Workshop
Best of Modern Java 21 – 25
Meine Lieblingsfeatures
Übungen

Ablauf

Dieser Workshop stellt wesentliche Neuerungen aus Java 25 LTS und Vorgängern überblicksartig vor. Zum Vertiefen des Erlernten sind ergänzend jeweils einige Übungsaufgaben von den Teilnehmern – idealerweise in Gruppenarbeit – am Rechner zu lösen.

Voraussetzungen

1) Aktuelles JDK 21 LTS (21.0.4 oder neuer) sowie aktuelles JDK 25 LTS installiert
2) Aktuelles Eclipse 2025-09 mit Java-25-Plugin oder IntelliJ IDEA 2025.2 oder neuer installiert

Teilnehmer

- Entwickler mit Java-Erfahrung sowie
- SW-Architekten, die Java 21 LTS bis 25 LTS kennenlernen/evaluieren möchten

Kursleitung und Kontakt

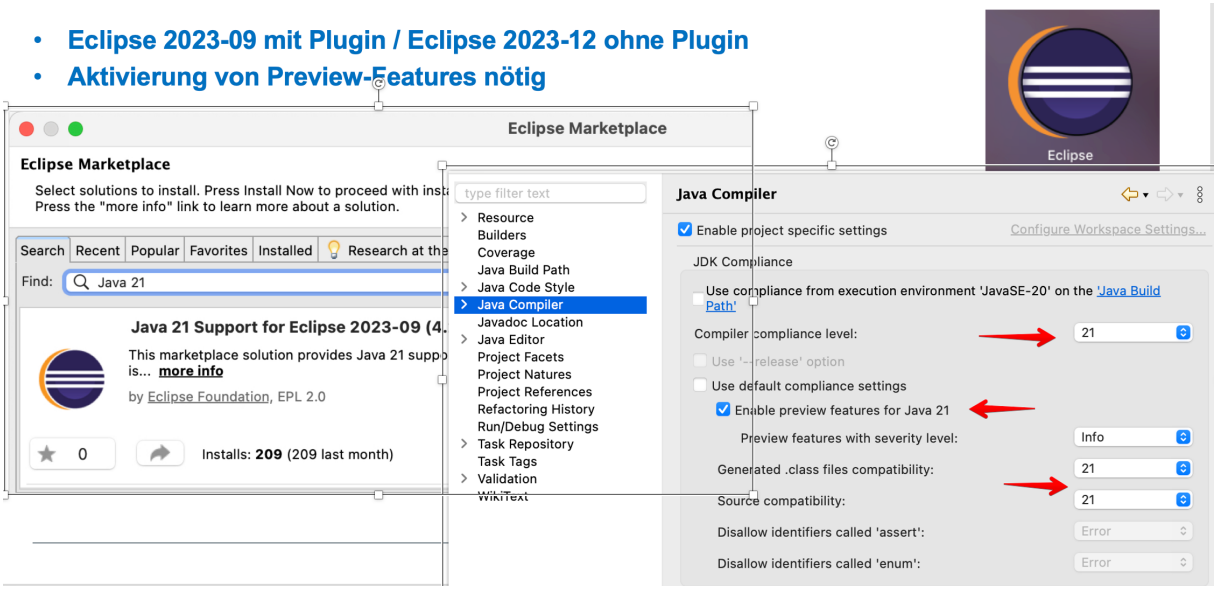
Michael Inden
Freiberuflicher Consultant, Buchautor, Trainer und Konferenz-Speaker
E-Mail: michael_inden@hotmail.com

Weitere Kurse (Java, Unit Testing, Design Patterns, JPA, Spring) biete ich gerne auf Anfrage als Online- oder Inhouse-Schulung an.

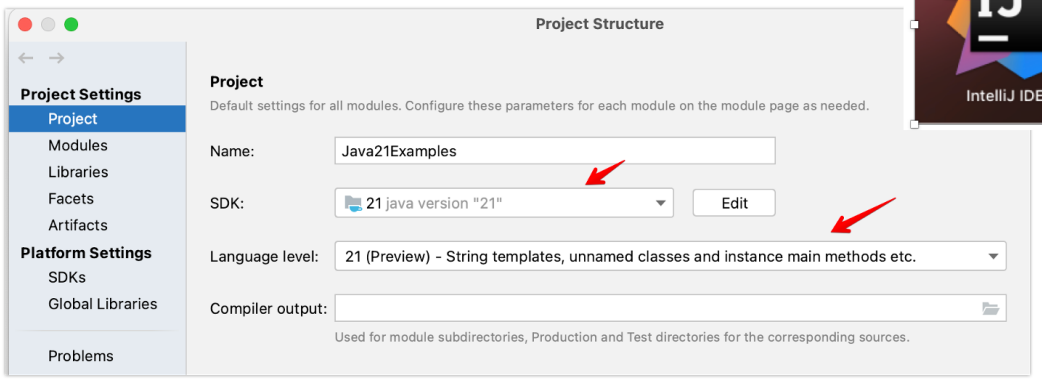
Konfiguration Eclipse / IntelliJ für Java 21 LTS

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

- **Eclipse 2023-09 mit Plugin / Eclipse 2023-12 ohne Plugin**
- **Aktivierung von Preview-Features nötig**



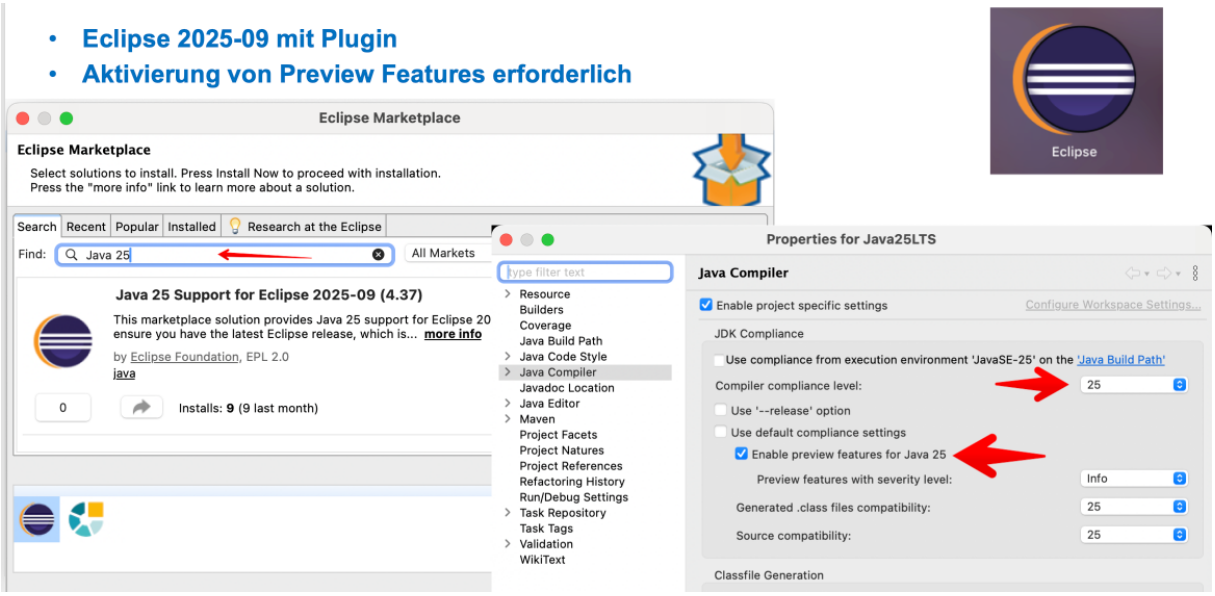
- **Aktivierung von Preview-Features nötig**



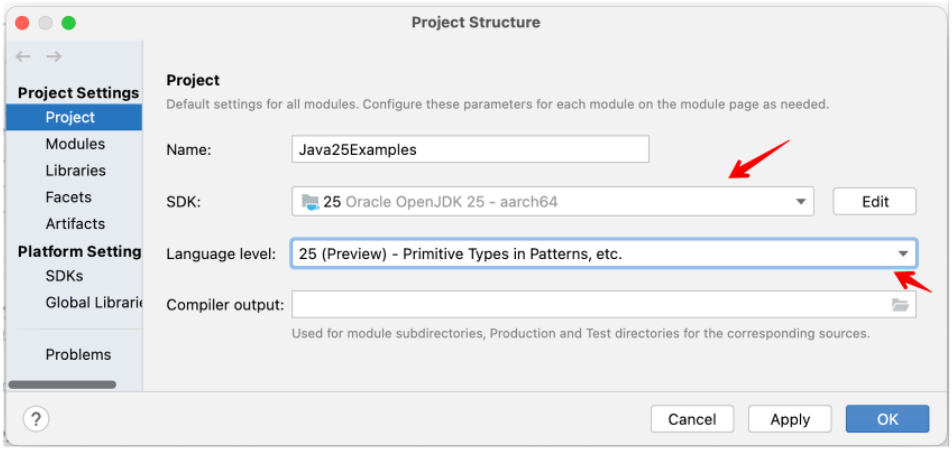
Konfiguration Eclipse / IntelliJ für Java 25 LTS

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

- **Eclipse 2025-09 mit Plugin**
- **Aktivierung von Preview Features erforderlich**



- **IntelliJ 2025.2.1**
- **Aktivierung von Preview Features erforderlich**



PART 1: Neuerungen in Java 18 bis 21 LTS

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

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(Integer zipCode, String name) {
}

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

Zur Gültigkeitsprüfung werden verschiedene Konsistenz-Checks und Prüfungen ausgeführt. Dabei werden verschachtelte Bestandteile wie Person oder City eines Journey-Objekts auf `!= null` geprüft und somit deren Existenz für eine nachfolgende Abfrage abgesichert. Dazu sieht man mitunter – vor allem in Legacy-Code – Implementierungen, die tief verschachtelte `if`s und diverse `null`-Prüfungen enthalten, etwa wie folgt:

```
static boolean checkFirstNameTravelTimeAndDestZipCode(final Object obj) {
    if (obj instanceof Journey journey) {
        if (journey.person() != null) {
            var person = journey.person();

            if (journey.travelInfo() != null) {
                var travelInfo = journey.travelInfo();

                if (journey.to() != null) {
                    var to = journey.to();
```

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

Bonus: Vereinfache die Angaben in den Record Patterns 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 Punkts multipliziert. Das ist für `Point` bereits realisiert. Das `switch` soll so ergänzt werden, dass `case`s für `Line` und `Triangle` hinzugefügt werden. Als Berechnung sollen die jeweiligen Teilkomponenten in Form von Points addiert werden, indem die bisherige Methode `process()` aufgerufen wird:

```
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

Als Ausgangsbasis für diese Aufgabe ist eine Ausführung verschiedener Tasks mithilfe eines `ExecutorService` und einer Pool-Size von 50 folgendermaßen gegeben:

```
try (var executor = Executors.newFixedThreadPool(50)) {
    IntStream.range(0, 1_000).forEach(i -> {
        executor.submit(() -> {
            Thread.sleep(Duration.ofSeconds(2));

            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 Methode 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.

```
static void primeNumbers()
{
    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
}
```

Bonus

Experimentiere mit dem Interface `SequencedSet<E>` und erstellen mit den passenden Methoden eine sortierte Menge, bestehend aus den Buchstaben A, B und C:

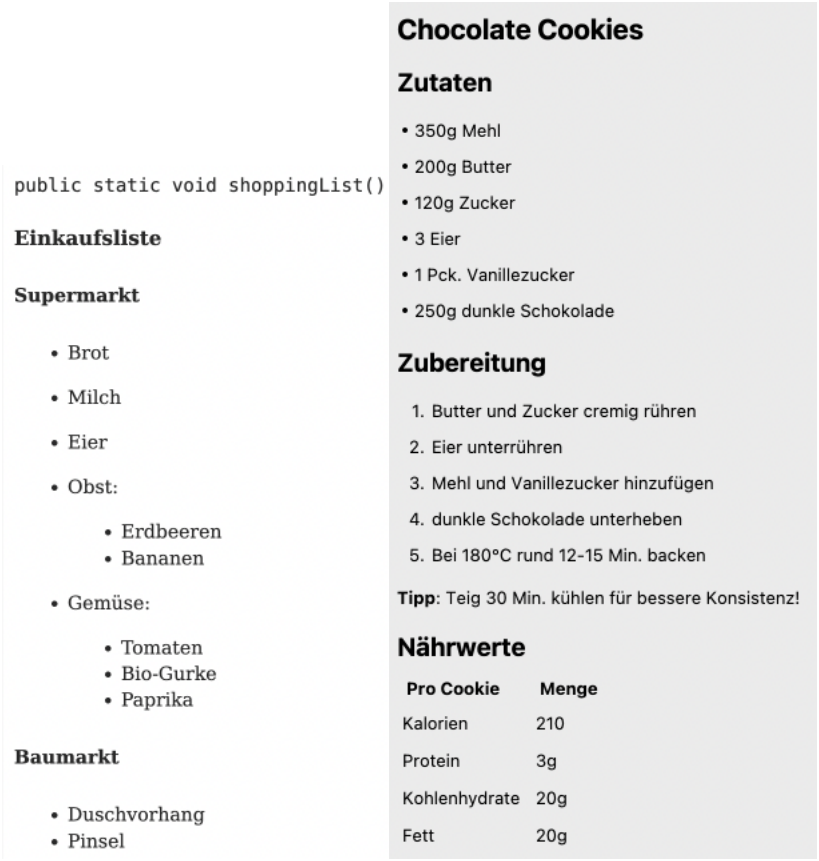
```
static void createABCSet()
{
    Set<String> numbers = new LinkedHashSet<>();
    // TODO

    // TODO print first and last element
    // TODO print reverser order
}
```

PART 2: Neuerungen in Java 22 bis 25 LTS

Lernziel: In diesem Abschnitt beschäftigen wir uns mit Erweiterungen in Java 22 bis 25 LTS.

Aufgabe 1 – Kennenlernen von Markdown-Kommentaren

Seit Java 23 kann man Markdown zur Definition von JavaDoc-Kommentaren verwenden. Damit lässt sich einiges präziser ausdrücken. Experimentiere ein wenig mit Markdown herum, um beispielsweise eine ToDo-Liste oder eine Einkaufsliste oder ein Backrezept beschreiben – Überschriften erzeugt man mit `# Level 1` und `## Level 2` usw. Der Kreativität sind in dieser Aufgabe keine Grenzen gesetzt. Gerne dürfen auch verschiedene Schriftarten oder kleinere Code-Schnipsel in die Dokumentation aufgenommen werden.



```
public static void shoppingList()
```

Einkaufsliste

Supermarkt

- Brot
- Milch
- Eier
- Obst:
  - Erdbeeren
  - Bananen
- Gemüse:
  - Tomaten
  - Bio-Gurke
  - Paprika

Baumarkt

- Duschvorhang
- Pinsel

Chocolate Cookies

Zutaten

- 350g Mehl
- 200g Butter
- 120g Zucker
- 3 Eier
- 1 Pck. Vanillezucker
- 250g dunkle Schokolade

Zubereitung

1. Butter und Zucker cremig rühren
2. Eier unterrühren
3. Mehl und Vanillezucker hinzufügen
4. dunkle Schokolade unterheben
5. Bei 180°C rund 12-15 Min. backen

Tipp: Teig 30 Min. kühlen für bessere Konsistenz!

Nährwerte

Pro Cookie	Menge
Kalorien	210
Protein	3g
Kohlenhydrate	20g
Fett	20g

Aufgabe 2 – Kennenlernen der Standard-Gatherers

Lerne das Interface `Gatherer` als Grundlage für Erweiterungen von Intermediate Operations mit seinen Möglichkeiten kennen.

Ergänze den folgenden Programmschnipsel, um das Produkt aller Zahlen im Stream zu bilden. Nutze dazu einen der neuen vordefinierten `Gatherer` aus der Klasse `Gatherers`.

```
var crossMult = Stream.of(1, 2, 3, 4, 5, 6, 7);
// TODO
// crossMult ==> Optional[5040]
```

Außerdem sollen folgende Eingabedaten in jeweils Gruppen von drei Elementen aufgeteilt werden:

```
var values = Stream.of(1, 2, 3, 10, 20, 30, 100, 200, 300);
// TODO
// [[1, 2, 3], [10, 20, 30], [100, 200, 300]]
```

Aufgabe 3 – Nutze passende Standard-Gatherer, um Koordinateninformationen zu verarbeiten und Temperatursprünge zu finden

Gegeben sind 3D-Koordinaten in Form eines Stream mit einzelnen Werten für x, y, z:

```
record Point3d(int x, int y, int z) {
}

var coordinates = Stream.of(0, 0, 0, 10, 20, 30, 100, 200, 300,
                            1000, 2000, 3000).
                         gather(/* TODO*/)
                         /* TODO */;
```

Als Ergebnis wird Folgendes erwartet:

```
coordinates: [Point3d[x=0, y=0, z=0], Point3d[x=10, y=20, z=30],
Point3d[x=100, y=200, z=300], Point3d[x=1000, y=2000, z=3000]]
```

BONUS: Ergänze einen Konstruktor im Record, um die Konstruktion zu vereinfachen.

Basierend auf eine Zeitreihe mit Temperaturdaten sollen diejenigen Paare gefunden werden, wo es Temperaturschwankungen von über 20 Grad gab:

```
var temps = Stream.of(0, 10, 7, 20, 25, 12, 17, 40, 10, 20, 42, 30).
                  // gather(/* TODO */).
                  // filter(/* TODO */).
                  toList();
```

Das folgende Resultat wird erwartet:

```
temp jumps: [[17, 40], [40, 10], [20, 42]]
```

Aufgabe 4 – Kennenlernen von Flexible Constructor Bodies

Entdecke die Eleganz durch die neue Syntax, Aktionen vor dem Aufruf von `super()` ausführen zu können. Es soll eine Gültigkeitsprüfung von Parametern vor der Konstruktion der Basisklasse erfolgen.

```
public Rectangle(Color color, int x, int y, int width, int height)
{
    super(color, x, y);

    if (width < 1 || height < 1) throw
          new IllegalArgumentException("width and height must be positive");

    this.width = width;
    this.height = height;
}
```

Aufgabe 5 – Kennenlernen von Flexible Constructor Bodies

In dieser Aufgabe nimmt die Basisklasse einen anderen Typ entgegen als die Subklasse `StringMsgOld`. Dabei kommt der Trick mit der Hilfsmethode ins Spiel. Neben einer Prüfung und Konvertierung erfolgen dort noch aufwändige Aktionen. Die Aufgabe ist nun, das Ganze lesbarer mit der neuen Syntax umzuwandeln – was sind die weiteren Vorteile dieser Variante?

```
public StringMsgOld(String payload)
{
    super(convertToByteArray(payload));
}

private static byte[] convertToByteArray(final String payload)
{
    if (payload == null)
        throw new IllegalArgumentException("payload should not be null");

    String transformedPayload = heavyStringTransformation(payload);
    return switch (transformedPayload) {
        case "AA" -> new byte[]{1, 2, 3, 4};
        case "BBBB" -> new byte[]{7, 2, 7, 1};
        default -> transformedPayload.getBytes();
    };
}

private static String heavyStringTransformation(String input) {
    return input.repeat(2);
}
```

Aufgabe 6 – 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 dem Standard-Joiner `awaitAllSuccessfulOrThrow` die Verarbeitung im Fehlerfall klarer machen. Analysiere die Abarbeitungen im Fehlerfall.

Führen wir die Methode einmal mit beiden Wertebelegungen aus:

```
jshell> import java.util.concurrent.*

jshell> executeTasks(false)
 result: 1 / 2 / 3

jshell> executeTasks(true)
| Ausnahme java.util.concurrent.ExecutionException:
java.lang.IllegalStateException: FORCED BUG
   at FutureTask.report (FutureTask.java:122)
   at FutureTask.get (FutureTask.java:191)
   at executeTasks (#19:16)
   at (#21:1)
|  Verursacht von: java.lang.IllegalStateException: FORCED BUG
|        at lambda$executeTasks$1 (#19:8)
```

Aufgabe 7 – Besonderheiten der Structured Concurrency

Structured Concurrency bietet nicht nur den bereits (bestens) bekannte Joiner `awaitAllSuccessfulOrThrow`, die beim Auftreten eines Fehlers alle anderen Berechnungen stoppt, sondern auch die für einige Anwendungsfälle praktische Strategie `anySuccessfulResultOrThrow`. Damit lassen sich mehrere Berechnungen beginnen und nachdem eine ein Ergebnis geliefert hat, alle anderen Teilaufgaben stoppen. Wozu kann das nützlich sein? Stellen wir uns verschiedene Suchanfragen vor, bei denen die Schnellste gewinnen soll.

Als Aufgabe sollen wir den Verbindungsaufbau zum Mobilnetz in den Varianten 5G, 4G, 3G und WiFi modellieren. Befülle nachfolgendes Programmstück mit Leben:

```
public static void main(final String[] args) throws ExecutionException,
        InterruptedException
{
    var joiner = StructuredTaskScope.Joiner.
                 <NetworkConnection>anySuccessfulResultOrThrow();
    try (var scope = StructuredTaskScope.open(joiner))
    {
        // TODO
        StructuredTaskScope.Subtask<NetworkConnection> result1 = null;
        StructuredTaskScope.Subtask<NetworkConnection> result2 = null;
        StructuredTaskScope.Subtask<NetworkConnection> result3 = null;
        StructuredTaskScope.Subtask<NetworkConnection> result4 = null;
        // TODO

        NetworkConnection result = null; // TODO

        System.out.println("Wifi " + result1.state() +
                           "/5G " + result2.state() +
                           "/4G " + result3.state() +
                           "/3G " + result4.state());

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