Technische Hochschule Rosenheim
Technical University of Applied Sciences

# Objektorientierte Programmierung
# Kapitel 4 – Vererbung

Prof. Dr. Kai Höfig

# Inhalt

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Motivation

- Prinzip der Vererbung
  - Hierarchiebildung
  - Vererben der Implementierung, Vererben der Schnittstelle

- Implementierungsvererbung in Java
  - Schlüsselwort `extends`
  - Konstruktoren, `super()`
  - Sichtbarkeit `protected`

- Datentypen in Vererbungshierarchien
  - Polymorphie

Literatur: http://openbook.rheinwerk-verlag.de/javainsel/ (Kapitel 5.8)

# Motivation

Technische Hochschule Rosenheim
Technical University of Applied Sciences


- Zentrale Frage der Software-Entwicklung:
  - Wie macht man Code wiederverwendbar?

# Typen in C – Vererbung?

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```
struct konto {
   int kontonummer;
   double saldo;
};

void zahleEin(double betrag, konto k){
        k.saldo+=betrag;
};
```

```
struct gemeinschaftskonto {
        int kontonummer;
        double saldo;
        char besitzer[10][100];
};

void zahleEin2(  double betrag,
                 gemeinschaftskonto k){
        k.saldo+=betrag;
};
```

- Nachteile von Typen in C:
  - Umliegender Programmcode muss sich um die **Konsistenz** der inneren primitiven Datentypen kümmern. Was passiert wenn z.B. `=` mit `+=` bei der Zuweisung von Saldi eines Kontos verwechselt wird? Wer stellt sicher, dass Methoden auch immer benutzt werden (z.B. `zahleEin(double betrag)`)?
  - **Redundanter** Programmcode entsteht schnell, wenn ähnliche Typen verwendet werden. Hier: `zahleEin` und `zahleEin2`

# Die Vererbung behebt dieses Problem

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```
class Konto {
    private int kontonummer;
    private double saldo;
        public void zahleEin(double betrag){
        saldo+=betrag;
    }
}
```

```
class Gemeinschaftskonto extends Konto{
        String[] besitzer = new String[10];
}

Gemeinschaftskonto gmk = new Gemeinschaftskonto();
gmk.zahleEin(200);
```

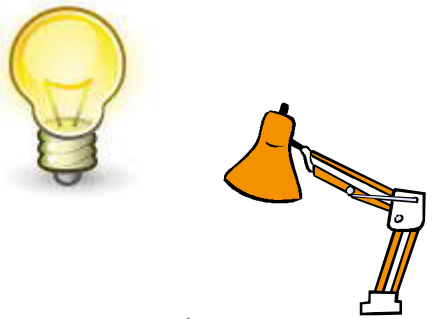
- Vorteile von Klassen in Java:
  - **Konsistenz** wird sichergestellt durch ausschließliche Verwendung der Methoden und Einschränkung des Zugriffs auf primitive Typen einer Klasse.
  - **Redundanter** Programmcode kann durch geschickte Vererbung drastisch reduziert werden. Hier muss die Methode `zahleEin` nur einmal implementiert werden.

# Grundprinzipien der Objektorientierung Teil 2

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- **Wiederholung: Abstraktion, Modularität, Datenkapselung**

- **Vererbung**
  - Repräsentieren von Abstraktionsebenen
  - Klassifizieren von Gemeinsamkeiten und Unterschieden
  - Ordnungsprinzip Vererbung: Ermöglicht die Definition neuer Klassen auf Grundlage von bereits bestehenden Klassen



- **Polymorphie (**"Vielgestaltigkeit")
  - ***Beispiel:*** Lampe / Glühbirne
    - Man kann jede Glühbirne einschrauben, die in die Lampenfassung passt.
    - Verschiedene Glühbirnen verhalten sich dennoch unterschiedlich (brennen hell oder weniger hell).
  - Objektorientierung erlaubt einfaches **Austauschen von Code** ("Glühbirne") solange die ***Schnittstelle*** ("Fassung") gleich bleibt.

**Lernziel: Wie unterstützt Java das Umsetzen dieser Prinzipien?**

# Zentrale Ziele

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Wie macht man Code ***wiederverwendbar***?
  - Erweiterung von bestehendem Code
  - Modifikation von bestehendem Code

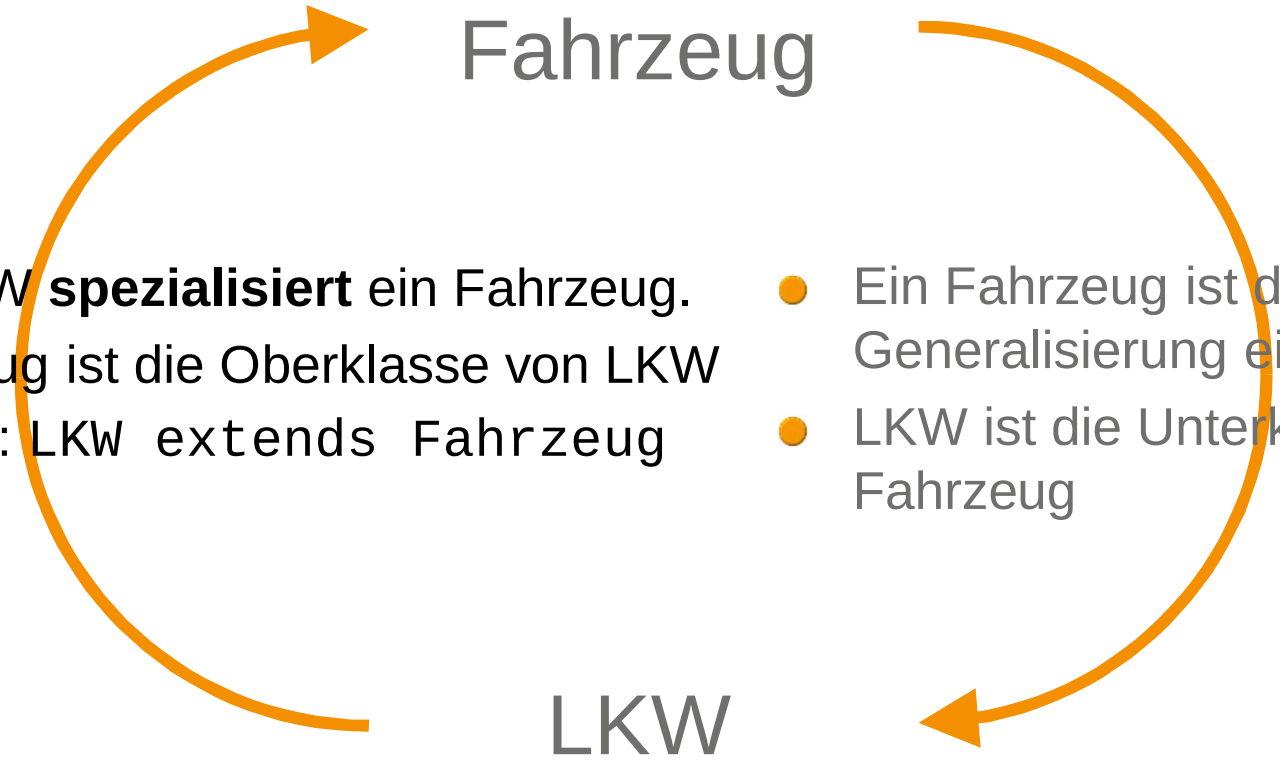
- Wie vermeidet man ***Redundanzen*** im Code?
  - ***Prinzip der einzigen Verantwortung***
  - Schwierige Wartung, falls 2 Module existieren, die die gleiche Aufgabe erfüllen.

Zentrales Vorgehen (im Alltag): **Generalisierung** und **Spezialisierung**

- Beispiel: Ein `Auto` und ein `Motorrad` sind Spezialisierungen eines `Fahrzeugs`. Ein `Fahrzeug` ist Generalisierung eines `Autos/Motorrads`.
- Spezialisierungen verfügen über alle Merkmale der Generalisierung, haben aber weitere Merkmale.

# Generalisierung und Spezialisierung

Technische Hochschule Rosenheim
Technical University of Applied Sciences




- Ein LKW **spezialisiert** ein Fahrzeug.
- Fahrzeug ist die Oberklasse von LKW
- In Java: `LKW extends Fahrzeug`

- Ein Fahrzeug ist die Generalisierung eines LKWs
- LKW ist die Unterklasse von Fahrzeug

# Übung: Generalisierung und Spezialisierung

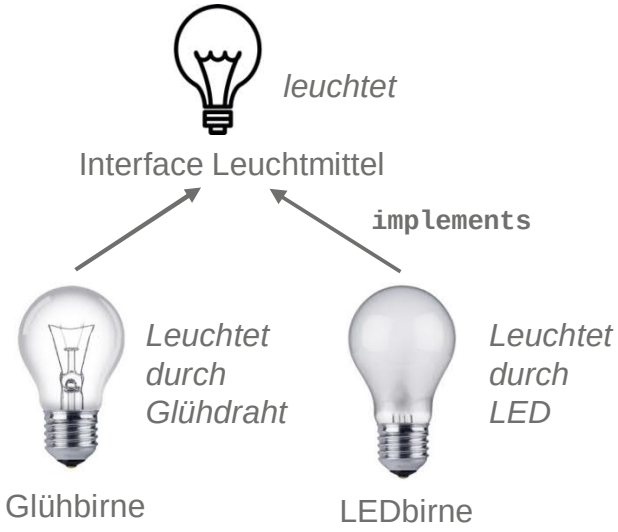
Technische Hochschule Rosenheim
Technical University of Applied Sciences

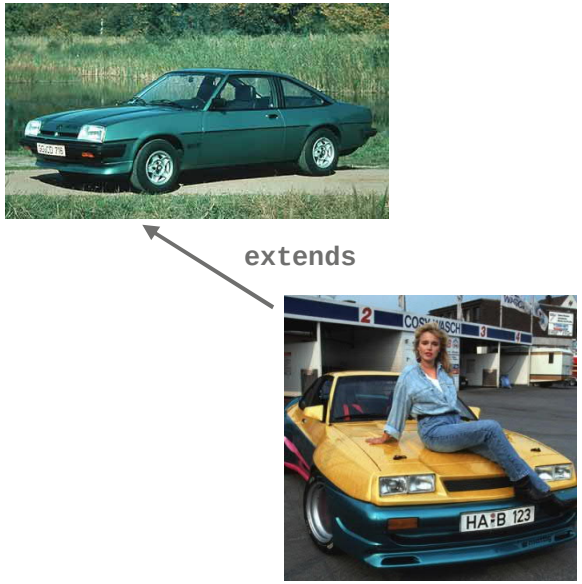
- Wie stehen die folgenden Begriffe zueinander in Beziehung?

*Boot* *Pekinese* *Venusfliegenfalle* *Gegenstände*
*Hausboot*
*Pflanze* *Vögel* *Säugetier*
*Aal*
*Lebewesen*
*Katze*
*Tier*
*Fisch*
*Delphin*
*Haus* *Wirbeltier* *Insekt*
*Hund*
*Biene*
*Hauskatze*
*Reale Einheiten*

- Welche Generalisierungs- und Spezialisierungsabhängigkeiten gibt es?
- Welche gemeinsamen Eigenschaften gibt es in einer solchen Abhängigkeit?

# Formen der Vererbung




- **Vererben der Schnittstelle**
  - Abstrakte Spezifikation gibt nur vor, welche Eigenschaften und welches Verhalten Objekte haben müssen (=Schnittstelle)
  - Beispiel: Glühbirnen müssen in bestimmte Lampenfassungen passen und dann brennen. Wie sie Licht erzeugen, wird nicht vorgegeben.
  - Java: `interfaces, implements` (siehe später).

- **Vererben der Implementierung**
  - Verfahren werden bereits oben in der Hierarchie (grob) implementiert / beschrieben.
  - Verfahren werden weiter unten genauer beschrieben bzw. können überschrieben werden.
  - Beispiel: Opel Manta GTI ist eine erweiterte Version des Opel Mantas. (z.B. Fuchsschwanz)
  - Java: `extends`

# Vererbung

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- **Definition: Vererbung**

  *Von bestehenden Klassen ausgehend können neue Klassen erstellt werden, die zunächst die **gleichen Eigenschaften und Methoden** besitzen wie die Ausgangsklasse. Die neue Klasse wird als abgeleitete Klasse oder **Unterklasse** bezeichnet, die Ausgangsklasse als Super- bzw. **Oberklasse**. Die abgeleitete Klasse kann die von ihrer Superklasse geerbten Eigenschaften und Methoden **überschreiben** oder durch zusätzliche **ergänzen**.*

# Schlüsselwort "`extends`"

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Verweis auf **Oberklasse** durch Schlüsselwort ***`extends`*** im Kopf der **abgeleiteten Klasse** (Unterklasse)
  - Beispiel:

```
Class Cat extends Pet {…}
```

- Abgeleitete Klasse erbt *alle* Variablen und *alle* Methoden der Oberklasse.

- Ändern der Funktionalität der Oberklasse möglich durch
  - Hinzufügen neuer Elemente (Attribute, Methoden, …)
  - Überladen der vorhandenen Methoden
    - Bsp: `public String getName(String greeting)`
  - Redefinieren (Überschreiben) der vorhandenen Methoden

# Konstruktoren in der Vererbung

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Jeder Konstruktor einer abgeleiteten Klasse sollte einen Konstruktor der Oberklasse aufrufen.
  - Ansonsten würden Attribute der Oberklasse gegebenenfalls niemals initialisiert.

- Expliziter Aufruf des Default-Konstruktors der Oberklasse:
  - `super();`

- Expliziter Aufruf eines Werte-Konstruktors der Oberklasse:
  - `super(name,…);`

- Bei fehlendem explizitem Aufruf:
  - Impliziter Aufruf des Default-Konstruktors der Oberklasse. Dieser muss explizit angegeben werden, sonst tritt ein Fehler auf.

- Regel: Ein Konstruktoraufruf **muss** immer *erstes* Statement im Konstruktor der Unterklasse sein

# Übung: Was ist jeweils falsch?

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```
class Point {
    double x,y;
    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
class Pixel extends Point {
    private long color;
    Pixel(int x1, int y1, int c) {
        x = x1;
        y = y1;
        this.color = c;
    }
}
```

```
class Point {
    int x,y;
    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

class Pixel extends Point {
    private long color;
    Pixel(int x1, int y1, int c) {
        this.color = c;
        super(x1,y1);
    }
}
```

## Schlüsselwort "super"

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- super
  - Zeigt auf Oberklasse
  - super.methode() bzw. super.attr greift auf Methode bzw. Attribut der Oberklasse zu.
  - Spezialfall: super() entspricht Aufruf des Konstruktors der Oberklasse

- this
  - Zeigt auf aktuelle Klasse
  - Erlaubt Zugriff auf Attribute und Methoden der aktuellen Klasse

### Oberklasse

```
public class Pet {

    public void talk() {
        …
    }
}
```

### Unterklasse

```
public class Cat extends Pet {

    . . .

    // redefine method of superclass
    public void talk() {
        super.talk();
        System.out.println("Miau");
    }
}
```

# Sichtbarkeit "protected"

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Weitere Sichtbarkeit: `protected`

- Programmiert man in der Unterklasse, so hat man Zugriff auf Elemente der Oberklasse, falls diese die folgenden Sichtbarkeiten haben:
  - **`public`**
  - **`protected`**
  - Keine Sichtbarkeit angegeben, d.h. sichtbar im ganzen Package.

## Oberklasse

```
public class Pet {
    …
    private State state;
    private String name;
    private String owner;
    protected String favouriteDish;
    …
}
```

## Unterklasse

```
public class Cat extends Pet {

    public Cat(String name,
               int jumpHeigwhiskerSize) {
        super(name);
        this.jumpHeight = jumpHeight;
        favouriteDish = "mice";
    }
    …
}
```

Wäre nicht möglich, falls favouriteDish private

# Aufruf ererbter Methoden

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```
Cat cat = new Cat("garfield", 100);
cat.eat();
```

## Oberklasse

```
public class Pet {
    public void eat() {
        …
    }
    …
}
```

## Unterklasse

```
public class Cat extends Pet {
    …
    // eat() nicht implementiert
}
```

Methode `eat` nur in Oberklasse implementiert.

- Vererbung für Anwender einer Klasse nicht erkennbar!
- Bei Methodenaufruf wird zur Laufzeit zuerst in der Klasse des Objekts nach der passenden Methode gesucht, dann in Oberklasse, dann in Oberklasse der Oberklasse, etc.

# Anwendung der Implementierungsvererbung

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- ***Neue Funktionalität*** für eine Klasse
  - Hinzufügen neuer Attribute und Methoden
  - Überladen vorhandener Methoden: Parameterliste muss sich von Parameterliste der gleichnamigen Methode der Oberklasse unterscheiden.

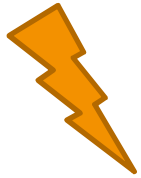
- ***Ändern bestehender Funktionalität***
  - Redefinition / Überschreiben vorhandener Methoden: Rumpf kann komplett ersetzt werden.
  - Name und Parameterliste der überschriebenen Methode müssen *exakt* übereinstimmen.
  - Sichtbarkeit darf in Unterklasse gelockert werden bzw. Attribut / Methode der Unterklasse darf nicht "privater" sein als in der Oberklasse.

# Annotation @Override

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Methode `talk()` von Klasse `Pet` soll in Klasse `Cat` überschrieben werden.

```
@Override
public void talks() {
```

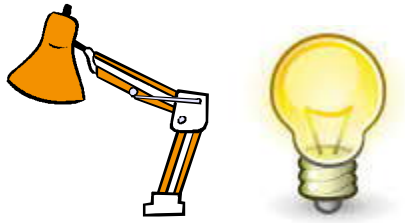


- Vorteil von `@Override`
  - Leichtsinnsfehler des Benutzers werden erkannt.
  - IDE bzw. Compiler kann Warnung generieren, dass die Methode talks() nichts überschreibt.

- Korrekt wäre gewesen:

```
@Override
public void talk() {
```

# Polymorphie



- ***Bisheriges Beispiel:*** Lampe / Glühbirne
  - Man kann jede Glühbirne einschrauben, die in die Lampenfassung passt.
  - Verschiedene Glühbirnen verhalten sich dennoch unterschiedlich (brennen unterschiedlich hell).

- ***Definition: Polymorphie***
  - Konzept in der objektorientierten Programmierung, in der ein ***Bezeichner*** abhängig von seiner Verwendung ***unterschiedliche Datentypen*** annimmt.
  - In älteren typisierten Programmiersprachen wird dagegen jedem Namen und jedem Wert im Quelltext eines Programms höchstens ein Typ zugeordnet (*Monomorphie)*.

- Vererbungshierarchien bzw. mehrere Datentypen erlauben einfaches **Austauschen von Code**
  - Schreibe neue Unterklasse.
  - *Substitutionsprinzip*: An Stelle eines Objektes kann immer auch ein Objekt der Unterklasse auftauchen.

# Datentypen in Vererbungshierarchien

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Zu jeder Klasse gibt es in Java auch einen ***Datentyp***
  - ***Primitive Datentypen***: `int, boolean,` etc.
  - Beispiele für Klassen-Datentypen / ***Referenztypen***:
    - `Pet, Cat, Dog, Object, …`

- Jede Unterklasse hat automatisch auch den Datentyp der Oberklasse.
  - Ein `Cat`-Objekt hat auch die Datentypen `Pet` und `Object`.
  - Welche der folgenden Zuweisungen ist nicht korrekt?

```
class Pet{…}
class Cat extends Pet {…}
class Dog extends Pet {…}
…
Cat cat  = new Cat("garfield1", 100);
Pet pet1 = new Cat("garfield2", 200);
Pet pet2 = new Dog("rantanplan", 100);
Dog dog  = new Cat("garfield3", 300);
Cat cat2 = new Pet();
```

# Datentypen in Vererbungshierarchien

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Was passiert?
  - Fehlermeldung: `"Type Mismatch: cannot convert from Pet to Cat"`
  - Der Compiler "vergisst" nach der ersten Anweisung, dass *p* ein Cat-Objekt ist.
  - Die Laufzeitumgebung kann später aber sehr wohl erkennen, dass es sich um ein Cat-Objekt handelt.

```
Pet p = new Cat("garfield", 200);
Cat cat = p;
```

- Abhilfe: **Expliziter Typecast**
  - Man teilt dem Compiler mit, dass das Objekt als Katze zu interpretieren ist.

```
Pet p = new Cat("garfield", 200);
Cat cat = (Cat)p;
```

- **Substitutionsprinzip**
  - An Stelle eines Objektes kann immer auch ein Objekt der Unterklasse (eine Spezialisierung) auftauchen.
  - Im umgekehrten Falle ist ggfs. ein expliziter Typecast zu verwenden. Das funktioniert natürlich nur, wenn es auch stimmt, also `p` auch wirklich vom Typ `Cat` ist.

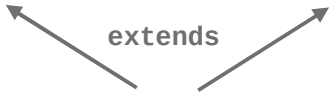
# Das Problem mit der Mehrfachvererbung

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Klassen erlauben es Probleme in handhabbare, modulare Teile zu zerlegen, Datenkapselung hilft dabei Objekte konsistent zu halten und Vererbung verhindert Redundanzen durch die Wiederverwendung von Eigenschaften und Methoden in Spezialisierungen.
- Polymorphie erlaubt es uns Objekte wieder zu generalisieren und sie so universeller einsetzbar zu machen.
- Aber was ist damit?

```
class Haus{…}
class Boot {…}
class HausBoot extends Haus, Boot {…}
```

In Java gibt es keine Mehrfachvererbung mit `extend`. Wieso nicht?




# Das Problem mit der Mehrfachvererbung

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Bei Mehrfachvererbung ist nicht eindeutig klar, welche Methoden und Attribute jetzt gültig sind. In Java realisiert durch ***Interfaces***

```
class Haus{
   …
   public void beleuchte(){
      this.wohnzimmerlicht.an();
   }
}
class Boot {
   …
   public void beleuchte(){
      this.positionslichter.an();
   }

}
class HausBoot extends Haus, Boot {
   …
   // welche Methode beleuchte() gilt jetzt?
}
```


# Interfaces: Deklaration

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- **Schnittstelle in Programmiersprache**
  - Vereinbarung gemeinsamer Signaturen von Methoden.
  - Definiert das Verhalten von Objekten
    - "Was?" aber nicht "Wie?"

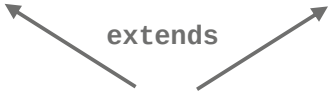
- **Schnittstellen / Interfaces in Java**
  - Besondere Form einer Klasse.
  - Schlüsselwort `interface`
  - Konvention: 1 Interface pro Java-Datei, Name endet oft auf "`able`".
  - Interfaces enthalten
    - **Methoden ohne Implementierung**  Wieso?
    - Default Methoden (ab Java 8), die nur auf die (abstrakten) Methoden des Interfaces zugreifen.
    - Konstanten
  - Interfaces enthalten keine Konstruktoren!

```
public interface Trainable {

   int train();  // trainable objects must implement a method train()

}
```

# Problemlösung durch Interfaces

- **Interfaces** geben vor *was* implementiert werden muss, aber nicht *wie.* Dadurch können Objekte die bestimmte Interfaces implementieren universeller eingesetzt werden. Sie haben einen zusätzlichen Typ, den des Interfaces.

```
interface Haus{
   …
   public void beleuchte();
}
interface Boot {
   …
   public void beleuchte();
}
class HausBoot implements Haus, Boot {
   …
   public void beleuchte(){
      this.rundumbeleuchtung.an();
   }
}
```

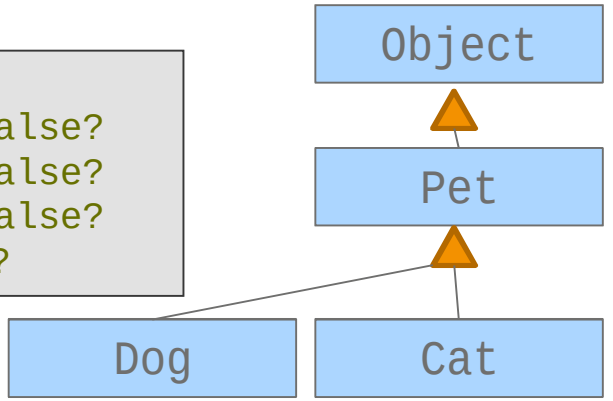



# Motivation: Mehrere Datentypen

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Unterschied: **Klasse** und **Datentyp**
  - Jedes Objekt o gehört genau zu 1 Klasse → `getClass()`
  - Jedes Objekt o kann mehrere Typen haben.

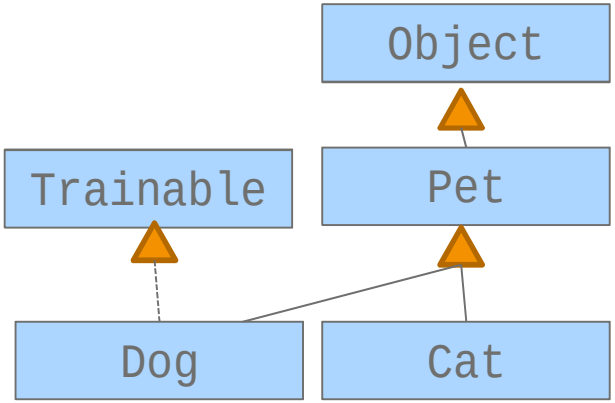
```
Pet p = new Cat();
boolean b1 = p instanceof Cat;      → true oder false?
boolean b2 = p instanceof Pet;      → true oder false?
boolean b3 = p instanceof Object;   → true oder false?
Class c = p.getClass()              → which class?
```




- Wie erhält `Dog` zusätzlich den Datentyp `Trainable`?
  - Bedeutung: Tier ist dressierbar. Jedes dressierbare Tier muss die Methode `train()` implementieren.

- **Probleme**
  - Mehrfachvererbung in Java nicht erlaubt!
  - `Dog` kann nicht von `Pet` **und** gleichzeitig von `Trainable` erben

- **Java Lösung:** `Interface`



# Interfaces: Implementierung

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Eine Klasse kann eine oder **mehrere** Schnittstellen verwenden bzw. **implementieren**.
  - Erweiterung der `class` Anweisung um `implements`-Klausel.
  - Überschreiben der Methode mit einer Implementierung.

```
public class Dog extends Pet implements Trainable{
    private int speed;

    public Dog(String name, int speed) {
        super(name);
        this.speed = speed;
    }

    public void talk() {
        super.talk();  // produce some noise + dog-specific sound
        System.out.println("Wau! Wau!");
    }

    @Override
    public void train() {
        speed++;
}}
```

Verpflichtung, alle Methoden des Interface zu implementieren

# Interfaces

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```
Dog d = new Dog(.);
boolean b1 = d instanceof Trainable;   true/false?
boolean b2 = d instanceof Pet;         true/false?
```

- Compiler verleiht der Instanz zusätzlich den Datentyp `Trainable`.
- Implementierende Klasse implementiert
  - entweder **jede** Methode im Interface
  - oder nur einen Teil der Methoden und die Klasse wird als **abstrakte** Klasse definiert (siehe späterer Abschnitt)
- Damit ein Objekt einer Klasse angelegt werden kann, müssen **alle** Methoden **aller** angegebenen Schnittstellen implementiert sein.

# Diskussion: Interfaces

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Verleihen eines **zusätzlichen Datentyps** an eine Klasse
  - Objekte der Klasse können in verschiedenen Rollen auftreten
  - Polymorphie, siehe Extra-Abschnitt!

- **Gemeinsames Verhalten** von **unabhängigen Klassen** wird abstrahiert
  - "Versprechen" bestimmte Operationen umzusetzen.

- Beschreiben von Eigenschaften einer Klasse, die nicht direkt durch normale Implementierungsvererbung abbildbar ist.
  - Ein Objekt `Trainable` ergibt keinen Sinn.
  - Keine "`is-a`"-Beziehung wie bei Implementierungsvererbung

- **Trennung** zwischen Schnittstelle und Implementierung
  - *Schnittstelle*: Enthält alle Informationen, die ein Anwender der Klasse benötigt.
  - *Implementierung*: Umsetzung der versprochenen Funktionalität – Methoden & Datenelemente

# Jedes Interface ist ein Datentyp

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Interfaces sind ganz normale Datentypen!
  - Können z.B. als Methodenparameter verwendet werden.

```
public static boolean trainAll(Trainable[] array) {
    for (Trainable t : array) {
        t.train();
    }
}
```

  - Typecasts können notwendig sein.

```
public static int price(Object o) {
    if (o instanceof Trainable) {
        Trainable t = (Trainable) o;
        return 100;
    }
    else {
        return 50;
    }
}
```

# Subinterfaces und Konstanten

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Interfaces lassen sich um zusätzliche Methoden **erweitern**
  - **Subinterfaces**: "Implementierungsvererbung von Schnittstellen"
  - Beispiel: Zusätzliche Methode, die den Mehrwertsteuersatz zurückgibt.

```
public interface Buyable {
    int price();     // return price of buyable objects
}
```

```
public interface BuyableTaxable extends Buyable {
    double salesTax();      // return sales tax (in percent)
}
```

- Interfaces dürfen **benannte Konstanten** enthalten
  - Alle Attribute einer Schnittstelle sind immer implizit `public static` final.

```
public interface Trainable {
    int MAX_PRICE = 10000000;
    int price();   // return price of buyable objects
}
```

# Markierungsschnittstellen

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Leere Schnittstellen ohne jegliche Methoden
- Einzige Anwendung
  - Mit `instanceof` kann leicht überprüft werden, ob ein Objekt einen bestimmten Typ besitzt oder nicht.
- Beispiele:
  - `java.lang.Cloneable`
  - `java.util.EventListener`
  - `java.io.Serializable`
- Bei neuen Bibliotheken sind Markierungsschnittstellen eher selten anzutreffen.

# Abstrakte Klassen

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- **Motivation**
  - Vorgabe der Verhaltensweise (Was?) **und** Teile der Implementierung (Wie?).
  - Mischung zwischen Schnittstellenvererbung und Implementationsvererbung

- **Abstrakte Klasse**
  - Enthält Datenelemente, vollständigen Methoden und Methodensignaturen.
  - Kann nicht instanziiert werden → Gegenteil von konkreten Klassen.
  - Wird durch Modifizierer `abstract` gekennzeichnet.

```
abstract class Counter  {

    protected int count = 0;

    void reset(){
        count = 0;
    }

    int read() {
        return count;
    }

    // nur Schnittstelle
    abstract void step();
}
```

**Beispiel:** Zähler schreibt Verhaltensweise und Kernelement der Implementierung vor, lässt aber offen *wie* und um *wieviel* der Zähler jeweils erhöht wird.

# Vererbung bei abstrakten Klassen

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Abstrakte Klasse ist unvollständig und kann nicht instanziiert werden.

- 2 Möglichkeiten bei Vererbung
  - Überschreibe in Unterklasse alle abstrakten Methoden → Unterklasse ist **konkret**.
  - Überschreibt nur einen Teil oder gar keine der abstrakten Methode in Unterklasse → Unterklasse bleibt **abstrakt**.

- Konkretisierung einer abstrakten Klasse kann schrittweise über mehrere Vererbungsstufen erfolgen.

```
public class AddCounter
extends Counter {
    @Override
    void step() {
        count++;
    }
}
```

```
public class DoubleCounter
extends Counter {
    @Override
    void step() {
        count *= 2;
    }
}
```

# Zusammenfassung Klasse, abstrakte Klasse und Interface

Technische Hochschule Rosenheim
Technical University of Applied Sciences

| | Klasse | Abstrakte Klasse | Interface |
|---|---|---|---|
| Enthält Attribute | Ja | Ja | Nur als public static final |
| Enthält Methoden | Ja | Ja | Nur Signaturen (als Methodenrumpf) |
| Vererbung | Einfach mit extends | Einfach mit extends | **Mehrfach** mit implements |
| Kann mit new() instanziiert werden | Ja | **Nein** | Nein |
| Ist polymorph | Ja | Ja | Ja |
| Bei Erweiterung um Attribute/Methoden.. | Wird vererbt | Wird vererbt | Alle Klassen, die das Interface implementieren müssen angepasst werden. |

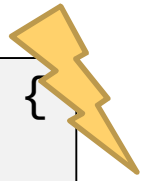
# Finale Klassen

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Klassen können mit dem Schüsselwort **`final`** versehen werden.
- Von finalen Klassen kann man nicht ableiten.

- Beispiel:

```
final public class Pet {
   . . .
}
```



```
public class Cat extends Pet {
  . . .
}
```

# Wiederholung: Immutables

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- **Wie macht man in Java eine Klasse *immutable*?**
  - Deklariere die Klasse als `final`
    - Verhindert, dass man von der Klasse ableiten darf.
    - ***Wieso?*** Weil abgeleitete Objekte durch Methoden diese Eigenschaft zunichte machen können.
  - Deklariere alle Attribute als `private` und `final`
  - Keine Methoden, die Attribute verändern
    - Ausnahme: Konstruktor

Die Methode `setI()` erlaubt das nachträgliche Ändern eines Objekts, welches auch vom Typ `immutable` ist (polymorphie)

```
public class immutable {
    private int i;
    public immutable(int i){
        this.i=i;
    }
}
public class notImmutable extends immutable{
    public notImmutable(int i){
        super(i);
    }
    public void setI(int i){
        this.i=i;
    }
}
```

# Quellenverzeichnis

Technische Hochschule Rosenheim
Technical University of Applied Sciences


[1] C. Ullenboom. *Java ist auch eine Insel*, 11. Auflage, Galileo Computing, Kapitel 6.2

[2] http://dilbert.com/strip/2014-08-12 (abgerufen am 07.05.2017)

[3] www.itwissen.de, 01.April 2016)