ÜBUNGSSTUNDE W10

Einführung in die Programmierung

Ablauf

- Graph Execution
- Inheritance
  - *Grundprinzip*
  - *Object Klasse*
  - *Konstruktoren*
  - *Visability*
  - *Polymorphismus*
- Prüfungsaufgabe
- Kahoot

FEEDBACK BONUSAUFGABE

GRAPH EXECUTION

Übungsnachbesprechung

INHERITANCE

Theorie

GRUNDPRINZIP

Theorie

Grundprinzip

```
public class Animal {
    int weight;

    void eat(){
        println (“mampf”);
    }
    void sleep(){
    // Implementation
    }
}
```

Grundprinzip

```
public class Animal {
    int weight;

    void eat(){
        println (“mampf”);
    }
    void sleep(){
    // Implementation
    }
}
```

```
public class Cat {
    int weight;

    void eat(){
        println (“mampf”);
    }
    void sleep(){
    // Implementation
    }
    void meow(){
    // Implementation
    }
}
```

Grundprinzip

```
public class Animal {
    int weight;

    void eat(){
          println (“mampf”);
    }
    void sleep(){
    // Implementation
    }
}
```

```
public class Cat extends Animal {

void meow(){
    // Implementation
    }
}
```

```
public class Dog extends Animal {
int loudness;
void wuf(){
    // Implementation
    }
}
```

```
public static void main(String[] args) {
    Animal animal = new Animal();
    animal.weight = 10;
    animal.eat();
    animal.sleep();

    Cat cat = new Cat();
    cat.meow();
    cat.weight = 5;
    cat.eat();
    cat.sleep();

    Dog dog = new Dog();
    dog.loudness = 0;
    dog.wuf();
    dog.weight = 20;
    dog.eat();
    dog.sleep();
}
```

Animal

Cat

Animal

Dog

Animal

Grundprinzip

```
public class Animal {
   int weight;

   void eat(){
        println ("mampf");
   }
   void sleep(){
   // Implementation
   }
}
```

```
public class Cat extends Animal {

void meow(){
   // Implementation
   }
}
```

```
public class Dog extends Animal {
int loudness;
void wuf(){
   // Implementation
   }
}
```

```
public static void main(String[] args) {
    Animal animal = new Animal();
    animal.weight = 10;
    animal.eat();
    animal.sleep();

    Fish fish = new Fish();
    fish.weight = 10;
    fish.eat();
    fish.sleep();

}
```

Animal

Fish

Animal

Output:

mampf

mampf

Grundprinzip

```
public class Animal {
   int weight;

   void eat(){
        println ("mampf");
   }
   void sleep(){
   // Implementation
   }
}
```

```
public class Cat extends Animal {

void meow(){
   // Implementation
   }
}
```

```
public class Dog extends Animal {
int loudness;
void wuf(){
   // Implementation
   }
}
```

```
public class Fish extends Animal {

}
```

```
public static void main(String[] args) {
    Animal animal = new Animal();
    animal.weight = 10;
    animal.eat();
    animal.sleep();

    Fish fish = new Fish();
    fish.weight = 10;
    fish.eat();
    fish.sleep();

}
```

Animal

Fish

Animal

Fish

Animal

Output:

mampf

blubblub

The annotation @Override helps the compiler catch errors by ensuring that the annotated method is indeed intended to override a method from the superclass.

Grundprinzip

```
public class Animal {
   int weight;

   void eat(){
        println ("mampf");
   }
   void sleep(){
   // Implementation
   }
}
```

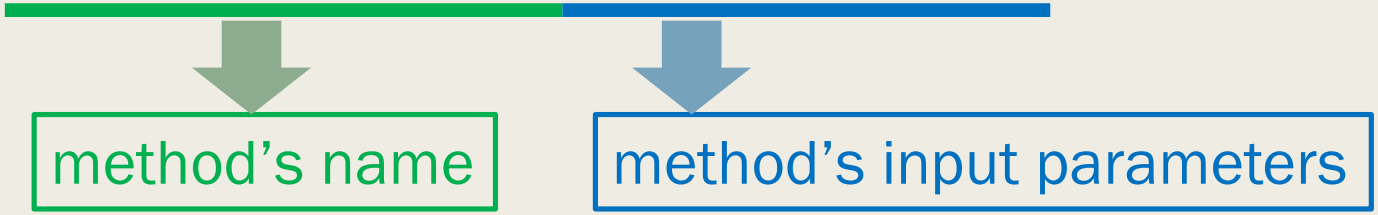
```
public class Cat extends Animal {

void meow(){
   // Implementation
   }
}
```

```
public class Dog extends Animal {
int loudness;
void wuf(){
   // Implementation
   }
}
```

```
public class Fish extends Animal {
@Override
void eat(){
        println ("blubblub");
   }
}
```

Reminder – Method Signature

```
public void addNumbers(int a, int b){...}
```



Examples:

```
public void ADDNUMBERS(int a, int b)
public void addNumbers(int c, int d)
public void addNumbers(double a, int c)
void addNumbers(int a, int b)
```

```
public static void main(String[] args) {
    Animal animal = new Animal();
    animal.weight = 10;
    animal.eat();
    animal.sleep();

    Fish fish = new Fish();
    fish.weight = 10;
    fish.eat();
    fish.sleep();

}
```

Animal

Fish

Animal

Fish

Animal

Output:

mampf

blubblub

The annotation @Override helps the compiler catch errors by ensuring that the annotated method is indeed intended to override a method from the superclass.

Grundprinzip

```
public class Animal {
   int weight;

   void eat(){
        println ("mampf");
   }
   void sleep(){
   // Implementation
   }
}
```

```
public class Cat extends Animal {

void meow(){
   // Implementation
   }
}
```

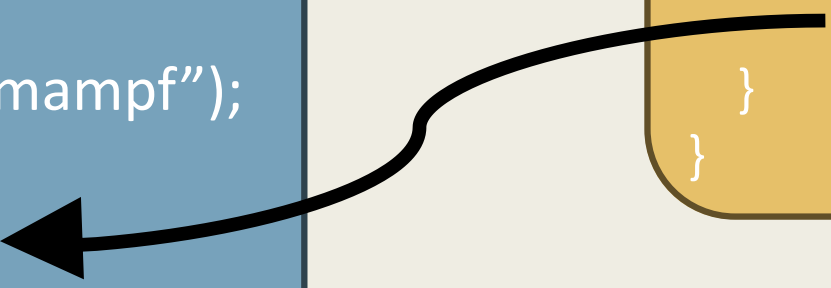
```
public class Dog extends Animal {
int loudness;
void wuf(){
   // Implementation
   }
}
```

```
public class Fish extends Animal {
@Override
void eat(){
        println ("blubblub")
   }
}
```

Super

```
public class Animal {
    int weight;

    void eat(){
        println (“mampf”);
    }
    void sleep(){
    // Implementation
    }
}
```

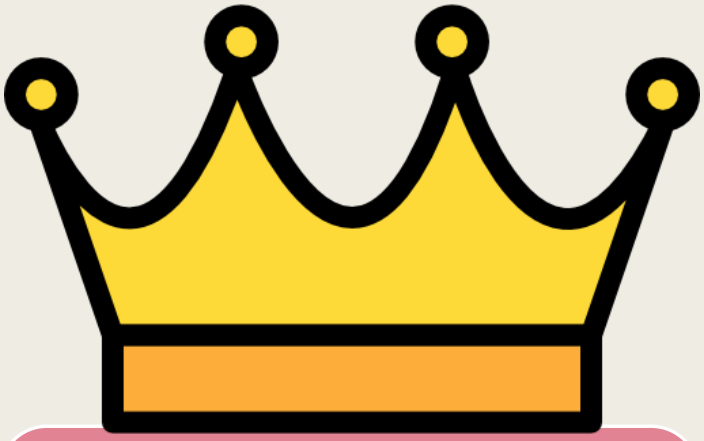
```
public class Fish extends Animal {
@Override
void eat(){
        println (“blubblub”);
        super.sleep()
    }
}
```



OBJECT KLASSE

Theorie

Klasse Object



Klasse Animal

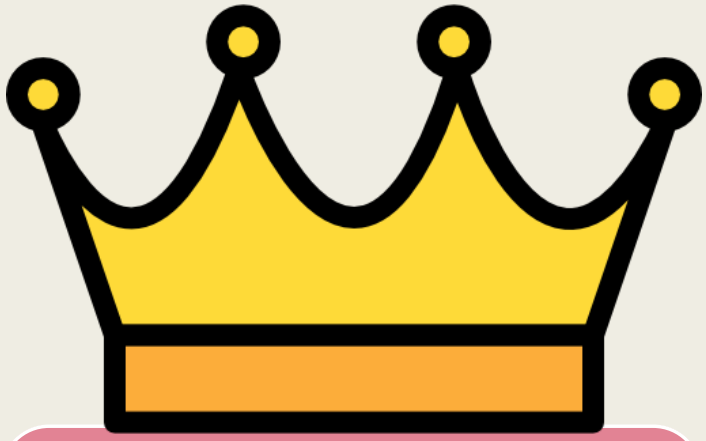
Direkte Vererbung der Klasse Object

Klasse Object

Indirekte Vererbung der Klasse Object

Klasse Fisch erweitert Animal

Klasse Object




```
public String toString() {
      return getClass().getName() + "@" + Integer.toHexString(hashCode());
}

public boolean equals(Object obj) {
      return (this == obj);
}
```

Klasse Object

```
public class Car {
    String brand;
    String model;

     car(String brand, String model){
          this.brand = brand;
          this.model = model;
     }
}
```

```
public static void main(String[] args) {
    Car firstVan = new Car("VW","Multivan");
    Car secondVan = new Car("VW","Multivan");

    System.out.println(firstVan.equals(secondVan));
}
```

Output: false

Klasse Object

```
public class Car {
    String brand;
    String model;

     car(String brand, String model){
          this.brand = brand;
          this.model = model;
     }
     @Override
     public boolean equals (Object obj){..}


}
```

Klasse Object

```
@Override
public boolean equals(Car otherCar) {
    boolean equalBrand = brand.equals(otherCar.brand);
    boolean modelBrand = model.equals(otherCar.model);

    return equalBrand&&modelBrand;
}
```

Klasse Object

```
@Override
public boolean equals(Object inObj) {
    if (!(inObj instanceof Car))return false;
    Car otherCar = (Car) inObj;
    boolean equalBrand = brand.equals(otherCar.brand);
    boolean modelBrand = model.equals(otherCar.model);

    return equalBrand&&modelBrand;
}
```

equals

```
public class Car {
    String brand;
    String model;
      car(String brand, String model){
            this.brand = brand;
            this.model = model;
      }
      @Override
      public boolean equals (Object obj){..}
}
```

```
public static void main(String[] args) {
    Car firstVan = new Car("VW","Multivan");
    Car secondVan = new Car("VW","Multivan");


System.out.println(firstVan.equals(secondVan));
}
```

Output: true

toString()

```
public static void main(String[] args) {
    Car bmw = new Car(”BMW",”M5");
    System.out.println(bmw);
}
```

```
@Override
public String toString() {
    return "Car{" +
            "brand='" + brand + '\'' +
            ", model='" + model + '\'' +
            '}';
}
```

KONSTRUKTOREN

Theorie

Konstruktoren

- **Wenn Klasse Y die Klasse `Base` erweitert**
  - **Y erbt die Methoden von `Base`**
  - **Y kann Methoden von `Base` überschreiben**
  - **Y erbt die Attribute von `Base`**
  - **Y erbt *nicht* die Konstruktoren von** Base

Default Konstruktor (Vorstellung)

```
public class Animal {
    int weight;

    void eat(){…}
    void sleep(){…}

     Animal(){
  //standard constructor
          weight = 0;
     }

}
```

```
public class Cat extends Animal {
     Cat(){ //standard constructor
          super();
     }
}
```

```
public static void main(String[] args) {
    Cat cat = new Cat();
    System.out.println(cat.weight);
}
```

Output: 0

Default Konstruktor (Vorstellung)

```
public class Animal {
    int weight;

    void eat(){…}
    void sleep(){…}

     Animal(){ //modified
          weight = 5;
     }

}
```

```
public class Cat extends Animal {
     Cat(){ //standard constructor
          super();
     }
}
```

```
public static void main(String[] args) {
    Cat cat = new Cat();
    System.out.println(cat.weight);
}
```

Output: 5

Default Konstruktor

```
public class Animal {
    int weight;

    void eat(){…}
    void sleep(){…}

     Animal(){ //modified
          weight = 5;
     }

}
```

```
public class Cat extends Animal {
     Cat(){ //modified
          weight = 10;
     }
}
```

```
public static void main(String[] args) {
    Cat cat = new Cat();
    System.out.println(cat.weight);
}
```

Output: 10

Default Konstruktor (Vorstellung)

```
public class Animal {
    int weight;

    void eat(){…}
    void sleep(){…}

     Animal(int weight){
     //modified
      this.weight = weight;
     }

}
```

```
public class Cat extends Animal {
     Cat(){ //standard constructor
          super();
     }
}
```

```
public static void main(String[] args) {
    Cat cat = new Cat();
    System.out.println(cat.weight);
}
```

Output: Error

Default Konstruktor

```
public class Animal {
    int weight;

    void eat(){…}
    void sleep(){…}

     Animal(int weight){
     //modified
      this.weight = weight;
     }

}
```

```
public class Cat extends Animal {
      Cat(int age){ //standard
            super(age);
      }
}
```

```
public static void main(String[] args) {
    Cat cat = new Cat(15);
    System.out.println(cat.weight);
}
```

Output: 15

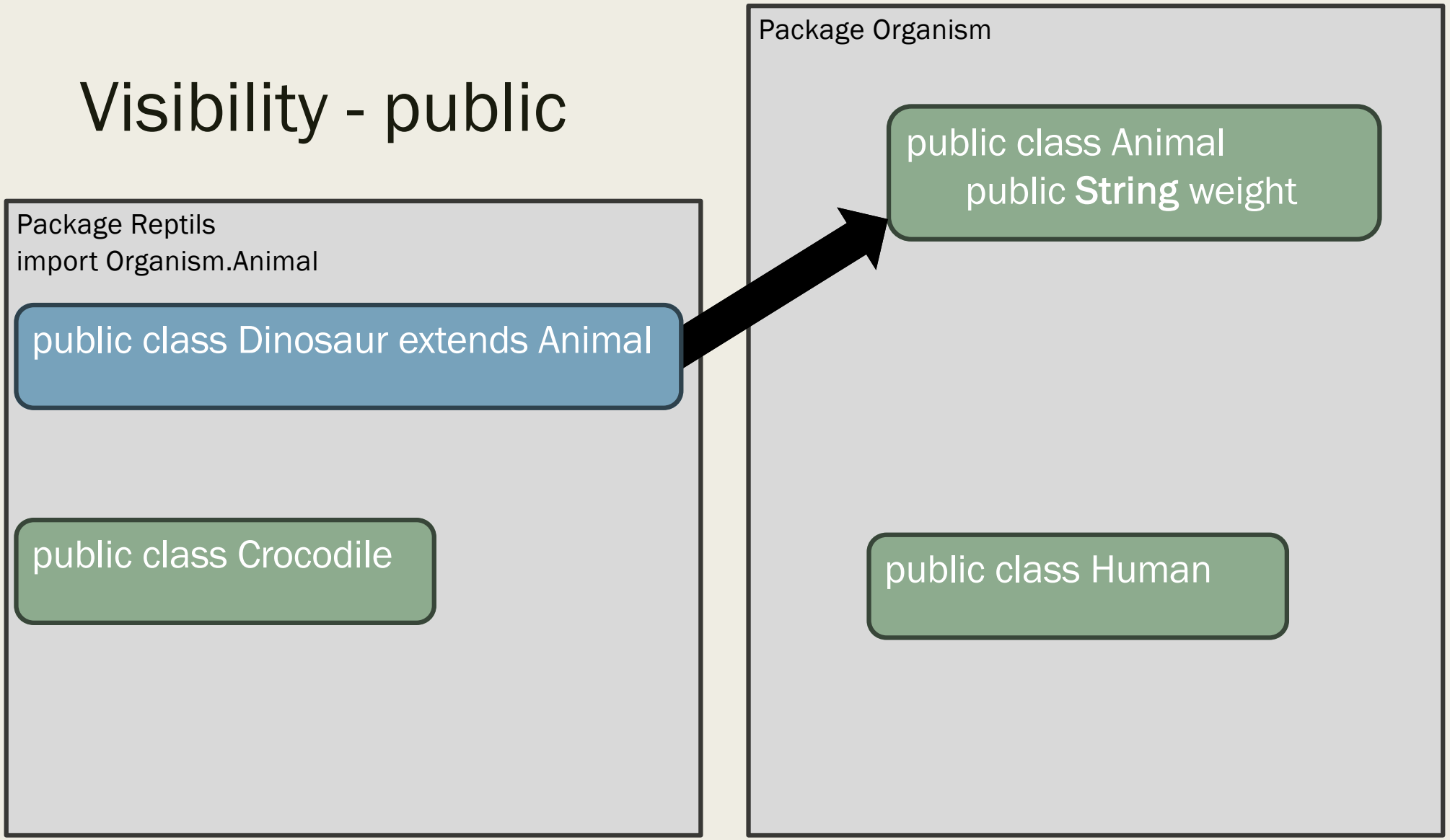
VISIBILITY

Theorie

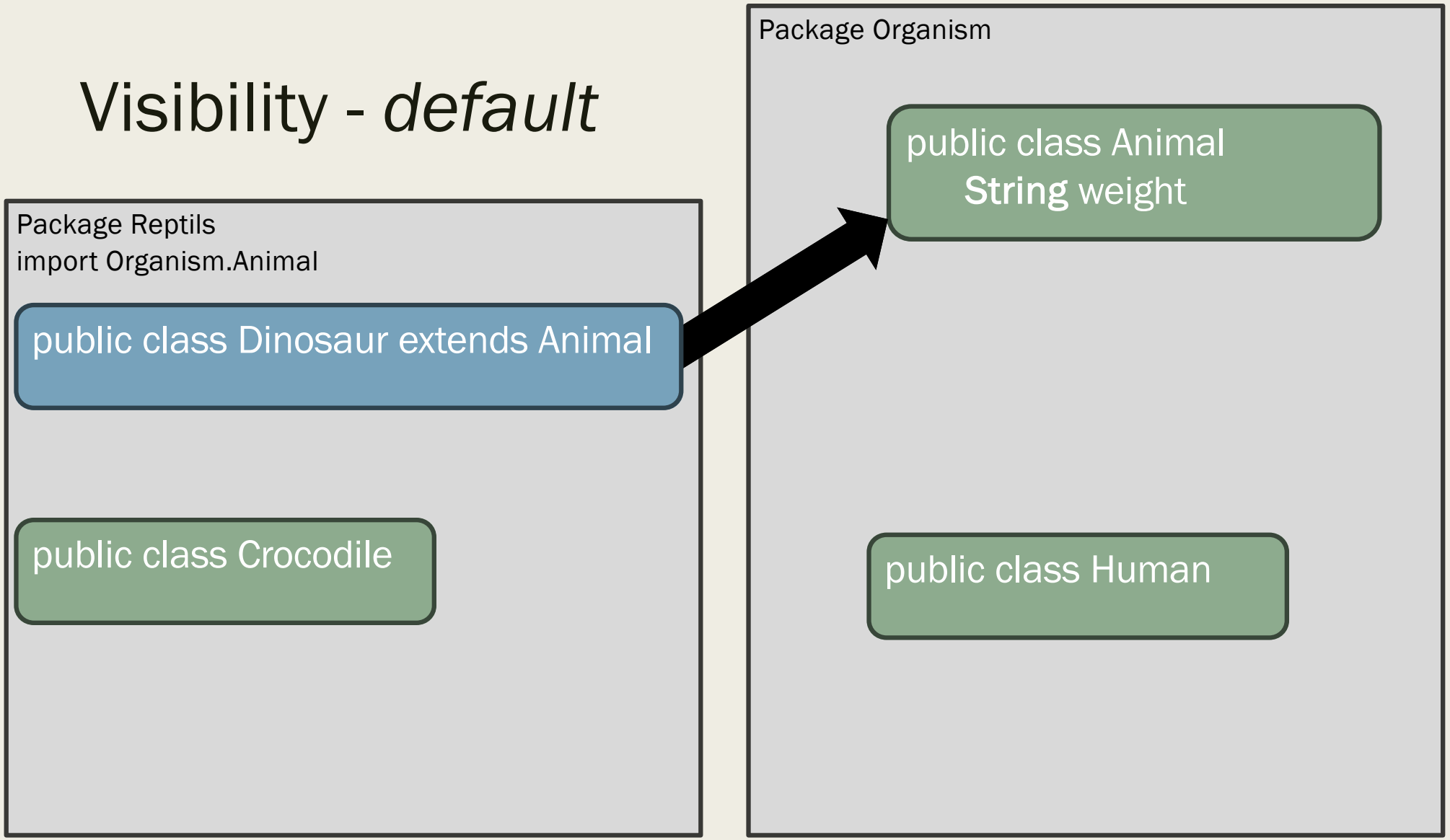
Visibility

Modifier	Description
public	The code is accessible for all classes
private	The code is only accessible within the declared class
default	The code is only accessible in the same package. This is used when you don't specify a modifier.
protected	The code is accessible in the same package and **subclasses**.

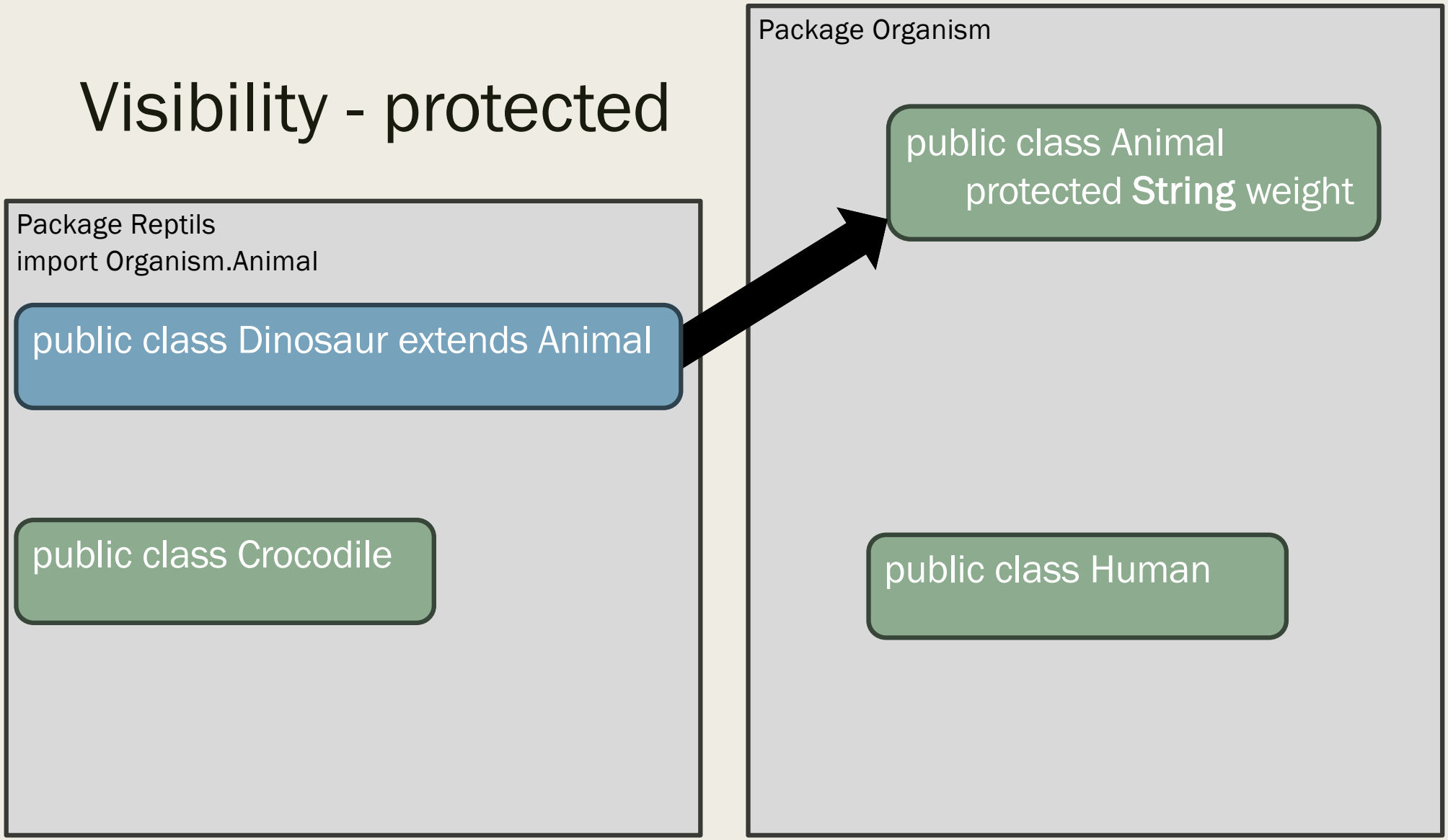
Visibility - public



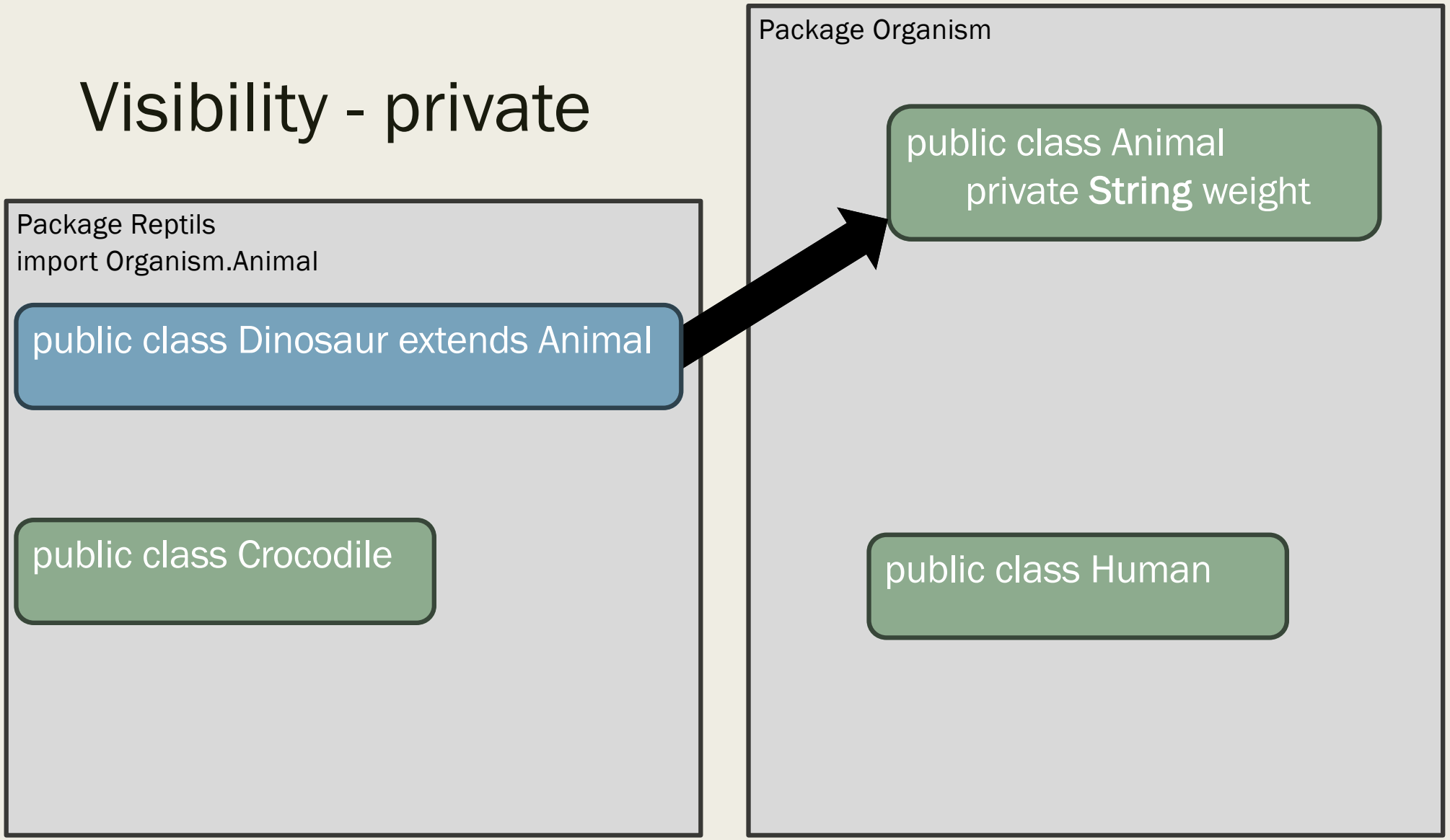
Visibility - *default*



Visibility - protected



Visibility - private

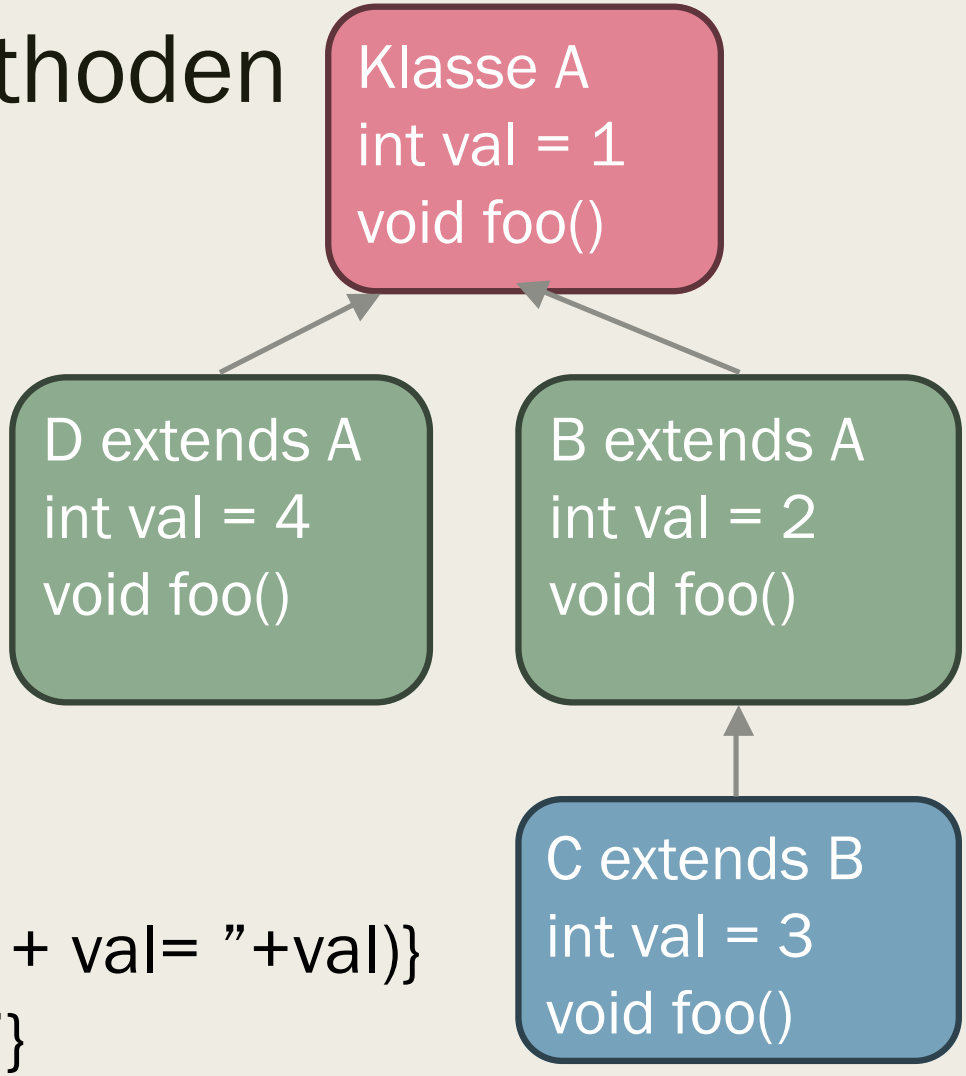


POLYMORPHISMUS

Theorie

Polymorphismus - Methoden

```
public static void main(String[] args) {
   //Code
}
```



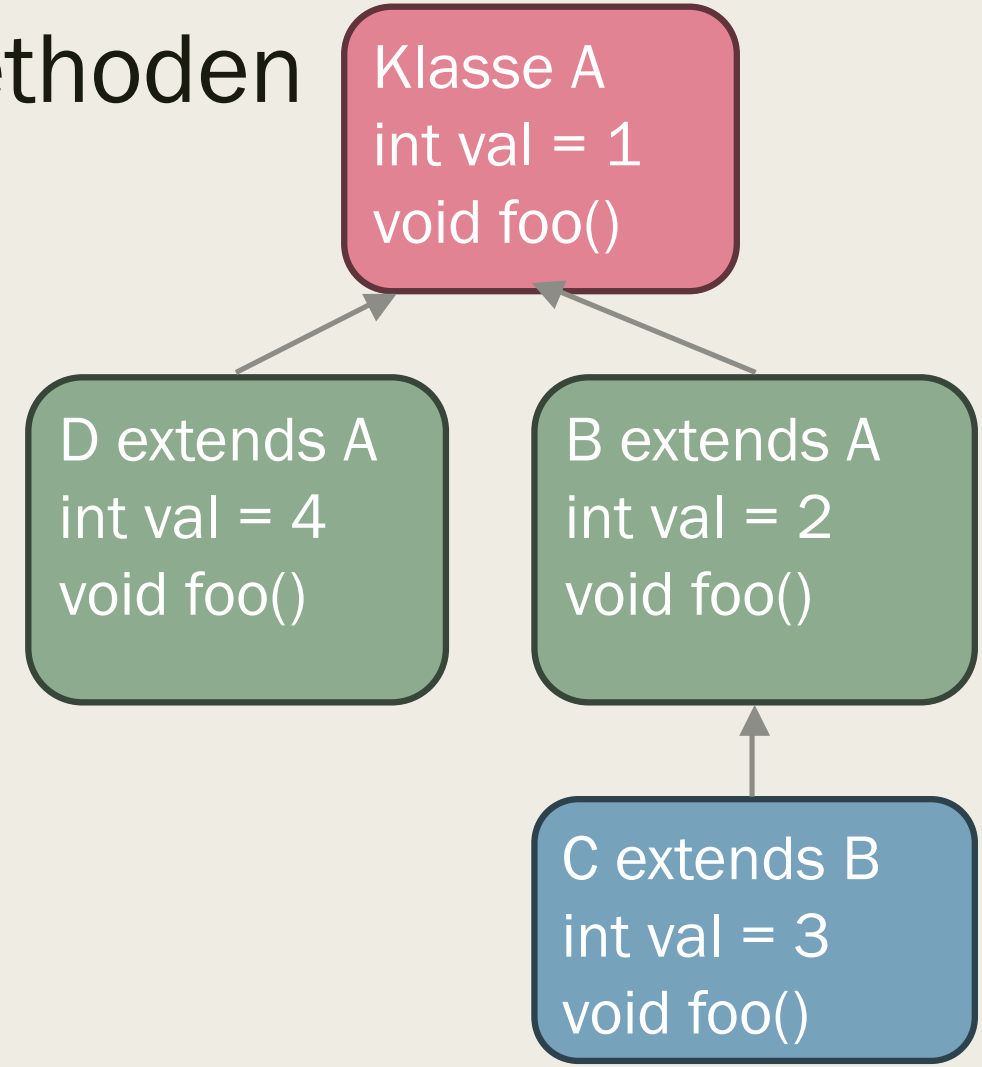
void foo(){print(className+” + val= ”+val)}

className = {“A”,”B”,”C”,”D”}

Polymorphismus - Methoden

```
public static void main(String[] args) {
    B b = new B();
    b.foo();
}
```

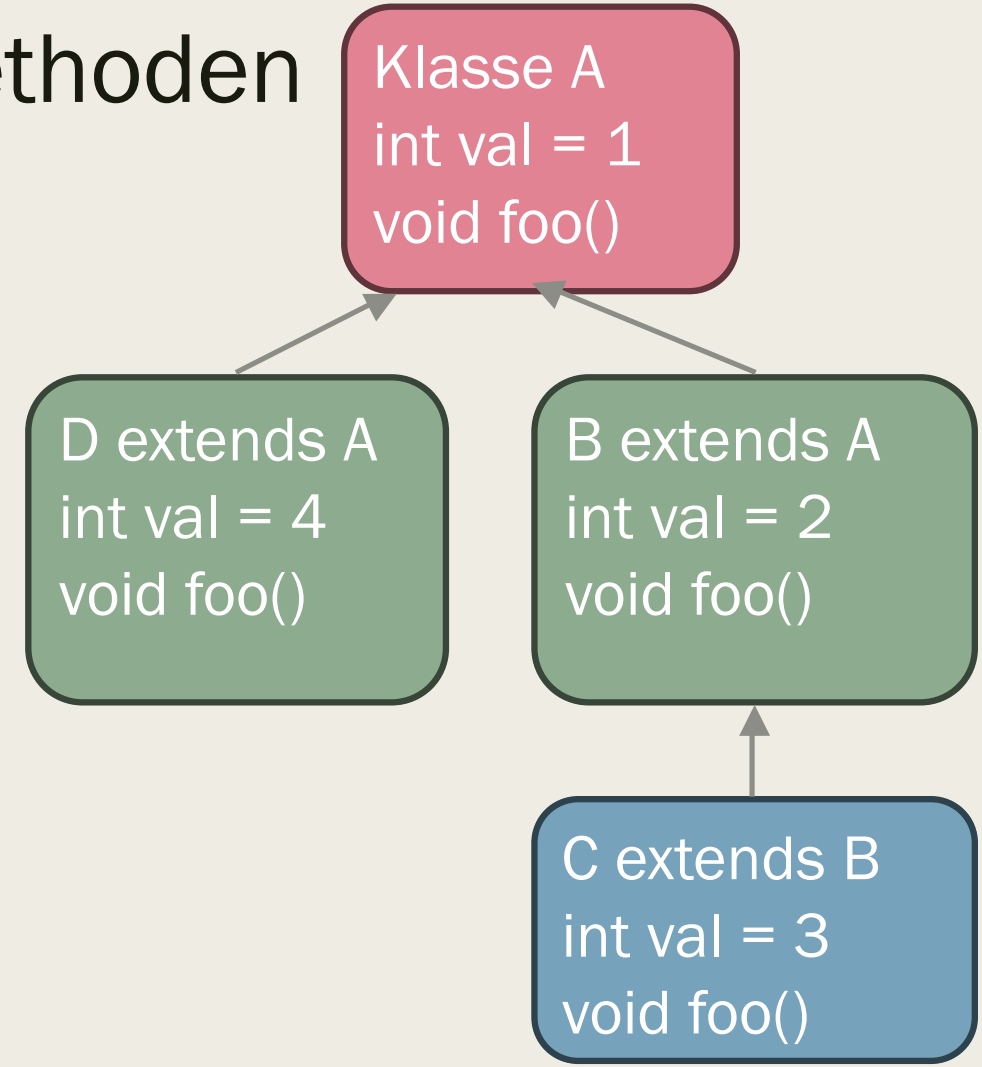
Output: B + val = 2



Polymorphismus - Methoden

```
public static void main(String[] args) {
    B b = new B();
    A bToA = (A)b;
    bToA.foo();
}
```

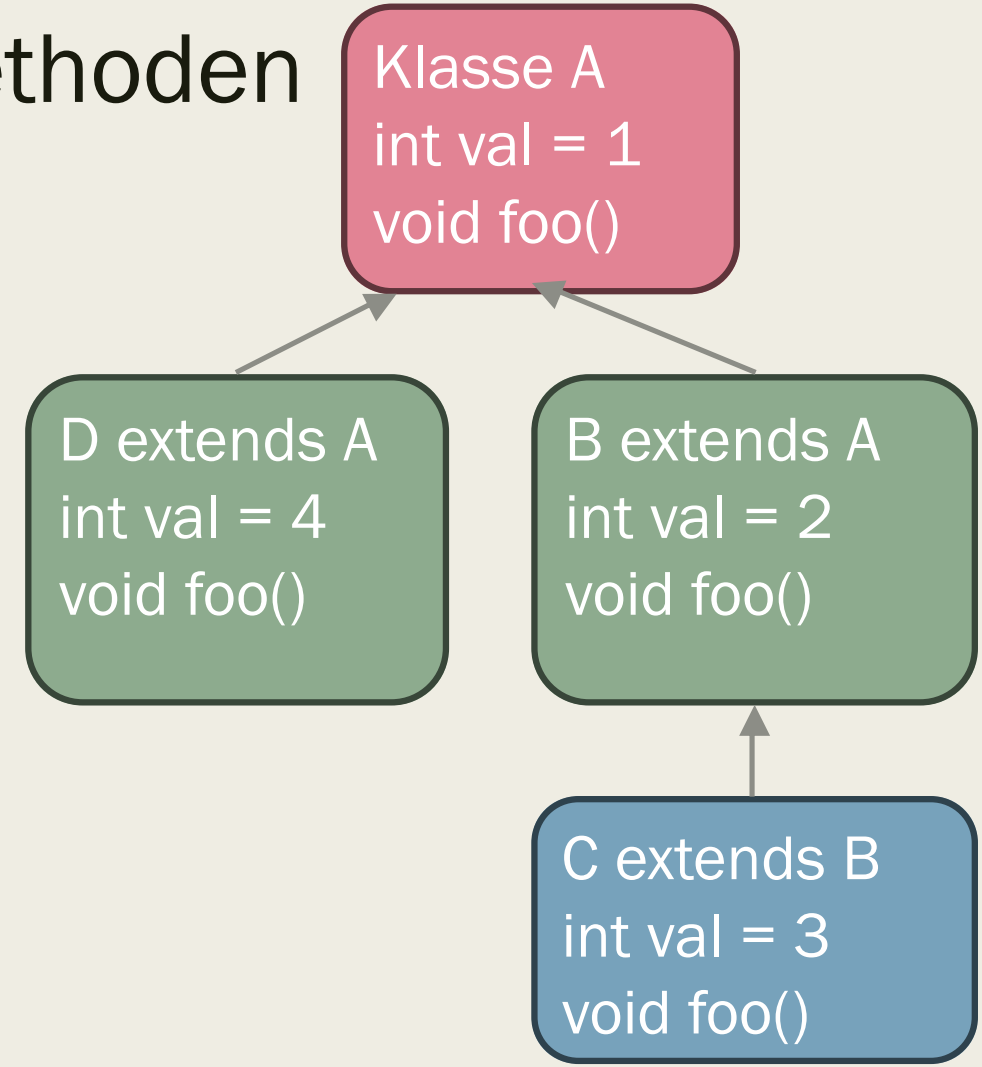
Output: B + val = 2



Polymorphismus - Methoden

```
public static void main(String[] args) {
    B b = new B();
    A bToA = b;
    bToA.foo();
}
```

Output: B + val = 2

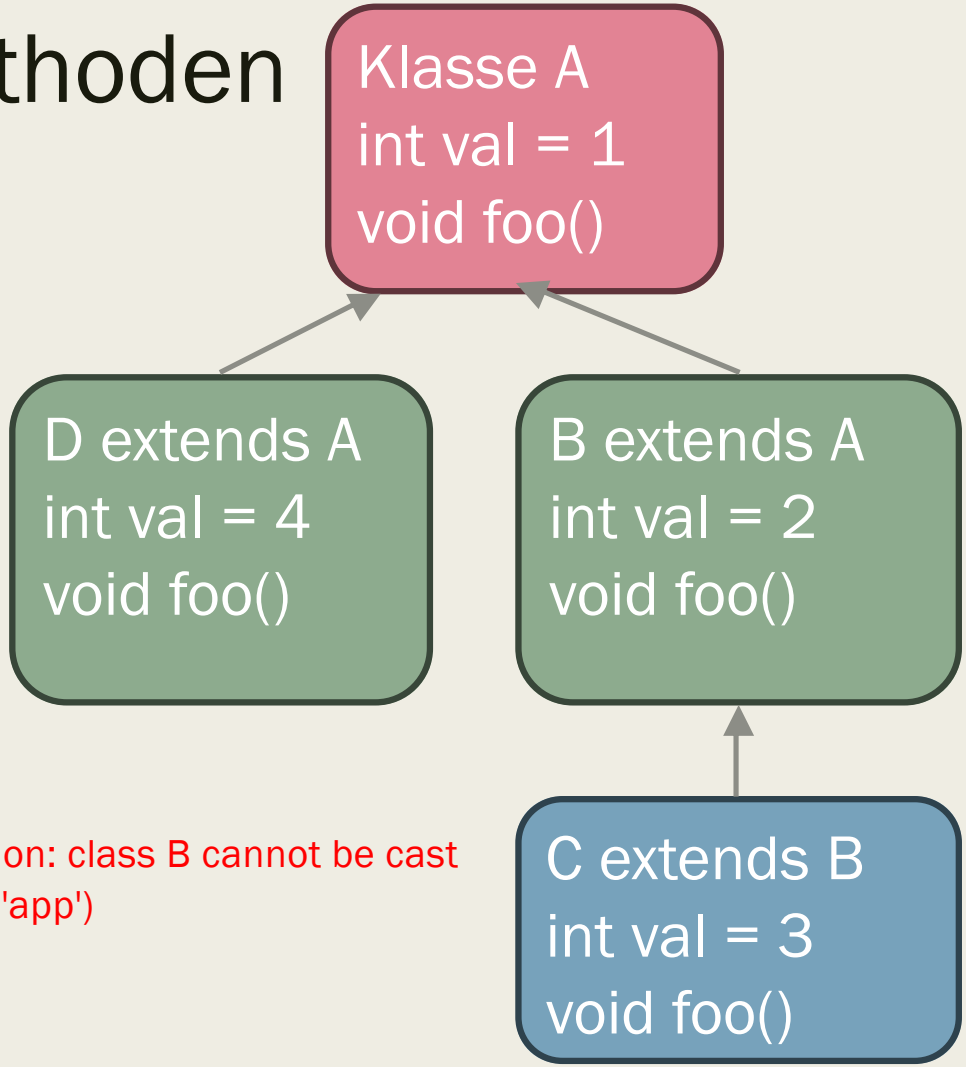


Polymorphismus - Methoden

```
public static void main(String[] args) {
    B b = new B();
    C bToC = (C)b;
    bToC.foo();
}
```

Output:

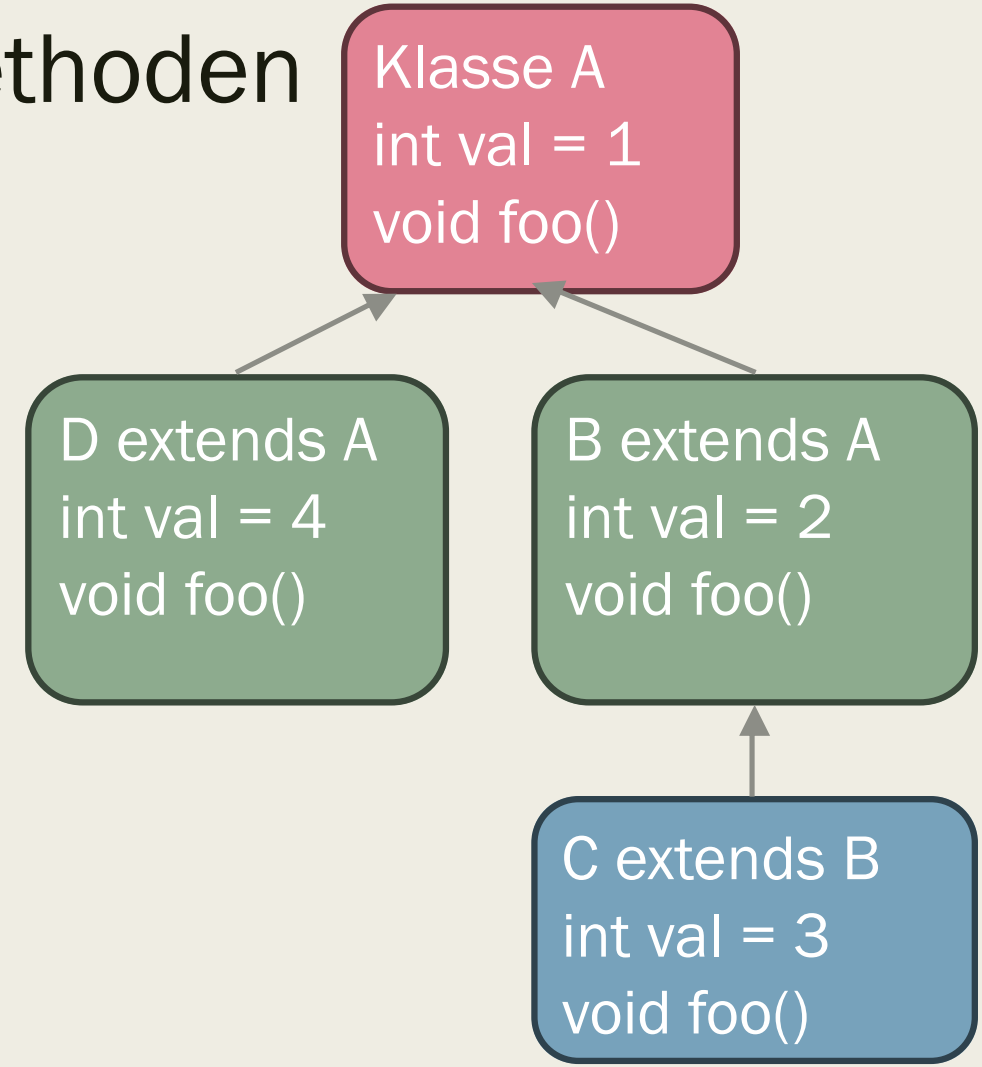
```
Exception in thread "main" java.lang.ClassCastException: class B cannot be cast
to class C (B and C are in unnamed module of loader 'app')
        at Main.main(Main.java:6)
```



Polymorphismus - Methoden

```
public static void main(String[] args) {
    B b = new C();
    b.foo();
}
```

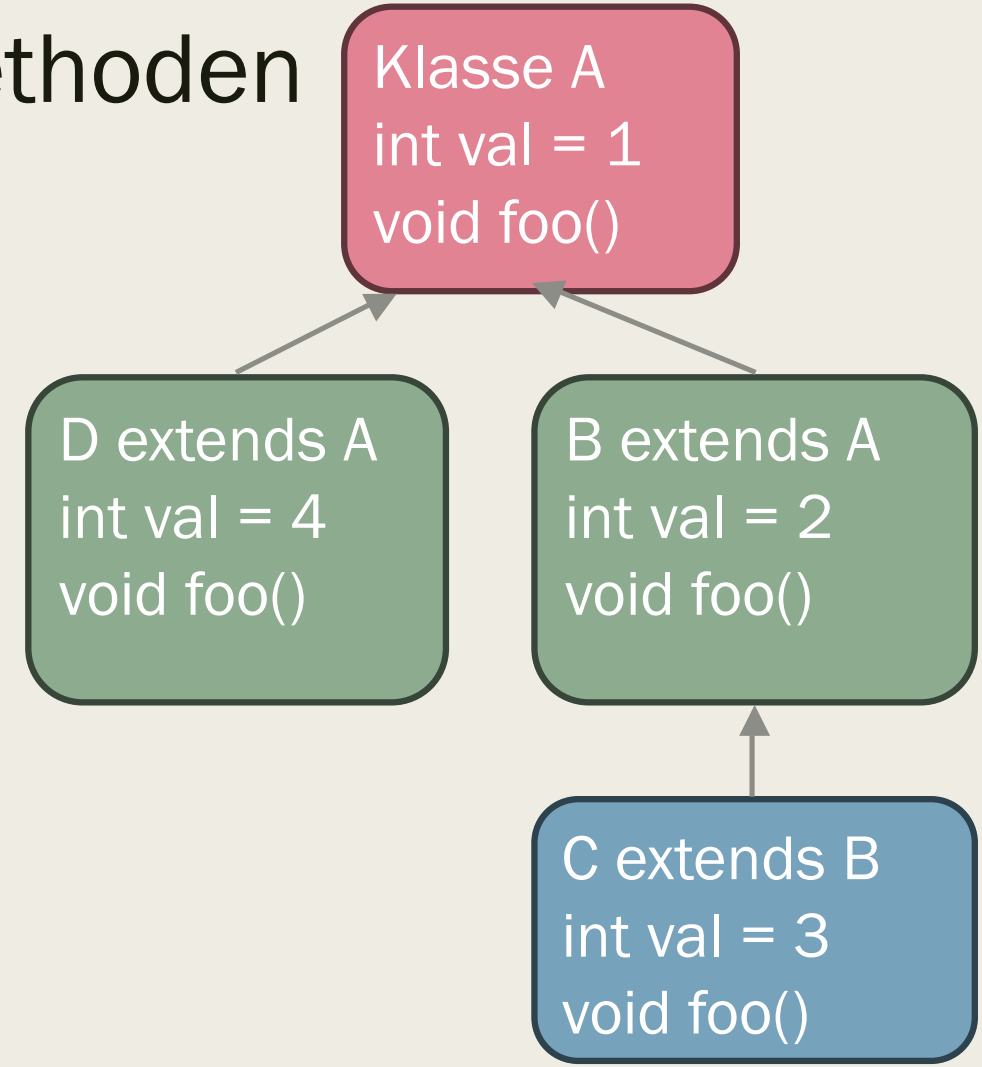
Output: C + val = 3



Polymorphismus - Methoden

```
public static void main(String[] args) {
    B b = new C();
    ((C)b).foo();
}
```

Output: C + val = 3



Polymorphismus - Methoden

“Compile-time” Polymorphism:

This includes any selection of methods and attributes selected during compilation, i.e. static binding or method overloading.

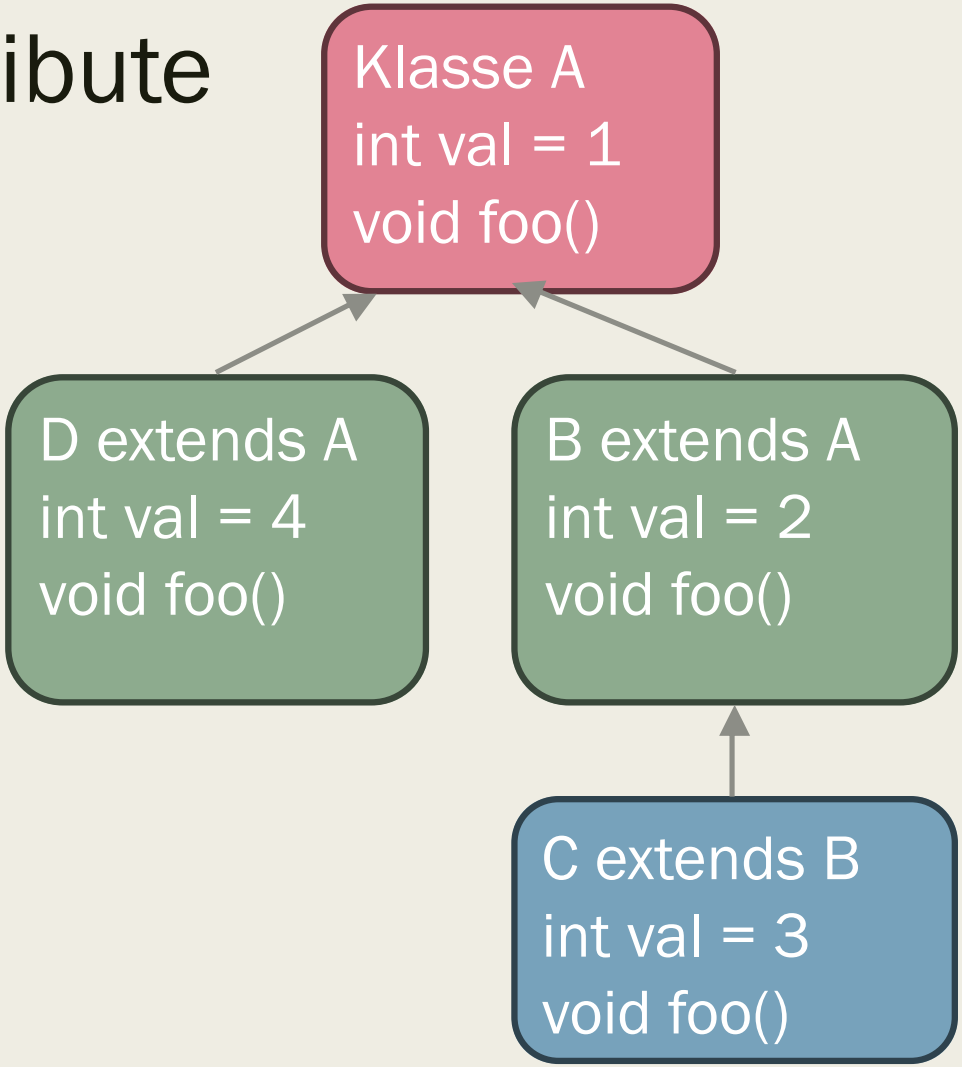
“Run-time” Polymorphism:

This includes any selection of methods and attributes during runtime, i.e. dynamic binding or method overriding.

Polymorphismus - Attribute

```
public static void main(String[] args) {
    B b = new B();
    System.out.println(b.val);
}
```

Output: 2

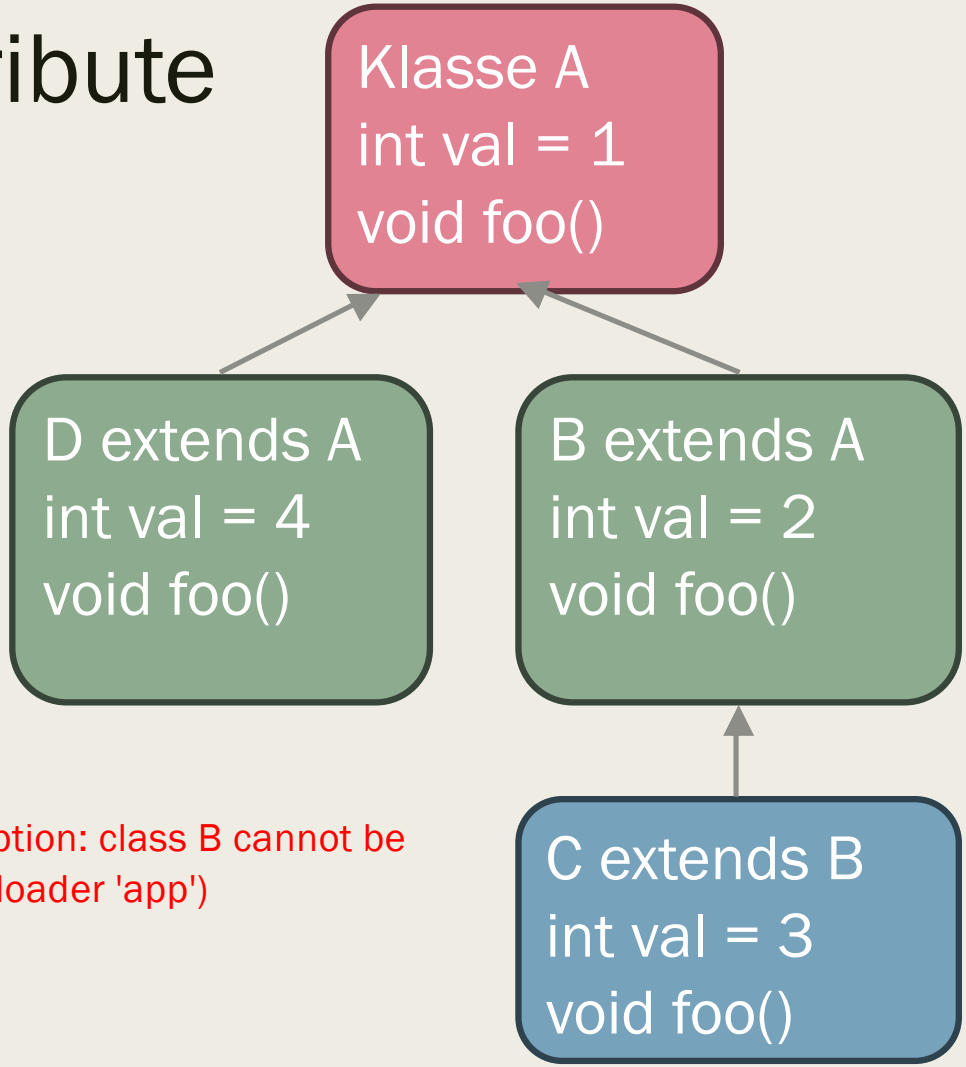


Polymorphismus - Attribute

```
public static void main(String[] args) {
    B b = new B();
    C bToC = (C)b;
    System.out.println(bToC.val);
}
```

Output:

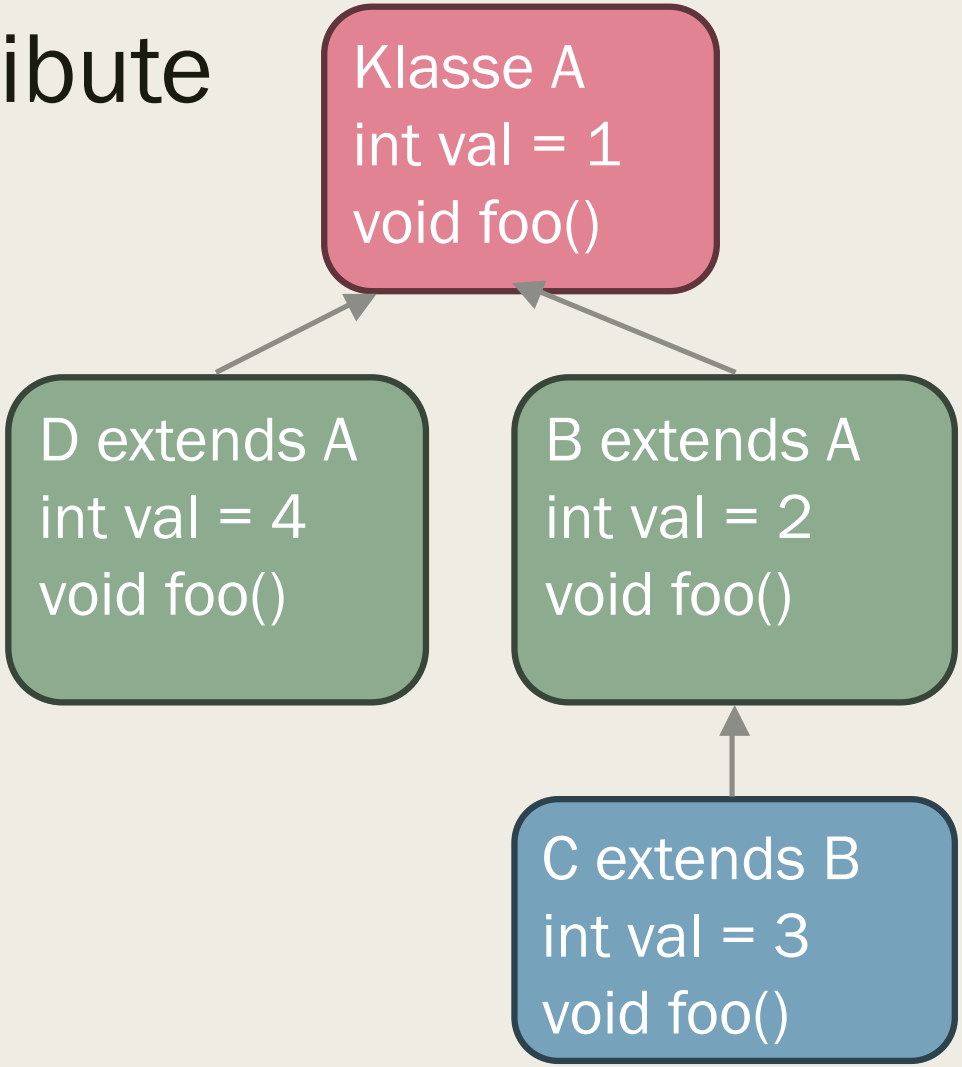
```
Exception in thread "main" java.lang.ClassCastException: class B cannot be
cast to class C (B and C are in unnamed module of loader 'app')
        at Main.main(Main.java:6)
```



Polymorphismus - Attribute

```
public static void main(String[] args) {
    B b = new B();
    A bToA = b;
    System.out.println(bToA.val);
}
```

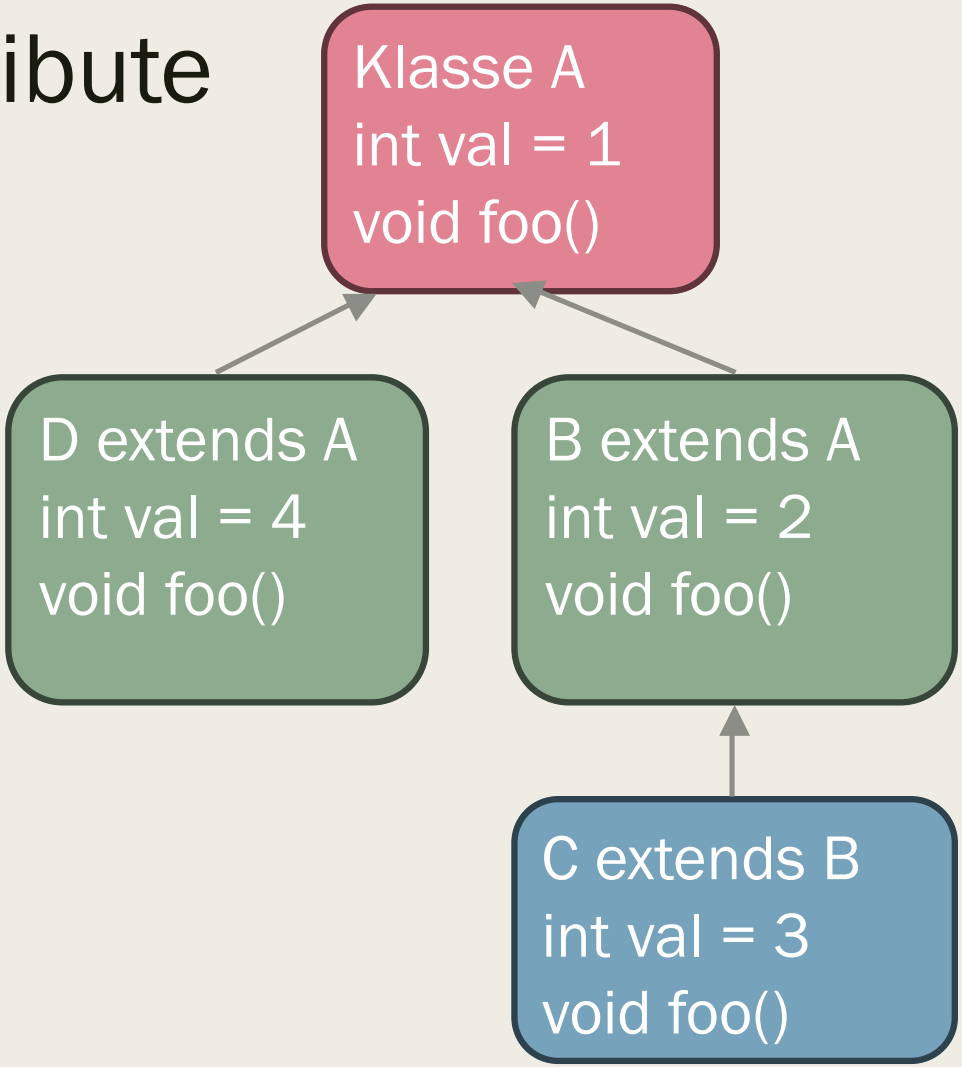
Output: 1



Polymorphismus - Attribute

```
public static void main(String[] args) {
    B b = new C();
    System.out.println(b.val);
}
```

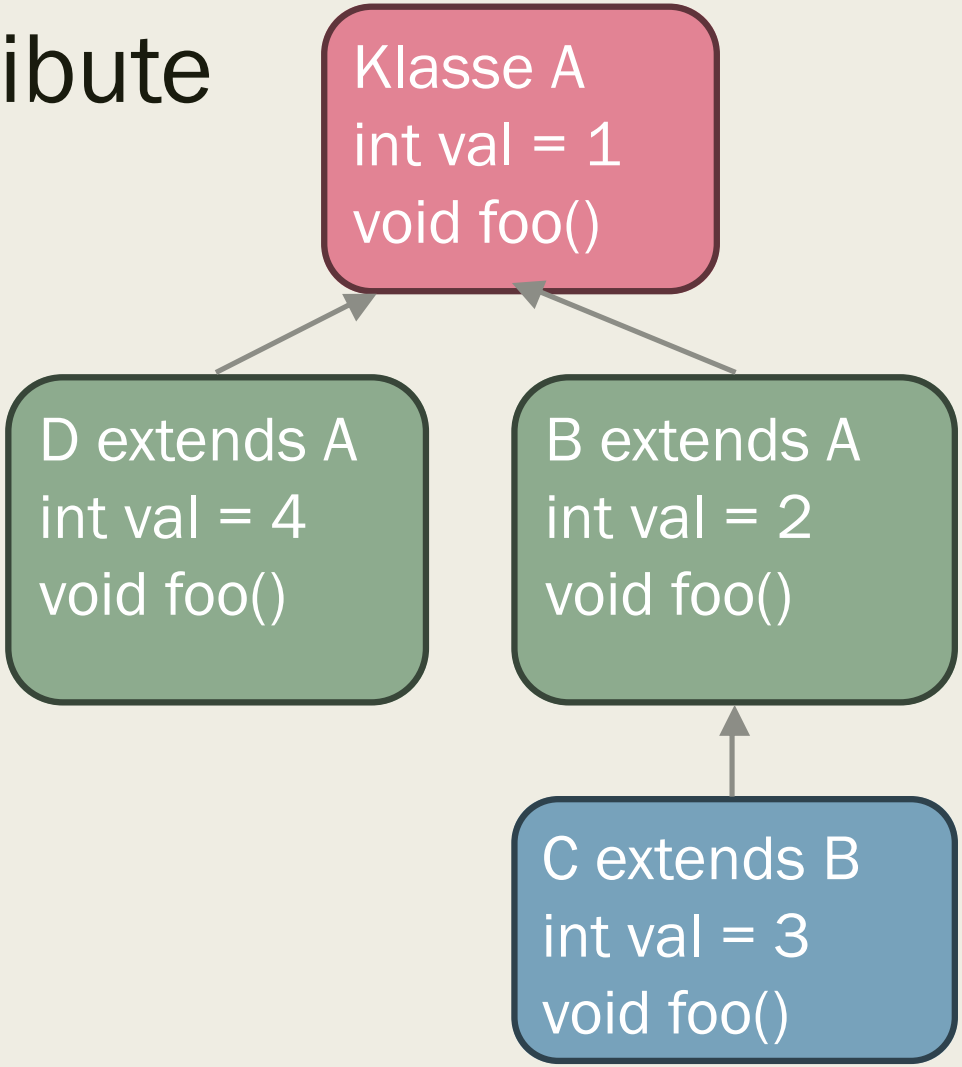
Output: 2



Polymorphismus - Attribute

```
public static void main(String[] args) {
    B b = new C();
    System.out.println(((A)b).val);
}
```

Output: 1



Polymorphismus - Attribute

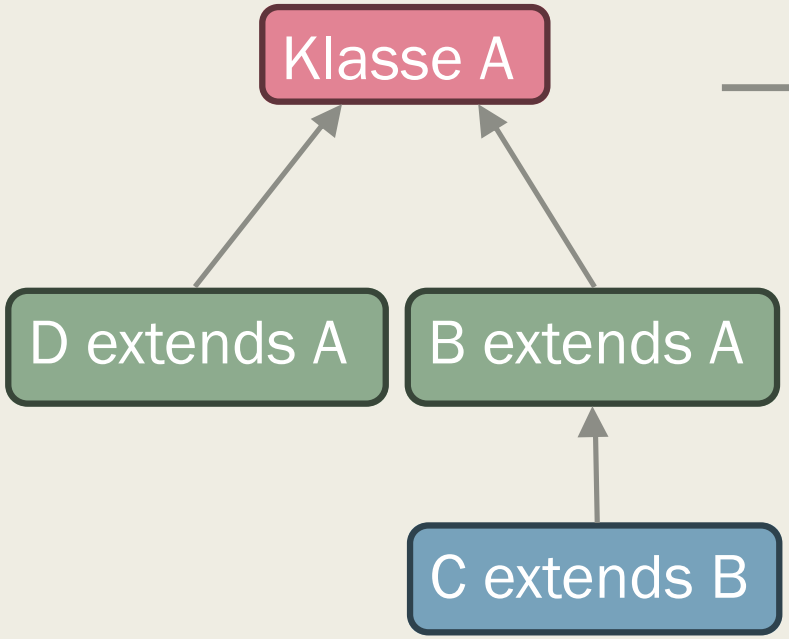
“Compile-time” Polymorphism:

This includes any selection of methods and attributes selected during compilation, i.e. static binding or method overloading.

“Run-time” Polymorphism:

This includes any selection of methods and attributes during runtime, i.e. dynamic binding or method overriding.

Casts



Exception

illegale Casts von Runtimetype

Beispiel:

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

Error

Quercasts

Beispiel:

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

PRÜFUNGSAUFGABE

Inheritance, FS22

Pause

Aaron: aazeller

Cédric: cdeteindre

KAHOOT