

Workshop: Best of Java 9 – 13 Exercises

Procedure

This workshop is divided into several lecture parts, which introduce the subject Java 9 to 13, and gives an overview of the innovations. Following this, some exercises are to be solved by the participants - ideally in group work - on the computer.

Requirements

- 1) Current JDK 11, ideally also JDK 12/13 are installed,
- 2) Current Eclipse is installed (Alternatively: NetBeans or IntelliJ IDEA)

Attendees

- Developers with Java experience, as well as
- SW-Architects, who would like to learn/evaluate Java 9 to 13.

Course management and contact

Michael Inden

CTO & Teamlead SW-Development & Head ASMIQ Academy

ASMIQ AG, Geerenweg 2, 8048 Zürich

E-Mail: michael.inden@asmiq.ch

Course offer: <https://asmiq.ch/>

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

PART 1: Syntax Enhancements / News and API-Changes in Java 9

Objective: Learn about syntax innovations and various API extensions in Java 9 using examples.

Exercise 1 – Process Management

Identify the process ID and other properties of the current process. Use the interface `ProcessHandle` and its methods.

Exercise 1a

How much CPU time has the current process consumed and when was it started?

Exercise 1b

How many processes are currently running?

Exercise 1c

List all Java-processes. Use `Predicate<Info> isJavaProcess` for this task.

Exercise 1d

Try to terminate the current process. What happens then?

Exercise 2 – Collection Factory Methods

Define a list, set, and map using the Collection Factory methods of `()` which are newly introduced in JDK 9. The following program fragment with JDK 8 serves as a starting point.

```
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);
}
```

Exercise 3 – Streams

The Stream API has been enhanced with methods that allow you to read elements only as long as a condition is met or to skip elements while a condition is met. The following two streams are used as starting point:

```
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);
```

Exercise 3a

Determine values from the stream `values1` until an empty string is found. Print out the values to the console.

Exercise 3b

In the stream `values2`, skip the values as long as the value is smaller than 10. Print out the values to the console.

Exercise 3c

What is the difference between the two methods `dropWhile()` and `filter()`?

Tip: The expected result is as follows:

```
takeWhile  
a  
b  
c  
  
dropWhile  
11  
22  
33  
7  
10  
  
with filter  
11  
22  
33  
10
```

Exercise 4 – Streams Take / Drop While

Extract the head and body information with appropriate predicates and the previously presented methods.

```
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>Hello everyone @ JAX London</h1>",
    "    <p>Welcome to this hands-on lab</p>",
    "</body>",
    "</html>");

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

Tip: Create a help method with the following signature:

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

Exercise 5 – The Class Optional

Given the following method, which performs a person search and, depending on the result of the match, calls the method doHappyCase(person) or otherwise doErrorCase().

```
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");
}

```

Use the new methods from class `Optional<T>` to make the program fragment more elegant within a `findJdk9()` method that produces the following outputs like `findJdk8()`:

```

Result: Person: Tim
not found$

```

Exercise 6 – The Class `Optional`

The following program fragment is given, which executes a multi-level search: first in the cache, then in memory, and finally in the database (just simulated here). This search chain is indicated by three `find()` methods and implemented as shown below.

```

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();
}

```

Simplify the call chain using the new methods from the `Optional<T>` class. See how it all becomes clearer.

Exercise 7 BONUS – The Class `Optional` already implemented in JDK 8

With Java 9, useful functionality would be included in the `Optional<T>` class. Create a Utility class `OptionalUtils`, which provides the methods `ifPresentOrElse()`, `or()` and `stream()` for JDK 8. The following signatures serve as support:

```

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)

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

```

Exercise 8 – The Class Collectors

Exercise 8a

Filter all long names (> 5 characters) from the given list of names and provide the result values in the console.

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

Exercise 8b

Group the previously filtered names according to the first letter and provide the result values in the console. The expected result is:

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

Exercise 8c

Combine now the knowledge about a filtering when grouping. Use the following map as a starting point to group only all adults according to the first letter:

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

The expected result is:

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

Exercise 8d

Deepen your knowledge of combining collectors. The above result should be restricted to the name using collector `mapping()`. The expected result is:

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

Exercise 8e

Compress a stream with multiple sets of letters to a set.

```
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"));
```

The expected result is:

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

Exercise 9 BONUS – The Class Collectors already implemented in JDK 8

Java 9 has added useful functionality to the Collectors class. Create a utility class CollectorUtils that provides the filtering() method for JDK 8. The following basic structure serve as support:

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

Tip: Use the following method of() from the class 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) {
```

as well as the following methods:

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

PART 2: Multi-Threading and Reactive Streams

Objective: In this section, we will look at more advanced APIs, such as `CompletableFuture <T>` and Reactive Streams.

Exercise 1 – The class `CompletableFuture<T>`

Refresh your knowledge about the class `CompletableFuture <T>`.

Exercise 1a

Analyze the following program lines, which read a file asynchronously in the `main ()` method. Then two filters are defined, which are then executed with `thenApplyAsync()`, if the file is actually read. With the addition `Async ()` both filter actions happen in parallel. Finally, the results must be brought together again. For this there is the method `thenCombine()`, where a combination function must be passed.

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

    // Possibly longer lasting action
    final CompletableFuture<List<String>> contents = CompletableFuture
        .supplyAsync(extractWordsFromFile(exampleFile));
    contents.thenAccept(text -> System.out.println("Initial: " + text));

    // Run filters in parallel
    final CompletableFuture<List<String>> filtered1 =
        contents.thenApplyAsync(removeIgnorableWords());
    final CompletableFuture<List<String>> filtered2 =
        contents.thenApplyAsync(removeShortWords());

    // Bring the results together
    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;
    };
}
```

Exercise 1b

Imagine that one would run parallel data retrievals that provide a list as a result and would like to combine the results. How does the combination function change then? Rewrite the above code to use two methods: `retrieveData1 ()` and `retrieveData2 ()` and `combineResults()` (analogous to `calcIntersection ()`). Start with the following lines:

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

For two lists with names, the result should be something like this:

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

Exercise 2 – The class `CompletableFuture<T>`

Experiment with the class `CompletableFuture<T>` and the new methods introduced in JDK 9: `failedFuture()`, `orTimeout()` and `completeOnTimeout()`. Use your knowledge about `exceptionally()` to handle exceptions during processing. Start with the following basic structure and complete the error and 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);
    }

```

Output is expected to be analogous to the following:

```

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

```

Exercise 3 – Reactive Streams

Consider a program Exercise3_ReactiveStreamsExample with a Publisher<String> publishing names from a list to registered Subscribers<String> in the doWork () method:

```

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
    }
}

```

Outputs should be as follows:

```

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

```

The Publisher<String> is implemented as follows:

```
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 }
        }
    }
}
```

The following class ConsoleOutSubscriber lists all occurrences on the console:

```
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();
    }
}
```

Based on the above class `ConsoleOutSubscriber`, implement a separate `Subscriber<String>` called `SkipAndTakeSubscriber`, which skips the first `n` occurrences and then outputs `m` occurrences. After that, the communication should be stopped.

Expenses are expected to be approximately as follows:

```
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: Misc

Objective: In this section we want to learn about some practical API extensions: Arrays as well as LocalDate and InputStream.

Exercise 1 – The class Arrays

Determine if the text described by search appears in the original text. To do this, first convert the strings to the type byte[] and find the position of match of search:

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

Exercise 2 – The class Arrays

Determine for the two arrays given below

```
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) the first deviation and
- b) the subsequent deviation.

Exercise 3 – The class Arrays

Compare the two arrays given below:

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

- a) Which is «bigger»?
- b) From which position first is greater than second, if you always remove one letter from second in every further comparison. For a better understanding, log the compared values for all start positions.

Exercise 4 – The Class `LocalDate`

Get to know useful things in the `LocalDate` class.

Exercise 4a

Write a program that counts all Sundays in 2017.

Exercise 4b

Write a program that determines all Fridays 13th in the years from 2013 to 2017:

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

The following values should appear as the result:

```
[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]
```

Exercise 4 Bonus

Make an overview of the number of each day in a week in 2012, as follows:

```
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
```

Tip: Use a helper method with the following signature:

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

Exercise 5 – The class `LocalDate`

Learn useful things in the `LocalDate` class.

Exercise 5a

Write a program that determines all Fridays 13th in the years 2013 to 2017. Use the following lines as starting point:

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

As a result, the following values should appear:

```
[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]
```

Group the occurrences by year. The following issues should appear:

```
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]
```

Exercise 5b

How many times have there been February 29 between the beginning of 2010 and the end of 2017?

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

Exercise 5c

How often was your birthday a Sunday between the beginning of 2010 and the end of 2017? For the 7th of February the following result should be calculated:

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

Exercise 6 – The Class InputStream

Write a program that creates a copy of a file given by a filename. Simplified Java 8-based source code below:

```
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);
        }
    }
}
```


Exercise 7 – The Class InputStream

Simplify the following Java 8-based source code and write methods that do the following:

- Read in the data from an InputStream completely.
- Only read the first 6 characters from an InputStream.
- Transfer all data from an InputStream to an 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);
    }
}
```

Tip: Use a ByteArrayInputStream as input.

PART 4: Innovations in Java 10

Objective: In this section, we'll look at syntax extensions and API improvements in Java 10.

Exercise 1 – Getting to know var

Get to know the new reserved word var.

Exercise 1a

Start the JShell or an IDE of your choice. Create a method `funWithVar()`. Define the variables `name` and `age` with the values `Mike` and `47`.

Exercise 1b

Expand your know-how regarding `var` and generics. Use it for the following definition. Initially create a local variable `personsAndAges` and then simplify with `var`:

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

Exercise 1c

Simplify the following definition with `var`. What should be considered? Where's the difference?

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

Exercise 1d

Why do the following lambdas cause problems? How do you solve these?

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

Why then compile the following?

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

Exercise 2 – Unmodifiable Collections

Java 9 introduced Unmodifiable Collections and Collection Factory Methods. Java 10 now offers the ability to create fixed copies and special collectors for streams.

Exercise 2a

Which two variants does Java 10 offer to create a fixed copy of the following list?

```
List<String> words = List.of("Hello", "World");

List<String> copy1 = null; /* TODO */
List<String> copy2 = null; /* TODO */
```

What type do the resulting copies have? Print this on the console.

Exercise 2b

What is the difference in the resulting unmodifiable lists when you make a modification to the array?

```
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";
```

What happens when changing in the list analogous to the following?

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

Exercise 2c

What differentiates the three variants of copies or wrappers?

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

// 3 variants to create copies of unmodifiable lists
List<String> copy1 = new ArrayList<>(names);
List<String> copy2 = List.copyOf(names);
List<String> wrapper = Collections.unmodifiableList(names);
```

Can elements be changed? Can elements be deleted? Fill in the following table.

	Deletion	Modification
copy1		
copy2		
wrapper		

Tip: Use among other things the following helper method:

```
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");
    }
}
```

Exercise 3: Extensions in Optional<T>

Given a simple but uncertain value extraction from an Optional<T>. Write these using the new Java 10 API to expose potentially unsafe access.

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

Exercise 4: Enhancement in the class Reader

Transfer the content from a StringReader to a file hello.txt. Read this again and give out the content. Complete the following lines:

```
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: Extensions in Java 11

Objective: In this section, we will look at extensions and API improvements in Java 11.

Exercise 1 – Direct compilation and execution

Write a HelloWorld class in the package `java11.direct.compilation` and save it in a Java file of the same name (or use `cat > HelloWorld.java` *Ctrl-C). Then execute these directly with the command `java`.

```
package java11.direct.compilation;

public class Exercise1_HelloWorld
{
    public static void main(String... args)
    {
        System.out.println("Hello Execute After Compile");

        var procesInfo = ProcessHandle.current().info();
        System.out.println(procesInfo);
    }
}
```

What is special about that?

Exercise 2: Extensions in `Optional<T>`

Simplify the following construct with the new method `isEmpty()` from `Optional<T>`:

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

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

Exercise 3: Strings

The processing of strings has been made easier in Java 11 with some useful methods.

Exercise 3a

Use the following stream as input:

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

Implement a method that outputs the numbers one below the other, repeated as often as the digit as follows:

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

Exercise 3b

Modify the output so that the numbers are right aligned with a maximum of 10 characters:

```
'      4444'
'   7777777'
' 999999999'
```

Tip: Use a helper method

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

Exercise 3c

Modify the whole thing so that instead of spaces leading zeros are output, as follows:

```
'0000004444'
'0007777777'
'0999999999'
```

Bonus: Expand the whole thing so that any fill characters can be used.

Exercise 3d

Modify the output so that the largest numbers are output last. Find at least two variants:

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

Where is the difference?

Exercise 4: Predicates

Simplify the following Predicates in terms of negation:

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

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

Exercise 5: Strings und Files

Until Java 11 it was a bit difficult to write texts directly into a file or to read them from it. Now you can use the methods `writeString()` and `readString()` from the class `Files`. Use them to write the following lines to a file. Read this again and prepare a `List<String>` from it.

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

Exercise 6 – HTTP/2

The following HTTP communication is given, which accesses the Oracle Web page and formats it textually.

```
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();
    }
}
```

Exercise 6a

Convert the source code to use the new HTTP/2 API from JDK 11. Use the classes `HttpRequest` and `HttpResponse` and create a method `printResponseInfo(HttpResponse)`, which reads the body analogous to the method `readContent(InputStream)` above and also provides the HTTP status code. Start with the following program fragment:

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

    printResponseInfo(response);
}

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());
}
```

Exercise 6b

Start the queries asynchronously by calling `sendAsync()` and process the received `CompletableFuture<HttpResponse>`.

PART 6/7: News in Java 12 / 13

Objective: In this section we will look at extensions and API improvements in Java 12 & 13.

Exercise 1 – Syntax changes for switch

Simplify the following source code with a conventional switch-case with the new syntax of Java 12 / 13.

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

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

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

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

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

Exercise 1a

First use the Arrow syntax to write the method shorter and clearer.

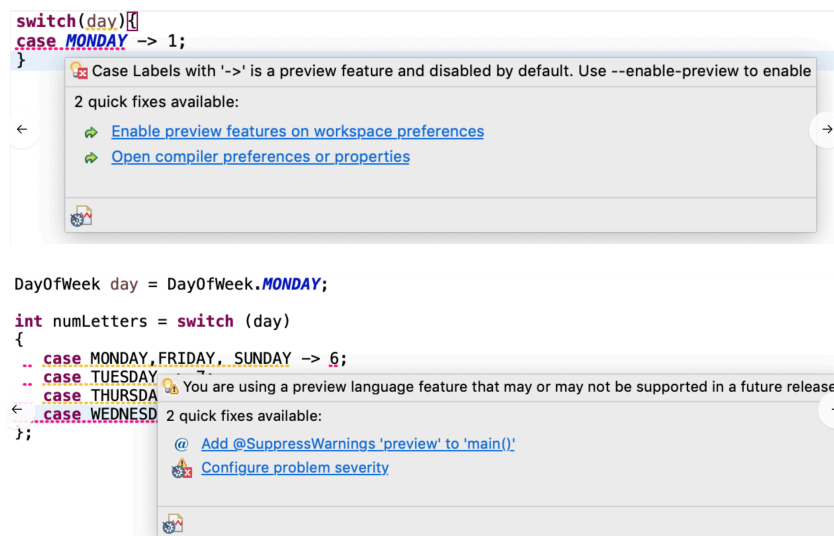
Exercise 1b

Now use the possibility to specify returns directly and change the signature in `String dumpEvenOddChecker(int value)`.

Exercise 1c

Convert the above code so that it uses the special form "break with return value" (Java 12) or "yield with return value" (Java 13).

Tips: Activation of the preview feature and suppression of warnings:



Exercise 2: The class CompactNumberFormat

Write a program to output and parse 1,000, 1,000,000 and 1,000,000,000 depending on locale and style. Use the Locale GERMANY for SHORT and ITALY for LONG.

Use the following values for parsing:

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

Exercise 3: Strings

The processing of strings has been extended by two methods in Java 12. First, get to know `indent()` better.

Exercise 3a

Enter the following input by 7 characters, output this and remove 3 more characters from the indentation.

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

Exercise 3b

What happens if you put a left-aligned text with negative values for the indent? What happens if you use an -10 index for subsequent input?

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

Exercise 4: Strings

The processing of strings has been extended by two methods in Java 12. Learn more about `transform()`. Given the following comma-separated input:

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

Exercise 4a

Convert these completely to lowercase and replace the commas with spaces.

Exercise 4b

Use other transformations and replace HELLO with the Swiss greeting "GRÜEZI", then split the whole thing into individual components, resulting in the following list as result:

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

Exercise 5: Teeing-Collector

Use the Teeing Collector to find both minimum and maximum in one pass. Start with the following lines:

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

Exercise 6: Teeing- Collector

Vary the BiFunction to properly influence the results of the Teeing Collector. Start with the following lines and complete them at the marked places:

```
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));
```

The expected results are

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

Exercise 7: Teeing-Collector

Use a Teeing Collector to find all cities in Europe by name, as well as the number of cities in Asia. Start with the following lines:

```
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(...
```

Given the class City is as follows:

```
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);
    }
}
```

Exercise 8 – Text Blocks

Simplify the following source code with a conventional string that spans multiple lines and use the syntax introduced in Java 13.

```
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"
    + "}";
```

Exercise 9 – Text Blocks with Placeholders

Simplify the following source code with a conventional string that spans multiple lines and use the syntax introduced in Java 13:

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

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

Produce the following output with the new syntax:

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