



Erweitern und Festigen der erlernten Konzepte

03. Februar 2025

Plan für die Woche

Montag

- Wiederholung
Encapsulation,
Methods,
Konstruktoren

Dienstag

- **OOP-Konzepte:**
 - Abstrakte Klassen vs.
Interfaces
 - Vertiefung Interfaces

Mittwoch

- **OOP-Konzepte:**
 - Vertiefung Vererbung
und Polymorphismus
 - Casting
 - Super

Donnerstag

- Overwritten/Overloaded
Methods
- Casting
- Super

Freitag

- Operatoren und
Bedingungen
- Ternary
- Switch

Plan für heute

- Wiederholung abstrakte Klassen
- Vertiefung Interfaces
- Vergleich abstrakte Klassen vs. Interfaces

Konzepte der OOP

- OOP steht für objektorientierte Programmierung
- Konzepte:
 - Kapselung (encapsulation)
 - Abstraktion (abstraction)
 - Vererbung (inheritance)
 - Polymorphismus (polymorphism)

Konzepte der OOP

- OOP steht für objektorientierte Programmierung
- Konzepte:
 - Kapselung (encapsulation)
 - **Abstraktion** (abstraction)
 - Vererbung (inheritance)
 - Polymorphismus (polymorphism)

Wiederholung Abstrakte Klassen

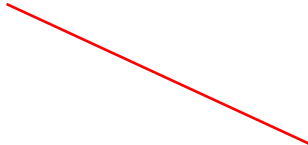
FESTIGEN UND ERWEITERN DES WISSENS

Was sind abstrakte Klassen?

- sind „normale“ Klassen, die jedoch nicht instanziiert werden können
 - Dienen also nicht zur Bildung von Objekten
 - Haben dennoch einen Konstruktor!
- dienen als Basis
 - Definieren, welche allgemeinen Eigenschaften ein Typ haben soll
 - Mit den Subklassen(Vererbung) kann diese Klasse um Eigenschaften erweitert werden

Syntax


Abstract - Keyword



```
abstract class Tier {  
  
}
```


Inhalt einer abstrakten Klasse

- können Datenfelder halten
- können einen Konstruktor oder mehrere Konstruktoren haben
- eine abstrakte Klasse kann sowohl **abstrakte** als auch **konkrete** Methoden beinhalten



Wenn kein Access-Modifier vor einem Datenfeld oder Methode steht, dann gilt **package-private**! (für Vererbung wichtig)

Inhalt einer abstrakten Klasse

—abstrakte Methode:

- Können nur public, protected oder package-private sein. Nicht private

Abstract - Keyword

```
abstract class Tier {  
    abstract void makeSound();  
}
```

Semikolon direkt
nach
Methodensignatur

Kein Methodenkörper!

Inhalt einer abstrakten Klasse

– konkrete Methoden:

Abstrakte
Klasse

Konkrete Methode
mit Methodenkörper

```
abstract class Tier {  
    void makeSound(){  
        System.out.println("Tier macht default-sound");  
    }  
}
```

Inhalt einer abstrakten Klasse

Abstrakte
Klasse

Sowohl
abstrakte als
auch konkrete
Methoden
akzeptabel!

```
abstract class Tier {  
    abstract void makeSound();  
    void move(){  
        System.out.println("Es rennt");  
    }  
}
```

Was sind abstrakte Klassen? - Zusammenfassung

- sind Klassen, die nicht instanziiert werden können
- dienen als Basis
- können Datenfelder haben
- können einen Konstruktor oder mehrere Konstruktoren haben
- können abstrakte und konkrete Methoden haben
 - Sobald eine Methode innerhalb einer Klasse abstrakt ist, muss die Klasse selbst als abstrakt markiert werden!

Ist der Code valide?

```
abstract class Tier {  
    void schlafen() {  
        System.out.println("Das Tier schläft.");  
    }  
}
```

- sind Klassen, die nicht instanziiert werden können
- dienen als Basis
- können Datenfelder haben
- können einen Konstruktor oder mehrere Konstruktoren haben
- können abstrakte und konkrete Methoden haben
 - Sobald eine Methode innerhalb einer Klasse abstrakt ist, muss die Klasse selbst als abstrakt markiert werden!



Ist der Code valide?

```
abstract class Tier {  
    void schlafen() {  
        System.out.println("Das Tier schläft.");  
    }  
}
```

Ja!

- sind Klassen, die nicht instanziiert werden können
- dienen als Basis
- können Datenfelder haben
- können einen Konstruktor oder mehrere Konstruktoren haben
- können abstrakte und konkrete Methoden haben
 - Sobald eine Methode innerhalb einer Klasse abstrakt ist, muss die Klasse selbst als abstrakt markiert werden!



Ist der Code valide?

```
class Fahrzeug {  
    abstract void fahren();  
}
```

- sind Klassen, die nicht instanziiert werden können
- dienen als Basis
- können Datenfelder haben
- können einen Konstruktor oder mehrere Konstruktoren haben
- können abstrakte und konkrete Methoden haben
 - Sobald eine Methode innerhalb einer Klasse abstrakt ist, muss die Klasse selbst als abstrakt markiert werden!



Ist der Code valide?

```
class Fahrzeug {  
    abstract void fahren();  
}
```

- sind Klassen, die nicht instanziiert werden können
- dienen als Basis
- können Datenfelder haben
- können einen Konstruktor oder mehrere Konstruktoren haben
- können abstrakte und konkrete Methoden haben
 - Sobald eine Methode innerhalb einer Klasse abstrakt ist, muss die Klasse selbst als abstrakt markiert werden!

Nein! Sobald eine abstrakte Methode in einer Klasse auftritt, muss die Klasse selbst als abstrakt markiert werden!



Ist der Code valide?

```
abstract class Lebewesen {  
    void atmen() {  
        System.out.println("Atmet Luft.");  
    }  
}
```

- sind Klassen, die nicht instanziiert werden können
- dienen als Basis
- können Datenfelder haben
- können einen Konstruktor oder mehrere Konstruktoren haben
- können abstrakte und konkrete Methoden haben
 - Sobald eine Methode innerhalb einer Klasse abstrakt ist, muss die Klasse selbst als abstrakt markiert werden!



Ist der Code valide?

```
abstract class Lebewesen {  
    void atmen() {  
        System.out.println("Atmet Luft.");  
    }  
}
```

- sind Klassen, die nicht instanziiert werden können
- dienen als Basis
- können Datenfelder haben
- können einen Konstruktor oder mehrere Konstruktoren haben
- können abstrakte und konkrete Methoden haben
 - Sobald eine Methode innerhalb einer Klasse abstrakt ist, muss die Klasse selbst als abstrakt markiert werden!

Passt! Abstrakte Klassen können auch konkrete Methoden haben



Ist der Code valide?

```
abstract class Pflanze {  
    abstract void wachsen() {  
        System.out.println("Die Pflanze wächst.");  
    }  
}
```

- sind Klassen, die nicht instanziiert werden können
- dienen als Basis
- können Datenfelder haben
- können einen Konstruktor oder mehrere Konstruktoren haben
- können abstrakte und konkrete Methoden haben
 - Sobald eine Methode innerhalb einer Klasse abstrakt ist, muss die Klasse selbst als abstrakt markiert werden!



Ist der Code valide?

```
abstract class Pflanze {  
    abstract void wachsen() {  
        System.out.println("Die Pflanze wächst.");  
    }  
}
```

- sind Klassen, die nicht instanziiert werden können
- dienen als Basis
- können Datenfelder haben
- können einen Konstruktor oder mehrere Konstruktoren haben
- können abstrakte und konkrete Methoden haben
 - Sobald eine Methode innerhalb einer Klasse abstrakt ist, muss die Klasse selbst als abstrakt markiert werden!

Abstrakte Methoden dürfen keinen Methodenkörper haben!



Ist der Code valide?

```
abstract class Werkzeug { }  
public class Main {  
    public static void main(String[] args) {  
        Werkzeug w = new Werkzeug();  
    }  
}
```

- sind Klassen, die nicht instanziiert werden können
- dienen als Basis
- können Datenfelder haben
- können einen Konstruktor oder mehrere Konstruktoren haben
- können abstrakte und konkrete Methoden haben
 - Sobald eine Methode innerhalb einer Klasse abstrakt ist, muss die Klasse selbst als abstrakt markiert werden!



Ist der Code valide?

```
abstract class Werkzeug { }  
public class Main {  
    public static void main(String[] args) {  
        Werkzeug w = new Werkzeug(); // FEHLER!  
    }  
}
```

- sind Klassen, die nicht instanziiert werden können
- dienen als Basis
- können Datenfelder haben
- können einen Konstruktor oder mehrere Konstruktoren haben
- können abstrakte und konkrete Methoden haben
 - Sobald eine Methode innerhalb einer Klasse abstrakt ist, muss die Klasse selbst als abstrakt markiert werden!

Abstrakte Klassen können nicht instanziiert werden!



Können abstrakte Klassen final sein?

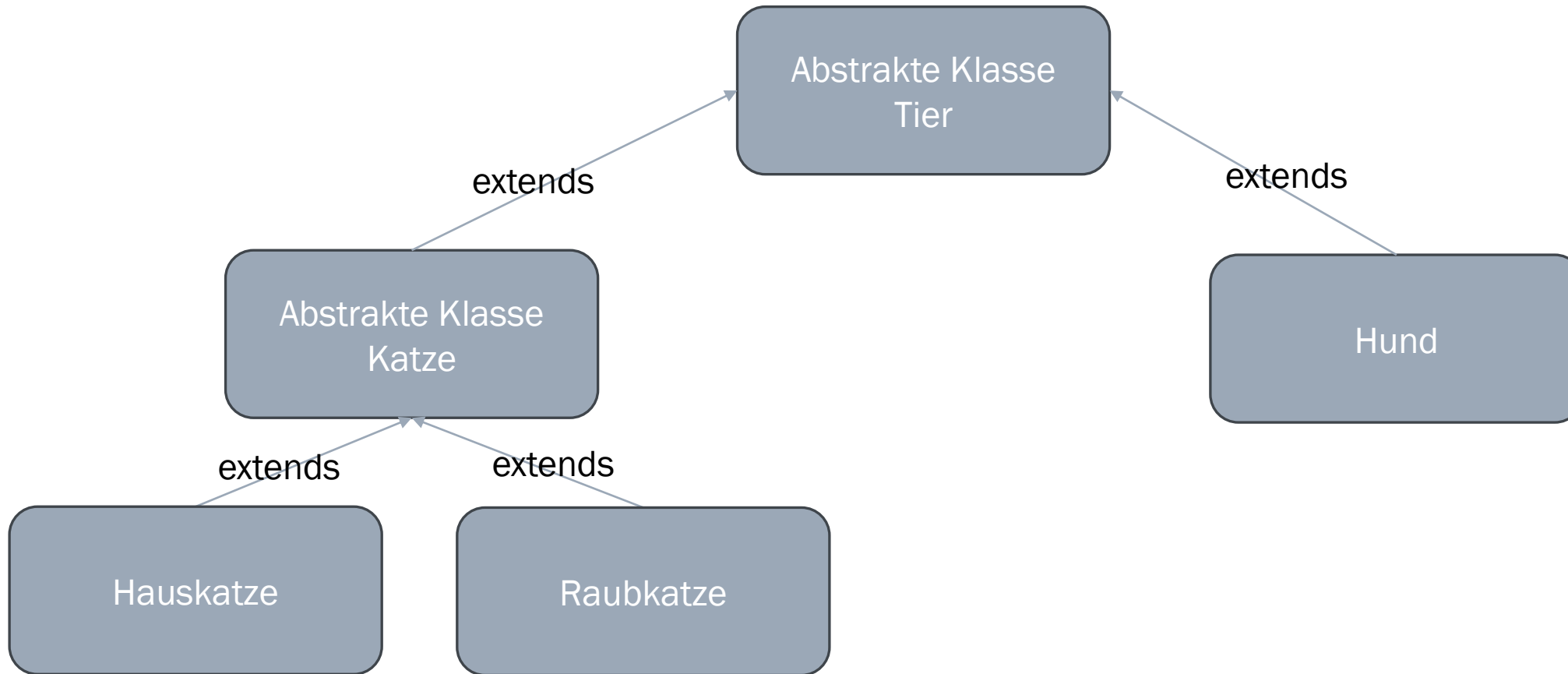
- Nein! – final und abstrakt sind nämlich Gegensätze
- final beschreibt, dass eine Klasse in ihrer Implementierung nicht mehr geändert werden kann
- abstrakt beschreibt, dass eine Klasse erweitert, also geändert werden kann



Warum abstrakte Klassen?

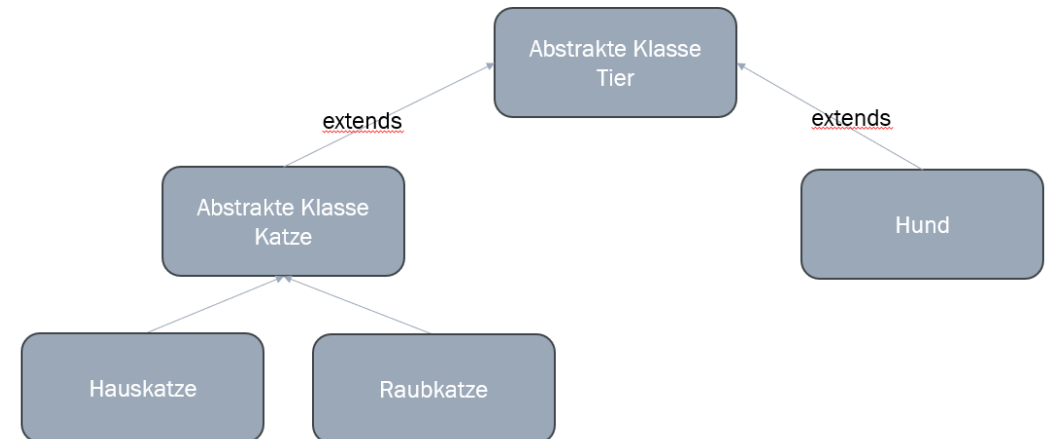
- ermöglichen Code-Wiederverwendung
- definieren eine allgemeine Struktur
- verhindern die Instanziierung von Klassen, die nur als Basis dienen sollen
- Wie nutzen ohne Instanziierung/Objekterstellung? Wie kann man Code wiederverwenden?
 - Mittels Vererbung

Kurzer Einschub: Abstrakte Klassen und Vererbung



Kurzer Einschub: Abstrakte Klassen und Vererbung

Codebeispiel in der IDE



Abstrakte Klassen und Vererbung - Zusammenfassung

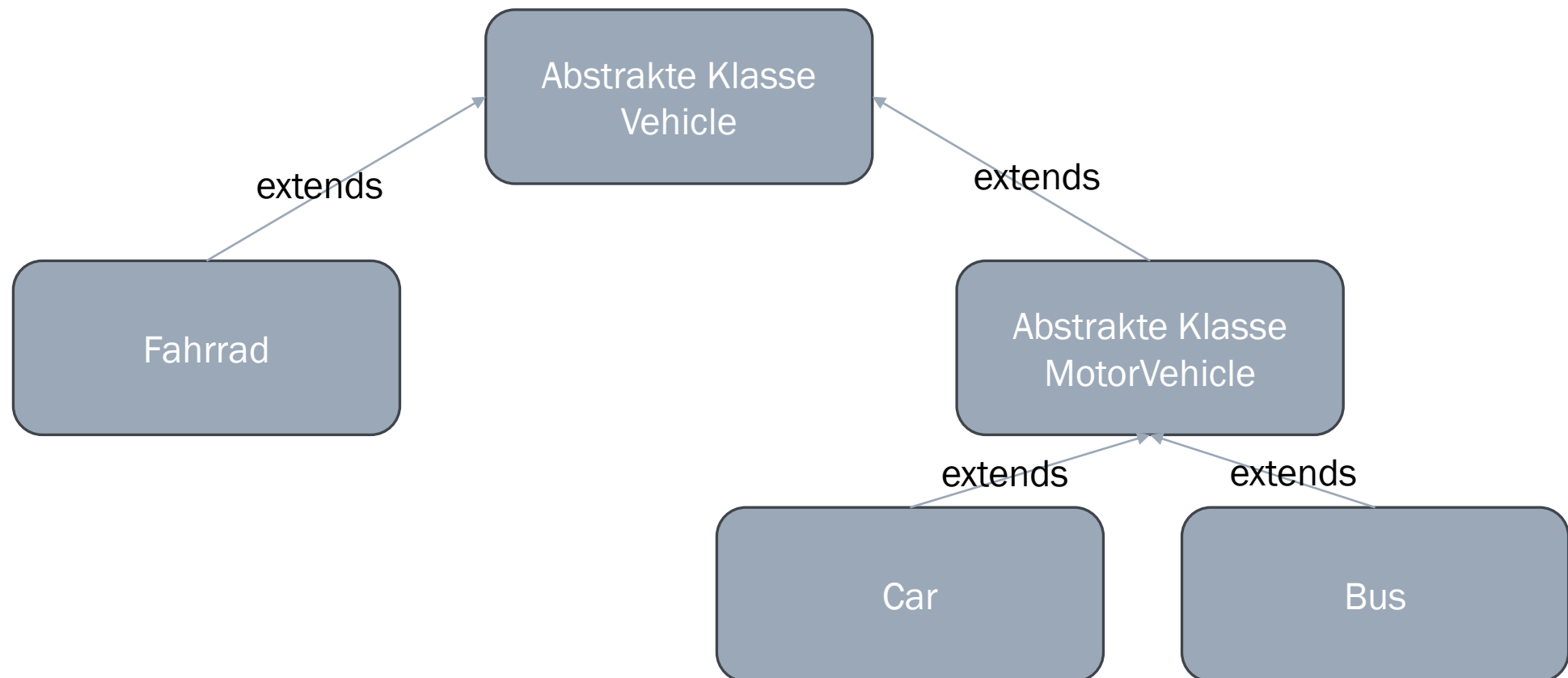
- Ist eine Unterklasse abstrakt, müssen die Methoden der Basisklasse nicht direkt implementiert werden
- Die erste konkrete Unterklasse muss alle noch nicht implementierten Methoden implementieren
- die Unterklasse kann eigene Methoden hinzufügen

Warum einen Konstruktor in abstrakten Klassen?

- zur Initialisierung von Variablen

```
abstract class Fahrzeug {  
    String hersteller;  
  
    public Fahrzeug(String hersteller) {  
        this.hersteller = hersteller;  
    }  
  
    abstract void fahren();  
}  
  
class Auto extends Fahrzeug {  
    public Auto(String hersteller) {  
        super(hersteller);  
    }  
  
    void fahren() {  
        System.out.println(hersteller + " Auto fährt.");  
    }  
}
```

Weiteres Beispiel zum experimentieren



Vertiefung Interfaces

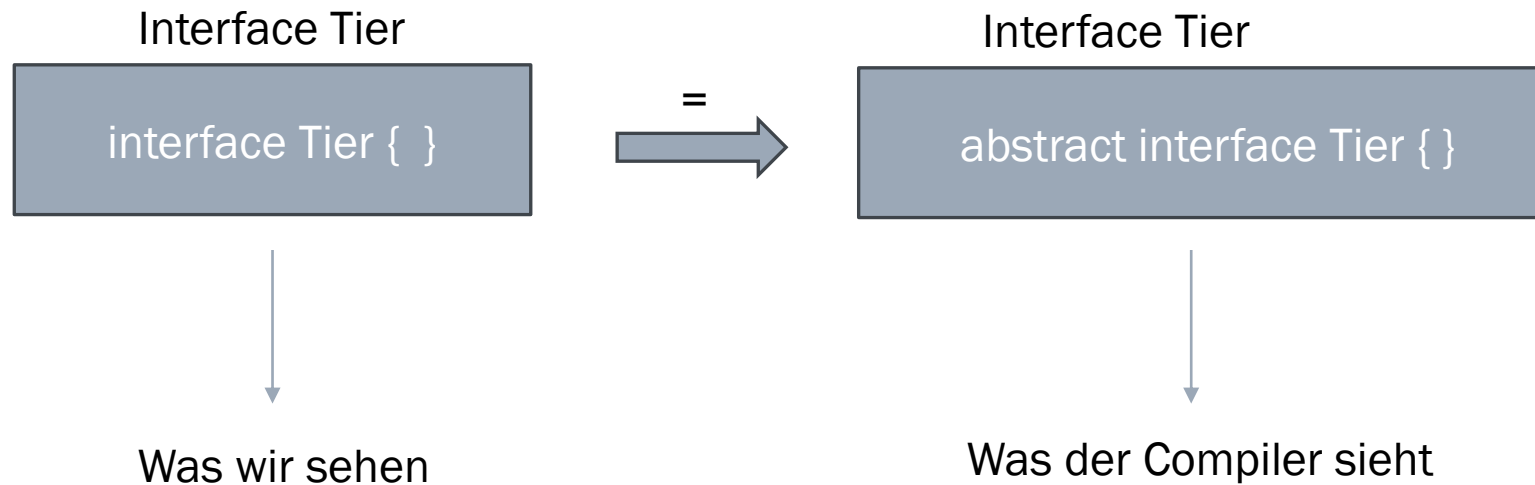
FESTIGEN UND ERWEITERN DES WISSENS

Grundlagen

- ein Interface sagt, **WAS** eine Klasse machen soll, nicht **WIE**
- Interfaces sind 100% abstrakt

Syntax der Interface-Deklaration

– Klassendefinition:



Inhalt eines Interfaces

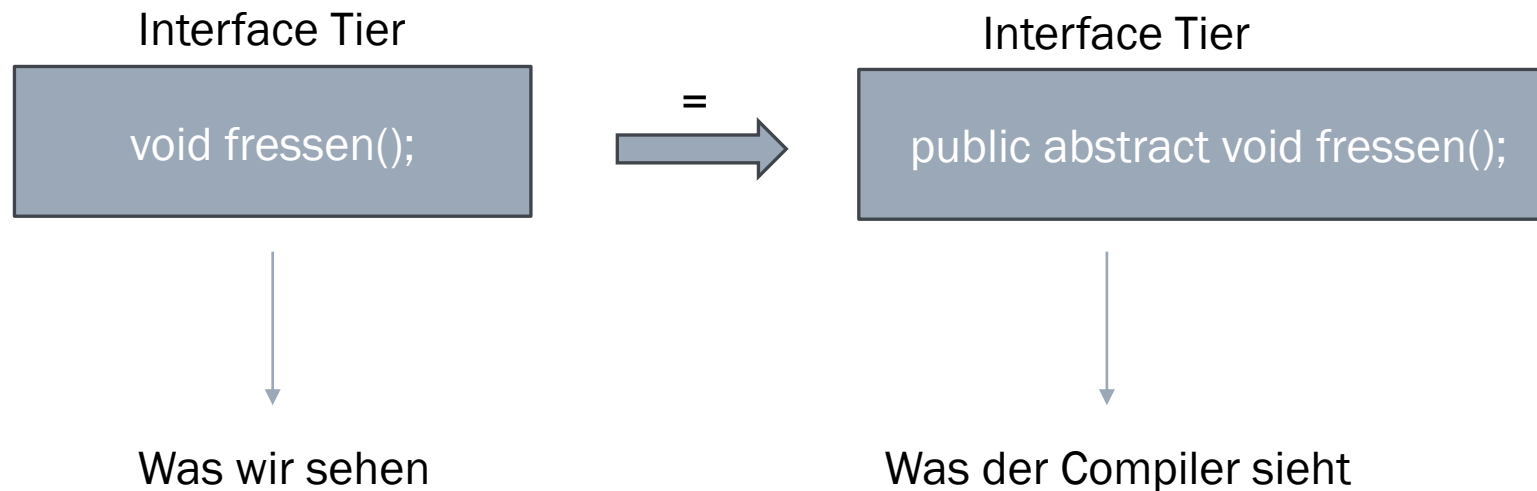
- können NUR Konstanten definieren
 - Der Compiler setzt automatisch `public`, `static`, `final` vor die Konstante
- es gibt keinen Konstruktor!
 - Beim Versuch kommt ein Compilerfehler
- es gibt abstrakte, default – und static- Methoden

Abstrakte Methoden

- es können abstrakte Methoden erzeugt werden
 - müssen IMMER public sein
- können nicht final sein -> sollen implementiert werden
- Methodendefinition:



Man muss deswegen
weder public noch
abstract davor schreiben



Abstrakte Methoden

- da die Methoden abstrakt sind, brauchen sie keinen Methodenkörper

```
public abstract void test();
```



```
public abstract void test(){  
    System.out.println(„test“);  
}
```



COMPILERFEHLER!

default und statische Methoden

- seit Java 8
- sowohl *default* und *static* – Methoden müssen einen Methodenkörper haben
- *default*- oder *static*-Methoden
 - dürfen **nicht** abstract sein -> Gegensätze!
 - dürfen **nicht** final sein -> sollen verändert werden!
 - Dürfen **NUR** public sein -> sollen auch in den implementierenden Klassen zugreifbar sein
 - Können **nicht** *default* und *static* gleichzeitig sein
 - Default: Gehört der Instanz
 - Static: Gehört dem Interface

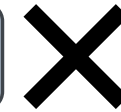
Methoden

Methoden	Public	Protected	Package-private	private
abstract	ok	x	x	x
Default	ok	x	x	x
static	ok	x	x	x

- ➔ Alle Methoden dürfen nicht final sein
- ➔ Default und static nicht kombinierbar
- ➔ Abstract und default/static nicht kombinierbar


Methoden

```
public default void test();
```



COMPILERFEHLER!

```
public default void test(){  
    System.out.println(„test“);  
}
```



Das default-
Keyword gibt es
nur im Kontext
von Interfaces

Das gleiche gilt auch für static!

Methoden

```
interface Fliegen{    //ANLEITUNG: WIE FLIEGE ICH?

    default void fluegelSchlag(){ //Ich muss mit den Flügeln schlagen, um zu fliegen
        //GEHÖRT DER INSTANZ
        System.out.println("Zum Fliegen muss man mit den Flügeln schlagen");
    }

    public static void aerodynamikRegeln(){    //eine Regel fürs Fliegen
        //hat aber nichts EXPLIZIT mit dem Vogel zu tun
        //GEHÖRT DEM INTERFACE
        System.out.println("Flugobjekte müssen den Luftwiderstand minimieren");
    }
}

class Vogel implements Fliegen{

    public static void main(String[] args) {
        Vogel v = new Vogel();
        Fliegen.aerodynamikRegeln(); //kann nicht über ein Objekt aufgerufen werden
        v.fluegelSchlag();
    }
}
```


Methoden - Aufgabe

In einem Musikstudio gibt es verschiedene Instrumente und Lautsprecher, die Klänge erzeugen können.

Erstelle ein Interface `Klanggeber`, das folgendes enthält:

Eine default-Methode `spieleTon()`, die einen Standard-Klang ausgibt: "Standardklang wird abgespielt...".

Eine static-Methode `info()`, die ausgibt: "Alle Klanggeber erzeugen Töne.".

Eine abstrakte Methode `erzeugeKlang()`, die jede Klasse individuell umsetzen muss.

Erstelle eine Klasse `Gitarre`, die `Klanggeber` implementiert:

Überschreibe die Methode `erzeugeKlang()`, sodass "Gitarrenklang: Strumm!" ausgegeben wird.

Erstelle eine Klasse `Lautsprecher`, die `Klanggeber` implementiert:

Überschreibe die Methode `erzeugeKlang()`, sodass "Bass: Wumm Wumm!" ausgegeben wird.

Teste deine Methoden!



Inhalt eines Interfaces – Zusammenfassung

- können NUR Konstanten definieren
 - Der Compiler setzt automatisch public, static, final vor die Konstante
- es gibt keinen Konstruktor!
 - Beim Versuch kommt ein Compilerfehler
- abstrakte Methoden haben keinen Methodenkörper
 - Sind per-default public abstract
- static und default- Methoden müssen einen Methodenkörper haben

Ist folgendes valider Code?

```
interface ABC{  
    void fressen(){  
    }  
}
```

- können NUR Konstanten definieren
 - Der Compiler setzt automatisch public, static, final vor die Konstante
- es gibt keinen Konstruktor!
 - Beim Versuch kommt ein Compilerfehler
- abstrakte Methoden haben keinen Methodenkörper
 - Sind per-default public abstract
- static und default- Methoden müssen einen Methodenkörper haben

Nein! Eine abstrakte Methode darf keinen Methodenkörper haben.



Ist folgendes valider Code?

```
class ABC{  
    void fressen(){  
    }  
}
```

Ist valider Code, ist aber kein Interface!

- können NUR Konstanten definieren
 - Der Compiler setzt automatisch public, static, final vor die Konstante
- es gibt keinen Konstruktor!
 - Beim Versuch kommt ein Compilerfehler
- abstrakte Methoden haben keinen Methodenkörper
 - Sind per-default public abstract
- static und default- Methoden müssen einen Methodenkörper haben



Ist folgendes valider Code?

```
interface ABC {  
    ABC() {  
    }  
  
    default void fressen() {  
    }  
}
```

- können NUR Konstanten definieren
 - Der Compiler setzt automatisch public, static, final vor die Konstante
- es gibt keinen Konstruktor!
 - Beim Versuch kommt ein Compilerfehler
- abstrakte Methoden haben keinen Methodenkörper
 - Sind per-default public abstract
- static und default- Methoden müssen einen Methodenkörper haben

Ein Interface darf keinen Konstruktor haben!



Ist folgendes valider Code?

```
interface ABC {  
    public int x = 1;  
}
```

- können NUR Konstanten definieren
 - Der Compiler setzt automatisch public, static, final vor die Konstante
- es gibt keinen Konstruktor!
 - Beim Versuch kommt ein Compilerfehler
- abstrakte Methoden haben keinen Methodenkörper
 - Sind per-default public abstract
- static und default- Methoden müssen einen Methodenkörper haben

Ja! Der Compiler setzt ein public, static und final davor.



Ist folgendes valider Code?

```
interface ABC {  
    public int x = 1;  
}
```

```
interface ABC {  
    int x = 1;  
}
```

```
interface ABC {  
    static int x = 1;  
}
```

```
interface ABC {  
    final int x = 1;  
}
```

```
interface ABC {  
    public final int x = 1;  
}
```

```
interface ABC {  
    public static int x = 1;  
}
```

... alles valide

Ja! Der Compiler setzt ein public, static und final davor.



Ist folgendes valider Code?

```
interface ABC{  
    default int m1(){return 1;}    //n1  
    public default void m2() {}    //n2  
    static default void m3(){};    //n3  
    default void m4();             //n4  
}
```

- können NUR Konstanten definieren
 - Der Compiler setzt automatisch public, static, final vor die Konstante
- es gibt keinen Konstruktor!
 - Beim Versuch kommt ein Compilerfehler
- abstrakte Methoden haben keinen Methodenkörper
 - Sind per-default public abstract
- static und default- Methoden müssen einen Methodenkörper haben

Zeile n3 und n4 funktionieren nicht!

n3 – static und default geht nicht zusammen

n4 – default müssen einen Methodenkörper haben



Interface – Implementierung

- Können von **konkreten** und **abstrakten** Klassen implementiert werden
 - Schlüsselwort: **implements**
- Eine Klasse kann eine oder mehrere Interfaces implementieren

Interface – Implementierung

- Können von **konkreten** und **abstrakten** Klassen implementiert werden
 - Schlüsselwort: **implements**
- Eine Klasse kann eine oder mehrere Interfaces implementieren

```
interface Fliegen {  
    void fliegen();  
}  
  
interface Schwimmen {  
    void schwimmen();  
}  
  
class Ente implements Fliegen, Schwimmen {  
    public void fliegen() {  
        System.out.println("Die Ente fliegt.");  
    }  
    public void schwimmen() {  
        System.out.println("Die Ente schwimmt.");  
    }  
}
```

Interface – Implementierung

- Können von **konkreten** und **abstrakten** Klassen implementiert werden
 - Schlüsselwort: **implements**
- Eine Klasse kann eine oder mehrere Interfaces implementieren
- Ein Interface kann keine anderen Interfaces implementieren

```
interface Fliegen {  
    void fliegen();  
}  
  
interface Schwimmen implements Fliegen { //FEHLER  
    void schwimmen();  
}
```

Interface – Implementierung

- Können von **konkreten** und **abstrakten** Klassen implementiert werden
 - Schlüsselwort: **implements**
- Eine Klasse kann eine oder mehrere Interfaces implementieren
- Ein Interface kann keine anderen Interfaces implementieren
- Ein Interface kann von anderen Interfaces erben, nicht von normalen Klassen

```
interface Fliegen {  
    void fliegen();  
}  
  
interface Schwimmen extends Fliegen { //Passt  
    void schwimmen();  
}
```

Interface – Implementierung

- Können von **konkreten** und **abstrakten** Klassen implementiert werden
 - Schlüsselwort: **implements**
- Eine Klasse kann eine oder mehrere Interfaces implementieren
- Ein Interface kann keine anderen Interfaces implementieren
- Ein Interface kann von anderen Interfaces erben
- die erste konkrete Klasse muss die noch nicht implementierten Methoden implementieren

Interface – Implementierung Beispiele

- abstrakte Klassen müssen Methoden des Interfaces nicht implementieren

```
interface MyInterface {  
    void myMethod();  
}  
  
abstract class MyAbstrClass implements MyInterface {  
}
```

Interface – Implementierung Beispiele

- die erste konkrete Klasse jedoch schon!

```
interface MyInterface {  
    void myMethod();  
}  
  
abstract class MyAbstrClass implements MyInterface {  
}
```

Methoden werden
überladen

```
public class MyClass extends MyAbstrClass {  
    @Override  
    public void myMethod() {  
        // Implementierung der Methode  
    }  
}
```

Interface – Implementierung Beispiele

```
interface MyInterface {  
    void myMethod();  
    void myMethod2();  
}  
  
abstract class MyAbstrClass implements MyInterface  
{  
    public void myMethod(){  
        System.out.println(„Hello“);  
    }  
}
```


Interface – Implem

Funktioniert!

MyAbstrClass implementiert die Methode myMethod().

MyClass erbt von MyAbstrClass und muss somit nur myMethod2() implementieren.

```
interface MyInterface {  
    void myMethod();  
    void myMethod2();  
}  
  
abstract class MyAbstrClass implements MyInterface {  
    public void myMethod(){  
        System.out.println(„Hello“);  
    }  
}  
  
public class MyClass extends MyAbstrClass {  
    public void myMethod2() {  
        // Implementierung der Methode  
    }  
}
```

Ist folgendes valider Code?

```
interface MyInterface {  
    void myMethod();  
}  
  
public abstract class MyAbstractClass implements MyInterface {  
  
}  
  
public class MyClass extends MyAbstractClass {  
  
}
```

- Können von konkreten und abstrakten Klassen implementiert werden
 - Schlüsselwort: implements
- Eine Klasse kann eine oder mehrere Interfaces implementieren
- Methoden müssen erst in der ersten konkreten (non-abstract) Klasse implementiert werden
 - D.h. ist eine Klasse abstrakt müssen die Methoden nicht zwingend implementiert werden

Problem! Die erste nicht-abstrakte Klasse muss die Methoden implementieren



Ist folgendes valider Code?

```
interface ABC{  
    default void m2() {}  
}  
  
interface DEF{  
    default void m3() {}  
}  
  
class Alphabet implements ABC, DEF{  
  
}
```

- Variablen sind Konstanten
- Methoden haben keinen Methodenkörper, außer sie sind static – oder default
 - Die Methoden dürfen nicht private, protected, final sein, bei static oder default auch nicht abstract sein
- Methoden dürfen nicht final sein
- Interfaces implementieren keine anderen Interfaces, sondern erben von einer oder mehr Interfaces
- Interfaces haben keinen Konstruktor

Funktioniert. Eine Klasse kann mehrere Interfaces implementieren



Ist folgendes valider Code?

```
interface ABC{
    default void m2() {};
}
interface DEF{
    default void m3() {};
}

interface Alphabet implements ABC, DEF{
}
```

- Variablen sind Konstanten
- Methoden haben keinen Methodenkörper, außer sie sind static – oder default
 - Die Methoden dürfen nicht private, protected, final sein, bei static oder default auch nicht abstract sein
- Methoden dürfen nicht final sein
- Interfaces implementieren keine anderen Interfaces, sondern erben von einer oder mehr Interfaces
- Interfaces haben keinen Konstruktor

Funktioniert NICHT. Ein Interface kann keine Interfaces implementieren



Ist folgendes valider Code?

```
interface ABC{  
    default void m2() {}  
}  
interface DEF{  
    default void m3() {}  
}  
  
interface Alphabet extends ABC, DEF{  
  
}
```

- Variablen sind Konstanten
- Methoden haben keinen Methodenkörper, außer sie sind static – oder default
 - Die Methoden dürfen nicht private, protected, final sein, bei static oder default auch nicht abstract sein
- Methoden dürfen nicht final sein
- Interfaces implementieren keine anderen Interfaces, sondern erben von einer oder mehr Interfaces
- Interfaces haben keinen Konstruktor

Funktioniert. Ein Interface kann ein oder mehrere Interfaces extenden



Ist folgendes valider Code?

```
class DEF{  
    void m3() {}  
}  
  
interface Alphabet extends DEF{  
  
}
```

- Variablen sind Konstanten
- Methoden haben keinen Methodenkörper, außer sie sind static – oder default
 - Die Methoden dürfen nicht private, protected, final sein, bei static oder default auch nicht abstract sein
- Methoden dürfen nicht final sein
- Interfaces implementieren keine anderen Interfaces, sondern erben von einer oder mehr Interfaces
- Interfaces haben keinen Konstruktor

Funktioniert NICHT. Ein Interface kann keine konkrete Klasse extenden



Ist folgendes valider Code?

```
interface ABC{  
    int x = 1;  
    void go();  
}  
  
class DEF implements ABC {  
    @Override  
    public void go() {  
        x = 23423;  
    }  
}
```

- können NUR Konstanten definieren
 - Der Compiler setzt automatisch public, static, final vor die Konstante
- es gibt keinen Konstruktor!
 - Beim Versuch kommt ein Compilerfehler
- abstrakte Methoden haben keinen Methodenkörper
 - Sind per-default public abstract
- static und default- Methoden müssen einen Methodenkörper haben

Nein! Der Wert von Konstanten kann nicht geändert werden.



Interface – Sonstiges

- Welche test()-Methode wählt die Klasse ABCD?
 - -> keine! Der Compiler fordert, dass die Methode überschrieben werden **MUSS**

```
interface ABC{
    default void test(){
        System.out.println(1);
    }
}

interface DEF{
    default void test(){
        System.out.println(2);
    }
}

class ABCD implements ABC, DEF{
    public static void main(String[] args) {
        new ABCD().test(); //Compilerfehler
    }
}
```


Abstrakte Klassen vs. Interfaces

Feature	Abstrakte Klassen	Interface
Methoden	Ja, abstrakt und konkrete	Ja, abstrakte und mittels default und static auch „konkrete“ (ab Java 8)
Konstruktoren	Ja	Nein
Felder	Ja	Nur Konstanten
Mehrfachvererbung	Nein, kann nur von einer Klasse erben	Ja, eine Klasse kann mehrere Interfaces implementieren -> Mehrfachvererbung soll initiiert/ermöglicht werden
Vererbung	Wie gewohnt: Methoden, Konstanten, Variablen werden vererbt	Default-Methoden und Konstanten werden „vererbt“

Interfaces - Aufgabe

Erstelle eine abstrakte Klasse Tier

- Attribute: name (String)
- Konstruktor, der den Namen setzt.
- Abstrakte Methode geraeuschMachen(), die jedes Tier individuell implementieren muss.
- Konkrete Methode zeigeTier(), die den Namen ausgibt.

Erstelle ein Interface Laufen

- Enthält die abstrakte Methode laufen(), die jede laufende Tierart implementieren muss.

Erstelle zwei abstrakte Unterklassen von Tier

- Säugetier
- Vogel
- Diese Klassen sind noch abstrakt und müssen geraeuschMachen() nicht zwingend implementieren.

Erstelle zwei konkrete Klassen

- Löwe (erbt von Säugetier und implementiert Laufen)
- Papagei (erbt von Vogel)

Teste deine Methoden!

