Übung 04: Generics und Lambdas

Abgabetermin: 16.4.2015, 8:15

Name: _____ **Matrikelnummer**: _____

Informatik: ☐ G1 (Prähofer) ☐ G2 (Prähofer) ☐ G3 (Grimmer) ☐ G4 (Grimmer)

WIN: ☐ G1 (Khalil) ☐ G2 (Kusel) ☐ G3 (Kusel)

Aufgabe	Punkte	abzugeben schriftlich	abzugeben elektronisch	korr.	Punkte
Übung 4	24	Java-Programm, Ausgabe von Testläufen	Java-Programm	☐	

Übung 4: Generische Funktionsbausteine

In dieser Übung soll aufbauend auf den funktionalen Interfaces des Packages `java.util.function` (siehe Anhang) ein generisches Baukastensystem mit Funktionsbausteinen realisiert werden. Mit dem Baukasten soll es möglich sein, Ketten von Funktionsbausteinen zu bilden und dadurch komplexe Verarbeitungen zu realisieren. Für die Implementierung der Bausteine sollen Sie dabei ein generisches Basissystem implementieren, bei dem die Typen der verarbeiteten Werte generisch sind und die Verarbeitung mit Funktionsobjekten passieren soll (*Lambda Expressions*).

Die folgende Abbildung zeigt ein Beispiel einer solchen funktionalen Kette für die Verarbeitung von Zahlenwerten. Die Werte werden beim ersten Baustein eingegeben und dann durch eine Reihe von Funktionen geschoben (Skalierung, Durchschnittsbildung, Rundung), um sie dann durch den letzten Baustein anzeigen zu lassen (vergleiche Aufgabe 4.b).

Anmerkung: Das Beispielszenario ist der Verarbeitung von Sensorwerten nachempfunden.

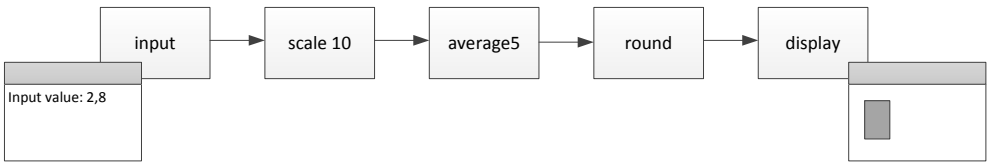

Abbildung 1

In der Kette kennt jeder Baustein seinen Nachfolger, an den er die Werte weiterleitet. Dazu dient das generische Interface `Receiver<A>` mit einer abstrakten Methode `receive(A a)`. Das heißt, ein Baustein hat einen nachfolgenden `Receiver` und ruft die Methode `receive` auf, um den Wert weiterzureichen.

```
public interface Receiver<A> {
   public void receive(A a);
}
```

Folgende Implementierung eines Bausteins `Source` (siehe auch Programmvorgabe im Download) soll das Vorgehen veranschaulichen.

- Der Baustein dient zur Generierung von Werten beliebigen Typs A:

```
public class Source<A> { … }
```

- Der Baustein definiert einen nächsten Receiver, der mit `setNext` gesetzt werden kann:

```
public class Source<A> {
   private Receiver<A> next;
   public void setNext(Receiver<A> next) {
      this.next = next;
   }
   …
}
```

- Die Funktion des Bausteins wird durch ein Funktionsobjekt realisiert. Daher hat der Baustein ein Feld `generator` vom Typ `Supplier<A>`, mit dem Objekte von Typ `A` generiert werden können:

```
public class Source<A> {
   …
   private final Supplier<A> generator;
   …
}
```

- Bei den Funktionsbausteinen soll man Zusätze definieren können, die vor der Anwendung (`onReceive`) oder nach der Anwendung (`onSend`) ausgeführt werden. Diese Zusätze werden wieder mit Funktionsobjekten definiert. Da Baustein `Source<A>` nur Objekte sendet und keine empfängt hat er nur ein Feld `onSend` vom Typ `Consumer<A>`:

```
public class Source<A> {
   …
   private final Consumer<A> onSend;
   …
}
```

- Beide Funktionsobjekte werden mit dem Konstruktor initialisiert

```
public class Source<A> {
   …
   public Source(Supplier<A> generator, Consumer<A> onSend) {
      this.generator = generator;
      this.onSend = onSend;
   }
   …
}
```

- Somit kann für `Source<A>` eine Methode `generate` implementiert werden, mit der Objekte vom Typ `A` generiert und an den Receiver `next` geschickt werden.

```
public class Source<A> {
   …
   public void generate() {
      A a = generator.get();
      if (next != null) {
         onSend.accept(a);
         next.receive(a);
      }
   }
}
```

Die folgende `main`-Methode zeigt nun die Erzeugung eines konkreten Bausteins für die Eingabe von Float-Werten. Dieser Baustein liest Float-Werte von der Konsole ein und erzeugt beim Senden eine Trace-Ausgabe auf die Konsole. Dieses Verhalten wird durch Setzen der Funktionsobjekte `generator` und `onSend` erzielt, wobei diese im Konstruktor mit Hilfe von Lambda-Expressions übergeben werden. Darüber hinaus wird mit `setNext` der Receiver des Bausteins gesetzt. Mit dem Aufruf von `generate` in einer Schleife wird der Prozess gestartet.

```
public static void main(String[] args) {
   Source<Float> input = new Source<Float>(
         () -> {
            Out.print("Input new float value: ");
            return In.readFloat();
         },
         f -> {
            Out.println(" -> input: value " + f + " sent!");
         }
   );
   input.setNext( … );
   …
   Window.clear();
   while (true) {
      input.generate();
   }
 }
```

Aufgabe 4.a) Funktionsbausteine `Step` und `Sink` (6 Punkte)

Implementieren Sie nun folgende generische Bausteine:

`Transform`

Transform soll eine Transformation eines empfangenen Wertes vom Typ `A` in einen Wert von Typ `B` realisieren. Die eigentliche Transformation wird durch ein Funktionsobjekt vom Typ `Function<A, B>` durchgeführt. Es hat Funktionsobjekte `onReceive` und `onSend`, die beim Empfangen und beim Weiterleiten an den nächsten Receiver aufgerufen werden.

Sink

`Sink` ist ein Receiver von Werten vom generischen Typ `A`. Diese Werte werden in der `Sink` verarbeitet und es wird nichts weitergeschickt (`Sink` ist also eine Datensenke). Die Verarbeitung erfolgt mit einem Funktionsobjekt vom Typ `Consumer<A>`. Es hat ein Funktionsobjekt `onReceive`, das beim Empfangen aufgerufen wird.

Aufgabe 4.b) Kette zur Verarbeitung von Zahlenwerten (6 Punkte)

Implementieren Sie nun auf Basis der generischen Bausteine `Source`, `Transform` und `Sink` die Kette wie in Abbildung 1 dargestellt:

- `input`: liest wie in obiger `main`-Methode dargestellt Float-Werte von der Konsole ein
- `scale10`: multipliziert den Wert mit 10
- `average5`: bildet den Durchschnitt der letzten 5 Werte
- `display`: gibt ein Rechteck mit Höhe gleich dem empfangenen Wert auf `Window` aus

Zusätzlich soll mit den `onSend`- und `onReceive`-Funktionen ein Trace der Berechnung ausgegeben werden, z.B.:

```
Input new float value: 3.8
 -> input: value 3.8 sent!
 -> scale10: value 3.8 received!
 -> scale10: value 38.0 sent!
 -> average: value 38.0 received!
 -> average: value 38.0 sent!
 -> round: value 38.0 received!
 -> round: value 38 sent!
 -> display: value 38.0 received!
Input new float value: 4.2
 -> input: value 4.2 sent!
 -> scale10: value 4.2 received!
 -> scale10: value 42.0 sent!
 -> average: value 42.0 received!
 -> average: value 40.0 sent!
 -> round: value 40.0 received!
 -> round: value 40 sent!
 -> display: value 40.0 received!
Input new float value:
```

Aufgabe 4.c) Funktionsbausteine `Filter` und `Branch` (6 Punkte)

Implementieren Sie nun folgende generische Bausteine

`Filter`

Mit einem Filter sollen nur Werte, die eine gegebene Eigenschaft erfüllen, weitergeleitet werden. Der Baustein verwendet daher ein Funktionsobjekt vom Typ `Predicate<A>`. Sehen Sie bitte wiederrum Funktionsobjekte `onReceive` und `onSend` vor.

`Branch`

`Branch`es sind Bausteine, die den empfangenen Wert auf Basis einer Boolschen Entscheidung an zwei mögliche Nachfolger schicken. Für die Entscheidung wird ein Funktionsobjekt vom Typ `Predicate<A> verwendet`. Sehen Sie bitte wiederrum Funktionsobjekte `onReceive` und `onSend` vor.

Aufgabe 4.d) Kette zur Verarbeitung von Kommandos (6 Punkte)

Implementieren Sie nun auf Basis der generischen Bausteine `Source`, `Transform`, `Sink`, `Filter` und `Branch` eine Kette zur Verarbeitung von Kommandos, wobei mit den Kommandos graphische Objekte erzeugt werden sollen.

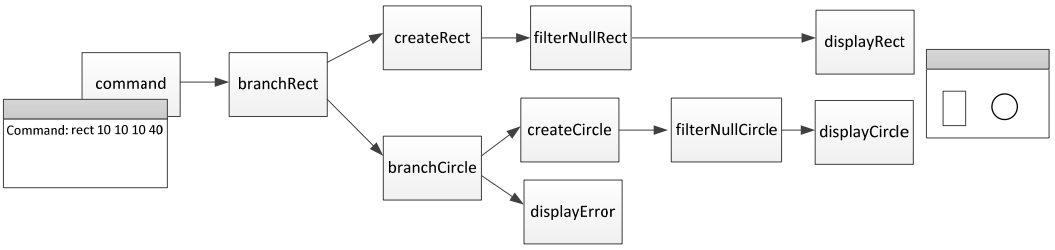

Abbildung 2

- `command`: liest einen String von der Konsole ein
- `branchRect`: entscheidet aufgrund des Command-Strings (`startsWith("rect")`), ob ein Rechteck angezeigt werden soll
- `createRect`: liest die Daten aus dem String und erzeugt ein `Rect`-Objekt; bei einem Fehler soll `null` weitergegeben werden
- `branchCircle`: entscheidet aufgrund des Command-Strings (`startsWith("circle")`), ob ein Kreis angezeigt werden soll
- `createCircle`: liest die Daten aus dem String und erzeugt ein `Circle`-Objekt; bei einem Fehler soll `null` weitergegeben werden
- `filterNullRect` und `filterNullCircle`: filtert `null`-Werte aus
- `displayRect und displayCircle`: gibt ein Rechteck oder einen Kreis auf Window aus
- `displayError`: gibt eine Fehlermeldung auf der Konsole aus

Wiederrum soll zusätzlich mit den `onSend`- und `onReceive`-Funktionen ein Trace der Berechnung ausgegeben werden, z.B.:

```
Input command: rect 10 20 30 40
 -> command: sent rect 10 20 30 40
 -> branchRect: received rect 10 20 30 40
 -> branchRect: sent rect 10 20 30 40
 -> createRect: received rect 10 20 30 40
 -> createRect: sent Rect [x=10, y=20, w=30, h=40]
 -> filterRectNull: received Rect [x=10, y=20, w=30, h=40]
 -> filterRectNull: sent Rect [x=10, y=20, w=30, h=40]
 -> displayRect: received Rect [x=10, y=20, w=30, h=40]
Input command: circle 40 40 20
 -> command: circle 40 40 20
 -> branchRect: received circle 40 40 20
 -> branchRect: sent circle 40 40 20
 -> branchCircle: received circle 40 40 20
 -> branchCircle: sent circle 40 40 20
 -> createCircle: received circle 40 40 20
 -> createCircle: sent Circle [x=40, y=40, r=20]
 -> filterRectNull: received Circle [x=40, y=40, r=20]
 -> filterRectNull: sent Circle [x=40, y=40, r=20]
 -> displayCircle: received Circle [x=40, y=40, r=20]
Input command:
```

Anhang: Funktionale Interfaces aus `java.util.function` (Auszug)

```
@FunctionalInterface
public interface Function<T,R> {
       R apply(T t);
       ...
}

@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
    ...
}

@FunctionalInterface
public interface Consumer<T> {
       void   accept(T t);
       ...
}

@FunctionalInterface
public interface Supplier<T> {
       T get();
}
```