Workshop: Best of Java 17 – 23
Exercises especially for Java 23

Procedure

This workshop is divided into several lecture parts, which introduce the subject Java 17 to 23, and gives an overview of the innovations. Following this, some exercises are to be solved by the participants - ideally in pair / group programming - on the computer.

Requirements

1) Current JDK 23 is installed
2) Current Eclipse 2024-09 with Java 23 Plugin or IntelliJ IDE 2024.2.2 is installed

Attendees

- Developers with Java experience, as well as
- SW-Architects, who would like to learn/evaluate Java 17 to 23.

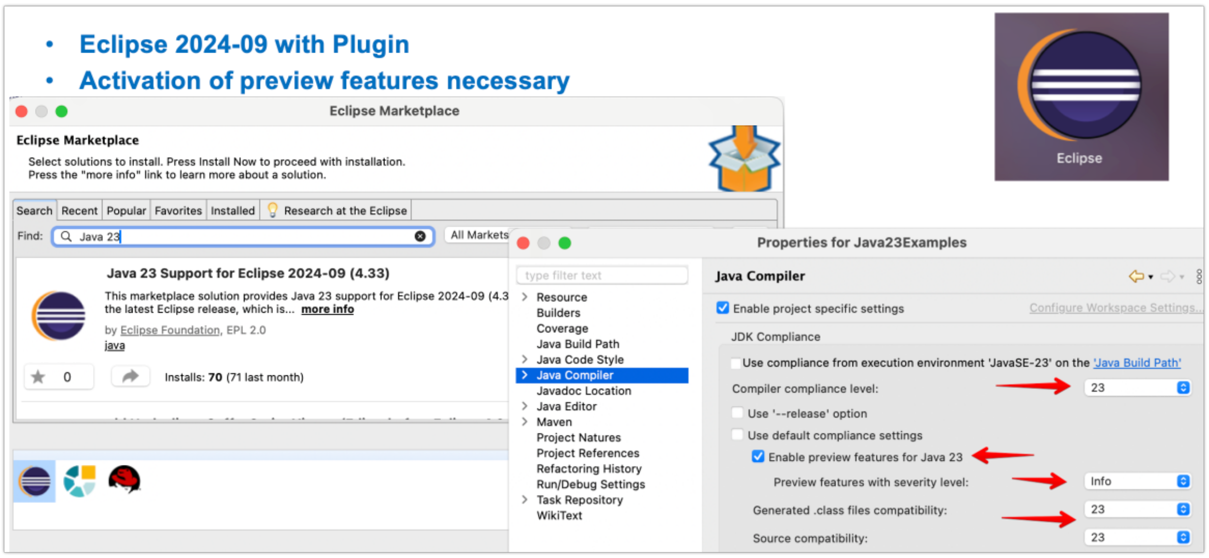
Course management and contact

Michael Inden
Head of Development, independent SW Consultant, Author, and Trainer
E-Mail: michael_inden@hotmail.com

Please do not hesitate to ask: Further courses (Java, Unit Testing, Design Patterns) are available on request as in-house training.

Configuration of Eclipse / IntelliJ for Java 23

Please note that we have to configure some small things before the exercises if you are not using the latest IDEs.



- Eclipse 2024-09 with Plugin
- Activation of preview features necessary

Eclipse Marketplace

Select solutions to install. Press Install Now to proceed with installation.
Press the "more info" link to learn more about a solution.

Find: Java 23

Java 23 Support for Eclipse 2024-09 (4.33)

This marketplace solution provides Java 23 support for Eclipse 2024-09 (4.3 the latest Eclipse release, which is... more info

by Eclipse Foundation, EPL 2.0
java

Installs: 70 (71 last month)

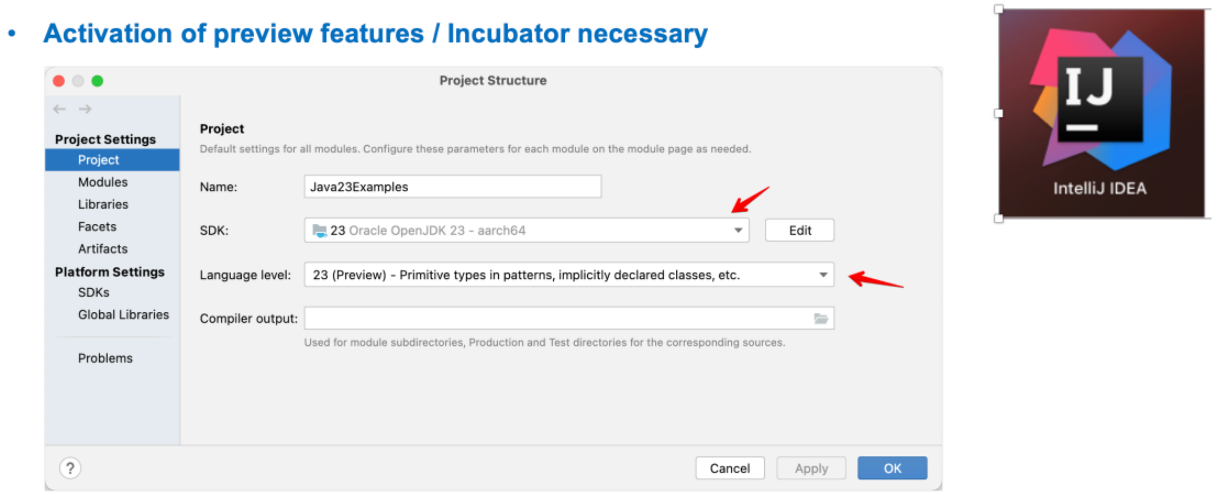
Properties for Java23Examples

Java Compiler

- [✓] Enable project specific settings

JDK Compliance

- [ ] Use compliance from execution environment 'JavaSE-23' on the 'Java Build Path'
- Compiler compliance level: 23
- [ ] Use '--release' option
- [ ] Use default compliance settings
- [✓] Enable preview features for Java 23
- Preview features with severity level: Info
- Generated .class files compatibility: 23
- Source compatibility: 23



- Activation of preview features / Incubator necessary

Project Structure

Project

Default settings for all modules. Configure these parameters for each module on the module page as needed.

- Name: Java23Examples
- SDK: 23 Oracle OpenJDK 23 - aarch64
- Language level: 23 (Preview) - Primitive types in patterns, implicitly declared classes, etc.
- Compiler output:

Used for module subdirectories, Production and Test directories for the corresponding sources.

PART 4: Enhancements in Java 22 and 23

Objective: To get to know syntax innovations and various API and JVM enhancements in Java 22 and 23 using examples.

Exercise 1 – Statements before super(…)

Discover the elegance of the new syntax for executing actions before `super()` is called. You should perform a validity check of parameters before constructing the base class.

```
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;
}
```

Exercise 2 – Statements before super(…)

The base class accepts a different type than the subclass `StringMsgOld`. Use the ability to execute actions before `super()` is called. Here, the base class accepts a different type than the subclass. This is where the trick with the helper method comes into play. In addition to a check and conversion, further complex actions are carried out there. The task now is to convert the whole thing into a more readable form using the new syntax – what are the other advantages of this variant?

```
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);
}
```

Exercise 3 – Predefined Gatherers

Get to know the new interface `Gatherer` as a basis for extensions of intermediate operations and their possibilities.

Complete the following program snippet to calculate the product of all numbers in the stream. Use one of the new predefined gatherers from the `Gatherers` class.

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

In addition, the following input data is to be divided into groups of 3 elements:

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

Exercise 4 – Structured Concurrency

Structured concurrency offers not only the well-known `ShutdownOnFailure` strategy, which stops all other calculations when an error occurs but also the `ShutdownOnSuccess` strategy, which is helpful for some applications. This strategy allows multiple calculations to be started and all other subtasks to be stopped after one has returned a result. What can this be useful for? Let's imagine various search queries in which the fastest should win.

As an example, we should model the process of establishing a connection to the mobile network in the variants 5G, 4G, 3G, and WiFi. Bring the following program to life:

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

BONUS: What happens if an exception is thrown in one of the methods?

Exercise 5 – Scoped Values as an Alternative to `ThreadLocal`

In this exercise, a program that is traditionally implemented using parameter passing is to be converted to scoped values. This involves simulating request processing with a multi-layered call hierarchy and simplified controller and service classes. The starting point is the `ScopedValuesExample` class, which defines two scoped values to provide information about the logged-in user and the request time. At present, these are currently unused and the information is passed as a parameter:

```
public static void main(String[] args) throws Exception
{
    // Simulate 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);
    }
}
```

Your task is to convert the application to Scoped Values, fill them with appropriate values, and implement their propagation in the call chain of ScopedValuesExample via Controller and Service. The information should then be accessed in both classes without having to be passed as parameters in the method calls.

Exercise 6 – Getting to know and experimenting with the Vector API

The Vector API allows you to perform SIMD calculations. In this exercise, an implementation given as a scalar calculation is to be converted to the Vector API.

Consider the following scalar calculation of the formula $a_i * b_i + a_i * b_i – (a_i + b_i)$ with this implementation:

```
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;
}
```

Convert this into suitable calls using the Vector API. Experiment a little:

1) What happens if you don't map the loop correctly with respect to all passes?
2) How can you simplify the formula? Can you benefit from a scalar multiplication, i.e. with a fixed value?

Exercise 7 – Use appropriate Gatherer to collect coordinate data and find temperature jumps

Determine the appropriate 3D points from a stream of x, y, z coordinates:

```
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 */;
```

This should result in the following:

```
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: Provide a suitable constructor to ease construction of points.

Based on a series of temperatures, find those where the temperature values differ over 20 degrees:

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

This should result in the following:

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

Exercise 8 – Use module imports to get rid of a bunch of single imports

Imagine a program that imports a lot of things from the jdk, like the following:

```
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;
```

Simplify these individual import statements by using appropriate module imports and consider possible ambiguities.

Exercise 9 – Use module imports to get rid of a bunch of single imports

In Java, before version 23, it was not possible to switch over primitive values. To benefit from pattern matching with guards, you were forced to add a boxing assignment like this:

```
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);
}
```

Your task is to simply this switch using Primitive Type Patterns.