

# Workshop: Java Update 9 bis 11 Übungen

## Ablauf

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

## Teilnehmer

- Entwickler mit Java-Erfahrung sowie
- SW-Architekten, die Java 9 bis 11 kennenlernen/evaluieren möchten

## Kursleitung und Kontakt

### Michael Inden

Derzeit freiberuflicher Buchautor und Trainer

E-Mail: [michael.inden@hotmail.com](mailto:michael.inden@hotmail.com)

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

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

## 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 – Private Interface-Methoden

Versuche die Motivation für private Methoden in Interfaces zu verstehen. Vereinfache dazu folgenden Programmcode und beginne mit dem Extrahieren einer Hilfsvariable `sum` in der Methode `calc1()`:

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

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

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

## 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);
```

**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 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 – BONUS – Die Klasse `Optional` schon in JDK 8 erweitern

Mit Java 9 wurde 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:

```

public 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)

```

## Aufgabe 9 – 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 10 – Die Klasse Collectors

### Aufgabe 10a

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 10b

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 10c

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 10d

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 10e

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 `groupBy()`).

## Aufgabe 11: Strings

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

### Aufgabe 11a

Nutze folgenden Stream als Eingabe

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

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
```

**Aufgabe 11b**

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 11c**

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 11d**

Modifiziere die Ausgabe so, dass die grössten Zahlen zuletzt ausgegeben werden. Worin liegt der Unterschied in folgenden 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).sorted()
```

**Aufgabe 12: 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 13 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 mit CompletableFuture<T> und Reactive Streams

Lernziel: In diesem Abschnitt beschäftigen wir uns mit fortgeschrittenen APIs wie 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(LocalDate.now() + " onNext(): " + item);
    }

    public void onComplete()
    {
        System.out.println(LocalDate.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 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. Falls dann die Fallback-Berechnung allerdings immer ausgeführt wird, dann ist dies Performance-technisch ungünstig. Schaue dazu auf folgendes Code-Fragment:

```
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 wie folgt:

```

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 };

```

- die erste Abweichung und
- die darauffolgende Abweichung.

### Aufgabe 5 – Die Klasse Arrays

Vergleiche die beiden wie folgt gegebenen Arrays:

```

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

```

- Welches ist «grösser»?
- 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 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 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]
```

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 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.

- Runde das auf Stunden sowie auf Stunden und Minuten.
- Wandle die Zeitdauer in Minuten und was ist der reine Minutenanteil?
- 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:

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

### Aufgabe 11: 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 12: 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 13 – 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 13a

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 13b

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

### Aufgabe 14 – 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 Neuerung `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 2020 erwarten wir folgende Ausgaben:

```
2020-01-31 reported {"rates":{"CHF":1.0694},"base":"EUR","date":"2020-01-31"}
2020-02-29 reported {"rates":{"CHF":1.0614},"base":"EUR","date":"2020-02-28"}
2020-03-31 reported {"rates":{"CHF":1.0585},"base":"EUR","date":"2020-03-31"}
2020-04-30 reported {"rates":{"CHF":1.0558},"base":"EUR","date":"2020-04-30"}
2020-05-31 reported {"rates":{"CHF":1.072},"base":"EUR","date":"2020-05-29"}
2020-06-30 reported {"rates":{"CHF":1.0651},"base":"EUR","date":"2020-06-30"}
2020-07-31 reported {"rates":{"CHF":1.0769},"base":"EUR","date":"2020-07-31"}
2020-08-31 reported {"rates":{"CHF":1.0774},"base":"EUR","date":"2020-08-31"}
2020-09-30 reported {"rates":{"CHF":1.0804},"base":"EUR","date":"2020-09-30"}
```

### Aufgabe 15 – 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`).

### Aufgabe 16 – JShell und REPL

Experimentiere ein wenig mit der JShell. Definiere eine Methode zur Berechnung der Fibonacci-Zahlen. Diese sind rekursiv wie folgt definiert:

- $\text{fib}(0) = 0$
- $\text{fib}(1) = 1$
- $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2), \forall n \geq 2$

Alle die es etwas einfacher mögen, berechnen die Fakultät, die als das Produkt der natürlichen Zahlen von 1 bis  $n$  definiert ist:

- $n! = 1 * 2 * 3 * \dots * n$

Dabei gilt im Speziellen  $0! = 1$ . Damit ist auch hier eine rekursive Definition folgendermaßen möglich:

- $0! = 1$
- $n! = n * (n-1)!, \forall n \geq 1$

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

**Tipp** Starte mit folgenden Zeilen für die Fakultät:

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

## Aufgabe 17 – Multi-Release-JAR

Erstelle ein Multi-Release-JAR für folgende einfache Applikation, die lediglich eine Hilfsklasse Generator aufruft, um Daten zu erzeugen, und diese dann auf der Konsole ausgibt:

```
public class Application
{
    public static void main(final String[] args)
    {
        final Generator gen = new Generator();
        System.out.println("Generated strings: " + gen.createStrings());
    }
}
```

Die als Multi-Release zu verarbeitende Applikation soll in der Hilfsklasse einmal für Java 8 und einmal für Java 9 spezifische Funktionalität nutzen, etwa die Collection-Factory-Methoden aus JDK 9 zur Konstruktion einer Liste, mit einem Hinweis auf die verwendete Java-Version.

### Aufgabe 17a

Implementiere die beiden Generator-Klassen für Java 8 und Java 9. Da ein Multi-Release-JAR erstellt werden soll, benötigen wir auch noch eine Textdatei `MANIFEST.txt`. mit dem Inhalt `Multi-Release: true`. Insgesamt ergibt sich folgende Verzeichnisstruktur:

```
.
├─ MANIFEST.txt
├─ src
│   └─ multireleaseexample
│       ├── Application.java
│       └─ Generator.java
└─ srcjdk9
    └─ multireleaseexample
        └─ Generator.java
```



**Aufgabe 17b**

Momentan gibt es noch keinen Tool-Support und man muss Multi-Release-JARs von Hand kompilieren. Dies erfordert zwei Phasen und vor allem unterschiedliche Versionen vom Bytecode über das Flag `--release`. Wir starten mit folgendem Kommando:

```
javac -d build --release 8 src/multireleaseexample/*.java
```

Was muss für die Sourcen von Java 9 geändert werden, damit das Ergebnis wie folgt aussieht:

```
├─ build
│   └─ META-INF
│       └─ versions
│           └─ 9
│               └─ multireleaseexample
│                   └─ Generator.class
└─ multireleaseexample
    └─ Application.class
    └─ Generator.class
```

**Aufgabe 17c**

Erzeuge aus dem zuvor erzeugten Kompilat im Verzeichnis `build` ein Multi-Release-JAR.

**Aufgabe 17d**

Wenn wir das Programm per

```
java -jar multireleaseexample.jar
```

starten, sollten wir mit Java 9 in etwa folgenden Hinweis sehen:

```
Generated strings: [New, JDK 9, Collection, Factory, Methods]
```

Führen wir das nun mit Java 8 aus, ist in etwa Folgendes erwartet:

```
Generated strings: [Old, Java, 8]
```

**Tipp:** Ermitteln der Java-Installationen:

```
/usr/libexec/java_home -V
```

=>

**Tipp:** Setzen der jeweiligen Java-Version:

```
export JAVA_HOME="/usr/libexec/java_home -v 1.8"
export JAVA_HOME="/usr/libexec/java_home -v 9"
```