Lambda-Ausdrücke und funktionale Interfaces

Carsten Gips (FH Bielefeld)

Unless otherwise noted, this work is licensed under CC BY-SA 4.0.

Problem: Sortieren einer Studi-Liste

```
List<Studi> sl = new ArrayList<>();

// Liste sortieren?
sl.sort(???);  // Parameter: java.util.Comparator<Studi>
```

Problem: Sortieren einer Studi-Liste

```
List<Studi> sl = new ArrayList<>();

// Liste sortieren?
sl.sort(???);  // Parameter: java.util.Comparator<Studi>
```

```
public class MyCompare implements Comparator<Studi> {
    @Override  public int compare(Studi o1, Studi o2) {
        return o1.getCredits() - o2.getCredits();
    }
}
```

```
// Liste sortieren?
MyCompare mc = new MyCompare();
sl.sort(mc);
```

Lösung: Comparator als anonyme innere Klasse

```
List<Studi> sl = new ArrayList<>();

// Parametrisierung mit anonymer Klasse
sl.sort(
        new Comparator<Studi>() {
            @Override
            public int compare(Studi o1, Studi o2) {
                return o1.getCredits() - o2.getCredits();
            }
        });  // Semikolon nicht vergessen!!!
```

Demo: nested.DemoAnonymousInnerClass

Vereinfachung mit Lambda-Ausdruck

```
List<Studi> sl = new ArrayList<>();

// Parametrisierung mit anonymer Klasse
sl.sort(
        new Comparator<Studi>() {
            @Override
            public int compare(Studi o1, Studi o2) {
                return o1.getCredits() - o2.getCredits();
            }
        });  // Semikolon nicht vergessen!!!


// Parametrisierung mit Lambda-Ausdruck
sl.sort( (Studi o1, Studi o2) -> o1.getCredits() - o2.getCredits() );
```

Hinweis auf **funktionales Interface**

Demo: nested.DemoLambda

Syntax für Lambdas

```
(Studi o1, Studi o2) -> o1.getCredits() - o2.getCredits()
```

Varianten:

- `(parameters) -> expression`
- `(parameters) -> { statements; }`

Quiz: Welches sind keine gültigen Lambda-Ausdrücke?

1. `() -> {}`
2. `() -> "wuppie"`
3. `() -> { return "fluppie"; }`
4. `(Integer i) -> return i + 42;`
5. `(String s) -> { "foo"; }`
6. `(String s) -> s.length()`
7. `(Studi s) -> s.getCredits() > 300`
8. `(List<Studi> sl) -> sl.isEmpty()`
9. `(int x, int y) -> { System.out.println("Erg: "); System.out.println(x+y); }`
10. `() -> new Studi()`
11. `s -> s.getCps() > 100 && s.getCps() < 300`
12. `s -> { return s.getCps() > 100 && s.getCps() < 300; }`

Definition “Funktionales Interface” (“*functional interfaces*”)

```java
@FunctionalInterface
public interface Wuppie<T> {
    int wuppie(T obj);
    boolean equals(Object obj);
    default int fluppie() { return 42; }
}
```

`Wuppie<T>` ist ein **funktionales Interface** (“*functional interface*”)

- Hat **genau *eine* abstrakte Methode**
- Hat evtl. weitere Default-Methoden
- Hat evtl. weitere abstrakte Methoden, die `public` Methoden von `java.lang.Object` überschreiben

Quiz: Welches ist kein funktionales Interface?

```java
public interface Wuppie {
    int wuppie(int a);
}

public interface Fluppie extends Wuppie {
    int wuppie(double a);
}

public interface Foo {
}

public interface Bar extends Wuppie {
    default int bar() { return 42; }
}
```

Lambdas und funktionale Interfaces: Typprüfung

```
interface java.util.Comparator<T> {
    int compare(T o1, T o2);    // abstrakte Methode
}
```

```
// Verwendung ohne weitere Typinferenz
Comparator<Studi> c1 = (Studi o1, Studi o2) -> o1.getCredits() - o2.getCredits();

// Verwendung mit Typinferenz
Comparator<Studi> c2 = (o1, o2) -> o1.getCredits() - o2.getCredits();
```

Wrap-Up

- Anonyme Klassen: “Wegwerf”-Innere Klassen
  - Müssen Interface implementieren oder Klasse erweitern
- Java8: **Lambda-Ausdrücke** statt anonymer Klassen (**funktionales Interface nötig**)
  - Zwei mögliche Formen:
    - Form 1: `(parameters) -> expression`
    - Form 2: `(parameters) -> { statements; }`
  - Im jeweiligen Kontext muss ein **funktionales Interface** verwendet werden, d.h. ein Interface mit **genau** einer abstrakten Methode
  - Der Lambda-Ausdruck muss von der Signatur her dieser einen abstrakten Methode entsprechen

LICENSE

Unless otherwise noted, this work is licensed under CC BY-SA 4.0.