Workshop:
Best of Java 9 bis 13
Übungen

Ablauf

Dieser Workshop gliedert sich in mehrere Vortragsteile, die den Teilnehmern die Thematik Java 9 bis 13 sowie die dortigen Neuerungen überblicksartig näherbringen. Im Anschluss daran sind jeweils einige Übungsaufgaben von den Teilnehmern – idealerweise in Gruppenarbeit – am Rechner zu lösen.

Voraussetzungen

1) Aktuelles JDK 11, idealerweise auch JDK 12/13, installiert
2) Aktuelles Eclipse installiert (Alternativ: NetBeans oder IntelliJ IDEA)

Teilnehmer

- Entwickler mit Java-Erfahrung sowie
- SW-Architekten, die Java 9 bis 13 kennenlernen/evaluieren möchten

Kursleitung und Kontakt

Michael Inden
CTO & Teamleiter SW-Entwicklung & Leiter ASMIQ Academy
ASMIQ AG, Geerenweg 2, 8048 Zürich
E-Mail: michael.inden@asmiq.ch
Kursangebot: https://asmiq.ch/
Blog: https://jaxenter.de/author/minden

Konfiguration Eclipse / IntelliJ

Bedenken Sie bitte, dass wir vor den Übungen noch einige Kleinigkeiten konfigurieren müssen.

- **Eclipse 2019-09: Installation von speziellem Plugin nötig**
- **Aktivierung von Preview-Features nötig**

- **Aktivierung von Preview-Features nötig**

PART 1/2: Syntax- und API-Erweiterungen in Java 9 bis 11

Aufgabe 1 – Kennenlernen von var

Lerne das neue reservierte Wort var mit seinen Möglichkeiten und Beschränkungen kennen.

Aufgabe 1a

Starte die JShell oder eine IDE deiner Wahl. Erstelle eine Methode funWithVar(). Definiere dort die Variablen name und age mit den Werten Mike bzw. 47.

```
void funWithVar()
{
	// TODO
}
```

Aufgabe 1b

Erweitere dein Know-how bezüglich var und Generics. Nutze es für folgende Definition. Erzeugen initial zunächst eine lokale Variable personsAndAges und vereinfache dann mit var:

```
Map.of("Tim", 47, "Tom", 7, "Mike", 47);
```

Aufgabe 2 – Collection-Factory-Methoden

Definiere eine Liste, eine Menge und eine Map mithilfe der in JDK 9 neu eingeführten Collection-Factory-Methoden namens of(). Als Ausgangsbasis dient nachfolgendes Programmfragment mit JDK 8. Nutze einen statischen Import wie folgt: import static java.util.Map.entry;

```
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);
}
```

Aufgabe 3 – Streams Take / Drop While

Extrahiere die Head- und die Body-Informationen mit geeigneten Prädikaten und den zuvor vorgestellten Methoden.

```
final Predicate<String> isBodyStart = // TODO
final Predicate<String> isBodyEnd = // TODO

final List<String> tokens = List.of("<html>",
      "<head>", "<title>This is TTLE</title>", "</head>",
      "<body>",
           "<h1>THIS IS THE H1 HEADER</h1>",
           "<p>Paragraph content</p>",
      "</body>",
    "</html>");

extractor(tokens, isBodyStart, isBodyEnd).forEach(System.out::println);
```

Tipp: Erstelle eine Hilfsmethode mit folgender Signatur:

```
private static List<String> extractor(final List<String> tokens,
                                      final Predicate<String> isStart,
                                      final Predicate<String> isEnd)
```

Aufgabe 4 – Die Klasse `Optional`

Gegeben sei folgende Methode, die eine Personensuche ausführt und abhängig vom Ergebnis bei einem Treffer die Methode `doHappyCase(Person)` bzw. ansonsten `doErrorCase()` aufruft.

```
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");
}
```

Gestalte das Programmfragment mithilfe der neuen Methoden aus der Klasse `Optional<T>` eleganter innerhalb einer Methode `findJdk9()`, die wie `findJdk8()` folgende Ausgaben produziert:

```
Result: Person: Tim
not found$
```

Aufgabe 5 – Die Klasse `LocalDate`

Lerne Nützliches in der Klasse `LocalDate` kennen.

Aufgabe 5a

Schreibe ein Programm, das alle Sonntage im Jahr 2017 zählt.

Aufgabe 5b

Schreibe ein Programm, dass alle Freitage der 13. in den Jahren 2013 bis 2017 ermittelt. Nutze folgende Zeilen als Ausgangspunkt:

```
final LocalDate start = LocalDate.of(2013, 1, 1);
final LocalDate end = LocalDate.of(2018, 1, 1);
```

Als Ergebnis sollten folgende Werte erscheinen:

```
[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 6: Strings

Die Verarbeitung von Strings wurde in Java 11 mit einigen nützlichen Methoden erleichtert.

Aufgabe 6a

Nutze folgenden Stream als Eingabe

```
Stream.of(2,4,7,3,1,9,5)
```

Realisiere eine Ausgabe, die die sieben Zahlen untereinander ausgibt, jeweils so oft wiederholt, wie die Ziffer, also verkürzt wie folgt:

```
22
4444
7777777
333
1
999999999
55555
```

Aufgabe 6b

Modifiziere die Ausgabe so, dass die Zahlen rechtsbündig mit maximal 10 Zeichen ausgegeben werden:

```
‘      4444’
‘   7777777’
‘ 999999999’
```

Tipp: Nutze eine Hilfsmethode

```
private static String formatRightAligned(final int num,
                                         final int desiredLength)
{
      // TODO
}
```

Aufgabe 6c KÜR

Modifiziere das Ganze so, dass nun statt Leerzeichen führende Nullen ausgegeben werden, etwa wie folgt:

```
‘0000004444’
‘0007777777’
‘0999999999’
```

BONUS: Erweitere das Ganze so, dass beliebige Füllzeichen genutzt werden können.

Aufgabe 7: Strings und Files

Bis Java 11 war es etwas mühsam, Texte direkt in eine Datei zu schreiben bzw. daraus zu lesen. Dazu gibt es nun die Methoden `writeString()` und `readString()` aus der Klasse `Files`. Schreibe mit deren Hilfe folgende Zeile in eine Datei.

```
1: One
2: Two
3: Three
```

Lies diese wieder ein und bereite daraus eine `List<String>` auf.

PART 3: Multi-Threading und Reactive Streams

Aufgabe 1 – Die Klasse `CompletableFuture<T>`

Frische dein Wissen zur Klasse `CompletableFuture<T>` auf.

Aufgabe 1a

Analysiere folgender Programmzeilen, die asynchron zur `main()`-Methode eine Datei einlesen. Danach werden zwei Filterungen definiert, die erst dann mit `thenApplyAsync()` ausgeführt werden, wenn die Datei tatsächlich eingelesen wurde. Durch den Zusatz `Async()` geschehen beide Filteraktionen parallel. Schließlich müssen die Ergebnisse wieder zusammengeführt werden. Dazu dient die Methode `thenCombine()`, wobei eine Kombinationsfunktion übergeben werden muss.

```
public static void main(final String[] args) throws IOException,
                                                     InterruptedException,
                                                     ExecutionException
{
    final Path exampleFile = Paths.get("./Example.txt");

    // Möglicherweise längerdauernde Aktion
    final CompletableFuture<List<String>> contents = CompletableFuture
                    .supplyAsync(extractWordsFromFile(exampleFile));
    contents.thenAccept(text -> System.out.println("Initial: " + text));

    // Filterungen parallel ausführen
    final CompletableFuture<List<String>> filtered1 =
                            contents.thenApplyAsync(removeIgnorableWords());
    final CompletableFuture<List<String>> filtered2 =
                            contents.thenApplyAsync(removeShortWords());

    // Verbinde die Ergebnisse
    final CompletableFuture<List<String>> result =
                                  filtered1.thenCombine(filtered2,
                                                        calcIntersection());

    System.out.println("result: " + result.get());
}

private static BiFunction<? super List<String>,
                          ? super List<String>,
                          ? extends List<String>> calcIntersection()
{
    return (list1, list2) ->
    {
        list1.retainAll(list2);
        return list1;
    };
}
```

Aufgabe 1b

Stelle dir vor, man würde Datenermittlungen, die eine Liste als Ergebnis liefern, parallel ausführen und möchte die Ergebnisse kombinieren. Wie ändert sich dann die Kombinationsfunktion? Schreibe den obigen Code um, sodass er zwei Methoden `retrieveData1()` und `retrieveData2()` sowie `combineResults()` (analog zu `calcIntersection()`) verwendet. Starte mit folgenden Zeilen:

```
public static void main(final String[] args) throws IOException,
                                                     InterruptedException,
                                                     ExecutionException
{
        final CompletableFuture<List<String>> data1 =
                              CompletableFuture.supplyAsync(()->retrieveData1());
```

Für zwei Listen mit Namen sollte das Ergebnis in etwa wie folgt sein:

```
retrieveData1(): ForkJoinPool.commonPool-worker-9
combineResults(): main
retrieveData2(): ForkJoinPool.commonPool-worker-2
result: [Jennifer, Lili, Carol, Tim, Tom, Mike]
```

Aufgabe 2 – Die Klasse `CompletableFuture<T>`

Experimentiere mit der Klasse `CompletableFuture<T>` und den in JDK 9 neu eingeführten Methoden `failedFuture()`, `orTimeOut()` und `completeOnTimeout()`. Nutze dein Wissen zu `exceptionally()` zum Behandeln von Exceptions während der Verarbeitung. Starte mit folgendem Grundgerüst und ergänze das Fehler- und Time-out-Handling.

```
public static void main(final String[] args) throws ExecutionException
{
    // CompletableFuture.// TODO
    //   .exceptionally(ex -> { System.out.println("ALWAYS FAILING"); return -1;});

    CompletableFuture.supplyAsync(() -> longRunningCreateMsg(5))
                     // TODO
                     .thenAccept(CompletableFutureJdk9Example::notifySubscribers);

    CompletableFuture.supplyAsync(() -> longRunningCreateMsg(5))
                     // TODO
                     .thenAccept(CompletableFutureJdk9Example::notifySubscribers);

    sleepInSeconds(10); // Give Completablefutures the chance to complete
}

public static String longRunningCreateMsg(final int durationInSecs)
{
    System.out.println(getCurrentThread() + " >>> longRunningCreateMsg");
    sleepInSeconds(durationInSecs);
    System.out.println(getCurrentThread() + " <<< longRunningCreateMsg");
    return "longRunningCreateMsg";
}
```

```
public static String getCurrentThread()
{
    return Thread.currentThread().getName();
}

public static void notifySubscribers(final String msg)
{
    System.out.println(getCurrentThread() + " notifySubscribers: " + msg);
}
```

Erwartet werden Ausgaben analog zu den Folgenden:

```
ALWAYS FAILING
ForkJoinPool.commonPool-worker-9 >>> longRunningCreateMsg
ForkJoinPool.commonPool-worker-2 >>> longRunningCreateMsg
CompletableFutureDelayScheduler notifySubscribers: TIMEOUT-FALLBACK
CompletableFutureDelayScheduler notifySubscribers: exception occurred:
java.util.concurrent.TimeoutException
ForkJoinPool.commonPool-worker-2 <<< longRunningCreateMsg
ForkJoinPool.commonPool-worker-9 <<< longRunningCreateMsg
```

Aufgabe 3 – Reactive Streams

Gegeben sei ein Programm `Exercise3_ReactiveStreamsExample` mit einem `Publisher<String>`, der Namen aus einer Liste an registrierte `Subscriber<String>` in der Methode `doWork()` veröffentlicht:

```
public class Exercise3_ReactiveStreamsExample
{
    public static void main(final String[] args) throws IOException,
                                                         InterruptedException
    {
        final NamePublisher publisher = new NamePublisher();
        publisher.subscribe(new ConsoleOutSubscriber());

        publisher.doWork();

        Thread.sleep(10_000); // auf das Ende der Verarbeitung warten
    }
}
```

Es kommt zu Ausgaben wie

```
2018-04-11T18:00:09.788635 onNext(): Tim
2018-04-11T18:00:10.742126 onNext(): Tom
…
```

Dazu ist der `Publisher<String>` wie folgt realisiert:

```
public class NamePublisher implements Flow.Publisher<String>
{
	private static final List<String> names = Arrays.asList("Tim", "Tom",
	   "Mike", "Alex", "Babs", "Jörg", "Karthi", "Marco", "Peter", "Numa“);

	private int counter = 0;
	private final SubmissionPublisher<String> publisher =
	                                                new SubmissionPublisher<>();

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

	public void doWork()
	{
		for (;;)
		{
			final String item = names.get(counter++ % names.size());
			publisher.submit(item);

			try
			{
				Thread.sleep(1_000);
			}
			catch (InterruptedException e)
			{ // ignore }
		}
	}
}
```

Zur Protokollierung dient folgende einfache Klasse `ConsoleOutSubscriber`, die alle Vorkommen auf der Konsole auflistet:

```
class ConsoleOutSubscriber 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();
	}
}
```

Implementiere basierend auf der obigen Klasse `ConsoleOutSubscriber` 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 `NamePublisher` diesem `Subscriber<String>` keine Daten mehr senden. Erwartet werden Ausgaben in etwa wie folgt:

```
SkipAndTakeSubscriber - Subscription:
java.util.concurrent.SubmissionPublisher$BufferedSubscription@23c34259
SkipAndTakeSubscriber 1 x onNext()
SkipAndTakeSubscriber 2 x onNext()
SkipAndTakeSubscriber 3 x onNext()
Mike
SkipAndTakeSubscriber 4 x onNext()
Alex
SkipAndTakeSubscriber 5 x onNext()
Babs
SkipAndTakeSubscriber 6 x onNext()
Jörg
SkipAndTakeSubscriber 7 x onNext()
Karthi
```

PART 4: HTTP/2

Aufgabe 1 – HTTP/2

Gegeben sei folgende HTTP-Kommunikation, die auf die Webseite von Oracle zugreift:

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

    final String content = readContent(connection.getInputStream());
    System.out.println(content);
}

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();
    }
}
```

Aufgabe 1a

Wandle den Sourcecode so um, dass das neue HTTP/2-API zum Einsatz kommt. Nutze die Klassen `HttpRequest` und `HttpResponse` und erstelle eine Methode `printResponseInfo(HttpResponse)`, die den Body ausliest und ausgibt. Zusätzlich soll noch der HTTP-Statuscode ausgegeben werden. Starte mit folgendem Programmfragment:

```
private static void readOraclePageJdk11() throws URISyntaxException,
                                                 IOException,
                                                 InterruptedException
{
    final URI uri = new URI("https://www.oracle.com/index.html");
    final HttpClient httpClient = // TODO
    final HttpRequest request = // TODO
    final BodyHandler<String> asString = // TODO
    final HttpResponse<String> response = // TODO

    printResponseInfo(response);
}
```

Aufgabe 1b

Starte die Abfragen durch Aufruf von `sendAsync()` asynchron und verarbeite das erhaltene `CompletableFuture<HttpResponse>`.

PART 5: Neuerungen in Java 13

Aufgabe 1 – Syntaxänderungen bei switch

Vereinfache folgenden Sourcecode mit einem herkömmlichen switch-case durch die neue Syntax von Java 13.

```
private static void dumpEvenOddChecker(int value)
{
      String result;

      switch (value)
      {
      case 1, 3, 5, 7, 9:
            result = "odd";
            break;

      case 0, 2, 4, 6, 8, 10:
            result = "even";
            break;

      default:
            result = "only implemented for values < 10";
      }

      System.out.println("result: " + result);
}
```

Aufgabe 1a

Nutze zunächst nur die Arrow-Syntax, um die Methode kürzer und übersichtlicher zu schreiben.

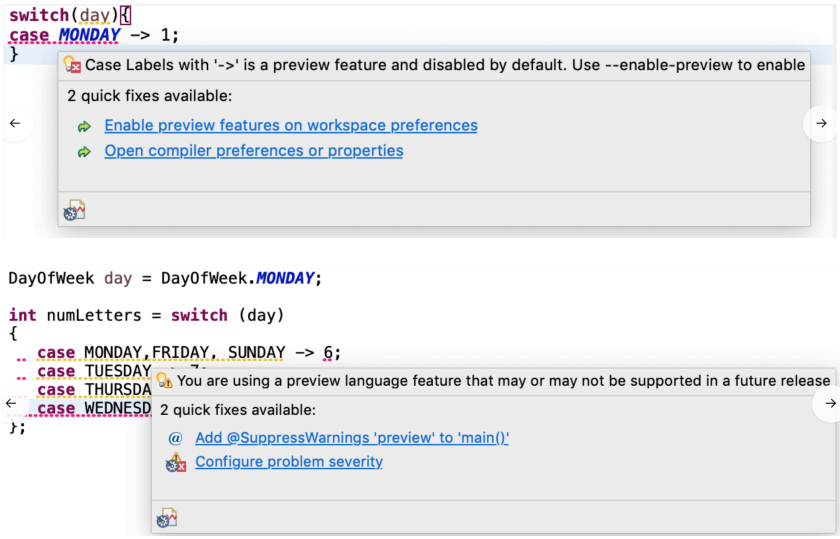
Aufgabe 1b

Verwende nun noch die Möglichkeit, Rückgaben direkt zu spezifizieren und ändere die Signatur in `String dumpEvenOddChecker(int value)`

Aufgabe 1c

Wandle das Ganze so ab, dass du die Spezialform «break mit Rückgabewert» (Java 12) bzw. «yield mit Rückgabewert» (Java 13) verwendest.

Tipps: Aktivierung des Preview-Features und Unterdrücken von Warnungen:



Aufgabe 2 – Text Blocks

Vereinfache folgenden Sourcecode mit einem herkömmlichen String, der über mehrere Zeilen geht und nutze die in Java 13 eingeführte Syntax.

```
String multiLineStringOld = "THIS IS\n" +
            "A MULTI\n" +
            "LINE STRING\n" +
            "WITH A BACKSLASH \\\n";

String multiLineHtmlOld = "<html>\n" +
            "    <body>\n" +
            "        <p>Hello, world</p>\n" +
            "    </body>\n" +
            "</html>";

String java13FeatureObjOld = ""
                + "{\n"
                + "    version: \"Java13\",\n"
                + "    feature: \"text blocks\",\n"
                + "    attention: \"preview!\"\n"
                + "}\n";
```

Aufgabe 3 – Text Blocks mit Platzhaltern

Vereinfache folgenden Sourcecode mit einem herkömmlichen String, der über mehrere Zeilen geht und nutze die in Java 13 eingeführte Syntax:

```
String multiLineStringWithPlaceHoldersOld =
       String.format("HELLO \"%s\"!\n" +
                     "  HAVE %s\n" +
                     "  NICE \"%s\"!",
                     new Object[]{"WORLD", "A", "DAY"});

System.out.println(multiLineStringWithPlaceHoldersOld);
```

Produziere folgende Ausgaben mit der neuen Syntax:

```
HELLO "WORLD"!
  HAVE A
  NICE "DAY"!
```