2.7 Übungen zu Lambdas, Methodenreferenzen und Defaultmethoden

Lernziel: Es sollen die Basisbausteine der funktionalen Programmierung, also Functional Interfaces, SAM-Typen, Lambdas, Methodenreferenzen und Defaultmethoden, anhand von einfachen Beispielen kennengelernt werden.

✍ Aufgabe 1 ✍

Schauen Sie sich das folgende Interface `LongBinaryOperator` an:

```
public interface LongBinaryOperator
{
   long applyAsLong(final long left, final long right);
}
```

Nachfolgend sind einige Lambdas gezeigt. Welche davon sind gültig? Wofür erhält man Kompilierfehler?

```
final LongBinaryOperator v1 = (long x, Long y) -> { return x + y; };
final LongBinaryOperator v2 = (long x, long y) -> { return x + y; };
final LongBinaryOperator v3 = (long x, long y) -> x + y;
final LongBinaryOperator v4 = (long x, y) -> x + y;
final LongBinaryOperator v5 = (x, y) -> x + y;
final LongBinaryOperator v6 = x, y -> x + y;
```

Tipp: Überlegen Sie etwas und prüfen Sie Ihre Vermutungen dann mithilfe der IDE.

✍ Aufgabe 2 ✍

Aufgabe 2a: Schreiben Sie ein `Runnable` mit einem Lambda, der "`I say hello lambda world`" auf der Konsole ausgibt. Nutzen Sie dazu einen `Thread` wie folgt:

```
final Runnable runner = // ... TODO ...
new Thread(runner).start();
```

Aufgabe 2b: Die zuvor erstellte Konsolenausgabe soll einen variablen Anteil bekommen. Wie kann man Werte oder Informationen an den Lambda übergeben?

Tipp 1: Greifen Sie auf eine außerhalb des Lambdas definierte Variable `messageText` zu, um statt des Worts "`world`" einen beliebigen anderen Text auszugeben. Was ist für lokale Variablen zu bedenken?

Tipp 2: Nutzen Sie eine Methode `Runnable provideTextOuptut(String)` und einen Lambda vom Typ `Runnable`, um eine Nachricht konfigurierbar zu gestalten.

✍ Aufgabe 3 ✍

Vereinfachen Sie die zwei im nachfolgenden Listing gezeigten `Comparator<String>`-Implementierungen, die auf JDK-7 basieren und a) nach Länge und b) case-insensitive sortieren. Vereinfachen Sie das Ganze durch den Einsatz von Lambdas.

```
final List<String> names = Arrays.asList("Josef", "JÖrg", "Jürgen");

final Comparator<String> byLength = new Comparator<String>()
{
    @Override
    public int compare(final String str1, final String str2)
    {
        return Integer.compare(str1.length(), str2.length());
    }
};
final Comparator<String> caseInsensitive = new Comparator<String>()
{
    @Override
    public int compare(final String str1, final String str2)
    {
        return str1.compareToIgnoreCase(str2);
    }
};

Collections.sort(names, byLength);
System.out.println(names);

Collections.sort(names, caseInsensitive);
System.out.println(names);
```

Tipp: Definieren Sie zwei Instanzen von Komparatoren wie folgt:

```
final Comparator<String> byLengthJDK8 = (str1, str2) -> // ... TODO ...
final Comparator<String> caseInsensitiveJDK8 = (str1, str2) -> // ... TODO ...
```

✍ Aufgabe 4 ✍

Rekapitulieren Sie die neuen Begrifflichkeiten.

Aufgabe 4a: Was bedeutet SAM? Was ist ein Functional Interface? Was haben diese mit Lambdas zu tun?

Aufgabe 4b: Wie fügen sich Lambdas und Methodenreferenzen zusammen? Was versteht man unter Defaultmethoden? Was sollte man bei diesen als Applikationsentwickler bedenken?

✍ Aufgabe 5 ✍

Konvertieren Sie die nachfolgend gezeigten, anonymen Klassen vom Typ `FileFilter` in Lambdas. Gegeben sind die zwei Implementierungen `directoryFilter` und `pdfFileFilter`, die gemäß ihrem Namen a) Verzeichnisse und b) PDF-Dateien filtern.

```
final FileFilter directoryFilter = new FileFilter()
{
    @Override
    public boolean accept(final File pathname)
    {
        return pathname.isDirectory();
    }
};

final FileFilter pdfFileFilter = new FileFilter()
{
    @Override
    public boolean accept(final File pathname)
    {
        return (pathname.isFile() &&
                pathname.getName().toLowerCase().endsWith(".pdf"));
    }
};
```

Tipp: Verwenden Sie für den ersten `FileFilter` eine Methodenreferenz auf die Methode `isDirectory()` und für die zweite Umsetzung einen Lambda.

✍ Aufgabe 6 ✍

Die gezeigte Klasse `Conflict` implementiert die beiden Interfaces `IF1` und `IF2`, die jeweils die Defaultmethode `sameMethod()` besitzen. Lösen Sie den Konflikt auf.

```
public class Conflict implements IF1, IF2
{
    public static void main(final String[] args)
    {
        new Conflict().sameMethod();
    }
    // TODO: sameMethod()...
}

public interface IF1
{
    default void sameMethod()
    {
        System.out.println("IF1");
    }
}

public interface IF2
{
    default void sameMethod()
    {
        System.out.println("IF2");
    }
}
```

Tipp: Realisieren Sie dazu eine Methode `sameMethod()` mit

1. einer eigenen Implementierung und
2. einem Aufruf einer der Methoden aus `IF1` oder `IF2`.

✍ Aufgabe 7 ✍

Überlegen Sie sich eine einfache grafische Notation (Kästchen und Pfeile) für die folgenden drei Functional Interfaces mit Ein- und Ausgaben:

```
public interface Predicate<T>
{
    boolean test(T t)
}

public interface BinaryOperator<T>
{
    T apply(T t1, T t2);
}

public interface Comparator<T>
{
    int compare(T t1, T t2);
}
```

Tipp: Das Functional Interface `Function<T,R>`

```
public interface Function<T, R>
{
    R apply(T t)
}
```

könnte man etwa wie in Abbildung 2-1 darstellen.

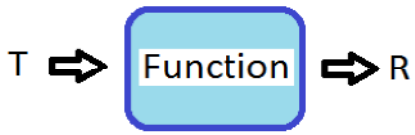

Abbildung 2-1 *Grafische Notation für das Function Interface* `Function<T, R>`