

Tutorium Programmieren

Tut Nr.3: Checkstyle, Kontrollstrukturen

Michael Friedrich | 12. / 14.11.2013

INSTITUT FÜR THEORETISCHE INFORMATIK



- 1 Checkstyle
- 2 Kontrollstrukturen
- 3 Getter / Setter
- 4 Einschub
- 5 Tutoriumsaufgabe
- 6 Ende

- Beinhaltet Konventionen, um Code einheitlich und damit leserlich zu machen
 - Wir bewegen uns im Spielraum, der uns der Compiler bietet
- Daher Pflicht ab diesem Übungsblatt
 - in 2 Stufen, vorerst nur Checkstyle 1
- Daher Pflicht ab diesem Übungsblatt
 - in 2 Stufen, vorerst nur Checkstyle 1
 - Sonst: Abzug!

- Zunächst, wie schon letzte Woche erwähnt
 - Whitespace verpflichtend um
 - = (Zuweisung)
 - +, + =, -, - =
 - {, }
 - ==, !=, <, <=, >, >=
 - &&, ||
 - if, else, for, while, return
 - ...

- Kein Whitespace nach
 - Whitespace verpflichtend um
 - (Bitweises Komplement)
 - ++ (Prefix-Inkrementierung, z.B. ++i;)
 - – (Prefix-Dekrementierung, z.B. –i;)
 - . (Punkt)
 - - (Unäres Minus, z.B. -5)
 - + (Unäres Plus, z.B. +4)
 - ...

- Kein Whitespace nach
 - Whitespace verpflichtend um
 - (Bitweises Komplement)
 - ++ (Prefix-Inkrementierung, z.B. ++i;)
 - – (Prefix-Dekrementierung, z.B. –i;)
 - . (Punkt)
 - - (Unäres Minus, z.B. -5)
 - + (Unäres Plus, z.B. +4)
 - ...

- Spätestens der Praktomat **prüft** ob ihr euch an die Richtlinien hält
 - nur konforme Lösungen werden angenommen (ca. 3ÜB)
 - Um einer stressigen Abgabe vorzubeugen, solltet ihr Checkstyle in eure IDE einbinden
 - .xml Dateien auf <http://baldur.iti.uka.de/programmieren/>

→ Demo für Eclipse

Wofür?

Kontrollstrukturen nutzen wir, um den linearen Programmfluss zu steuern.

- ermöglicht Fallunterscheidungen - **if**
- verhindert redundanten Code - **while**

if-then-else - Verzweigungen

```
if (boolean1) {  
  //code to run if boolean1 is true  
} elseif (boolean2) {  
  //code to run if boolean2 is true AND boolean1 is false  
} else {  
  //code to run otherwise  
}
```

Hinweis: boolean kann auch ein zusammengesetzter Ausdruck sein
(Operatoren dazu siehe letztes Tut)

while-Schleifen

```
while (boolean) {  
  //code to run while boolean is true  
}
```

→ Bedingung vor erstem Schleifendurchlauf erfüllt

do-while - Schleifen

```
do{  
  //code to run while boolean is true  
}while (boolean)
```

→ Was ist hier anders?

for-Schleifen

```
for (var ; condition; var modification) {  
    //code to run in loop  
}
```

Beispiel:

```
for (int i = 0; i < n; i++) {  
    // loop  
}
```

Alternativ kann „i“ auch vorher initialisiert werden, die Zuweisung muss aber hier geschehen. Vorteile?

switch

```
switch (var) {  
  case 1:  
    // code if var is 1  
    break;  
  case 2:  
  case 3:  
    // code if var is 2 or 3  
    break;  
  default:  
    // code to run if none of the above conditions hold  
    break;  
}
```

Switch über **primitive Datentypen** möglich

Wie letztes Mal erwähnt ist String ein Sonderfall - Seit Java 1.7 auch (offiziell) möglich

- Methoden, die kontrollierten Zugriff auf Attribute ermöglichen, statt Zugriff mit Punkt

Getter

```
returntype getVar(){  
    return this.var;  
}
```

Setter

```
void setVar(varType param) {  
    this.var = param;  
}
```

- Nutzen?
 - Prinzip der **Kapselung**

Modifier	Class	Package	Subclass	World
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
<i>no modifier*</i>	✓	✓	✗	✗
private	✓	✗	✗	✗

- Es gibt 2 weitere Modifikatoren

final - definiert eine Konstante

```
final int MAX_VALUE;
```

static - definiert eine Klassenvariable

```
final static int NUMBER_OF_WHEELS;
```

Klassenvariable bedeutet, dass dieses Attribut nicht an ein bestimmtes Objekt gebunden ist.

- Wir wollen das bisher Gelernte umsetzen und uns einen eigenen Datentyp bauen
Die rationalen Zahlen

→ Brainstorming

- Was brauchen wir also alles?
 - Ausgabe (toString)
 - Addition
 - Substraktion
 - Beides mit int (Hinweis: Überladen)
 - Kapselung

Danke für eure Aufmerksamkeit!

The ONLY VALID MEASUREMENT
OF CODE QUALITY: WTFs/MINUTE

