Workshop: Best of Java 12 bis 16
Übungen

Ablauf

Dieser Workshop gliedert sich in mehrere Vortragsteile, die den Teilnehmern die Thematik Java 12 bis 16 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 16 installiert
2) Aktuelles Eclipse installiert (Alternativ: NetBeans oder IntelliJ IDEA)

Teilnehmer

- Entwickler mit Java-Erfahrung sowie
- SW-Architekten, die Java 12 bis 16 kennenlernen/evaluieren möchten

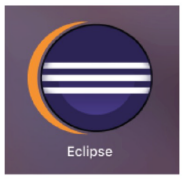
Kursleitung und Kontakt

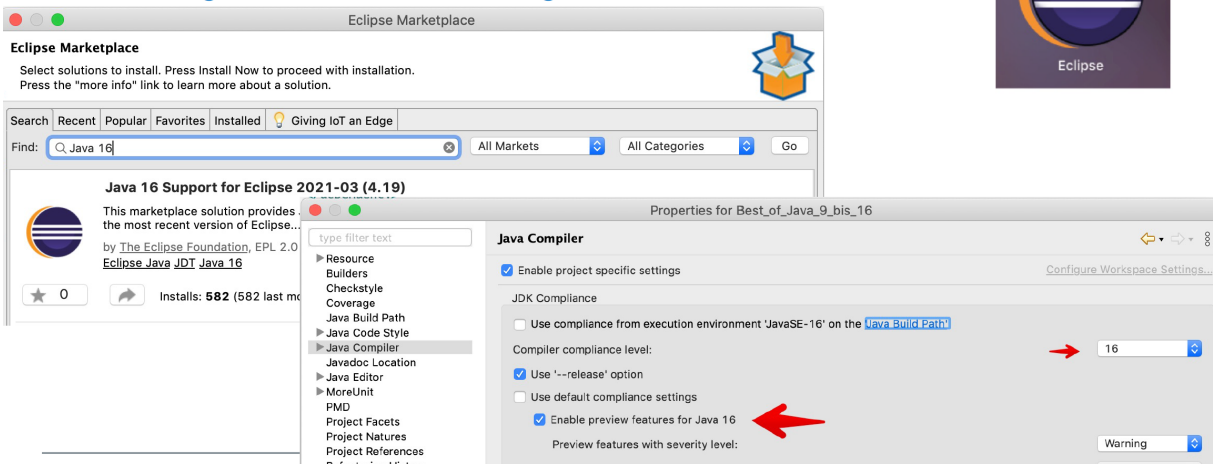
Michael Inden
Freiberuflicher Buchautor, Trainer und Konferenz-Speaker
E-Mail: michael.inden@hotmail.com
Blog: https://jaxenter.de/author/minden

Konfiguration Eclipse / IntelliJ

Bedenken Sie bitte, dass wir vor den Übungen noch einige Kleinigkeiten konfigurieren müssen.

- **Eclipse 2021-03: Installation von Plugin nötig**
- **Aktivierung von Preview-Features nötig**





- **Aktivierung von Preview-Features nötig**

PART 1/2: Neuerungen in Java 12 bis 16

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

Aufgabe 1 – Syntaxänderungen bei switch

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

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

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

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

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

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

Aufgabe 1a

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

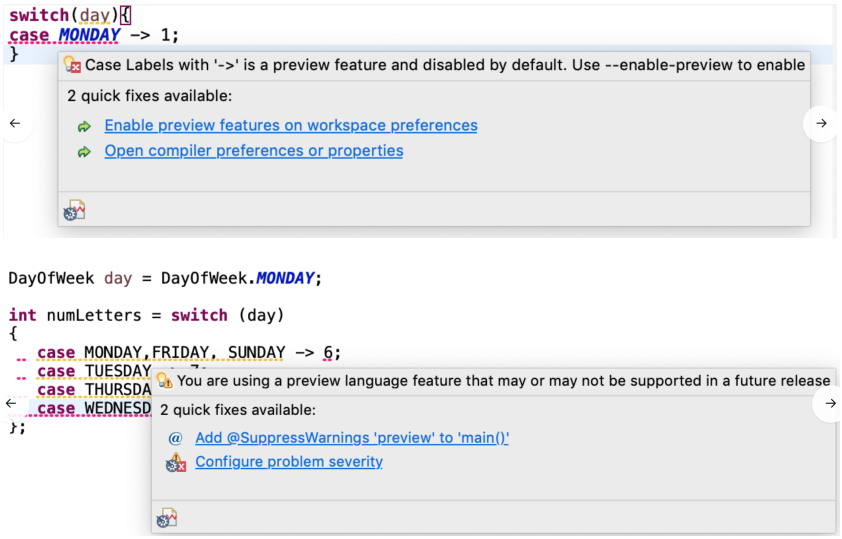
Aufgabe 1b

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

Aufgabe 1c

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

Tipps: Aktivierung des Preview-Features und Unterdrücken von Warnungen:



Aufgabe 2: Die Klasse `CompactNumberFormat`

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

Nutze die nachfolgenden Werte zum Parsing:

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

Aufgabe 3: Strings

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

Aufgabe 3a

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

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

Aufgabe 3b

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

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

Aufgabe 4: Strings

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

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

Aufgabe 4a

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

Aufgabe 4b

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

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

Aufgabe 5: Teeing-Kollektor

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

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

Aufgabe 6: Teeing-Kollektor

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

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

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

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

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

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

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

Die erwarteten Resultate sind mit

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

Aufgabe 7: Teeing-Kollektor

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

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

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

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

Gegeben sei noch die Klasse `City` wie folgt:

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

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

      public String getName()
      {
            return name;
      }

      public String getRegion()
      {
            return region;
      }

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

Aufgabe 8 – Text Blocks

Vereinfache folgenden Sourcecode mit einem herkömmlichen String, der über mehrere Zeilen geht und nutze die in Java 13 eingeführte Syntax.

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

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

String java13FeatureObjOld = ""
              + "{\n"
              + "    version: \"Java13\",\n"
              + "    feature: \"text blocks\",\n"
              + "    attention: \"preview!\"\n"
              + "}\n";
```

Aufgabe 9 – Text Blocks mit Platzhaltern

Vereinfache folgenden Sourcecode mit einem herkömmlichen String, der über mehrere Zeilen geht und nutze die in Java 13 eingeführte Syntax:

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

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

Produziere folgende Ausgaben mit der neuen Syntax:

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

Aufgabe 10 – Record-Grundlagen

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

```
class Square
{
     public final double sideLength;

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

class Circle
{
     public final double radius;

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

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

Aufgabe 11 – Record

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

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

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

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

Aufgabe 12 – `instanceof`-Grundlagen

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

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

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

Aufgabe 13 – `instanceof` und `record`

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

```
record Square(double sideLength) {
}

record Circle(double radius) {
}

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

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

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