Best of Modern Java 21 – 23
My Favorites

https://github.com/Michaeli71/Best-Of-Modern-Java-21-23-My-Favorites

Michael Inden

Head of Development, freelance author and trainer

Speaker Intro

- **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
- Independent **Consultant, Conference Speaker and Trainer**
- **Since January 2022 Head of Development at Adcubum in Zurich**
- Author @ dpunkt.verlag and APress

E-Mail: michael_inden@hotmail.com

https://github.com/Michaeli71/Best-Of-Modern-Java-21-23-My-Favorites
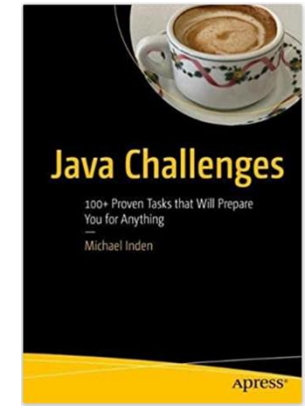
All Time Favorites + My Top 10 from Java 21 LTS to Java 23

All Time Favorites:

- **ATF1: Switch Expressions / Text Blocks / Records (Java 17)**

- **ATF2: Helpful `NullPointerExceptions` / Pattern Matching for instanceof (Java 17)**

My top 10 from Java 21 LTS to Java 23:

1. **Record Patterns (Java 21)**

2. **Pattern Matching for switch (Java 21)**

3. **Virtual Threads (Java 21)**

4. *Structured Concurrency (Preview) (Java 21 & 23)*

5. **Unnamed Variables & Patterns (Java 21 & 22)**

6. **Markdown Comments (Java 23)**

7. *Flexible Constructor Bodies (Second Preview) (Java 23)*

8. **Launch Multi-File Source-Code Programs (Java 22)**

9. *Stream Gatherers (Second Preview) (Java 23)*

10. *Implicitly Declared Classes and Instance Main Methods (Third Preview) (Java 23)*

Build-Tools, IDEs and Sandbox
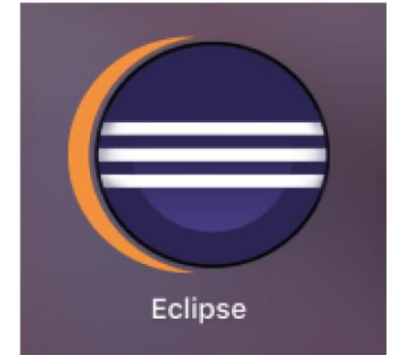
IDE & Tool Support for Java 23

- **Eclipse: Version 2024-09 (with Plugin)**

- **IntelliJ: Version 2024.2.2**

- **Maven: 3.9.9, Compiler Plugin: 3.13.0**

- **Gradle: 8.10**

- **Activation of preview features / Incubator necessary**
  - In dialogs
  - In build scripts

IDE & Tool Support Java 23

- **Eclipse 2024-09 with Plugin**

Activation of preview features necessary

IDE & Tool Support

- **Activation of preview features / Incubator necessary**

- **Activation of preview features / Incubator necessary**

```
java {
    toolchain {
        languageVersion = JavaLanguageVersion.of(23)
    }
}


tasks.withType(JavaCompile).configureEach {
    options.compilerArgs += ["--enable-preview"]
}


tasks.withType(JavaCompile).configureEach {
    options.compilerArgs += ["--enable-preview",
                "--add-modules", "jdk.incubator.vector"]
}
```

IDE & Tool Support

- ## Activation of preview features / Incubator necessary

```xml
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.13</version>
    <configuration>
      <source>23</source>
      <target>23</target>
      <!-- Important for Preview Features -->
      <compilerArgs>--enable-preview</compilerArgs>
    </configuration>
  </plugin>
</plugins>
```

```xml
<plugin>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.13.0</version>
  <configuration>
    <release>23</release>
    <compilerArgs>
      <arg>--enable-preview</arg>
      <arg>--add-modules</arg>
      <arg>jdk.incubator.vector</arg>
    </compilerArgs>
  </configuration>
</plugin>
```

https://javaalmanac.io/

The Java Version Almanac

Systematic collection of information about the history and the future of Java.

Details	Status	Documentation	Download	Compare API to										
Java 24	DEV	API | Notes	JDK | JRE	23	22	21	20	19	18	17	16	15	14	...
Java 23	REL	API | Lang | VM | Notes	JDK | JRE	22	21	20	19	18	17	16	15	14	13	...
Java 22	EOL	API | Lang | VM | Notes	JDK | JRE	21	20	19	18	17	16	15	14	13	12	...
Java 21	LTS	API | Lang | VM | Notes	JDK | JRE	20	19	18	17	16	15	14	13	12	11	...
Java 20	EOL	API | Lang | VM | Notes	JDK | JRE	19	18	17	16	15	14	13	12	11	10	...
Java 19	EOL	API | Lang | VM | Notes	JDK | JRE	18	17	16	15	14	13	12	11	10	9	...
Java 18	EOL	API | Lang | VM | Notes	JDK | JRE	17	16	15	14	13	12	11	10	9	8	...
Java 17	LTS	API | Lang | VM | Notes	JDK | JRE	16	15	14	13	12	11	10	9	8	7	...
Java 16	EOL	API | Lang | VM | Notes	JDK | JRE	15	14	13	12	11	10	9	8	7	6	...
Java 15	EOL	API | Lang | VM | Notes	JDK | JRE	14	13	12	11	10	9	8	7	6	5	...
Java 14	EOL	API | Lang | VM | Notes	JDK | JRE	13	12	11	10	9	8	7	6	5	1.4	...
Java 13	EOL	API | Lang | VM | Notes	JDK | JRE	12	11	10	9	8	7	6	5	1.4	1.3	...
Java 12	EOL	API | Lang | VM | Notes	JDK | JRE	11	10	9	8	7	6	5	1.4	1.3	1.2	...
Java 11	LTS	API | Lang | VM | Notes	JDK | JRE	10	9	8	7	6	5	1.4	1.3	1.2	1.1	...
Java 10	EOL	API | Lang | VM | Notes	JDK | JRE	9	8	7	6	5	1.4	1.3	1.2	1.1	1.0	
Java 9	EOL	API | Lang | VM | Notes	JDK | JRE	8	7	6	5	1.4	1.3	1.2	1.1	1.0		
Java 8	LTS	API | Lang | VM | Notes	JDK | JRE	7	6	5	1.4	1.3	1.2	1.1	1.0			
Java 7	EOL	API | Lang | VM | Notes	JDK | JRE	6	5	1.4	1.3	1.2	1.1	1.0				
Java 6	EOL	API | Lang | VM | Notes	JDK | JRE	5	1.4	1.3	1.2	1.1	1.0					
Java 5	EOL	API | Lang | VM | Notes		1.4	1.3	1.2	1.1	1.0						
Java 1.4	EOL	API		1.3	1.2	1.1	1.0							
Java 1.3	EOL	API		1.2	1.1	1.0								
Java 1.2	EOL	API | Lang		1.1	1.0									
Java 1.1	EOL	API		1.0										
Java 1.0	EOL	API | Lang | VM												
Pre 1.0	EOL													

Data Source

Sandbox – https://javaalmanac.io/

Sandbox

Instantly compile and run Java 23 snippets without a local Java installation.

```
Java23.java    ▶ Run                                          23+37-2369
 1  import java.lang.reflect.ClassFileFormatVersion;
 2
 3  public class Java23 {
 4
 5      public static void main(String[] args) {
 6          var v = ClassFileFormatVersion.latest();
 7          System.out.printf("Hello Java bytecode version %s!", v.major());
 8      }
 9
10  }
```

No Support for Preview Features!

```
Java23.java    ▶ Run                                          23+37-2369
Hello Java bytecode version 67!
```

All Time Favorites

ATF 1:
Switch Expressions /
Text Blocks / Records

Switch Expressions

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

Return-Value

```java
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 case-through either
    - switch can now return a value, avoids artificial auxiliary variables

Switch Expressions

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

```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";
    };
}
```

Switch Expressions: yield with return value

```java
public static void main(final String[] args)
{
    DayOfWeek day = DayOfWeek.SUNDAY;

    int numOfLetters = switch (day)
    {
        case MONDAY, FRIDAY, SUNDAY -> {
            if (day == DayOfWeek.SUNDAY)
                System.out.println("SUNDAY is FUN DAY");
            yield 6;
        }
        case TUESDAY            -> 7;
        case THURSDAY, SATURDAY -> 8;
        case WEDNESDAY          -> 9;
    };
    System.out.println(numOfLetters);
}
```

SUNDAY is FUN DAY

6

Text Blocks

Text Blocks

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

Text Blocks

```java
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

Records

Enhancement Record

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
- **Implementations of accessor methods as well as `equals()` and `hashCode()` automatically and adhering to contracts**

ATF 2:
Helpful `NullPointerExceptions` / Pattern Matching for `instanceof`

Helpful `NullPointerExceptions`

```java
public static void main(final String[] args)
{
    SomeType a = null;
    a.value = "ERROR";
}
```

Exception in thread "main" java.lang.NullPointerException
at java14.NPE_Example.main(NPE_Example.java:8)

https://www.freepik.com/free-vector/realistic-bomb-illustration_3886825.htm
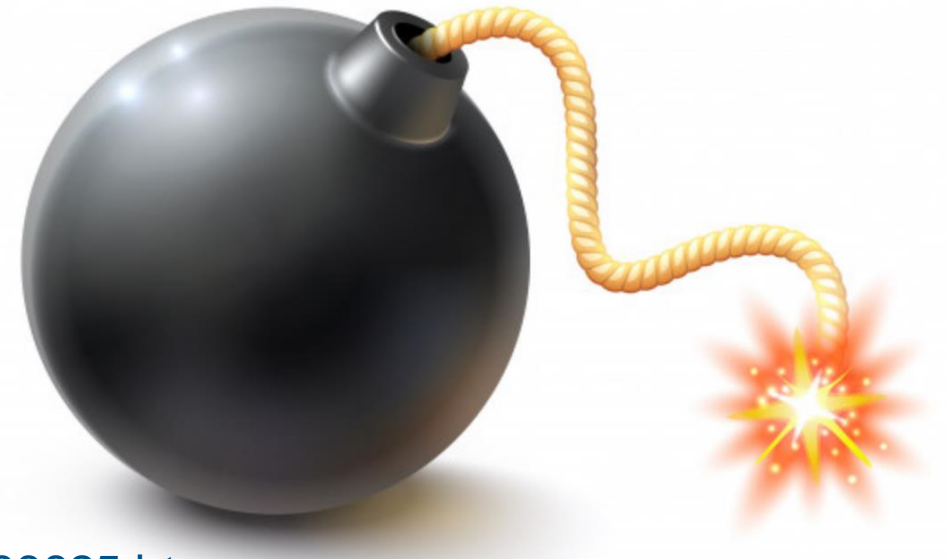
Helpful `NullPointerExceptions`

```java
public static void main(final String[] args)
{
    SomeType a = null;
    a.value = "ERROR";
}
```

Exception in thread "main" java.lang.NullPointerException
at java14.NPE_Example.main(NPE_Example.java:8)

-XX:+ShowCodeDetailsInExceptionMessages

Exception in thread "main" java.lang.NullPointerException: Cannot assign field "value" because "a" is null
at java14.NPE_Example.main(NPE_Example.java:8)

```
public static void main(final String[] args)
{
    int width = getWindowManager().getWindow(5).size().width();
    System.out.println("Width: " + width);
}
```

Exception in thread "main" java.lang.NullPointerException: Cannot invoke "jvm.NPE_Third_Example$Window.size()" because the return value of "jvm.NPE_Third_Example$WindowManager.getWindow(int)" is null
at jvm.NPE_Third_Example.main(NPE_Third_Example.java:7)

Pattern Matching
instanceof

Pattern Matching `instanceof`

- **OLD STYLE**

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

- **NEW STYLE**

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

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

My Top 10

Position 1: Record Patterns

JEP 440: Record Patterns

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

```java
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: Pattern Matching for `switch`

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

```java
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));
    }
}
```

- **=>**

```java
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: Pattern Matching for `switch`

- **Record patterns can be nested**

- **provide a declarative, powerful, and combinable form of data navigation and processing.**

```java
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:**

```java
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)`.**

```java
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`

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

```java
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 427: 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");
    }
}
```
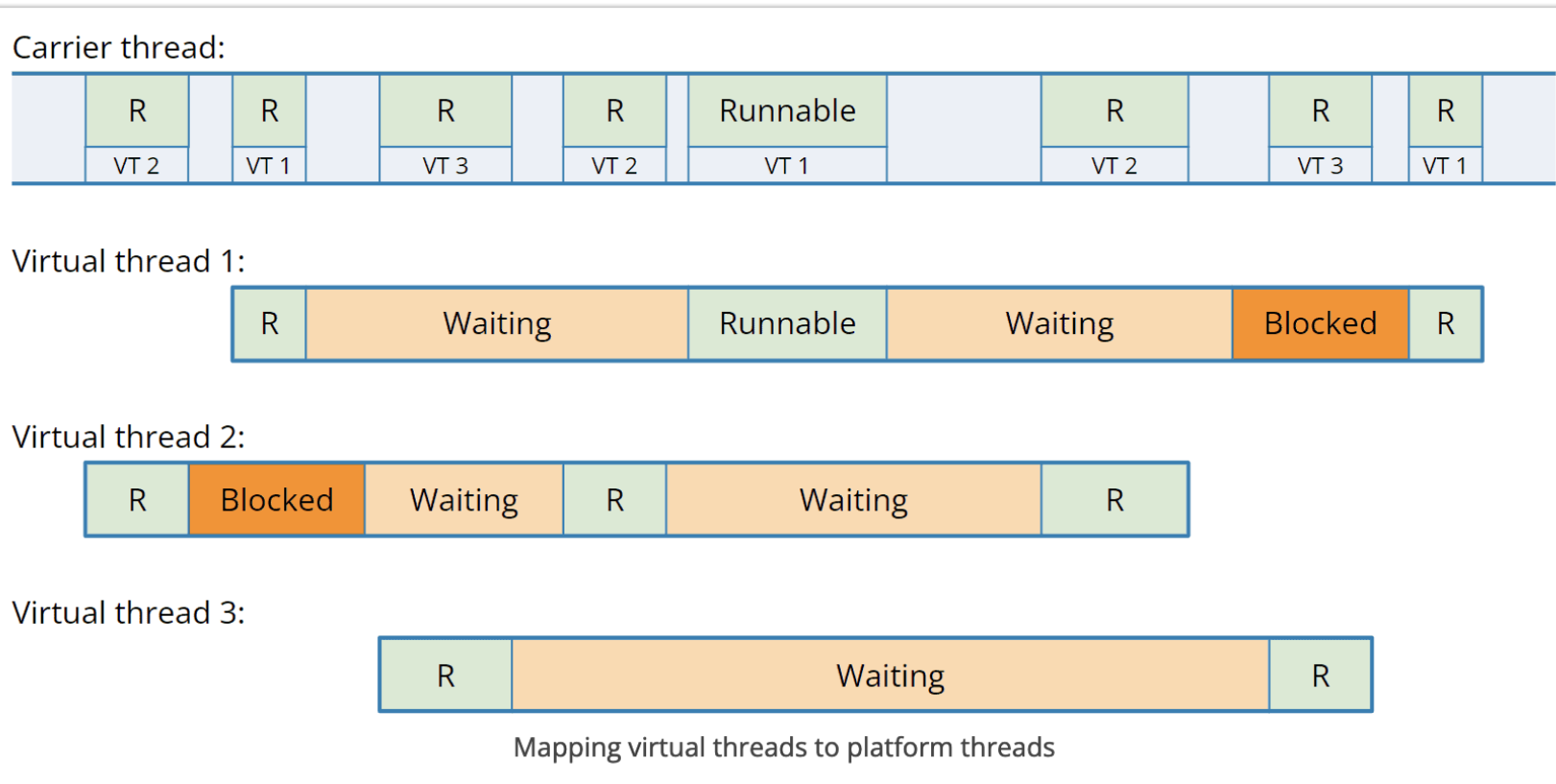
Position 3: Virtual Threads

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.**

Mapping virtual threads to platform threads

JEP 444: Virtual Threads

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

Concurrency Issues

Why is it bad to block?

```
Json request          = buildContractRequest(id);
String contractJson   = contractServer.getContract(request);
Contract contract     = Json.unmarshal(contractJson);
```

ns   ms   ns

- **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.**

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

    try (var executor = Executors.new  Executors.newVirtualThreadPerTaskExecutor())
    {
        for (int  i = 0; i < 10_000; 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");
}
```

Position 4: Structured Concurrency (Preview)

JEP 453: 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:**

```java
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`

```java
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`:**

```java
static Response handle(Long userId) throws ExecutionException,
                       InterruptedException
{
    try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
        var userSubtask  = scope.fork(() -> findUser(userId));
        var ordersSubtask = scope.fork(() -> fetchOrder(userId));

        scope.join();          // Join both forks
        scope.throwIfFailed(); // ... and propagate errors

        // Here, both forks have succeeded, so compose their results
        return new Response(userSubtask.get(), ordersSubtask.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.**

The `StructuredTaskScope` class offers two strageies:

- `ShutdownOnFailure` – 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.

- `ShutdownOnSuccess` – 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 453: Structured Concurrency

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

```
public static void main(final String[] args) throws ExecutionException,
                                  InterruptedException
{
    try (var scope =
        new StructuredTaskScope.ShutdownOnSuccess<NetworkConnection>())
    {
        var result1 = scope.fork(() -> tryToGetWifi());
        var result2 = scope.fork(() -> tryToGet5g());
        var result3 = scope.fork(() -> tryToGet4g());
        var result4 = scope.fork(() -> tryToGet3g());
        scope.join();

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

Position 5: Unnamed Variables and Patterns (Preview)

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

```
Point p3_4 = new Point(3, 4);
var green_p3_4 = new ColoredPoint(p3_4, Color.GREEN);

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

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

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

JEP 443: Unnamed Patterns and Variables (Preview)

- **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 443: Unnamed Patterns and Variables (Preview)

- **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 _)

- **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 443: Unnamed Patterns and Variables (Preview)

- **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 _)**

- **Unnamed pattern**

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

- **=>**

```java
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

```java
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;
}
```

Position 6:
Markdown Comments

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.

```
/// **FETT** \
/// *kursiv* \
/// _kursiv_ \
/// _**FETT und KURSIV**_ \
/// `code-font` \
/// _**`code-font FETT und KURSIV`**_ \
///
/// Mehrzeiliger Sourcecode:
/// ```
/// public static int max(int a, int b) {
///    return (a >= b) ? a : b;
/// }
/// ```
```

📁 syntax

```java
public class MarkDownComment
```

FETT
kursiv
kursiv
FETT und KURSIV
`code-font`
`code-font FETT und KURSIV` \

Mehrzeiliger Sourcecode:

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

📁 Java23Examples

```
/// - Punkt A
/// * Punkt B
/// - Punkt C
///
/// 1. Eintrag 1
/// 1. Eintrag 2 -- **wird automatisch nummeriert, also 2.**
/// 1. Eintrag 3
/// 2. Eintrag 4 -- **wird automatisch auf 4. geändert**
```

- Punkt A
- Punkt B
- Punkt C
1. Eintrag 1
2. Eintrag 2 -- **wird automatisch nummeriert, als**
3. Eintrag 3
4. Eintrag 4 -- **wird automatisch auf 4. geändert**

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

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

Position 7: Flexible Constructor Bodies / Statements before super(...) (Second Preview)

JEP 482: 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.**

- **Conventional workaround: static helper method**

```java
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 482: Flexible Constructor Bodies – News in Java 22/23

- **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:**

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

        super(value);
    }
}
```

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

```java
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 zy)
{
   public MyPoint3dOld(final String values)
   {
      this(Integer.parseInt(values.split(",")[0].strip()),
         Integer.parseInt(values.split(",")[1].strip()),
         Integer.parseInt(values.split(",")[2].strip()));
   }
}
```

- **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:**

```java
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 482: Flexible Constructor Bodies – News in Java 23

```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.

```java
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 8: Launch Multi-File Source-Code Programs

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 9: Stream Gatherers (Preview)

JEP 473: 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 473: 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 473: 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]]
```

- **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]]
```

- **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]
```

- **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()`:**

```
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(): [122333444455555666666777777]
```

JEP 473: 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:**

```java
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, 12233344445555, 122333444455555666666, 1223334444555556666667777777]

Position 10: Implicitly Declared Classes and Instance Main Methods (Preview)

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**

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

- **PRESENT**

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

- **PRESENT OPTIMIZED**

```java
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 477: Implicitly Declared Classes and Instance Main Methods

- **JEP 477 from Java 23 contains the innovations from Java 22. In addition, two significant innovations have been added in Java 23:**

    - **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()
{
    println("Shortest and Python-like 'Hello World!'");
}
```

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**
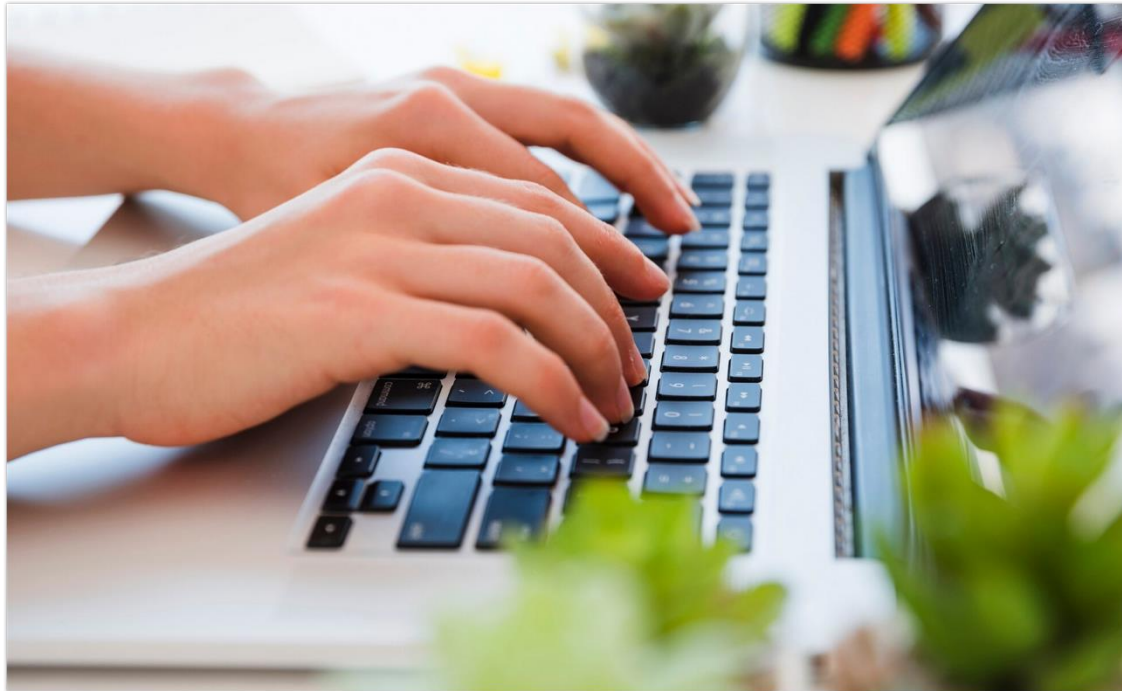
On the negative side

- Next releases were on time, but sometimes bringing just a few new features (execpt Java 14), even only preview features.

- Java 21 LTS contains lots of 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

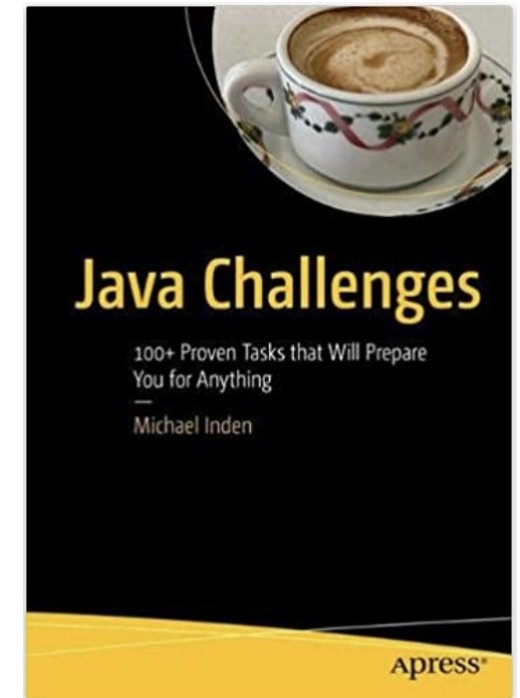- Why is the syntax of pattern matching inconsistent for `instanceof` and `switch`?

Try it out :-)

https://github.com/Michaeli71/Best-Of-Modern-Java-21-23-My-Favorites

O'REILLY®

Java
lernen
kurz & gut

O'Reillys Taschenbibliothek

Michael Inden

Michael Inden

Java
Die Neuerungen in Version 17 LTS, 18 und 19

dpunkt.verlag

5. Auflage

Michael Inden

Der Weg zum Java-Profi

Konzepte und Techniken für die professionelle Java-Entwicklung

dpunkt.verlag

Java Challenges

100+ Proven Tasks that Will Prepare You for Anything

Michael Inden

Apress®

Questions?

Thank You