

PGdP Tutorium: Neunte Stunde

Benedikt Werner

München, 19. Dezember 2017



Wiederholung: Klassen und Instanzen

Klasse

Keyword: **static**

Klasse Dog

```
static int numberOfDogs;  
static Dog createDog();
```

```
String name;  
int age;  
  
void bark();  
void run();
```

Instanzen

Instanz Winston

```
String name = "Winston";  
int age = 3;
```

Instanz Bello

```
String name = „Bello“;  
int age = 7;
```

Wiederholung: Klassen und Instanzen

Klasse

```
int x = Dog.numberOfDogs;  
Dog.createDog();
```

Instanzen

```
Dog winston = new Dog();  
winston.bark();  
  
new Dog().run();
```

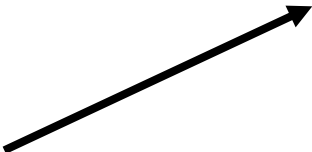
Wiederholung: Instanzen

Konstruktor

- "Methode ohne Rückgabewert"
- Name = Klassenname
- Wird bei **new** aufgerufen

```
class Dog {  
    String name;  
  
    public Dog(String name) {  
        this.name = name;  
    }  
}
```

new Dog("Winston");



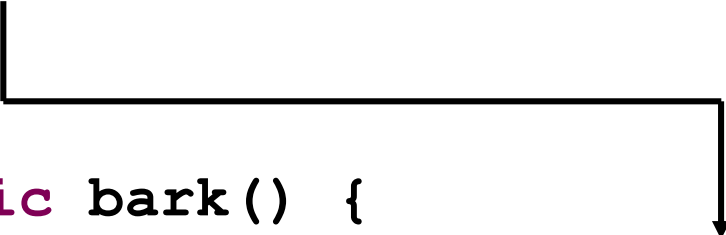
Wiederholung: Getter und Setter

```
class Dog {  
    private String name;           Attribute private  
  
    public String getName() {      Methoden public  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

Vorteil: Man kann zum Beispiel nur einen Getter implementieren um nur das Lesen eines Werts zu erlauben oder beim setzen prüfen ob der neue Wert überhaupt gültig ist. **Allgemein:** mehr Kontrolle

Wiederholung: this

```
Dog winston = new Dog("Winston");  
winston.bark();
```



```
public bark() {  
    System.out.println(this.name + " barked");  
}
```

Beispiel

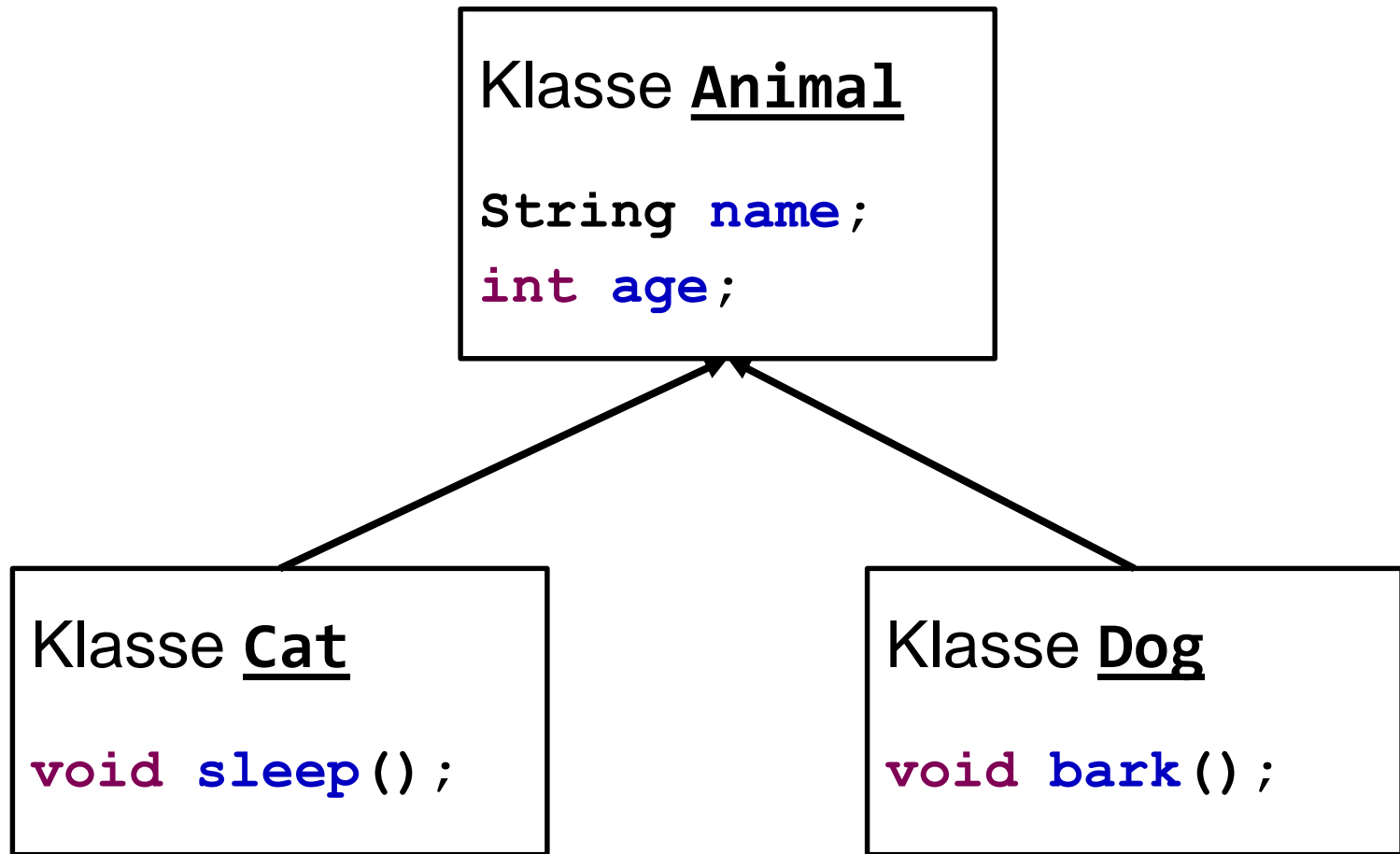
```
class Cat {                                class Dog {
    String name;                            String name;
    int age;                                int age;

    void sleep() {                          void bark() {
        //...                               //...
    }                                        }
}
```

Doppelter Code

Außerdem: Man kann Katzen und Hunde nicht mischen.
z.B. in einem Array mit Hunden und Katzen

Lösung: Vererbung



Lösung: Vererbung

```
class Animal {  
    String name;  
    int age;  
}
```

```
class Cat extends Animal {  
    void sleep() {  
        System.out.println(name);  
    }  
}
```

Cat erbt von Animal



Membervariablen und Methoden von Animal
können auch in Cat verwendet werden
(wenn sie nicht private sind)

Aufgabe 9.1 – Dynamischer vs statischer Typ

- Statischer Typ: Typ einer Variable/Instanz beim Kompilieren
 - Wird benutzt um zu prüfen, ob nur Attribute und Methoden abgerufen werden, die die Klasse besitzt
- Dynamischer Typ: Typ einer Variable/Instanz beim Ausführen
 - Wird benutzt um die konkrete Methode die ausgeführt werden soll zu bestimmen

```
class A
```

```
class B extends A
```

```
A a = new A();
```

```
A b = new B();
```

```
B b = new B();
```

Aufgabe 9.1

```
public class A
    public int min(C c, B b) { return 0; }           // Methode 1
    public void min(A a, B b) {}                   // Methode 2

public class B extends A
    public void min(A a1, A a2) {}                 // Methode 3

public class C extends B
    public B min(A a, C c) { return new B(); }     // Methode 4

A a = (B) (new C());
B b = new B();
C c = new C();
c.min(a, c);           // Aufruf 1
b.min(a, (B)c);        // Aufruf 2
((B)c).min(c, c);      // Aufruf 3
((A)b).min(a, b);      // Aufruf 4
```

Aufgabe 9.3

- Der Ansatz aus Aufgabe 9.2 besitzt in manchen Situationen gewisse Nachteile:
 - Wenn es viele Unterklassen gibt ist die Implementierung einer Funktion auf sehr viele Klassen und damit Dateien verteilt
→ Code schwerer zu verstehen
 - Auch wenig benutzte Methoden, die nicht für alle Klassen gleich funktionieren müssen in jeder Klasse implementiert werden
→ Die Klassen werden unübersichtlicher, man sieht die wichtigen Methoden nicht mehr so gut
 - Andere Personen können die Funktionalität der Klassen nicht erweitern ohne die originalen Dateien zu verändern, was oft nicht möglich oder wünschenswert ist

Aufgabe 9.3 – Visitor Pattern

- Allgemeine Visitor-Oberklasse, die auf alle Klassen durch `accept(Visitor visitor)` angewendet werden kann
- Unterklassen implementieren eine bestimmte Funktionalität, die beim „besuchen“ einer Klasse ausgeführt wird

```
class myClass {  
    public void accept(Visitor v) {  
        v.visit(this);  
    }  
}  
  
class Visitor {  
    public void visit(MyClass m) {  
        System.out.println(m);  
    }  
}
```

```
MyClass someInstance = new MyClass();
```

```
someInstance.accept(new Visitor());
```

Aufgabe 9.3 – Visitor Pattern Rückgabewert

- Da die accept()-Methode immer **void** zurückgibt müssen Rückgabewerte anders geregelt werden
- Dazu kann der Rückgabewert einfach im Visitor gespeichert werden

```
class Visitor {  
    public int result;  
  
    public void visit(MyClass m) {  
        result = calculateSomething(m);  
    }  
}
```

Normalerweise mit Getter

```
MyClass someInstance = new MyClass();  
Visitor visitor = new Visitor()  
someInstance.accept(visitor);  
int result = visitor.result;
```

Aufgabe 9.4

- Codegerüst auch auf home.in.tum.de/~wernerbe/pgdp unter „Neunte Stunde“