

# Praxis der Programmierung

**Zuweisungskompatibilität,  
Pakete, Exceptions, Generics**

**Institut für Informatik und Computational Science  
Universität Potsdam**

**Henning Bordihn**

# Zuweisungskompatibilität

## Abstrakte Klassen – Beispiel

- stehen für abstrakte Konzepte der Anwendungsdomäne
- können Datenelemente, implementierte und abstrakte Methoden enthalten
- Beispiel: Bewegliche Objekte
  - können lebendig sein oder nicht
  - bewegliche Objekte sind z.B. Tiere
  - nicht lebende bewegliche Objekte sind z.B. Fahrzeuge
  - Fahrzeuge haben stets eine Geschwindigkeit und können beschleunigt werden
  - alle beweglichen Objekte können eine Art Laut/Geräusch von sich geben

## Bewegliche Objekte

```
public abstract class Bewegliches {  
  
    protected String klang;  
    protected boolean lebend;  
  
    public void setKlang(String klang) {  
        this.klang = klang;  
    }  
  
    public abstract void rede();  
  
}
```

```
public abstract class Fahrzeug  
    extends Bewegliches {  
    protected float geschwindigkeit;  
  
    protected Fahrzeug() {  
        lebend = false;  
    }  
  
    public void gibGas(float dg) {  
        geschwindigkeit += dg;  
    }  
  
    public float tacho() {  
        return geschwindigkeit;  
    }  
}
```

## Autos

```
public class Pkw extends Fahrzeug {  
  
    public Pkw() {  
        super();  
        klang = "tuut";  
        geschwindigkeit = 0;  
    }  
  
    public void rede() {  
        System.out.println(  
            klang + " " + klang  
        );  
    }  
}
```

```
public class Lkw extends Fahrzeug {  
  
    public Lkw() {  
        super();  
        klang = "honk";  
        geschwindigkeit = 0;  
    }  
  
    public void rede() {  
        System.out.println(klang);  
    }  
}
```

## Tiere

```
public class Tier extends Bewegliches {

    private int hungrig;

    public Tier(String klang) {
        this.klang = klang;
        lebend = true;
        hungrig = 10;
    }

    public void fressen() {
        hungrig-- ;
    }

    public void rede() {
        if (hungrig == 0);
        for(int i=1; i<hungrig; i++)
            System.out.print(klang);
        System.out.println(klang);
    }
}
```

## Eine Kolonne beweglicher Objekte

```
Bewegliches[] kolonne = new Bewegliches[3];
```

```
Bewegliches hund = new Tier("wau");
```

```
Bewegliches porsche = new Pkw();
```

```
Bewegliches truck = new Lkw();
```

```
kolonne[0] = hund;
```

```
kolonne[1] = truck;
```

```
kolonne[2] = porsche;
```

```
for(int i=0; i<kolonne.length; i++)
```

```
    kolonne[i].rede();
```

## Statischer *versus* dynamischer Typ

```
porsche.gibGas(100);    // schlaegt fehl (Warum?)
hund.fressen();        // schlaegt auch fehl

// Downcast
Pkw cast;
// cast = (Pkw) porsche;    // ok aber gefaehrlich!!!

// besser vorher dynamischen Typ pruefen:
if (porsche instanceof Pkw) { // true falls porsche auf Pkw-Objekt verweist
    cast = (Pkw) porsche;
    cast.gibGas(100);
}

// alternativ: lokaler Downcast
if (hund instanceof Tier)
    ((Tier)hund).fressen();
```



## Casting von Verweisdatentypen

### Situation:

- Variable `varA` vom Typ Verweis auf Instanzen der Klasse `A`
- Zuweisung eines Exemplars einer Unterklasse `B` von `A`: `A varA = new B();`

↪ Ausdruck `(varA instanceof B)` liefert `true`

1. **Indikation:** Zuweisung an eine Variable vom Typ `B`

**Syntax:** `B varB = (B)varA;`

2. **Indikation:** Aufruf einer Methode `methode()`, die in `B`, nicht aber in `A` existiert

**Syntax:** `((B)varA).methode();`

## Interfaces

- Interfaces sind reine Schnittstellen, die keinerlei Implementierung enthalten.
- Alle Methoden sind implizit `abstract`, alle Datenelemente sind implizit `final static` (ohne Angabe dieser Schlüsselwörter!).

```
public interface Bewegliches {  
    public void setKlang(String klang);  
    public void rede();  
}  
  
public interface Lebend {} // flag-Interface
```

## Interfaces (2)

- Eine Klasse kann ein oder mehrere Interfaces implementieren.

```
public class Tier implements Bewegliches, Lebend { ... }
```

Dann müssen alle Methoden dieser Interfaces implementiert werden.

- Die Eigenschaft einer Klasse, ein Interface zu implementieren, wird an ihre Unterklassen vererbt.
- Interfaces definieren auch Datentypen:

```
Bewegliches katze = new Tier("mau");
```

- Ist ein Interface implementiert?:

```
object instanceof interf // true falls Klasse von object interf implementiert
```

# **Pakete und Importe, Die Systemvariable CLASSPATH**

## Pakete

- Namensräume werden durch Pakete definiert (kein namespace)
- hierarchische Definition von Paketen möglich
- vor dem Schlüsselwort `class`:
  - `package paketname;`
    - ↪ so definierte Klassen gehören zum Paket `paketname`
    - ↪ Datei muss im Unterverzeichnis `paketname` liegen
  - `import paket.Klasse;`  
`import paket2.*;`
    - ↪ Klasse aus `paket` und alle Klassen aus `paket2` können hier benutzt werden
  - alle Klassen der Standardbibliothek außer jener in `java.lang` müssen importiert werden

## Zusammenspiel mit Systemvariablen

- Angaben nach `package` und `import` sind **relative** Pfadangaben
- Ergänzung zu absoluten Pfadangaben über Systemvariablen
  - bei Paketen der Klassenbibliothek bei Installation vordefiniert
  - bei selbst definierten Paketen:  
Wert von `CLASSPATH` zeigt auf das Verzeichnis,  
von dem aus die relativen Pfadangaben beginnen

```
import paket.Klasse;
```

↪ Klasse wird gesucht in `$CLASSPATH/paket`

# Exceptions und ihre Behandlung

## Exceptions vs. Errors

### Exceptions

---

eher leichte Laufzeitfehler  
können abgefangen werden

Beispiele:

`ArithmeticException`

`ArrayIndexOutOfBoundsException`

`StringIndexOutOfBoundsException`

`NumberFormatException`

`NullPointerException`

`EOFException`

### Errors

---

eher schwere Laufzeitfehler  
führen zum Programmabbruch

`NoClassDefFoundError`

`OutOfMemoryError`

`Internal Error`



## Exceptions sind Objekte

- Exception-Typ  $\longrightarrow$  Klasse (Bsp.: `java.lang.NullPointerException`).
- Tritt ein Laufzeitfehler ein, wird ein Exemplar der jeweiligen Exception-Klasse erzeugt (*Throwing*).
- Alle Exception-Klassen sind Unterklassen von `java.lang.Exception`.
  - hierarchischer Aufbau der Exceptions
  - Verteilung auf verschiedene Pakete
- Sie besitzen daher alle gewisse Methoden, u.a.:
  - `getMessage()`, die bestimmte Informationen über den Fehlerfall liefert  
Beispiel: `java.lang.ArrayIndexOutOfBoundsException: -1`
  - `printStackTrace()`, die die Fehlermeldung und die dynamische Aufrufhierarchie auf `stdout` ausgibt

## Explizites Abfangen von Exceptions

```
try {  
    // Anweisungen, in denen eine XYException  
    // ausgelöst werden kann  
}  
  
catch (XYException e) {  
    // Anweisungen, die beim Auftreten einer  
    // XYException ausgeführt werden, z.B.  
    System.out.println(e.getMessage());  
}
```

mehrere Fehlerarten → mehrere catch-Blöcke

## Weitergeben von Exceptions

- Alternativ werden Exceptions zur Behandlung weitergegeben, und zwar
  - erst an übergeordnete Programmblöcke derselben Methode,
  - dann (entlang der dynamischen Aufrufhierarchie) an die Aufrufer der Methode,

in der die Exception ausgelöst wurde.

Wird sie nirgends explizit abgefangen, so bricht das Programm ab.

- die `throws`-Klausel signalisiert, welche Exceptions die jeweilige Methode nicht selbst behandelt:

```
public void methode() throws IOException { ... }
```

- Auf alle Exceptions außer `RuntimeExceptions` muss der Programmierer reagieren!!!

## Benutzerdefinierte Exceptions

- für Fehlersituationen im Zusammenhang mit benutzerdefinierten Klassen
- Auslösen mit `throw <Exemplar einer Exception-Klasse>`, z.B.:

```
if ( //Fehlersituation )
    throw new RuntimeException("Denominator is zero.");
```

- Jede Exception-Klasse besitzt Konstruktoren

```
XYException()    und    XYException(String message)
```

- benutzerdefinierte Exceptions als Unterklasse von einer Exception-Klasse:

```
MyException() {super()}    und
MyException(String msg) {super(msg)}
```

# Generische Typen, Generics

## Generische Klassen und generische Typen

- generische Klassen (**Generics**) werden als Klassen mit Typvariablen definiert:

```
public class Pair<T,U>
```

- Typvariablen können in der Klassendefinition jetzt benutzt werden wie Typen:

```
private T firstComp;  
private U secondComp;
```

(auch für Parametertypen und Rückgabetypen)

- Typvariablen werden durch Typen gebunden, z.B.:

```
Pair<String, Integer> pp;
```

↪ **generischer Typ**