzlich gleich große Datenblöcke, deren interne Struktur vollkommen identisch aufgebaut ist. Wenn Sie in Ihrem Code die Objektvariable der Klasse `Circle` deklarieren, wird Speicherbereich reserviert, der groß genug ist, um alle Daten aufzunehmen. Mit

```
Circle kreis1 = new Circle();
```

zeigt die Objektvariable `kreis1` auf die Startadresse dieses Datenblocks im Speicher: Sie referenziert das Objekt. Daher stammt auch die gebräuchliche Bezeichnung **Objektreferenz**.

Das Speicherprinzip ist in der folgenden Abbildung 3.4 anhand der beiden Objekte *kreis1* und *kreis2* dargestellt. Tatsächlich sind die Vorgänge zur Laufzeit deutlich komplexer, aber zum Verständnis des Begriffs »Datenblock« und zur Erkenntnis, dass sich hinter jeder Objektvariablen eigentlich eine Speicheradresse verbirgt, trägt die Abbildung anschaulich bei.

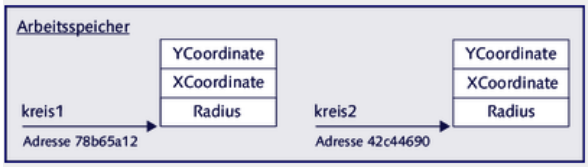

Abbildung 3.4 Prinzipielle Verwaltung von Objekten im Arbeitsspeicher

Jedes Objekt beansprucht einen eigenen Datenblock. Diese Notwendigkeit besteht nicht für die Methoden, also den Code einer Klasse. Dieser befindet sich nur einmal »en bloc« im Speicher. Der Code arbeitet zwar mit den Daten eines Objekts, ist aber trotzdem völlig unabhängig von diesen. Im objektorientierten Sprachgebrauch wird dies auch als die **Trennung von Code und Daten** bezeichnet. Der Code der Methoden wird nur einmal im Speicher abgelegt, und zwar auch dann, wenn noch kein Objekt dieses Typs existiert.

3.5.12 Umbenennen von Methoden und Eigenschaften

3.5.12 Umbenennen von Methoden und Eigenschaften▲

Häufig werden Sie Programmcode schreiben und Variablen- oder Methodenbezeichner wählen, die Sie später ändern wollen. An dieser Stelle sei daher auch noch ein Hinweis gegeben, wie Sie mit der Unterstützung von Visual Studio 2012 auf sehr einfache Weise Methoden oder auch Eigenschaften umbenennen können.

Setzen Sie dazu den Eingabecursor auf den umzubenennenden Bezeichner. Öffnen Sie das Kontextmenü, und wählen Sie hier UMGESTALTEN und dann UMBENENNEN (siehe Abbildung 3.5). Ändern Sie nun den Bezeichner ab. Visual Studio 2012 wird Ihnen auch in einem weiteren Fenster anzeigen, welche Stellen im Code von der Änderung betroffen sind (z. B. Methodenaufrufe), und diese nach Bestätigung ebenfalls an den neuen Bezeichner anpassen.

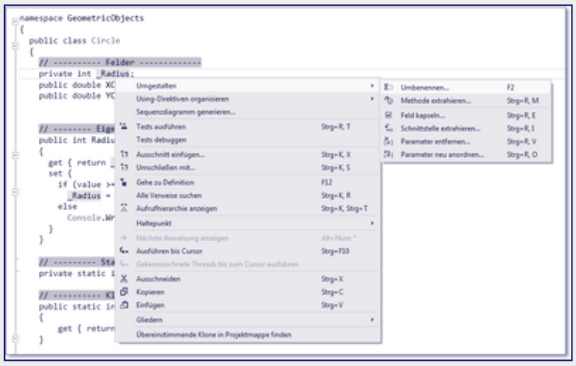

Abbildung 3.5 Umbenennen von Eigenschaften und Methoden

Alternativ können Sie das Umbenennen eines Bezeichners auch aus dem Menü UMGESTALTEN heraus erreichen.