

Javakurs für Anfänger

Einheit 13: Interfaces

Lorenz Schauer

Lehrstuhl für Mobile und Verteilte Systeme



1. Teil: Interfaces

- Motivation
- Eigenschaften
- Besonderheiten
- Anonyme Klassen
- Lambda-Ausdrücke

Praxis:

- Übungen zu Interfaces

Lernziele

- Interfaces und deren Bedeutung kennenlernen
- Interfaces schreiben und benutzen können

Warum brauchen wir Interfaces?

- Beschreibung von Schnittstellen
- Prinzip: Trennung von **Spezifikation** und **Implementierung**
 - Interfaces kümmern sich um das **WAS**, aber nicht um das **WIE**
- Schutz der eigenen Implementierung vor Dritten
 - Kommunikationsschnittstelle
- Alternative zur Mehrfachvererbung
 - Eine konkrete Klasse kann mehrere Interfaces implementieren
- Polymorphismus

Eigenschaften von Interfaces:

- Ähnlich zu (abstrakten) Klassen
 - Können Konstanten, abstrakte Methoden und innere Schnittstellen und Klassen, sowie statische Methoden enthalten
- Können **nicht instanziiert** werden (kein Aufruf mit **new** möglich!)
 - Besitzen demnach auch **keinen Konstruktor**
- Können andere Interfaces erweitern (Hierarchie)

Weitere Eigenschaften:

- Variablen (also Konstanten) sind implizit `public`, `static` und `final`.
- Methoden sind implizit `public` und `abstract`
 - Muss nicht extra angegeben sein
 - `Private` oder `Protected` sind in Interfaces verboten, `Static` hingegen ist erlaubt.
- Implementierende Klassen **müssen alle Methoden** implementieren
 - Ansonsten muss die implementierende Klasse als `abstract` deklariert sein!

// Beispiel: Auto-Interface

```
public interface IAuto {  
  
    // Konstanten  
    public final int anzahlRaeder = 4;  
    String name = "Auto Interface";  
  
    // Methoden  
    public abstract void fahren();  
    void bremsen();  
}
```

// Beispiel: Auto-Implementierung

```
public class AutoImpl implements IAuto  
{
```

```
    @Override  
    public void fahren() {  
        // TODO Auto-generated method stub  
    }  
  
    @Override  
    public void bremsen() {  
        // TODO Auto-generated method stub  
    }  
}
```

Erzeugen Sie ein neues Eclipse-Projekt „Uebung13“ und schreiben Sie:

- Ein Interface mit dem Namen `IHund`, das folgende Methoden zur Verfügung stellt:
 - `void bellen();`
 - `String doAction(int action);`
- Eine Implementierende Klasse `HundImpl`, welche die beiden Methoden implementieren muss:
 - Ein Hund hat eine Größe (`int`), welche beim Erzeugen angegeben werden muss
 - `bellen()` gibt „wuff“ auf der Konsole aus, falls der Hund kleiner ist als 100, andernfalls wird „wau“ ausgegeben
 - `String doAction(int action)` gibt je nach angegebener Action den jeweiligen String zurück:
 - 0: „schlafen“, 1: „gehen“, 2: „laufen“

Erzeugen Sie nun in der Main-Methode 2 Hunde von unterschiedlicher Größe und lassen Sie diese bellen und eine Aktion ausführen.

Interfaces können andere Interfaces erweitern

- mittels **extends**
- Besonderheit: Multi-Erweiterung möglich
 - Bei Klassen ist die Mehrfachvererbung in Java nicht möglich!

```
// Beispiel für Interface und Erweiterungen
```

```
//Filename: Sports.java
public interface Sports
{
    public void setHomeTeam(String name);
    public void setVisitingTeam(String name);
}
```

```
//Filename: Event.java
import java.util.Date;
public interface Event
{
    setTitel(String title);
    setDate(Date d);
}
```

```
//Filename: Football.java
public interface Football extends Sports
{
    public void homeTeamScored(int points);
    public void visitingTeamScored(int points);
    public void endOfQuarter(int quarter);
}
```

```
//Filename: Hockey.java
public interface Hockey extends Sports, Event
{
    public void homeGoalScored();
}
```

Eine Klasse, welche das Interface Football implementiert muss **alle** Methoden implementieren, auch die des Basisinterfaces: Also insg. 5

Eine Klasse, welche das Interface Hockey implementiert muss auch alle Methoden implementieren:
Also insg. $1+2+2 = 5$

Tagging Interfaces (dt.: Markierungsschnittstellen)

- **Leere** Interfaces (Ohne Methoden)
 - Bsp.: `public interface EventListener{} (aus java.util)`
- Werden oft durch andere Interfaces erweitert
- Markieren Zusammenhänge zwischen Klassen
 - Mittels einem gemeinsamen Vorgänger
 - Bspw.: wird der `EventListener` von vielen Interfaces (Mouse, Tastatur,...) erweitert, welche die JVM gleich als Event Delegation erkennen kann
 - Durch hinzufügen des gleichen Datentyps
 - Polymorphismus erlaubt uns den Datentyp der Oberklasse zu verwenden
 - Bsp.: `EventListener e = new MouseListener() {...}`

Functional Interfaces (dt. Funktionsinterface)

- Besitzen **genau eine** Methode
 - Bsp.: `Runnable`, `Callable`, `Comparator`
- Können einfach mit Lambda-Ausdrücken implementiert werden
 - Seit Java 8 möglich (kommt später)

Anonyme Klassen:

- Haben keinen Namen und erzeugen immer automatisch ein Objekt
 - Klassendeklaration und Objekterzeugung sind zu einem Sprachkonstrukt verbunden
- Keine zusätzlichen `extends`- oder `implements` Angaben erlaubt!
- Kein eigener Konstruktor
- Nur Objektmethoden und finale statische Variablen erlaubt
 - Allg. Syntax:
 - `new KlasseOderSchnittstelle() { /*Eigenschaften der inneren Klasse */ }`

new KlassenName

- Unterklasse von KlassenName
- Argumente für Konstruktor von KlassenName evtl. nötig

new InterfaceName

- Unterklasse von Object
- Implementiert InterfaceName, also alle Methoden müssen implementiert werden

Eigenschaften der inneren Klasse

- Methoden
- Attribute


```
// Anonyme Klasse zum beerben von java.awt.Point
```

```
Point p = new Point( 10, 12 ) {  
    @Override  
    public String toString() {  
        return "(" + x + "," + y + ")";  
    }  
};
```

Anonyme Klasse:
Erbt von Point
Überschreibt Methode
`toString()`

```
System.out.println( p );  
// Ausgabe: (10,12)
```

; nicht vergessen!

```
// Anonyme Klassen zum implementieren von Interfaces
```

```
public static void main(String[] args){  
    IAuto auto = new IAuto() {  
        @Override  
        public void fahren() {  
            // TODO Auto-generated method stub  
        }  
        @Override  
        public void bremsen() {  
            // TODO Auto-generated method stub  
        }  
    };  
}
```

Anonyme Klasse:
Implementiert Interface
`IAuto` erbt von `Object`

; nicht vergessen!

Gängiges Beispiel für anonyme innere Klassen mit Interfaces sind **Threads** (für nebenläufige Programmierung)

- Die Klasse **Thread** kann im Konstruktor eine Implementierung des Interface **Runnable** übergeben bekommen
 - Interface **Runnable** hat eine Methode **run()**, welche beim Starten des Threads den entsprechenden Code parallel ausführen kann.

```
// Gängiges Beispiel für Implementierung von Runnable und Threads:
```

```
public void myParallelOperation(){  
    Runnable myRun = new Runnable() {  
        @Override  
        public void run() {  
            for(int i=0; i<10; i++)  
                System.out.println("In Zeile: "+i);  
        }  
    };  
    Thread t = new Thread(myRun);  
    t.start();  
}
```

Schreiben Sie nun Ihr Programm (Main-Methode) aus der vorherigen Aufgabe so um, dass Sie direkt mittels anonymer Klasse und dem Interface IHund einen Hund erzeugen, der anschließend bellt.

Ihre Implementierende Klasse `HundImpl` aus der vorherigen Aufgabe sollen Sie hier also nicht verwenden.

- Was ist dabei der Nachteil?
- Wie muss mit der Instanzvariablen `groesse` umgegangen werden?

Lambda Ausdrücke wurden in Java 8 eingeführt:

- Ersetzen in den meisten Fällen anonyme Klassen
- Führen zu einfacherem Code
- Hängen direkt mit Funktionsinterfaces zusammen
 - Interface mit genau einer Methode
 - Bsp.: `Runnable`
 - 1 Methode `run()`;

Allgemeine Syntax:

- `(Parameter) -> {Body}`
- Passt zu jedem Funktionsinterface, dessen einzige Methode die passende Parameterliste und den passenden Ergebnistyp verlangt.
- Bsp.:
 - `() -> {System.out.println(„Servus Lambda!“);}`
 - Ist kompatibel zum Interface `Runnable`, da kein Parameter und kein Ergebnis verlangt wird

Haben wir also ein Funktionsinterface (genau eine Methode), können wir die Implementierung des Interfaces mit minimalem Code-Aufwand erreichen:

```
// Funktionsinterface:
```

```
public interface MyFunctionalInterface {  
    String outPut(int number);  
}
```

```
// Implementierung mittels Lambda-Ausdruck:
```

```
public static void main(String[] args) {  
  
    MyFunctionalInterface i = (int number)->{  
        String output = "The number returned is: "+number;  
        return output;  
    };  
  
    System.out.println(i.outPut(23));  
}
```

Schreiben Sie nun ein weiteres Interface „DooferHund“ mit nur einer Methode:

- `void bellen(int groesse);`

Schreiben Sie nun Ihr Programm (Main-Methode) aus der vorherigen Aufgabe so um, dass Sie einen „doofenHund“ erzeugen, welcher mittels Lambda Ausdruck das Interface direkt implementiert.

- Ab einer Größe von über 100 wird „wau“ ausgegeben, sonst „wuff“
- Lassen Sie Ihren doofen Hund mit der entsprechenden Größe wieder bellen

Wir haben nun Interfaces kennengelernt:

- Motivation:
 - Beschreibung von Schnittstellen
 - Prinzip: Trennung von Spezifikation und Implementierung
- Eigenschaften:
 - Keine Erzeugung mittels `new`
 - Abstrakte Methoden, welche von einer konkreten Klasse implementiert werden müssen
- Implementierungen:
 - Eigene Klasse implements Interface:
 - `AutoImpl implements IAuto{ ... }`
 - Mittels anonymer innerer Klassen:
 - `IAuto a = new IAuto(){...};`
 - Mittels Lambda-Ausdrücke (Bei Funktionsinterfaces)
 - `IAutoFunctional af = () -> {...};`