

Tutorium Programmieren

Tut Nr.4: Kontrollstrukturen

Michael Friedrich | 19. / 21.11.2013

INSTITUT FÜR THEORETISCHE INFORMATIK



- 1 Wiederholung
 - Signatur von Methoden
 - Überladen von Methoden
 - lokale, statische und Instanz-Variablen
- 2 Sichtbarkeit
 - Zugriffsfunktionen
- 3 Kontrollstrukturen
 - if-Verzweigung
 - Mehrfachverzweigung
 - ternärer Operator
- 4 Aufgaben
 - Coding-Aufgaben
- 5 Tutoriumsaufgabe
 - while-Schleifen

Was ist eine **Signatur**?

Was ist eine **Signatur**?

Definition Signatur

- formale Schnittstelle einer Funktion oder Prozedur
- besteht aus ...
 - **Name** der Funktion
 - **Anzahl** und **Reihenfolge der Parameterdatentypen**
 - Typ des **Rückgabewerts**

Beispiel:

```
int getValue(void) { }  
void doSomething() { }  
int add(int x, int y) { }
```

Überladen von Methoden

Wer weiß, was **Überladen** ist?

Überladen von Methoden

Wer weiß, was **Überladen** ist?

Überladen

Besitzen zwei Methoden den **gleichen Bezeichner**, aber **unterschiedliche Signaturen**, bezeichnet man das als überladen!
Es können **Anzahl und Typen** der Parameter abweichen.
Achtung: Es muss mehr, als nur der Rückgabewert abweichen!

Überladen von Methoden

Wer weiß, was **Überladen** ist?

Überladen

Besitzen zwei Methoden den **gleichen Bezeichner**, aber **unterschiedliche Signaturen**, bezeichnet man das als überladen!

Es können **Anzahl und Typen** der Parameter abweichen.

Achtung: Es muss mehr, als nur der Rückgabewert abweichen!

Beispiel:

```
public class Ueberladen {  
    public int max(int x, int y) {  
        return (x > y) ? x : y;  
    }  
    public float max(float x, float y) {  
        return (x > y) ? x : y;  
    }  
}
```

Überladen von Methoden

Wer weiß, was **Überladen** ist?

Überladen

Besitzen zwei Methoden den **gleichen Bezeichner**, aber **unterschiedliche Signaturen**, bezeichnet man das als überladen!

Es können **Anzahl und Typen** der Parameter abweichen.

Achtung: Es muss mehr, als nur der Rückgabewert abweichen!

Beispiel:

```
public class Ueberladen {  
    public int max(int x, int y) {  
        return (x > y) ? x : y;  
    }  
    public float max(float x, float y) {  
        return (x > y) ? x : y;  
    }  
}
```


statische Typisierung

- der Datentyp jeder Variablen und Methode ist während der Compilezeit festgelegt
- durch explizite Deklaration oder Typisierung
- Gegenteil: dynamische Typisierung (PHP oder Ruby)

statische Typisierung

- der Datentyp jeder Variablen und Methode ist während der Compilezeit festgelegt
- durch explizite Deklaration oder Typisierung
- Gegenteil: dynamische Typisierung (PHP oder Ruby)

Vorteile:

- + Erkennung von Fehlern während der Übersetzungszeit, vermeidet potentielle Laufzeitfehler
- + Kein Rechenaufwand für Typüberprüfungen
- + Optimierungen besser möglich

Überladen von Methoden

Wo wird das häufig verwendet?

Überladen von Methoden

Wo wird das häufig verwendet?⇒ Verwendung mehrerer Konstruktoren

Beispiel:

```
public class Person {  
    String name;  
    String vorname;  
  
    Person() {  
        /* default Konstruktor */  
    }  
  
    Person(String name) {  
        this.name = name;  
    }  
  
    Person(String name, String vorname) {  
        this.name = name;  
        this.vorname = vorname;  
    }  
}
```

lokale Variable

lokale Variablen sind innerhalb eines Blocks oder einer Methode definiert und sind nur dort gültig.

Beispiel:

```
public class Test {  
    public void doSomething() {  
        int count = 5;    // lokale Variable count ist nur innerhalb  
                           // der Methode doSomething() gültig  
  
        ...  
    }  
  
    public static void main(String[] args) {  
        System.out.println(count);    // Was passiert hier?  
    }  
}
```

Instanzvariablen

Instanzvariablen werden innerhalb einer Klassendefinition definiert und werden zusammen mit dem Objekt angelegt.

Beispiel:

```
public class User {  
    public String username;  
    public String password;  
  
    User(String username) {  
        this.username = username; // Zugriff auf Instanzvariable  
        /* warum braucht man das "this"? */  
    }  
  
    public static void main(String[] args) {  
        User user = new User("Hans");  
        System.out.println(user.name);  
        // Zugriff auf Instanzvariable des Objekts  
    }  
}
```

Klassenvariablen

Was ist eine **Klassenvariable**?

Was ist eine **Klassenvariable**?

Klassenvariablen

Klassenvariablen werden innerhalb einer Klassendefinition definiert, aber **unabhängig von einem konkreten Objekt**!

Was ist eine **Klassenvariable**?

Klassenvariablen

Klassenvariablen werden innerhalb einer Klassendefinition definiert, aber **unabhängig von einem konkreten Objekt!**

Achtung: Objektorientiert Programmieren, nicht Klassenorientiert!

Beispiel:

```
public class Math {  
    public static final double Pi = 3.14159265359  
    public static final double E = 2.7182818284590452  
    ...  
}  
  
public class Calculator {  
    public double circleArea(double radius) {  
        // Zugriff auf Klassenvariable PI der Klasse Math  
        return radius * radius * Math.PI;  
    }  
}
```

Das ganze geht auch bei Funktionen.

http:

[//docs.oracle.com/javase/7/docs/api/java/lang/Math.html](http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html)

Das ganze geht auch bei Funktionen.

http:

[//docs.oracle.com/javase/7/docs/api/java/lang/Math.html](http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html)

Warum statische Funktionen?

- unabhängig von konkreten Objekten
- kein Zugriff auf Attribute des Objektes
- \Rightarrow Verwendung von `this` nicht möglich!

Wrap up: Variablen und Attribute

	Lokale Variable	Attribut
Deklaration	innerhalb von Methoden	außerhalb von Methoden
Lebensdauer	Methoden-Aufruf	Lebensdauer des zugehörigen Objekts
Zugänglichkeit	nur innerhalb einer Methode	für alle Methoden der Klasse
Zweck	Zwischenspeicher für Werte	Zustand des Objekts

mit speziellen Schlüsselworten wird die Sichtbarkeit einer bestimmten Komponente (Methode, Attribut, Klasse) festgelegt.

- **kein Schlüsselwort**

Komponente ist innerhalb des Pakets bekannt

mit speziellen Schlüsselworten wird die Sichtbarkeit einer bestimmten Komponente (Methode, Attribut, Klasse) festgelegt.

- **kein Schlüsselwort**

Komponente ist innerhalb des Pakets bekannt

- **private**

Komponente ist innerhalb der Klasse bekannt

mit speziellen Schlüsselworten wird die Sichtbarkeit einer bestimmten Komponente (Methode, Attribut, Klasse) festgelegt.

- **kein Schlüsselwort**

Komponente ist innerhalb des Pakets bekannt

- **private**

Komponente ist innerhalb der Klasse bekannt

- **public**

Komponente ist überall bekannt

mit speziellen Schlüsselworten wird die Sichtbarkeit einer bestimmten Komponente (Methode, Attribut, Klasse) festgelegt.

- **kein Schlüsselwort**

Komponente ist innerhalb des Pakets bekannt

- **private**

Komponente ist innerhalb der Klasse bekannt

- **public**

Komponente ist überall bekannt

- **protected**

Komponente ist innerhalb des Pakets und in allen Unterklassen bekannt

Beispiel von vorhin mit **private**-Attributen

```
public class User {  
    private String username;  
    private String password;  
  
    User(String username) {  
        this.username = username; // Zugriff auf Instanzvariable  
    }  
  
    public static void main(String[] args) {  
        User user = new User("Hans");  
        System.out.println(user.name);  
        // Was passiert?  
    }  
}
```

Beispiel von vorhin mit **private**-Attributen

```
public class User {  
    private String username;  
    private String password;  
  
    User(String username) {  
        this.username = username; // Zugriff auf Instanzvariable  
    }  
  
    public static void main(String[] args) {  
        User user = new User("Hans");  
        System.out.println(user.name);  
        // Was passiert?  
    }  
}
```

Lösung?

getter und setter

- spezielle Methode zum Zugriff auf Eigenschaft eines Objekts
- **getter**-Methode gibt den Wert eines Attributs zurück
- **setter**-Methode ändert den Wert eines Attributs

```
public class User {  
    private String username;  
    private String password;  
  
    public setPassword(String password) { // Setter  
        this.password = password;  
    }  
  
    public getPassword() { // Getter  
        return this.password  
    }  
}
```

Gründe für die Verwendung von Gettern und Settern?

- Einhaltung des Prinzips der Datenkapselung (Geheimnisprinzip)

Gründe für die Verwendung von Gettern und Settern?

- Einhaltung des Prinzips der Datenkapselung (Geheimnisprinzip)
- Validierung der zu setzenden Werte

Gründe für die Verwendung von Gettern und Settern?

- Einhaltung des Prinzips der Datenkapselung (Geheimnisprinzip)
- Validierung der zu setzenden Werte
- Nebenbedingungen bei get oder set

Gründe für die Verwendung von Gettern und Settern?

- Einhaltung des Prinzips der Datenkapselung (Geheimnisprinzip)
- Validierung der zu setzenden Werte
- Nebenbedingungen bei get oder set

```
public class User {  
    private String password;  
  
    public setPassword(String password) { // Setter  
        if ( password.length >= 6 ) {  
            this.password = password;  
        }  
    }  
  
    ...  
}
```

if-Verzweigung

- Wertet einen Ausdruck aus und verzweigt, je nach Ergebnis
- der Ausdruck muss **true** oder **false** zurückliefern (**boolean**)
- Ausdruck **true** \Rightarrow nachfolgender Block wird ausgeführt
- optional: **else**-Bedingung (bei **false** ausgeführt)

Syntax einer if-else-Verzweigung

```
if (<Bedingung>) {  
    <ausgefuehrt bei true>  
} else {  
    <ausgefuehrt bei false>  
}
```


Verzweigung (if)

Geht auch ohne geschweifte Klammern.

```
boolean exit = false;  
if (exit)  
    System.out.println("Shutting down");  
System.exit(0);
```

Wo ist das Problem?

Geht auch ohne geschweifte Klammern.

```
boolean exit = false;  
if (exit)  
    System.out.println("Shutting down");  
    System.exit(0);
```

Wo ist das Problem?

Lösung: Zeile 4 wird immer ausgeführt!

Deshalb immer:

- { und } setzen
- Blockinhalt einrücken



```
if (5 == count)
```

- Using `if` (constant = variable) instead of `if` (variable = constant)
- Its like saying “if blue is the sky” or “if tall is the man”

Quelle: codinghorror.com

Auch möglich:

```
int x = 3;
if (x == 1) {
    System.out.println("x ist eins");
} else if (x == 2) {
    System.out.println("x ist zwei");
} else if (x == 3) {
    System.out.println("x ist drei");
}
```

Mehrfachverzweigung: switch

- Macht `if – else` Verzweigungen mit mehreren Möglichkeiten übersichtlicher
- Switch kann mit `int`, `short`, `byte`, `char`, `enum` und in Java 7 auch mit `String` ausgeführt werden
- Verzweigung anhand `case`-Blöcke innerhalb des `switch`-Blocks
- `default`-Block wird ausgeführt, wenn keine Übereinstimmung
- `break`; als Abbruch nach jedem `case`

```
int month = 11;
switch (month) {
    case 1: System.out.println("Januar");
        break;
    case 2: System.out.println("Februar");
        break;

    ...

    case 11: System.out.println("November");
        break;
    case 12: System.out.println("Dezember");
        break;
    default: System.out.println("Monat existiert nicht!");
}
```

```
int count = 1;
switch (count) {
    case 1: System.out.println("one");
    case 2: System.out.println("two");
    case 3: System.out.println("three");
    default: System.out.println("Counting is fun");
}
```

Welche ausgabe wird erzeugt?

switch **Beispiel 2**

```
int count = 1;  
switch (count) {  
    case 1: System.out.println("one");  
    case 2: System.out.println("two");  
    case 3: System.out.println("three");  
    default: System.out.println("Counting is fun");  
}
```

Welche ausgabe wird erzeugt?

one

two

three

Counting is fun

- Besonders kompakte Darstellung einer `if-else`-Verzweigung in einer Zeile
- Kann zu unleserlichem code führen!

Syntax:

```
(<boolscher Ausdruck>) ? <true-Block> : <false-Block>;
```

```
if (gender.equals("männlich")) {  
    System.out.println ("Sehr geehrter Herr");  
} else {  
    System.out.println ("Sehr geehrte Frau");  
}
```

Wie sieht die äquivalente Darstellung mithilfe des ternären Operators aus?

```
if (gender.equals("männlich")) {  
    System.out.println ("Sehr geehrter Herr");  
} else {  
    System.out.println ("Sehr geehrte Frau");  
}
```

Wie sieht die äquivalente Darstellung mithilfe des ternären Operators aus?

Lösung:

```
System.out.println( "Sehr geehrte"  
    +(gender.equals("männlich") ? "r Herr" : " Frau" ));
```

- **weekday** ist true, falls es ein Wochentag ist
- **vacation** ist true, falls wir in Urlaub sind

Vervollständige die Funktion:

```
public boolean sleepIn(boolean weekday, boolean vacation) {  
    // TODO  
}
```

- **weekday** ist true, falls es ein Wochentag ist
- **vacation** ist true, falls wir in Urlaub sind

Vervollständige die Funktion:

```
public boolean sleepIn(boolean weekday, boolean vacation) {  
    // TODO  
}
```

Lösung:

```
public boolean sleepIn(boolean weekday, boolean vacation) {  
    return !weekday || vacation;  
}
```

Aufgabe

We have two monkeys, a and b, and the parameters **aSmile** and **bSmile** indicate if each is smiling. We are in trouble if they are both smiling or if neither of them is smiling. Return true if we are in trouble.

```
public boolean monkeyTrouble(boolean aSmile, boolean bSmile) {  
    // TODO  
}
```

Aufgabe

We have two monkeys, a and b, and the parameters **aSmile** and **bSmile** indicate if each is smiling. We are in trouble if they are both smiling or if neither of them is smiling. Return true if we are in trouble.

```
public boolean monkeyTrouble(boolean aSmile, boolean bSmile) {  
    // TODO  
}
```

Lösung:

```
public boolean monkeyTrouble(boolean aSmile, boolean bSmile) {  
    return (aSmile && bSmile) || (!aSmile && !bSmile);  
}
```

substring-Methode

- Methode der Klasse **String**
- Signatur: `String substring(int beginIndex, int endIndex)`
- Gibt den Teilstring zwischen `beginIndex` und `endIndex` zurück

substring-Methode

- Methode der Klasse **String**
- Signatur: `String substring(int beginIndex, int endIndex)`
- Gibt den Teilstring zwischen `beginIndex` und `endIndex` zurück

Vervollständige die Funktion:

```
public String notString(String str) {  
  
}
```

Erwartetes Verhalten:

`notString(„candy“)` → „not candy“

`notString(„x“)` → „not x“

`notString(„not bad“)` → „not bad“

Lösung:

```
public String notString(String str) {  
    if (str.length() >= 3 && str.substring(0, 3).equals("not")) {  
        return str;  
    }  
  
    return "not " + str;  
}
```

while - Schleifen Aufgabe 1 (Schleifen) Schreiben Sie in einer Klasse namens `Loops` die Methoden

- `static boolean isPrimeWhile(int candidate)` und
- `static boolean isPrimeDoWhile(int candidate)`.

Diese sollen als Rückgabewert den Wert `true` haben, falls `candidate` eine Primzahl ist. In den Methoden soll dabei eine `while`-Schleife (a)) bzw. eine `do-while`-Schleife (b)) verwendet werden. Beachten Sie auch die Fälle, in denen der Wert des Parameters ganz offensichtlich keine Primzahl ist.

- Schreiben Sie zusätzlich eine `main`-Methode
`public static void main(String[] args)`, die alle Zahlen zwischen -1 und einer festgelegten konstanten Zahl `N` auf ihre Primzahl-Eigenschaft überprüft.

Danke für eure Aufmerksamkeit!

