JUGS — Java 8 Hands On Übungen

michael.inden@zuehlke.com

Lambdas

Aufgabe 1a:
Was sind gültige Lambdas? Schaue auf das folgende Interface `LongBinaryOperator`.

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

Nachfolgend sind einige Lambdas gezeigt. Welche davon sind gültig? Wofür erhält man Fehler beim Kompilieren?

```
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: Überlege ein wenig und prüfe deine Vermutungen dann mithilfe der IDE.

Aufgabe 1b:
Welche Varianten kompilieren, wenn man das Interface wie folgt abändert:

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

Aufgabe 2a:
Schreibe Functional Interfaces für die folgenden Abbildungen / Aufgaben

a) Bilde Objekte von Typ `String` auf `int` ab: `StringToIntConverter`
b) Bilde Objekte von Typ `String` auf `String`: `StringToStringMapper`

und implementiere diese mit mindestens einem Lambda.

Aufgabe 3a:
Gegeben sei folgendes rudimentäres Fragment eines Functional Interfaces als Ergänzung der obigen String-Mapper:

```
@FunctionalInterface
public interface Mapper<S, T>
{
       // TODO: map S -> T
       // TODO: mapAll List<S> -> List<T>
}
```

Vervollständige die Implementierung, sodass nachfolgendes Programm kompiliert:

```
public static void main(final String[] args)
{
    final List<String> names = Arrays.asList("Tim", "Andi", "Michael");
    final Mapper<String, Integer> intMapper = String::length;
    System.out.println(intMapper.mapAll(names));
    final Mapper<String, String> stringMapper = str -> ">> " +
                                                str.toUpperCase() + " << ";
    final List<String> uppercaseNames = stringMapper.mapAll(names);
    System.out.println(uppercaseNames);
}
```

Tipp: Stelle die `mapAll()`-Funktionalität mithilfe einer Defaultmethode bereit.

Aufgabe 3b:
Welche Alternativen bietet das JDK 8? Schaue ins Package `java.util.function`.

Aufgabe 3c:
Was ist von Defaultmethoden in Applikationsklassen zu halten? Welche Vorteile bringen diese, welche Fallstricke sollte man bedenken? Diskutiere mit deinem Nachbarn!

Aufgabe 4a:
Was ist bei Lambdas und dort ausgelösten Checked Exceptions zu bedenken? Wie kann man folgendes `Runnable` so korrigieren, dass es kompiliert?

```
final Runnable runner = () -> { System.out.println("Throwing");
                                throw new IOException(); };
```

Aufgabe 4b:
Realisiere ein Functional Interface als Erweiterung zu `Runnable`, dass eine Checked Exception in den Typ `RuntimeException` wandelt.

```
@FunctionalInterface
public interface RunnableThatThrows extends Runnable
{
    @Override
    default void run()
    {
         // TODO
    }

    public void runThrows() throws Exception;
}
```

Der Aufruf soll dann wie folgt möglich sein:

```
public class Exercise4_RunnableThatThrowsExample
{
    public static void main(String[] args)
    {
        final RunnableThatThrows runner2 = () ->
            { System.out.println("RunnableThatThrows"); throw new IOException(); };
        runner2.run();
    }
}
```

Aufgabe 4c:
Realisiere ein Functional Interface als Erweiterung zu `Comparator`, dass eine Checked Exception in den Typ `RuntimeException` wandelt basierend auf folgenden Zeilen:

```
@FunctionalInterface
public interface ComparatorThatThrows<T> extends Comparator<T>
{
    public int compareThrows(final T t1, final T t2) throws Exception;
}
```

KÜR:
Beschäftige dich mit der Frage: Wie kann man mit Lambdas rekursive Aufrufe formulieren? Das geht mit etwas tricksen. Widerlege also die Aussage von Angelika Langer, dass Rekursion in Lambdas nicht möglich ist.[1] Starte mit den folgenden, *nicht kompilierfähigen* Ausdrücken und bringe diese zum Laufen:

```
UnaryOperator<Integer> myFactorialInt =
                        i -> i == 0 ? 1 : i * myFactorialInt.apply(i - 1);
Function<Integer, Long> myFactorialLong =
                         i -> i == 0 ? 1 : i * myFactorialLong.apply(i - 1);
```

Tipp: Versuche es mit einem statischen Attribut oder einem Instanzattribut.
Was könnte an dem obigen `UnaryOperator` problematisch sein?

Prüfe deine Lösung mit folgenden Aufrufen:

```
myFactorialInt.apply(5)
myFactorialLong.apply(5)
myFactorialInt.apply(20)
myFactorialLong.apply(20)
myFactorialInt.apply(50)
myFactorialLong.apply(50)
```

Die Ausgaben sind vermutlich wie folgt:

```
120
120
-2102132736
2432902008176640000
0
-3258495067890909184
```

Man erkennt die Überläufe für den Typ `Integer` bzw. `int`. Was gibt es für Lösungsmöglichkeiten? Schaue mal in die Klasse `Math`. Seit Java 8 findet man dort Abhilfe.

---

[1] Maurice Naftalin schreibt dazu Folgendes:
Yes, provided that the recursive call uses a name defined in the enclosing environment of the lambda. This means that recursive definitions can only be made in the context of variable assignment and, in fact—given the assignment-before-use rule for local variables—only of instance or static variable assignment. So, in the following example, factorial must be declared as an instance or static variable.

Collections

Aufgabe 1a: Formuliere die Bedingungen „Zahl ist gerade", „Zahl ist positiv", „Zahl ist Null" und „Wert ist `null`" als `Predicate<Integer>` und/oder als `IntPredicate` und prüfe diese mit verschiedenen Eingaben.

```
final Predicate<Integer> isEven = // ... TODO ...
final IntPredicate isPositive = // ... TODO ...
```

Aufgabe 1b: Formuliere die Bedingung "Wort kürzer als 4 Buchstaben" und prüfe das damit realisierte `Predicate<String> isShortWord`.

Aufgabe 1c: Kombiniere die Prädikate wie folgt und prüfe wieder:

- Zahl ist positiv und Zahl ist gerade (Nutze die Methode `and()`)
- Zahl ist positiv und ungerade (Nutze einen Aufruf von `negate()`)

Aufgabe 2:
Lösche aus einer Liste von Namen all diejenigen Einträge mit kurzem Namen. Wandle dazu die externe in eine interne Iteration um. Die JDK-7-Implementierung aus dem Listing soll mithilfe von JDK-8-Mitteln einfacher gestaltet werden:

```
private static List<String> removeIf_External_Iteration()
{
    final List<String> names = createNamesList();
    final Iterator<String> it = names.iterator();
    while (it.hasNext())
    {
        final String currentName = it.next();
        if (currentName.length() < 4)
           it.remove();
    }
    return names;
}

private static List<String> createNamesList()
{
    final List<String> names = new ArrayList<>();
    names.add("Michael");
    names.add("Tim");
    names.add("Flo");
    names.add("Clemens");
    return names;
}
```

Tipp: Nutze das in Aufgabe 1 erstellte `Predicate<String> isShortWord` und die Methode `removeIf(Predicate<T>)` aus dem Interface `Collection<E>`.

Streams

Aufgabe 1:
Quadriere alle ungeraden Zahlen von 1 bis 10 und ermittle deren Summe:

```
final int result = IntStream.of(1,2,3,4,5,6,7,8,9,10).filter(…).
                                                      map(…).
                                                      reduce(0, …);
```

Tipp: Verwende lokale Hilfsvariablen, um die Verarbeitungsabfolgen besser lesbar zu gestalten. Möglicherweise ist `Integer::sum` beim Aufruf von `reduce()` nützlich.

Aufgabe 2a:
Finde mindesten drei unterschiedliche Arten, die Werte von 1 bis 10 als Stream zu erzeugen und (kommasepariert) auszugeben.

Aufgabe 2b:
Generiere die Ziffernfolge der natürlichen Zahlen als unendlichen Stream. Starte die Ausgabe bei 11 und begrenze diese auf 10 Elemente.

Tipp: Nutze die Methoden `skip()`, `limit()`, `generate(Supplier)` und `iterate()`. Zur Umwandlung eines `IntStream` in einen `Stream<Integer>` dient die Methode `boxed()`. `collect()` und `Collectors.joining()` hilft bei der Ausgabe.

Aufgabe 2c: Bedenke, dass unendliche Streams problematisch sein können — insbesondere beim Einsatz der Methode `sorted()`. Führe folgende Anweisungen zunächst ohne `sorted()` und danach mit aus. Schaue was passiert:

```
public class Exercise2_InfiniteStreamSurprises
{
      public static void main(String[] args)
      {
            IntStream.iterate(0, i -> i + 1).
                      boxed().
            //        sorted().
                      limit(10).
                      map(e -> "" + e).
                      forEach(System.out::println);
      }
}
```

Aufgabe 3:
Mitunter muss man eine Statistikauswertung erstellen, die auf einen `IntStream` operiert, etwa wenn man Minimum, Maximum, Durchschnitt usw. berechnen soll. Mit JDK 7 wird dies aufwendig, aber folgende Möglichkeit mit Stream ist auch nicht optimal:

```
final OptionalInt min = IntStream.of(1,2,3,4,5,6,7,8,9,10).min();
final OptionalInt max = IntStream.of(1,2,3,4,5,6,7,8,9,10).max();
final OptionalDouble average = IntStream.of(1,2,3,4,5,6,7,8,9,10).average();
final long count = IntStream.of(1,2,3,4,5,6,7,8,9,10).count();
final long sum = IntStream.of(1,2,3,4,5,6,7,8,9,10).sum();
```

Was ist daran unschön? Gibt es eine andere Variante? Schaue dich im `IntStream` um.

Aufgabe 4:
Gruppiere den Inhalt eines Streams von Strings nach dem Anfangsbuchstaben, sodass wir folgendes Ergebnis erhalten: `A=[Andy], T=[Tim, Tom], M=[Mike, Merten]`

```
final Stream<String> inputs = Stream.of("Tim", "Tom", "Andy", "Mike", "Merten");
final Map<Character, List<String>> grouped = inputs. // ... TODO ...
System.out.println(grouped);
```

Tipp: Nutze `collect()` und die Methode `Collectors.groupingBy()`.

Aufgabe 5a:
Erstelle ein Programm, das den Ablauf der Verarbeitungs-Pipeline mithilfe von Konsolenausgaben visualisiert. Ein Startpunkt liefern diese Zeilen:

```
Stream.of("Andi", "Barbara", "Carsten", "Marius", "Micha", "Tim").
       map(name -> { System.out.println("map: " + name);
                     return name.toUpperCase();  }).
       filter(name -> { System.out.println("filter: " + name);
                        return name.startsWith("M"); }).
       forEach(name -> System.out.println("forEach: " + name));
```

Welche Erkenntnisse gewinnst du anhand der Ausgaben? In welcher Reihenfolge sollte man die Operationen ausführen? Tausche einmal die Aufrufe von `filter()` und `map()`.

Aufgabe 5b:
Verbessere das Hineinschauen in den Stream durch Aufruf von `peek(Consumer)`.

Date and Time API

Aufgabe 1:
Berechne die Zeit zwischen heute und deinem Geburtstag bzw. deinem Geburtstag und heute.

```
final LocalDate now = LocalDate.now();
final LocalDate birthday = LocalDate.of(1971, 2, 7);
```

Tipp: Verwende die Klasse `LocalDate` und die Methode `until()`. Probiere als Variante auch den Aufruf von `Period.between()`.

Aufgabe 2:
Ermittle alle Zeitzonen, die mit "`America/L`" oder "`Europe/S`" starten und gib diese sortiert auf der Konsole aus oder befülle damit eine Liste.

Tipp: Nutze die Klasse `ZoneId` sowie deren Methode `getAvailableZoneIds()`. Setze Streams und das Filter-Map-Reduce-Framework ein, um die zu den zuvor angegebenen Präfixen passenden Zeitzonen-Ids zu ermitteln.

Aufgabe 3a:
Welcher Wochentag war der Heiligabend 2013 (24.12.2013)? Was für ein Wochentag war der erste und letzte Tag im Dezember? Starte mit folgenden Programmzeilen:

```
final LocalDate christmasEve = LocalDate.of(2014, 12, 24);
System.out.println(DayOfWeek.from(christmasEve));
```

Aufgabe 3b:
Berechne das Datum des ersten und letzten Freitags und Sonntags im März 2014. Starte mit folgendem Sourcecode-Schnipsel:

```
final LocalDate midOfMarch = LocalDate.of(2014, 3, 15);
final TemporalAdjuster toFirstSunday =
                       TemporalAdjusters.firstInMonth(DayOfWeek.SUNDAY);
```

Tipp: Nutze dabei die Klasse `TemporalAdjusters` und ihre Hilfsmethoden und die Methode `with()` aus der Klasse `LocalDate`.

Hinweis: Hier gibt es noch folgende alternative Lösung mit der Methode `adjustInto()` aus dem Interface `TemporalAdjuster`. `toFirstSunday.adjustInto(midOfMarch)`
Welche Variante ist zu bevorzugen bzw. besser zu verstehen?

Aufgabe 4:
Gegeben ist folgender selbstdefinierter TemporalAdjuster, der einen Datumswert auf den Anfang des jeweiligen Quartals setzt: Demnach springt man am 18. August auf den 1. Juli usw. Eine Implementierung für diese Datumsarithmetik findet man im Internet unter http://www.leveluplunch.com/java/examples/first-day-of-quarter-java8-adjuster/.
Betrachte den Sourcecode und überlege, was ungünstig an dieser Implementierung ist:[2]

```
public class Exercise04_FirstDayOfQuarterOrig implements TemporalAdjuster
{
    @Override
    public Temporal adjustInto(Temporal temporal)
    {
        int currentQuarter = YearMonth.from(temporal).
                                    get(IsoFields.QUARTER_OF_YEAR);

        if (currentQuarter == 1)
        {
            return LocalDate.from(temporal).
                            with(TemporalAdjusters.firstDayOfYear());
        }
        else if (currentQuarter == 2)
        {
            return LocalDate.from(temporal).withMonth(Month.APRIL.getValue()).
                                        with(firstDayOfMonth());
        }
        else if (currentQuarter == 3)
        {
            return LocalDate.from(temporal).withMonth(Month.JULY.getValue()).
                                        with(firstDayOfMonth());
        }
        else
        {
            return LocalDate.from(temporal).withMonth(Month.OCTOBER.getValue()).
                                        with(firstDayOfMonth());
        }
    }
}
```

Tipp: Was kann man vereinfachen? Entferne Spezialbehandlungen und Inkonsistenzen! Wenn du eine auf Monaten basierende Lösung hast, schaue nochmals in die Aufzählung `IsoFields`. Dort findet man auch `DAY_OF_QUARTER`.

KÜR:
Schreibe eine eigene Realisierung des Interface `TemporalAdjuster`, die

1. den 2. Samstag oder 3. Freitag usw. ermittelt (parametrierbar) oder
2. den Tag der nächsten Gehaltszahlung ermittelt (25. des Monats, oder Werktag davor, falls der 25. auf ein Wochenende fällt)

[2] Leider sieht man auch eine der wenigen kleinen Schwächen im neuen Date and Time API: Die Methode `withMonth()` arbeitet nicht mit Konstanten vom Typ `Month`, sondern leider mit `int`-Werten.