# Prozeduren, Funktionen, Datenströme, JavaDoc

Thomas Röfer

Unterprogramme, Prozeduren und Funktionen
Formale Parameter
Überladen von Funktionen
Rekursion
Dokumentieren von Quelltexten
Datenströme

# 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*
- **Marke, Abbruch, Fortsetzung, Rückgabe**
  - *bezeichner* **:**
  - **break** [ *bezeichner* ] **;**
  - **continue** [ *bezeichner* ] **;**
  - **return** [ *ausdruck* ] **;**
- **Block**
  - **{** { *anweisung* } **}**

# Unterprogramme

- **Ziele**
  - Wiederverwendung von Berechnungen
  - Strukturierung von Programmen
- **Funktionen**
  - Unterprogramme mit Rückgabewert (Ergebnis)
  - Können Seiteneffekte haben, z.B. Attribute eines Objekts ändern
- **Prozeduren**
  - Unterprogramme ohne Rückgabewert
  - In Java: Funktionen mit Rückgabetyp *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

- **Definition**
  - 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 Rückgabetyp reicht nicht
- **Beispiele**
  - plus : int × int → int
    - *static int plus(int a, int b) {return a + b;}*
  - plus : float × float → float
    - *static float plus(float a, float b) {return a + b;}*
  - plus : int × int → float
    - *Nicht zulässig, da Parameterliste identisch mit plus : int × int → int*

# Überladene Funktionen – Aufruf

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

# Aufruf von Objekt-Methoden

- **Definition**
  - 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 × Name →
  - copyTo : Name × Name →

```
class Name
{
  String name;

  void copyFrom(Name name)
  {
    this.name = name.name;
  }

  void copyTo(Name other)
  {
    other.copyFrom(this);
  }
}
```

```
Name a, b;
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: Fibunacci-Zahlen
  - Beispiel: Ackermann-Funktion (s. Buch)

```
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);
}
```

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

# Java ohne BlueJ

- **Deklaration**
  - Enthält eine Klasse eine Funktion mit der Signatur

    public static void main(String[ ] args)

    so kann diese von der Konsole aus aufgerufen werden
- **Aufruf**
  - Virtuelle Maschine *java*
  - Name der Klasse, die *main* enthält
  - Eventuelle Parameter für *main*
  - Die Parameter werden als String-Array übergeben

```
>java Klasse Hallo 21333 "dies und das"
```

String[] args

| args.length | 3 |
|---|---|
| args[0] | "Hallo" |
| args[1] | "21333" |
| args[2] | "dies und das" |

```
public static void main(String[] args) ...
```

# Beispiel

```
class Greetings
{
  public static void main(String[ ] args)
  {
    for(int i = 0; i < args.length; ++i)
    {
      if(args[i].equals("Walter"))
        System.out.println("Mein Gott " + args[i]);
      else
        System.out.println("Hallo " + args[i]);
    }
  }
}
```

# 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: // Dies ist ein Kommentar bis zum Zeilenende
  - Block-Kommentar: /* Dies ist ein Kommentar bis zur Endmarkierung */
  - JavaDoc-Kommentar: /** Dies ist 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

# JavaDoc-Schlüsselwörter (allgemein)

- **@author *autorenname***
  - Angabe des Autors, z.B.:
  - *@author Thomas Röfer*
  - *@author <a href="mailto:roefer@tzi.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)

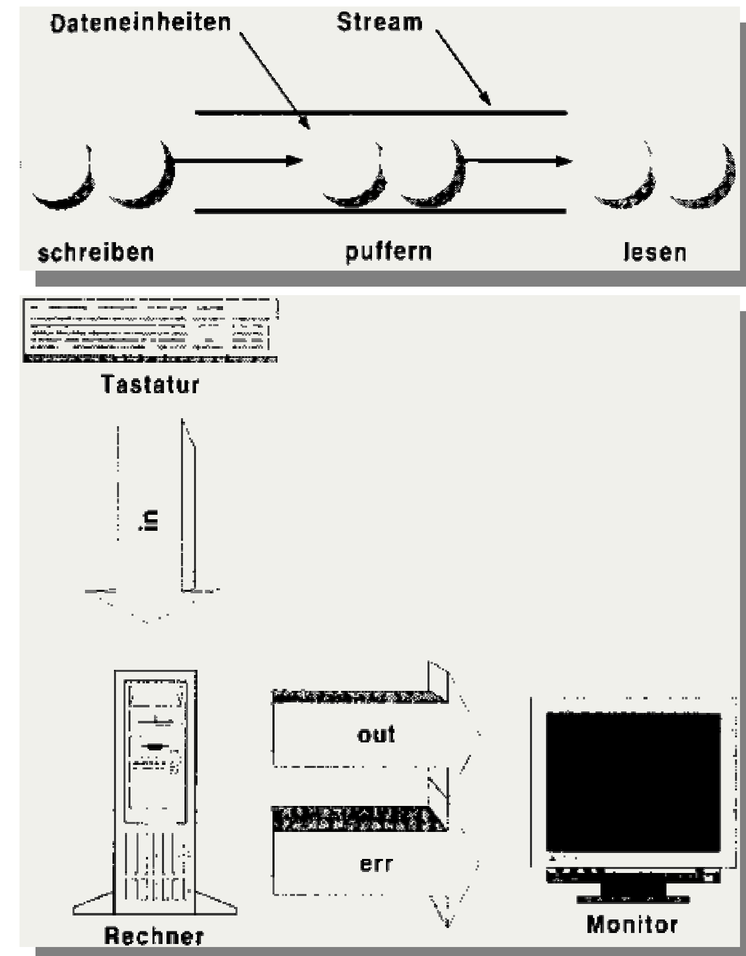
- **{@link [ [ *paket* . ] *klasse* # ] [ *methode* | *attribut* ] *text* }**
  - Fügt einen Querverweis zu einer Methode oder einem Attribut hinzu. Der Querverweis wird als Quelltext formatiert. Z.B.:
  - *@depricated Verwenden Sie stattdessen bitte {@link #betterMethod() betterMethod()}.*
- **{@linkplain [ [ *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 "Übungsblatt 9"*
  - *@see #betterMethod() betterMethod*
  - *@see <a href="http://www.tzi.de/pi1">PI-1</a>*

# Beispiel

```
/**
 * Die Klasse berechnet die Hasenpopulation für einen bestimmten Monat
 * mit ganz unterschiedlichen Methoden, teils rekursiv, teils iterativ.
 *
 * @author <a href="mailto:roefer@tzi.de">Thomas Röfer</a>
 * @version 1.0
 * @see <a href="http://www.tzi.de/pi1/pi1-blatt4.pdf">Übungsblatt 4</a>
 */
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 angegeben Monat.
   */
  static int calcRabbitsI0(int month)
  {
```

# Datenströme

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



# Datenströme in Java

- **Byte-orientierte Datenstöme**
  - InputStream
    - *FileInputStream, FilterInputStream, PushbackInputStream ...*
  - OutputStream
    - *FileOutputStream, FilterOutputStream, PrintStream ...*
- **Unicode-orientierte Datenströme**
  - Reader
    - *BufferedReader, FilterReader, InputStreamReader, StringReader ...*
  - Writer
    - *BufferedWriter, FilterWriter, OutputStreamWriter, PrintWriter, StringWriter ...*
- **Anmerkung**
  - Datenströme befinden sich im Paket *java.io* (z.B. java.io.FileInputStream)
  - *java.io.* vor den Klassen kann eingespart werden, wenn der Quelltext eine Importanweisung am Anfang enthält: *import java.io.**

# Beispiel 1

```
import java.io.*;

class Count
{
  static int count(String fileName) throws FileNotFoundException, IOException
  {
    FileInputStream stream = new FileInputStream(fileName);
    int chars = 0;
    int c = stream.read();
    while(c != -1)
    {
      ++chars;
      c = stream.read();
    }
    return chars;
  }
}
```

# Beispiel 2 (Auszug)

```
static void more(String fileName) throws FileNotFoundException, IOException
{
  FileInputStream stream =    new FileInputStream(fileName);
  InputStreamReader reader =  new InputStreamReader(stream);
  BufferedReader buffer =     new BufferedReader(reader);
  InputStreamReader console = new InputStreamReader(System.in);
  System.out.print("\f");                           // Konsole löschen
  String line = buffer.readLine();                  // erste Zeile lesen
  int count = 1;
  while(line != null)                               // solange noch Zeilen da
  {
    System.out.println(count + ":\t" + line);       // ausgeben
    if(++count % 20 == 1)                           // alle 20 Zeilen
    {
      console.read();                               // auf Eingabe warten
      System.out.print("\f");                       // und Konsole löschen
    }
    line = buffer.readLine();                       // nächste Zeile lesen
  }
}
```

# Beispiel 2 (Auszug)

```
static void more(String fileName) throws FileNotFoundException, IOException
{
  FileInputStream stream =    new FileInputStream(fileName);
  InputStreamReader reader =  new InputStreamReader(stream);
  BufferedReader buffer =     new BufferedReader(reader);
  InputStreamReader console = new InputStreamReader(System.in);
  System.out.print("\f");
  String line = buffer.readLine();
  int count = 1;
  while(line != null)
  {
    System.out.println(count + ":\t" + line);
    if(++count % 20 == 1)
    {
      console.read();
      System.out.print("\f");
    }
    line = buffer.readLine();
  }
}
```