JAVA 09

Fortgeschrittene OOP

JAVA 09 – Fortgeschrittene OOP

1. **Vererbung**
2. **Abstrakte Klassen**
3. **Interfaces**
4. **Verschiedenes**

1. Vererbung

1.1. Ableiten mit `extends`

- Eine Klasse kann von einer – **MAXIMAL nur einer** – anderen **abgeleitet** werden
- Dies macht Sinn, wenn ähnliche Dinge sich in nur wenigen Eigenschaften und Methoden unterscheiden
- Dabei **erbt** sie alle Eigenschaften der „**Basisklasse**“

```
public class Animal {
        private int age = 1;
        public int getAge() { return age; }
}

public class Dog extends Animal {}


Dog dog = new Dog();
int ageOfTheDog = dog.getAge(); // Dog hat alle Eigenschaften von Animal
```

1. Vererbung

1.2. Zugriff mit `protected`

- Auf **`private`** Eigenschaften und Methoden, kann **niemand** außerhalb einer Klasse zugreifen – auch keine abgeleitete Klasse!
- Um den Zugriff (**NUR**) für abgeleitete Klassen zu ermöglichen, muss das Schlüsselwort **`protected`** genutzt werden

```
public class Animal {
        private int age = 1;
        protected int getProtectedAge() { return age; }
}

public class Dog extends Animal {
        public int getAge() { return getProtectedAge(); }        // erlaubt
}

Dog dog = new Dog();
int ageOfTheDog = dog.getProtectedAge();          // FEHLER - kein Zugriff
ageOfTheDog = dog.getAge();                       // ganz normal benutzbar
```

1. Vererbung

1.3. Überschreiben von Methoden (1/2)

- Wenn die Methode einer Basisklasse nicht das tut, was die abgeleitete Klasse möchte, kann sie **überschrieben** werden
- Will die abgeleitete Klasse die Methode der Basisklasse aber nutzen, kann sie dies mit dem Schlüsselwort **`super`** tun
- Das gängigste Beispiel ist das **Überschreiben des Konstruktors**:

```
public class Animal {
        private int age;
        public int getAge() { return age; }
        public Animal(int age) {
                this.age = age;
        }
}
public class Dog extends Animal {
        private String name;
        public Dog(int age, String name) {    // Konstruktor-Methode wird überschrieben
                super(age);                   // Aufruf des Basis-Konstruktors
                this.name = name;
        }
}
```

1. Vererbung

1.3. Überschreiben von Methoden (2/2)

- Die abgeleitete Klasse kann alle Methoden überschreiben und trotzdem mittels `super` auf die Basismethode zugreifen
- Außerdem ist es möglich, eine zuvor versteckte (proteced) Methode beim überschreiben zu veröffentlichen (public)!

```
public class Animal {
        private int age = 1;
        protected int getAge() { return age; }
}

public class Dog extends Animal {
        public int getAge() {       // hier wird die Basismethode überschrieben!
                System.out.println(„getAge of Dog-Methode!“);
                return super.getAge();      // hier auf die Basismethode zugegriffen!
        }
}

Dog dog = new Dog();
int ageOfDog = dog.getAge();          // hier gibt es eine Ausgabe über die Konsole
```

1. Vererbung

1.4. Zuweisungskompatibilität

- Bei der Definition einer abgeleiteten Klasse kann die Basisklasse verwendet werden

```
public class Animal {    }
public class Dog extends Animal {        }
public class Cat extends Animal {        }
public class Horse extends Animal {      }

Animal animal;
animal = new Dog();
animal = new Cat();
animal = new Horse();
```

- Ein Animal-Objekt kann also sowohl ein Hund, eine Katze oder ein Pferd sein/werden
- Alle können gleiche Eigenschaften und Methoden von Animal **erben**
- Gleichzeitig können alle abgeleitete Klassen individuelle Eigenschaften und Methoden zusätzlich haben

2. Abstrakte Klassen

2.1. Abstrakte Methoden

- Abstrakte Klassen implementieren bereits einen Teil der Logik
- Noch zu implementierende Logik für abgeleitete Klassen werden durch **abstrakte Methoden** ermöglicht
- Abstrakte Methoden definieren **Parameter** und **Rückgabewert**, aber **keine Logik**. Sie haben keinen „Rumpf“

```
public abstract class Animal {
        private int age;
        public int getAge() { return age; }
        public abstract void move(float distance);
}

pubic class Dog extends Animal {
        public void move(float distance) { ... }       // walk like a dog ...
}

pubic class Snake extends Animal {
        public void move(float distance) { ... }       // walk like a snake ...    }
```

2. Abstrakte Klassen

2.2. Einschränkungen bei der Nutzung

- Eine abstrakte Klasse kann nicht als Typ eines Objekts genutzt werden
- Es können aber Klassenmethoden und Klassenmember genutzt werden

```
public abstract class Animal {
        private int age;
        public int getAge() { return age; }

        public static int getDefaultAge() { return 1; }
}


Animal animal = new Animal();    // FEHLER – ist nicht möglich
int defaultAge = Animal.getDefaultAge();                  // OK
```

3. Interfaces

3.1. Klassen implementieren Interfaces

- Wenn **ausschließlich abstrakte Methoden** in einer Basisklasse sind, kann man auch ein Interface verwenden
- Interfaces geben vor, was eine Klasse zur Verfügung stellen MUSS – Wie sie das macht, ist der Klasse überlassen

```
public interface Lifeform {
        public abstract void feed();
        public abstract void sleep();
}

public class Animal implements Lifeform {
        private int age;
        public int getAge() { return age; }

        public void feed()  { System.out.println(„Animals feed...“); }
        public void sleep() { System.out.println(„Animals sleep...“); }
}
```

3. Interfaces

3.2. Interface vs. Basisklasse – Redundanz versus Abhängigkeit

- Bei einer Basisklasse können immer gleiche Vorgänge einmalig für alle ableitenden Klassen implementiert werden
- Bei Interfaces muss jede Klasse die Vorgänge selbst beschreiben, was **Redundanz** bedeutet (sollte man vermeiden)
- Der Vorteil eines Interfaces bietet sich aber, sobald ein Projekt weiterentwickelt oder gewartet wird

```
public abstract class Animal {
        public void feed()  { System.out.println(„Animals feed...“); }
        public void sleep() { System.out.println(„Animals sleep...“); }
}
```

- ***Ändert sich einmal das Verhalten der Klasse Animal, müsste man alle ableitenden Klassen überprüft werden!***

```
public interface Lifeform {
        public abstract void feed();
        public abstract void sleep();
}
```

- ***Bei einem Interface muss „nur“ die verwendende Klasse angepasst werden, die anders funktionieren soll…***

3. Interfaces

3.3. Abstrakte Klassen und Interfaces zusammen nutzen

- Eine Klasse kann **nur eine Basisklasse** haben, aber **beliebig viele Interfaces** implementieren
- Eine abstrakte Klasse kann, muss aber die Methoden eines Interfaces nicht implementieren

```
public interface Lifeform {
        public abstract void feed();
        public abstract void sleep();
}

public abstract class Animal implements Lifeform {
        public void sleep() { System.out.println(„All Animals sleep...“); }
}

public class Dog extends Animal {
        public void feed()  { System.out.println(„Dogs eat meat and bones...“); }
}
```

3. Interfaces

3.4. Einsatzbeispiel von Interfaces

```
public interface Lifeform {
        public abstract void feed();
}

public class Human implements Lifeform      { ... }    // Human muss feed() implementieren!
public class Animal implements Lifeform     { ... }    // Animal muss feed() implementieren!

public class Nature {
        public void feedLifeform(Lifeform live)  {
                // wir füttern humans und animals, die wissen selbst wies geht!
                live.feed();
        }
}

Nature nature = new Nature();
nature.feedLifeform(new Human());
nature.feedLifeform(new Animal());
```

4. Verschiedenes

Polymorphie

- Das Überschreiben einer Methode kann in ableitenden Klassen, oder auch der Klasse selbst passieren
- Innerhalb der Klasse kann der JAVA-Interpreter anhand der Parameter-Art und/oder Anzahl entscheiden welche genutzt werden soll

```
public class Car {
        int age = 3;
        int defaultSpeed = 50;
        public Car() {  }                              // age ist automatisch 3;
        public Car(int anotherAge) { age = anotherage; }
        public void drive()          { ... }           // fährt mit 50 km/h
        public void drive(int speed)          { ... }  // fährt so schnell wie angegeben

}
Car car = new Car();
Car oldCar = new Car(10);
oldCar.drive();    // fährt mit 50 km/h
car.drive(200);    // fährt 200...
```