Best of Modern Java 21 – 25 My Favorites

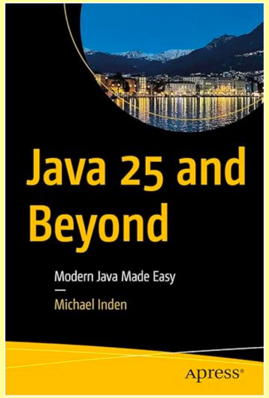
https://github.com/Michaeli71/Best-Of-Modern-Java-21-25-My-Favorite-Features

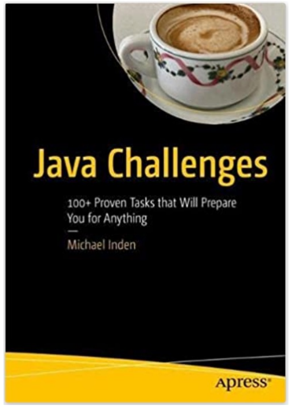
Michael Inden
Freelance consultant, author and trainer

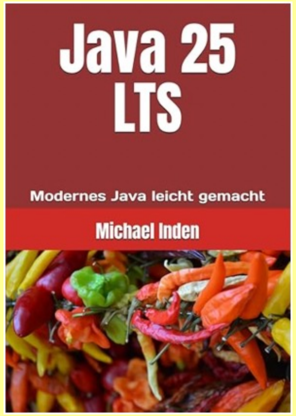
Speaker Intro

michael_inden@hotmail.com

- **Michael Inden**, Year of Birth 1971
- Diploma Computer Science, C.v.O. Uni Oldenburg
- ~8 ¼ Years **SSE** at Heidelberger Druckmaschinen AG in Kiel
- ~6 ¾ Years **TPL, SA** at IVU Traffic Technologies AG in Aachen
- ~4 ¼ Years **LSA / Trainer** at Zühlke Engineering AG in Zurich
- ~3 Years **TL / CTO** at Direct Mail Informatics / ASMIQ in Zurich
- ~3 ½ Years **Head of Development at Adcubum in Zurich**
- Independent **Consultant, Conference Speaker and Trainer**
- Author @ dpunkt.verlag, OßReilly and APress







5. Auflage
Michael Inden
Der Weg zum Java-Profi
Konzepte und Techniken für die professionelle Java-Entwicklung
dpunkt.verlag

https://github.com/Michaeli71/Best-Of-Modern-Java-21-25-My-Favorite-Features

Agenda

All Time Favorites + My Top 10 from Java 21 LTS to Java 25 LTS (in chronological order)

All Time Favorites:

- Switch Expressions / Text Blocks / Records / Pattern Matching bei instanceof (Java 17)

My top 10 from Java 21 LTS to Java 25 LTS:

1. Record Patterns (Java 21)
2. Pattern Matching for switch (Java 21)
3. Virtual Threads (Java 21)
4. *Structured Concurrency (Preview) (Java 21 & 25)*
5. Unnamed Variables & Patterns (Java 22)
6. Launch Multi-File Source-Code Programs (Java 22)
7. Markdown Comments (Java 23)
8. Stream Gatherers (Java 24)
9. Flexible Constructor Bodies (Java 25)
10. Compact Source Files and Instance Main Methods (Java 25)

All Time Favorites

Switch Expressions
Text Blocks
Records
Pattern Matching bei `instanceof`

Switch Expressions

- **Mapping of weekdays to their length … elegantly with modern Java:**

Return-Value

```
DayOfWeek day = DayOfWeek.FRIDAY;
int numOfLetters = switch (day)
{
    case MONDAY, FRIDAY, SUNDAY -> 6;
    case TUESDAY                -> 7;
    case THURSDAY, SATURDAY     -> 8;
    case WEDNESDAY              -> 9;
};
```

- **More elegance using case:**
  - Besides the obvious arrow instead of the colon also several values allowed
  - No break necessary, no fall-through either
  - `switch` can now return a value, avoids artificial auxiliary variables

Switch Expressions

- **Mapping months to their names... elegantly with modern Java:**

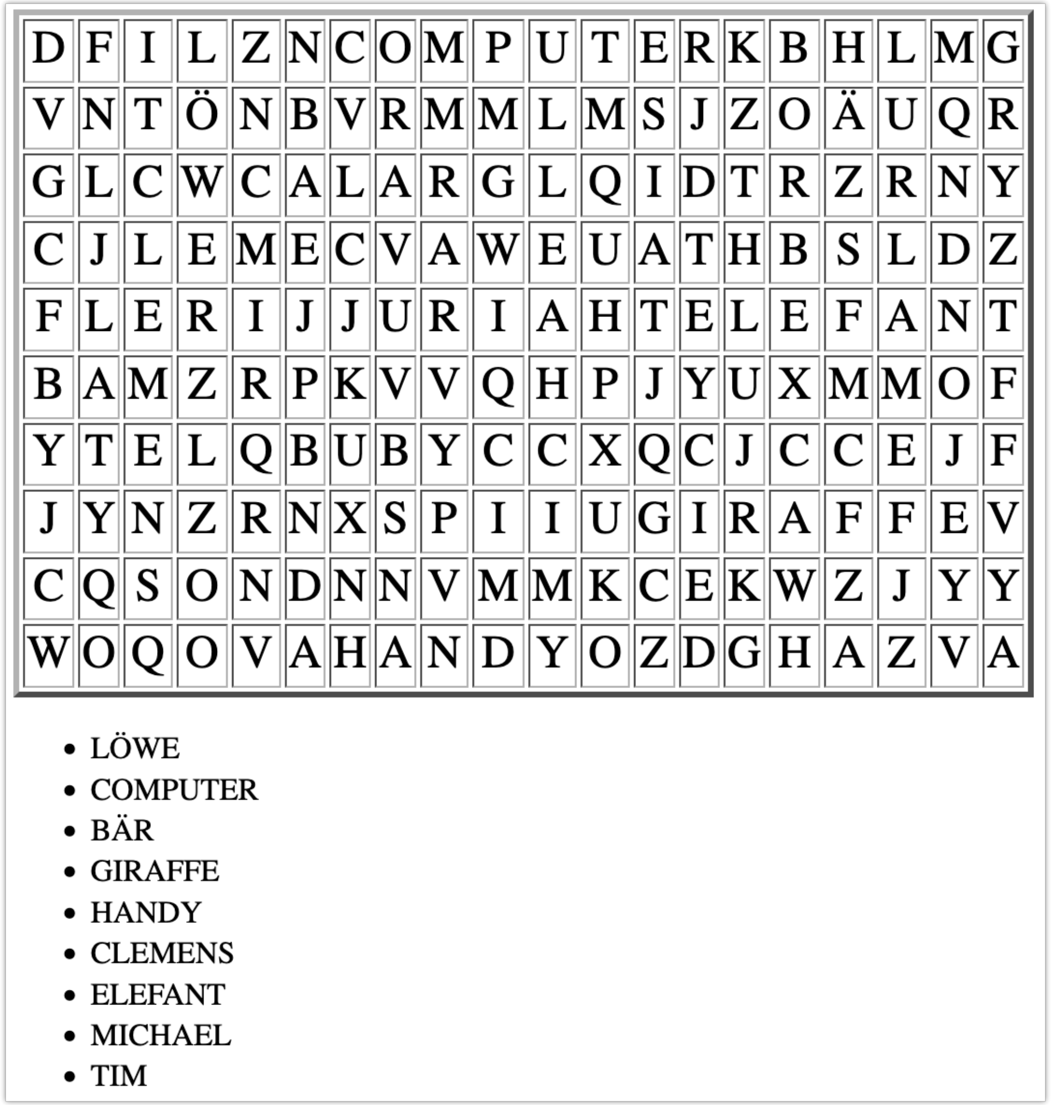
```
static String monthToName(final Month month)
{
    return switch (month)
    {
        case JANUARY -> "January";
        default -> "N/A";   // here is NO Fall Through
        case FEBRUARY -> "February";
        case MARCH -> "March";
        case JULY -> "JULY";
    };
}
```

Text Blocks

```
String jsonObj = """
                 {
                     "name": "Mike",
                     "birthday": "1971-02-07",
                     "comment": "Text blocks are nice!"
                 }
                 """;
```

Text Blocks

```
public String exportAsHtml()
{
    String result = """
           <html>
               <head>
                   <style>
                       td {
                           font-size: 18pt;
                       }
                   </style>
               </head>
            <body>
            """;

    result += createTable();
    result += createWordList();
    result += """
                  </body>
              </html>
              """;

    return result;
}
```



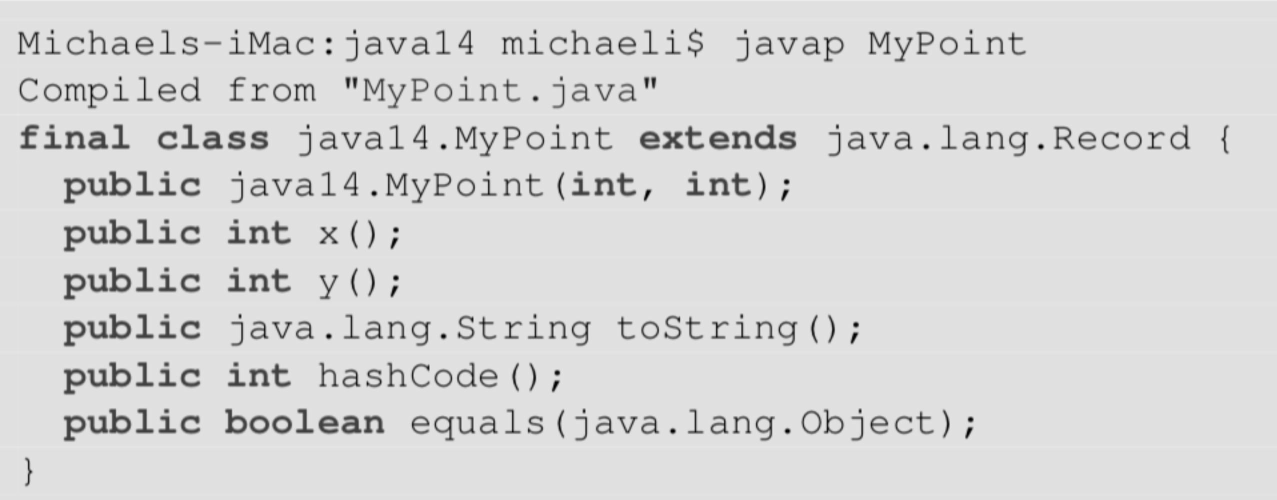
D	F	I	L	Z	N	C	O	M	P	U	T	E	R	K	B	H	L	M	G
V	N	T	Ö	N	B	V	R	M	M	L	M	S	J	Z	O	Ä	U	Q	R
G	L	C	W	C	A	L	A	R	G	L	Q	I	D	T	R	Z	R	N	Y
C	J	L	E	M	E	C	V	A	W	E	U	A	T	H	B	S	L	D	Z
F	L	E	R	I	J	J	U	R	I	A	H	T	E	L	E	F	A	N	T
B	A	M	Z	R	P	K	V	V	Q	H	P	J	Y	U	X	M	M	O	F
Y	T	E	L	Q	B	U	B	Y	C	C	X	Q	C	J	C	C	E	J	F
J	Y	N	Z	R	N	X	S	P	I	I	U	G	I	R	A	F	F	E	V
C	Q	S	O	N	D	N	N	V	M	M	K	C	E	K	W	Z	J	Y	Y
W	O	Q	O	V	A	H	A	N	D	Y	O	Z	D	G	H	A	Z	V	A

- LÖWE
- COMPUTER
- BÄR
- GIRAFFE
- HANDY
- CLEMENS
- ELEFANT
- MICHAEL
- TIM

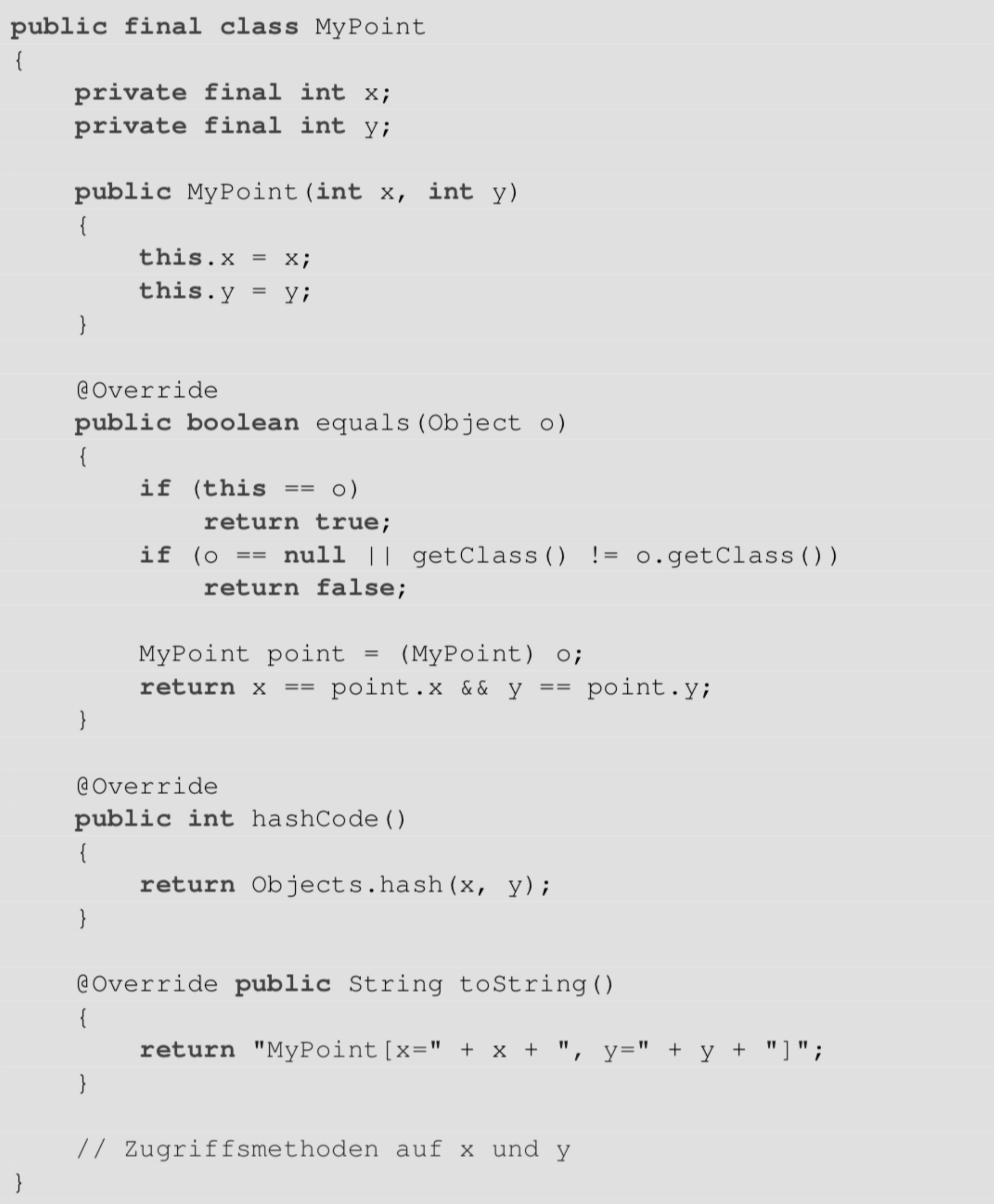
Enhancement Record

```java
record MyPoint(int x, int y) { }
```

What you get



```
Michaels-iMac:java14 michaeli$ javap MyPoint
Compiled from "MyPoint.java"
final class java14.MyPoint extends java.lang.Record {
  public java14.MyPoint(int, int);
  public int x();
  public int y();
  public java.lang.String toString();
  public int hashCode();
  public boolean equals(java.lang.Object);
}
```



```java
public final class MyPoint
{
    private final int x;
    private final int y;

    public MyPoint(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    @Override
    public boolean equals(Object o)
    {
        if (this == o)
            return true;
        if (o == null || getClass() != o.getClass())
            return false;

        MyPoint point = (MyPoint) o;
        return x == point.x && y == point.y;
    }

    @Override
    public int hashCode()
    {
        return Objects.hash(x, y);
    }

    @Override public String toString()
    {
        return "MyPoint[x=" + x + ", y=" + y + "]";
    }

    // Zugriffsmethoden auf x und y
}
```

Records for Complex Return Types or Parameters

```java
record IntStringReturnValue(int code, String info) { }
record IntListReturnValue(int code, List<String> values) { }

record ReturnTuple(String first, String last, int amount) { }
record CompoundKey(String name, int age) { }


IntStringReturnValue calculateTheAnswer()
{
    // Some complex stuff here
    return new IntStringReturnValue(42, "the answer");
}

IntListReturnValue calculate(CompoundKey inputKey)
{
    // Some complex stuff here
    return new IntListReturnValue(201,
                                  List.of("This", "is", "a", "complex", "result"));
}
```

Records for modelling Pairs and Tupels

```
record IntIntPair(int first, int second) {};

record StringIntPair(String name, int age) {};

record Pair<T1, T2>(T1 first, T2 second) {};

record Top3Favorites(String top1, String top2, String top3) {};

record CalcResultTuple(int min, int max, double avg, int count) {};
```

- Extremely little effort
- Very practical for Pairs, Tuples etc.
- Records work great with primitive types and generics
- Implementations of accessor methods as well as `equals()` and `hashCode()` automatically and adhering to contracts

Pattern Matching `instanceof`

- **OLD STYLE**

```
final Object obj = new Person("Michael", "Inden");
if (obj instanceof Person)
{
    final Person person = (Person) obj;
    // ... Access to person...
}
```

- **NEW STYLE**

```
if (obj instanceof Person person)
{
    // here is is possible to access variable person directly
}
```

Pattern Matching `instanceof`

```java
if (obj2 instanceof String str2 && str2.length() > 5)
{
    System.out.println(“Length: " + str2.length());
}
```

My Top 10

https://depositphotos.com/photo/top-10-red-grunge-textured-vintage-isolated-stamp-48094793.html

Position 1:
Record Patterns

https://www.freepik.com/premium-vector/realistic-gold-silver-bronze-medal-with-red-ribbon_10027399.htm

JEP 440: Record Patterns

- The basis for this JEP is the pattern matching for `instanceof` from Java 16:

```
record Point(int x, int y) {}

static void printCoordinateInfo(Object obj)
{
    if (obj instanceof Point point)
    {
        int x = point.x();
        int y = point.y();

        System.out.println("x: %d, y: %d, sum: %d".formatted(x, y, x + y));
    }
}
```

- Although this is often already practical, you still have to access the individual components in some cases in a cumbersome way.
- The goal is to be able to decompose records into their components and access them.

JEP 440: Record Patterns

- **Decompose records declaratively into their components and make them accessable**

```
static void printCoordinateInfo(Object obj)
{
    if (obj instanceof Point point)
    {
        int x = point.x();
        int y = point.y();
        System.out.println("x: %d y: %d, sum: %d".formatted(x, y, x + y));
    }
}
```

- **=>**

```
static void printCoordinateInfoNew(Object obj)
{
    if (obj instanceof Point(int x, int y))
    {
        System.out.println("x: %d y: %d, sum: %d".formatted(x, y, x + y));
    }
}
```

JEP 440: Record Patterns

- Record patterns can be nested
- provide a declarative, powerful, and combinable form of data navigation and processing.

```
record Point(int x, int y) {}

enum Color { RED, GREEN, BLUE }
record ColoredPoint(Point point, Color color) {}
record Rectangle(ColoredPoint upperLeft, ColoredPoint lowerRight) {}


static void printColorOfUpperLeftPoint(Rectangle rect)
{
    if (rect instanceof Rectangle(ColoredPoint(Point point, Color color),
                                  ColoredPoint lowerRight))
    {
        System.out.println(color);
    }
}
```

Where can Record Patterns show their strength?

JEP 440: Record Patterns

- **Let's assume the following records as a data model:**

```
record Person(String firstname, String lastname, LocalDate birthday) {
}

record Phone(String areaCode, String number) {
}

record City(String name, String country, String languageCode) {
}

record FlightReservation(Person person,
                         Phone phoneNumber,
                         City origin,
                         City destination) {
}
```

JEP 440: Record Patterns

- **Legacy code contains deeply nested queries like this:**

```java
boolean checkAgeAndDestinationLanguageCorrectOld(Object obj)
{
    if (obj instanceof FlightReservation reservation)
    {
        if (reservation.person() != null)
        {
            Person person = reservation.person();
            LocalDate birthday = person.birthday();

            if (reservation.destination() != null) {
                City destination = reservation.destination();
                String languageCode = destination.languageCode();

                if (birthday != null && languageCode != null) {
                    long years = ChronoUnit.YEARS.between(birthday, LocalDate.now());
                    return years >= 18 && List.of("EN", "DE", "FR").contains(languageCode);
                }
            }
        }
    }
    return false;
}
```

JEP 440: Record Patterns

- **Nested record patterns allow a more elegant and much more understandably way**
- **Checking with `instanceof` automatically fails if one of the record components is `null`, i.e. here `Person` or `City` (destination).**

```
boolean checkAgeAndDestinationLanguageCorrectNew(Object obj)
{
    if (obj instanceof FlightReservation(
                Person(String firstname, String lastname, LocalDate birthday),
                Phone phoneNumber, City origin,
                City(String name, String country, String languageCode)))
    {
        if (birthday != null && languageCode != null)
        {
            long years = ChronoUnit.YEARS.between(birthday, LocalDate.now());
            return years >= 18 &&
                   List.of("EN", "DE", "FR").contains(languageCode);
        }
    }
    return false;
}
```

JEP 440: Record Patterns

```java
boolean checkAgeAndDestinationLanguageCorrectNew(Object obj)
{
    if (obj instanceof FlightReservation(
            Person(String firstname, String lastname, LocalDate birthday),
            Phone phoneNumber, City from,
            City(String name, String country, String languageCode)))
    {
        if (birthday != null && languageCode != null)
        {
            long years = ChronoUnit.YEARS.between(birthday, LocalDate.now());
            return years >= 18 &&
                   List.of("EN", "DE", "FR").contains(languageCode);
        }
    }
    return false;
}
```

- **Only the attributes are not protected in this way and may need to be checked for null.**
- **However, if you get into the good habit of avoiding** `null` **as a value of parameters in calls, you can even do without it.**

Position 2:
Pattern Matching and `switch`

https://www.freepik.com/premium-vector/realistic-gold-silver-bronze-medal-with-red-ribbon_10027399.htm

JEP 441: Pattern Matching for `switch`: Dominance check

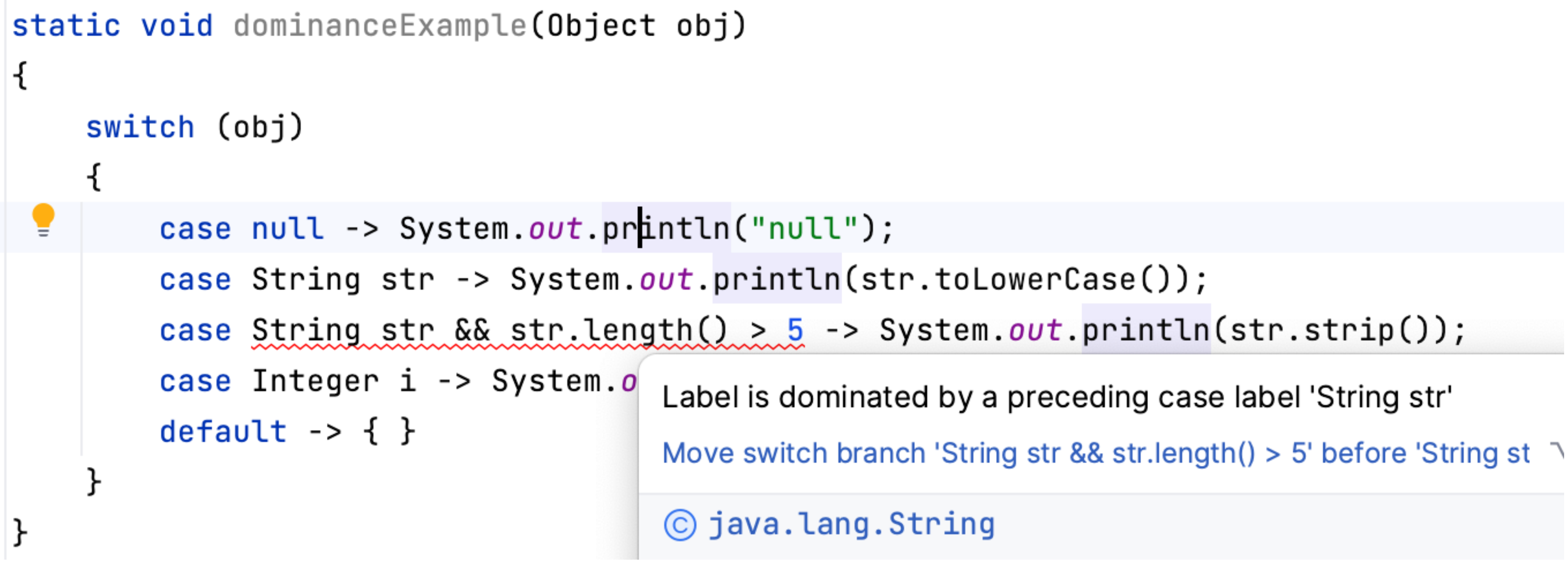
- **Problem area: Multiple patterns can match on one input.**

```
public static void main(String[] args) {
    multiMatch("Python");
    multiMatch(null);
}

static void multiMatch(Object obj) {
    switch (obj) {
       case null -> System.out.println("null");
       case String s && s.length() > 5 -> System.out.println(s.toUpperCase());
       case String s                   -> System.out.println(s.toLowerCase());
       case Integer i                  -> System.out.println(i * i);
       default -> {}
    }
}
```

- **The one that fits "most generally" is called the dominant pattern.**
- **In the example, the shorter pattern `String s` dominates the longer one specified before it.**

JEP 441: Pattern Matching for `switch`: Dominance check

- The whole thing becomes problematic when the order of the patterns is reversed:



```
static void dominanceExample(Object obj)
{
    switch (obj)
    {
        case null -> System.out.println("null");
        case String str -> System.out.println(str.toLowerCase());
        case String str && str.length() > 5 -> System.out.println(str.strip());
        case Integer i -> System.o
        default -> { }
    }
}
```

Label is dominated by a preceding case label 'String str'

Move switch branch 'String str && str.length() > 5' before 'String st

ⓒ java.lang.String

- The dominance check uncovers the problem and leads to a compile error since Java 18 and Java 17.0.6, because the second `case` is de facto unreachable code.
- With the first Java 17 versions this was not detected as error!

SwitchDominanceExample.java

JEP 441: Pattern Matching for switch with Record Patterns

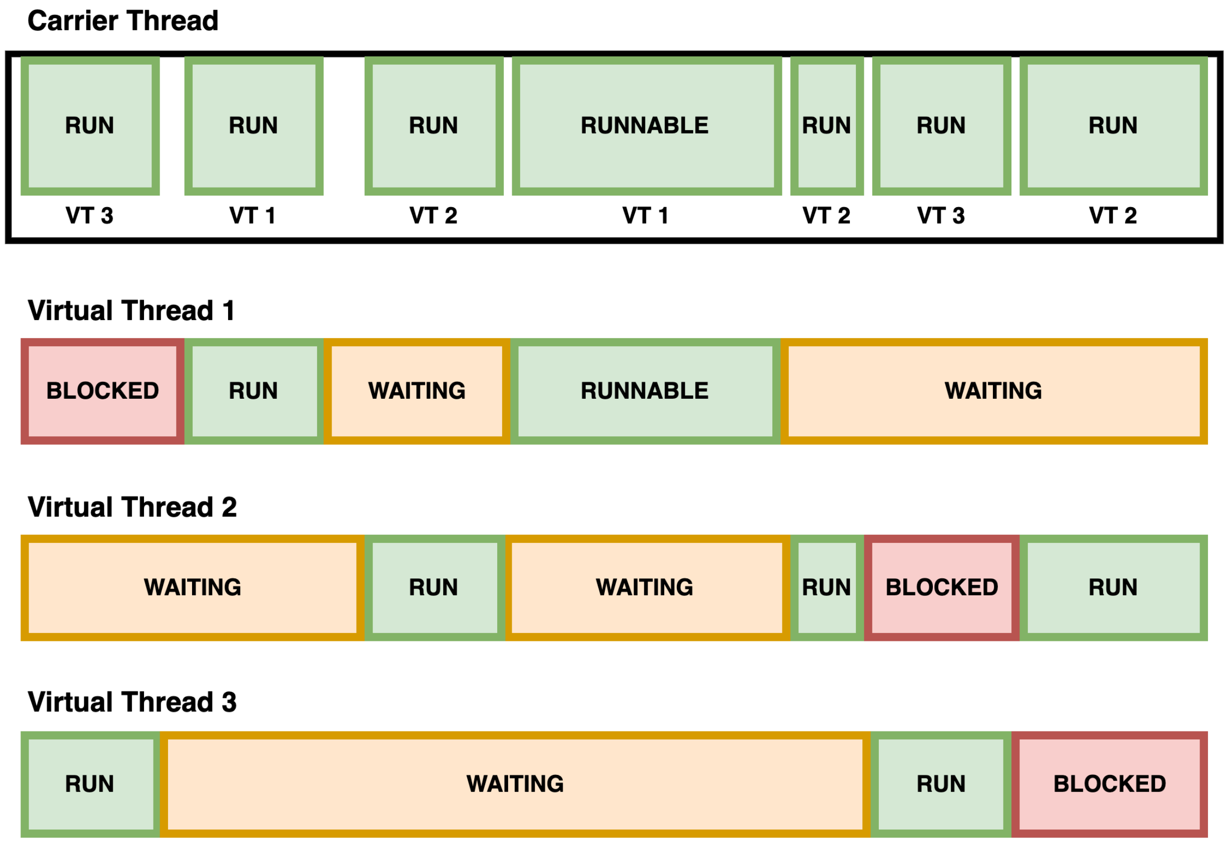
```java
record Pos3D(int x, int y, int z) { }

enum RgbColor {RED, GREEN, BLUE}


static void recordPatternsAndMatching(Object obj) {
    switch (obj) {
        case RgbColor color when color == RgbColor.RED ->
             System.out.println("RED WARNING");

        case Pos3D pos when pos.z() == 0 ->
             System.out.println("Record: " + pos);

        case Pos3D(int x, int y, int z) when y > 0 ->
             System.out.println("Record decomposed: " + x + ", " + y + ", " + z);

        default -> System.out.println("Something else");
    }
}
```

Position 3: Virtual Threads

https://www.freepik.com/premium-vector/realistic-gold-silver-bronze-medal-with-red-ribbon_10027399.htm

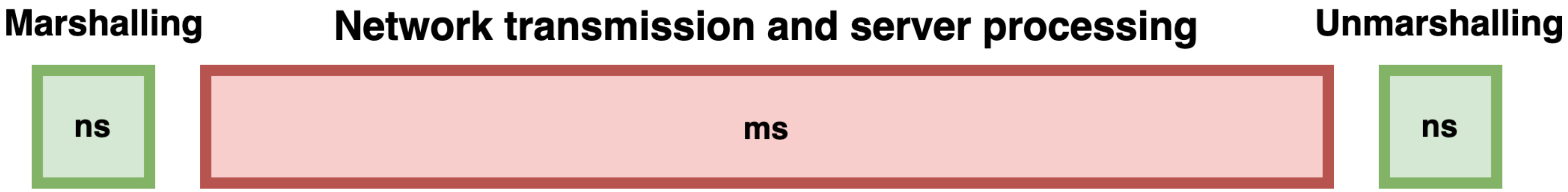
JEP 444: Virtual Threads

- Concept of lightweight virtual threads
- Virtual threads "feel" like normal threads, but are not mapped 1:1 to operating system threads.



JEP 444: Virtual Threads

- What is the problem with blocking I/O? => miserable server utilization



```
var request = buildContractRequest(contractId);
var response = contractServer.retrieveDetails(request);
var contractDetails = convertFromJson(response.getPayload());
```

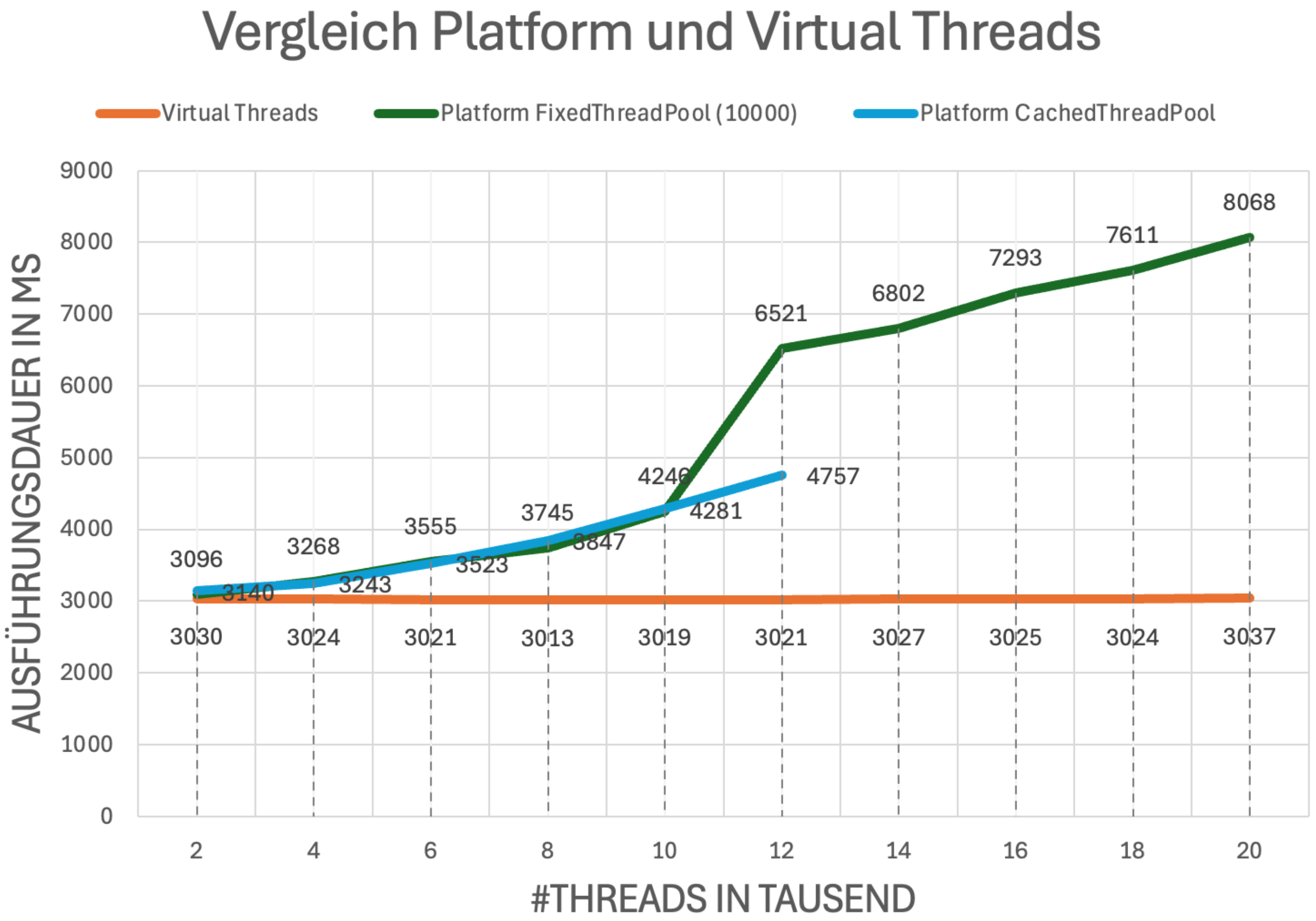
- Virtual threads permit to work with a separate thread per request.
- This is helpful because many client requests usually perform blocking I/O such as retrieving resources.

JEP 444: Platform Threads => Virtual Threads

```java
public static void main(String[] args)
{
    System.out.println("Start");

    try (var executor = Executors.newVirtualThreadPerTaskExecutor())
    {
        for (int  i = 0; i < 10_000_000; i++)
        {
            final int pos = i;
            executor.submit(() -> {
                Thread.sleep(Duration.ofSeconds(5));
                return pos;
            });
        }
    }
    // executor.close() is called implicitly,
    // and waits until all tasks are completed
    System.out.println("End");
}
```

JEP 444: Virtual Threads

- **Performance comparison in increments of thousands for platform and virtual threads**
- **The threads wait a few seconds each to simulate access to an external interface. The total time is measured at the end.**



Position 4:
Structured Concurrency (Preview)

Sigificant changes in Java 25

https://www.freepik.com/premium-vector/realistic-gold-silver-bronze-medal-with-red-ribbon_10027399.htm

JEP 505: Structured Concurrency

- Structured Concurrency as a simplification of multithreading.

- Different tasks executed in multiple threads are considered as a single unit.
- This improves reliability, reduces the risk for errors and simplifies their handling.

- Let's consider determining a user and their orders based on a user ID:

```
static Response handleSynchronously(Long userId) throws InterruptedException
{
    var user = findUser(userId);
    var orders = fetchOrders(userId);

    return new Response(user, orders);
}
```

- Both actions could run in parallel.
- If one of both throws an exception, complete execution is aborted

Try 1: Classical ExecutorService

```
static Response handleOldStyle(Long userId) throws ExecutionException,
                                                   InterruptedException
{
    var executorService = Executors.newCachedThreadPool();

    var userFuture = executorService.submit(() -> findUser(userId));
    var ordersFuture = executorService.submit(() -> fetchOrders(userId));

    var user = userFuture.get();      // Join findUser
    var orders = ordersFuture.get();  // Join fetchOrders

    return new Response(user, orders):
}
```

- **Because the subtasks are executed in parallel, they can succeed or fail independently.**
- **Error handling can become quite complicated.**
- **Often, for example, one does not want the second `get()` to be called if an exception has already occurred during the processing of the `findUser()` method.**

Remedy: Structured Concurrency

- **Implementation of Structurd Concurrency with class `StructuredTaskScope`:**

```
static Response handleJava25(Long userId) throws InterruptedException
{
    //var joiner = StructuredTaskScope.Joiner.awaitAllSuccessfulOrThrow();
    try (var scope = StructuredTaskScope.open())
    {
        var userSubtask = scope.fork(() -> findUser(userId));
        var orderSubtask = scope.fork(() -> fetchOrder(userId));

        scope.join();          // Join both forks

        // Here, both forks have succeeded, so compose their results
        return new Response(userSubtask.get(), orderSubtask.get());
    }
}
```

- **With structured concurrency, one splits off competing subtasks with `fork()`.**
- **The results are collected with a blocking call to `join()`, which waits until all subtasks are processed or an error occurred.**

JEP 453: Structured Concurrency

The `StructuredTaskScope` class offers two main joining strategies:

- `awaitAllSuccessfulOrThrow()` – catches the first exception and terminates the `StructuredTaskScope`. This class is intended when results of all subtasks are needed ("invoke all"); if one subtask fails, the results of the other uncompleted subtasks are no longer needed.

- `anySuccessfulResultOrThrow()` – determines the first incoming result and then terminates the `StructuredTaskScope`. This helps when the result of an arbitrary subtask is already sufficient ("invoke any") and it is not necessary to wait for the results of other uncompleted tasks.

JEP 505: Structured Concurrency

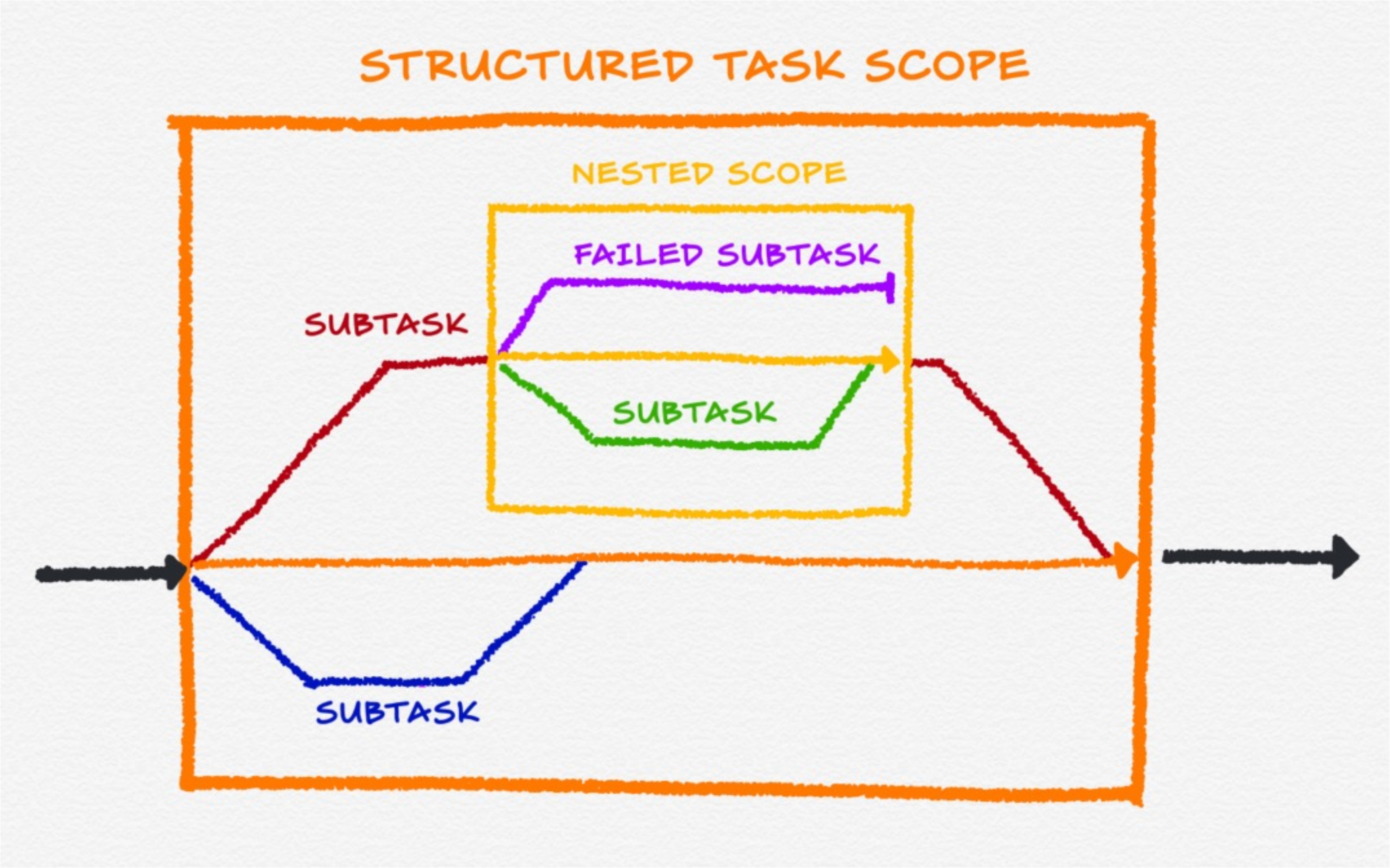
- **Implementation of Structurd Concurrency with class `StructuredTaskScope`:**

```
var joiner =
    StructuredTaskScope.Joiner.<NetworkConnection>anySuccessfulResultOrThrow();
try (var scope = StructuredTaskScope.open(joiner))
{
    var result1 = scope.fork(() -> tryToGetWifi());
    var result2 = scope.fork(() -> tryToGet5g());
    var result3 = scope.fork(() -> tryToGet4g());
    var result4 = scope.fork(() -> tryToGet3g());

    NetworkConnection result = scope.join();

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

- **Konkurrierende Teilaufgaben mit `fork()` abspalten und mit blockierendem Aufruf von `join()` das zuerst vorliegende Ergebnis einsammeln**
- **`join()` wartet, bis eine Teilaufgabe erfolgreich ist**
- **`state()` liefert `SUCCESS` (erster), `UNAVAILABLE` (andere) oder `FAILED` (Exception)**

JEP 505: Structured Concurrency

- **Structured concurrency can also be nested:**



https://belief-driven-design.com/looking-at-java-21-structured-concurrency-39a81/

JEP 505: Structured Concurrency

```
Proposal performCityTripProposals(String city,
                                  LocalDate startD                              n
{
    try (var scope = StructuredTaskScope.open())
    {
        var hotelSubtask = scope.fork(() -> findHotels(city, startDate));
        var carRentalSubtask = scope.fork(() -> fetchCarRentals(city,
                                                                startDate));

        scope.join();

        // Hier waren beide Zweige erfolgreich, also Ergebnisse zusammenfügen
        return new Proposal(hotelSubtask.get(), carRentalSubtask.get());
    }
}
```

In each method, new, nested `StructuredTaskScope` objects are created with their own `Joiners`.

JEP 505: Structured Concurrency

Inner StructuredTaskScope used a diffentert Joiner

- **Find recommendations for a hotel**

```
private static String findHotels(String city, LocalDa[illegible]te) throws
InterruptedException
{
    var anySuccessful = StructuredTaskScope.Joiner.<String>anySuccessfulResultOrThrow();
    try (var scope = StructuredTaskScope.open(anySuccessful)) {

        var proposal1 = scope.fork(() -> findProposalTrivago(city, startDate));
        var proposal2 = scope.fork(() -> findProposalBooking(city, startDate));
        var proposal3 = scope.fork(() -> findProposalCheck24(city, startDate));

        // Hier kommt vielleicht der Wunsch nach FirstNSuccessful auf,
        // falls man doch mehrere Vorschläge möchte
        return scope.join();
    }
}
```

- **Custom joiners are fairly easy to implement and use:**

```
var twoSuccessful = new OwnJoinersExample.FirstNSuccessful<String>(2);
```

Position 5:
Unnamed Variables and Patterns

https://www.freepik.com/premium-vector/realistic-gold-silver-bronze-medal-with-red-ribbon_10027399.htm

JEP 456: Unnamed Patterns and Variables

- **What do you observe using record patterns?**

```
Point point = new Point(3, 4);
var coloredPoint = new ColoredPoint(point, Color.GREEN);

if (coloredPoint instanceof ColoredPoint(Point point, Color color))
{
    System.out.println("x = " + point.x());
}

if (coloredPoint instanceof ColoredPoint(Point(int x, int y),
                                         Color color))

{
    System.out.println("x = " + x);
}
```

- **Only a few parts are really of interest**

JEP 456: Unnamed Patterns and Variables

- **And what about similar situations in «normal» Java code:**

```
BiFunction<String, String, String> doubleFirst =
                                   (String str1, String str2) -> str1.repeat(2);


try
{
    Files.writeString(Path.of("UnnamedVars.txt"), "Underscore");
}
catch (IOException ex)
{
    // just some logging
}
```

- **Some variables are unused**

JEP 456: Unnamed Patterns and Variables

- **JEP 456 finalizes its predecessor JEP 443 and allows variables or parts within record patterns to be marked as unused and unusable with an underscore (_).**

- **As a reminder, there are three variants:The following three variations exist:**

1. **unnamed variable** – allows to use _ for naming or marking unused variables

2. **unnamed pattern variable** – allows the identifier that would normally follow the type (or var) in a record pattern to be omitted

3. **unnamed pattern** – allows to omit the type and name of a record pattern component completely (and replace with single _)

JEP 456: Unnamed Patterns and Variables

- **Unnamed variable**

```
BiFunction<String, String, String> doubleFirst =
                                   (String str1, String _) -> str1.repeat(2);

interface IntTriFunction
{
    int apply(int x, int y, int z);
}

IntTriFunction addFirstTwo = (int x, int y, int _) -> x + y;

IntTriFunction doubleSecond = (int _, int y, int _) -> y * 2;
```

- **Interestingly, multiple unnamed variables can also be used in the same scope, which (besides simple lambdas) is of interest especially for record patterns and in `switch`.**

JEP 456: Unnamed Patterns and Variables

- **Unnamed pattern variable**

```
if (green_p3_4 instanceof ColoredPoint(Point point, Color color))
{
    System.out.println("x = " + point.x());
}
```

- **=>**

```
if (green_p3_4 instanceof ColoredPoint(Point point, Color _))
{
    System.out.println("x = " + point.x());
}
```

- **Same applies for `case ColoredPoint(Point point, Color _)`**

JEP 456: Unnamed Patterns and Variables

- **Unnamed pattern**

```
if (green_p3_4 instanceof ColoredPoint(Point(int x, int y), Color color))
{
    System.out.println("x = " + x);
}
```

- **=>**

```
if (cp instanceof ColoredPoint(Point(int x, _), _))
{
    System.out.println("x = " + x);
}
```

```
instanceof _
instanceof _(int x, int y)
```

- **Same applies for `case`.**

Unnamed Variables & Patterns

```
static boolean checkFirstNameTravellingTimeAndZipCode(Object obj)
{
    if (obj instanceof Journey(
          Person(var firstname, _, _),
          TravelInfo(_, var maxTravellingTime), _,
          City(var zipCode, _))) {

       if (firstname != null && maxTravellingTime != null && zipCode != null) {

          return firstname.length() > 2 && maxTravellingTime.toHours() < 7 &&
                 zipCode >= 8000 && zipCode < 8100;
       }
    }
    return false;
}
```

Attention:
Postal codes are numeric here, as in Switzerland.

In Germany, special features such as leading zeros must be taken into account so that simple int is not suitable

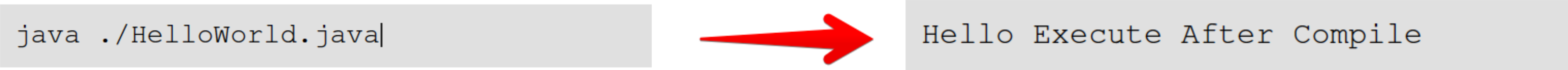
Position 6:
Launch Multi-File Source-Code Programs

https://www.freepik.com/premium-vector/realistic-gold-silver-bronze-medal-with-red-ribbon_10027399.htm

JEP 458: Launch Multi-File Source-Code Programs: Direct Compilation

- Allows Java applications consisting of only one file to be compiled and executed directly in one go.
- Saves work and requires no knowledge of bytecode or .class files
- Particularly useful for executing smaller Java files as scripts and for getting started with Java

```java
package direct.compilation;

public class HelloWorld
{
    public static void main(String... args)
    {
        System.out.println("Hello Execute After Compile");
    }
}
```



```
java ./HelloWorld.java
```

```
Hello Execute After Compile
```

- However, up to and including Java 21 LTS, only for a single Java file!

JEP 458: Launch Multi-File Source-Code Programs

```java
package jep458_Launch_MultiFile_SourceCode_Programs;

public class MainApp
{
    public static void main(final String[] args) {
        var result = Helper.performCalculation();
        System.out.println(result);
    }
}
```

```java
package jep458_Launch_MultiFile_SourceCode_Programs;

class Helper
{
    public static String performCalculation() {
        return "Heavy, long running calculation!";
    }
}
```

```
$ java MainApp.java
Heavy, long running calculation!
```

JEP 458: Launch Multi-File Source-Code Programs

```java
package jep458_Launch_MultiFile_SourceCode_Programs;

public class MainAppV2
{
    public static void main(final String[] args) {
        var result = StringHelper.mark(Helper.performCalculation());
        System.out.println(result);
    }
}
```

```java
package jep458_Launch_MultiFile_SourceCode_Programs;

class StringHelper
{
    public static String mark(String input) {
        return ">>" + input + "<<";
    }
}
```

```
$ java MainAppV2.java
>>Heavy, long running calculation!<<
```

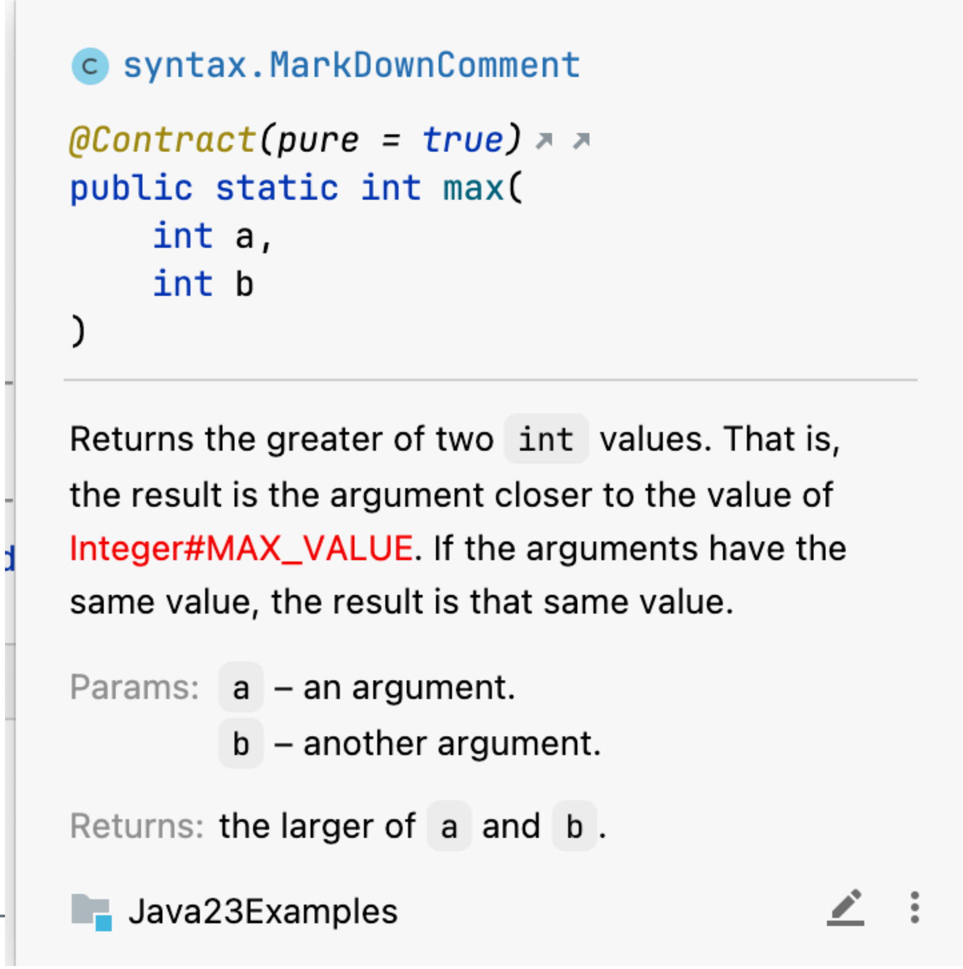
Position 7:
Markdown Comments

https://www.freepik.com/premium-vector/realistic-gold-silver-bronze-medal-with-red-ribbon_10027399.htm

JEP 467: Markdown Documentation Comments

- Document your code using mark down
- It is also displayed in IntelliJ directly on the commented program element (class / method)

```
/// Returns the greater of two `int` values. That is, the
/// result is the argument closer to the value of
/// [Integer#MAX_VALUE]. If the arguments have the same
/// value, the result is that same value.
///
/// @param   a   an argument.
/// @param   b   another argument.
/// @return  the larger of `a` and `b`.
public static int max(int a, int b) {
    return (a >= b) ? a : b;
}
```



```
syntax.MarkDownComment

@Contract(pure = true)
public static int max(
    int a,
    int b
)
```

Returns the greater of two `int` values. That is, the result is the argument closer to the value of Integer#MAX_VALUE. If the arguments have the same value, the result is that same value.

Params: `a` – an argument.
`b` – another argument.

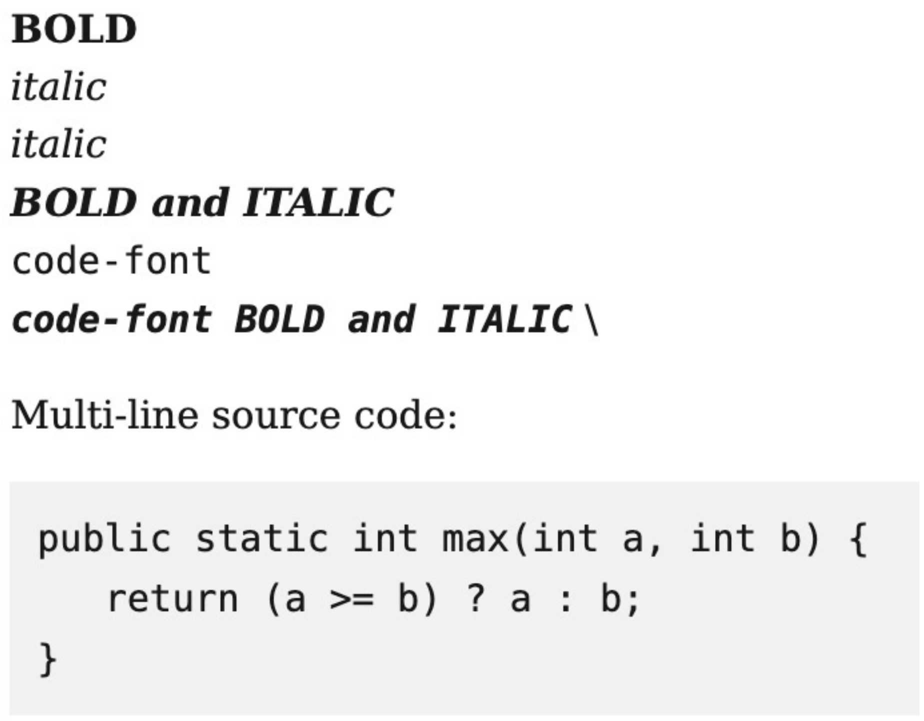
Returns: the larger of `a` and `b`.

Java23Examples

JEP 467: Markdown Documentation Comments -- Formatierung

- **Text passages should be emphasized from time to time**
  - *italics* (*...* or _..._) or **bold** (**...**)
  - Change font by backtick ('...') to `typewriter` font. Bold and/or italics are also possible.
  - Integrate multi-line source code snippets with ('''...''') into a comment.

~~~~
/// **BOLD**   \
/// *italic*   \
/// _italic_   \
/// _** BOLD and ITALIC **_\
/// `code-font` \
/// _**`code-font BOLD and ITALIC`**_ \
///
/// Multi-line source-code:
/// ```
/// public static int max(int a, int b) {
///    return (a >= b) ? a : b;
/// }
/// ```
~~~~



BOLD

italic

italic

BOLD and ITALIC

`code-font`

`code-font BOLD and ITALIC`\

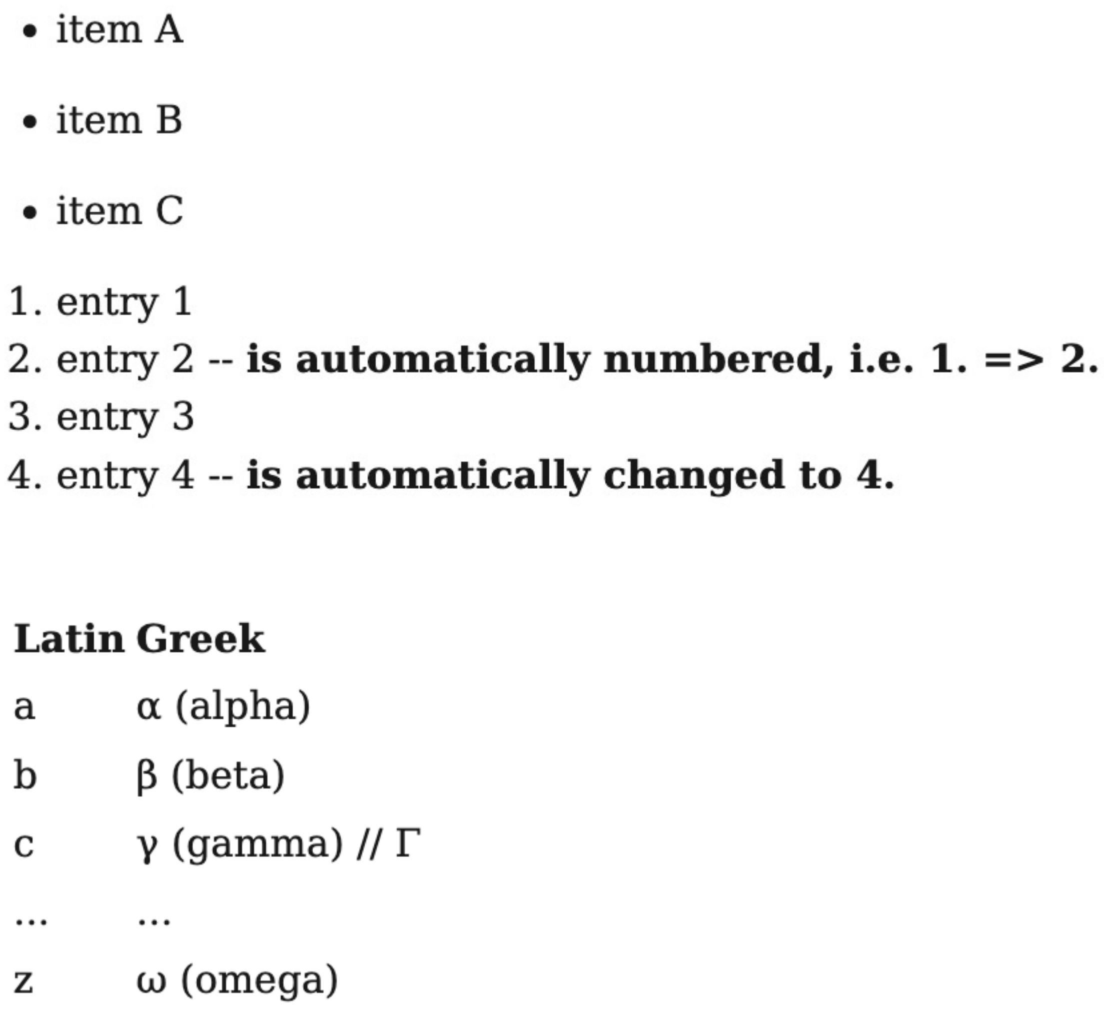
Multi-line source code:

```
public static int max(int a, int b) {
   return (a >= b) ? a : b;
}
```

JEP 467: Markdown Documentation Comments – Lists & Tables

```
/// - item A
/// * item B
/// - item C
///
/// 1st entry 1
/// 1st entry 2 -- **is automatically numbere
/// 1st entry 3
/// 2nd entry 4 -- **is automatically changec
```

```
/// | Latin | Greek |
/// |-------|-------|
/// | a     | &alpha; (alpha) |
/// | b     | &beta; (beta)  |
/// | c     | &gamma; (gamma) // &Gamma;  |
/// | ...   |  ...  |
/// | z     | &omega; (omega) |
```



- item A
- item B
- item C

1. entry 1
2. entry 2 -- **is automatically numbered, i.e. 1. => 2.**
3. entry 3
4. entry 4 -- **is automatically changed to 4.**

Latin	Greek
a	α (alpha)
b	β (beta)
c	γ (gamma) // Γ
...	...
z	ω (omega)

Position 8: Stream Gatherers

https://www.freepik.com/premium-vector/realistic-gold-silver-bronze-medal-with-red-ribbon_10027399.htm

JEP 485: Stream Gatherers

- **Let's assume we want to filter out all duplicates from a stream and specify a criterion for this:**

```
var result = Stream.of("Tim", "Tom", "Jim", "Mike").
                   distinctBy(String::length).     // Hypothetical
                   toList();
```

- **You can solve this conventionally with a trick as follows:**

```
var result = Stream.of("Tim", "Tom", "Jim", "Mike").
                   map(DistinctByLength::new).
                   distinct().
                   map(DistinctByLength::str).
                   toList();
```

JEP 485: Stream Gatherers

- **Another example is grouping a stream's data into sections of a fixed size.**

```
var result = Stream.iterate(0, i -> i + 1).
                    windowFixed(4).         // Hypothetical
                    limit(3).
                    toList();

// result ==> [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]
```

- **Over the years, various intermediate operations such as `distinctBy()` or `windowFixed()` have been proposed as additions to the Stream API.**

- **These are often useful in specific contexts, but they would make the Stream API rather bloated and (further) complicate access to the (already extensive) API.**

JEP 485: Stream Gatherers

- Java 24 now provides a `gather(Gatherer)` method for providing a user-defined intermediate operation, analogous to `collect(Collector)` for terminal operations.

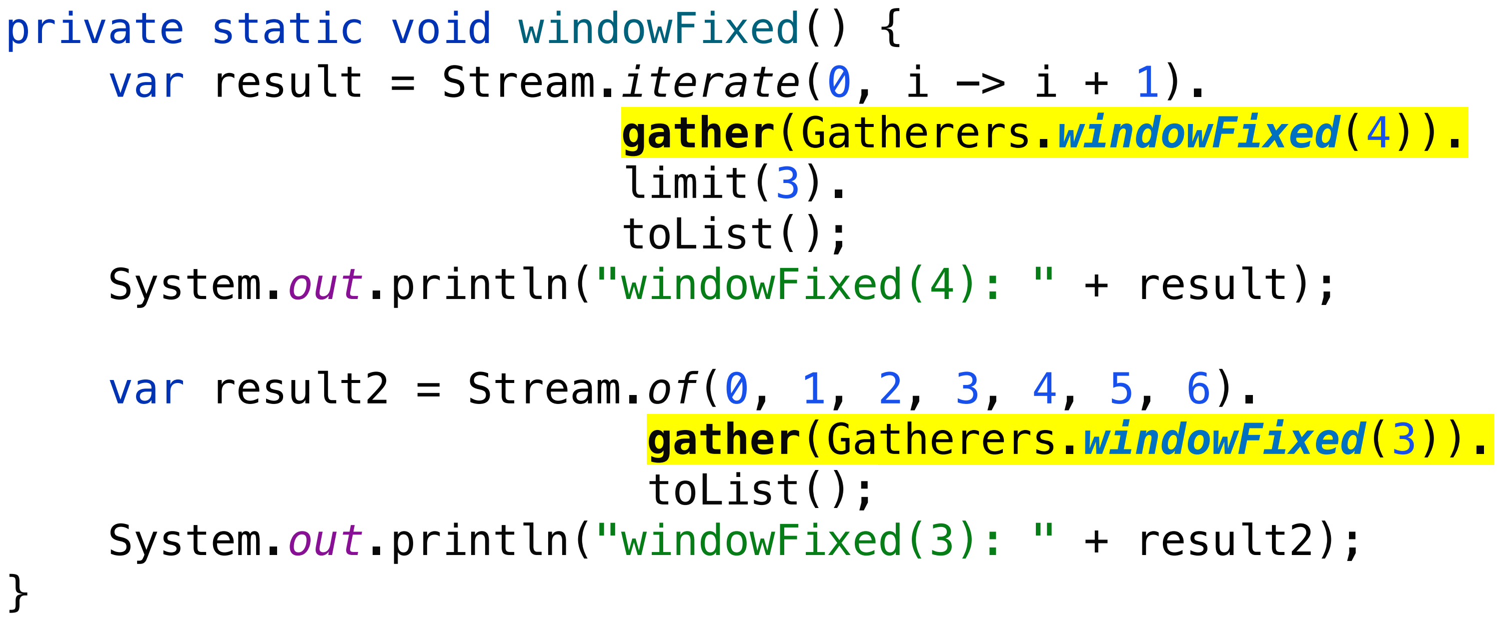
- This is done using the `java.util.stream.Gatherer` interface, which may initially seem a little challenging to implement yourself.

- Conveniently, the `java.util.stream.Gatherers` utility class provides various predefined gatherers such as:

  - `windowFixed()`
  - `windowSliding()`
  - `fold()`
  - `scan()`

JEP 485: Stream Gatherers – `windowFixed()`

- **To divide a stream into smaller components of fixed size without overlapping, `windowFixed()` is used.**



```java
private static void windowFixed() {
    var result = Stream.iterate(0, i -> i + 1).
                       gather(Gatherers.windowFixed(4)).
                       limit(3).
                       toList();
    System.out.println("windowFixed(4): " + result);

    var result2 = Stream.of(0, 1, 2, 3, 4, 5, 6).
                        gather(Gatherers.windowFixed(3)).
                        toList();
    System.out.println("windowFixed(3): " + result2);
}
```

- **In some cases, the dataset does not contain enough elements. This means that the last sub-range simply contains fewer elements.**

```
windowFixed(4): [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]
windowFixed(3): [[0, 1, 2], [3, 4, 5], [6]]
```

JEP 485: Stream Gatherers – `windowSliding()`

- To divide a stream into smaller components of fixed size with overlapping, `windowSliding()` is used:

```java
private static void windowSliding() {
    var result = Stream.iterate(0, i -> i + 1).
                         gather(Gatherers.windowSliding(4)).
                         limit(3).
                         toList();
    System.out.println("windowSliding(4): " + result);

    var result2 = Stream.of(0, 1, 2, 3, 4, 5, 6).
                          gather(Gatherers.windowSliding(3)).
                          toList();
    System.out.println("windowSliding(3): " + result2);
}
```

- In some cases, the dataset does not contain enough elements. This means that the last sub-range simply contains fewer elements (not shown here):

```
windowSliding(4): [[0, 1, 2, 3], [1, 2, 3, 4], [2, 3, 4, 5]]
windowSliding(3): [[0, 1, 2], [1, 2, 3], [2, 3, 4], [3, 4, 5], [4, 5, 6]]
```

JEP 485: Stream Gatherers – fold()

- The fold() method is used to combine the values of a stream. Similar to reduce(), a start value and a calculation rule are specified:

```
private static void foldMult()
{
    var crossMult = Stream.of(10, 20, 30, 40, 50).
                           gather(Gatherers.fold(() -> 1L,
                                  (result, number) -> result * number)).
                           findFirst();
    System.out.println("mult with fold(): " + crossMult);
}
```

- To access the value, the call to findFirst() is used again, which returns an Optional<T>:

```
mult with fold(): Optional[12000000]
```

JEP 485: Stream Gatherers – fold()

- What happens if we also want to execute actions for the value combination and these actions are not defined for the types of the values, in this case int?
- As an example, a numerical value is converted into a string and this is repeated according to the numerical value with repeat():

```java
private static void foldRepeat()
{
    var repeatedNumbers = Stream.of(1, 2, 3, 4, 5, 6, 7).
                                 gather(Gatherers.fold(() -> "",
                                        (result, number) -> result + ("" +
                                        number).repeat(number))).
                                 toList();
    System.out.println("repeat with fold(): " + repeatedNumbers);
}
```

- The output is as follows:

```
repeat with fold(): [1223334444555556666667777777]
```

JEP 485: Stream Gatherers – `scan()`

- If the elements of a stream are to be merged into new combinations so that one element is added at a time, then this is the task of `scan()`.
- `scan()` works in a similar way to `fold()`, however, a new result is produced for each combination of values:

```
private static void scanRepeat()
{
    var repeatedNumbers = Stream.of(1, 2, 3, 4, 5, 6, 7).
                                 gather(Gatherers.scan(() -> "",
                                        (result, number) -> result + ("" +
                                        number).repeat(number))).
                                 toList();
    System.out.println("repeat with scan(): " + repeatedNumbers);
}
```

- The output is as follows:

```
repeat with scan(): [1, 122, 122333, 1223334444, 122333444455555,
122333444455555666666, 1223334444555556666667777777]
```

JEP 485: Stream Gatherers – Practical example: Prefix sums

```
private static void cummulatedExpensesPrefixSum() {
    var expenses = Stream.of(10, 15, 20, 30, 40, 150, 75,
                             10, 20, 30, 70, 50, 120, 100,
                             10, 50, 20, 25, 50, 150, 110).
                          gather(Gatherers.scan(() -> 0L, (result, number) -> result + number)).
                          toList();
    IO.println("cummulatedExpenses: " + expenses);

// Queries take O(1), instead of having to repeatedly sum all values for each query => O(n)
    var week1 = expenses.get(6);
    var week2 = expenses.get(13) - expenses.get(6);
    var week3 = expenses.get(20) - expenses.get(13);
    IO.println("week1: " + week1);
    IO.println("week2: " + week2);
    IO.println("week3: " + week3);
}
```

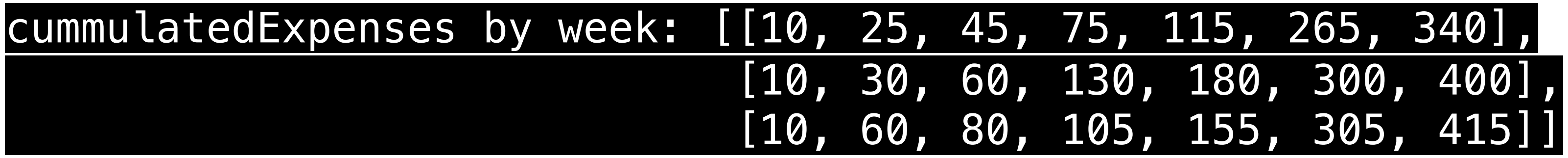
```
cummulatedExpenses: [10, 25, 45, 75, 115, 265, 340, 350, 370, 400, 470, 520, 640, 740,
                     750, 800, 820, 845, 895, 1045, 1155]
week1: 340
week2: 400
week3: 415
```

JEP 485: Stream Gatherers – Grouped prefix sums

- **Task: Group expenses by week and calculate the cumulative total for each week.**
- **To do this, combine the gatherers `windowFixed()` and `scan()`:**

```java
private static void cummulatedExpensesByWeek() {
    var expensesByWeek = Stream.of(10, 15, 20, 30, 40, 150, 75,
                                   10, 20, 30, 70, 50, 120, 100,
                                   10, 50, 20, 25, 50, 150, 110).
                                gather(Gatherers.windowFixed(7)).map(window ->
                                       window.stream().gather(Gatherers.scan(() -> 0,
                                              (result, number) -> result + number)).toList()).
                                toList();
    IO.println("cummulatedExpenses by week: " + expensesByWeek);
}
```

- **Gatherer `windowFixed()` returns a list that can be converted back into a stream.**
- **Output:**



```
cummulatedExpenses by week: [[10, 25, 45, 75, 115, 265, 340],
                             [10, 30, 60, 130, 180, 300, 400],
                             [10, 60, 80, 105, 155, 305, 415]]
```

JEP 485: Stream Gatherers – Speciality `mapConcurrent()`

- **Traditional approach to parallelism:**

```
var result3 = IntStream.rangeClosed(0, 499).
        boxed().
        parallel().
        map(LookupService::lookup).
        toList();
```

- **=> Waiting time of approx. 30 seconds if `lookup()` takes approx. 1 second**

- **Advantages of controlling parallelism, switching to virtual threads reduces waiting time to approx. 2 seconds:**

```
var result4 = IntStream.rangeClosed(0, 499).
        boxed().
        gather(Gatherers.mapConcurrent(250, LookupService::lookup)).
        toList();
```

- **Experimenting with the number of parallel processes: 500 reduced to 1 second**

Position 9:
Flexible Constructor Bodies

https://www.freepik.com/premium-vector/realistic-gold-silver-bronze-medal-with-red-ribbon_10027399.htm

JEP 513: Flexible Constructor Bodies

```java
public class PositiveBigIntegerOld1 extends BaseInteger
{
    public PositiveBigIntegerOld1(long value)
    {
        super(value);                   // Potentially unnecessary work

        if (value <= 0)
            throw new IllegalArgumentException("non-positive value");
    }
}
```

- **If we look at the source code, it doesn't look very elegant. Furthermore, the check only takes place after the base class has been constructed ...**

- **Potentially unnecessary calls and object constructions have already taken place**

- **Espacially older legacy code is often ingloriously characterized by the fact that (too) many actions already take place in the constructor.**

JEP 513: Flexible Constructor Bodies

- **Conventional workaround: static helper method**

```
public class PositiveBigIntegerOld2 extends BaseInteger
{
    public PositiveBigIntegerOld2(final long value)
    {
        super(verifyPositive(value));
    }

    private static long verifyPositive(final long value)
    {
        if (value <= 0)
            throw new IllegalArgumentException("non-positive value");

        return value;
    }
}
```

JEP 513: Flexible Constructor Bodies

- **The argument check is much easier to read and understand if the validation logic takes place directly in the constructor before `super()` is called.**

- **JEP 482 allows the arguments of a constructor to be validated before the constructor of the super class is called:**

```
public class PositiveBigIntegerNew extends BaseInteger
{
    public PositiveBigIntegerNew(final long value) {
        if (value <= 0)
            throw new IllegalArgumentException("non-positive value");

        super(value);
    }
}
```

JEP 513: Flexible Constructor Bodies

- **Sometimes it makes sense to execute actions before calling `this()` to avoid multiple actions, like calls to `split()`, in the following:**

```
record MyPointOld(int x, int y)
{
    public MyPointOld(final String values)
    {
        this(Integer.parseInt(values.split(",")[0].strip()),
             Integer.parseInt(values.split(",")[1].strip()));
    }
}

record MyPoint3dOld(int x, int y, int z)
{
    public MyPoint3dOld(final String values)
    {
        this(Integer.parseInt(values.split(",")[0].strip()),
             Integer.parseInt(values.split(",")[1].strip()),
             Integer.parseInt(values.split(",")[2].strip()));
    }
}
```

JEP 513: Flexible Constructor Bodies

- With the new syntax, we can extract the actions from the call to `this()` and, in particular, call the `split()` only once.
- An additional helper method `parseInt()` may be introduced if you want to make the stripping more elegant and the constructor easier to read:

```
record MyPoint3d(int x, int y, int z)
{
    public MyPoint3d(final String values)
    {
        var separatedValues = values.split(",");
        int x = parseInt(separatedValues[0]);
        int y = parseInt(separatedValues[1]);
        int z = parseInt(separatedValues[2]);

        this(x, y, z);
    }

    private static int parseInt(final String strValue)
    {
        return Integer.parseInt(strValue.strip());
    }
}
```

JEP 513: Flexible Constructor Bodies – New since Java 23

- **When using inheritance, surprises can sometimes occur when methods are called in constructors that are overridden in subclasses.**

```java
public class BaseClass
{
    private final int baseValue;

    public BaseClass(int baseValue)
    {
        this.baseValue = baseValue;

        logValues();
    }

    protected void logValues()
    {
        System.out.println("baseValue: " + baseValue);
    }
}
```

JEP 513: Flexible Constructor Bodies

```java
public class SubClass extends BaseClass
{
    private final String subClassInfo;

    public SubClass(int baseValue, String subClassInfo)
    {
        super(baseValue);
        this.subClassInfo = subClassInfo;
    }

    protected void logValues()
    {
        super.logValues();
        System.out.println("subClassInfo: " + subClassInfo);
    }

    public static void main(final String[] args)
    {
        new SubClass(42, "SURPRISE");
    }
}
```

```
baseValue: 42
subClassInfo: null
```

During the processing of the base class constructor, the attribute `subClassInfo` is still unassigned, as the call to `super()` takes place BEFORE the assignment to the variable. This results in the above but unexpected output.

JEP 513: Flexible Constructor Bodies

```
public class NewSubClass extends BaseClass {

    private final String subClassInfo;

    public NewSubClass(int baseValue, String subClassInfo)
    {
         this.subClassInfo = subClassInfo;
         super(baseValue);
    }

    protected void logValues()
    {
        super.logValues();
        System.out.println("subClassInfo: " + subClassInfo);
    }

    public static void main(final String[] args)
    {
        new NewSubClass(42, "AS_EXPECTED");
    }
}
```

```
baseValue: 42
subClassInfo: AS_EXPECTED
```

During the processing of the base class constructor, the attribute `subClassInfo` is now already unassigned, as the call to `super()` takes place AFTER the assignment to the variable. This results in the above and expected output.

Position 10: Compact Source Files and Instance Main Methods

https://www.freepik.com/premium-vector/realistic-gold-silver-bronze-medal-with-red-ribbon_10027399.htm

Implicitly Declared Classes and Instance Main Methods

- Maybe it's been a while since you learned Java, too.
- If you want to teach Java to novice programmers, you realize **how difficult it is to get started**.
- From the beginner's perspective Java has a **really steep learning curve**.
- It already starts with the simplest Hello-World.

```java
package preview;

public class OldStyleHelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

- Python – reduced to the essentials:

```python
print("Hello, World!")
```

You as trainer mention the following facts for beginners:

1) Forget about `package`, `public` , `class`, `static`, `void`, etc. they are not important right now …
2) Just look at the one line with the `System.out.println()`
3) Oh yes, `System.out` is an instance of a class, but even that is not important now.

Quite a lot of confusing and distracting words and concepts apart from the actual task.

Implicitly Declared Classes and Instance Main Methods

- **PAST**

```
public class InstanceMainMethodOld {
    public static void main(final String[] args) {
        System.out.println("Hello, World!");
    }
}
```

- **PRESENT**

```
class InstanceMainMethod {
    void main() {
        System.out.println("Hello, World!");
    }
}
```

- **PRESENT OPTIMIZED**

```
void main() {
    System.out.println("Hello World!");
}
```

Implicitly Declared Classes and Instance Main Methods

Further possibilities

```java
String greeting = "Hello again!!";

String enhancer(String input, int times)
{
   return " ---> " + input.repeat(times) + " <---";
}

void main()
{
    System.out.println("Hello World!");
    System.out.println(greeting);
    System.out.println(enhancer("Michael", 2));
}
```

```
$ java --enable-preview --source 21\
src/main/java/preview/UnnamedClassesMoreFeatures.java
Hello, World!
Hello again!
 ---> MichaelMichael <---
```

JEP 512: Compact Source Files and Instance Main Methods

- **Two significant innovations were already added in Java 23 and got finalized in Java 25 LTS:**

  - **Interaction with the console:** Implicitly declared classes automatically import threee static methods `print()`, `println()` and `readln()` defined in the `java.io.IO` class that simplify textual interaction with the console.

  - **Automatic module import from `java.base`:** Implicitly declared classes automatically import all public classes and interfaces of the packages exported by the `java.base` module.

- **Based on both, the `main()` method in Java 23 can be written more clearly and briefly as follows:**

```
void main()
{
    IO.println("Shortest and Python-like 'Hello World!'");
}
```

Outlook
Upcoming JEPs in Java 26

Java 26 – What will be included?

Java 26

- JEP 500: Prepare to Make Final Mean Final
- JEP 504: Remove the Applet API
- JEP 516: Ahead-of-Time Object Caching with Any GC
- JEP 517: HTTP/3 for the HTTP Client API
- JEP 522: G1 GC: Improve Throughput by Reducing Synchronization
- JEP 524: PEM Encodings of Cryptographic Objects (Second Preview)
- JEP 525: Structured Concurrency (Sixth Preview)
- JEP 526: Lazy Constants (Second Preview)
- JEP 529: Vector API (Eleventh Incubator)
- JEP 530: Primitive Types in Patterns, instanceof, and switch (Fourth Preview)

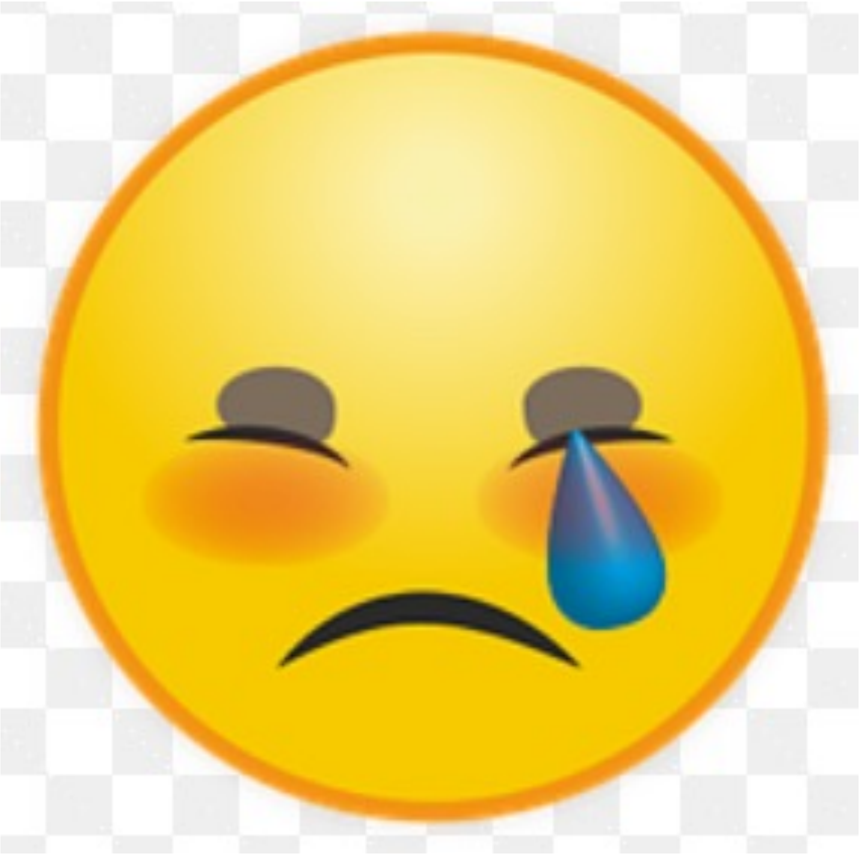
Conclusion

Positive things

- Reliable 6-month release cadence and LTS versions will be released every 2 years
- Java becomes easier and more attractive
- Many nice improvements in syntax and APIs like switch, records, text blocks, …
- Pattern Matching and record patterns
- Virtual Threads & Structured Concurrency
- JAVA 25 LTS has recently been released ☺

On the negative side

- Releases were on time, but sometimes bringing just a few new features, even lot’s of preview features.
- Java 25 LTS and Java 21 LTS contain some unfinished things … in my opinion LTS should contain only few preview and incubators, ideally none
- We have to wait 2 more years to have the nice unnamed classes and vars accessible for stable use
- Why is the syntax of pattern matching inconsistent for `instanceof` and `switch`?

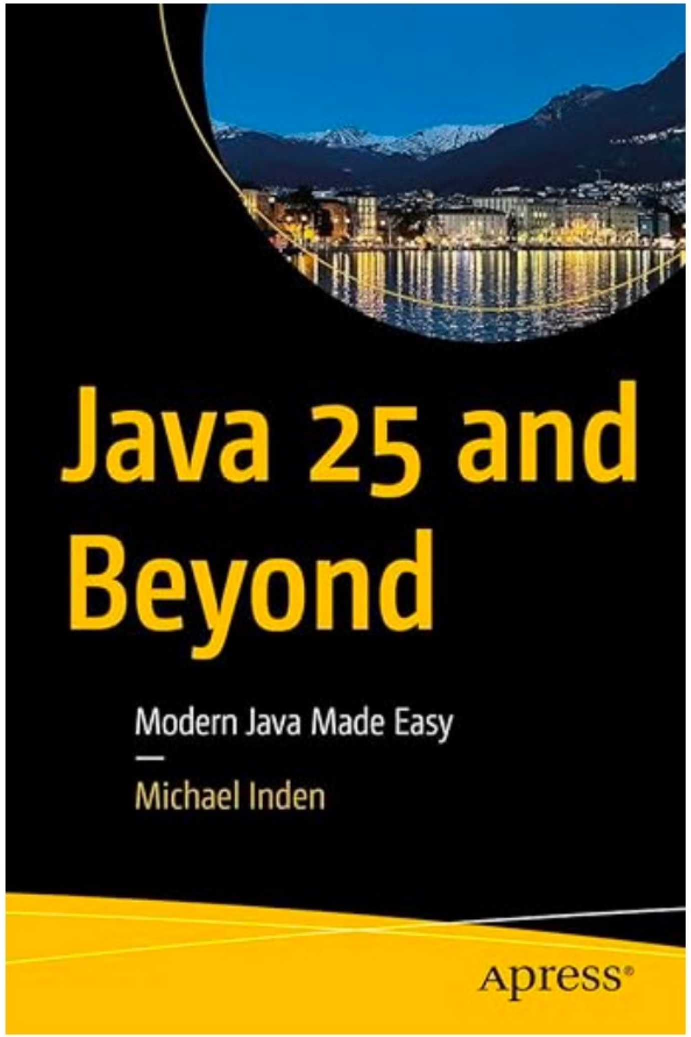


Try it out :-)

https://github.com/Michaeli71/Best-Of-Modern-Java-21-25-My-Favorite-Features

Help

In preparation

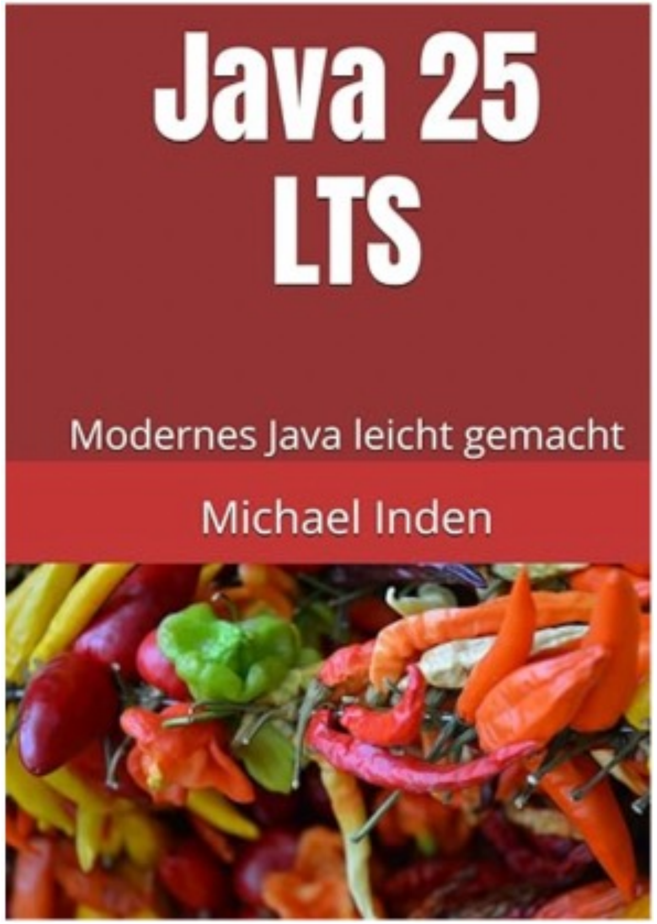

Java 25 and Beyond

Modern Java Made Easy

—

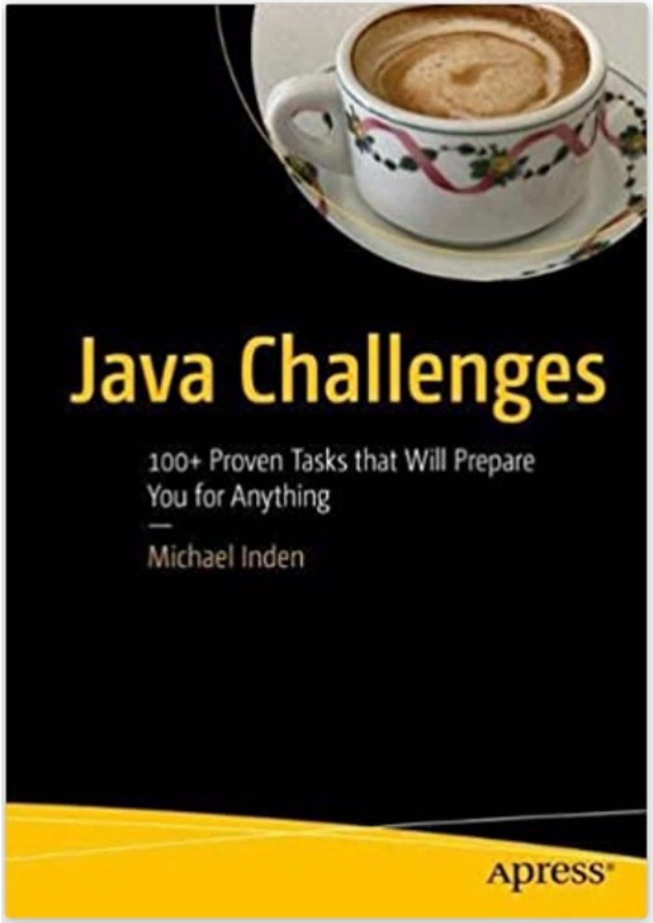
Michael Inden

Apress®


Java 25 LTS

Modernes Java leicht gemacht

Michael Inden


Java Challenges

100+ Proven Tasks that Will Prepare You for Anything

—

Michael Inden

Apress®

https://www.amazon.de/dp/B0FRYX5F6N

Questions?

Thank You