

Workshop: Best of Java 9 bis 17

Übungen

Ablauf

Dieser Workshop gliedert sich in mehrere Vortragsteile, die den Teilnehmern die Thematik Java 9 bis 17 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 17 installiert
- 2) Aktuelles Eclipse installiert (Alternativ: NetBeans oder IntelliJ IDEA)

Teilnehmer

- Entwickler mit Java-Erfahrung sowie
- SW-Architekten, die Java 9 bis 17 kennenlernen/evaluieren möchten

Kursleitung und Kontakt

Michael Inden

Freiberuflicher Buchautor, Trainer und Konferenz-Speaker

E-Mail: michael.inden@hotmail.com

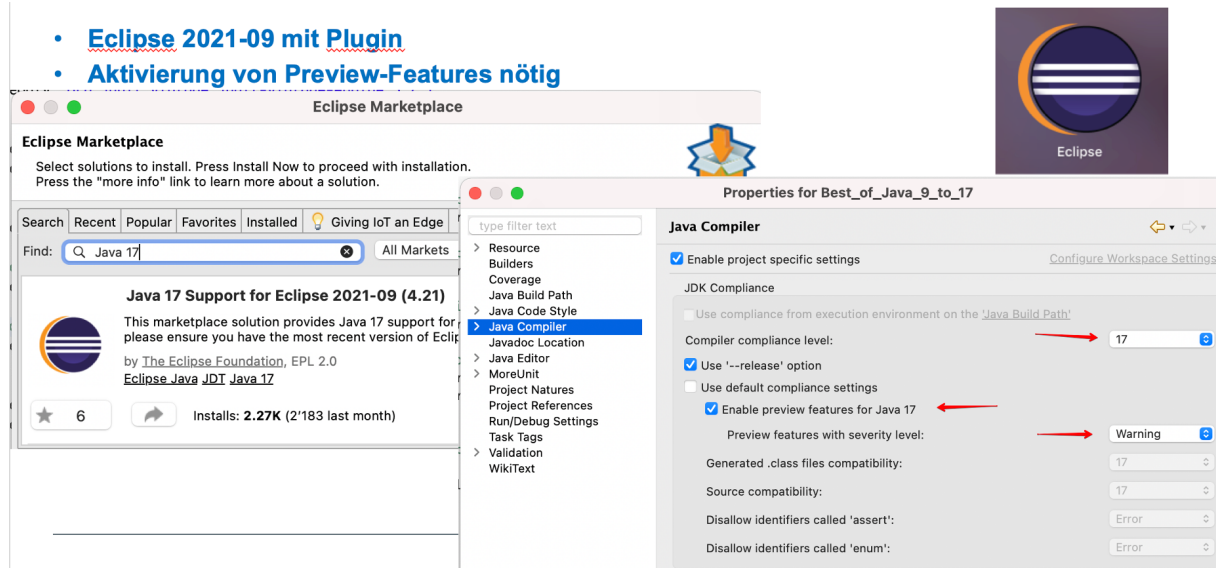
Blog: <https://jaxenter.de/author/minden>

Weitere Kurse (Java, Unit Testing, Design Patterns, JPA, Spring) biete ich gerne auf Anfrage als Online- oder Inhouse-Schulung an.

Konfiguration Eclipse / IntelliJ

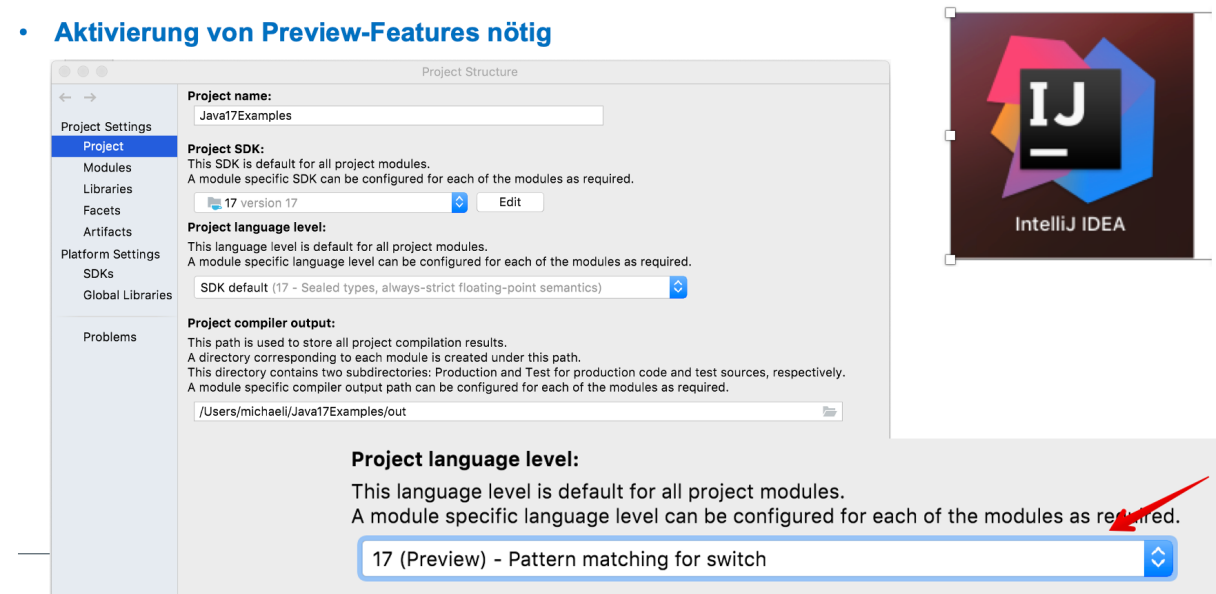
Bedenken Sie bitte, dass wir vor den Übungen noch einige Kleinigkeiten bezüglich Java / JDK und Compiler-Level konfigurieren müssen.

- [Eclipse 2021-09 mit Plugin](#)
- [Aktivierung von Preview-Features nötig](#)



The screenshot shows the Eclipse Marketplace interface on the left, displaying the 'Java 17 Support for Eclipse 2021-09 (4.21)' plugin. On the right, the 'Properties for Best_of_Java_9_to_17' dialog is open, specifically the 'Java Compiler' tab. Red arrows point to the 'Compiler compliance level' set to '17', the 'Enable preview features for Java 17' checkbox which is checked, and the 'Preview features with severity level' dropdown set to 'Warning'.

- [Aktivierung von Preview-Features nötig](#)



The screenshot shows the IntelliJ IDEA 'Project Structure' dialog. The 'Project language level' is set to 'SDK default (17 - Sealed types, always-strict floating-point semantics)'. A red arrow points to the 'Project language level' dropdown menu, which is expanded to show '17 (Preview) - Pattern matching for switch' as the selected option.

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

Lernziel: Kennenlernen von Syntax-Neuerungen und verschiedenen API-Erweiterungen in Java 9 bis 11 anhand von Beispielen.

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. Erzeuge 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 – Process-Management

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

Aufgabe 2a

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

Aufgabe 2b

Wie viele Prozesse werden momentan insgesamt ausgeführt?

Aufgabe 2c

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

Aufgabe 2d

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

Aufgabe 3 – 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 4 – 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 4a

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

Aufgabe 4b

Überspringe im Stream `values2` die Werte, solange der Wert kleiner als 10 ist. Gib die Werte auf der Konsole aus.

Aufgabe 4c

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 5 – 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);
```

Tip: Erstelle eine Hilfsmethode mit folgender Signatur:

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

Aufgabe 6 – 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 7 – 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 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

Gruppieren Sie die zuvor gefilterten Namen gemäß dem Anfangsbuchstaben und geben Sie 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]

Hinweis: In diesen Aufgaben geht es vor allem um das API-Kennenlernen, weniger um die Praxis (also die Kombination mit groupingBy()).

Aufgabe 9: Strings

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

Aufgabe 9a

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 9b

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

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

Modifiziere die Ausgabe so, dass die grössten Zahlen zuletzt ausgegeben werden, etwa folgendermaßen:

```
'    55555'
'  7777777'
' 999999999'
```

Tipp: Nutze eine Hilfsmethode

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

Aufgabe 9c

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

```
'0000055555'
'0007777777'
'0999999999'
```

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

Aufgabe 9d

Worin liegt der Unterschied in folgenden zwei Varianten? Finde eine Wertebelegung, die diese besser zeigt, als die hier vorgegebene!

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

PART 2: Multi-Threading mit CompletableFuture<T>

Lernziel: In diesem Abschnitt beschäftigen wir uns mit fortgeschrittenen APIs wie CompletableFuture<T>

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
```

PART 3: Diverses

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

Aufgabe 1 – 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 2 – 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 3 – 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 4 – Die Klasse LocalDate

Lerne Nützliches in der Klasse LocalDate kennen.

Aufgabe 4a

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

Aufgabe 4b

Liste die Sonntage auf, startend mit dem 6. und endend mit dem 12 (inklusive). Das Ergebnis sollte wie folgt sein:

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

Aufgabe 5 – Die Klasse LocalDate

Lerne Nützliches in der Klasse LocalDate kennen.

Aufgabe 5a

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]
```

Gruppier 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 5b

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 5c

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 6 – 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 7 – Die Klasse InputStream

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

- Lesen Sie die Daten aus einem InputStream vollständig ein.
- Lesen Sie nur die ersten 6 Zeichen aus einem InputStream ein.
- Transferieren Sie 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);
}
}
```

Aufgabe 8: 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
}
```

Aufgabe 9: 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 10: 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 11 – 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 11a

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 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);
}
```

Aufgabe 11b

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

Aufgabe 12 – REST-CALL

Stell dir vor, du sollst die Wechselkurse für einen Zeitabschnitt von einigen Monaten berechnen. Im Netz können diese unter <http://data.fixer.io/> (früher frei verfügbar* unter <https://exchangeratesapi.io/>) per REST abgefragt werden. Hierzu kann man das HTTP/2-API gewinnbringend nutzen, wozu wir eine Methode `performGet()` schreiben. Zur Definition des Zeitabschnitts nutzen wir die Neuerungen `datesUntil()`, um einen `Stream<LocalDate>` mit monatlicher Schrittweite zu erzeugen. Mit einem `TemporalAdjuster` springen wir an das Monatsende und führen dann den GET-Aufruf aus.

Für den Zeitraum Januar bis Oktober 2021 erwarten wir folgende Ausgaben:

```
2021-01-31 reported
{"success":true,"timestamp":1612137599,"historical":true,"base":"EUR","date":"2021-01-31","rates":{"CHF":1.080718}}
2021-02-28 reported
{"success":true,"timestamp":1614556799,"historical":true,"base":"EUR","date":"2021-02-28","rates":{"CHF":1.09705}}
2021-03-31 reported
{"success":true,"timestamp":1617235199,"historical":true,"base":"EUR","date":"2021-03-31","rates":{"CHF":1.107039}}
2021-04-30 reported
{"success":true,"timestamp":1619827199,"historical":true,"base":"EUR","date":"2021-04-30","rates":{"CHF":1.09715}}
2021-05-31 reported
{"success":true,"timestamp":1622505599,"historical":true,"base":"EUR","date":"2021-05-31","rates":{"CHF":1.099181}}
2021-06-30 reported
{"success":true,"timestamp":1625097599,"historical":true,"base":"EUR","date":"2021-06-30","rates":{"CHF":1.096862}}
2021-07-31 reported
{"success":true,"timestamp":1627775999,"historical":true,"base":"EUR","date":"2021-07-31","rates":{"CHF":1.074508}}
2021-08-31 reported
{"success":true,"timestamp":1630454399,"historical":true,"base":"EUR","date":"2021-08-31","rates":{"CHF":1.081158}}
2021-09-30 reported
{"success":true,"timestamp":1633046399,"historical":true,"base":"EUR","date":"2021-09-30","rates":{"CHF":1.079352}}
```

*mittlerweile benötigt man einen Schlüssel, für den Workshop kann folgender genutzt werden:

["5e9375c8c908bdc0d6e6a356ea14b860"](#)

Aufgabe 13 – 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.

Bonus

Erstelle ein bash-Skript `exec_hello.sh` zum direkten Ausführen, denke an die korrekten Rechte (`chmod u+x`).

PART 4/5: Neuerungen in Java 12 bis 16

Lernziel: In diesem Abschnitt beschäftigen wir uns mit Erweiterungen und API-Neuerungen in Java 12 bis 16.

Aufgabe 1 – Syntaxänderungen bei switch

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

```
private static void dumbEvenOddChecker(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 dumbEvenOddChecker(int value)`

Aufgabe 1c

Wandle das Ganze so ab, dass du die Spezialform «yield mit Rückgabewert» verwendest.

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 multipleIndentedString =
    "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 und wandle die Klasse `City` zunächst in einen record.

```
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);
    }
}
```

Aufgabe 8 – Text Blocks

Vereinfache folgenden Sourcecode mit einem herkömmlichen String, der über mehrere Zeilen geht und nutze die in neu 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 9 – Text Blocks mit Platzhaltern

Vereinfache folgenden Sourcecode mit einem herkömmlichen String, der über mehrere Zeilen geht und nutze die in neu 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"!
```


Aufgabe 10 – Record-Grundlagen

Gegeben seien zwei einfache Klassen, die reine Datencontainer darstellen und somit lediglich ein öffentliches Attribut bereitstellen. Wandle diese in Records um:

```
class Square
{
    public final double sideLength;

    public Square(final double sideLength)
    {
        this.sideLength = sideLength;
    }
}

class Circle
{
    public final double radius;

    public Circle(final double radius)
    {
        this.radius = radius;
    }
}
```

Welche Vorteile ergeben sich – außer der kürzeren Schreibweise – durch den Einsatz von Records statt eigener Klassen?

Aufgabe 11 – Record

Erstelle auf Basis des nachfolgend gezeigten Records zwei Methoden, die eine JSON- und eine XML-Ausgabe erzeugen. Ergänze eine Gültigkeitsprüfung, sodass Name und Vorname mindestens 3 Zeichen lang sind und der Geburtstag nicht in der Zukunft liegt.

```
record Person(String firstName, String lastName,
              LocalDate birthday) {}
```

```
<Person>
  <firstName>Michael</firstName>
  <lastName>Inden</lastName>
  <birthday>1971-02-07</birthday>
</Person>

{
  "firstName": "Michael",
  "lastName": "Inden",
  "birthday": "1971-02-07"
}
```

Aufgabe 12 – instanceof-Grundlagen

Gegeben seien folgende Zeilen mit einem instanceof sowie einem Cast. Vereinfache das Ganze mit den Neuerungen aus Java 14.

```
Object obj ="BITTE ein BIT";

if (obj instanceof String)
{
    final String str = (String)obj;
    if (str.contains("BITTE"))
    {
        System.out.println("It contains the magic word!");
    }
}
```

Aufgabe 13 – instanceof und record

Vereinfache den Sourcecode mithilfe der Syntaxneuerungen bei instanceof und danach mithilfe der Besonderheiten bei Records.

```
record Square(double sideLength) {
}

record Circle(double radius) {
}

public double computeAreaOld(final Object figure)
{
    if (figure instanceof Square)
    {
        final Square square = (Square) figure;
        return square.sideLength * square.sideLength;
    }
    else if (figure instanceof Circle)
    {
        final Circle circle = (Circle) figure;
        return circle.radius * circle.radius * Math.PI;
    }
    throw new IllegalArgumentException("figure is not a
recognized figure");
}
```

Zwar haben wir durch `instanceof` sicher eine Verbesserung bezüglich Lesbarkeit und Anzahl Zeilen erzielt, jedoch deuten mehrere derartige Prüfungen auf einen Verstoß gegen das Open-Closed-Prinzip, eines der SOLID-Prinzipien guten Entwurfs, hin. Was wäre ein objektorientiertes Design? Die Antwort ist in diesem Fall einfach: Oftmals lassen sich `instanceof`-Prüfungen vermeiden, wenn man einen Basistyp einführt. **Vereinfache das Ganze durch ein Interface `BaseFigure` und nutze dieses passend.**

Bonus

Führe mit Rechtecken einen weiteren Typ von Figuren ein. Das sollte aber keine Modifikationen in der Methode `computeArea()` erfordern.