

Programmieren I

Methoden-Special



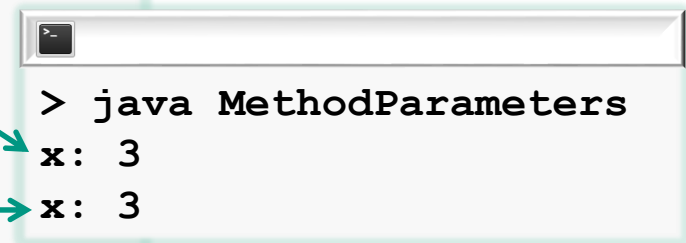
Heusch ---
Ratz 6.1, 6.2

Institut für Automation und angewandte Informatik

```
final List<String> allResults = new ArrayList<String>();  
final Map<String, Integer> typeWordResultCount = new HashMap<String, Integer>();  
final Map<String, Integer> typePoints = new HashMap<String, Integer>();  
evaluation.put(type, typePoints);  
  
for (final Sheet sheet : this.sheets) {  
    final String sheetResult = sheet.getPlayerInput(type);  
    if (sheetResult.startsWith(start) && this.isValidWord(sheetResult, type)) {  
        validWordCountForType++;  
        allResults.add(sheetResult);  
    }  
}
```

Parameterübergabe: Wertkopie („Call-By-Value“)

```
public class MethodParameters {  
  
    public static void someMethod(int a) {  
        a += 2;  
    }  
  
    public static void main(String[] args) {  
        int x = 3;  
  
        System.out.println("x: " + x);  
        someMethod(x);  
        System.out.println("x: " + x);  
    }  
}
```



```
> java MethodParameters  
x: 3  
x: 3
```

Der Parameter wird als Kopie des Wertes (Wertkopie, „Call-By-Value“) übergeben. Keine Veränderung des ursprünglichen Wertes!

Parameterübergabe: Referenzkopie („Call-By-Reference“)

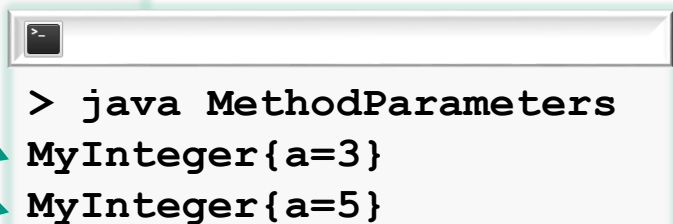
```
public class MyInteger {
    int a;
    public MyInteger( int a ){ this.a = a; }
    public String toString() {
        return "MyInteger{a=" + a + "}";
    }
}

public class MethodParameters {

    public static void someMethod(MyInteger ma){
        ma.a += 2;
    }

    public static void main(String[] args) {
        MyInteger mi = new MyInteger(3);
        System.out.println(mi);
        someMethod(mi);
        System.out.println(mi);
    }
}
```

**Parameter ma enthält
eine Kopie der Referenz
des Objektes.
→ Veränderung des
Objektes!**



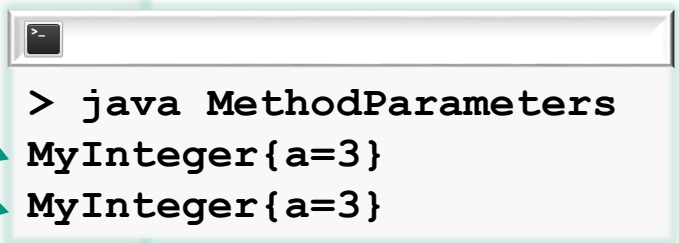
```
> java MethodParameters
MyInteger{a=3}
MyInteger{a=5}
```

Unterschied Call-By-Reference in Java und C

```
public class MyInteger {  
    int a;  
    public MyInteger( int a ){ this.a = a; }  
    public String toString() {  
        return "MyInteger{a=" + a + "}";  
    }  
}  
  
public class MethodParameters {  
  
    public static void someMethod(MyInteger ma){  
        ma = new MyInteger(6);  
    }  
  
    public static void main(String[] args) {  
        MyInteger mi = new MyInteger(3);  
        System.out.println(mi);  
        someMethod(mi);  
        System.out.println(mi);  
    }  
}
```

**Referenzkopie
(Kein echtes „Call-By-Reference“ wie z.B. in C)**

**Der Parameter ma
enthält die Kopie der
Referenz des Objektes.**



```
> java MethodParameters  
MyInteger{a=3}  
MyInteger{a=3}
```

Parameterübergabe: Wrapperklasse

```
public class MethodParameters {  
    public static void someMethod(Integer a){  
        a += 2;  
        System.out.println("a: " + a); → a: 5  
    }  
  
    public static void main(String[] args) {  
        Integer y = 3;  
        System.out.println("y: " + y); → y: 3  
        someMethod(y);  
        System.out.println("y: " + y); → y: 3  
    }  
}
```

- Wrapperklassen-Objekte sind immutable (unveränderbar)
- Bei der Addition wird ein neues `Integer`-Objekt zugewiesen, das ursprüngliche `Integer`-Objekt bleibt unangetastet
- Keine Veränderung des Parameters `y` sichtbar, da die kopierte Referenz `a` beim Beenden der Methode verloren geht

Zur Verdeutlichung die ausführliche Schreibweise:

```
//...  
public static void someMethod(Integer a){  
    a = new Integer(a.intValue() + 2);  
}  
//...
```

Parameterübergabe: String

```
public class MethodParameters {
    public static void aMethod(String a){
        a += "Hello";
        System.out.println("a: " + a);
    }
    public static void main(String[] args) {
        String s = "Test";
        System.out.println("s: " + s); → s: Test
        aMethod(s);
        System.out.println("s: " + s); → s: Test
    }
}
```

↗ a: TestHello

- String-Objekte sind immutable (unveränderbar)
- Bei `a += "Hello";` wird ein neues String-Objekt angelegt. Die einzige Referenz `a` darauf geht beim Verlassen der Methode `aMethod` verloren.

Zur Verdeutlichung die ausführliche Schreibweise:

```
//...
public static void aMethod(String a){
    a = new String(a + "Hello");
}
//...
```

Parameterübergabe: StringBuffer

```
public class MethodParameters {  
    public static void anotherMethod(StringBuffer a){  
        a.append("Hello");  
        System.out.println("a: " + a);    → a: TestHello  
    }  
  
    public static void main(String[] args) {  
        StringBuffer sb = new StringBuffer("Test");  
        System.out.println("sb: " + sb); → sb: Test  
        anotherMethod(sb);  
        System.out.println("sb: " + sb); → sb: TestHello  
    }  
}
```

- StringBuffer-Objekte sind mutable (veränderbar)
- Das referenzierte StringBuffer-Objekt selbst wird in der Methode `anotherMethod` mit `append()` verändert.
Dadurch ist die Änderung auch über die Referenz `sb` sichtbar.

Parameterübergabe: Array

```
public class ArrayParameters {  
  
    int[] nums;  
  
    public void setArray1(int[] nums) {  
        this.nums = nums;  
    }  
  
    public void setArray2(int... nums) {  
        this.nums = nums;  
    }  
  
    public static void main(String[] args) {  
        ArrayParameters ap = new ArrayParameters();  
  
        int[] n = { 1, 2, 3, 4, 5, 6 };  
        ap.setArray1(n);  
        ap.setArray2(1, 2, 3, 4, 5, 6, 7);  
        ap.setArray2(n);  
    }  
}
```

 **Ab Java 1.5**

Rekursion

- Innerhalb des Anweisungsblocks einer Methode können wiederum Methodenaufrufe stehen
- Methoden können nicht nur andere Methoden aufrufen, sondern auch sich selbst
- Eine Methode `m ()` heißt rekursiv, wenn sie sich selbst aufruft

`m (...) ⇒ m (...)`

direkt rekursiv

`m (...) ⇒ n (...) ⇒ m (...)`

indirekt rekursiv

- Viele Probleme (in der Informatik) lassen sich durch Rekursion besonders einfach und elegant lösen
- Schon die Beschreibung eines Problems kann iterativ oder rekursiv geschehen

Berechnung der Fakultät

iterative Definition

$$n! = 1 * 2 * 3 * \dots * (n-1) * n$$

rekursive Definition

$$n! = \begin{cases} (n-1)! * n & \text{für } n > 1 \\ 1 & \text{für } n = 1 \text{ und } n = 0 \end{cases}$$

■ Iterative Methode zur Berechnung der Fakultät

```
long fact (int n) {  
    long result = 1;  
    for (int i = 2; i <= n; i++)  
        result *= i;  
    return result;  
}
```

■ Rekursive Methode zur Berechnung der Fakultät

```
long fact(int n) {  
    if (n == 0 || n == 1)  
        return 1;  
    else  
        return fact(n - 1) * n;  
}
```

Allgemeines Muster

```
if ( Problem klein genug )  
    nichtrekursiver Zweig  
else  
    rekursiver Zweig
```

Ablauf einer rekursiven Methode (1): Fakultät

$n==4$   24

```
long fact(int n) {
    if (n == 0 || n == 1) return 1;
    else return fact(n - 1) * n;
}
```

Jede Aktivierung von `fact` hat ihr eigenes `n` und rettet es über den rekursiven Aufruf hinweg.

$n==3$   6

```
long fact(int n) {
    if (n == 0 || n == 1) return 1;
    else return fact(n - 1) * n;
}
```

$n==2$   2

```
long fact(int n) {
    if (n == 0 || n == 1) return 1;
    else return fact(n - 1) * n;
}
```

$n==1$   1

```
long fact(int n) {
    if (n == 0 || n == 1) return 1;
    else return fact(n - 1) * n;
}
```

Ablauf einer rekursiven Methode (2): Fakultät

```
long fact(int n) {  
    if (n == 0 || n == 1) return 1;  
    else return fact(n - 1) * n;  
}
```

Beispiel: Aufruf von fact(4)

■ Rekursiver Zweig:

1. `fact(4)` ruft `fact(3)` auf, bewahrt den Wert 4 in lokaler Variable `n`
2. `fact(3)` ruft `fact(2)` auf, bewahrt den Wert 3 in lokaler Variable `n` (verschieden von `n` aus 1.)
3. `fact(2)` ruft `fact(1)` auf, bewahrt den Wert 2 in lokaler Variable `n` (wiederum neue Variable)

■ Nicht rekursiver Zweig

- `fact(1)` gibt den Wert 1 zurück
- **Auf dem Weg nach oben** wird der vor dem rekursiven Aufruf ermittelte Wert mit der jeweiligen lokalen Variablen `n` des Rufers multipliziert und das Ergebnis weiter nach oben gegeben.

Größter gemeinsamer Teiler (ggT)

■ Algorithmus:

*Bestimme ggT von zwei positiven Zahlen,
z.B. von 24 und 15:*

$$\begin{array}{rclcl}
 24 & : & 15 & = & 1 \text{ Rest } 9 \\
 15 & : & 9 & = & 1 \text{ Rest } 6 \\
 9 & : & 6 & = & 1 \text{ Rest } 3 \\
 6 & : & 3 & = & 2 \text{ Rest } 0
 \end{array}$$

↓

$$\text{ggT} = 3$$

Immer: „größere durch kleinere Zahl“

Beispiel: Größter gemeinsamer Teiler

rekursiv

```
static int gcd(int x, int y) {  
    int rest = x % y;  
    if (rest == 0) {  
        return y;  
    } else {  
        return gcd(y, rest);  
    }  
}
```

iterativ

```
static int gcd(int x, int y) {  
    int rest = x % y;  
    while (rest != 0){  
        x = y;  
        y = rest;  
        rest = x % y;  
    }  
    return y;  
}
```

Vor- und Nachteile rekursiver Algorithmen

- Anmerkung
 - zu jedem rekursiv formulierten Algorithmus gibt es einen äquivalenten iterativen Algorithmus
- Vorteile rekursiver Algorithmen
 - kürzere Formulierung
 - leichter verständliche Lösung
 - Einsparung von Variablen
 - teilweise sehr effiziente Problemlösungen (z.B. Quicksort)
 - Bei rekursiven Datenstrukturen (zum Beispiel Bäume, Graphen) besonders empfehlenswert
- Nachteile rekursiver Algorithmen
 - oft weniger effizientes Laufzeitverhalten (Overhead bei Funktionsaufruf)
 - Verständnisprobleme bei Programmieranfängern
 - Konstruktion rekursiver Algorithmen "gewöhnungsbedürftig"

Rekursion in Programmiersprachen

Nicht alle Programmiersprachen unterstützen rekursive Algorithmen

- Rekursion erlaubt:

- ALGOL-Familie (ALGOL 60, Simula 67, ALGOL 68, PASCAL, MODULA-2, Ada)
- PL/1
- C und C++
- Java
- C#

- Rekursion nicht möglich:

- FORTRAN
- COBOL

- LISP arbeitet überwiegend mit rekursiven Algorithmen
(gilt i.d.R. auch für PROLOG)