2.1 Einstieg in Lambdas

Das Sprachkonstrukt Lambda kommt aus der funktionalen Programmierung. Ein ***Lambda*** ist ein Behälter für Sourcecode ähnlich einer Methode, allerdings ohne Namen und ohne die explizite Angabe eines Rückgabetyps oder ausgelöster Exceptions. Vereinfacht ausgedrückt kann man einen Lambda am ehesten als anonyme Methode mit folgender Syntax und spezieller Kurzschreibweise auffassen:

```
(Parameter-Liste) -> { Ausdruck oder Anweisungen }
```

2.1.1 Lambdas am Beispiel

Ein paar recht einfache Beispiele für Lambdas sind die Addition von zwei Zahlen vom Typ `int`, die Multiplikation eines `long`-Werts mit dem Faktor 2 oder eine parameterlose Funktion zur Ausgabe eines Textes auf der Konsole. Diese Aktionen kann man als Lambdas wie folgt schreiben:

```
(int x, int y) -> { return x + y; }
(long x) -> { return x * 2; }
() -> { String msg = "Lambda"; System.out.println("Hello " + msg); }
```

Das sieht recht unspektakulär aus, und insbesondere wird klar, dass ein Lambda lediglich ein Stück ausführbarer Sourcecode ist, der

- keinen Namen besitzt, sondern lediglich Funktionalität, und dabei
- keine explizite Angabe eines Rückgabetyps und
- keine Deklaration von Exceptions erfordert und erlaubt.

Lambdas im Java-Typsystem

Wir haben bisher gesehen, dass sich einfache Berechnungen mithilfe von Lambdas ausdrücken lassen. Wie können wir diese aber nutzen und aufrufen? Versuchen wir zunächst, einen Lambda einer `java.lang.Object`-Referenz zuzuweisen, so wie wir es mit jedem anderen Objekt in Java auch tun können:

```
// Compile-Error: incompatible types: Object is not a functional interface
Object greeter = () -> { System.out.println("Hello Lambda"); };
```

Die gezeigte Zuweisung ist nicht erlaubt und führt zu einem Kompilierfehler. Die Fehlermeldung gibt einen Hinweis auf inkompatible Typen und verweist darauf, dass `Object` kein Functional Interface ist. Aber was ist denn ein Functional Interface?

Besonderheit: Lambdas im Java-Typsystem

Bis JDK 8 konnte in Java jede Referenz auf den Basistyp `Object` abgebildet werden. Mit Lambdas existiert nun ein Sprachelement, das nicht direkt dem Basistyp `Object` zugewiesen werden kann, sondern nur an Functional Interfaces.

2.1.2 Functional Interfaces und SAM-Typen

Ein ***Functional Interface*** ist eine neue Art von Typ, die mit JDK 8 eingeführt wurde, und repräsentiert ein Interface mit genau einer abstrakten Methode. Ein solches wird auch ***SAM-Typ*** genannt, wobei SAM für Single Abstract Method steht. Diese Art von Interfaces gibt es nicht erst seit Java 8 im JDK, sondern schon seit Langem und vielfach – wobei es früher für sie aber keine Bezeichnung gab. Vertreter der SAM-Typen und Functional Interfaces sind etwa `Runnable`, `Callable<V>`, `Comparator<T>`, `FileFilter`, `FilenameFilter`, `ActionListener`, `EventHandler` usw.

```
@FunctionalInterface
public interface Runnable
{
    public abstract void run();
}
```

```
@FunctionalInterface
public interface Comparator<T>
{
    int compare(T o1, T o2);
    boolean equals(Object obj);
}
```

Im Listing sehen wir die mit JDK 8 eingeführte Annotation `@FunctionalInterface` aus dem Package `java.lang`. Damit wird ein Interface explizit als Functional Interface gekennzeichnet. Die Angabe der Annotation ist optional: Jedes Interface mit genau nur einer abstrakten Methode (SAM-Typ) stellt auch ohne explizite Kennzeichnung ein Functional Interface dar. Wenn die Annotation angegeben wird, kann der Compiler eine Fehlermeldung produzieren, falls es (versehentlich) mehrere abstrakte Methoden gibt.

Tipp: Besondere Methoden in Functional Interfaces

Wenn wir im obigen Listing genauer hinsehen, könnten wir uns fragen, wieso denn `java.util.Comparator<T>` ein Functional Interface ist, wo es doch zwei Methoden enthält und keine davon abstrakt ist, oder? Als Besonderheit gilt in Functional Interfaces folgende Ausnahme für die Definition von abstrakten Methoden: Alle im Typ `Object` definierten Methoden können zusätzlich zu der abstrakten Methode in einem Functional Interface angegeben werden.

Verbleibt noch die Frage, warum wir in der Definition des Interface `Comparator<T>` keine abstrakte Methode sehen. Mit ein wenig Java-Basiswissen oder nach einem Blick in die Java Language Specification (JLS) erinnern wir uns daran, dass alle Methoden in Interfaces automatisch `public` und `abstract` sind, auch wenn dies nicht explizit über Schlüsselwörter angegeben ist.

Basierend auf den Argumentationen ist die Methode `compare(T, T)` abstrakt und die Methode `equals(Object)` entstammt dem Basistyp `Object`. Sie darf damit zusätzlich im Interface zur abstrakten Methode aufgeführt werden.

Implementierung von Functional Interfaces

Herkömmlicherweise wird ein SAM-Typ bzw. Functional Interface durch eine anonyme innere Klasse implementiert. Seit JDK 8 kann man alternativ zu dessen Implementierung auch Lambdas nutzen. Voraussetzung dafür ist, dass das Lambda die abstrakte Methode des Functional Interface erfüllen kann, d. h., dass die Anzahl der Parameter übereinstimmt sowie deren Typen und der Rückgabetyp kompatibel sind. Schauen wir zur Verdeutlichung zunächst auf ein allgemeines, etwas abstraktes Modell zur Transformation von bisherigen Realisierungen eines SAM-Typs mithilfe einer anonymen inneren Klasse in einen Lambda-Ausdruck:

```
// SAM-Typ als anonyme innere Klasse
new SAMTypeAnonymousClass()
{
    public void samTypeMethod(METHOD-PARAMETERS)
    {
        METHOD-BODY
    }
}

// SAM-Typ als Lambda
(METHOD-PARAMETERS) -> { METHOD-BODY }
```

Bei kurzen Methodenimplementierungen, wie sie für SAM-Typen häufig vorkommen, ist das Verhältnis von Nutzcode zu Boilerplate-Code (oder auch Noise genannt) bislang recht schlecht. Wenn man für derartige Realisierungen Lambdas einsetzt, so kann man mit einer Zeile das ausdrücken, was sonst fünf oder mehr Zeilen benötigt. Nachfolgend wird dies für die Interfaces `Runnable` und `Comparator<T>` verdeutlicht.

Beispiel 1: `Runnable` Konkretisieren wir die allgemeine Transformation anhand eines `java.lang.Runnable`, das eine triviale Konsolenausgabe implementiert:

```
Runnable runnableAsNormalMethod = new Runnable()
{
    @Override
    public void run()
    {
        System.out.println("runnable as normal method");
    }
}
```

In diesem `Runnable` wird keine wirklich sinnvolle Funktionalität realisiert. Vielmehr dient dies nur der Verdeutlichung der Kurzschreibweise mit einem Lambda wie folgt:

```
Runnable runnableAsLambda = () -> System.out.println("runnable as lambda");
```

Beispiel 2: `Comparator<T>` Die Vorteile von Lambdas lassen sich für das Functional Interface `Comparator<T>` prägnanter zeigen. Ich möchte kurz in Erinnerung rufen, dass mit einem Komparator ein Vergleich von zwei Instanzen vom Typ `T` realisiert wird. Dazu muss die abstrakte Methode `int compare(T, T)` passend implementiert