2.4 Übungen zu den Neuerungen aus JDK 9

Lernziel: Es sollen die Neuerungen aus JDK 9 anhand von Übungen kennengelernt werden. Hier geht es vor allem um Sprachbesonderheiten und API-Erweiterungen.

✍ Aufgabe 1 – Verbesserungen ARM und Variablen ✍

Gestalten Sie folgendes Programmfragment um, das recht aufwändig die erste Zeile einer Datei `abc.txt` einliest und auf der Konsole ausgibt. Erstellen Sie die Datei bitte selbst und legen diese direkt im Verzeichnis `resources` ab.

```
public static void main(final String[] args) throws FileNotFoundException
{
    final FileReader fileReader = new FileReader("./resources/abc.txt");
    final BufferedReader bufferedReadrr = new BufferedReader(fileReader);

    try
    {
        final String firstLine = bufferedReadrr.readLine();
        System.out.println("firstLine = " + firstLine);
    }
    catch (final Exception ioe)
    {
        try
        {
            fileReader.close();
        }
        catch (final IOException e)
        {
        }

        try
        {
            bufferedReadrr.close();
        }
        catch (final IOException e)
        {
        }
    }
}
```

Vereinfachen Sie den Sourcecode durch Einsatz von Automatic Resource Management (ARM), auch try-with-resources genannt. Verwenden Sie vor allem auch die mit JDK 9 neu eingeführte Syntax mit »effectively final«-Variablen im ARM.

✍ Aufgabe 2 – Process-Management ✍

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

✍ Aufgabe 3 – HTTP/2 ✍

Wandeln Sie folgende HTTP-Kommunikation so um, dass das neue HTTP/2-API aus JDK 9 zum Einsatz kommt.

```
private static void readOraclePageJdk8() throws MalformedURLException,
                                                 IOException
{
    final URL url = new URL("https://www.oracle.com/index.html");

    final String contents = readContent(url.openStream());
    System.out.println(contents);
}

public static String readContent(final InputStream is) throws IOException
{
    try (final InputStreamReader isr = new InputStreamReader(is);
         final BufferedReader br = new BufferedReader(isr))
    {
        final StringBuilder content = new StringBuilder();

        String line;
        while ((line = br.readLine()) != null)
        {
            content.append(line + "\n");
        }

        return content.toString();
    }
}
```

Tipp: Nutzen Sie die neuen Klassen `HttpRequest` und `HttpResponse` und erstellen Sie eine Methode `printResponseInfo(HttpResponse)`, die analog zu der obigen Methode `readContent(InputStream)` den Body ausliest und ausgibt. Zusätzlich soll noch der HTTP-Statuscode ausgegeben werden.

Bonus: Starten Sie die Abfragen mit `responseAsync()` asynchron und verarbeiten Sie das erhaltene `CompletableFuture<HttpResponse>` in etwa wie folgt:

```
final CompletableFuture<HttpResponse> asyncResponse = request.responseAsync();

waitForCompletionInSeconds(asyncResponse, 4);

if (asyncResponse.isDone())
{
    final HttpResponse response = asyncResponse.get();
    printResponseInfo(response);
}
else
{
    handleTimeout(asyncResponse);
}
```

✍ Aufgabe 4 – Die Klasse `Optional` ✍

Gegeben sei folgendes Programmfragment, das eine Personensuche ausführt und abhängig vom Ergebnis bei einem Treffer die Methode `doHappyCase(Person)` bzw. 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 den 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>`. Schauen Sie, wie das Ganze an Klarheit gewinnt.

✍ Aufgabe 6 – Private Interface-Methoden ✍

Versuchen Sie die Motivation für private Methoden in Interfaces zu verstehen. Vereinfachen Sie dazu folgenden Programmcode:

```
public interface PrivateMethodsExample
{
    public abstract int method();

    public default void calc1(float a, float b)
    {
        float avg = (a + b) / 2;
        System.out.println("sum: " + (a + b) + " / avg: " + avg);
    }

    public default void calc2(int a, int b)
    {
        int sum = a + b;
        float avg = (float)sum / 2;
        System.out.println("sum: " + sum + " / avg: " + avg);
    }
}
```

Beginnen Sie mit dem Extrahieren einer Hilfsvariable `sum` in der oberen Methode.

✍ Aufgabe 7 – 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?

Bonus: 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 8 – Streams ✍

Das Stream-API wurde um Methoden erweitert, die es erlauben, nur solange 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 8a: Ermitteln Sie aus dem Stream `values1` solange Werte, bis ein Leerstring gefunden wird. Geben Sie die Werte auf der Konsole aus.

Aufgabe 8b: Überspringen Sie in Stream `values2` die Werte, solange der Wert kleiner als 10 ist. Geben Sie die Werte auf der Konsole aus.

Aufgabe 8c: 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 9 – Reactive Streams ✍

Gegeben sei ein Programm mit einem `Publisher`, der Worte aus einer Datei liest und diese an registrierte `Subscriber` 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` 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 einfach 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` namens `SkipAndTakeSubscriber`, der die ersten n Vorkommen überspringt und danach m Vorkommen ausgibt. Danach soll die Kommunikation gestoppt werden, also der `Publisher` diesem `Subscriber` 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
```

✍ Aufgabe 10 – `jshell` und REPL ✍

Experimentieren Sie ein wenig mit der `jshell`. Definieren Sie eine Methode zur Berechnung der Fibonacci-Zahlen. Diese sind rekursiv wie folgt definiert:

$$\begin{aligned} fib(0) &= 0 \\ fib(1) &= 1 \\ fib(n) &= fib(n-1) + fib(n-2),\ \forall n \geq 2 \end{aligned}$$

Falls Sie es etwas einfacher mögen, berechnen Sie die Fakultät, die als das Produkt der natürlichen Zahlen von 1 bis n definiert ist: $n! = 1 * 2 * 3 * ... * n$, wobei im Speziellen $0! = 1$ gilt. Damit ist auch hier eine rekursive Definition folgendermaßen möglich:

$$\begin{aligned} 0! &= 1 \\ n! &= n * (n-1)!,\ \forall n \geq 1 \end{aligned}$$

Berechnen Sie die Fakultät oder die Fibonacci-Zahlen für die Werte 5 und 10.

Tipp Starten Sie mit folgenden Zeilen für die Fakultät:

```
long factorial(final long n)
   ...> {
   ...>     if (n < 0)
   ...>         throw new IllegalArgumentException("n must be non-negative");
   ...>
   ...>     if (n == 0)
   ...>         return 1;
```