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. Bekannte Vertreter der SAM-Typen und Functional Interfaces sind etwa `Runnable`, `Callable`, `Comparator`, `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. Sofern die Annotation angegeben wird, kann der Compiler eine Fehlermeldung produzieren, falls es 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 Noise) bislang recht schlecht. Wenn man für derartige Realisierungen Lambdas einsetzt, so kann man mit einer Zeile das ausdrücken, was sonst fünf Zeilen benötigt. Nachfolgend wird dies für das Interface `Runnable` 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 realisiert

werden und über ihren Rückgabewert die Reihenfolge der Werte ausdrücken (vgl. folgenden Praxishinweis). Wollte man zwei Strings nach deren Länge sortieren, so entsteht herkömmlicherweise recht viel Sourcecode:

```
// Hinweis: Diamond Operator ist nicht für anonyme innere Klassen möglich
Comparator<String> compareByLength = new Comparator<String>()
{
    @Override
    public int compare(final String str1, final String str2)
    {
        final int length1 = str1.length();
        final int length2 = str2.length();

        if (length1 < length2)
            return 1;
        if (length1 > length2)
            return -1;

        return 0;
    }
};
```

Mit JDK 7 wurde die Klasse `Integer` um eine Methode `compare(int, int)` erweitert, die einen komparatorkonformen Rückgabewert liefert und so die Implementierung deutlich vereinfacht und verkürzt:

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

Wenn man Lambdas nutzt, lässt sich der Komparator knackig wie folgt schreiben:

```
Comparator<String> compareByLength = (final String str1, final String str2) ->
{
    return Integer.compare(str1.length(), str2.length());
};
```

Hinweis: Realisierung von Komparatoren und Rückgabewerte

Der Rückgabewert der `compare(T, T)`-Methode bestimmt den Ausgang des Vergleichs der beiden Objekte: Ein Wert > 0 besagt, dass das erste Objekt größer ist, 0 beschreibt Gleichheit und < 0 signalisiert, dass das erste Objekt kleiner als das zweite ist. Einen `Comparator<T>` zu implementieren, ist nicht besonders schwierig, erfordert aber häufig einige Fallunterscheidungen und wird dadurch recht schnell unübersichtlich. Im obigen Beispiel sehen wir einige `if`-Anweisungen. Stattdessen scheint es einfacher, die beiden Werte voneinander zu subtrahieren. Zum Teil sieht man solche Lösungen. Dabei besteht jedoch die Gefahr von einem Überlauf des Wertebereichs des `int` und von Berechnungsfehlern. Außerdem spiegelt diese Art der Implementierung die Intention nicht gut wider und ist eher schlecht verständlich.

2.1.3 Type Inference und Kurzformen der Syntax

Es existieren einige Besonderheiten bezüglich der Syntax von Lambdas: Um den Sourcecode recht knapp formulieren zu können. Dabei nutzt man für Lambdas vor allem auch die sogenannte ***Type Inference***: Ähnlich wie beim Diamond Operator bei der Definition generischer Klassen ist es für Lambdas möglich, auf die Typangaben für die Parameter im Sourcecode zu verzichten. Dazu ermittelt der Compiler die passenden Typen aus dem Einsatzkontext. Den vorherigen Komparator schreibt man ohne Typangabe wie folgt:

```
Comparator<String> compareByLength = (str1, str2) ->
{
    return Integer.compare(str1.length(), str2.length());
};
```

Eine weitere Verkürzung in der Schreibweise eines Lambdas kann man durch folgende Regeln erzielen: Falls das auszuführende Stück Sourcecode ein Ausdruck ist, können die geschweiften Klammern um die Anweisungen entfallen. Ebenfalls kann dann das Schlüsselwort `return` weggelassen werden und der Rückgabewert entspricht dem Ergebnis des Ausdrucks. Außerdem gilt: Existiert lediglich ein Eingabeparameter, so sind die runden Klammern um den Parameter optional. Damit ergibt sich für die Ausdrücke

```
(int x, int y) -> { return x + y; }
(long x) -> { return x * 2; }
```

folgende Kurzschreibweise:

```
(x, y) -> x + y
x -> x * 2
```

Neben dem offensichtlichen Vorteil einer recht kompakten Schreibweise, ist etwas anderes viel entscheidender: Lambdas können flexibler als streng typisierte Methoden genutzt werden. Für die gezeigten Berechnungen ist ein Einsatz überall dort möglich, wo für die Parameter die Operatoren + bzw. * definiert sind, also für die Typen `int`, `float`, `double` usw. Anders formuliert: ***Alles, was hergeleitet werden kann (und soll), darf in der Syntax weggelassen werden***. Als Beispiel betrachten wir folgende `ActionListener`-Implementierung, die schrittweise vereinfacht wird:

```
// Alter Stil
button.addActionListener(new ActionListener()
{
    @Override
    public void actionPerformed(final ActionEvent e)
    {
        System.out.println("button clicked (old way)");
    }
});
```

Diese herkömmliche Realisierung mithilfe einer anonymen inneren Klasse lässt sich als Lambda und mit Type Inference deutlich kürzer schreiben:

```
// Lambda-Variante mit Type Inference
button.addActionListener( (e) -> { System.out.println("button clicked!"); } );
```

Nutzt man zusätzlich die Regeln zur Schreibweisenabkürzung, so entsteht Folgendes:

```
// Lambda-Kurzschreibweise
button.addActionListener( e -> System.out.println("button clicked!") );
```

2.1.4 Lambdas als Parameter und als Rückgabewerte

Wir haben mittlerweile ein wenig Gespür für Lambdas gewonnen und wissen, dass man Lambdas anstelle einer anonymen inneren Klasse zur Realisierung eines SAM-Typs nutzen kann. Ebenso kann man Lambdas auch als Methodenparameter und als Rückgabe einer Methode verwenden, um Aufrufe lesbar zu gestalten.

Wir greifen das Beispiel aus der Einleitung wieder auf und betrachten das Sortieren einer Liste von Namen. Das können wir mit folgenden zwei Varianten eines Lambdas für das Interface `Comparator<T>` schreiben:

```
public static void main(final String[] args)
{
   final List<String> names = Arrays.asList("Andy", "Michael", "Max", "Stefan");

   // Lambda als Methodenparameter
   Collections.sort(names, (str1, str2) -> Integer.compare(str1.length(),
                                                           str2.length()));

   // Alternative mit Lambda als Rückgabe einer Methode
   names.sort(compareByLength());
}

public static Comparator<String> compareByLength()
{
    return (str1, str2) -> Integer.compare(str1.length(), str2.length());
}
```

Wenn Sie im Listing genau hingeschaut haben, dann könnte Ihnen aufgefallen sein, dass bei der zweiten Variante gar nicht `Collections.sort()`, sondern `names.sort()` aufgerufen wird – also direkt auf einer Instanz von `List<String>`. Wie geht das denn? Diese Methode existiert doch gar nicht im Interface `java.util.List<T>`, oder etwa doch? Bis JDK 7 ist sie dort nicht vorhanden. Jedoch wurden mit JDK 8 das Interface `List<T>` und viele andere Interfaces erweitert. Darauf gehe ich gleich im Anschluss nach der Betrachtung einiger Details zu Lambdas und deren Verwendung ein.

2.1.5 Unterschiede: Lambdas vs. anonyme innere Klassen

Wir wissen zwar schon das eine oder andere über Lambdas, aber es gibt noch kleine Unterschiede zu anonymen inneren Klassen kennenzulernen. Nämlich bei der Bedeutung von `this` sowie dem Zugriff auf Variablen und der Erweiterbarkeit um Methoden.

Bedeutung von `this`

Lambdas repräsentieren lediglich ein Stück Funktionalität und haben keine Bindung zu einem Objekt. Wenn man dort `this` referenziert, dann liegt der Bezugspunkt also außerhalb des Lambdas und demnach besitzt `this` innerhalb eines Lambdas die gleiche Bedeutung wie in den Zeilen direkt außerhalb davon. Für innere Klassen referenziert `this` dagegen die innere Klasse selbst. Das hat insbesondere Einfluss darauf, wie man auf Attribute der äußeren Klasse zugreifen kann.

Zugriff auf Variablen

Kommen wir zum Zugriff auf Variablen, die außerhalb des Lambdas bzw. der anonymen inneren Klasse definiert sind. Bis JDK 7 konnte man auf derartige Variablen nur dann zugreifen, wenn diese explizit `final` definiert waren. Mit JDK 8 wird das Ganze etwas gelockert: Es reicht nun sowohl für Lambdas als auch für anonyme innere Klassen, wenn die Variablen »effectively« `final` sind. Darunter versteht man, dass die Variablen nicht mehr explizit `final` deklariert werden müssen, sondern es genügt, wenn diese ihren Wert zur Programmlaufzeit nicht ändern. Dieser Sachverhalt wird vom Compiler geprüft und Verstöße werden als Fehler angemahnt.

Erweiterungen eines SAM-Typs

Innerhalb von anonymen inneren Klassen kann man beliebige weitere Methoden definieren. Für Lambdas ist das nicht möglich. Sie können zwar stellvertretend für SAM-Typen genutzt werden, erlauben aber keine Erweiterung um Methoden, da sie eher anonymen Methoden als anonymen Klassen entsprechen.

Beispiel

Um die obigen Ausführungen zu verdeutlichen, schauen wir auf folgendes Beispiel:

```
public class LambdaVsInnerClassExample
{
   private String outerAttribute = "fromOutside";

   public static void main(final String[] args)
   {
      new LambdaVsInnerClassExample().executeMethodAndLambda();
   }

   private void executeMethodAndLambda()
   {
      // Nicht finale Variable war bis JDK 7 nicht referenzierbar
      /* final */ int effectivelyFinal = 4711;

      final Runnable asNormalMethod = new Runnable()
      {
         @Override
         public void run()
         {
            System.out.println(this);
```

```
            // Nicht finale Variable war bis JDK 7 nicht referenzierbar
            System.out.println("effectivelyFinal = " + effectivelyFinal);
            // Spezielle Syntax zum Zugriff auf Attribute der äußeren Klasse
            System.out.println("outerAttribute = " +
                               LambdaVsInnerClassExample.this.outerAttribute);
         }

         // In inneren Klassen kann man weitere Methoden definieren
         public String anotherMethod()
         {
            return "Anonymous Runnable";
         }
      };

      // Man kann keine weiteren Methoden in Lambdas definieren
      final Runnable asLambda = () ->
      {
         System.out.println(this);
         // Nicht finale Variable war bis JDK 7 nicht referenzierbar
         System.out.println("effectivelyFinal = " + effectivelyFinal);
         System.out.println("outerAttribute = " + outerAttribute);
      };

      asNormalMethod.run();
      asLambda.run();
   }
}
```

Das Programm LAMBDAVSINNERCLASSEXAMPLE produziert folgende Ausgaben:

```
jdk8.LambdaVsInnerClassExample$1@4617c264      // $1 => Innere Klasse
effectivelyFinal = 4711
outerAttribute = fromOutside

jdk8.LambdaVsInnerClassExample@36baf30c
effectivelyFinal = 4711
outerAttribute = fromOutside
```

Zwar sieht man bei der Ausgabe nur einen kleinen Unterschied, jedoch zeigt der Sourcecode durchaus Unterschiede und Besonderheiten, die oben fett hervorgehoben sind.

Kommen wir kurz auf die Bedeutung von `this` zurück. Diese hat insbesondere Einfluss darauf, wie man auf Attribute der äußeren Klasse zugreifen kann. Im Beispiel muss man für die innere Klasse etwas umständlich `LambdaVsInnerClassExample.this.outerAttribute` schreiben. Für den Lambda nutzt man nur den Variablennamen, also hier `this.outerAttribute` oder kürzer einfach `outerAttribute`.

2.2 Default-Methoden

Beim Entwurf von Lambdas und deren Integration in das JDK stellte sich heraus, dass für eine sinnvolle Nutzbarkeit auch die bestehenden Klassen und Interfaces erweitert werden mussten. Bis zur Einführung von JDK 8 war es allerdings nicht möglich, ein Interface nach seiner Veröffentlichung zu verändern, ohne dass dies Auswirkungen bei allen einsetzenden Klassen gehabt hätte. Vielmehr führte die Erweiterung eines Interface bis inklusive JDK 7 immer zu einem Kompatibilitätsproblem: Wenn eine Methode