Workshop: Best of Java 18 bis 20 Übungen

Ablauf

Dieser Workshop gliedert sich in mehrere Vortragsteile, die den Teilnehmern die Thematik Java 18 bis 20 sowie die dortigen Neuerungen überblicksartig näherbringen. Im Anschluss daran sind jeweils einige Übungsaufgaben von den Teilnehmern – idealerweise in Gruppenarbeit – am Rechner zu lösen.

Dieser Add on Teil fokussiert auf modernes Java 20.

Voraussetzungen

1) Aktuelles JDK 20 installiert
2) Aktuelles Eclipse installiert (Alternativ: IntelliJ IDEA)

Teilnehmer

- Entwickler mit Java-Erfahrung sowie
- SW-Architekten, die Java 20 kennenlernen/evaluieren möchten

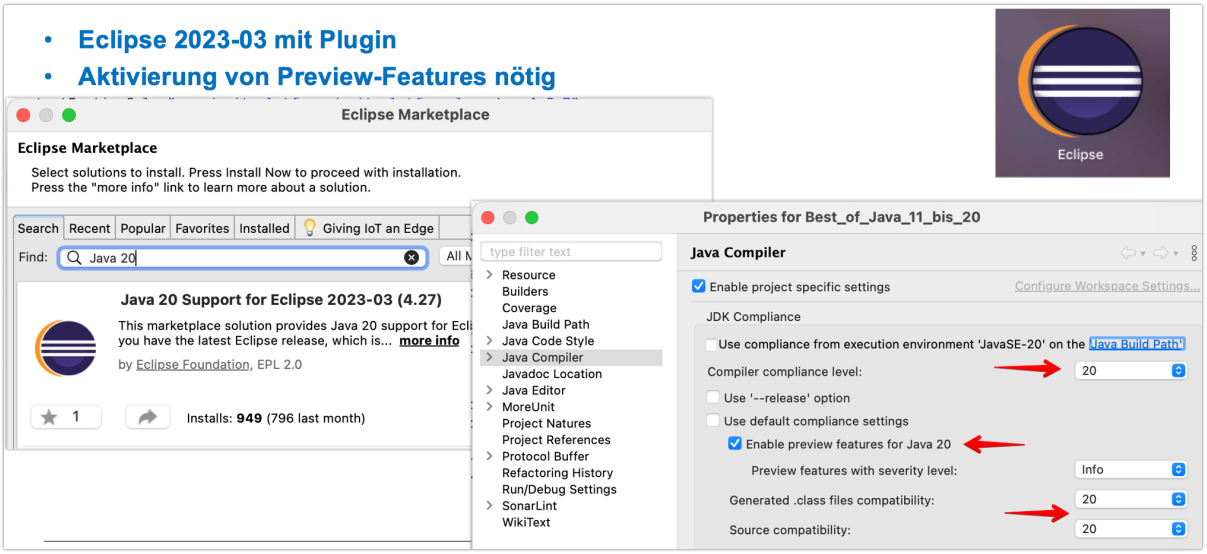
Kursleitung und Kontakt

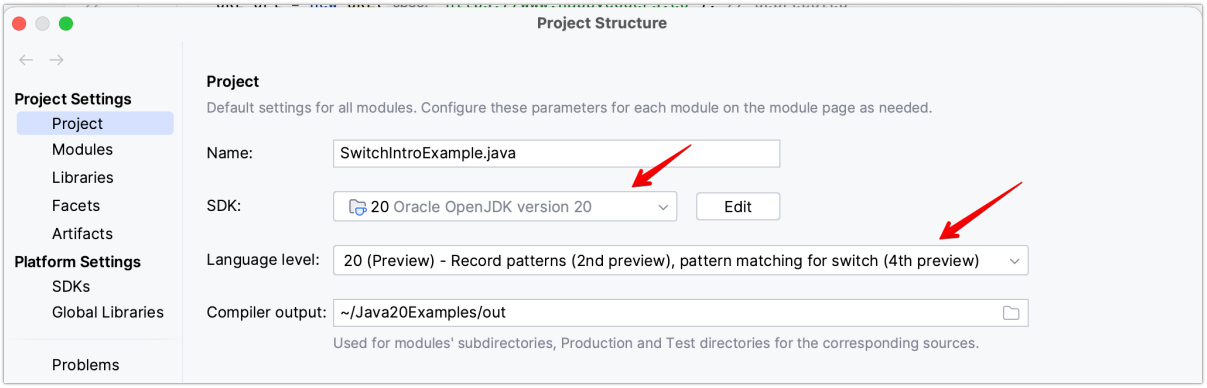
Michael Inden

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

Konfiguration Eclipse / IntelliJ

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





PART 6: Neuerungen in Java 18 bis 20

Lernziel: In diesem Abschnitt beschäftigen wir uns mit Erweiterungen in Java 18 bis 20.

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 ergänzt werden soll, sodass für die beiden Figuren `Line` und `Triangle` die Punkte addiert werden sollen:

```
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 – Wandle mit Structured Concurrency um

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

```
private 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 um ein Incubator-Feature, weshalb man beim Kompilieren und Start passende Konfigurationen vornehmen muss.