Workshop: Best of Java 17 LTS bis 24

Übungen speziell für Java 24

Ablauf

Dieser Workshop gliedert sich in mehrere Vortragsteile, die den Teilnehmern die Thematik Java 17 LTS bis 24 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 24 und JDK 21 LTS (21.0.4 oder neuer) installiert
2) Aktuelles Eclipse 2025-03 mit Java-24-Plugin oder IntelliJ IDEA 2024.3.1 (oder neuer) installiert

Teilnehmer

- Entwickler mit Java-Erfahrung sowie
- SW-Architekten, die Java 17 LTS bis 24 kennenlernen/evaluieren möchten

Kursleitung und Kontakt

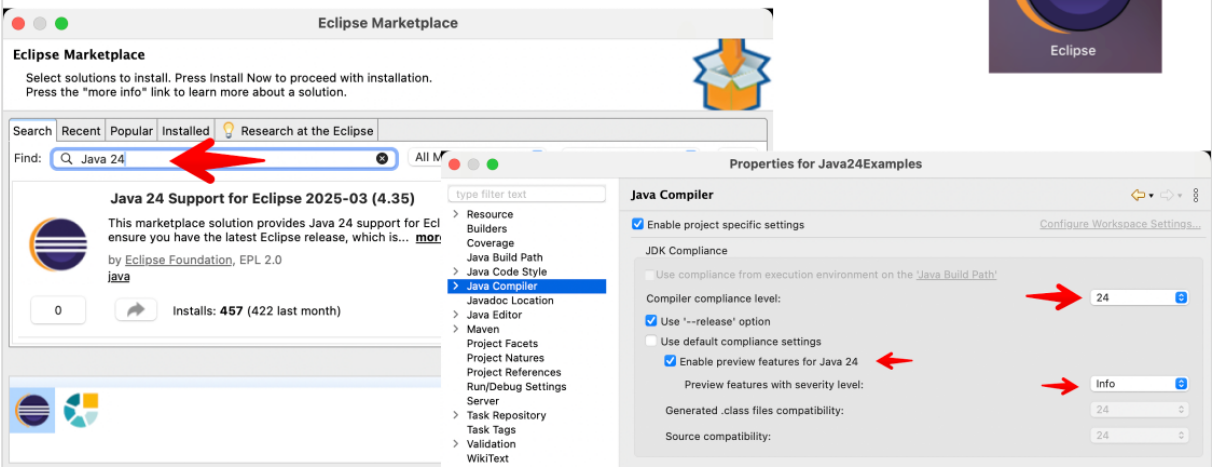
Michael Inden
Head of Development, freiberuflicher Buchautor, Trainer und Konferenz-Speaker
E-Mail: michael_inden@hotmail.com

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

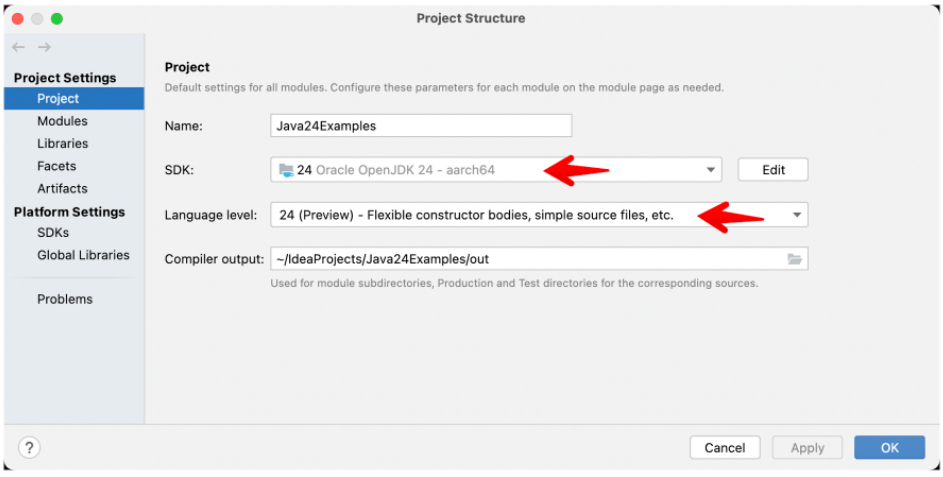
Konfiguration Eclipse / IntelliJ für Java 24

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

- **Eclipse 2025-03 mit Plugin**
- **Aktivierung von Preview Features erforderlich**



- **Aktivierung von Preview Features erforderlich**



PART 3: Erweiterungen in Java 22, 23 und 24

Lernziel: Kennenlernen von Syntax-Neuerungen und verschiedenen API- sowie JVM-Erweiterungen in Java 22 bis 24 anhand von Beispielen.

PART 3a:

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 – Erstelle einen DistinctBy-Gatherer, um Duplikate bezüglich eines Kriteriums zu entfernen

Es soll ein eigener Gatherer `distinctBy()` erstellt werden, der wie nachfolgend gezeigt arbeitet – als Bonus kann man sich an einer parallelen Variante mit `Combiner` versuchen.

```
Stream.of(1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5).
       gather(distinctBy(n -> n)).findFirst());
// => distinctBy(n -> n): Optional[[1, 2, 3, 4, 5]]

Stream.of("Maik", "Mike", "Jim", "Tim", "Tom", "Jim",
          "Jim", "John", "Sophie", "Joelle", "Stefan",
          "Anne", "Lili", "Michael", "Andreas").
       gather(distinctBy(String::length)).findFirst());
// => distinctBy(String::length): Optional[[Maik, Jim, Sophie, Michael]]
```

Aufgabe 5 – Scoped Values als `ThreadLocal`-Alternative

In dieser Aufgabe soll ein herkömmlich per Parameterübergabe realisiertes Programm auf Scoped Values umgestellt werden. Dabei wird eine Request-Verarbeitung mit einer mehrschichtigen Aufrufhierarchie mit simplifizierten Controller- und Service-Klassen nachempfunden. Den Ausgangspunkt bildet die Klasse `ScopedValuesExample`, die zwei Scoped Values definiert, um Informationen zum eingeloggten Benutzer sowie zum Zeitpunkt des Requests bereitstellen zu können. Derzeit sind diese noch ungenutzt und die Informationen werden jeweils als Parameter weitergegeben:

```
public static void main(String[] args) throws Exception
{
    // Simuliere Requests
    for (String name : List.of("ATTACKER", "ADMIN"))
    {
        var user = new User(name, name.toLowerCase());
        controller.consumingMethod(user, ZonedDateTime.now());

        controller.consumingMethod(user, null); // no time passed

        String answer = controller.process(user, ZonedDateTime.now());
        System.out.println(answer);

        String answer2 = controller.process(user, null); // no time passed
        System.out.println(answer2);
    }
}
```

Ihre Aufgabe ist es, dass auf Scoped Values umzustellen und diese passend mit Werten zu befüllen und deren Propagation in der Aufrufkette von `ScopedValuesExample` über `Controller` und `Service` zu implementieren. In den beiden Klassen soll dann auf die Informationen zugegriffen werden, ohne dass diese bei den Methodenaufrufen als Parameter übergeben werden müssen.

Aufgabe 6 – Primitive Types in Patterns, instanceof, and switch

Vor Java 23 war es nicht möglich, `switch` mit primitiven Typen zu nutzen. Vereinfache den Sourcecode, um auf das Boxing verzichten zu können und von Pattern Matching mit Bedingungen profitieren zu können:

```
int value = 42;

// necessary to work with switch below Java 23
Integer boxedValue = value;
switch (boxedValue) {
    case Integer i when i > 0 && i < 10 -> log(i + " is lower than 10");
    case Integer i when i >= 10 && i < 40 -> log(i + " is lower than 40");
    case Integer i when i >= 40 && i < 70 -> log(i + " is >= 40");
    case Integer i -> log("Some unexpected value is perfect: " + i);
}
```

Wende Primitive Type Patterns an, um das `switch` zu vereinfachen.

PART 3b:

Aufgabe 7 – Experimentieren mit dem Vector API

Das Vector API erlaubt es, SIMD-Berechnungen durchzuführen. In dieser Aufgabe soll eine als Skalarberechnung vorgegebene Implementierung auf des Vector API umgestellt werden.

Gegeben ist die Skalarberechnung der Formel $a_i * b_i + a_i * b_i – (a_i + b_i)$ mit dieser Implementierung:

```
static float[] scalarComputation(float[] a, float[] b)
{
    float[] c = new float[a.length];

    for (int i = 0; i < a.length; i++)
    {
        c[i] = a[i] * b[i] + a[i] * b[i] - (a[i] + b[i]);
    }

    return c;
}
```

Wandle das in passende Aufrufe mit dem Vector API um. Experimentiere ein wenig:

1) Was passiert, wenn man die Schleife nicht korrekt bezüglicher aller Durchläufe abbildet?
2) Wie kann man die Formel vereinfachen? Lässt sich dort von einer Skalarmultiplikation, also mit einem fixen Wert, profitieren?

Aufgabe 8 – Kennenlernen von Flexibile Constructor Bodies/Statements before super(…)

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 9 – Kennenlernen von Statements before super(…)

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 10 – Modul-Import zur Vereinfachung von vielen einfachen Imports

Stelle dir ein Programm mit einigen Import-Anweisungen für Dinge aus dem JDK wie folgt vor:

```
import javax.swing.*;
import java.awt.*;
import java.io.IO;
import java.time.LocalDate;
import java.util.List;
import java.util.Map;
import java.util.function.Function;
import java.util.stream.Collectors;
import java.util.stream.Stream;
```

Vereinfache diese individuellen Imports durch passende Modul-Imports und bedenke und behandle potenzielle Mehrdeutigkeiten.

Aufgabe 11 – Besonderheiten der Structured Concurrency

Structured Concurrency bietet nicht nur die bereits (bestens) bekannte Strategie `ShutdownOnFailure`, die beim Auftreten eines Fehlers alle anderen Berechnungen stoppt, sondern auch die für einige Anwendungsfälle praktische Strategie `ShutdownOnSuccess`. 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
{
    try (var scope =
         new StructuredTaskScope.ShutdownOnSuccess<NetworkConnection>())
    {
        // TODO
        StructuredTaskScope.Subtask<NetworkConnection> result1 = null;
        StructuredTaskScope.Subtask<NetworkConnection> result2 = null;
        StructuredTaskScope.Subtask<NetworkConnection> result3 = null;
        StructuredTaskScope.Subtask<NetworkConnection> result4 = null;
        // TODO

        System.out.println(STR."Wifi \{result1.state()}/5G \{result2.state()}" +
                           STR."/4G \{result3.state()}/3G \{result4.state()}");
        System.out.println("found connection: " + scope.result());
    }
}
```

BONUS: Was passiert, wenn in einer der Methoden eine Exception ausgelöst wird?