Workshop: Neues in Java 12 — Best of Java 9, 10 und 11 Übungen

Ablauf

Dieser Workshop gliedert sich in mehrere Vortragsteile, die den Teilnehmern die Thematik Java 9, 10, 11 und 12 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, installiert
2) Aktuelles Eclipse installiert (Alternativ: NetBeans oder IntelliJ IDEA)

Teilnehmer

- Entwickler mit Java-Erfahrung sowie
- SW-Architekten, die Java 9, 10, 11 und 12 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

PART 1: Syntax- und API-Erweiterungen in Java 9

Lernziel: Kennenlernen von Syntax-Neuerungen und verschiedenen API-Erweiterungen in Java 9 anhand von Beispielen.

Aufgabe 1 – Process-Management

Ermitteln Sie die Process-ID und weitere Eigenschaften des aktuellen Prozesses. Nutze dazu das Interface `ProcessHandle` und seine Methoden.

Aufgabe 1a

Wie viel CPU-Zeit hat der aktuelle Prozess bislang verbraucht und wann wurde er gestartet?

Aufgabe 1b

Wie viele Prozesse werden momentan insgesamt ausgeführt?

Aufgabe 1c

Liste alle Java-Prozesse auf. Nutze dazu ein `Predicate<Info> isJavaProcess`.

Aufgabe 1d

Versuche einmal den aktuellen Prozess zu terminieren. Was passiert dann?

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

Das Stream-API wurde um Methoden erweitert, die es erlauben, nur solange Elemente zu lesen, wie eine Bedingung erfüllt ist bzw. solange Elemente zu überspringen, wie 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

Ermittle aus dem Stream `values1` solange Werte, bis ein Leerstring gefunden wird. Gib die Werte auf der Konsole aus.

Aufgabe 3b

Überspringe im Stream `values2` die Werte, solange der Wert kleiner als 10 ist. Gib 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 – Streams

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 5 – 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 6 – 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 nachfolgend gezeigt 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)
{
    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)
{
    return Optional.empty();
}

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

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

Aufgabe 7 BONUS – Die Klasse `Optional` schon in JDK 8 erweitern

Mit Java 9 würde nützliche Funktionalität in die Klasse Optional<T> aufgenommen. Erstelle eine Utility-Klasse `OptionalUtils`, die die beiden Methoden `ifPresentOrElse()` sowie `or()` für JDK 8 bereitstellt. Als Hilfestellung dienen folgende Signaturen:

```
publice static <T> void ifPresentOrElse(Optional<T> optional,
                                        Consumer<? super T> action,
                                        Runnable elseAction)

public static <T> Optional<T> or(Optional<T> first,
                 Supplier<? extends Optional<? extends T>> supplier)

public static <T> Optional<T> or(Optional<T> first,
                 Supplier<? extends Optional<? extends T>>... suppliers)

public static <T> Stream<T> stream(Optional<T> optValue)
```

Aufgabe 8 – Die Klasse `Collectors`

Aufgabe 8a

Filtere alle langen Namen (> 5 Zeichen) aus der gegebenen Namensliste und gib die Werte auf der Konsole aus.

```
final Stream<String> names = Stream.of("Tim", "Tom", "Michael",
                                       "Thomas", "Karthikeyan", "Marius");
```

Aufgabe 8b

Gruppiere die zuvor gefilterten Namen gemäß dem Anfangsbuchstaben und gib die Werte auf der Konsole aus. Das erwartete Ergebnis ist:

```
{T=[Thomas], K=[Karthikeyan], M=[Michael, Marius]}
```

Aufgabe 8c

Kombiniere nun das Wissen zu einer Filterung beim Gruppieren. Nutze folgende Map als Ausgangsbasis, um lediglich alle Erwachsenen gemäß dem Anfangsbuchstaben zu gruppieren:

```
final Map<String, Long> personAgeMapping = Map.of("Tim", 47L, "Tom", 12L,
                                                  "Michael", 47L, "Max", 5L);
```

Das erwartete Ergebnis ist:

```
{T=[Tim=47], M=[Michael=47, Max=5]}
```

Aufgabe 8d

Vertiefe dein Wissen zur Kombination von Kollektoren. Das obige Ergebnis soll mithilfe Kollektor `mapping()` es auf den Namen eingeschränkt werden. Das erwartete Ergebnis ist:

```
{T=[Tim], M=[Max, Michael]}
```

Aufgabe 8e

Verdichte einen Stream mit mehreren Mengen von Buchstaben zu einer Menge.

```
final Stream<Set<String>> characters = Stream.of(Set.of("a", "c", "e"),
                                                 Set.of("a", "b"),
                                                 Set.of("a", "b", "c", "d"),
                                                 Set.of("a", "b", "c", "d", "f"));
```

Das erwartete Ergebnis ist:

```
[a, b, c, d, e, f]
```

Aufgabe 9 BONUS – Die Klasse `Collectors` schon in JDK 8 erweitern

Mit Java 9 wurde nützliche Funktionalität in die Klasse `Collectors` aufgenommen. Erstelle eine Utility-Klasse `CollectorUtils`, die die Methode `filtering()` für JDK 8 bereitstellt. Als Hilfestellung dienen folgendes Grundgerüst:

```
public static <T, A, R> Collector<T, A, R> filtering(Predicate<? super T> filter,
                                                     Collector<T, A, R> collector)
{
    return Collector.of(...,
                   (accumulator, input) -> {
                      ...
                   },
                   ...,
                   ...);
}
```

Tipp: Benutze folgende Methode `of()` aus dem `Collector`

```
public static<T, A, R> Collector<T, A, R> of(Supplier<A> supplier,
                                             BiConsumer<A, T> accumulator,
                                             BinaryOperator<A> combiner,
                                             Function<A, R> finisher,
                                             Characteristics... characteristics) {
```

sowie die folgenden Methoden:

`collector.supplier()`, `collector.combiner()` und `collector.finisher()`

PART 2: Multi-Threading und Reactive Streams

Lernziel: In diesem Abschnitt beschäftigen wir uns mit fortgeschritteneren APIs, etwa `CompletableFuture<T>` 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 3: Diverses

Lernziel: In diesem Abschnitt wollen wir einige praktische API-Erweiterungen kennenlernen: `Arrays` und `Objects` sowie `LocalDate` und `InputStream`.

Aufgabe 1 – Die Klasse `Objects`

Setze die Neuerungen in der Klasse `Objects` ein.

Aufgabe 1a

Nutze die Klasse `java.util.Objects` zur Vereinfache folgendes Programmfragments:

```
public void adjustStudyStartYear_OldStyle(Person person, Year beginOfStudy)
{
	if (person == null)
	{
		throw new NullPointerException("person cannot be null!");
	}
	if (beginOfStudy == null)
	{
		throw new NullPointerException("beginOfStudy cannot be null!");
	}
	// Rest of method
}
```

Aufgabe 1b

Nutze die neue Funktionalität, um einen Default-Parameter zu simulieren, wie es andere Programmiersprachen erlauben. Schreibe folgendes Code-Fragment um:

```
public Instant dateToInstantWithNowAsDefault_OldStyle(final Date origDate)
{
	Date adjustedDate = origDate;
	if (adjustedDate == null)
	{
		adjustedDate = new Date();
	}

	return adjustedDate.toInstant();
}
```

Aufgabe 2 – Die Klasse `Objects`

Zum Teil kann die Berechnung des Fallback-Werts einige Zeit dauern. Es ist dann ungünstig, die Fallback-Berechnung allerdings immer auszuführen, etwa wie im folgenden Code:

```
public List<String> provideDefault_OldStyle_Bad(final List<String> names)
{
	List<String> namesWithFallback = longRunningCalcOfNames();
	if (names != null && !names.isEmpty())
	{
		namesWithFallback = names;
	}
	return namesWithFallback;
}
```

Eine verbesserte Variante arbeitet folgendermaßen:

```
public List<String> provideDefault_OldStyle(final List<String> names)
{
       List<String> namesWithFallback = names;
       if (namesWithFallback == null || namesWithFallback.isEmpty())
       {
              namesWithFallback = longRunningCalcOfNames();
       }

       return namesWithFallback;
}
```

Realisierte diese Funktionalität mit den Neuerungen aus Java 9.

Aufgabe 3 – Die Klasse `Arrays`

Ermittle, ob der durch `search` beschriebene Text im Originaltext `originaltext` vorkommt. Wandle dazu die Strings zunächst in den Typ `byte[]` um und finde die Position der Übereinstimmung von `search`:

```
final String originaltext = "BLABLASECRET-INFO:42BLABLA";
final String search = "SECRET-INFO:42";
```

Aufgabe 4 – Die Klasse `Arrays`

Ermittle für die beiden wie folgt gegebenen Arrays

```
final byte[] first =  { 1,1,0,1,1,0,1,1,1,1,0,1,1 };
final byte[] second = { 1,1,0,1,1,0,1,0,1,1,1,1,1 };
```

a) die erste Abweichung und
b) die darauffolgende Abweichung.

Aufgabe 5 – Die Klasse `Arrays`

Vergleiche die beiden wie folgt gegebenen Arrays:

```
final byte[] first =  "ABCDEFGHIJk".getBytes();
final byte[] second = "XYZABCDEXYZ".getBytes();
```

a) Welches ist «grösser»?
b) Ab welcher Position ist `first` grösser als `second`, wenn man von `second` immer bei jedem weiteren Vergleich einen Buchstaben vorne entfernt. Protokolliere zum besseren Verständnis die verglichenen Werte für alle Start-Positionen.

Aufgabe 6 – Die Klasse `LocalDate`

Lerne Nützliches in der Klasse `LocalDate` kennen.

Aufgabe 6a

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

Aufgabe 6b

Liste die Sonntage auf, startend mit dem 5. und endend mit dem 10. Das Ergebnis sollte wie folgt sein:

```
[2017-02-05, 2017-02-12, 2017-02-19, 2017-02-26, 2017-03-05]
```

Aufgabe 6 Bonus

Erstelle eine Übersicht über die Anzahl jedes Wochentags im Jahr 2012, etwa wie folgt:

```
Day: MONDAY / count: 53
Day: TUESDAY / count: 52
Day: WEDNESDAY / count: 52
Day: THURSDAY / count: 52
Day: FRIDAY / count: 52
Day: SATURDAY / count: 52
Day: SUNDAY / count: 53
```

Tipp: Nutze eine Hilfsmethode mit folgender Signatur:

```
private static Stream<LocalDate> allBetween(DayOfWeek dayOfWeek,
                                            LocalDate start,
                                            LocalDate end)
```

Aufgabe 7 – Die Klasse `LocalDate`

Lerne Nützliches in der Klasse `LocalDate` kennen.

Aufgabe 7a

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]
```

Gruppiere die Vorkommen nach Jahr. Es sollten folgende Ausgaben erscheinen:

```
Year 2013: [2013-09-13, 2013-12-13]
Year 2014: [2014-06-13]
Year 2015: [2015-02-13, 2015-03-13, 2015-11-13]
Year 2016: [2016-05-13]
Year 2017: [2017-01-13, 2017-10-13]
```

Aufgabe 7b
Wie viele mal gab es den 29. Februar zwischen Anfang 2010 und Ende 2017?

```
final LocalDate start2010 = LocalDate.of(2010, 1, 1);
final LocalDate end2017 = LocalDate.of(2018, 1, 1);
```

Aufgabe 7c
Wie häufig war dein Geburtstag ein Sonntag zwischen Anfang 2010 und Ende 2017? Für den 7. Februar sollte folgendes Ergebnis berechnet werden:

```
My Birthday on Sunday between 2010-2017: [2010-02-07, 2016-02-07]
```

Aufgabe 8 – Die Klasse `Duration`

Berechne die Anzahl an Stunden und die jeweiligen Untereinheiten: Gegeben sei eine Zeitspanne von 3 Stunden, 45 Minuten und 57 Sekunden.

a) Runde das auf Stunden sowie auf Stunden und Minuten.
b) Wandle die Zeitdauer in Minuten und was ist der reine Minutenanteil?
c) Wie oft entspricht das 7 Minuten und wie viel Mal passen 22 Sekunden in die Zeitspanne?

Aufgabe 9 – Die Klasse `InputStream`

Schreibe ein Programm, das eine Kopie einer Datei, gegeben durch einen Dateinamen, erstellt. Vereinfache folgenden, auf Java 8 basierenden Sourcecode:

```
private static void copyFileUsingStream(final File source,
                                        final File dest) throws IOException
{
    try (final InputStream is = new FileInputStream(source);
         final OutputStream os = new FileOutputStream(dest))
    {
        final byte[] buffer = new byte[2048];
        int length;
        while ((length = is.read(buffer)) > 0)
        {
            os.write(buffer, 0, length);
        }
    }
}
```

Aufgabe 10 – Die Klasse `InputStream`

Vereinfache folgenden, auf Java 8 basierenden Sourcecode und schreibe Methoden, die folgende Aktionen ausführen:

a) Lese die Daten aus einem `InputStream` vollständig ein.
b) Lese nur die ersten 6 Zeichen aus einem `InputStream` ein.
c) Transferiere alle Daten aus einem `InputStream` in einen `OutputStream`.

```
public static void main(final String[] args) throws IOException
{
    final byte[] buffer = { 72, 65, 76, 76, 79 };

    final byte[] resultJdk8 = readAllBytesJdk8(/* TODO */);
    System.out.println(Arrays.toString(resultJdk8));

    transferToJdk8(/* TODO */ System.out);
}

private static byte[] readAllBytesJdk8(final InputStream is) throws IOException
{
    try (final ByteArrayOutputStream os = new ByteArrayOutputStream())
    {
        transferToJdk8(is, os);
        os.flush();

        return os.toByteArray();
    }
}

private static void transferToJdk8(final InputStream in,
                                   final OutputStream out) throws IOException
{
    final byte[] buffer = new byte[1024];
    int len;
    while ((len = in.read(buffer)) != -1)
    {
        out.write(buffer, 0, len);
    }
}
```

Tipp: Nutze als Eingabe einen `ByteArrayInputStream`.

PART 4: Neuerungen in Java 10

Lernziel: In diesem Abschnitt beschäftigen wir uns mit Syntaxerweiterungen und API-Neuerungen in Java 10.

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`. Gib anschliessend deren Typ aus.

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

Tipp: Für die zweite Variable hilft. `((Object)age).getClass().`

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 1c

Vereinfache folgende Definition mit `var`. Was ist zu beachten? Worin liegt der Unterschied?

```
List<String> names = new ArrayList<>();
ArrayList<String> names2 = new ArrayList<>();
```

Aufgabe 1d

Wieso führen folgende Lambdas zu Problemen? Wie löst man diese?

```
var isEven = n -> n % 2 == 0;
var isEmpty = String::isEmpty;
```

Wieso kompiliert dann aber Folgendes?

```
Predicate<Long> isEven = n -> n % 2 == 0;
var isOdd = isEven.negate();
```

Aufgabe 2 – Unmodifiable Collections

In Java 9 wurden Unmodifiable Collections und Collection Factory Methods eingeführt. Java 10 bietet nun das Erzeugen unveränderlicher Kopien und spezielle Kollektoren für Streams.

Aufgabe 2a

Welche zwei Varianten bietet Java 10, um eine unveränderliche Kopie folgender Liste zu erstellen?

```
List<String> words = List.of("Hello", "World");
List<String> copy1 = null; /* TODO */
List<String> copy2 = null; /* TODO */
```

Welchen Typ haben die entstehenden Kopien? Gib diesen auf der Konsole aus.

Aufgabe 2b

Worin liegt der Unterschied innerhalb der entstehenden unmodifizierbaren Listen, wenn man eine Modifikation im Array vornimmt?

```
final String[] nameArray = { "Tim", "Tom", "Mike" };

// 3 Varianten von unmodifizierbaren Listen
final List<String> names1 = Arrays.asList(nameArray);
final List<String> names2 = List.of(nameArray);
final List<String> names3 = List.of("Tim", "Tom", "Mike");

// Modifikation im Array
nameArray[2] = "Michael";
```

Was geschieht beim Ändern in der Liste analog zu Folgendem?

```
names1.set(1, "XXX");
```

Aufgabe 2c

Was unterscheidet die drei Varianten von Kopien bzw. Wrapper?

```
List<String> names = List.of("Tim", "Tom", "Mike");

// 3 Varianten Kopien von unmodifizierbaren Listen zu erzeugen
List<String> copy1 = new ArrayList<>(names);
List<String> copy2 = List.copyOf(names);
List<String> wrapper = Collections.unmodifiableList(names);
```

Lassen sich Elemente ändern? Lassen sich Elemente löschen? Befülle nachfolgende Tabelle.

		Löschen	Ändern
copy1			
copy2			
wrapper			

Tipp: Nutze unter anderem folgende Hilfsmethode:

```
private static void tryModification(final String info,
                                    final List<String> list)
{
      try
      {
            list.set(1, "XXX");
            System.out.println(info + ": " + list);
      }
      catch (java.lang.UnsupportedOperationException ex)
      {
            System.out.println(info + ": set() not allowed");
      }
}
```

Aufgabe 3: Erweiterung in `Optional<T>`

Gegeben sei eine simple, aber unsichere Wertextraktion aus einem `Optional<T>`:

```
static <T> T badStyleExtractValue(final Optional<T> opt)
{
      T value = opt.get();
      return value;
}
```

Aufgabe 3a

Schreibe diese mit dem neuen Java-10-API so um, dass der potenziell unsichere Zugriff offensichtlich wird.

Aufgabe 3b

Wie würde man die Methode oder Aufrufer dann umgestalten?

Aufgabe 4: Erweiterungen in der Klasse `Reader`

Transferiere den Inhalt aus einem `StringReader` in eine Datei `hello.txt`. Lies diese wieder ein und gib den Inhalt aus. Ergänze folgende Zeilen:

```
var textFile = new File("hello.txt");
var sr = new StringReader("Hello\nWorld");
try (Writer bfw = null /*TODO*/)
{
      // TODO
}

var sw = new StringWriter();
try (Reader bfr = null /*TODO*/)
{
      // TODO
}
```

PART 5: Neuerungen in Java 11

Lernziel: In diesem Abschnitt beschäftigen wir uns mit Erweiterungen und API-Neuerungen in Java 11.

Aufgabe 1 – Direkte Kompilierung und Ausführung

Schreibe eine Klasse `HelloWorld` im Package `direct.compilation` und speichere diese in einer gleichnamigen Java-Datei. Führe diese direkt mit dem Kommando `java` aus.

Aufgabe 2: Erweiterungen in `Optional<T>`

Vereinfache folgendes Konstrukt mit der neuen Methode `isEmpty()` aus der Klasse `Optional<T>`:

```
static <T> Stream<T> asStream(final Optional<T> opt)
{
	if (!opt.isPresent())
	{
		return Stream.empty();
	}

	return Stream.of(opt.get());
}
```

Aufgabe 3: Strings

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

Aufgabe 3a

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 3b
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 3c
Modifiziere das Ganze so, dass nun statt Leerzeichen führende Nullen ausgegeben werden, etwa wie folgt:

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

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

Aufgabe 3d
Modifiziere die Ausgabe so, dass die grössten Zahlen zuletzt ausgegeben werden. Finde mindestens zwei Varianten:

```
Stream.of(2,4,7,3,1,9,5).sorted().map(mapper1)
Stream.of(2,4,7,3,1,9,5).map(mapper2)
```

Worin liegt der Unterschied?

Aufgabe 4: Predicates

Vereinfache folgende Predicates bezüglich der Negation:

```
Predicate<Long> isEven = n -> n % 2 == 0;
var isOdd = isEven.negate();

Predicate<String> isBlank = String::isBlank;
var notIsBlank = isBlank.negate();
```

Aufgabe 5: 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.

Aufgabe 6 – HTTP/2

Gegeben sei folgende HTTP-Kommunikation, die auf die Webseite von Oracle zugreift und diese textuell aufbereitet.

```
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 6a

Wandle den Sourcecode so um, dass das neue HTTP/2-API aus JDK 9 zum Einsatz kommt. Nutze die Klassen `HttpRequest` und `HttpResponse` und erstelle 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.

Starte mit folgendem Programmfragment:

```
private static void readOraclePageJdk9() 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);
}

private static void printResponseInfo(final HttpResponse<String> response)
{
    final int responseCode = response.statusCode();
    final String responseBody = response.body();
    final HttpHeaders headers = response.headers();

    System.out.println("Status:  " + responseCode);
    System.out.println("Body:    " + responseBody);
    System.out.println("Headers: " + headers.map());
}
```

Aufgabe 6b

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

PART 6: Neuerungen in Java 12

Lernziel: In diesem Abschnitt beschäftigen wir uns mit Erweiterungen und API-Neuerungen in Java 12.

Aufgabe 1 – Syntaxänderungen bei switch

Vereinfache folgenden Sourcecode mit einem herkömmlichen switch-case durch die neue Syntax von Java 12.

```
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.

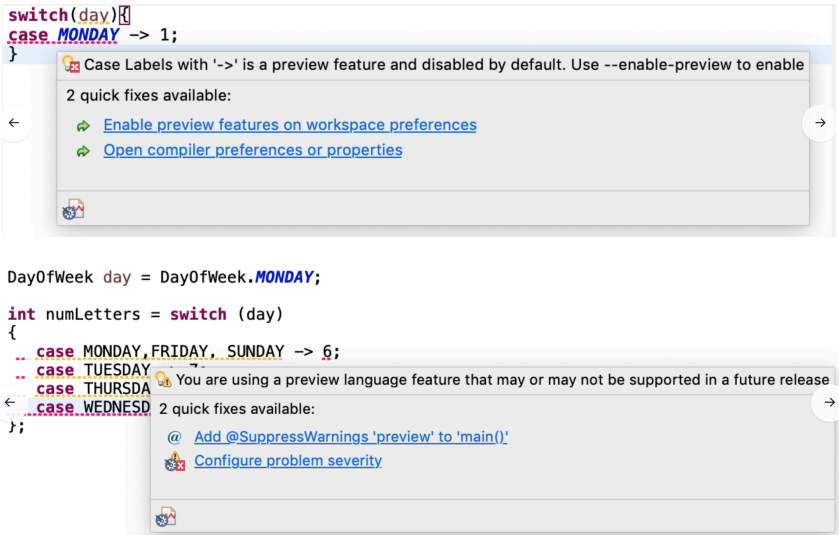
Aufgabe 1b

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» verwendest.

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



Aufgabe 2: Die Klasse `CompactNumberFormat`

Schreibe ein Programm, um die Kurzversionen für 1.000, 1.000.000 und 1.000.000.000 abhängig von Locale und Style auszugeben und zu parsen. Verwende die Locale GERMANY für SHORT und ITALY für LONG.

Nutze die nachfolgenden Werte zum Parsing:

```
List.of("13 KILO", "1 Mio.", "1 Mrd.")
List.of("1 mille", "1 milione")
```

Aufgabe 3: Strings

Die Verarbeitung von Strings wurde in Java 12 um zwei Methoden erweitert. Lerne hier zunächst `indent()` genauer kennen.

Aufgabe 3a

Rücke die folgende Eingabe um 7 Zeichen ein, gib diese aus und entferne wieder 3 Zeichen von der Einrückung.

```
String originalString = "first_line\nsecond_line\nlast_line";
```

Aufgabe 3b

Was passiert, wenn man einen linksbündigen Text mit negativen Werten für den Indent versieht? Was passiert, wenn man für die nachfolgende Eingabe, einen Indent von -10 nutzt?

```
String multipleIndendedString =
       "class A {\n    public static void main(String[] args) {" +
       "\n        System.out.println(\"Hello\"";
```

Aufgabe 4: Strings

Die Verarbeitung von Strings wurde in Java 12 um zwei Methoden erweitert. Lerne hier `transform()` genauer kennen. Gegeben sei dazu folgende kommaseparierte Eingabe:

```
var csvText = "HELLO,WORKSHOP,PARTICIPANTS,!,LET'S,HAVE,FUN";
```

Aufgabe 4a

Wandle diese vollständig in Kleinbuchstaben und ersetze die Kommas durch Leerzeichen.

Aufgabe 4b

Nutze andere Transformationen und ersetze HELLO mit dem Schweizer Gruß «GRÜEZI», spalte das Ganze dann in Einzelbestandteile auf, sodass folgende Liste als Ergebnis entsteht:

```
[GRÜEZI, workshop, participants, !, let's, have, fun]
```

Aufgabe 5: Teeing-Kollektor

Nutze den Teeing-Kollektor, um in einem Durchlauf sowohl das Minimum als auch das Maximum zu finden. Beginne mit folgenden Zeilen:

```
Stream<String> values = Stream.of("CCC", "BB", "A", "DDDD");
List<Optional<String>> optMinMax = values.collect(teeing(…
```

Aufgabe 6: Teeing-Kollektor

Variiere die `BiFunction`, um die Ergebnisse des Teeing-Kollektors geeignet zu beeinflussen. Beginne mit folgenden Zeilen und ergänze diese an den markierten Stellen:

```
var names = Stream.of("Michael", "Tim", "Tom", "Mike", "Bernd");

final Predicate<String> startsWithMi = text -> text.startsWith("Mi");
final Predicate<String> isShort = text -> text.length() <= 4;

final BiFunction<List<String>, List<String>, List<List<String>>>
      combineResults = (list1, list2) -> List.of(list1, list2);

final BiFunction<List<String>, List<String>, Set<String>>
      combineResultsUnique = null; // TODO;

final BiFunction<List<String>, List<String>, Set<String>>
      combineResultsIntersection = null; // TODO;

var result = names.collect(teeing(
                           filtering(startsWithMi, toList()),
                           filtering(isShort, toList()),
                           combineResults));
```

Die erwarteten Resultate sind mit

- `combineResults: [[Michael, Mike], [Tim, Tom, Mike]]`
- `combineResultsUnique: [Mike, Tom, Michael, Tim]`
- `combineResultsIntersection: [Mike]`

Aufgabe 7: Teeing-Kollektor

Nutze einen Teeing-Kollektor, um in einem Durchlauf sowohl alle europäischen Städte namentlich als auch die Anzahl der Städte in Asien zu ermitteln. Beginne mit folgenden Zeilen:

```
Stream<City> exampleCities = Stream.of(
                  new City("Zürich", "Europe"),
                  new City("Bremen", "Europe"),
                  new City("Kiel", "Europe"),
                  new City("San Francisco", "America"),
                  new City("Aachen", "Europe"),
                  new City("Hong Kong", "Asia"),
                  new City("Tokyo", "Asia"));

Predicate<City> isInEurope = city -> city.locatedIn("Europe");
Predicate<City> isInAsia = city -> city.locatedIn("Asia");

var result = exampleCities.collect(teeing(…
```

Gegeben sei noch die Klasse City wie folgt:

```
static class City
{
      private final String name;
      private final String region;

      public City(final String name, final String region)
      {
            this.name = name;
            this.region = region;
      }

      public String getName()
      {
            return name;
      }

      public String getRegion()
      {
            return region;
      }

      public boolean locatedIn(final String region)
      {
            return this.region.equalsIgnoreCase(region);
      }
}
```