

# **Praktische Informatik 1**

## **Prozeduren, Funktionen, JavaDoc, Datenströme**

Thomas Röfer

- Unterprogramme, Prozeduren und Funktionen
- Formale Parameter
- Überladen von Funktionen
- Rekursion
- Dokumentieren von Quelltexten
- Datenströme, Dateisystem
- Musterlösungen, Übungsblatt

# Rückblick „Anweisungen, Verzweigungen und Schleifen“

- Variablendeklaration
  - `typ bezeichner [ = ausdruck ]  
{ , bezeichner [ = ausdruck ] } ;`
- Zuweisungen
  - `l-wert [ operator ] = ausdruck ;`
- Erhöhen/verringern einer Variablen
  - `( ++ | -- ) bezeichner ; | bezeichner ( ++ | -- ) ;`
- Funktionsaufruf
  - `[ ausdruck . ] bezeichner ( [ ausdruck { ,  
ausdruck } ] ) ;`
- Zusicherung und Ausnahmen
  - `assert bedingung [ : ausdruck ] ;`
  - `throw ausdruck ;`
- Ausnahmen fangen
  - `try { { anweisung } }  
{ catch ( typ bezeichner ) { { anweisung } } }  
[ finally { { anweisung } } ]`
- Verzweigungen
  - `if ( bedingung ) anweisung [ else  
anweisung ]`
  - `switch ( ausdruck )  
{ { ( case konstausdruck | default ) :  
{ anweisung } } }`
- Schleifen
  - `while ( bedingung ) anweisung`
  - `do anweisung while ( bedingung ) ;`
  - `for ( [ initialisierungen ] ; [ bedingung ] ; [  
aktualisierungen ] ) anweisung`
  - `for ( typ bezeichner : ausdruck )  
anweisung`
- Marke, Abbruch, Fortsetzung, Rückgabe
  - `bezeichner :`
  - `break [ bezeichner ] ;`
  - `continue [ bezeichner ] ;`
  - `return [ ausdruck ] ;`
- Block
  - `{ { anweisung } }`

# Unterprogramme

- Wiederverwendung von Berechnungen
- Strukturierung von Programmen
- Funktionen
  - Unterprogramme mit Rückgabewert (Ergebnis)
  - Können Seiteneffekte haben, z.B. Attribute eines Objekts ändern
- Prozeduren
  - Unterprogramme ohne Rückgabewert
  - Java: Funktionen mit Rückgabebetyp *void*
  - Müssen Seiteneffekte haben, z.B. Attribute eines Objekts ändern oder etwas auf der Konsole ausgeben

```
int mod(int a, int b) {  
    if (a < b) {  
        return a;  
    } else {  
        return mod(a - b, b);  
    }  
}
```

```
void printMod16(int x) {  
    System.out.println(  
        mod(x, 16));  
}
```

# Teile eines Unterprogramms

- Kopf (Signatur / Aufrufschnittstelle)

- Name des Unterprogramms
- Namen und Typen der *formalen Parameter*
- Ergebnistyp

- Rumpf

- Anweisungsblock mit Berechnungsvorschriften
- Kann *lokale Variablen* vereinbaren

- Aufruf

- Belegung der *formalen Parameter* mit passenden *aktuellen Parametern* und Ausführung des Anweisungsblocks

```
int mod(int a, int b) {  
    if (a < b) {  
        return a;  
    } else {  
        return mod(a - b, b);  
    }  
}
```

```
void printMod16(int x) {  
    System.out.println(  
        mod(x, 16));  
}
```



# Formale Parameter

- Eingabeparameter
  - Nur Übergabe von Werten an Funktion
  - In Java: Werden Objekte übergeben, kann nicht angegeben werden, dass eine Funktion diese nicht ändern wird
- Ausgabeparameter
  - Funktion liefert Ergebnis (teilweise) in Parametern zurück
  - Seiteneffekte (?)
  - In Java: Nur durch Übergabe von Referenzen auf Objekte zu realisieren, wobei die Funktion den Inhalt der Objekte ändert
- Transiente Parameter
  - Parameter werden sowohl zur Eingabe, als auch zur Ausgabe genutzt

```
int mod(int a, int b) {  
    return a - a / b * b;  
}
```

```
void divMod(int a, int b,  
            Value d, Value r) {  
    d.value = a / b;  
    r.value = a - d.value * b;  
}
```

```
void swap(Value a, Value b) {  
    int t = a.value;  
    a.value = b.value;  
    b.value = t;  
}
```

# Überladen von Funktionen

- Eine Funktion ist überladen, wenn mehrere Funktionen mit gleichem Namen im selben Sichtbarkeitsbereich definiert werden, die sich in ihrer Signatur unterscheiden
- In Java müssen sich überladene Funktionen durch die Typen ihrer formalen Parameter unterscheiden, ein ausschließlich unterschiedlicher Rückgabetyt reicht nicht

*plus (plus (1, 2), plus (3, 4))*

- Beispiele

- $\text{plus} : \text{int} \times \text{int} \rightarrow \text{int}$ 
  - `static int plus(int a, int b) {return a + b;}`
- $\text{plus} : \text{float} \times \text{float} \rightarrow \text{float}$ 
  - `static float plus(float a, float b) {return a + b;}`
- $\text{plus} : \text{int} \times \text{int} \rightarrow \text{float}$ 
  - Nicht zulässig, da Parameterliste identisch mit  $\text{plus} : \text{int} \times \text{int} \rightarrow \text{int}$

# Überladene Funktionen – Aufruf

- Es wird immer die speziellste Überladung aufgerufen, d.h. diejenige, die die wenigsten automatischen Typenerweiterungen erfordert
- Ist dies nicht eindeutig, erzeugt der Übersetzer einen Fehler
- Beispiel 1
  - $\text{plus} : \text{int} \times \text{long} \rightarrow \text{int}$
  - $\text{plus} : \text{long} \times \text{long} \rightarrow \text{long}$
  - Aufruf  $\text{plus}(1, 2)$  wählt Überladung 1,  $\text{plus}(1L, 2)$  wählt Überladung 2
- Beispiel 2
  - $\text{plus} : \text{long} \times \text{int} \rightarrow \text{long}$  (Zusätzlich zu Beispiel 1)
  - Aufruf  $\text{plus}(1, 2)$  ist mehrdeutig, da die Überladungen 1 und 3 gleich viele Typenerweiterungen erfordern

# Überladene Funktionen – Beispiel

```
class WithType {  
    static void println(boolean value) {  
        System.out.println("boolean: " + value);  
    }  
    static void println(int value) {  
        System.out.println("int: " + value);  
    }  
    static void println(long value) {  
        System.out.println("long: " + value);  
    }  
    static void println(double value) {  
        System.out.println("double: " + value);  
    }  
    static void println(String value) {  
        System.out.println("String: " + value);  
    }  
}
```

```
WithType.println(true);  
WithType.println(1);  
WithType.println(1L);  
WithType.println(1.0);  
WithType.println("1");  
WithType.println(1.0F);  
WithType.println((byte) 1);
```

```
boolean: true  
int: 1  
long: 1  
double: 1.0  
String: 1  
double: 1.0  
int: 1
```

# Methoden mit variabler Parameterzahl

- Teilweise erleichtert es die Notation, wenn eine Methode eine beliebige Anzahl von Parametern akzeptiert
- Dies ist übersichtlicher, als die Parameter in ein Array zu verpacken und dieses zu übergeben.
- Ansatz
  - Erlaube beliebige Parameteranzahl mit und verpacke die Werte automatisch in ein Array
  - Ein solcher Parameter kann nur einmal vorkommen und muss der letzte sein
- Realisierung
  - *sum(int... values)* wird umgesetzt als *sum(int[] values)*
  - *sum(1, 2, 3)* wird umgesetzt als *sum(new int[]{1, 2, 3})*

```
static int sum(int... values) {  
    int s = 0;  
    for (int v : values) {  
        s += v;  
    }  
    return s;  
}
```

```
int s = sum(1, 5, 6);
```



# Überladene Funktionen – Beispiel

```
static void println(Object... objects) {  
    for (Object o : objects) {  
        if (o instanceof Boolean) {  
            println((Boolean) o);  
        } else if (o instanceof Integer) {  
            println((Integer) o);  
        } else if (o instanceof Long) {  
            println((Long) o);  
        } else if (o instanceof Double) {  
            println((Double) o);  
        } else if (o instanceof String) {  
            println((String) o);  
        } else {  
            System.out.println("? " + o.toString());  
        }  
    }  
}
```

```
WithType.println(true, 1, 1L,  
                 1.0, "1", 1.0F, (byte) 1);
```

```
boolean: true  
int: 1  
long: 1  
double: 1.0  
String: 1  
? 1.0  
? 1
```



# Aufruf von Objekt-Methoden

- Beim Aufruf einer Objekt-Methode wird ein versteckter Parameter mit übergeben, die *this*-Referenz
- Java nutzt diese Referenz, um auf Objekt-Attribute und -Methoden zuzugreifen
- Sie kann aber auch direkt verwendet werden, falls die Referenz benötigt wird oder um Mehrdeutigkeiten aufzulösen
- Signaturen
  - `copyFrom` : **Name**  $\times$  *Name*  $\rightarrow$
  - `copyTo` : **Name**  $\times$  *Name*  $\rightarrow$

```
class Name {  
    String name;  
  
    void copyFrom(Name name) {  
        this.name = name.name;  
    }  
  
    void copyTo(Name other) {  
        other.copyFrom(this);  
    }  
}
```

```
Name a = new Name();  
Name b = new Name();  
a.name = "Markus";  
a.copyTo(b);
```

# Rekursion

- Rekursiver Aufruf
  - Eine Funktion ruft sich selbst auf
- Verschränkt rekursiver Aufruf
  - Funktionen rufen sich gegenseitig immer wieder auf
- Endrekursion
  - Ist der rekursive Aufruf in einer Funktion die letzte Anweisung, ist sie *endrekursiv*
  - Eine *Endrekursion* kann immer durch eine Iteration ersetzt werden
- Komplexität
  - Mehrfach (nicht primitiv) rekursive Funktionen können schnell wachsen
  - Beispiel: Fibonacci-Zahlen
  - Beispiel: Ackermann-Funktion (s. Buch)

$$A(a, b) = \begin{cases} b + 1 & \text{falls } a = 0 \\ A(a - 1, 1) & \text{falls } b = 0 \\ A(a - 1, A(a, b - 1)) & \text{sonst} \end{cases}$$

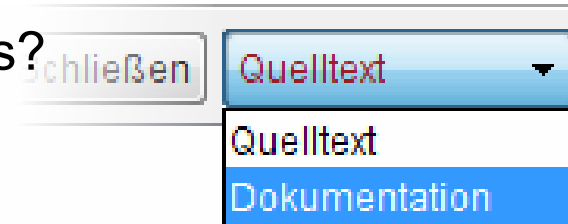
```
int mod(int a, int b) {  
    if (a < b)  
        return a;  
    } else {  
        return mod2(a, b);  
    }  
}  
  
int mod2(int a, int b) {  
    return mod(a - b, b);  
}
```

# Dokumentieren von Quelltexten

- **Klassen**
  - Allgemeine Beschreibung, was die Aufgabe der Klasse ist
- **Attribute**
  - Was wird in dem Attribut gespeichert?
  - Haben bestimmte Belegungen besondere Bedeutungen?
- **Methoden**
  - Was tut die Methode?
  - Welche Parameter hat sie und was bedeuten sie?
  - Welches Ergebnis liefert die Methode?
  - Welche Seiteneffekte hat die Methode?
  - Was sind die Vorbedingungen für die Anwendbarkeit der Methode?
  - Welche Nachbedingungen sind erfüllt?

# Dokumentieren von Quelltexten

- In externen Dokumenten
  - Wie ist die Gesamtarchitektur des Programms?
  - Wie arbeiten die Klassen zusammen?
  - Wie benutzt man das Programm?
- Kommentare in Java
  - Zeilen-Kommentar:           // Ein Kommentar bis zum Zeilenende
  - Block-Kommentar:           /\* Ein Kommentar bis zur  
Endmarkierung \*/
  - JavaDoc-Kommentar:       /\*\* Ein JavaDoc-Kommentar \*/
- JavaDoc
  - Ein Übersetzer, der aus einem Java-Quelltext eine Dokumentation erstellt
  - Die Dokumentation wird im HTML-Format (Hyper Text Markup Language) erzeugt
  - In JavaDoc-Kommentaren kann deshalb auch HTML verwendet werden
  - Aufruf in BlueJ aus dem Texteditor



## Beispiel

```
/**
 * Die Klasse berechnet die Hasenpopulation für einen bestimmten Monat
 * mit ganz unterschiedlichen Methoden, teils rekursiv, teils iterativ.
 *
 * @author <a href="mailto:Thomas.Roefer@dfki.de">Thomas Röfer</a>
 * @version 1.0
 * @see "Übungsblatt 5, Aufgabe 2"
 */
public class Rabbits {
    /**
     * Die Methode berechnet die Anzahl der Hasenpaare für einen bestimmten
     * Monat iterativ. Sie stellt quasi die klassische iterative Lösung dar.
     * @param month Der Monat, für den die Hasenpopulation bestimmt werden
     *             soll. Muss mindestens 1 sein.
     * @return Die Anzahl der Hasenpaare im angegebenen Monat.
     */
    static int calcRabbitsI0(int month) {
```

# JavaDoc-Schlüsselwörter (allgemein)

- **@author** *autorenname*
  - Angabe des Autors, z.B.:
  - `@author Thomas Röfer`
  - `@author <a href="mailto:Thomas.Roefer@dfki.de">Thomas Röfer</a>`
- **@version** *versionsnummer*
  - Angabe der Versionsnummer, z.B.:
  - `@version 3.14beta`
- **@deprecated** *hinweistext*
  - Die/das Klasse/Methode/Attribut soll nicht mehr verwendet werden, z.B.:
  - `@deprecated Verwenden Sie stattdessen bitte betterMethod().`
- **@since** *versionsnummer*
  - Die Klasse/Methode funktioniert erst seit einer bestimmten Version, z.B.:
  - `@since 3.13`



# JavaDoc-Schlüsselwörter (Methoden)

- **@param** *parametername beschreibung*
  - Beschreibung eines Funktionsparameters, z.B.:
  - *@param month Der Monat, für den die Hasenpopulation errechnet werden soll.*
- **@return** *beschreibung*
  - Beschreibung des Rückgabewerts, z.B.:
  - *@return Die Anzahl der Hasenpaare im angegebenen Monat.*
- **@throws** *ausnahmenname beschreibung*
  - Beschreibung von möglicherweise von einer Funktion erzeugten Ausnahmen, z.B.:
  - *@throws FalscherWertAusnahme Diese Ausnahme wird erzeugt, wenn ein falscher Monat übergeben wurde (kleiner 1 oder zu groß).*

# JavaDoc-Schlüsselwörter (Querverweise)

- **{@link [ [ *paket* . ] *klasse* # ] [ *methode* | *attribut* ] *text* }**
  - Fügt einen Querverweis zu einer Methode oder einem Attribut hinzu. Der Querverweis wird als Quelltext formatiert. Z.B.:
  - @deprecated Verwenden Sie stattdessen bitte {@link #betterMethod() betterMethod()}.
- **{@linkplain [ [ *paket* . ] *klasse* # ] [ *methode* | *attribut* ] *text* }**
  - Wie @link, aber der Querverweis wird als normaler Text formatiert.
- **@see [ [ *paket* . ] *klasse* # ] [ *methode* | *attribut* ] *text***
  - Fügt einen Verweis in einem separaten Abschnitt hinzu. Der Text muss in Anführungszeichen stehen, falls er aus mehreren Worten besteht, oder durch einen HTML-Verweis geklammert sein. Z.B.:
  - @see #betterMethod() betterMethod
  - @see "Übungsblatt 9"
  - @see <a href="http://www.informatik.uni-bremen.de">FB3-Homepage</a>

# Datenströme

- Datenströme leiten Daten seriell in ein Programm hinein bzw. wieder heraus
- Sie können umgeleitet werden. So kann die Eingabe aus einer Datei kommen bzw. die Ausgabe in eine Datei geschrieben werden
- Standardeingabe
  - System.in (Instanz von InputStream)
- Standardausgabe
  - System.out (Instanz von PrintStream)
- Standardfehlerausgabe
  - System.err (Instanz von

