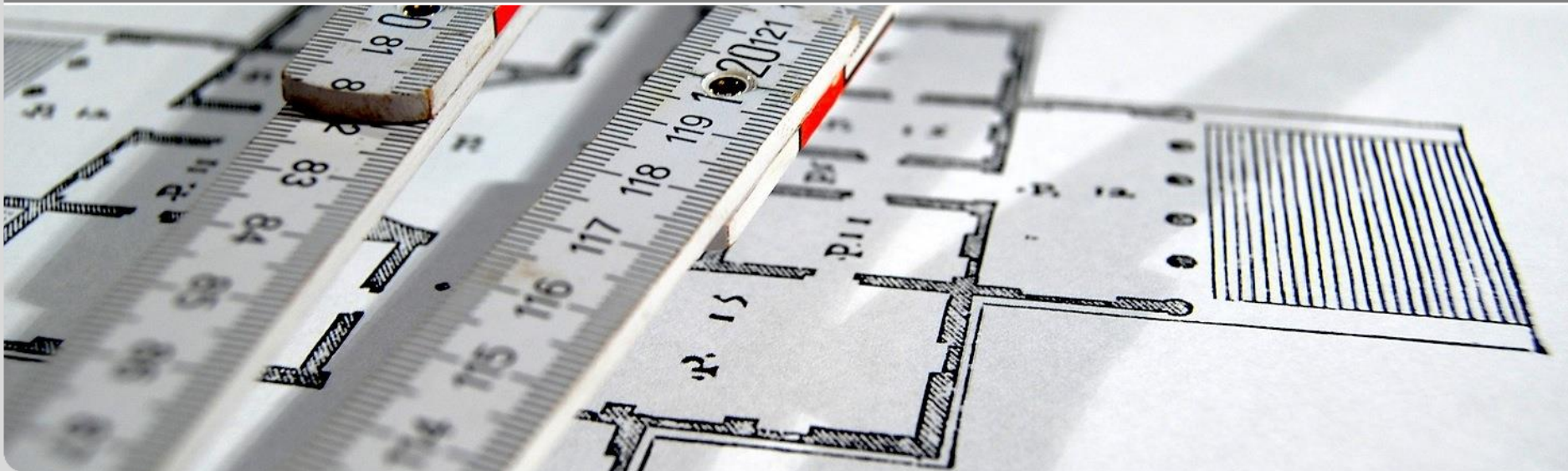


Programmieren-Tutorium Nr. 10

8. Tutorium | Jonas Ludwig
Vererbung, JavaDoc, Reguläre Ausdrücke

Architecture-driven Requirements Engineering – Institut für Programmstrukturen und Datenorganisation – Fakultät für Informatik



Was machen wir heute?

- Korrektur ÜB2
- JavaDoc
- Reguläre Ausdrücke
- Vererbung

Organisatorisches

- **Zusatztutorium am 07.01.19!**
- Anmeldung für den Übungsschein bis 08.01.20 12:00
- Präsenzübung am 15.01.20 17:30 – 19:00
 - Vorbereitung am 14.01.20 im Tutorium
 - Beispielaufgaben im ILIAS im Tutoriums-Ordner



- <https://b.socrative.com/login/student/>
- Raum-Name: **PROGGENTUT**

■ Aufgabe A Ø4.11P von 5P

- Häufiger Fehler: Falsche Anzahl an Iterationen (eine zu viel / zu wenig)
- Überprüfung der Kommandozeilenargumente unvollständig
 - Was passiert, wenn gar keine eingegeben werden?
 - Was passiert, wenn zu wenige eingegeben werden?
 - Was passiert, wenn zu viele eingegeben werden?

■ Aufgabe B Ø3.58P von 4P

- Geforderte Methodensignaturen einhalten
- Bei Arrayindizes bzw. Schleifenvariablen immer aufpassen!

Übungsblatt 2

■ Aufgabe C Ø7.36P von 11P

- getPassword() nicht wirklich sinnvoll. Besser:

```
public boolean checkPassword(String password) {  
    return password.equals(this.password);  
}
```

- Fehlende Begründung für Getter/Setter
- Aufpassen mit Counter für Article/Account im Shop!
 - Was passiert, wenn Article/Account in der Mitte vom Array gelöscht wird?
- Nicht emptyAccount o.ä. sondern null verwenden
- Testet eure Programme! Bei dieser Aufgabe war sogar die geforderte Ausgabe auf dem Übungsblatt angegeben!

Abschreiben → Kein Übungsschein im WS

→ Keine Teilnahme an Abschlussaufgaben im WS

Kommentare

■ Implementierungskommentare

- `/* [Kommentar] */`
- `// [Kommentar]`

■ Dokumentationskommentare

- Über jeder Klasse
- Über jedem Attribut
- Über jeder Methode
- `/** [Kommentar] */`

```
/**  
 * Only a example class  
 */  
class Example {  
    /*  
     * Example of imple-  
     * mentation comments  
     * with multiple lines.  
     */  
    int i; // the Integer  
    double /* the Double */ d;  
}
```

- Javadoc-Kommentare sind für Entwickler, die nichts vom eigentlichen Quelltext sehen Können, ihn aber dennoch selber nutzen wollen

Javadoc bei Klassen

■ @author

- Wird verwendet um den Autor einer Klasse zu beschreiben
- `@author [Name]`

■ @version

- Erzeugt einen Versionseintrag (Maximal einmal pro Klasse)
- `@version [Version]`

■ Beispiel

```
/**  
 * A bike model in Java  
 *  
 * @author Yves Schneider  
 * @version 1.2  
 */  
public class Bike {  
    //...  
}
```


Javadoc bei Methoden

■ @param

- Beschreibung eines Parameters der Methode
- `@param [Name] [Beschreibung]`

■ @return

- Beschreibung des Rückgabewerts der Methode
- `@return [Beschreibung]`

■ Beispiel

```
/**  
 * Returns the positive square root of a value.  
 *  
 * @param a the value  
 * @return the positive square root  
 */  
double sqrt(double a) {  
    //...  
}
```

Reguläre Ausdrücke und die Klasse String

■ `String[] split(String regex)`

Teilt die Zeichenfolge an den Stellen die dem angegebenen regulären Ausdruck übereinstimmen

■ `"boo:and:foo".split(":"); // {"boo","and","foo"}`

■ `"boo:and:foo".split("o"); // {"b","","":and:f"}`

■ `String replace(String target, String replacement)`

Ersetzt jede Zeichenfolge die dem angegebenen regulären Ausdruck übereinstimmt mit einer neuen Zeichenfolge

■ `"the war of baronets".replace("r", "y");`

■ `boolean matches(String regex)`

Sagt, ob diese Zeichenfolge dem angegebenen regulären Ausdruck übereinstimmt

Reguläre Ausdrücke

- Ein regulärer Ausdruck ist eine Zeichenkette, die der Beschreibung von Mengen von Zeichenketten mit Hilfe bestimmter syntaktischer Regeln dient.
- `String args[] = arg.split("\\s+");`
- `String inNum = args[0].replace(",", ".");`
- `boolean isValid =
inNum.matches("((-|\\s+)?[0-9]+(\\.([0-9]+)?)");`

Escape-Sequenz

- Zeichenkombinationen, die aus einem umgekehrten Schrägstrich \ gefolgt von einem Buchstaben oder einer Zahlenkombination, bestehen.
 - \n ■ \u00df ■ \"
- Eine Escape-Sequenz wird als ein einzelnes Zeichen betrachtet.
- Java wird versuchen einen umgekehrten Schrägstrich in einem String als Anfang einer Escape-Sequenz zu behandeln!
- Wenn tatsächlich ein umgekehrten Schrägstrich in dem regulären Ausdruck enthalten sein soll, muss die Escape-Sequenz für den umgekehrten Schrägstrich verwendet werden: \\

Reguläre Ausdrücke in Java

Zeichen	
a	Das Zeichen a
\\	Backslash
\'	Einfaches Anführungszeichen
\"	Doppeltes Anführungszeichen
\?	Fragezeichen
*	Sternchen
\+	Pluszeichen
\.	Punkt
\t	Tabulator
\n	Zeilenumbruch (LF)

■ `String input = args[3].replace("\\\"", "\"");`

Reguläre Ausdrücke in Java

Zeichenklassen

[abc]	Das Zeichen a oder b oder c
[^abc]	Alles außer a, b oder c
[A-Z]	Alle Zeichen von A bis Z
[0-9]	Alle Zeichen von 0 bis 9
[a-zA-Z]	Alle Zeichen von a bis z oder A bis Z
[a-d1-7]	Alle Zeichen von a bis d oder 1 bis 7

- `String args[] = arg.split(" ");`
- `String input = args[0].replace("[^A-Z]", "#");`
- `boolean isID = str.matches("[1-9][0-9][0-9][0-9][0-9][0-9][0-9]");`

Reguläre Ausdrücke in Java

Vordefinierte Zeichenklassen

.	Beliebiges Zeichen
\d	Ziffern: [0-9]
\D	Alles außer Ziffern: [^0-9]
\s	Leerraumzeichen: [\t\n\r\x08\x0c]
\S	Alles außer Leerraumzeichen: [^\s]
\w	Wortzeichen: [a-zA-Z_0-9]
\W	Alles außer Wortzeichen: [^\w]

■ `String args[] = arg.split("\\s");`

■ `boolean isID = str.matches("[1-9]\\d\\d\\d\\d\\d");`

Reguläre Ausdrücke in Java

Wiederholungen

?	einmal oder gar nicht
*	mehrmals oder gar nicht
+	mindestens einmal
{n}	exakt n-mal
{n, }	mindestens n-mal
{n, m}	n- bis m-mal

■ `String args[] = arg.split("\\s+");`

■ `boolean isID = str.matches("[1-9](\\d){6}");`

Reguläre Ausdrücke in Java

Grenzen

<code>^</code>	Zeilenanfang
<code>\$</code>	Zeilenende
<code>\b</code>	Wortgrenze
<code>\A</code>	Anfang der Eingabe
<code>\Z</code>	Ende der Eingabe

Logische Operatoren

<code>ab</code>	a gefolgt von b
<code>a b</code>	a oder b
<code>(a)</code>	a als Gruppe

■ `boolean isValid =`
`inNum.matches("^((-|\\+)?[0-9]+(\\. [0-9]+)?)$");`

Reguläre Ausdrücke in Java

■ Zum Nachlesen

- <http://www.vogella.com/tutorials/JavaRegularExpressions/article.html>
- http://openbook.rheinwerk-verlag.de/javainsel9/javainsel_04_007.htm

■ Zum Ausprobieren

- <https://regexr.com/>

Polymorphie

- Es besteht die Möglichkeit, dass Objekte einer Unterklasse wie Objekte ihrer Oberklasse auftreten können
- Erfordert eine Operation einen Parameter vom Typ einer Oberklasse kann ohne weiteres ein Objekt vom Typ einer Unterklasse übergeben werden
- Das Objekt der Unterklasse bietet mindestens alle Methoden und Attribute, wie ein Objekt der Oberklasse
 - Es bietet sogar noch mehr!

Polymorphie Beispiel

```
Kraftfahrzeug[] fahrzeuge = new Kraftfahrzeug[7];
```

```
fahrzeuge[0] = new Kraftfahrzeug();
```

```
fahrzeuge[1] = new Automobil();
```

```
fahrzeuge[2] = new Motorrad();
```

```
fahrzeuge[3] = new LKW();
```

```
fahrzeuge[4] = new PKW();
```

```
fahrzeuge[5] = new Cabriolet();
```

```
fahrzeuge[6] = new Limousine();
```

```
for (int i = 0; i < fahrzeuge.length; i++) {  
    fahrzeuge[i].zeigeAttribute();  
}
```

Polymorphe Argumente

- Aufgrund der Polymorphie ist es möglich, Methoden zu schreiben, denen Argumente übergeben werden, die einem allgemeinen Typ entsprechen
- Als Parameter können dann Objekte dieser Klasse und Objekte aller Unterklassen dieser Klasse übergeben werden

```
public class Tankstelle {  
    public void tanken(Kraftfahrzeug fahrzeug) {  
        //...  
    }  
}
```

```
Tankstelle t = new Tankstelle();  
LKW lkw = new LKW();  
t.tanken(lkw);  
PKW pkw = new PKW();  
t.tanken(pkw);
```

Aufgabe

- Implementieren Sie eine statische Methode, welche eine Instanz der Klasse `Person` entgegen nimmt und, wenn dieses nicht `null` ist, die Methode `print()` aufruft.
- Falls jedoch die übergebenen Referenz auf eine Person `null` ist, wird lediglich die Zeichenkette `"null"` auf der Kommandozeile ausgegeben.

```
public static void print(Person person) {
```

```
}
```

Aufgabe – Lösung


- Implementieren Sie eine statische Methode, welche eine Instanz der Klasse `Person` entgegen nimmt und, wenn dieses nicht `null` ist, die Methode `print()` aufruft.
- Falls jedoch die übergebenen Referenz auf eine Person `null` ist, wird lediglich die Zeichenkette `"null"` auf der Kommandozeile ausgegeben.

```
public static void print(Person person) {  
    if (person == null) {  
        System.out.println("null");  
    } else {  
        person.print();  
    }  
}
```

Der Operator instanceof

- Oft ist es notwendig, den Typ eines Objektes festzustellen, das von einer polymorphen Referenz referenziert wird
- Der **instanceof**-Operator testet zur Laufzeit den eigentlichen Typ des referenzierten Objekts und liefert `true` oder `false` als Ergebnis

```
public void tanken(Kraftfahrzeug fahrzeug) {  
    if (fahrzeug instanceof LKW) {  
        //...  
    } else if (fahrzeug instanceof PKW) {  
        //...  
    } else if (fahrzeug instanceof Motorrad) {  
        //...  
    }  
}
```



So besser nicht!
Lieber mit
dynamischer
Bindung arbeiten.

Dynamische Bindung

- Vererbung macht es einfacher neue Klassen hinzuzufügen

```
public class Tankstelle {  
    public void tanken(Kraftfahrzeug fahrzeug) {  
        fahrzeug.betanken();  
    }  
}
```

```
public class Kraftfahrzeug {  
    //...  
    public void betanken() {  
        //...  
    }  
}
```

Dynamische Bindung

- Welche Methode wird verwendet?
 - Methode wird überschrieben
 - Objekt der Unterklasse wird in einer Oberklasse-Variable gespeichert
- Dynamische Bindung entscheidet zur Laufzeit anhand des Objekttyps, welche Methode tatsächlich aufgerufen wird
- Statische Typisierung garantiert, dass es die Methode gibt
- Upcasts schalten nicht die dynamische Bindung ab!
- Attributvariablen sind statisch gebunden

Dynamische Bindung – Aufgabe

```
class O {  
    int x = 42;  
    void f() { System.out.println(x); h(); }  
    void g() { x = 103; }  
    void h() { System.out.println(x); }  
}
```

```
class U extends O {  
    int x = 17;  
    void g() { f(); }  
    void h() { System.out.println(x); }  
}
```

```
class Main {  
    public static void main(String[] args) {  
        O y = new U(); y.g();  
    }  
}
```

■ Ausgabe: 42 17

getClass()-Methode


- „Nachteile“ des instanceof-Operators:
 - Bei der Kompilierung muss die Klasse bereits bekannt sein
 - Der Operator testet ein Objekt auf seine Hierarchie
- Eine Bessere Variante ist die **getClass()**-Methode
 - [http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#getClass\(\)](http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#getClass())
- `objekt.getClass()`
 - Gibt die Referenz auf die zur Laufzeit verwendete Klasse des Objekts.
- `Klasse.class`
 - Gibt die statische Referenz der Klasse zurück.
- `objekt.getClass() == Klasse.class`
 - Ist das „gleichwertig“ zu: `objekt instanceof Klasse?`

equals-Methode

- Eine equals()-Methode sollte Objekte auf Gleichheit prüfen
 - [http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#equals\(java.lang.Object\)](http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#equals(java.lang.Object))

```
public class Person {  
    String name;  
    int age;  
  
    /**  
        * JavaDoc  
    */  
    @Override  
    public boolean equals(Object obj) {  
        //...  
    }  
}
```

Nicht unbedingt der
gleiche Typ!



equals-Methode 1

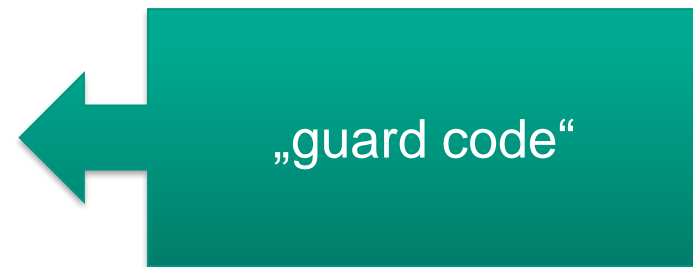
```
/*  
 * (non-Javadoc)  
 *  
 * @see java.lang.Object#equals(java.lang.Object)  
 */
```

@Override

```
public boolean equals(Object obj) {
```

```
    if (this == obj) {  
        return true;  
    } else if (obj == null) {  
        return false;  
    }
```

```
    boolean ret = true;
```



equals-Methode 2

```
    if (this.getClass() != obj.getClass()) {  
        ret = false;  
    } else {  
  
        final Person other = (Person) obj;  
  
        if (this.age != other.age) {  
            ret = false;  
        } else if (this.name == null) {  
            if (other.name != null) {  
                ret = false;  
            }  
        } else if (!this.name.equals(other.name)) {  
            ret = false;  
        }  
    }  
  
    return ret;  
}
```

- Überschreiben Sie die Methode `equals(Object obj)` in den Klassen Person, Student und Professor.
 - `this == obj`?
 - `obj == null`?
 - `this.getClass == obj.getClass`?
 - Auf Klasse casten und Attribute nacheinander auf Gleichheit überprüfen

Aufgabe – Person

```
@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj == null || getClass() != obj.getClass()) {
        return false;
    }

    Person other = (Person) obj;
    if (age != other.age) {
        return false;
    }
    if (name == null) {
        if (other.name != null) {
            return false;
        }
    } else if (!name.equals(other.name)) {
        return false;
    }

    return true;
}
```

Aufgabe - Student

```
@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj == null || getClass() != obj.getClass()) {
        return false;
    }

    Student other = (Student) obj;
    if (!super.equals(obj) || matriculationNumber !=
other.matriculationNumber) {
        return false;
    }

    return true;
}
```

Aufgabe – Professor

@Override

```
public boolean equals(Object obj) {  
    if (this == obj) {  
        return true;  
    }  
    if (obj == null || getClass() != obj.getClass()) {  
        return false;  
    }  
  
    Professor other = (Professor) obj;  
    if (!super.equals(obj)) {  
        return false;  
    }  
    if (chair == null) {  
        if (other.chair != null) {  
            return false;  
        }  
    } else if (!chair.equals(other.chair)) {  
        return false;  
    }  
  
    return true;  
}
```

Abstrakte Klassen und abstrakte Methoden

- Nicht immer soll eine Klasse sofort ausprogrammiert werden
 - Wenn die Oberklasse lediglich Methoden für die Unterklassen vorgeben möchte, aber nicht weiß, wie sie diese implementieren soll
- Die Methode muss mit dem Schlüsselwort **abstract** versehen werden
 - `public abstract void einsteigen(int personen);`
- Da nun die Klasse eine abstrakte Methode besitzt, wird sie zu einer abstrakten Klasse
 - Damit ist es nicht mehr möglich, eine Instanz dieser abstrakten Klasse zu erzeugen
- Eine Unterklasse der abstrakten Klasse erbt nun die abstrakte Methode und muss diese überschreiben
 - Tut sie das nicht, muss sie selbst als abstrakte Klasse definiert werden

Abstrakte Klassen und abstrakte Methoden

```
public abstract class Kraftfahrzeug {  
    // ...  
    public abstract void einsteigen(int personen);  
}
```

```
public class Motorrad extends Kraftfahrzeug {  
    // ...  
    @Override  
    public void einsteigen(int personen) {  
        // ...  
    }  
}
```

Abstrakte Klassen und abstrakte Methoden

```
public abstract class Automobil extends Kraftfahrzeug {  
    // ...  
}
```

```
public class LKW extends Automobil {  
    // ..  
    @Override  
    public void einsteigen(int personen) {  
        // ...  
    }  
}
```

Aufgabe 6 – Lösung

- Da die Klasse Person lediglich als Strukturelement innerhalb einer Klassenhierarchie dient, sollte diese sich auch nicht instanziierten lassen.
- Verändern Sie ihre Klassen so, dass die Klasse Person nicht mehr instanziiert werden kann und nur noch die Klassen Student und Professor wie gewohnt instanziiert werden können.

```
public abstract class Person {  
    // ...  
}
```

Zugriffsmodifikatoren und Vererbung

Modifizierer	Klasse selbst	Paketklassen	Unterklassen	Sonstige Klassen
public	✓	✓	✓	✓
protected	✓	✓	✓	X
ohne	✓	✓	(✓)	X
private	✓	X	X	X


- Wenn ein Element **private** ist, ist es nur in dieser Klasse sichtbar in der es definiert wurde
 - Private Methoden sind immer statisch gebunden!

- Wenn eine Element **protected** ist, ist es nur innerhalb seiner Klasse, deren abgeleiteten Klassen und allen Klassen im selben Paket sichtbar
 - Der Modifikator wird verwendet, wenn es nur für Unterklassen Sinn hat, das betreffenden Element zu verwenden

Zugriffsmodifikatoren und Vererbung

```
public abstract class Kraftfahrzeug {  
    private Color farbe;  
    private String name;  
}
```

```
public class Motorrad extends Kraftfahrzeug {  
    //...  
    System.out.println(this.name);  
}
```



Nicht innerhalb der
Unterklassen
sichtbar!

Zugriffsmodifikatoren und Vererbung

```
public abstract class Kraftfahrzeug {  
    protected Color farbe;  
    protected String name;  
}  
  
public class Motorrad extends Kraftfahrzeug {  
    //...  
    System.out.println(this.name);  
}
```

Zugriffsmodifikatoren – Aufgabe

```
public abstract class Kraftfahrzeug {  
    private String name;  
    protected String typ;  
    public String farbe;  
    public abstract String id;  
}
```

✓
✓
✓
X

```
public class Motorrad extends Kraftfahrzeug {  
    System.out.println(this.name);  
    System.out.println(super.name);  
    System.out.println(this.typ);  
    System.out.println(super.typ);  
    System.out.println(this.farbe);  
    System.out.println(super.farbe);  
    System.out.println(this.id);  
    System.out.println(super.id);  
}
```

X
X
✓
✓
✓
✓
X
X

Fragen?

Was machen wir im nächsten Tutorium?

- Interfaces
- Vorbereitung Präsenzübung
- (Generics)

Organisatorisches

- **Zusatztutorium am 07.01.20!**
- Anmeldung für den Übungsschein bis 08.01.20 12:00
- Präsenzübung am 15.01.20 17:30 – 19:00
 - Vorbereitung am 14.01.20 im Tutorium
 - Beispielaufgaben im ILIAS im Tutoriums-Ordner

Schöne Weihnachten und guten Rutsch!

