5 Übungen zu den Neuerungen in JDK 9

Es sollen die Neuerungen aus JDK 9 anhand von Übungen kennengelernt werden.

✍ Aufgabe 1 – Prozessmanagement ✍

Ermitteln Sie die Prozess-ID und weitere Eigenschaften des aktuellen Prozesses, indem Sie dazu das Interface `ProcessHandle` und seine Methoden nutzen. Wie viel CPU-Zeit hat der aktuelle Prozess bislang verbraucht? Wie viele Prozesse werden momentan insgesamt ausgeführt?

✍ Aufgabe 2 – Collection-Factory-Methoden ✍

Definieren Sie eine Liste, eine Menge und eine Map mithilfe der in JDK 9 neu eingeführten Collection-Factory-Methoden namens `of()`. Als Ausgangsbasis dient folgendes Programmfragment mit JDK 8:

```
private static void collectionsExampleJdk8()
{
    final List<String> names = Arrays.asList("Tim", "Tom", "Mike");
    System.out.println(names);

    final Set<Integer> numbers = new TreeSet<>();
    numbers.add(1);
    numbers.add(3);
    numbers.add(4);
    numbers.add(2);
    System.out.println(numbers);

    final Map<Integer, String> mapping = new HashMap<>();
    mapping.put(5, "five");
    mapping.put(6, "six");
    mapping.put(7, "seven");
    System.out.println(mapping);
}
```

Was beobachtet man nach der Transformation und dem Einsatz der Collection-Factory-Methoden beim Ausführen mit JDK 9?

Nutzen Sie zur Definition der Map alternativ die Methode `ofEntries()` aus dem Interface `Map<K,V>`. Wie verändert sich die Lesbarkeit? Was sind die Vorteile dieser Variante?

✍ Aufgabe 3 – Streams ✍

Das Stream-API wurde um Methoden erweitert, die es erlauben, nur so lange Elemente zu lesen, wie eine Bedingung erfüllt ist, bzw. Elemente zu überspringen, solange eine Bedingung erfüllt ist. Als Datenbasis dienen folgende zwei Streams:

```
final Stream<String> values1 = Stream.of("a", "b", "c", "", "e", "f");
final Stream<Integer> values2 = Stream.of(1, 2, 3, 11, 22, 33, 7, 10);
```

Aufgabe 3a: Ermitteln Sie aus dem Stream `values1` so lange Werte, bis ein Leerstring gefunden wird. Geben Sie die Werte auf der Konsole aus.

Aufgabe 3b: Überspringen Sie in Stream `values2` die Werte, so lange der Wert kleiner als 10 ist. Geben Sie die Werte auf der Konsole aus.

Aufgabe 3c: Worin besteht der Unterschied zwischen den beiden Methoden `dropWhile()` und `filter()`.

Tipp Das erwartete Ergebnis ist folgendes:

```
takeWhile
a
b
c

dropWhile
11
22
33
7
10

with filter
11
22
33
10
```

✍ Aufgabe 4 – Die Klasse `Optional` ✍

Gegeben sei folgendes Programmfragment, das eine Personensuche ausführt und abhängig vom Ergebnis bei einem Treffer die Methode `doHappyCase(Person)` oder ansonsten `doErrorCase()` aufruft. Gestalten Sie das Programmfragment mithilfe der neuen Methoden aus der Klasse `Optional<T>` eleganter.

```
private static void findJdk8()
{
    final Optional<Person> opt = findPersonByName("Tim");
    if (opt.isPresent())
    {
        doHappyCase(opt.get());
    }
    else
    {
        doErrorCase();
    }

    final Optional<Person> opt2 = findPersonByName("UNKNOWN");
    if (opt2.isPresent())
    {
        doHappyCase(opt2.get());
    }
    else
    {
        doErrorCase();
    }
}

private static Optional<Person> findPersonByName(final String searchFor)
{
    final Stream<Person> persons = Stream.of(new Person("Mike"),
                                             new Person("Tim"),
                                             new Person("Tom"));

    return persons.filter(person -> person.getName().equals(searchFor))
                  .findFirst();
}

private static void doHappyCase(final Person person)
{
    System.out.println("Result: " + person);
}

private static void doErrorCase()
{
    System.out.println("not found");
}

static class Person
{
    private final String name;

    public Person(final String name)
    {
        this.name = name;
    }

    ...
}
```

✍ Aufgabe 5 – Die Klasse `Optional` ✍

Gegeben sei folgendes Programmfragment, das eine mehrstufige Suche zunächst im Cache, dann im Speicher und schließlich in der Datenbank ausführt. Diese Suchkette ist durch drei `find()`-Methoden angedeutet und wie folgt implementiert:

```
public static void main(final String[] args)
{
    final Optional<String> optCustomer = multiFindCustomerJdk8("Tim");
    optCustomer.ifPresentOrElse(str -> System.out.println("found: " + str),
                                () -> System.out.println("not found"));
}

private static Optional<String> multiFindCustomerJdk8(final String customerId)
{
    final Optional<String> opt1 = findInCache(customerId);

    if (opt1.isPresent())
    {
        return opt1;
    }
    else
    {
        final Optional<String> opt2 = findInMemory(customerId);

        if (opt2.isPresent())
        {
            return opt2;
        }
        else
        {
            return findInDb(customerId);
        }
    }
}

private static Optional<String> findInMemory(final String customerId)
{
    System.out.println("findInMemory");
    final Stream<String> customers = Stream.of("Tim", "Tom", "Mike", "Andy");

    return customers.filter(name -> name.contains(customerId))
                    .findFirst();
}

private static Optional<String> findInCache(final String customerId)
{
    System.out.println("findInCache");
    return Optional.empty();
}

private static Optional<String> findInDb(final String customerId)
{
    System.out.println("findInDb");
    return Optional.empty();
}
```

Vereinfachen Sie die Aufrufkette mithilfe der neuen Methoden aus der Klasse `Optional<T>`. Sehen Sie, wie das Ganze an Klarheit gewinnt.

✍ Aufgabe 6 – Berechnungen mit `LocalDate` ✍

Berechnen Sie mithilfe der neuen Methoden aus der Klasse `LocalDate` alle Vorkommen von Freitag, dem 13., in den Jahren 2013 bis einschließlich 2017. Nutzen Sie folgende Zeilen als Ausgangspunkt:

```
final LocalDate startDate = LocalDate.of(2013, Month.JANUARY, 1);
final LocalDate endDate = LocalDate.of(2018, Month.JANUARY, 1);

final Predicate<LocalDate> isFriday = null; // TODO
final Predicate<LocalDate> is13th = null; // TODO
```

Als Ergebnis sollten folgende Werte ausgegeben werden:

```
2013-09-13
2013-12-13
2014-06-13
2015-02-13
2015-03-13
2015-11-13
2016-05-13
2017-01-13
2017-10-13
```

✍ Aufgabe 7 – Reactive Streams ✍

Gegeben sei ein Programm mit einem `Publisher<String>`, der Worte aus einer Datei liest und diese an registrierte `Subscriber<String>` veröffentlicht:

```
public static void main(final String[] args) throws IOException,
                                                     InterruptedException
{
    final WordPublisher publisher = new WordPublisher();
    publisher.subscribe(new WordSubscriber());
    publisher.doWork();
    Thread.sleep(10_000);  // auf das Ende der Verarbeitung warten
}
```

Dabei ist der `Publisher<String>` einfach wie folgt realisiert:

```
public class WordPublisher implements Flow.Publisher<String>
{
    private final SubmissionPublisher<String> publisher;

    public WordPublisher()
    {
        this.publisher = new SubmissionPublisher<>();
    }

    public void subscribe(final Subscriber<? super String> subscriber)
    {
        publisher.subscribe(subscriber);
    }

    public void doWork() throws IOException
    {
        final Charset utf8 = StandardCharsets.UTF_8;
```

```
        final Path path = Paths.get("./resources/input.txt");
        final List<String> lines = Files.readAllLines(path, utf8);

        for (int i = 0; i <  lines.size(); i++)
        {
            publisher.submit("line: " + (i + 1) +  " : " + lines.get(i));
        }
    }
}
```

Eine Protokollierung geschieht mit folgender einfachen Klasse `WordSubscriber`, die alle Vorkommen auf der Konsole auflistet:

```
class WordSubscriber implements Subscriber<String>
{
    public void onSubscribe(final Subscription subscription)
    {
        subscription.request(Long.MAX_VALUE);
    }

    public void onNext(final String item)
    {
        System.out.println(LocalDateTime.now() + " onNext(): " + item);
    }

    public void onComplete()
    {
        System.out.println(LocalDateTime.now() + " onComplete()");
    }

    public void onError(final Throwable throwable)
    {
        throwable.printStackTrace();
    }
}
```

Implementieren Sie basierend auf der obigen Klasse `WordSubscriber` einen eigenen `Subscriber<String>` namens `SkipAndTakeSubscriber`, der die ersten n Vorkommen überspringt und danach m Vorkommen ausgibt. Danach soll die Kommunikation gestoppt werden, also der `WordPublisher` diesem `Subscriber<String>` keine Daten mehr senden.
Für die Eingabe

```
This
is
a
simple
text
file
----------------------
Hello
Java 9
----------------------
...
```

und die Parametrierung – 7 Einträge überspringen und 2 konsumieren – wird folgende Ausgabe erwartet:

```
SkipAndTakeSubscriber - Subscription: java.util.concurrent.
      SubmissionPublisher$BufferedSubscription@5dfaf274
SkipAndTakeSubscriber 1 x onNext()
SkipAndTakeSubscriber 2 x onNext()
SkipAndTakeSubscriber 3 x onNext()
SkipAndTakeSubscriber 4 x onNext()
SkipAndTakeSubscriber 5 x onNext()
SkipAndTakeSubscriber 6 x onNext()
SkipAndTakeSubscriber 7 x onNext()
SkipAndTakeSubscriber 8 x onNext()
line: 8 : Hello
SkipAndTakeSubscriber 9 x onNext()
line: 9 : Java 9
```