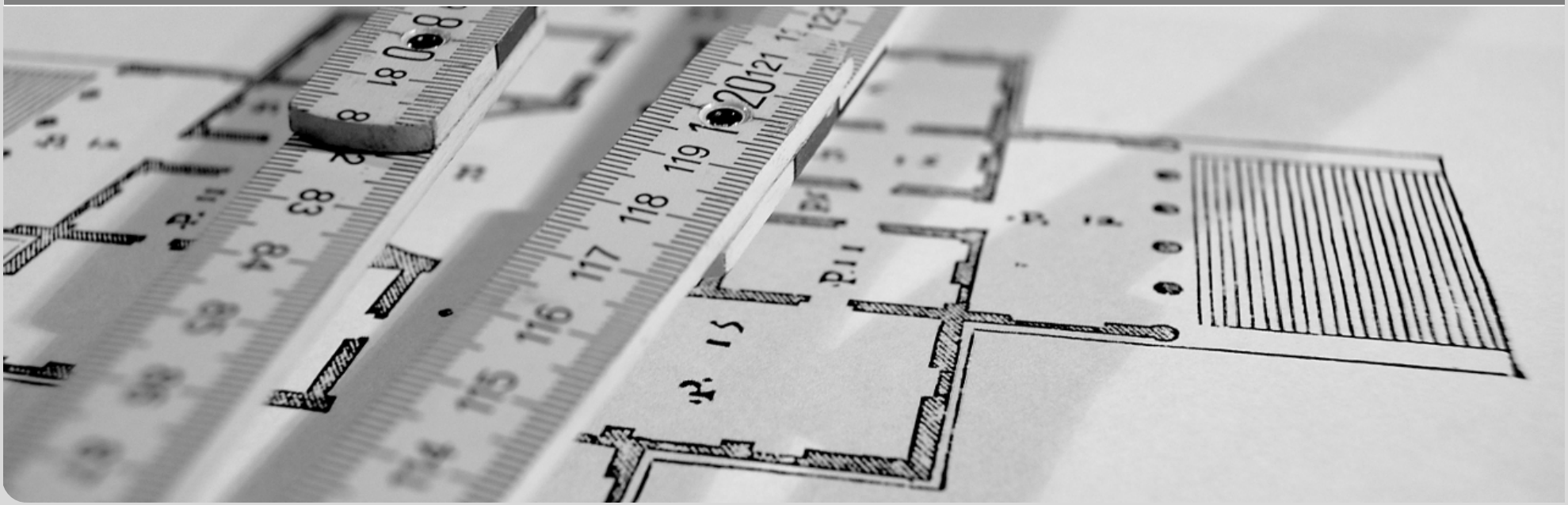
Karlsruher Institut für Technologie

Software Design and Quality

# Programmieren
# 07. Tutorium

Robin Rüde | 15. Dezember 2014



# Gliederung

# 1

# Blatt 3

# Tastatur


- Checkstyle erwarteet 4 Leerzeichen für einen Tab.
- –> Leertaste 4 Mal für einen Tab drücken.
- auch die Entertaste darf benutzt werden
- v. a. bei Texten auf die Länge der Zeilen achten.

# Teil A

- es sollte das eigentliche Tupel geändert werden (außer bei toSet())
- **genau** lesen, was gefordert ist
- toString(): **kein** Komma nach letzter Ziffer
- Schleifenbedingung bei arrays: `i < a.length`!
- keine endlos Rekursion. `remove(int i) { this.remove(i); }`

# Teil B

- **nicht nur** eine Gottklasse
- Eulerkreis $\leftrightarrow$ Kantenanzahl jedes Knotens gerade
- Spalten- bzw. Zeilensumme der Adjazenzmatrix musste gerade sein.
- 1,2,1 ist der Graph 1–2 (nur eine Kante)
- eher for als while-Schleifen verwenden (vor allem für Schleifen, die nur zählen)
- sinnvolle Kommentare - so wenig wie möglich, so viel wie nötig

# Teil D

- Modellierung: Klasse Graph mit einer Liste von Vertex
- Jeder Vertex/Knoten hat eine Liste von adjazenten (= verbundenen) Knoten
- Iteriere über die Liste der Knoten von Graph und prüfe bei jedem Knoten, ob die Kantenzahl gerade ist
- wenn dies bei jedem Knoten zutraf -> true, sonst false
- als Liste z. Bsp. NaturalNumberTuple
- Hinzufügen einer Kante zwischen 2 Knoten $\leftrightarrow$ 2 Knoten vorher nicht verbunden!

# Musterlösung

Musterlösung zum 3. Blatt befindet sich auf der Vorlesungsseite.

# 2

# Vererbung

# Vererbung

- Generalisierung bzw. Spezialisierung von Klassen
- „is a“-Beziehung
- Unterklasse: Subtyp von Oberklasse
- Oberklasse: vererbende Klasse
- Verwendung: Codeduplikation vermeiden

```
class Animal extends LivingOrganism { ... }
class Plant extends LivingOrganism { ... }
class Bird extends Animal { ... }
```

„Animal *is a* LivingOrganism“

# Eigenschaften der Vererbung

## Was erbt die Unterklasse von der Oberklasse? (3 Eigenschaften)

# Eigenschaften der Vererbung

## Was erbt die Unterklasse von der Oberklasse? (3 Eigenschaften)

- Attribute

# Eigenschaften der Vererbung

## Was erbt die Unterklasse von der Oberklasse? (3 Eigenschaften)

- Attribute
- Methoden

# Eigenschaften der Vererbung

## Was erbt die Unterklasse von der Oberklasse? (3 Eigenschaften)

- Attribute
- Methoden
- nested classes
- **keine** Konstruktoren

- Unterklasse hat speziellere/mehr Eigenschaften als Oberklasse
- Substitutionsprinzip: Unterklasse **muss** im Kontext der Oberklasse verwendet werden können

# Vererbung in Java

- Unterklasse `extends` Oberklasse
- Zuweisung: Oberklasse o = new Unterklasse();
- Aber **nicht**: Unterklasse u = new Oberklasse();

# Überschreiben einer Methode

- @Override-Annotation
- gleicher Methodenname
- gleiche Parameter
- Rückgabetyp: Subtyp des vorherigen Typs oder gleicher Typ (Spezialisierung des Rückgabetyps)

# Dynamische Bindung

Welche Methode wird jeweils aufgerufen?

```
1  class Animal {
2          void makeNoise() { }
3  }
4  class Dog extends Animal {
5          @Override
6          void makeNoise() {
7                  bark();
8          }
9  }
10 Dog d = new Dog();
11 Animal a = new Animal();
12 Animal b = d;
13
14 d.makeNoise();
15 a.makeNoise();
16 b.makeNoise();
```

## Aufgerufene Methoden

# Dynamische Bindung

Welche Methode wird jeweils aufgerufen?

```
1  class Animal {
2          void makeNoise() { }
3  }
4  class Dog extends Animal {
5          @Override
6          void makeNoise() {
7                  bark();
8          }
9  }
10 Dog d = new Dog();
11 Animal a = new Animal();
12 Animal b = d;
13
14 d.makeNoise();
15 a.makeNoise();
16 b.makeNoise();
```

## Aufgerufene Methoden

- Dog.makeNoise()

# Dynamische Bindung

Welche Methode wird jeweils aufgerufen?

```
1  class Animal {
2          void makeNoise() { }
3  }
4  class Dog extends Animal {
5          @Override
6          void makeNoise() {
7                  bark();
8          }
9  }
10 Dog d = new Dog();
11 Animal a = new Animal();
12 Animal b = d;
13
14 d.makeNoise();
15 a.makeNoise();
16 b.makeNoise();
```

## Aufgerufene Methoden

- Dog.makeNoise()
- Animal.makeNoise()

# Dynamische Bindung

Welche Methode wird jeweils aufgerufen?

```
1  class Animal {
2          void makeNoise() { }
3  }
4  class Dog extends Animal {
5          @Override
6          void makeNoise() {
7                  bark();
8          }
9  }
10 Dog d = new Dog();
11 Animal a = new Animal();
12 Animal b = d;
13
14 d.makeNoise();
15 a.makeNoise();
16 b.makeNoise();
```

## Aufgerufene Methoden

- Dog.makeNoise()
- Animal.makeNoise()
- Dog.makeNosie()

# Dynamische Bindung

- Aufruf der passenden Methode wird zur Laufzeit entschieden
- Hängt nur vom Typ des **referenzierten** Objekts an.
- Hängt also nicht vom Typ der Variablen ab die das Objekt referenziert

# Überschreiben von Attributen

In Java **nicht** möglich. Attribute werden überschattet.

# Schlüsselwort: super für Konstruktoren

```
class Mammal {
    String noise;
    Mammal(String noise) {
        this.noise = noise;
    }
    void makeNoise() {
        System.out.println(noise);
    }
}
class Dog extends Mammal {
    Dog() {
        super("Woof!");
    }
}
new Dog().makeNoise(); // Woof!
```

# Schlüsselwort: super

```
class Mammal { int age; }
class Dog extends Mammal {
    int age;
    void setMammalAge(int age) {
        super.age = age; // age of Mammal (super class)
    }
    void setDogAge(int age) {
        this.age = age; // age of Dog (this class)
    }
}
```

- Im Normalfall Überschattung vermeiden!

# Schlüsselwort: super

Alter von Dog ist äquivalent zum Alter von Mammal

```
class Mammal {
    int age;
    void setAge(int age) {
        this.age = age;
    }
}
class Dog extends Mammal {
    // some dog stuff
}
```

# Konstruktoren und Sichtbarkeiten

- **immer** Konstruktor der Oberklasse mit `super(...);` aufrufen
- wenn super(...); nicht aufgerufen wird, wird implizit super(); im Konstruktor der Subklasse aufgerufen

| Modifier | in Klasse | in Paket | in Subklassen | überall sichtbar |
|---|---|---|---|---|
| private | ✓ | ✗ | ✗ | ✗ |
| default | ✓ | ✓ | ✗ | ✗ |
| protected | ✓ | ✓ | ✓ | ✗ |
| public | ✓ | ✓ | ✓ | ✓ |

# final

Wenn eine Klasse als final deklariert ist,
kann von ihr nicht geerbt werden.
Wenn eine Methode als final deklariert ist,
kann sie in einer Unterklasse nicht überschrieben werden.

# instanceof

- Syntax: `Object instanceof Class`
- Prüft ob Object vom Typ Class ist.
- `Dog d; d instanceof Mammal == true;`
- `Cat c; c instanceof Dog == false;`

# instanceof

- Syntax: `Object instanceof Class`
- Prüft ob Object vom Typ Class ist.
- `Dog d; d instanceof Mammal == true;`
- `Cat c; c instanceof Dog == false;`
- Casten: upcasten geht immer: Mammal m = (Mammal) d;
- Casten: downcasten **nur** nach Prüfung, ob Objekt den richtigen Typ hat:

```
void playWith(Mammal m) {
    if (m instanceof Dog) {
        Dog dog = (Dog) m;
        dog.playFetch(new Ball());
    } else {
        System.err.println("You can't play with that!");
    }
}
```

# Klasse Object

- java.lang.Object ist Wurzel der Klassenhierarchie
- jedes Objekt erbt von Object (implizit)
- Methoden von Object:
- u. a. equals(Object o), String toString(), clone()

# abstract classes

- Klasse ohne instantiierte Objekte
- abstrakte Methode besitzt keine Implementierung -> muss in Unterklasse implementiert werden
- Deklaration: `abstract class <ClassName>` bzw. `abstract <ReturnTyp> methodName(<Parameters>);`

```
abstract class Mammal {
    abstract makeNoise();
}
class Dog extends Mammal {
    makeNoise() {
        System.out.println("Woof!");
    }
}
new Dog().makeNoise(); // Woof!
```

# 3

# Aufgaben

# Aufgabe: Tiere

Schreibe eine abstrakte Klasse `Animal`. Jedes Tier hat einen `Namen` und ein `Alter`. Die Klasse Animal beinhaltet für diese Attribute getter und setter.
Die Methode `makeNoise()` gibt das jeweilige Geräusch als Text auf der Kommandozeile aus.
Es gibt drei verschiedene Tiere: Katze ("Miau"), Dog("Wuff") und Kuh ("Muuh").

Die Liste `animals` soll eine Katze, einen Hund und eine Kuh enthalten:

```
for (Animal animal : animals) {
        animal.makeNoise();
}
// Output:
// Muuh
// Miau
// Wuff
```

# Aufgabe: Shape

Implementiere ein Programm, das mit dieser Main-Methode die angegebene Ausgabe erzeugt

```
public static void main(String[] args) {
        List<Shape> shapes = new ArrayList<Shape>();
        shapes.add(new Rectangle(100,200)); // Breite, Höhe
        shapes.add(new Circle(100)); // Radius
        shapes.add(new Triangle(100, 50)); // Grundseite, Höhe

        for (Shape s : shapes) {
                System.out.println(s.getName() + ": " + s.getArea());
        }
}
/* Output:
 * Rectangle: 20000.0
 * Circle: 31415.926535897932
 * Triangle: 2500.0
 */
```

# TODO-List

Input: `java TodoList <Pfad zur Datei> <Personenname>`

## list.txt

```
Hans;Wäsche waschen
Peter;Klo putzen
Hans;Abspülen
```

## Ausgabe

```
$ java TodoList list.txt Hans

Aufgaben von Hans
- Wäsche waschen
- Abspülen
```

# Ende

Fragen?

# Ende

Fragen?
Vielen Dank für eure Aufmerksamkeit!