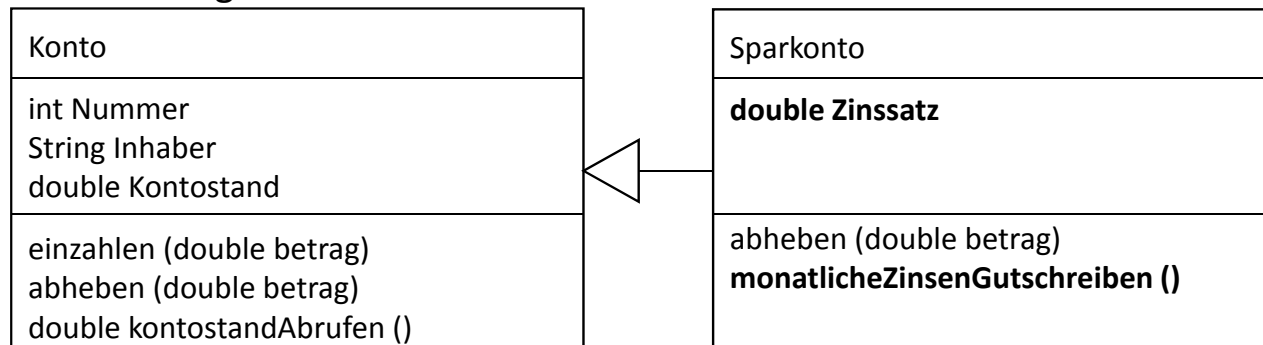


# Programmierung 2

## Kapitel 7 – Abstrakte Klassen und Interfaces

# Vererbung bis hierher

- Verfeinerung einer Oberklasse durch Spezialisierung
  - Unterklasse hat alle Merkmale der Oberklasse + zusätzliche Merkmale
- Beispiel Konto **spezialisiert zu** Sparkonto:
  - Sparkonto erbt von Konto
    - Objekte der Klasse Sparkonto haben dieselben Methoden und Attribute wie Konto
    - Klasse Sparkonto **überschreibt** die Methode auszahlen
      - Auszahlungen ziehen eine 2 Euro Gebühr nach sich



# Offene Fragen

- Vererbung ist **Spezialisierung**
  - Aus existierenden Klassen spezialisierte Klassen ableiten
    - Ein Mountainbike ist ein spezielles Fahrrad
    - Ein Fahrrad muss aber kein MTB sein → beides kann existieren
- Was ist mit **Generalisierung**?
  - Aus mehreren ähnlichen Klassen gemeinsame Attribute und Methoden herauslösen und separat modellieren?
    - Fahrrad, Auto, Dreirad, Zug, Schiff haben Eigenschaften von Fahrzeugen
    - Aber allgemeines „Fahrzeug“ existiert nicht
- **Keine Mehrfachvererbung** in Java
  - Was tun, wenn z.B. Klasse Amphibienfahrzeug sowohl Eigenschaften von Auto als auch von Schiff besitzt?

# Abstrakte Klassen und Methoden

# Beispiel Abstrakte Klassen

- Instanzen einer Klasse Fahrzeug dürfen **nicht existieren**
- Allgemeine Eigenschaften von Fahrzeugen hier deklarieren
  - Höchstgeschwindigkeit, Gewicht, Besitzer, Fahrer...
- Abgeleitete Klassen **existieren**
  - Fahrrad, Auto, Motorrad...
  - Spezielle Eigenschaften in diesen Klassen
    - Fahrrad: Anzahl Gänge, Rahmengröße...
    - Auto: Durchschnittsverbrauch, Zuladung, Sitze...

# Beispiel Abstrakte Klassen

```
public abstract class Fahrzeug {  
    private double gewicht;  
    private double maxGeschwindigkeit;  
    private Person fahrer;  
  
    public Person getFahrer() {  
        return fahrer;  
    }  
    //...
```

```
public class Fahrrad extends Fahrzeug {  
    private int anzGaenge;  
    private float rahmenGroesse;  
    //...
```

```
public class Auto extends Fahrzeug {  
    private double verbrauch;  
    private double zuladung;  
  
    public double berechneVerbrauch(double km) {  
        return verbrauch / 100.0 * km;  
    }  
    //...
```

# Abstrakte Klasse

- Deklaration über Schlüsselwort **abstract**
- Kann keine Objekte erzeugen (kann nicht instantiiert werden)
- Kann abgeleitet werden (Unterklassen erben von abstrakter Klasse)

# Abstrakte Methoden

- Abstrakte Klassen können Methoden enthalten, die als **abstract** deklariert sind
  - Keine Implementierung der Methode in der abstrakten Klasse
- Idee dahinter
  - Vorgabe von Name, Parameter, Rückgabetyp („Signatur“) von Methoden
  - Aber keine Festlegung, wie die Methode zu implementieren ist
- Unterklassen müssen alle abstrakten Methoden implementieren oder sie ebenfalls als **abstract** deklarieren
  - Abgeleitete Klasse ist dann ebenfalls abstrakt

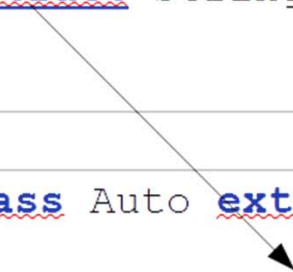


# Beispiel Abstrakte Methoden

- Klasse Fahrzeug soll *grundsätzlich* Routenberechnung ermöglichen
  - Implementierung in Fahrzeug unmöglich (Autobahn vs. Radweg)
  - Daher: abstrakte Methode in Fahrzeug

```
public abstract class Fahrzeug {  
    // keine Implementierung der Methode, daher kein Methodenrumpf!  
    public abstract String berechneRoute(String von, String nach);  
    //...  
}
```

```
public class Auto extends Fahrzeug {  
    //...  
    public String berechneRoute(String von, String nach) {  
        //...  
        return route;  
    }  
}
```



## Beispiele Zur Verwendung

```
// Fehler! Darf keine Instanz von Fahrzeug bilden
Fahrzeug f1 = new Fahrzeug();

// ok
Fahrzeug f2 = new Auto();
Fahrzeug f3 = new Fahrrad();
Auto f4 = new Auto();

// Fehler! Gemeinsame Oberklasse, aber unterschiedl. Spezialisierung
Auto f5 = new Fahrrad();

// ok: Aufruf von Methoden in Klasse vom Typ Auto
String route1 = f4.berechneRoute("Leipzig", "Berlin");

// ok: Aufruf von abstrakten Methoden,
// die in Klasse Fahrzeug nicht implementiert sind
String route2 = f2.berechneRoute("Leipzig", "Berlin");
```

## Weiteres Beispiel: Abstrakte Klassen

- Die Fragen einer Prüfung sollen modelliert werden
- Es gibt unterschiedliche Arten von Fragen, z.B.
  - Wahr/Falsch Fragen
  - MultipleChoice Fragen
- Allgemeine Eigenschaften einer Frage
  - Jede Frage hat einen Fragetext und eine zugeordnete Punktzahl
  - Fragen können gestellt werden
  - Fragen können beantwortet werden
    - Bei einer richtigen Antwort bekommt ein Prüfling die entsprechende Punktzahl gutgeschrieben

## Weiteres Beispiel: Abstrakte Klassen

```
abstract class Frage {  
  
    String text;        // Fragetext  
    int punkte;        // zu erreichende Punktzahl  
  
    Frage(String text, int punkte) {  
        this.text = text;  
        this.punkte = punkte;  
    }  
  
    // Frage auf den Bildschirm ausgeben  
    void frageStellen() {  
        System.out.println(this.text);  
    }  
  
    // Frage beantworten durch Prüfling, Antwort auswerten  
    // und Punkte vergeben  
    abstract void frageBeantworten(Pruefling person);  
  
    int getPunkte() {  
        return this.punkte;  
    }  
}
```

# Vorteile Abstrakter Klassen

- Standard-Verhalten (frageStellen) sowie Standard–Attribute (Text, Punkte) müssen nur einmal implementiert werden (Vererbung)
- Neue Fragetypen später hinzufügen, ohne Klasse Prüfung zu verändern (Polymorphie)

```
class Pruefung {  
    Frage[] fragen;  
  
    void pruefen (Pruefling p) {  
        for (Frage f: fragen) {  
            f.frageStellen(); // dynamisches Binden  
            f.frageBeantworten(p);  
        }  
    }  
}
```

# Aus der Java-Klassenbibliothek

- Klasse Calendar

```
package java.util;  
  
public abstract class Calendar  
    extends Object  
    implements Serializable, Cloneable, Comparable<Calendar>
```

- Klasse GregorianCalendar

```
package java.util;  
  
public class GregorianCalendar  
    extends Calendar
```

## Frage

- Wo könnte man eine abstrakte Klasse in unserer Bank Terminal Anwendung sinnvoll einsetzen?

- Begründen Sie Ihre Wahl

```
src
├── org.hftl.bankterminal.bank
│   └── Bank.java
├── org.hftl.bankterminal.bank.exception
│   ├── IncorrectUserInputException.java
│   ├── InvalidAccountParameterException.java
│   └── UnexpectedStateException.java
├── org.hftl.bankterminal.bank.konto
│   ├── Konto.java
│   └── Sparkonto.java
└── org.hftl.bankterminal.terminal
    └── Terminal.java
```

# Interfaces



# Interfaces

- Ein Interface gibt eine „Vorschrift“ vor, was eine Klasse können muss, die das Interface implementiert
  - Definiert einen Referenzdatentyp ähnlich einer Klasse
  - Deklariert **ausschließlich** Konstanten, abstrakte Instanz-Methoden und Klassenmethoden
- Es lassen sich keine Instanzen eines Interfaces erzeugen
  - Schlüsselwort **interface**
- Klassen können ein oder mehrere Interfaces implementieren
  - Schlüsselwort **implements**
  - Die neue Klasse muss alle abstrakten Methoden der implementierten Interfaces deklarieren (identische Signaturen!) und implementieren
  - Implementierte Methoden von Interfaces sind immer **public**

# Interfaces

```
interface Graphik {  
    public void draw();  
}
```

```
public abstract class Polyeder implements Graphik {  
    int ecken, flaechen;  
  
    int getKanten() { // Eulerscher Polyedersatz  
        return ecken+flaechen-2;  
    }  
}
```

```
public class Wuerfel extends Polyeder {  
  
    public Wuerfel() {  
        ecken = 8;  
        flaechen = 6;  
    }  
    public void draw() {  
        ...  
    }  
}
```

```
class Rechteck implements Graphik {  
    public void draw() {  
        // Implementation  
        ...  
    }  
}
```

# Polymorphie und Dynamisches Binden

- Polymorphie:

- Es lassen sich Objektvariablen vom Typ eines Interfaces definieren, denen Objekte implementierender Klassen zugewiesen werden können

```
Graphik obj1 = new Rechteck(); // Polymorphie  
obj1.draw(); // dynamisches Binden
```

- Dynamisches Binden:

- Zur Compile-Zeit steht nicht fest zu welcher Klasse die Instanz gehört, die in einer Variable gespeichert ist
- Java erfährt erst zur Laufzeit, was tatsächlich in der Objektvariable gespeichert ist

- Eine Klasse kann mehrere Interfaces implementieren

- siehe Beispiel Amphibienfahrzeug

# Beispiel: Fahrzeuge

```
class Amphibienfahrzeug
    extends Fahrzeug
    implements Landfahrzeug,
        Wasserfahrzeug {
    public void fahren() {
        ...
    }
    public void schwimmen() {
        ...
    }
}
```

```
interface Landfahrzeug {
    public void fahren();
}
interface Wasserfahrzeug {
    public void schwimmen();
}
class Fahrzeug {
    Motor m;
    ...
}

class PKW
    extends Fahrzeug
    implements Landfahrzeug {
    public void fahren() {
        ...
    }
}

class MotorBoot
    extends Fahrzeug
    implements Wasserfahrzeug {
    public void schwimmen() {
        ...
    }
}
```

## Weiteres Beispiel: Interfaces

- Ein Internet Zeit-Server informiert seine Klienten in regelmäßigen Abständen über das aktuelle Datum und die aktuelle Zeit
- Wie könnte das Interface aussehen, dass die Klienten implementieren müssen, um dem Server die Kommunikation zu ermöglichen?

## Weiteres Beispiel: Interfaces

```
import java.time.LocalDateTime;

public interface TimeClient {
    void setTime(int hour, int minute, int second);
    void setDate(int day, int month, int year);
    void setDateAndTime(int day, int month, int year, int hour, int minute, int second);
    LocalDateTime getLocalDateTime();
}
```

# Vererbung und Interfaces

```
interface I {  
    public void f();  
    public void g();  
}  
  
interface J extends I {  
    public void h();  
}  
  
class A implements J {  
    public void f() {}  
    public void g() {}  
    public void h() {}  
}  
  
public class ExtInterfaceBeispiel {  
    public static void main(String[] args) {  
        I obj1 = new A();  
        J obj2 = new A();  
    }  
}
```

# Erweiterung von Interfaces

- Erweiterung eines Interfaces um weitere Methoden
  - Alle implementierenden Klassen müssen angepasst werden
- Erweiterung mit **default** löst dieses Problem
  - Implementierung neuer Methoden wird in das Interface gezogen

```
interface Landfahrzeug {  
    public void fahren();  
}  
  
class PKW implements Landfahrzeug {  
    public void fahren() {  
        ...  
    }  
}
```

```
interface Landfahrzeug {  
    public void fahren();  
  
    default double berechneGeschwindigkeit  
    ( double distanz, long zeit ) {  
        return distanz / (double)zeit;  
    }  
}
```



# Aus der Java-Klassenbibliothek

- Interface KeyListener

```
package java.awt.event;

interface KeyListener extends EventListener {
    public void keyPressed(KeyEvent e);
    public void keyReleased(KeyEvent e);
    public void keyTyped(KeyEvent e);
}
```

- GUI Elemente die mit KeyListener interagieren

- Button, Canvas, Checkbox, Choice, Container, Label, List, Scrollbar, TextComponent

# Fragen

1. Was stimmt nicht mit dem folgenden Interface?

```
interface EinInterface {  
    void eineMethode( int aValue ) {  
        System.out.println("Hallo Welt"); }  
}
```

2. Ist das folgende Interface zulässig?

```
public interface Marker { }
```

3. Kann ich eine Methode in einem Interface als **final** deklarieren?

## Fragen

4. Welche Möglichkeiten bietet Java um Generalisierung zu modellieren?
5. Wie unterstützt Java Mehrfach-Vererbung?
  - Warum ist das möglich?

# Abstrakte Klassen und Interfaces

- **Abstrakte Klassen:** abgeleiteten Klassen soll bereits ein bestimmtes Grundverhalten zur Verfügung gestellt werden (→ Vererbung)
  - (Einfach-)Polymorphie
- **Interfaces:** ausschließliche Definition von Konstanten und Methoden-Signaturen, aber kein Verhalten
  - (Mehrfach-)Polymorphie
- **In beiden Fällen:** dynamisches Binden, d.h., zur Compile-Zeit muss nicht feststehen zu welcher Klasse die Instanzen gehören, die in einer Variable gespeichert werden
  - Java kennt zur Compile-Zeit nur das Interface bzw. die abstrakte Klasse

# Abstrakte Klassen und Interfaces

## Verwendung Interfaces wenn...

- Ein bestimmtes Verhalten definiert werden soll, ohne festzulegen, wer es implementiert
- Unabhängige Klassen dasselbe Interface implementieren sollen
- Mehrfachvererbung notwendig ist
- APIs definiert werden sollen
- Schnittstellen innerhalb eines Projektes für parallel Entwicklung benötigt werden

## Verwendung abstrakter Klassen wenn...

- Engverwandte Klassen denselben Code teilen sollen
- Generalisierung das Ziel ist (abgeleitete Klassen haben viele gemeinsame Eigenschaften)
- Zugriffsrechte auf Attribute und Methoden mit `protected` und `private` eingeschränkt werden müssen
- Instanzvariablen sowie dazugehörige Setter/Getter Methoden deklariert werden sollen

# Zusammenfassung

- Abstrakte Klasse: Klasse mit allgemeinen Eigenschaften
  - Methoden werden erst in abgeleiteten Unterklassen implementiert
  - Die Instanziierung abstrakter Klassen ist nicht möglich
- Interface: eine Art Klasse, die ausschließlich Konstanten und abstrakte Instanz-Methoden deklariert
  - Klassen können ein oder mehrere Interfaces implementieren
    - Diese müssen alle (abstrakten) Methoden der implementierten Interfaces definieren und implementieren
  - Die Instanziierung von Interfaces ist nicht möglich

# Kontrollfragen

- Inwiefern sind abstrakte Klassen und Interfaces nützlich, um das Konzept der Kapselung umzusetzen?
- Erläutern Sie das dynamische Binden.
- Geben Sie ein eigenes Beispiel für eine Klassenhierarchie an, die sich nur mit abstrakten Klassen (sinnvoll) realisieren lässt.
- Wann setzen Sie Interfaces oder abstrakte Klassen ein und wann nicht?