Lösungen zu den Übungsaufgaben Functional Programming

1. Functional Interfaces [TU]

a. Welche Interfaces aus dem Package `java.util.function` können Sie alles nutzen, um
   - die mathematische Funktion f(x) = x ^ 2 - 3 für Zahlen des Typs `long` abzubilden?
     1. `LongUnaryOperator`
     2. `LongFunction<R>` als `LongFunction<Long>`
     3. `ToLongFunction<T>` als `ToLongFunction<Long>`
     4. `UnaryOperator<T>` als `UnaryOperator<Long>`
     5. `Function<T,R>` als `Function<Long,Long>`
   - um den Zinsfaktor (double) für n (int) Jahre bei einem Zinssatz von p Prozent (float) zu berechnen mit der Formel zf = (1 + p / 100)^n ?
     1. `ToDoubleBiFunction<T,U>` als `ToDoubleBiFunction<Integer,Float>`
     2. `BiFunction<T,U,R>` als `BiFunction<Integer,Float,Double>`
   - ein Objekt vom Typ `Person` (ohne Parameter) zu generieren?
     1. `Supplier<T>` als `Supplier<Person>`

b. Welche Eigenschaft muss eine Funktion haben, damit Sie ein eigenes Interface schreiben müssen, also keines der in `java.util.function` vorhandenen Interfaces verwenden können?
   1. Sie muss mehr als zwei Parameter haben

c. Welche der Aussagen stimmen für ein funktionales Interface?
   - ☑ Es ist ein Java-Interface (Schlüsselwort `interface` im Code)
   - ☑ Es hat **genau eine** abstrakte Methode
   - ☐ Das Interface **muss** mit `@FunctionalInterface` markiert sein
   - ☐ Es hat **keine** default-Methoden (Schlüsselwort `default`)

d. Welche Aussagen stimmen?
   - ☑ Zu **jedem** funktionalen Interface können Lambda-Ausdrücke (*lambda expressions*) geschrieben werden
   - ☐ Ein Lambda-Ausdruck kann **ohne** passendes funktionales Interface erstellt werden
   - ☐ Eine Variable vom Typ `Optional` kann nie `null` sein.

2. Übungen auf der Stepik-Plattform [PU]

Übungen zu Functional Interface und Lambda Expression

a. Identify the correct lambdas and method references

   Korrekt sind:

   - `x -> { }`
   - `() -> 3`

b. Writing simple lambda expressions

```
(x, y) -> (x > y?x:y);
```

c. Too many arguments

`String.join()` dürfte effizienter sein als das Zusammenfügen von Strings mit `+`.

```
(a, b, c, d, e, f, g) -> String.join("", a, b, c, d, e, f, g).toUpperCase();
```

d. Writing closures

```
x -> a*x*x + b*x+c;
```

e. Replacing anonymous classes with lambda expressions Alles korrekt, ausser `Iterator<Integer> iterator = new Iterator<Integer>() …`

f. Matching the functional interfaces

function	lambda expression
IntSupplier	`() -> 3`
Consumer<String>	`System.out::println`
BiPredicate<Integer,Integer>	`(x,y) -> x % y == 0`
DoubleUnaryOperator	`Math::sin`
Function<Double,String>	`(x) -> String.valueOf(x*x)`

g. Your own functional interface

```
class Solution {
    @FunctionalInterface
    public interface TernaryIntPredicate {
        boolean test(int a, int b, int c);
    }
    public static final TernaryIntPredicate allValuesAreDifferentPredicate =
        (x, y, z) -> (x != y && y != z && z != x);
}
```

Übungen mit Streams

h. Calculating production of all numbers in the range

```
(l,r) -> LongStream.rangeClosed(l,r).reduce(1, (x,y) -> x*y);
```

i. Getting distinct strings

```
list -> list.stream().distinct().collect(Collectors.toList());
```

j. Composing predicates

```
class Solution {
    public static IntPredicate disjunctAll(List<IntPredicate> predicates) {
        return predicates.stream().reduce(x -> false, (a, b) -> a.or(b));
    }
}
```

Sie können auch den zweiten Parameter in Reduce durch `IntPredicate::or` ersetzen.

Oder mit meistens weniger Rechenaufwand:

```
class Solution {
    public static IntPredicate disjunctAllAnyMatch(List<IntPredicate> predicates) {
        return i -> predicates.stream().anyMatch(p -> p.test(i));
    }
}
```

k. Lösen Sie die folgenden Aufgaben mit Streams:

- Numbers filtering

```
class Solution {
    public static IntStream createFilteringStream(IntStream evenStream,
IntStream oddStream) {
        IntStream res = IntStream.concat(evenStream, oddStream);
        return res.filter(n -> n % 15 == 0).sorted().skip(2);
    }
}
```

- Calculating a factorial

```
class Solution {
  public static long factorial(long n) {
    return LongStream.rangeClosed(1L,n).reduce(1L, (a,b) -> a*b);
  }
}
```

- The sum of odd numbers

```
return LongStream.rangeClosed(start, end).filter(n -> n%2 == 1).sum();
```

- Collectors in practice: the product of squares

```
Collectors.reducing(1, (a, b) -> a * b*b);
```

- Almost like a SQL: the total sum of transactions by each account

```
Collectors.groupingBy(
    transaction -> transaction.getAccount().getNumber(),
    Collectors.summingLong(Transaction::getSum));
```

3. Design Pattern *Chain of responsibility* [PU]

```
class Solution {
    @FunctionalInterface
    interface RequestHandler {
        Request handle(Request request);
        default RequestHandler wrapFirst(RequestHandler otherHandler) {
            return request -> handle(otherHandler.handle(request));
        }
    }

    final static RequestHandler commonRequestHandler =
        wrapInRequestTag.wrapFirst(createDigest.wrapFirst(wrapInTransactionTag));
}
```

4. Company Payroll [PA]

Die Lösungen zu den bewerteten Pflichtaufgaben erhalten Sie nach der Abgabe und Bewertung aller Klassen.