Workshop: Best of Java 11 bis 21 Übungen

Ablauf

Dieser Workshop gliedert sich in mehrere Vortragsteile, die den Teilnehmern die Thematik Java 11 bis 21 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 21 LTS (21.0.1 oder neuer)
2) Aktuelles Eclipse oder IntelliJ IDEA installiert

Teilnehmer

- Entwickler mit Java-Erfahrung sowie
- SW-Architekten, die Java 11 bis 21 kennenlernen/evaluieren möchten

Kursleitung und Kontakt

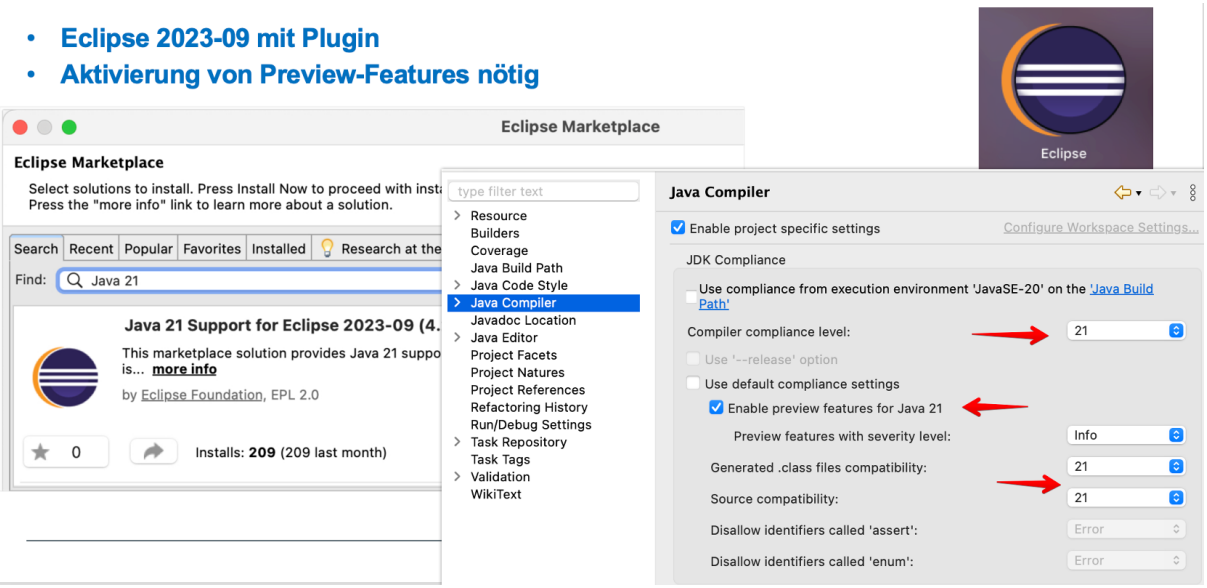
Michael Inden

Head of Development, freiberuflicher Buchautor, Trainer und Konferenz-Speaker
E-Mail: michael_inden@hotmail.com

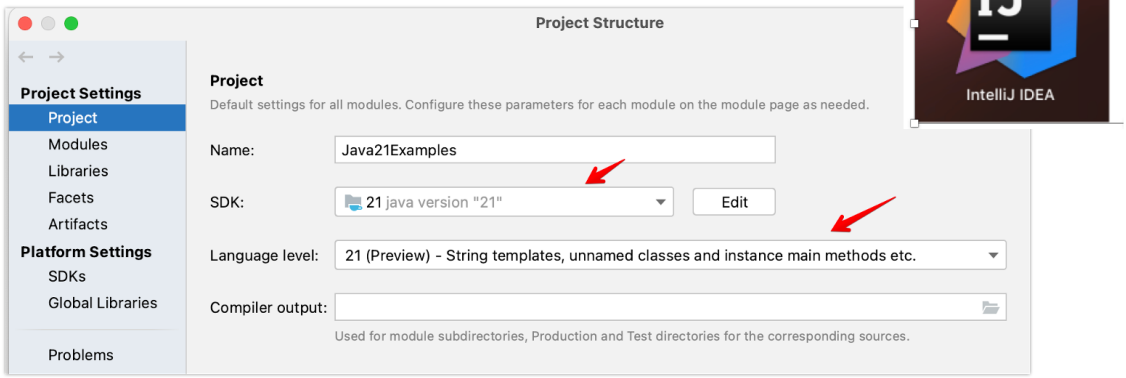
Konfiguration Eclipse / IntelliJ für Java 21

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

- **Eclipse 2023-09 mit Plugin**
- **Aktivierung von Preview-Features nötig**



- **Aktivierung von Preview-Features nötig**



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

Lernziel: Kennenlernen von Syntax-Neuerungen und verschiedenen API-Erweiterungen bis Java 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 – Collection-Factory-Methoden

Definiere eine Liste, eine Menge und eine Map mithilfe der in JDK 9 neu eingeführten Collection-Factory-Methoden namens `of()`. Als Ausgangsbasis dient nachfolgendes Programmfragment mit JDK 8. Nutze einen statischen Import wie folgt: `import static java.util.Map.entry;`

```
private static void collectionsExampleJdk8()
{
	final List<String> names = Arrays.asList("Tim", "Tom", "Mike");
	System.out.println(names);

	final Set<Integer> numbers = new TreeSet<>();
	numbers.add(1);
	numbers.add(3);
	numbers.add(4);
	numbers.add(2);
	System.out.println(numbers);

	final Map<Integer, String> mapping = new HashMap<>();
	mapping.put(5, "five");
	mapping.put(6, "six");
	mapping.put(7, "seven");
	System.out.println(mapping);
}
```

Aufgabe 3 – 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 4 – 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 5 – Strings

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

Aufgabe 5 a

Nutze folgenden Stream als Eingabe

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

Realisiere eine Ausgabe, die die sieben Zahlen untereinander ausgibt, jeweils so oft wiederholt, wie die Ziffer, also verkürzt wie folgt:

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

Aufgabe 5 b
Modifiziere die Ausgabe so, dass die Zahlen rechtsbündig mit maximal 10 Zeichen ausgegeben werden:

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

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

```
‘     55555’
‘   7777777’
‘ 999999999’
```

Tipp: Nutze eine Hilfsmethode

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

Aufgabe 5 c
Modifiziere das Ganze so, dass nun statt Leerzeichen führende Nullen ausgegeben werden, etwa wie folgt:

```
‘0000055555’
‘0007777777’
‘0999999999’
```

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

Aufgabe 5 d
Worin liegt der Unterschied zwischen folgenden zwei Varianten? Finde eine Wertebelegung, die diesen besser zeigt, als die hier vorgegebene!

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

PART 2: Java 11 Diverses

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

Aufgabe 1 – Die Klasse `LocalDate`

Lerne Nützliches in der Klasse `LocalDate` kennen.

Aufgabe 1 a

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

Aufgabe 1 b

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 2 – Die Klasse `LocalDate`

Lerne Nützliches in der Klasse `LocalDate` kennen.

Aufgabe 2 a

Schreibe ein Programm, dass alle Freitage der 13. in den Jahren 2013 bis 2017 ermittelt. Nutze folgende Zeilen als Ausgangspunkt:

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

Als Ergebnis sollten folgende Werte erscheinen:

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

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

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

Aufgabe 2 b

Wie viele Male 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 2 c
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 3 – 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 Zeilen in eine Datei.

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

Lies diese wieder ein und bereite daraus eine `List<String>` auf.

Aufgabe 4 – 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");
    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 4 a

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");
    final HttpClient httpClient = // TODO
    final HttpRequest request = // TODO
    final BodyHandler<String> asString = // TODO
    final HttpResponse<String> response = // TODO

    printResponseInfo(response);
}
```

Aufgabe 4 b

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

PART 3: Syntax-Erweiterungen in Java 12 bis 17

Lernziel: In diesem Abschnitt beschäftigen wir uns mit Syntax-Erweiterungen in Java 12 bis 17.

Aufgabe 1 – Syntaxänderungen bei switch

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

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

      switch (value)
      {
      case 1:
      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);
}
```

Aufgabe 1 a

Nutze zunächst nur die Arrow-Syntax, um die Methode kürzer und übersichtlicher zu schreiben.

Aufgabe 1 b

Verwende nun noch die Möglichkeit, Rückgaben direkt zu spezifizieren und ändere die Signatur in `String dumbEvenOddChecker(int value)`

Aufgabe 1 c

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

Aufgabe 2 – Text Blocks

Vereinfache folgenden Sourcecode mit einem herkömmlichen String, der über mehrere Zeilen geht und nutze die neu eingeführte Syntax.

```
String multiLineStringOld = "THIS IS\n" +
            "A MULTI\n" +
            "LINE STRING\n" +
            "WITH A BACKSLASH \\\n";

String multiLineHtmlOld = "<html>\n" +
           "    <body>\n" +
           "        <p>Hello, world</p>\n" +
           "    </body>\n" +
           "</html>";

String javaScriptObjOld = ""
               + "{\n"
               + "    \"version\": \"Java13\",\n"
               + "    \"feature\": \"text blocks\",\n"
               + "    \"attention\": \"preview!\"\n"
               + "}\n";
```

Aufgabe 3 – Text Blocks mit Platzhaltern

Vereinfache folgenden Sourcecode mit einem herkömmlichen String, der über mehrere Zeilen geht und nutze die neu eingeführte Syntax:

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

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

Produziere folgende Ausgaben mit der neuen Syntax:

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

Aufgabe 4 – Record-Grundlagen

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

```
class Square
{
     public final double sideLength;

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

class Circle
{
     public final double radius;

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

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

Aufgabe 5 – Record

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

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

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

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

Aufgabe 6 – `instanceof`-Grundlagen

Gegeben seien folgende Zeilen mit einem `instanceof` sowie einem Cast. Vereinfache das Ganze mit den Neuerungen aus modernem 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!");
    }
}
```

Aufgabe 7 – `instanceof` und `record`

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

```
record Square(double sideLength) {
}

record Circle(double radius) {
}

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

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

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

PART 4: API-Neuerungen in Java 12 bis 17

Lernziel: In diesem Abschnitt beschäftigen wir uns mit Erweiterungen und API-Neuerungen in Java 12 bis 17.

Aufgabe 1 – Die Klasse `CompactNumberFormat`

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

Nutze die nachfolgenden Werte zum Parsing:

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

Aufgabe 2 – Strings

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

Aufgabe 2 a

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

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

Aufgabe 2 b

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

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

Aufgabe 3 – Strings

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

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

Aufgabe 3 a

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

Aufgabe 3 b

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

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

Aufgabe 4 – Teeing-Kollektor

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

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

BONUS: Experimentiere mit einem Vergleich auf Länge statt der alphabetischen Sortierung.

Aufgabe 5 – Teeing-Kollektor

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

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

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

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

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

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

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

Die erwarteten Resultate sind mit

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

Aufgabe 6 – Teeing-Kollektor

Nutze einen Teeing-Kollektor, um in einem Durchlauf sowohl alle europäischen Städte namentlich als auch die Anzahl der Städte in Asien zu ermitteln. Beginne mit folgenden Zeilen und wandle die Klasse `City` zunächst in einen `record`.

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

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

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

Gegeben sei noch die Klasse `City` wie folgt:

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

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

      public String getName()
      {
            return name;
      }

      public String getRegion()
      {
            return region;
      }

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

BONUS: Diese Klasse soll durch einen Record vereinfacht werden.

PART 5: JVM-Neuerungen in Java 12 bis 17

Lernziel: In diesem Abschnitt beschäftigen wir uns mit JVM-Erweiterungen in Java 12 bis 17.

Aufgabe 1 – JMH

Erzeuge mit dem Maven-Kommando aus den Folien ein JMH-Projekt. Erweitere dieses Projekt und erstelle einen einfachen Benchmark (als Ideengeber: öffne das bestehende Projekt aus dem `JMH-Benchmarking.zip` und kopiere eine der Benchmark-Klassen). Baue das Projekt und führe den/die Benchmarks aus.

Aufgabe 2 – JPackage

Experimentiere mit dem PackackingDemo-Projekt und wandle dieses so ab, dass noch eine weitere Abhängigkeit verwendet wird, etwa auf Apache Commons.

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

Bei Bedarf ergänzen:

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

Aufgabe 3 – JShell-API

Führe mit dem JShell-API ein paar dynamische Berechnungen aus

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

Liste alle Variablen und deren Wert auf. Verwende dafür die Methode `variables()`. Gib die Liste der Werte auf der Konsole aus.

Tipp: Denke an die passenden Imports!

PART 6: Neuerungen in Java 18 bis 21

Lernziel: In diesem Abschnitt beschäftigen wir uns mit Erweiterungen in Java 18 bis 21.

Aufgabe 1 – Wandle in Record Pattern um

Gegeben ist eine Definition einer Reise durch folgende Records:

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

record TravelInfo(LocalDate start, Duration maxTravellingTime) {
}

record City(String zipCode, String name) {
}

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

Zudem werden verschiedene Konsistenz-Checks und Prüfungen ausgeführt, die auf verschachtelte Bestandteile zugreifen. Dazu sieht man mitunter – vor allem in Legacy-Code – Implementierungen, die tief verschachtelte `if`s und diverse `null`-Prüfungen enthält.

Die Aufgabe besteht nun darin, das Ganze mithilfe von Record Patterns verständlicher und kompakter zu realisieren.

Bonus: Vereinfache die Angabe mit `var`.

Aufgabe 2 – Nutze Record Patterns für rekursive Aufrufe

Gegeben sind Definitionen einiger Figuren durch folgende 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 {
}
```

Zudem ist die folgende Methode definiert, die die x- und y-Koordinate eines Punkt multipliziert. Diese soll nun ergänzt werden, sodass für die beiden Figuren `Line` und `Triangle` die jeweiligen Punkte addiert werden:

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

Aufgabe 3 – Wandle in virtuelle Threads um

Gegeben ist eine Ausführung verschiedener Tasks mithilfe eines klassischen `ExecutorService` und einer vorgegebenen Pool-Size von 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;
        });
    });
}
```

Wandle das Ganze so um, dass virtuelle Threads genutzt werden, und prüfe dies nach. Nutze dazu eine passende Methode in `Thread`.

Aufgabe 4 – Experimentiere mit Sequenced Collections

Gegeben sei folgende Klasse mit einigen TODO-Kommentaren, die die ersten Primzahlen als Liste aufbereiten soll. Zudem sollen vorne und hinten Elemente eingefügt sowie eine umgekehrte Reihenfolge aufbereitet werden.

```
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
}
```

PART 6: Neuerungen in Java 18 bis 21 Preview + Incubator

Aufgabe 5 – Wandle mit Structured Concurrency um

Gegeben ist eine Ausführung verschiedener Tasks mithilfe eines klassischen `ExecutorService` und einer Zusammenführung der Berechnungsergebnisse:

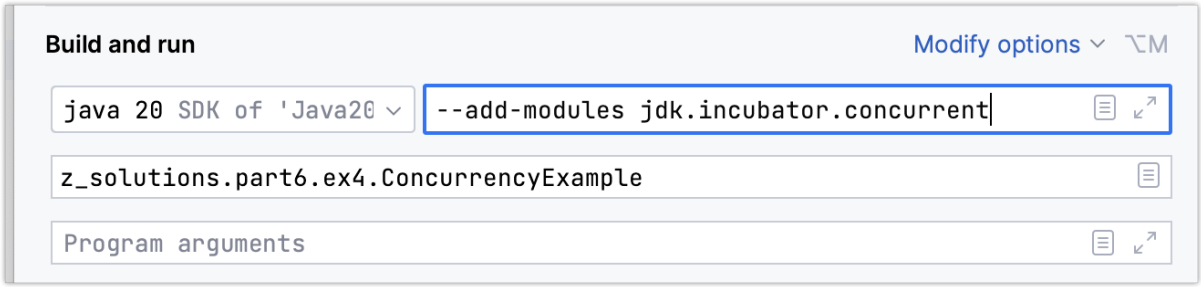
```
static void executeTasks(boolean forceFailure) throws InterruptedException,
                                                      ExecutionException
{
    try (var executor = Executors.newFixedThreadPool(50)) {
        Future<String> task1 = executor.submit(() -> {
            return "1";
      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

Mithilfe von Structured Concurrency soll der `ExecutorService` ersetzt werden und die Strategie `ShutdownOnFailure` die Verarbeitung im Fehlerfall klarer machen. Analysiere die Abarbeitungen im Fehlerfall.

Tipp: Es handelt sich in Java 20 noch um ein Incubator-Feature, weshalb man beim Kompilieren und Start passende Konfigurationen vornehmen muss. In Java 21 ist es bereits ein Preview Feature und erfordert keine Angabe des Incubator-Moduls mehr.



Aufgabe 6 – Experimentiere mit Template Processor

Schreibe einen eigenen Template Processor, der die Werte mit [[und]] oder alternativ jeweils vorne und hinten einem ‘ umschließt:

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

=>

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

a) Verwende dazu `fragments()` und `values()` und eine Schleife.
b) Vereinfache das Ganze durch `interpolate()` und das Stream-API.
c) Nutze einen parametrierbaren Lambda, um die Start- und Endsequenz frei wählbar zu machen.
d) Erzeuge einen Template Processor der ähnlich zu den f-Strings in Python (`f"Berechnung: {x} + {y} = {x + y}"`) arbeitet, ohne direkt STR zu referenzieren.

Aufgabe 7 – Experimentiere mit Unnamed Patterns and Variables

Vereinfache die folgende Methode durch Einsatz von Unnamed Patterns and Variables, um die Lesbarkeit und Verständlichkeit zu steigern – nutze aus, dass die IDEs unbenutzte Variablen anzeigen:

```
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;
}
```