Workshop:
Best of Java 11 – 21
Exercises

Procedure

This workshop is divided into several lecture parts, which introduce the subject Java 9 to 17, and gives an overview of the innovations. Following this, some exercises are to be solved by the participants - ideally in pair / group programming - on the computer.

Requirements

1) Current JDK 21 is installed
2) Current Eclipse 2023-09 with Java 21 Plugin or IntelliJ IDE2023.2.2 is installed

Attendees

- Developers with Java experience, as well as
- SW-Architects, who would like to learn/evaluate Java 11 to 21.

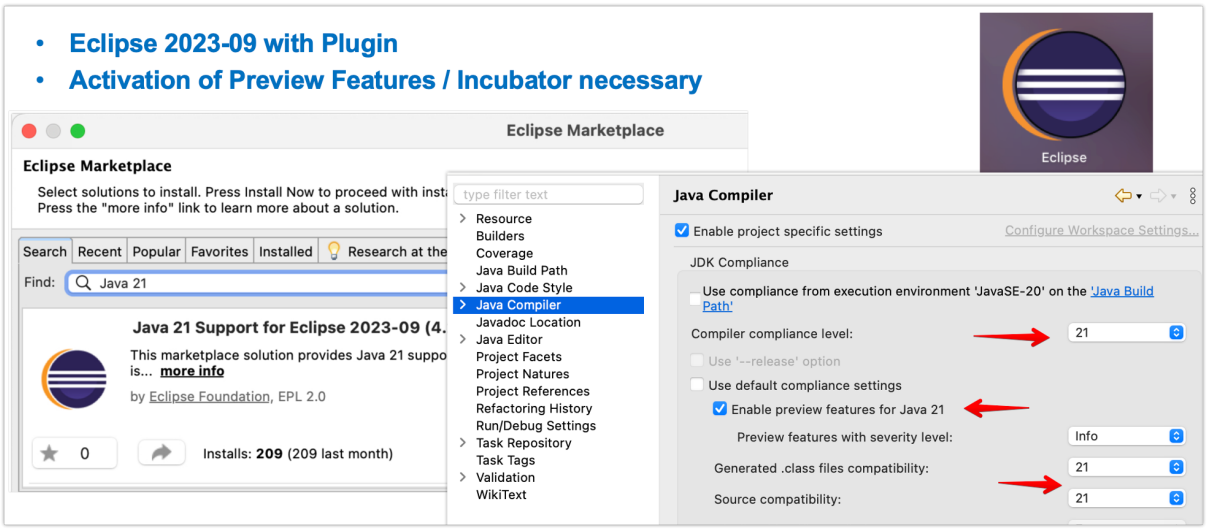
Course management and contact

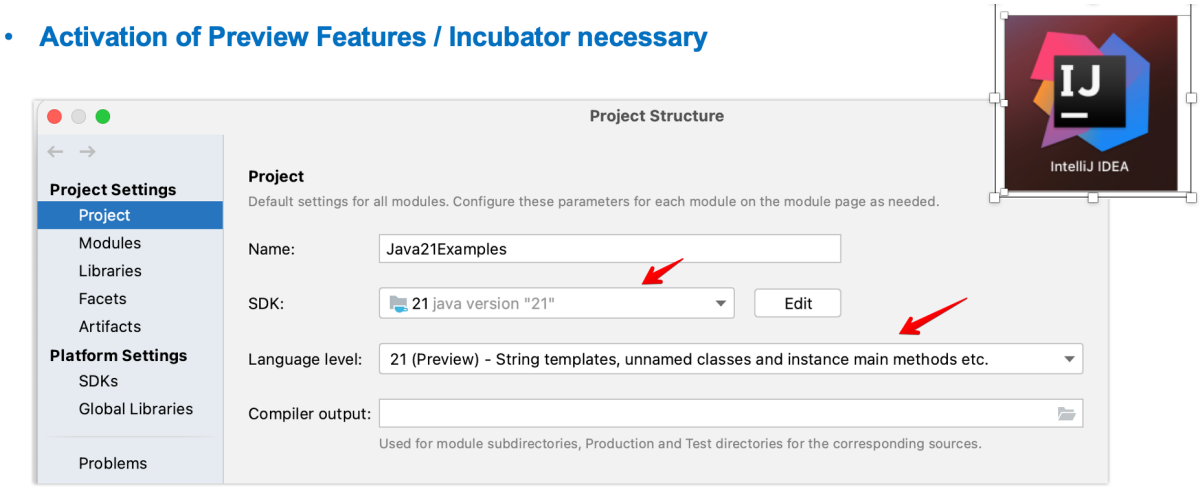
Michael Inden
Head of Development, independent SW Consultant, Author, and Trainer
E-Mail: michael_inden@hotmail.com

Please do not hesitate to ask: Further courses (Java, Unit Testing, Design Patterns) are available on request as in-house training.

Configuration Eclipse / IntelliJ

Please note that we have to configure some small things before the exercises if you are not using the latest IDEs.





PART 1: Syntax Enhancements / News / API Changes up to Java 11

Objective: Learn about syntax innovations and various API extensions in Java up to version 11 using examples.

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;
```

Then why does the following compile?

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

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.

Use a static import as follows: `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);
}
```

Exercise 3 – 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 4 – 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 5 – Strings

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

Exercise 5 a

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 specifies as follows:

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

Exercise 5 b
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 5 c
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 5 d
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?

PART 2: Misc in Java 11

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

Exercise 1 – The Class `LocalDate`

Get to know useful things in the LocalDate class.

Exercise 1 a

Write a program that counts all Sundays in 2017.

Exercise 1 b

List the Sundays, starting with the 5th and ending with the 10th. The result should be as follows:

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

Exercise 2 – The class `LocalDate`

Learn useful things in the LocalDate class.

Exercise 2 a

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 values 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 2 b

How many times did February 29 occur 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 2 c

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 3 – 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 4 – HTTP/2

The following HTTP communication is given, which accesses the Oracle web page and prints 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 4 a

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 4 b

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

PART 3: Syntax Enhancements in Java 12 to 17

Objective: In this section we will look at syntax extensions in Java 12 to 17.

Exercise 1 – Syntax changes for switch

Simplify the following source code containing a conventional switch-case with the new syntax.

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

      switch (value)
      {
      case 1:
      case 3:
      case 5:
      case 7:
      case 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 1 a

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

Exercise 1 b

Now use the option to specify returns directly and change the signature to `String dumbEvenOddChecker(int value).`

Exercise 1 c

Convert the above code so that it uses the special form "yield with return value".

Exercise 2 – Text Blocks

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

```
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";
```

Exercise 3 – Text Blocks with Placeholders

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

```
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"!
```

Exercise 4 – Record Basics

Two simple classes are given, which represent pure data containers and thus only provide a public attribute. Convert them into records:

```
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;
     }
}
```

What are the advantages – apart from the shorter spelling – of using records instead of separate classes?

Exercise 5 – Record

Based on the record shown below, create two methods that produce JSON and XML output. Add a validation check so that the last name and first name are at least 3 characters long and the birthday is not in the future.

```
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"
}
```

Exercise 6 – `instanceof` Basics

Given are the following lines with an `instanceof` and a cast. Simplify the whole thing with the new features of modern Java.

```
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!");
    }
}
```

Exercise 7 – `instanceof` and `record`

Simplify the source code using the syntax innovations in `instanceof` and then using the special features in 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");
}
```

Although we have certainly achieved an improvement in terms of readability and number of lines by using `instanceof`, several of these checks indicate a violation of the open-closed principle, one of the SOLID principles of good design. What would an object-oriented design be? The answer in this case is simple: Often `instanceof` checks can be avoided by introducing a base type. **Simplify the whole thing with an interface `BaseFigure` and use it appropriately.**

Bonus
Introduce with rectangles another type of figures. However, this should not require any modifications in the `computeArea()` method.

PART 4: API-News in Java 12 to 17

Objective: In this section, we look at extensions and API innovations in Java 12 through 17.

Exercise 1 – 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 2 – Strings

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

Exercise 2 a

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 2 b

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 3 – 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 3 a

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

Exercise 3 b

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

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

Exercise 4 – 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(…
```

BONUS: Experiment with comparing on length instead of alphabetical sorting.

Exercise 5 – 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 6 – 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 and convert the City class into a 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(…
```

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);
      }
}
```

PART 5: JVM enhancements in Java 12 to 17

Objective: In this section we look at JVM enhancements in Java 12 to 17.

Exercise 1 – JMH

Create a JMH project using the Maven command from the slides. Expand this project and create a simple benchmark (for inspiration: open the existing project from the JMH-Benchmarking.zip and copy one of the benchmark classes). Build the project and run the benchmark(s).

Exercise 2 – JPackage

Have a look at the PackackingDemo project and modify it to use another dependency, for example on Apache Commons.

```
jpackage --input target/ --name JPackageDemoApp
         --main-jar PackagingDemoExamples-1.0.0-SNAPSHOT.jar
         --main-class de.java17.ApplicationExample
         --type <dmg / msi / …>
```

Supplement as needed:

```
--java-options '--enable-preview'
```

Exercise 3 – JShell-API

Use the JShell API to perform some dynamic calculations

- `int result = x * y;`
- `var today = LocalDate.now();`
- `var values = List.of(1,  2,  3,  4);`

List all variables and their values. Use the `variables()` method for this. Output the list of values to the console.

Tip: Think of the appropriate imports!

PART 6/7: Enhancements in Java 18 to 21

Objective: In this section, we look at enhancements in Java 18 to 21.

Aufgabe 1 – Convert to Record Patterns

Given is a definition of a journey through the following records:

```
record Person(String firstname, String lastname, LocalDate birthday) {
}

record TravelInfo(LocalDate start, Duration maxTravellingTime) {
}

record City(Integer zipCode, String name) {
}

record Journey(Person person,
               TravelInfo travelInfo,
               City from,
               City to) {
}
```

In addition, various consistency checks and validations are performed that access nested components. For this purpose, one sometimes sees implementations - especially in legacy code - that contain deeply nested `if`s and various `null` checks.

The task now is to implement the whole thing in a more understandable and compact way using record patterns.

Bonus: Simplify the specification with `var`.

Exercise 2 – Use Record Patterns for recursive calls

Given are definitions of some graphical figures by the following records:

```
sealed interface Figure {
}

record Point(int x, int y) implements Figure {
}

record Line(Point start,
            Point end) implements Figure {
}
```

```
record Triangle(Point pointA,
                Point pointB,
                Point PointC) implements Figure {
}
```

In addition, the following method is defined, which is to be completed. The task is to add the points for the two figures `Line` and `Triangle` using recursive calls:

```
static int process(Figure figure) {
    return switch (figure) {
        case Point(int x, int y) -> x * y;
        // TODO
        default -> throw new IllegalStateException("Unexpected value: " +
                                                   figure);
    };
}
```

---------- Java 21 ----------

Exercise 3 – Convert to Virtual Threads

Given an execution of various tasks using a classic `ExecutorService` and a given pool size of 50:

```
try (var executor = Executors.newFixedThreadPool(50)) {
    IntStream.range(0, 1_000).forEach(i -> {
        executor.submit(() -> {
            Thread.sleep(Duration.ofSeconds(1));

            System.out.println("Task " + i + " finished!");
            return i;
        });
    });
}
```

Convert the whole thing to use virtual threads. Use a suitable method in `Thread`.

Exercise 4 – Convert with Structured Concurrency

Given an execution of various tasks using a classic `ExecutorService` and a "combining" of the calculation results to a console output:

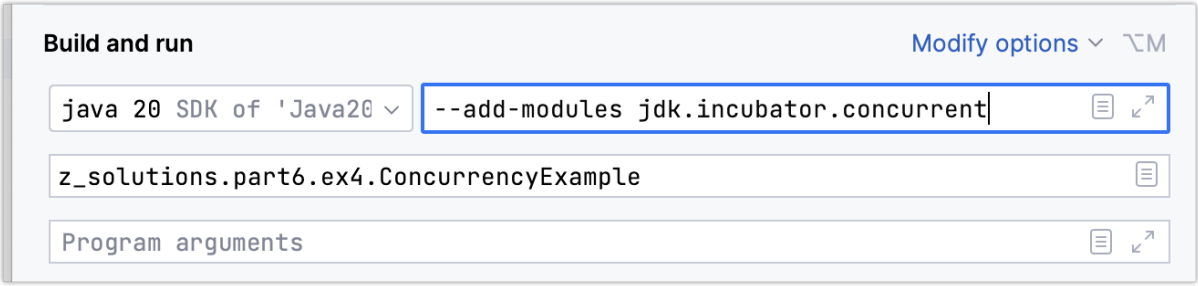
```
static void executeTasks(boolean forceFailure) throws InterruptedException,
                                                      ExecutionException {
    try (var executor = Executors.newFixedThreadPool(50)) {
        Future<String> task1 = executor.submit(() -> {
            return "1";
        });
        Future<String> task2 = executor.submit(() -> {
            if (forceFailure)
                throw new IllegalStateException("FORCED BUG");

            return "2";
        });
        Future<String> task3 = executor.submit(() -> {
            return "3";
        });

        System.out.println(task1.get());
        System.out.println(task2.get());
        System.out.println(task3.get());
    }
}
```

Use Structured Concurrency to replace the `ExecutorService` and use the `ShutdownOnFailure` strategy to clarify processing in case of failure. Analyze the processing in case of failure.

Tipp for Java 20 Users Only: It is an incubator feature in Java 20, so you have to make appropriate configurations when compiling and starting. In Java 21 it is already a preview feature.



Exercise 5 – Experiment with Sequenced Collections

Given the following class with some TODO comments, the first prime numbers are to be prepared as a list. In addition, elements are to be inserted at the front and at the back and a reverse sequence is to be generated.

```
public class Ex05_SequencedCollections
{
    public static void main(String[] args)
    {
        List<Integer> primeNumbers = new ArrayList<>();
        primeNumbers.add(3); // [3]
        // TODO: add 2
        primeNumbers.addAll(List.of(5, 7, 11));
        // TODO: add 13

        System.out.println(primeNumbers); // [2, 3, 5, 7, 11, 13]
        // TODO print first and last element
        // TODO print reverser order

        // TODO: add 17 as last
        System.out.println(primeNumbers); // [2, 3, 5, 7, 11, 13, 17]
        // TODO print reverser order
    }
}
```

Exercise 6 – Experiment with Template Processor

Write your own template processor which encloses the values with [[and]] or alternatively with a ' in front and behind:

```
System.out.println(DOUBLE_BRACES."Hello, \{name}! Next year, you'll be
\{age + 1}.");

=>

Hello, [[Michael]]! Next year, you'll be [[53]].
```

a) Use `fragments()` and `values()` and a loop.
b) Simplify the whole thing with `interpolate()` and the Stream API.
c) Use a parameterizable lambda to make the start and end sequence freely configurable.
d) Create a template processor that works similar to the f-strings in Python (`f"Calculation: {x} + {y} = {x + y}"`) without directly referencing STR.

Exercise 7 – Experiment with Unnamed Patterns and Variables

Simplify the following method by using Unnamed Patterns and Variables to increase readability and understandability – take advantage of the fact that the IDEs show unused variables:

```
static boolean checkFirstNameAndCountryCodeAgainImproved(Object obj) {
    if (obj instanceof Journey(
          Person(var firstname, var lastname, var birthday),
          TravelInfo(var start, var maxTravellingTime), var from,
          City(var zipCode, var name))) {

       if (firstname != null && maxTravellingTime != null
                             && zipCode != null) {

          return firstname.length() > 2
                  && maxTravellingTime.toHours() < 7
                  && zipCode >= 8000 && zipCode < 8100;
       }
    }
    return false;
}
```