Erweitern und Festigen der erlernten Konzepte

03. Februar 2025

SMART INDUSTRY CAMPUS

Plan für die Woche

Montag	Dienstag	Mittwoch	Donnerstag	Freitag

• Wiederholung Encapsulation, Methods, Konstruktoren	• OOP-Konzepte:
	  • Abstrakte Klassen vs. Interfaces
	  • Vertiefung Interfaces
		• OOP-Konzepte:
		  • Vertiefung Vererbung und ~~Polymorphismus~~
		  • Super vs this
		  • Overwritten/Overloaded Methods
		  • ~~Casting~~
			• Zusammenfassung zu gestrigen Themen
			• Casting und Polymorphismus
			• Operatoren, Bedingungen, Ternary
			• Parentheses, precedence
			• switch
				• Arrays (1D, 2D)
				• Loops, loops, loops

SMART INDUSTRY CAMPUS

Plan für heute

- Polymorphismus
- instanceof
- casting

SMART INDUSTRY CAMPUS

Zusammenfassung

- Bei der Vererbung muss man auf die Access-Modifier achten

Visibility	Public	Protected	Package-Private	Private
Innerhalb der **gleichen Klasse**	OK	Ok	Ok	Ok
Innerhalb **desselben Packages**	Ok	Ok	Ok	X
In einer **Subklasse** innerhalb **desselben Packages**	Ok	Ok	Ok	X
Innerhalb einer **Subklasse außerhalb** desselben **Packages**	Ok	OK, mittels Vererbung. Auf Referenzvariable achten!	X	X
Innerhalb **keiner Subklasse** in einem **anderen Package**	OK	X	X	X

SMART INDUSTRY CAMPUS

Zusammenfassung

- this()und super()

✓ this(...) darf nur innerhalb einer Klasse verwendet werden und muss die erste Anweisung im Konstruktor sein.

✓ super(...) ruft den Konstruktor der Oberklasse auf und muss ebenfalls die erste Anweisung im Konstruktor sein.

✓ Wenn kein super() angegeben wird, ruft Java automatisch den Standard-Konstruktor (super()) der Oberklasse auf.

SMART INDUSTRY CAMPUS

Zusammenfassung

✅ this.

Verweist auf die aktuelle Instanz der Klasse.

Greift auf Variablen oder Methoden der eigenen Klasse zu.

Wird genutzt, um Namenskonflikte zu vermeiden (this.name = name;).

✅ super.

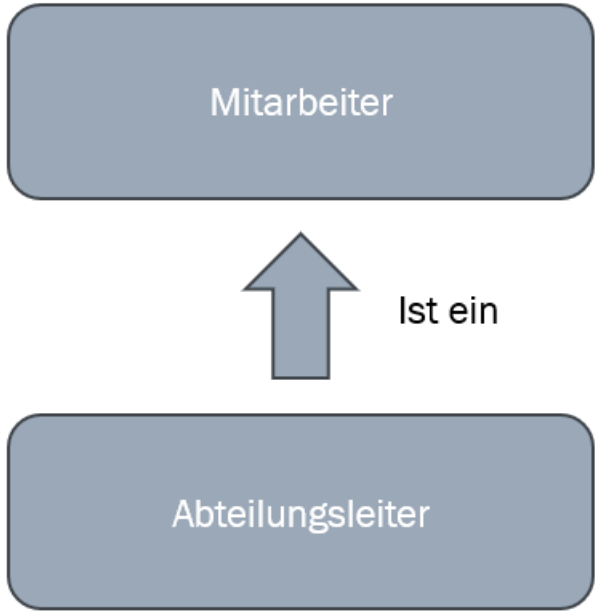
Verweist auf die Instanz der Oberklasse.

Greift auf Variablen oder Methoden der Oberklasse zu.

Wird verwendet, um überschriebene Methoden oder versteckte Variablen der Elternklasse aufzurufen.

Zusammenfassung

- Hierarchische Strukturen müssen beachtet werden!



Ein Abteilungsleiter ist ein Mitarbeiter

```
Mitarbeiter mitarbeiter = new Abteilungsleiter();
```

Ein Mitarbeiter ist nicht zwingend ein Abteilungsleiter!

```
Abteilungsleiter abteilungsleiter = new Mitarbeiter();
```

Compilerfehler

SMART INDUSTRY CAMPUS

Zusammenfassung

- Methoden können im Kontext von Vererbung auch überladen werden
  - Parameterliste muss angepasst werden!

- Methoden können im Kontext von Vererbung auch überschrieben werden
  - Methoden dürfen dafür nicht static / final sein
  - Methodenkopf muss komplett übereinstimmen
  - Access-Modifier dürfen nicht restriktiver sein
  - Exceptions haben Sonderregeln!

SMART INDUSTRY CAMPUS

Zusammenfassung

- Exceptions und überschriebene Methoden haben Regeln, die man sich merken muss

✅ 1. Eine überschreibende Methode darf keine neue, breitere Exception werfen.

✅ 2. Eine überschreibende Methode darf eine Unterklasse der Exception der Oberklasse werfen.

✅ 3. Eine überschreibende Methode muss keine Exception werfen, auch wenn die Oberklasse eine wirft.

✅ 4. Unchecked Excpetions dürfen immer geschmissen werden

SMART INDUSTRY CAMPUS

Grundlagen

- Sie ermöglicht es einer Klasse, die Eigenschaften und Methoden einer anderen Klasse zu erben

- Hauptzwecke/Benefits der Vererbung:

  Wiederverwendbarkeit des Codes ✓

  Hierarchische Struktur ✓

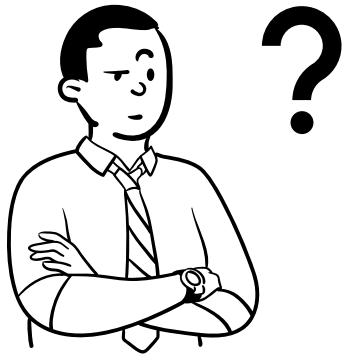
  Methodenüberschreibung ✓

  Polymorphismus

SMART INDUSTRY CAMPUS

Polymorphismus

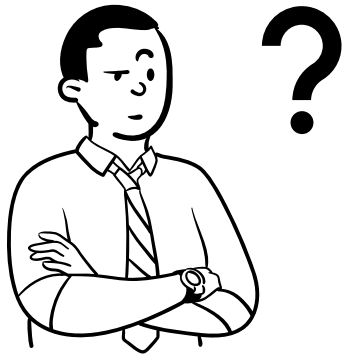
- Objekte verschiedener Klassen werden als Objekte einer gemeinsamen Superklasse behandelt
- wobei der spezifische Methodenaufruf zur Laufzeit basierend auf dem tatsächlichen Objekttyp erfolgt.



SMART INDUSTRY CAMPUS

Polymorphismus

- **Objekte verschiedener Klassen werden als Objekte einer gemeinsamen Superklasse behandelt**
- wobei der spezifische Methodenaufruf zur Laufzeit basierend auf dem tatsächlichen Objekttyp erfolgt.



SMART INDUSTRY CAMPUS

Methodensichtbarkeit!

- Ein Abteilungsleiter kann in eine Mitarbeiter-Referenz gesteckt werden

```
class Mitarbeiter {
    public void arbeitet(){
        System.out.println("Ich arbeite einfach");
    }
    public void faehrtBus(){
        System.out.println("Ich fahre Bus");
    }
}

class Abteilungsleiter extends Mitarbeiter {
    private Mitarbeiter[] untergebene;

    public void arbeitet(){
        System.out.println("Ich leite eine Abteilung");
    }
    public void faehrtLambo(){
        System.out.println("Ich fahre Lambo");
    }

    public static void main(String[] args) {
        Abteilungsleiter a = new Abteilungsleiter();
        a.faehrtLambo();
        a.faehrtBus();

        Mitarbeiter m = new Abteilungsleiter();
        m.faehrtBus();

    }
}
```

Nur Methoden vom Abteilungsleiter

Nur Methoden vom Mitarbeiter

Polymorphismus

- unsere Methode interagiereMitMitarbeiter hat einen Parameter, welcher ein Mitarbeiter-Objekte entgegennimmt

- d.h. alle Subklassen von Mitarbeiter können da eingesetzt werden und interagiereMitMitarbeiter() wird ausgeführt

-> **Prinzip von Polymorphismus**

```
class Mitarbeiter {
    public void arbeiten() {
        System.out.println("Ich arbeite.");
    }
}

class Abteilungsleiter extends Mitarbeiter {
    public void arbeiten() {
        System.out.println("Ich leite die Abteilung.");
    }
}

class Werkstudent extends Mitarbeiter {
    public void arbeiten() {
        System.out.println("Ich unterstütze das Team als Werkstudent.");
    }
}

public class TestMitarbeiter {
    public static void main(String[] args) {
        Abteilungsleiter chef = new Abteilungsleiter();
        Werkstudent student = new Werkstudent();

        interagiereMitMitarbeiter(chef);
        interagiereMitMitarbeiter(student);
    }

    public static void interagiereMitMitarbeiter(Mitarbeiter mitarbeiter) {
        mitarbeiter.arbeiten();
    }
}
```

Polymorphismus

- funktioniert auch bei Interfaces

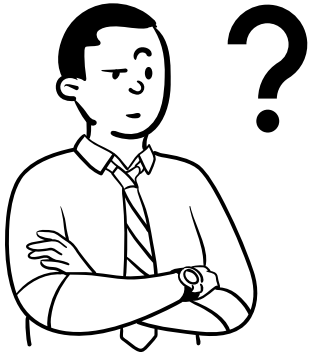
Mitarbeiter ist jetzt ein interface

Referenzvariable ist vom Typ: Mitarbeiter

```java
interface Mitarbeiter {
    void arbeiten();
}

class Abteilungsleiter implements Mitarbeiter {
    public void arbeiten() {
        System.out.println("Ich leite die Abteilung.");
    }
}

class Werkstudent implements Mitarbeiter {
    public void arbeiten() {
        System.out.println("Ich unterstütze das Team als Werkstudent.");
    }
}

public class TestMitarbeiter {
    public static void main(String[] args) {
        Mitarbeiter chef = new Abteilungsleiter();
        Mitarbeiter student = new Werkstudent();

        interagiereMitMitarbeiter(chef);
        interagiereMitMitarbeiter(student);
    }

    public static void interagiereMitMitarbeiter(Mitarbeiter mitarbeiter) {
        mitarbeiter.taetigkeit();
    }
}
```

Polymorphismus

- Objekte verschiedener Klassen werden als Objekte einer gemeinsamen Superklasse behandelt
- wobei der spezifische Methodenaufruf zur Laufzeit basierend auf dem tatsächlichen Objekttyp erfolgt.



Dynamic Method Dispatch

- Verteilung von Methoden zur Laufzeit

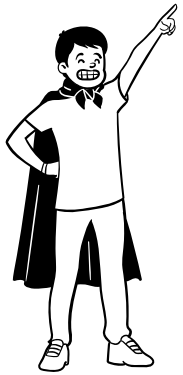
Objekt ist ein vom Typ Dog, deswegen wird die Ausgabe „Dog barks“ sein

```java
class Animal {
    public void makeSound() {
        System.out.println("Making sound");
    }
}


class Dog extends Animal {
    public void makeSound() {
        System.out.println("Dog barks");
    }
}

class Cat extends Animal {
    public void makeSound() {
        System.out.println("Cat meows");
    }
}

public class TestAnimals {
    public static void main(String[] args) {
        Animal dog = new Dog();
        Animal cat = new Cat();
        dog.makeSound();
        cat.makeSound();
    }

}
```

SMART INDUSTRY CAMPUS

Polymorphismus

- Objekte verschiedener Klassen werden als Objekte einer gemeinsamen Superklasse behandelt
- wobei der spezifische Methodenaufruf zur Laufzeit basierend auf dem tatsächlichen Objekttyp erfolgt.



Übung

- Erstelle ein interface Bewegbar mit der Methode bewegen()
- Erstelle die Klassen Auto, Fahrrad, Hund und Vogel, welche das Interface implementieren
- Überschreibe in jeder Klasse die Methode bewege()
- Übernehme folgende Main-Klasse:

```
public class Main {
    public static void main(String[] args) {

        Bewegbar auto = new Auto();
        Bewegbar fahrrad = new Fahrrad();
        Bewegbar hund = new Hund();
        Bewegbar vogel = new Vogel();


        Bewegbar[] bewegbarObjekte = {auto, fahrrad, hund, vogel};
        bewegeAlle(bewegbarObjekte);
    }

    public static void bewegeAlle(Bewegbar[] bewegbarObjekte) {
        for (Bewegbar obj : bewegbarObjekte) {
            obj.bewege();  // Polymorphismus: unterschiedliche Implementierungen je nach Typ
        }
    }
}
```

Casting

Unser Objekt ist doch ein Abteilungsleiter?!

```java
class Mitarbeiter {
    public void faehrtBus(){
        System.out.println("Ich fahre Bus");
    }
}

class Abteilungsleiter extends Mitarbeiter {

    public void faehrtLambo(){
        System.out.println("Ich fahre Lambo");
    }

    public static void main(String[] args) {
        Mitarbeiter a = new Abteilungsleiter();
        a.faehrtLambo(); //COMPILERFEHLER!
        a.faehrtBus();
    }
}
```

Casting

```
class Mitarbeiter {

   public void faehrtBus(){
      System.out.println("Ich fahre Bus");
   }
}

class Abteilungsleiter extends Mitarbeiter {

   public void faehrtLambo(){
      System.out.println("Ich fahre Lambo");
   }

   public static void main(String[] args) {
      Mitarbeiter a = new Abteilungsleiter();
      a.faehrtLambo();
      a.faehrtBus();                    //COMPILERFEHLER!

   }
}
```

Unser Objekt ist doch ein Abteilungsleiter?!

Compiler weiß das aber nicht, da er sich nach der Referenzvariable richtet!

Casting

- Casting notwendig!
- Es wird nicht das Objekt, sondern die Referenzvariable gecastet! Tatsächlicher Typ des Objektes bleibt beibehalten!

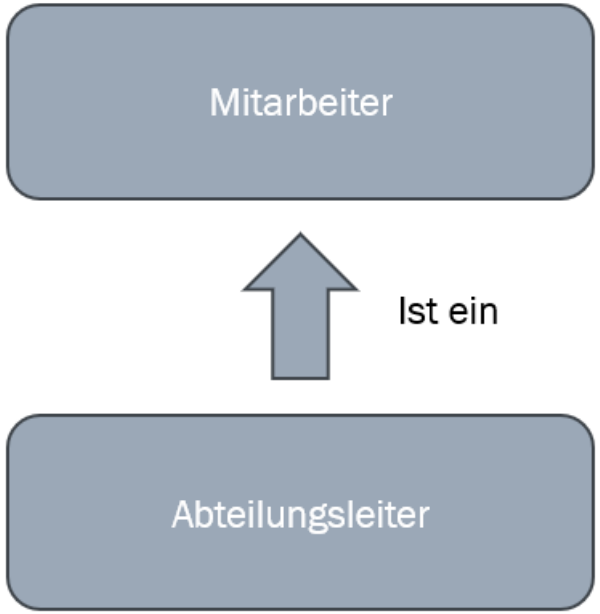
```
class Mitarbeiter {

   public void faehrtBus(){
      System.out.println("Ich fahre Bus");
   }
}

class Abteilungsleiter extends Mitarbeiter {

   public void faehrtLambo(){
      System.out.println("Ich fahre Lambo");
   }

   public static void main(String[] args) {
      Mitarbeiter a = new Abteilungsleiter();
      Abteilungsleiter a1 = (Abteilungsleiter) a;
      a1.faehrtLambo();
      a.faehrtBus();

   }
}
```

Expliziter Cast!

Upcasting (implizit)

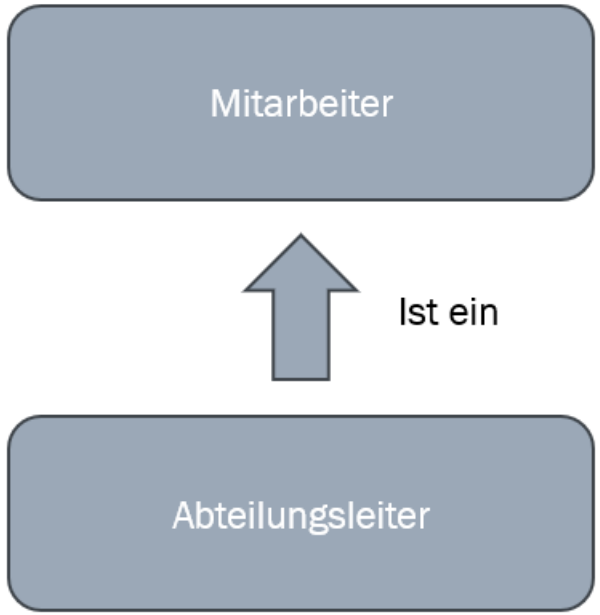
- eine Referenz der Unterklasse (Abteilungsleiter) wird in eine Referenz der Oberklasse (Mitarbeiter) umgewandelt
- geschieht implizit (automatisch), da Abteilungsleiter hierarchisch unter Mitarbeiter steht

```
Mitarbeiter a = new Abteilungsleiter();
```



Downcasting (explizit)

- eine Referenz der Oberklasse (Mitarbeiter) wird explizit in eine Referenz der Unterklasse (Abteilungsleiter) umgewandelt



Expliziter Cast!

```
Mitarbeiter a = new Abteilungsleiter();
Abteilungsleiter a1 = (Abteilungsleiter) a;
a1.faehrtLambo();
```

SMART INDUSTRY CAMPUS

Downcasting (explizit)

- Vorsicht ist geboten!
- ClassCastExceptions sind durch falsches Casten möglich!

```
Mitarbeiter a = new Mitarbeiter();
Abteilungsleiter a1 = (Abteilungsleiter) a;
```

Exception! Referenz a ist kein Abteilungleiter!

instanceof

- um eine ClassCastException zu umgehen
- überprüft, ob die Instanz (nicht die Referenz) vom Typ x ist

```
Mitarbeiter a = new Mitarbeiter();
if (a instanceof Abteilungsleiter) {
    Abteilungsleiter a1 = (Abteilungsleiter) a;
}
```

a ist ein Mitarbeiterobjekt

instanceof

- um eine ClassCastException zu umgehen
- überprüft, ob die Instanz (nicht die Referenz) vom Typ x ist

```
Mitarbeiter a = new Mitarbeiter();
if (a instanceof Abteilungsleiter) {
  Abteilungsleiter a1 = (Abteilungsleiter) a;
}
```

Cast wird nicht durchgeführt, da if-Bedingung false

SMART INDUSTRY CAMPUS

instanceof

- um eine ClassCastException zu umgehen
- überprüft, ob die Instanz (nicht die Referenz) vom Typ x ist

```
Mitarbeiter a = new Abteilungsleiter();
if (a instanceof Abteilungsleiter) {
    Abteilungsleiter a1 = (Abteilungsleiter) a;
}
```

a ist nun ein Abteilungsleiter. Was passiert folglich?

SMART INDUSTRY CAMPUS

Übung ca. 20 Minuten

- Erstelle eine Klasse Medium mit:
  - einer titel-Variable (String).
  - einem Konstruktor zur Initialisierung des Titels.
  - einer Methode zeigeInfo(), die "Medium: <Titel>" ausgibt.
- Buch erbt von Medium und hat zusätzlich:
  - eine autor-Variable (String).
  - eine eigene zeigeInfo()-Methode, die "Buch: <Titel> von <Autor>" ausgibt.
- Zeitschrift erbt von Medium und hat zusätzlich:
  - eine ausgabeNummer (int).
  - eine eigene zeigeInfo()-Methode, die "Zeitschrift: <Titel>, Ausgabe <Nummer>" ausgibt.
- Erstelle eine ArrayList<Medium> mit gemischten Medien (Buch und Zeitschrift).
  - Iteriere durch die Liste und prüfe mit instanceof, ob es sich um ein Buch handelt.
  - Falls ja, downcaste es und gib eine zusätzliche Nachricht "Dieses Buch wurde von <Autor> geschrieben." aus.
  - Rufe für jedes Medium die Methode zeigeInfo() auf.

Übung zu Cas

- Was ist der Output?

```
class Auto {
    public void printDetails() {
        System.out.println("Ich bin ein Auto");
    }
}

class Mercedes extends Auto {
    public void printDetails() {
        System.out.println("Ich bin ein Mercedes");
    }
}

class AKlasse extends Mercedes {
    public void printDetails() {
        System.out.println("Ich bin ein A-Klasse Mercedes");
    }

    public static void main(String[] args) {
        Auto b1 = new Mercedes();
        Auto b2 = new AKlasse();
        Auto b3 = new Mercedes();
        b1 = (Auto) b3;
        Auto b4 = (Mercedes) b3;
        b1.printDetails();
        b4.printDetails();
    }
}
```

- Es wird nicht das Objekt, sondern die Referenzvariable gecastet! Tatsächlicher Typ des Objektes bleibt beibehalten!

SMART INDUSTRY CAMPUS

Übung zu Cas

- Was ist der Output?

- Ich bin ein Mercedes
  Ich bin ein Mercedes

```
class Auto {
    public void printDetails() {
        System.out.println("Ich bin ein Auto");
    }
}

class Mercedes extends Auto {
    public void printDetails() {
        System.out.println("Ich bin ein Mercedes");
    }
}

class AKlasse extends Mercedes {
    public void printDetails() {
        System.out.println("Ich bin ein A-Klasse Mercedes");
    }

    public static void main(String[] args) {
        Auto b1 = new Mercedes();
        Auto b2 = new AKlasse();
        Auto b3 = new Mercedes();
        b1 = (Auto) b3;
        Auto b4 = (Mercedes) b3;
        b1.printDetails();
        b4.printDetails();
    }
}
```

- Es wird nicht das Objekt, sondern die Referenzvariable gecastet! Tatsächlicher Typ des Objektes bleibt beibehalten!

SMART INDUSTRY CAMPUS