Parameter-Übergabe-Mechanismen

in Java und in anderen Sprachen.

Inhaltsverzeichnis

1. Methoden vereinbaren mit Parametern

Wenn man (z.B. in Java) eine Methode *vereinbart*, legt man die Anzahl und die Typen ihrer *Parameter* fest.

Beispiel-01: Vereinbarung einer Methode in Java:

```
17  static void machWas(int n, String s) {
18     System.out.println(s + ": " + n);
19  }
```

Die Methode `machWas` wurde hier mit *zwei* Parametern *vereinbart*.
Parameter sind Variablen, die auf eine spezielle Weise initialisiert werden (nämlich erst und jedes mal, wenn die Methode *aufgerufen* wird).

2. Methoden aufrufen mit Argumenten

Jedes mal, wenn man eine Methode *aufruft*, muss man für jeden *Parameter* ein entsprechendes *Argument* angeben. Diese Argumente werden an die Methode *übergeben* und dann wird die Methode ausgeführt.

Beispiel-02: Mehrere Aufrufe der Methode `machWas`

```
25     ...
26     int    alter = 25;
27     String name  = "Karl";
28                                          // Ausgabe auf dem Bildschirm:
29     machWas(25, "Karl");                 // Karl: 25
30     machWas(alter, name);                // Karl: 25
31     machWas(2*alter+1, name + "-Heinz"); // Karl-Heinz: 51
32     ...
```

3. Parameter-Übergabe per Wert

Grundsätzlich können Argumente auf verschiedene Weisen an eine Methode übergeben werden. Die entsprechenden Mechanismen bezeichnet man als *Parameter-Übergabe-Mechanismen* (obwohl "Argumente-Übergabe-Mechanismen" logischer wäre). In Java gibt es nur *einen* solchen Parameter-Übergabe-Mechanismus:

Die **Übergabe per Wert** (engl.: pass by value)

1. Als Argument ist ein *beliebiger Ausdruck* (des betreffenden Typs) erlaubt.
2. An die Methode übergeben wird der ("ausgerechnete") *Wert* des Ausdrucks (nicht der Ausdruck!).

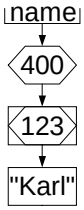
Genauer: Ein *Parameter* ist eine lokale Variable der betreffenden Methode, die bei jedem Aufruf der Methode neu erzeugt wird und dabei mit dem Wert des betreffenden *Arguments* initialisiert wird.

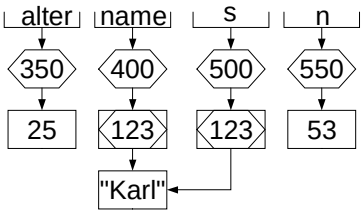
Beispiel-03:

```
31      ...
32      machWas(2*alter+3, name);
33      ...
```

In Zeile 32 wird die Methode `machWas` mit den Argumenten `2*alter+3` und `name` aufgerufen. Der Ausdruck `2*alter+3` hat den Wert `53`.

Die Variable `name` sieht als Boje z.B. wie hier rechts aus:
Der Ausdruck `name` hat dann den (Referenz-) Wert `[<123>]`, der auf das `String`-Objekt `["Karl"]` zeigt.





Durch den Aufruf in Zeile 32 werden also die Werte `53` und `[<123>]` an die Methode `machWas` übergeben, und mit diesen Werten werden die Parameter `n` und `s` initialisiert. Während der Ausführung des Aufrufs in Zeile 32 sehen die Variablen `alter`, `name`, `s` und `n` als Bojen etwa wie hier links aus.

Man kann sich das auch so vorstellen: Durch den *Aufruf* der Methode `machWas` in Zeile `32` wird in der *Vereinbarung* der Methode (siehe oben Abschnitt 1.) die erste Zeile wie folgt verändert:

```
17  void machWas(int n = 53, s = [<123>]) {
18      ...
```

und dann wird die Methode ausgeführt. Eine entsprechende Veränderung der ersten Zeile (so kann man sich vorstellen) findet bei jedem Aufruf (vor jeder Ausführung) der Methode statt.

Hinweis: Ein Befehl wie `int s = [<123>]` ist in einem Java-Programm (und ganz besonders in der Vereinbarung einer Methode) *nicht erlaubt*, für Bojen-Kenner aber hoffentlich verständlich.

4. Parameter-Übergabe per Referenz

Dass Java nur *einen* Parameter-Übergabe-Mechanismus hat (*Übergabe per Wert*), trägt zur Einfachheit und Übersichtlichkeit der Sprache bei.

In anderen Sprachen (z.B. in Ada, C++ und C#) gibt es noch einen anderen Parameter-Übergabe-Mechanismus, nämlich:

Die **Übergabe per Referenz** (engl.: pass by reference)

1. Als Argument ist nur eine *Variable* (des betreffenden Typs) erlaubt.
2. An die Methode übergeben wird die *Referenz* der Variablen (nicht ihr Wert oder ihr Name).

5. Parameter-Übergabe per Name

Dieser Übergabe-Mechanismus wurde ca. 1960 erfunden und in die Sprache Algol60 eingebaut.

Die **Übergabe per Name** (engl.: pass by name)

1. Als Argument ist ein *beliebiger Ausdruck* (des betreffenden Typs) erlaubt.
2. Übergeben wird der *Ausdruck* (nicht der Wert des Ausdrucks und nicht irgendeine Referenz)

Die Bezeichnung "Übergabe per Name" erklärt sich so: Wenn man als Argument nur eine Variable angibt, dann wird der Name der Variablen an die Methode übergeben.

Dieser Übergabe-Mechanismus hat interessante Eigenschaften, ist aber schwierig zu implementieren und wurde nach Algol60 in keine weiter verbreitete Sprache übernommen.

6. Aufgaben die man mit der Übergabe per Wert nicht lösen kann

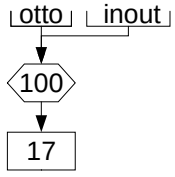
Unmöglich: Man kann in Java (mit der Übergabe *per Wert*) keine Methode schreiben, die als Argument eine *Variable* `v` erwartet und den *Wert* von `v` verändert.

In Sprachen mit Übergabe *per Referenz* (z.B. in Ada, C++ oder C#) kann man solche Methoden schreiben (und tut es häufig).

Beispiel-04: Den Wert einer `int`-Variablen von einer Methode verändern lassen
Geht nicht in Java, sieht aber in C# etwa so aus:

```
 1   void plus1(ref int inout) {
 2      // Der Wert von inout wird um 1 erhoeht
 3      inout++;
 4   }
 5   ...
 6      // Zwei Aufrufe der Methode plus1:
 7      int otto = 17;     // Hier  hat otto den Wert 17
 8      plus1(ref otto);   // Jetzt hat otto den Wert 18
 9      ...
10      plus1(ref otto);   // Jetzt hat otto den Wert 19
11      ...
```

Das Schlüsselwort `ref` in Zeile 1 und in Zeile 8 und 10 bewirkt, dass der Parameter `inout` *per Referenz* übergeben wird (und nicht *per Wert*). Während der Methoden-Aufruf in Zeile 8 ausgeführt wird, sind `otto` und `inout` zwei Namen für dieselbe `int`-Variable. Als Boje sieht das etwa so aus wie im Diagramm hier rechts:



Wenn der Wert von `inout` verändert wird (siehe Zeile 3) wird damit auch der Wert von `otto` verändert.

Der Name `inout` soll darauf hinweisen, dass mit diesem Parameter ein Wert in die Methode `plus1` *hinein* transportiert wird (im Beispiel: `17`) und ein Wert aus der Methode *hinaus* transportiert wird (im Beispiel: `18`).

7. Zusammenfassung

Wenn man in Java in einem Methodenaufruf eine *Referenzvariable* als Argument angibt, wird der Methode der *Wert* dieser Variablen übergeben. Die Variable wird also *per Wert* übergeben.

Wenn dieser (Referenz-) Wert auf ein Objekt zeigt (d.h. wenn er ungleich `null` ist), kann man die Übergabe auch so beschreiben: Das Objekt wird *per Referenz* übergeben (d.h. die Methode bekommt nicht eine Kopie des Objekts, sondern nur eine Kopie der Referenz, die auf das Objekt zeigt).

In **Java** gilt:

Objekte kann man nur **per Referenz** übergeben (nicht per Wert).
Referenzen (und primitive Werte) kann man nur **per Wert** übergeben (nicht per Referenz).

Anmerkung-01: Anstelle von "Parameter" und "Argument" sind auch die Bezeichnungen "formaler Parameter" und "aktueller Parameter" gebräuchlich.

Anmerkung-02: Anstelle von *pass by value* bzw. *pass by reference* sind die Bezeichnungen *call by value* und *call by reference* weit verbreitet, aber sprachlich verkorkst: *Aufgerufen* werden Methoden (methods are *called*), Parameter werden *übergeben* (parameters are *passed*). Man sollte die verkorksten Bezeichnungen verstehen, sie aber selbst möglichst nicht äußern.

8. Wie kann man die Übergabe per Referenz in Java nachmachen?

Wenn man daran gewöhnt ist, die Vorteile der Übergabe per Referenz zu benutzen, dann vermisst man sie in Java manchmal (z.B. beim Implementieren von binären Bäumen). Man kann sie in solchen Fällen aber "nachmachen".

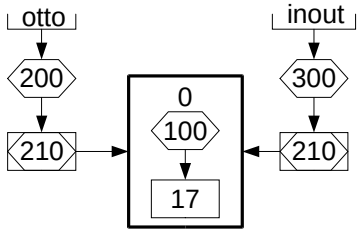
Grundidee für das "Nachmachen" der *Übergabe per Referenz*: Wenn man in Java einer Methode eine *Reihung* `r` übergibt, dann kann die Methode die Werte der Komponenten `r[i]` verändern.
Die folgende Methode soll eine einfache Anwendung dieser Grundidee zeigen:

Beispiel-05: Den Wert einer `int`-Variablen von einer Methode verändern lassen

```
 1    public static void plus3(int[] inout) {
 2       // Der Wert von inout[0] wird um 3 erhoeht.
 3       inout[0] = inout[0] + 3;
 4    }
 5
 6    // Zwei Aufrufe der Methode plus3:
 7       ...
 8       int[] otto = {17};  // Hier  hat otto[0] den Wert 17
 9       plus3(otto);        // Jetzt hat otto[0] den Wert 20
10       ...
11       plus3(otto);        // Jetzt hat otto[0] den Wert 23
12       ...
```

In Zeile 8 wird eine Variable namens `otto` (vom Typ *Reihung von int*) vereinbart. Aber eigentlich geht es hier um die Variable `otto[0]` vom Typ `int`. Diese Variable (so sollte man den Methodenaufruf in Zeile 9 lesen und verstehen) wird (in einem bestimmten Sinne *per Referenz*) an die Methode `plus3` übergeben und von ihr verändert.

Während der Methoden-Aufruf `plus3(otto);` (in Zeile 9) ausgeführt wird, sehen die Variablen `otto` und `inout` als Bojen etwa so aus wie im Diagramm hier rechts:



Wenn `inout[0]` um 3 erhöht wird (siehe oben Zeile 3) wird damit auch `otto[0]` um `3` erhöht.

Das **Beispiel-05** macht vermutlich keinen sehr nützlichen Eindruck. Das folgende Beispiel ist auch noch nicht besonders nützlich, aber einer nützlichen Anwendung schon etwas näher.

Eine *Funktion* (a non-void method) kann (in Java und vielen anderen Sprachen) nur *einen* Rückgabewert als Ergebnis liefern (mit dem `return`-Befehl). Mit der *Übergabe per Referenz* (auch wenn man sie in Java nur nachgemacht hat) kann man es einrichten, dass eine Methode *mehrere* Ergebnisse berechnet und in bestimmten Parametern ablegt, wie im folgenden Beispiel:

Beispiel-06: Eine Methode mit einem in-Parameter und zwei out-Parametern

```
 1    public static void quadrat(int in1, int[] out1, String[] out2) {
 2       // Schreibt das Quadrat von in1
 3       // (als int-Wert)      nach out1[0] und
 4       // (als String-Objekt) nach out2[0].
 5
 6       out1[0] = in1 * in1;                      // Ergebnis 1
 7       out2[0] = String.format("%+,d", out1[0]); // Ergebnis 2
 8    }
 9
10    // Ein Aufruf der Methode quadrat:
11       ...
12       int[]    outA = new int[1];
13       String[] outB = new String[1];
14       quadrat(100, outA, outB);  // Ausgabe:
15       println(outA[0]);          // 10000
16       println(outB[0]);          // +10.000
17       ...
```

Der in-Parameter `in1` dient dazu, "einen Wert in die Methode `quadrat` *hinein* zu transportieren", `out1` und `out2` dienen dazu, "je einen Wert aus der Methode *hinaus* zu transportieren".

Nach dem Muster der Methoden `plus3` und `quadrat` kann man (auch in Java) Methoden mit beliebig vielen **in**- , **out**- und **inout**-Parametern schreiben, nach der Regel:

in-Parameter sind ganz normale Parameter ("wie sonst auch"),
inout- und **out**-Parameter sind Reihungen (der Länge 1).

9. Neu in Java 8: Lambda-Ausdrücke

Schon vor Java 8 war es möglich, aber ziemlich umständlich, einer Methode `m` eine *Methode* `ma` als Argument zu übergeben. Ab Java 8 ist das einfacher geworden, weil man die Methode `ma` durch einen sogenannten *Lambda-Ausdruck* beschreiben kann.

Beispiel-05: Lambda-Ausdrücke als Argumente in Methoden-Aufrufen

```
 1 @FunctionalInterface
 2 interface Fi2i { // Funktionen mit int-Ergebnis und einem int Parameter
 3    abstract public int wendeAn(int n);
 4 }
 5
 6 public class Lambda01 {
 7    // ------------------------------------------------------------------------
 8    static void repeat(Fi2i f) {
 9       // Gibt die 5 Werte 1, f(1), f(f(1)), f(f(f(1))), f(f(f(f(1)))) aus.
10
11       int n = 1;
12       for (int i=1; i<=5; i++) {
13          printf("%3d ", n); // Gib einen Wert aus
14          n = f.wendeAn(n);  // Berechne den naechsten Wert
15       }
16       printf("%n");
17    }
18    // ------------------------------------------------------------------------
19    static public void main(String[] sonja) {
20       printf("Lambda01: Jetzt geht es los!%n");
21       printf("--------------------------%n");
22       printf("repeat(n->n+1): "); repeat(n->n+1);
23       printf("repeat(n->n+5): "); repeat(n->n+5);
24       printf("repeat(n->2*n): "); repeat(n->2*n);
25       printf("repeat(n->5*n): "); repeat(n->5*n);
26       printf("--------------------------%n");
27       printf("Lambda01: Das war's erstmal!%n");
28    } // main
29    // ------------------------------------------------------------------------
30    // Eine Methode mit einem kurzen Namen:
```

```
31    static void printf(String f, Object... v) {System.out.printf(f, v);}
32    // ---------------------------------------------------------------------
33 } // class Lambda01
34 /* ---------------------------------------------------------------------
35 Ausgabe:
36
37 Lambda01: Jetzt geht es los!
38 ---------------------------
39 repeat(n->n+1):   1   2   3   4   5
40 repeat(n->n+5):   1   6  11  16  21
41 repeat(n->2*n):   1   2   4   8  16
42 repeat(n->5*n):   1   5  25 125 625
43 ---------------------------
44 Lambda01: Das war's erstmal!
45 ------------------------------------------------------------------------ */
```

`Fi2i` ist eine *funktionale Schnittstelle* engl.: a functional interface), d.h. eine Schnittstelle, die genau *eine* `abstract public` Methode enthält.

Jedes Objekt des Typs `Fi2i` enthält garantiert eine Funktion mit einem `int`-Parameter und dem Rückgabetyp `int`. Man spricht und schreibt deshalb über `Fi2i`-Objekte manchmal so, als wären sie solche Funktionen.

Die Methode `repeat` hat einen Parameter namens `f` vom Typ `Fi2i`. Man sagt deshalb auch: `repeat` erwartet als Argument eine Funktion vom Typ `Fi2i`. Diese Funktion hat innerhalb der Methode `repeat` den Namen `f.wendeAn` ("`wendeAn`" wurde in der Schnittstelle `Fi2i` festgelegt, und "`f`" ist der Name des Parameters von `repeat`). Die Funktion `f.wendeAn` wird im Rumpf der `for`-Schleife 5 Mal aufgerufen.

In Zeile 22 wird die Methode `repeat` (zum ersten Mal) aufgerufen. Als Argument ist dort der Lambda-Ausdruck `n->n+1` angegeben. Dieser Lambda-Ausdruck bezeichnet eine Funktion, die man umständlicher auch so hätte vereinbaren können:

```
int wendeAn(int n) {
   return n+1;
}
```

In den Zeilen 23 bis 25 wird die Methode `repeat` mit anderen Lambda-Ausdrücken als Argument aufgerufen. Die Ausgaben des Programms sieht man in den Zeilen 37 bis 44.

In vielen Programmiersprachen kann man einem Unterprogramm `U` ein Unterprogramm `UA` als Argument übergeben. Die Übergabe erfolgt gewöhnlich *per Referenz*.

In Java kann man einer Methode `m` eine Methode `ma` als Argument übergeben. Die Übergabe erfolgt allerdings (egal ob man `ma` durch einen einfachen Lambda-Ausdruck oder durch eine "altmodische und umständlichere" Vereinbarung beschreibt) immer *per Wert* wie folgt:

Die zu übergebende Methode `ma` wird in ein Objekt "eingewickelt" und die Referenz dieses Objekts wird (per Wert) an die Methode `m` übergeben.

Für den Java-*Ausführer* ist die Übergabe eines Methoden-Arguments an eine Methode nicht leichter geworden, aber mit den Lambda-Ausdrücken seit Java 8 kann der *Programmierer* eine solche Übergabe einfacher und übersichtlicher beschreiben.