Tutorium zur Vorlesung Programmieren

8. Tutorium

Joshua Gleitze

19. Dezember 2016

IPD Koziolek

Beispiellösung

Beispiellösung für das 3. Übungsblatt, inklusive Tests:

joshuagleitze.de/tutorien/programmieren1617/code/Beispiellösung3

Codebeispiele

Codebeispiele, die in diesem Tutorium immer wieder verwendet werden:

joshuagleitze.de/tutorien/programmieren1617/folien/08_Codebeispiele.pdf

Vererbung und Interfaces

Vererbung

Modelliert eine „ist ein“-Beziehung

Wer von einer Klasse erbt, erhält alle Methoden und Attribute der Oberklasse

Konstruktoren werden nicht vererbt

statischer und dynamischer Typ

```
Weapon favouriteWeapon = new Claymore();
```

statischer Typ	`Weapon`
dynamischer Typ	`Claymore`

Überschreiben

Eine Unterklasse kann eine Methode überschreiben

Die Entscheidung, welche Implementierung verwendet wird, basiert auf dem dynamischen Typ

Wird konventionell mit `@Override` markiert.

Verschatten

Eine Unterklasse kann ein Attribut verschatten

Die Entscheidung, welche Implementierung verwendet wird, basiert auf dem statischen Typ

Verschatten

Eine Unterklasse kann eine statische Methode verschatten

Die Entscheidung, welche Implementierung verwendet wird, basiert auf dem statischen Typ

`super`

Analogon zu `this`: Referenziert die Implementierungen der Oberklasse.

- `super(Parameter)` ruft Konstruktor der Oberklasse auf
- `super.methode(Parameter)` ruft Implementierung von `methode` der Oberklasse auf

Konstruktoren

Jeder Konstruktor einer Unterklasse muss in der ersten Anweisung den Konstruktor der Oberklasse aufrufen

Hat die Oberklasse einen Konstruktor ohne Parameter, macht Java das implizit

Interfaces

Ein Interface definiert eine Schnittstelle.
$\rightsquigarrow$ Wer ein Interface implementiert, muss alle Methoden implementieren.

Interfaces: Details

- Alle Methoden sind `public abstract`
- Alle Attribute sind `public static final`
- statische Methoden sind möglich, wenn sie direkt implementiert werden

Interfaces: Details

```java
public interface Program {
    int VERSION = 1;

    String getName();
}
```

$\hat{=}$

```java
public interface Program {
    public static final int VERSION = 1;

    public abstract String getName();
}
```

Beispiel

```
Claymore claymore = new Claymore();
System.out.println(claymore.getPrice());
```

Beispiel

```
Claymore claymore = new Claymore();
System.out.println(claymore.getPrice());
```

```
3136
```

Beispiel

```
Katana katana = new Katana();
System.out.println(katana.getPrice());
```

Beispiel

```
Katana katana = new Katana();
System.out.println(katana.getPrice());
```

```
5040
```

Beispiel

```
Sword sword = new Katana();
System.out.println(sword.getPrice());
```

Beispiel

```
Sword sword = new Katana();
System.out.println(sword.getPrice());
```

```
5040
```

Beispiel

```
Weapon weapon = new Claymore();
System.out.println(weapon.getPrice());
```

Beispiel

```
Weapon weapon = new Claymore();
System.out.println(weapon.getPrice());
```

```
3136
```

Beispiel

```
System.out.println(Katana.PRICE_FACTOR);
```

Beispiel

```
System.out.println(Katana.PRICE_FACTOR);
```

```
3
```

Beispiel

```
System.out.println(Claymore.PRICE_FACTOR);
```

Beispiel

```
System.out.println(Claymore.PRICE_FACTOR);
```

```
2
```

Beispiel

```java
public class SimpleSword extends Sword {
    protected int age = 2;

    public SimpleSword(Material Material, int length) {
        super(Material, length);
    }

    public int getDamage() {
        return this.material.getPrice();
    }
}
```

```java
SimpleSword woodenSword = new SimpleSword(Material.WOOD, 4);
System.out.println(woodenSword.getAge());
```

Beispiel

```java
public class SimpleSword extends Sword {
    protected int age = 2;

    public SimpleSword(Material Material, int length) {
        super(Material, length);
    }

    public int getDamage() {
        return this.material.getPrice();
    }
}
```

```java
SimpleSword woodenSword = new SimpleSword(Material.WOOD, 4);
System.out.println(woodenSword.getAge());
```

```
0
```

Interface – abstrakte Klasse – Klasse

Interface

- Möglichst klein: Genau eine Aufgabe
- Eine Klasse kann mehrere Interfaces implementieren
- Repräsentiert gewisse zugesicherte Eigenschaften
  - Endet deshalb oft auf „-able“
  - `Iterable`, `Comparable`, `Serializable`, `Closeable`, `Runnable`, `Cloneable`

Interface: Beispiel

java.lang.Comparable (OpenJDK 8)

```
96  public interface Comparable<T> {

        ...

136     public int compareTo(T o);
137 }
```

abstrakte Klasse

- Nicht sinnvoll instanziierbar
- Implementiert gemeinsame Funktionalität für Unterklassen
    - Diese kann oft durch abstrakte „Schablonenmethoden“ parametrisiert werden

abstrakte Klasse: Beispiel

java.util.AbstractCollection (OpenJDK 8)

```
61  public abstract class AbstractCollection<E> implements
      ↪  Collection<E> {

        ...

78      public abstract int size();

        ...

85      public boolean isEmpty() {
86          return size() == 0;
87      }
```

Klasse

- Kann sinnvoll instanziiert werden

Vererbungslehre

Interfaces

- Können von mehreren anderen Interfaces erben

```
public interface ConcurrentNavigableMap<K,V> extends
 ↪  ConcurrentMap<K,V>, NavigableMap<K,V>
```

Abstrakte Klassen

- Können von höchstens einer (ggf. abstrakten) Klasse erben
- Können mehrere Interfaces implementieren

```
public abstract class AbstractQueue<E> extends
 ↪  AbstractCollection<E> implements Queue<E>
```

Vererbungslehre

Klassen

- Können von höchstens einer (ggf. abstrakten) Klasse erben
- Können mehrere Interfaces implementieren
- Müssen alle nicht implementierten Methoden implementieren

```
public class ArrayList<E> extends AbstractList<E> implements
 ↪  List<E>, RandomAccess, Cloneable, java.io.Serializable
```

Verwendung

Für die statischen Typen von Rückgabewerten und formalen Parametern sind immer Interfaces vorzuziehen!

Details zu Methoden

Überladen

Gibt es von einer Methode mit dem selben Namen mehrere Versionen mit verschiedenen formalen Parametern, so nennt man diese Methode *überladen*.

java.util.List

```
E remove(int item);
boolean remove(Object o);
```

Überladen: Auswahl

Die Variante überladener Methoden wird anhand der statischen Typen ausgewählt.

Beispiel: Überladen

```
1  public class SwordMerchant {
2      public int priceForWeapon(Weapon weapon) {
3          return weapon.getPrice() / 2;
4      }
5
6      public int priceForWeapon(Sword weapon) {
7          return weapon.getPrice() - 40;
8      }
9  }
```

Beispiel: Überladen

```java
public class SwordMerchant {
    public int priceForWeapon(Weapon weapon) {
        return weapon.getPrice() / 2;
    }
    public int priceForWeapon(Sword weapon) {
        return weapon.getPrice() - 40;
    }
}
```

```java
SwordMerchant merchant = new SwordMerchant();
Sword myBestWeapon = new Katana();
System.out.println(merchant.priceForWeapon(myBestWeapon));
```

Beispiel: Überladen

```
1  public class SwordMerchant {
2      public int priceForWeapon(Weapon weapon) {
3          return weapon.getPrice() / 2;
4      }
5      public int priceForWeapon(Sword weapon) {
6          return weapon.getPrice() - 40;
7      }
8  }
```

```
SwordMerchant merchant = new SwordMerchant();
Sword myBestWeapon = new Katana();
System.out.println(merchant.priceForWeapon(myBestWeapon));
```

```
5000
```

Beispiel: Überladen

```java
public class SwordMerchant {
    public int priceForWeapon(Weapon weapon) {
        return weapon.getPrice() / 2;
    }
    public int priceForWeapon(Sword weapon) {
        return weapon.getPrice() - 40;
    }
}
```

```java
SwordMerchant merchant = new SwordMerchant();
Weapon myBestWeapon = new Katana();
System.out.println(merchant.priceForWeapon(myBestWeapon));
```

Beispiel: Überladen

```java
public class SwordMerchant {
    public int priceForWeapon(Weapon weapon) {
        return weapon.getPrice() / 2;
    }
    public int priceForWeapon(Sword weapon) {
        return weapon.getPrice() - 40;
    }
}
```

```java
SwordMerchant merchant = new SwordMerchant();
Weapon myBestWeapon = new Katana();
System.out.println(merchant.priceForWeapon(myBestWeapon));
```

```
2520
```

Beispiel: Überladen

```
1  public class SwordMerchant {
2      public boolean willTradeWeapons(Weapon giveAway, Weapon getNew) {
3          return getNew.getPrice() > giveAway.getPrice();
4      }
5
6      public boolean willTradeWeapons(Weapon giveAway, Sword getNew) {
7          return true;
8      }
9
10     public boolean willTradeWeapons(Sword giveAway, Weapon getNew) {
11         return false;
12     }
13 }
```

Beispiel: Überladen

```
1  public class SwordMerchant {
2      public boolean willTradeWeapons(Weapon giveAway, Weapon getNew) {
3          return getNew.getPrice() > giveAway.getPrice();
4      }
5      public boolean willTradeWeapons(Weapon giveAway, Sword getNew) {
6          return true;
7      }
8      public boolean willTradeWeapons(Sword giveAway, Weapon getNew) {
9          return false;
10     }
11 }
```

```
final SwordMerchant merchant = new SwordMerchant();
final Weapon myBestSword = new Katana();
final Weapon weaponIWant = new Claymore();
System.out.println(
    merchant.willTradeWeapons(weaponIWant, myBestSword));
```

Beispiel: Überladen

```
1  public class SwordMerchant {
2      public boolean willTradeWeapons(Weapon giveAway, Weapon getNew) {
3          return getNew.getPrice() > giveAway.getPrice();
4      }
5      public boolean willTradeWeapons(Weapon giveAway, Sword getNew) {
6          return true;
7      }
8      public boolean willTradeWeapons(Sword giveAway, Weapon getNew) {
9          return false;
10     }
11 }
```

```
final SwordMerchant merchant = new SwordMerchant();
final Weapon myBestSword = new Katana();
final Weapon weaponIWant = new Claymore();
System.out.println(
    merchant.willTradeWeapons(weaponIWant, myBestSword));
```

```
true
```

Beispiel: Überladen

```
1  public class SwordMerchant {
2      public boolean willTradeWeapons(Weapon giveAway, Weapon getNew) {
3          return getNew.getPrice() > giveAway.getPrice();
4      }
5      public boolean willTradeWeapons(Weapon giveAway, Sword getNew) {
6          return true;
7      }
8      public boolean willTradeWeapons(Sword giveAway, Weapon getNew) {
9          return false;
10     }
11 }
```

```
final SwordMerchant merchant = new SwordMerchant();
final Weapon myBestSword = new Katana();
final Sword weaponIWant = new Claymore();
System.out.println(
    merchant.willTradeWeapons(weaponIWant, myBestSword));
```

Beispiel: Überladen

```
1  public class SwordMerchant {
2      public boolean willTradeWeapons(Weapon giveAway, Weapon getNew) {
3          return getNew.getPrice() > giveAway.getPrice();
4      }
5      public boolean willTradeWeapons(Weapon giveAway, Sword getNew) {
6          return true;
7      }
8      public boolean willTradeWeapons(Sword giveAway, Weapon getNew) {
9          return false;
10     }
11 }
```

```
final SwordMerchant merchant = new SwordMerchant();
final Weapon myBestSword = new Katana();
final Sword weaponIWant = new Claymore();
System.out.println(
    merchant.willTradeWeapons(weaponIWant, myBestSword));
```

```
false
```

Beispiel: Überladen

```
1  public class SwordMerchant {
2      public boolean willTradeWeapons(Weapon giveAway, Weapon getNew) {
3          return getNew.getPrice() > giveAway.getPrice();
4      }
5      public boolean willTradeWeapons(Weapon giveAway, Sword getNew) {
6          return true;
7      }
8      public boolean willTradeWeapons(Sword giveAway, Weapon getNew) {
9          return false;
10     }
11 }
```

```
final SwordMerchant merchant = new SwordMerchant();
final Sword myBestSword = new Katana();
final Sword weaponIWant = new Claymore();
System.out.println(
    merchant.willTradeWeapons(weaponIWant, myBestSword));
```

Beispiel: Überladen

```java
public class SwordMerchant {
    public boolean willTradeWeapons(Weapon giveAway, Weapon getNew) {
        return getNew.getPrice() > giveAway.getPrice();
    }
    public boolean willTradeWeapons(Weapon giveAway, Sword getNew) {
        return true;
    }
    public boolean willTradeWeapons(Sword giveAway, Weapon getNew) {
        return false;
    }
}
```

```java
final SwordMerchant merchant = new SwordMerchant();
final Sword myBestSword = new Katana();
final Sword weaponIWant = new Claymore();
System.out.println(
    merchant.willTradeWeapons(weaponIWant, myBestSword));
```

```
error: reference to willTradeWeapons is ambiguous
        System.out.println(merchant.willTradeWeapons(weaponIWant, myBestSword));
                                    ^
  both method willTradeWeapons(Weapon,Sword) in SwordMerchant and method
    ↪   willTradeWeapons(Sword,Weapon) in SwordMerchant match
```

Das waren Beispiele – so nicht programmieren!

„Richtige“ Lösung: Entwurfsmuster „Besucher“, siehe Vorlesung Softwaretechnik I

default-Methoden

default

In Java 8 können Interfaces auch Methoden implementieren.

```
1  public interface Food {
2      int getNutritiveValue();
3
4      default int getPrice() {
5          return this.getNutritiveValue() * 5;
6      }
7  }
```

default

In Java 8 können Interfaces auch Methoden implementieren.

```
1  public class Apple implements Food {
2      public int getNutritiveValue() {
3          return 1;
4      }
5  }
```

```
new Apple().getPrice(); // 5
```

default

In Java 8 können Interfaces auch Methoden implementieren.

Problem?

default

In Java 8 können Interfaces auch Methoden implementieren.

```
1  public interface Weapon {
2      int getDamage();
3
4      default int getPrice() {
5          return this.getDamage() * 10;
6      }
7  }
```

default

In Java 8 können Interfaces auch Methoden implementieren.

```
1  public class PoisonousApple implements Food, Weapon {
2      public int getNutritiveValue() {
3          return 1;
4      }
5
6      public int getDamage() {
7          return 2;
8      }
9  }
```

default

In Java 8 können Interfaces auch Methoden implementieren.

```
1  public class PoisonousApple implements Food, Weapon {
2      public int getNutritiveValue() {
3          return 1;
4      }
5
6      public int getDamage() {
7          return 2;
8      }
9  }
```

```
error: class PoisonousApple inherits unrelated defaults for getPrice() from types
  ↪   Weapon and Food
```

default

In Java 8 können Interfaces auch Methoden implementieren.

```
1  public class PoisonousApple implements Food, Weapon {
2      public int getNutritiveValue() {
3          return 1;
4      }
5
6      public int getDamage() {
7          return 2;
8      }
9
10     public int getPrice() {
11         return 20;
12     }
13 }
```

default

In Java 8 können Interfaces auch Methoden implementieren.

```
1  public class PoisonousApple implements Food, Weapon {
2      public int getNutritiveValue() {
3          return 1;
4      }
5
6      public int getDamage() {
7          return 2;
8      }
9
10     public int getPrice() {
11         return Weapon.super.getPrice();
12     }
13 }
```

default

In Java 8 können Interfaces auch Methoden implementieren.

```
1  public class PoisonousApple extends Apple implements Weapon {
2      public int getNutritiveValue() {
3          return 1;
4      }
5
6      public int getDamage() {
7          return 2;
8      }
9
10     public int getPrice() {
11         return super.getPrice();
12     }
13 }
```

Generische Typen

Generische Typen

Methoden und Klassen können Typvariablen deklarieren.

Begriffe

Begriff	Beispiel
Generischer Typ *(generic type)*	`List<T>`
Typvariable / formaler Typparameter *(formal type parameter)*	`T`
Parametrisierter Typ *(Parameterized type)*	`List<String>`
Typparameter *(actual type paramter)*	`String`
Originaltyp *(raw type)*	`List`

Diamant-Operator

Kurzzschreibweise, falls die Typparameter aus dem Kontext klar sind.

```
Map<Person, Set<Person>> siblings = new HashMap<Person,
 ↪  Set<Person>>();
```

Diamant-Operator

Kurzzschreibweise, falls die Typparameter aus dem Kontext klar sind.

```
Map<Person, Set<Person>> siblings = new HashMap<>();
```

Typeinschränkungen

`extends X` gibt an, dass eine Typvariable nur mit Typen, die `X` sind oder von `X` erben, belegt werden kann.

`super X` gibt an, dass eine Typvariable nur mit Typen, die `X` sind oder von denen `X` erbt, belegt werden kann.

java.util.Arrays

```
static <T extends Comparable<? super T>> void parallelSort(T[]
 ↪  a)
```