



Best of Modern Java 21 & 22 – meine Lieblingsfeatures

<https://github.com/Michaeli71/Best-Of-Modern-Java-21-22-My-Favorite-Features>



Michael Inden

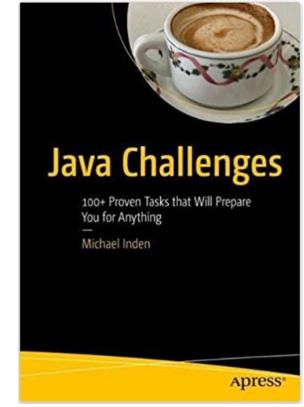
Head of Development, freiberuflicher Buchautor und Trainer

Speaker Intro



- **Michael Inden, Jahrgang 1971**
- **Diplom-Informatiker, C.v.O. Uni Oldenburg**
- **~8 ¼ Jahre SSE bei Heidelberger Druckmaschinen AG in Kiel**
- **~6 ¾ Jahre TPL, SA bei IVU Traffic Technologies AG in Aachen**
- **~4 ¼ Jahre LSA / Trainer bei Zühlke Engineering AG in Zürich**
- **~3 Jahre TL / CTO bei Direct Mail Informatics / ASMIQ in Zürich**
- **Freiberuflicher Consultant, Trainer und Konferenz-Speaker**
- **Seit Januar 2022 Head of Development bei Adcubum in Zürich**
- **Autor und Gutachter beim dpunkt.verlag / O'Reilly / APress**

E-Mail: michael_inden@hotmail.com



All Time Favorites + Meine Top 10 aus Java 21 und 22



All Time Favorites:

- **ATF1: Switch Expressions / Text Blocks / Records (Java 17)**
- **ATF2: Hilfreiche NullPointerExceptions / Pattern Matching for instanceof (Java 17)**

Meine Top 10 aus Java 21 LTS und 22:

1. **Record Patterns (Java 21)**
2. **Pattern Matching for switch (Java 21)**
3. **Virtual Threads (Java 21)**
4. **Structured Concurrency (Preview) (Java 21 & 22)**
5. **Statements before super(...) (Preview) (Java 22)**
6. **Unnamed Variables & Patterns (Java 21 & 22)**
7. **Launch Multi-File Source-Code Programs (Java 22)**
8. **String Templates (Second Preview) (Java 21 & 22)**
9. **Stream Gatherers (Preview) (Java 22)**
10. **Implicitly Declared Classes and Instance Main Methods (Second Preview) (Java 22)**





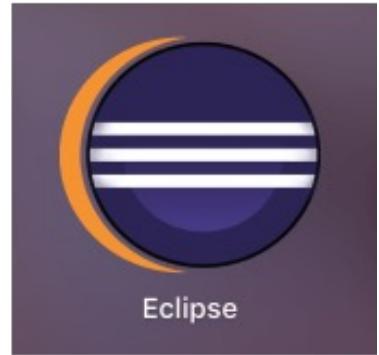
Build-Tools, IDEs und Sandbox



IDE & Tool Support für Java 21



- Eclipse: Version 2023-12 (nicht alle Previews supported)
- IntelliJ: Version 2023.3.x
- Maven: 3.9.6, Compiler Plugin: 3.11.0
- Gradle: 8.5
- Aktivierung von Preview-Features / Incubator nötig
 - In Dialogen
 - In Build Scripts



Maven™

 **Gradle**

IDE & Tool Support für Java 22



- Eclipse: Version 2024-03 (nicht alle Previews supported)
- IntelliJ: Version 2024.1
- Maven: 3.9.6, Compiler Plugin: 3.11.0
- Gradle: 8.7
- Aktivierung von Preview-Features / Incubator nötig
 - In Dialogen
 - In Build Scripts



Maven™

 **Gradle**

IDE & Tool Support Java 22



- Eclipse 2024-03 mit Plugin
- Aktivierung von Preview-Features nötig



Eclipse Marketplace

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

Search Recent Popular Favorites Installed Research at the Eclipse Marketplace

Find: All

Java 22 Support for Eclipse 2024-03 (4.31)
This marketplace solution provides Java 22 support for Eclipse 2024-03. It requires you have the latest Eclipse release, which is... [more info](#)
by [Eclipse Foundation](#), EPL 2.0

1 Installs: 4.91K (162 last month)

Eclipse Marketplace

Marketplaces

Eclipse Marketplace

Properties for Java22Examples

Java Compiler

Enable project specific settings [Configure Workspace Settings...](#)

JDK Compliance

Use compliance from execution environment 'JavaSE-22' on the [Java Build Path](#)

Compiler compliance level:

Use '--release' option

Use default compliance settings

Enable preview features for Java 22

Preview features with severity level:

Generated .class files compatibility:

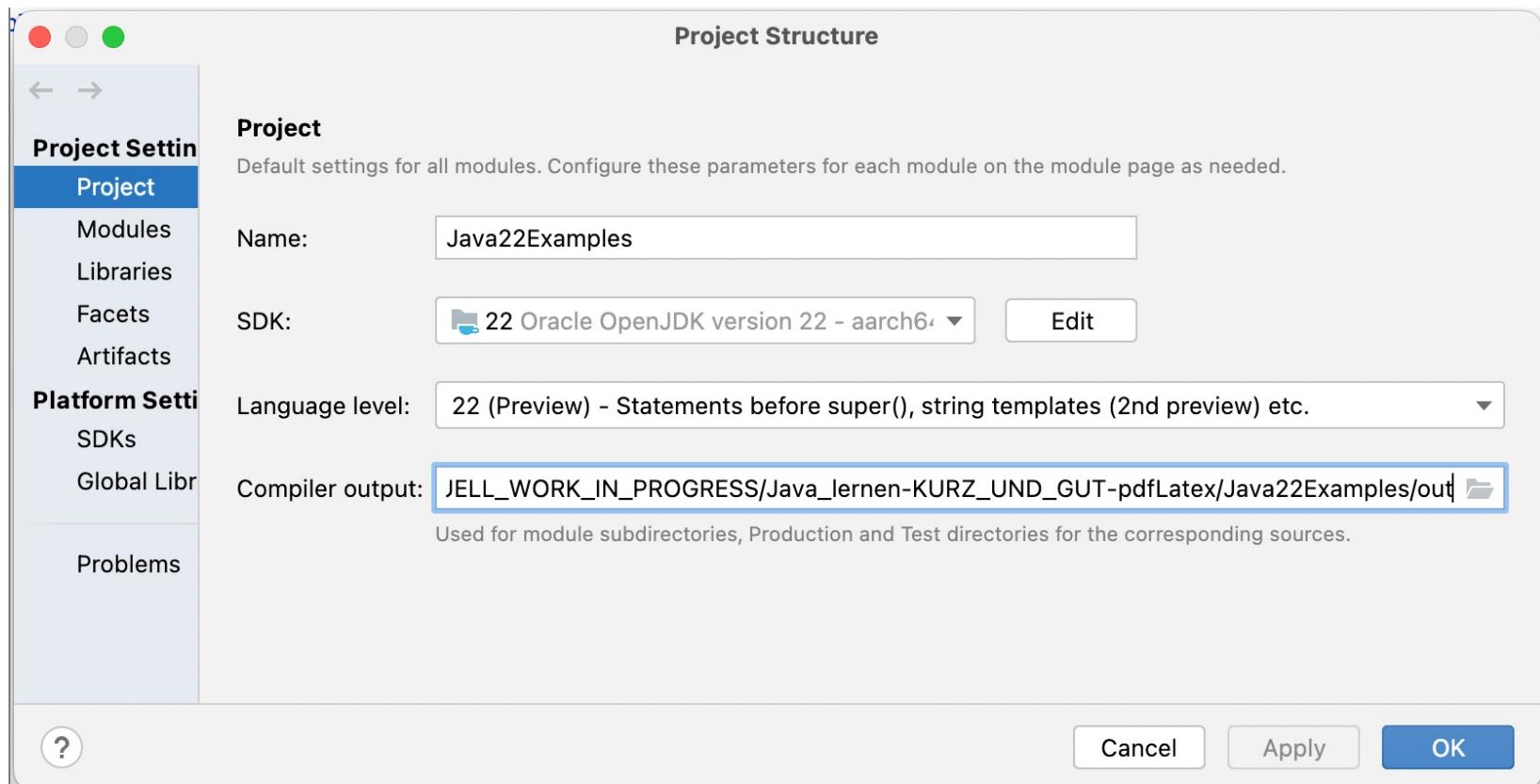
Source compatibility:

Disallow identifiers called 'assert':

IDE & Tool Support



- Aktivierung von Preview-Features nötig





- Aktivierung von Preview-Features / Incubator nötig

sourceCompatibility=22
targetCompatibility=22



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

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





- Aktivierung von Preview-Features / Incubator nötig

```
java {  
    toolchain {  
        languageVersion = JavaLanguageVersion.of(22)  
    }  
}
```

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

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





- Aktivierung von Preview-Features / Incubator nötig

```
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.11</version>
    <configuration>
      <source>22</source>
      <target>22</target>
      <!-- Important for Preview Features -->
      <compilerArgs>--enable-preview</compilerArgs>
    </configuration>
  </plugin>
</plugins>
```

```
<plugin>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.11.0</version>
  <configuration>
    <release>21</release>
    <compilerArgs>
      <arg>--enable-preview</arg>
      <arg>--add-modules</arg>
      <arg>jdk.incubator.vector</arg>
    </compilerArgs>
  </configuration>
</plugin>
```

Maven™

3.9.6

The Java Version Almanac

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

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



Sandbox

Instantly compile and run Java 22 snippets without a local Java installation.

Java22.java

▶ Run

22+36-2370

```
1 import java.text.ListFormat;
2 import java.util.List;
3
4 public class Java22 {
5
6     public static void main(String[] args) {
7         var f = ListFormat.getInstance();
8         System.out.printf(f.format(List.of("classes", "interfaces", "enums", "records")));
9     }
10
11 }
```

No Support
for Preview
Features!

Java22.java

▶ Run

22+36-2370

```
classes, interfaces, enums, and records
```



All Time Favorites





ATF 1: Switch Expressions / Text Blocks / Records



Switch Expressions



- Abbildung von Wochentagen auf deren Länge ... elegant mit modernem Java:

Return-
Value

```
DayOfWeek day = DayOfWeek.FRIDAY;  
int num0fLetters = switch (day)  
{  
    case MONDAY, FRIDAY, SUNDAY -> 6;  
    case TUESDAY -> 7;  
    case THURSDAY, SATURDAY -> 8;  
    case WEDNESDAY -> 9;  
};
```

- Elegantere Schreibweise beim case:

- Neben dem offensichtlichen Pfeil statt des Doppelpunkts
- auch mehrere Werte
- Kein break nötig, auch kein Fall-Through
- switch kann nun einen Wert zurückgeben, vermeidet künstliche Hilfsvariablen

Switch Expressions



- Abbildung von Monaten auf deren Namen ... elegant mit modernem Java:

```
static String monthToName(final Month month)
{
    return switch (month)
    {
        case JANUARY -> "January";
        default -> "N/A"; // hier KEIN Fall Through
        case FEBRUARY -> "February";
        case MARCH -> "March";
        case JULY -> "July";
    };
}
```

Switch Expressions



```
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



```
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



Records



Erweiterung Record

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

```
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);
}
```

```
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 für komplexere Rückgabewerte und Parameter



```
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 für Pairs und Tupel



```
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) {};
```

- **Extrem wenig Schreibaufwand**
- **Sehr praktisch für Pairs, Tuples usw.**
- **Records funktionieren prima mit primitiven Typen und auch mit Generics**
- **Implementierungen von Accessor-Methoden sowie equals() und hashCode()** automatisch und vor allem kontraktkonform, ebenso `toString()`



ATF 2: Hilfreiche NullPointerExceptions / Pattern Matching for instanceof



Hilfreiche NullPointerExceptions



```
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)



Hilfreiche NullPointerExceptions



```
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)

Hilfreiche NullPointerExceptions



```
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 bei instanceof



Pattern Matching bei instanceof



- **ALT**

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

- **NEU**

```
if (obj instanceof Person person)
{
    // Hier kann man auf die Variable person direkt zugreifen
}
```

Pattern Matching bei instanceof



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



Meine Top 10





Platz 1: Record Patterns





- Basis für diesen JEP und seine Vorgänger ist das Pattern Matching für instanceof aus 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: " + x + ", y: " + y + ", sum: " + (x + y));
    }
}
```

- Zwar ist das oftmals schon praktisch, aber man muss noch teilweise umständlich auf die einzelnen Bestandteile zugreifen.



- Ziel ist es, Records deklarativ in ihre Bestandteile zerlegen und auf diese zugreifen zu können.

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

- =>

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

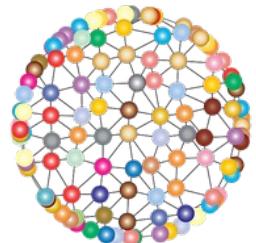


- Record Patterns können auch verschachtelt werden, um eine deklarative, leistungsfähige und kombinierbare Form der Datennavigation und -verarbeitung zu ermöglichen.

```
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, color),
                                  ColoredPoint lowerRight))
    {
        System.out.println(color);
    }
}
```



**Wo können Record Patterns
ihre Stärke ausspielen?**



- Nehmen wir einmal folgende Records als Datenmodell an:

```
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) {  
}
```



- In Legacy-Code findet man tief verschachtelte Abfragen wie diese:

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

            if (reservation.destination() != null)
            {
                LocalDate birthday = person.birthday();
                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;
}
```



- Mit verschachtelten Record Patterns schreiben wir die obige Verschachtelung der ifs elegant und viel verständlicher wie folgt:

```
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 405/440: Record Patterns



```
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;
}
```

- **Die Prüfung mit instanceof schlägt automatisch fehl, falls eine der Record-Komponenten null ist, also hier Person oder City (destination).**
- **Lediglich die Attribute werden so nicht abgesichert und sind ggf. auf null zu prüfen.**
- **Wenn man sich jedoch den guten Stil angewöhnt, null als Wert von Parametern bei Aufrufen zu vermeiden, kann man sogar darauf verzichten.**



Platz 2: Pattern Matching bei switch





- **Problemfeld:** Es können mehrere Patterns auf eine Eingabe matchen.

```
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 when s.length() > 5 -> System.out.println(s.toUpperCase());
        case String s
        case Integer i
        default -> {}
    }
}
```

- Dasjenige, das «am allgemeingültigsten» passt, wird dominierendes Pattern genannt.
- Im Beispiel dominiert das kürzere Pattern `String s` das längere davor angegebene.

JEP 441: Pattern Matching bei switch mit Record Patterns



```
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");
    }
}
```

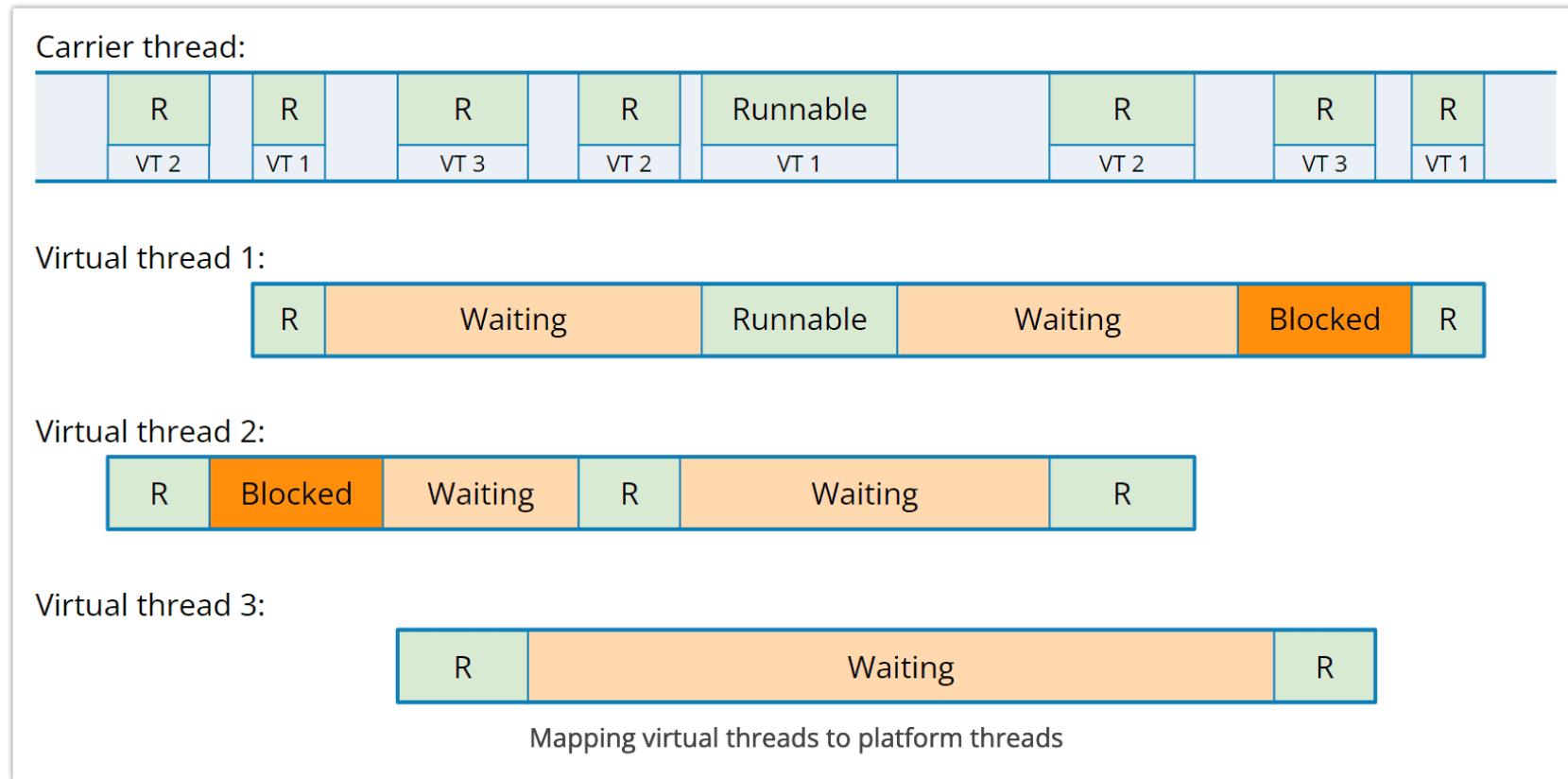


Platz 3: Virtual Threads





- Dieser JEP führt das Konzept leichtgewichtiger virtueller Threads ein.
- Virtuelle Threads «fühlen» sich wie normale Threads an, sind auf vom Typ `java.lang.Thread`, werden aber nicht 1:1 auf Betriebssystem-Threads abgebildet.





- Bereits bekannt: Die **leichtgewichtigen virtuellen Threads werden nicht direkt auf Threads des Betriebssystems abgebildet**.
- Besser noch: Bereits vorhandener Code, der das bisherige Thread-API verwendet, lässt sich mit minimalen Änderungen auf **virtuelle Threads umstellen**.
- **Virtuelle Threads sind ideal für Anwendungen, die hohen Durchsatz erfordern und insbesondere wenn viele der parallelen Aufgaben viel Zeit mit Warten verbringen**.
- **Factory-Methoden wie `newVirtualThreadPerTaskExecutor()` steuern, ob virtuelle Threads oder Plattform-Threads (z.B. mit `Executors.newCachedThreadPool()`) verwendet werden sollen**.

JEP 444: Virtual Threads



```
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");
}
```



- Virtuelle Threads erlauben im Bereich der Serverprogrammierung wieder mit jeweils einem separaten Thread pro Request zu arbeiten. Hilfreich weil in der Regel viele Client-Requests blockierendes I/O wie das Abrufen von Ressourcen durchführen.
- Was ist das Problem an blockierendem I/O? => miserable Serverauslastung

Concurrency Issues

Why is it bad to block?

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





Platz 4: Structured Concurrency (Preview)





- Dieser JEP bringt die Structured Concurrency als Vereinfachung von Multithreading.
- Wenn eine Aufgabe aus mehreren Teilaufgaben besteht, die parallel verarbeitet werden können, ermöglicht Structured Concurrency, dies auf besonders lesbare und wartbare Weise umzusetzen.
- Betrachten wir das Ermitteln eines Users und dessen Bestellungen anhand einer User-ID:

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

    return new Response(user, orders);
}
```
- Beide Aktionen könnten parallel ablaufen.



```
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);  
}
```

- Weil die Teilaufgaben parallel ausgeführt werden, können sie unabhängig voneinander erfolgreich sein oder fehlschlagen. Dann kann das Handling recht kompliziert werden.
- Oftmals möchte man beispielsweise nicht, dass das zweite get() aufgerufen wird, wenn bereits bei der Abarbeitung der Methode findUser() eine Exception aufgetreten ist.



- Umsetzung mit Structurd Concurrency in Form der Klasse **StructuredTaskScope**:

```
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  
  
        return new Response(userSubtask.get(), ordersSubtask.get());  
    }  
}
```

- Konkurrierende Teilaufgaben mit **fork()** abspalten und mit blockierendem Aufruf von **join()** Ergebnisse einsammeln
- **join()** wartet, bis alle Teilaufgaben abgearbeitet sind oder ein Fehler auftrat.



Die Klasse `StructuredTaskScope` besitzt zwei Spezialisierungen:

- `ShutdownOnFailure` – fängt die erste `Exception` ab und beendet den `StructuredTaskScope`. Diese Klasse ist dafür gedacht, wenn die Ergebnisse aller Teilaufgaben benötigt werden ("`invoke all`"); wenn jedoch eine Teilaufgabe fehlschlägt, werden die Ergebnisse der anderen, noch nicht abgeschlossenen Teilaufgaben nicht mehr benötigt.
- `ShutdownOnSuccess` – ermittelt das erste eintreffende Ergebnis und beendet dann den `StructuredTaskScope`. Das hilft, wenn bereits das Ergebnis einer beliebigen Teilaufgabe ausreicht ("`invoke any`") und es nicht notwendig ist, auf die Ergebnisse anderer, nicht abgeschlossener Aufgaben zu warten.



Platz 5: Statements before super(...)

(Preview)



JEP 447: Statements before super(...)



- Manchmal bietet es sich an, den oder die Konstruktorparameter zu validieren, bevor man diese (ansonsten ungeprüft) bei einem Aufruf des Basisklassenkonstruktors übergibt.

```
class BaseInteger
{
    private final long value;

    BaseInteger(final long value)
    {
        this.value = value;
    }

    public long getValue()
    {
        return value;
    }

    public static void main(final String[] args)
    {
        new BaseInteger(4711);
    }
}
```

JEP 447: Statements before super(...)



- Wenn wir auf den Sourcecode schauen, so wirkt dieser nicht gerade elegant – die Prüfung erfolgt auch erst nach Konstruktion der Basisklasse ...
- Dadurch sind potenziell schon unnötige Aufrufe sowie Objektkonstruktionen erfolgt – insbesondere etwas älterer Legacy-Code zeichnet sich unrühmlicherweise dadurch aus, dass (zu) viele Aktionen bereits im Konstruktor erfolgen.

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

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

JEP 447: Statements before super(...)



- **Herkömmliche Abhilfe: statische Hilfsmethode**

```
public class PositiveBigIntegerOld2 extends BigInteger
{
    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 447: Statements before super(...) – Neuerung in Java 22



- Die Argumentprüfung wird deutlich besser lesbar und verständlich, wenn die Validierungslogik direkt im Konstruktor noch vor dem Aufruf von super() geschieht.
- Durch JEP 447 lassen sich nun in einem Konstruktor dessen Argumente validieren, bevor dort der Konstruktor der Superklasse aufgerufen wird:

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

        super(value);
    }
}
```

JEP 447: Statements before super(...)



- Manchmal bietet es sich an, Aktionen vor dem Aufruf von `this()` auszuführen, um mehrfache Aktionen zu vermeiden, hier `split()`:

```
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()));
    }
}
```

JEP 447: Statements before super(...) – Neuerung in Java 22



- Mit der neuen Syntax können wir die Aktionen aus dem Aufruf von `this()` herausziehen und insbesondere das `split()` auch nur einmal aufrufen.
- Möchte man das für die Logik irrelevante Stripping eleganter und den Konstruktor leichter lesbar gestalten, so implementiert man noch eine zusätzliche Hilfsmethode `parseInt()`:

```
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());
    }
}
```



Platz 6: Unnamed Variables and Patterns (Preview)





- Was beobachten Sie bei der Verwendung von Record Patterns?

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

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

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

- Nur ein paar Bestandteile/Attribute im Record Pattern sind wirklich von Interesse!

JEP 443: Unnamed Patterns and Variables (Preview)



- Und was ist mit ähnlichen Situationen in "normalem" 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  
}
```

- Einige Variablen sind im nachfolgenden Code unbenutzt

Unnamed Variables & Patterns



- Dieser JEP finalisiert den Vorgänger JEP 443 und ermöglicht es, Variablen oder Teile innerhalb von Record Patterns durch ein `_` als unbenutzt und unbrauchbar zu markieren. Zur Erinnerung sei erwähnt, dass es die folgenden drei Varianten gibt:
 1. **Unnamed variable** – erlaubt die Verwendung von `_` für die Benennung oder Markierung von nicht verwendeten Variablen.
 2. **Unnamed pattern variable** – erlaubt das Weglassen des Bezeichners, der normalerweise dem Typ (oder var) in einem Record Pattern folgen würde.
 3. **Unnamed pattern** – erlaubt es, den Typ und den Namen einer Komponente eines Record Patterns vollständig wegzulassen (und durch ein `_` zu ersetzen).



- **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;
```

- Interessanterweise können auch mehrere unbenannte Variablen im selben Scope verwendet werden, was (neben einfachen Lambdas) vor allem für Record Patterns und in switch von Interesse ist.

Unnamed Variables & Patterns



```
String userInput = "E605";
try
{
    processInput(Integer.parseInt(userInput));
}
catch (NumberFormatException █) // UNNAMED VARIABLE
{
    System.out.println("Expected number, but was: '" + userInput + "'");
}

var cp = new ColoredPoint(new Point(1234, 567), Color.BLUE);

if (cp instanceof ColoredPoint(Point(int x,
                                         var █), // UNNAMED PATTERN VARIABLE
                                         █)) UNNAMED PATTERN
{
    System.out.println("x: " + x);
}
```



Platz 7: Launch Multi-File Source-Code Programs



JEP 458: Launch Multi-File Source-Code Programs



- Bereits seit Java 11 LTS existiert das sogenannte Direct Compilation, mit dem sich einzelne Java-Dateien direkt ohne explizites vorheriges Kompilieren ausführen lassen.
- Das ist für kleinere Experimente sowie einfache Kommandozeilentools recht nützlich.
- Als Einschränkung galt bis einschließlich Java 21 LTS, dass man mit Direct Compilation lediglich eine einzelne Java-Datei ausführen konnte.
- Je größer die Programme werden, desto mehr kommt der Wunsch auf, Funktionalitäten in verschiedene Klassen in eigener Java-Dateien zu untergliedern. Das wird bislang für Direct Compilation nicht unterstützt.

JEP 458: Launch Multi-File Source-Code Programs



```
package jep458_Launch_MultiFile_SourceCode_Programs;

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

```
package jep458_Launch_MultiFile_SourceCode_Programs;

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

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

JEP 458: Launch Multi-File Source-Code Programs



```
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);
    }
}
```

```
package jep458_Launch_MultiFile_SourceCode_Programs;
```

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

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



Platz 8: String Templates (Preview)



String Interpolation



- Als Alternative zur Stringkonkatenation existieren in vielen Programmiersprachen String-Interpolationen oder formulierte Strings als Text mit speziellen Platzhaltern:

- Python `f"Calculation: {x} + {y} = {x + y}"`
- Kotlin `"Calculation: $x + $y = ${x + y}"`
- Swift: `"Calculation: \(x) + \(y) = \(x + y)"`
- C# `$"Calculation: {x} + {y}= {x + y}"`

- String-Templates ergänzen die bisherigen Varianten um eine elegante Möglichkeit, Ausdrücke zu spezifizieren, die zur Laufzeit ausgewertet und entsprechend in den String integriert werden.

```
System.out.println(STR."Calculation: \{x} plus \{y} equals \{x + y}");
```

- STR ist ein sogenannter String-Prozessor, der in Kombination mit Platzhaltern, die als `\{varName}` angegeben sind, ein Resultat erzeugt und die Werte in den String einfügt.

String Templates



- **Beispiel aus alt mach neu**

```
System.out.println("color: " + color + " ==> " + num0fChars);
```

- =>

```
System.out.println(STR."color: \{color} ==> \{num0fChars}");
```

- **Beispiel**

```
String firstName = "Michael";
String lastName = "Inden";
```

```
String firstLastName = STR."\{firstName} \{lastName}";
String lastFirstName = STR."\{lastName}, \{firstName}";
```

```
System.out.println(firstLastName);
System.out.println(lastFirstName);
```

Michael Inden
Inden, Michael

String Templates und Text Blocks



- **Beispiel**

```
int statusCode = 201;
var msg = "CREATED";

String json = STR.=====
{
    "statusCode": \{statusCode},
    "msg": "\{msg}"
}=====;
System.out.println(json);
```

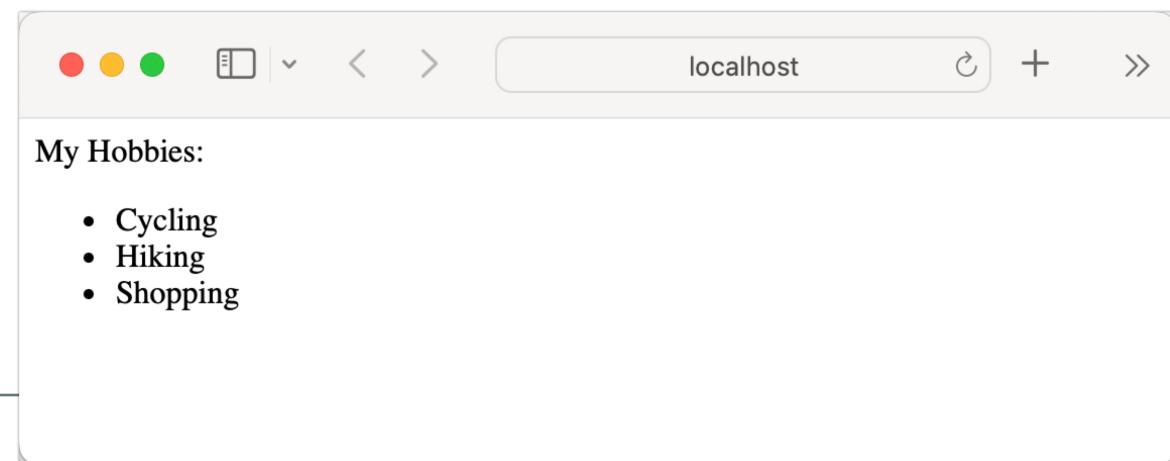
```
{
    "statusCode": 201,
    "msg": "CREATED"
}
```

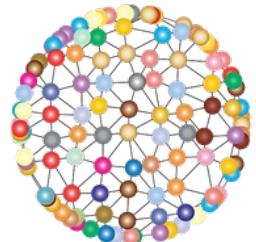
String Templates und Text Blocks



```
String title = "My First Web Page";
String text = "My Hobbies:";  
var hobbies = List.of("Cycling", "Hiking",
"Shopping");
String html = STR.!!!!  
    <html>
        <head><title>\{title}</title>
        </head>
        <body>
            <p>\{text}</p>
            <ul>
                <li>\{hobbies.get(0)}</li>
                <li>\{hobbies.get(1)}</li>
                <li>\{hobbies.get(2)}</li>
            </ul>
        </body>
    </html>"";
System.out.println(html);
```

```
<html>
    <head><title>My First Web Page</title>
    </head>
    <body>
        <p>My Hobbies:</p>
        <ul>
            <li>Cycling</li>
            <li>Hiking</li>
            <li>Shopping</li>
        </ul>
    </body>
</html>
```





**String Templates sind ja schon
ganz nett, aber wo spielen sie
ihre Vorteile wirklich aus?**

String Templates – In der Praxis ... Ein Blick zurück



- Generierung einer mehrzeiligen Bestellbestätigung, die vordefinierte Textbausteine mit den Angaben aus den Parametern kombiniert und für die Produktinformationen eine Methode `getProductFor()` aufruft:

```
private static void printOrderConfirmationOldStyle(long orderId, long productId,
                                                int quantity, LocalDate orderDate)
{
    System.out.println("Your order id is " + orderId + "\n" +
                       "On " + orderDate + " you ordered\n" +
                       "product '" + getProductFor(productId) + "'\n" +
                       "quantity " + quantity);
}
```

- Konkatenation kann noch einigermaßen gut nachvollziehen, allerdings sind die Kombinationen aus Anführungszeichen und + sowie Zeilentrennern "\n" eher unübersichtlich
- Beim Aufbereiten schleichen sich schnell kleinere Fehler wie vergessene Leerzeichen ein.
- Im besten Fall schwierig lesbar, im schlechtesten Fall drohen Inkonsistenzen und Verwirrung.
- Tendiert dazu, mit zunehmender Anzahl an Bestandteilen immer unübersichtlicher zu werden

String Templates – In der Praxis ... Ein Blick zurück



- Um die Lesbarkeit und Nachvollziehbarkeit zu erhöhen, könnte man statt der Stringkonkatenation mit + bevorzugt Text Blocks mit Platzhaltern in Kombination mit `formatted()` nutzen:

```
private static void printOrderConfirmationOldStyleV2(long orderId, long productId,
                                                    int quantity, LocalDate orderDate)
{
    System.out.println("""
        Your order id is %d
        On %tF you ordered
        product '%s'
        quantity %d""").formatted(orderId, orderDate,
                                    getProductName(productId), quantity));
}
```

- Ist deutlich übersichtlicher
- Art der Platzhalter nicht in jedem Fall gebräuchlich, %s und %d schon, aber %tF eher nicht
- Auch hier gilt, je mehr Platzhalter zu ersetzen sind, desto ungünstiger ist es mit der Zuordnung von Werten zu Platzhaltern.

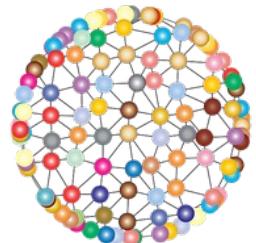
String Templates – In der Praxis ... Abhilfe



- Bei der Zusammenstellung der mehrzeiligen Bestellbestätigung profitiert man nun von Text Blocks und, dass diese mit String Templates kombiniert werden können. Damit ergibt sich folgende Vereinfachung in der Implementierung:

```
private static void printOrderConfirmationNewStyle(int orderId, int productId,  
                                                int quantity, LocalDate orderDate)  
{  
    System.out.println(STR.|||||  
                        Your order id is \{orderId}\n                        On \{orderDate} you ordered  
                        product '\\" \{getProductFor(productId)}'\n                        quantity \{quantity}\\"|||||);  
}
```

- Ist etwas kürzer als zuvor und auch übersichtlicher
- Platzhalter lassen sich direkt erkennen und zuordnen
- Auch wenn mehr Platzhalter zu ersetzen sind, bleibt die Übersichtlichkeit erhalten



**Was kann man sonst noch so mit
String Templates machen?**

String Templates – Alternative String Processors



```
private static void alternativeStringProcessors() { import static java.lang.StringTemplate.STR;
    int x = 47; import static java.util.FormatProcessor.FMT;
    int y = 11; String calculation1 = FMT.%6d\{x} + %6d\{y} = %6d\{x + y}";
    System.out.println("fmt calculation 1: " +calculation1);

    float base = 3.0f;
    float addon = 0.1415f;

    String calculation2 = FMT.%2.4f\{base} + %2.4f\{addon} = %2.4f\{base + addon}";
    System.out.println("fmt calculation 2 " + calculation2);

    String calculation3 = FMT.Math.PI * 1.000 = %4.6f\{Math.PI * 1000}";
    System.out.println("fmt calculation 3 " + calculation3);
}
```

```
fmt calculation 1: 47 + 11 = 58
fmt calculation 2: 3.0000 + 0.1415 = 3.1415
fmt calculation 3: Math.PI * 1.000 = 3141.592654
```



Platz 9: Stream Gatherers (Preview)





- Nehmen wir an, wir wollten alle Duplikate aus einem Stream herausfiltern und dazu ein Kriterium angeben:

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

- Mit einem Trick kann man das herkömmlich wie folgt lösen:

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



- Ein weiteres Beispiel ist die Gruppierung der Daten eines Streams in Abschnitte fixer Größe. Als Beispiel sollen jeweils vier Zahlen zu einer Einheit zusammengefasst/gruppiert werden, wobei nur die ersten drei Gruppen ins Ergebnis aufgenommen werden sollen.

```
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]]
```

- Im Laufe der Jahre sind diverse Intermediate Operations wie etwa `distinctBy()` oder `windowFixed()` als Ergänzung für das Stream-API vorgeschlagen worden.
- Oftmals sind diese in spezifischen Kontexten sinnvoll, allerdings würden diese das Stream-API ziemlich aufblähen und den Einstieg in das (ohnehin schon umfangreiche) API (weiter) erschweren.

JEP 461: Stream Gatherers – windowFixed()



- Um einen Stream in kleinere Bestandteile fixer Größe ohne Überlappung zu unterteilen, dient `windowFixed()`.

```
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);  
}
```

- Teilweise enthält der Datenbestand nicht genug Elemente. Das führt dazu, dass der letzte Teilbereich dann einfach weniger Elemente beinhaltet.

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

JEP 461: Stream Gatherers – windowSliding()



- Um einen Stream in kleinere Bestandteile fixer Größe mit Überlappung zu unterteilen, dient `windowSliding()`.

```
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);  
}
```

- Teilweise enthält der Datenbestand nicht genug Elemente. Das führt dazu, dass der letzte Teilbereich dann einfach weniger Elemente beinhaltet.

```
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 461: Stream Gatherers – fold()



- Um die Werte eines Streams miteinander zu verknüpfen, dient die Methode `fold()`. Ähnlich wie bei `reduce()` gibt man einen Startwert und eine Berechnungsvorschrift an:

```
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);
}
```

- Um einen Wert auszulesen, dient wiederum der Aufruf von `findFirst()`, das liefert einen `Optional<T>`:

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



- Was passiert, wenn wir zur Kombination der Werte auch Aktionen ausführen wollen, die nicht für die Typen der Werte, hier int, definiert sind?
- Als Beispiel wird ein Zahlenwert in einen String gewandelt und dieser gemäß dem Zahlenwert mit repeat() wiederholt:

```
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);
}
```

- Als Ausgabe ergibt sich die Folgende:

```
repeat with fold(): [122333444455555666667777777]
```



- Sollen die Elemente eines Streams zu neuen Kombinationen zusammengeführt werden, sodass jeweils immer ein Element dazukommt, dann ist dies das Aufgabe von `scan()`.
- Die Methode arbeitet ähnlich wie `fold()`, das die Werte zu einem Ergebnis kombiniert. Bei `scan()` wird dagegen bei jeder Kombination der Werte ein neues Ergebnis produziert:

```
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);
}
```

- Die Ausgabe ist Folgende:

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



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



Implicitly Declared Classes and Instance Main Methods



- Vielleicht ist es auch bei Ihnen schon eine Weile her, dass Sie Java gelernt haben.
- Wenn Sie Programmieranfängern Java beibringen wollen, wissen Sie, wie schwierig der Einstieg ist.
- Aus der Sicht von Anfängern besitzt Java eine wirklich steile Lernkurve.
- Es fängt schon mit dem einfachsten Hello-World an.

```
package preview;

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

- Mit Python reduziert auf das Wesentliche:

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

Sie als Trainer weisen auf die folgenden Fakten für Anfänger hin:

1. Vergessen Sie package, public , class, static, void, etc. die sind momentan noch unwichtig ...
2. Schauen Sie sich nur die Zeile mit System.out.println() an
3. Oh ja, System.out ist eine Instanz einer Klasse, aber auch das ist jetzt nicht wichtig.

Ziemlich viele verwirrende Wörter und Konzepte, die von der eigentlichen Aufgabe ablenken.

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



Weitere Möglichkeiten

```
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 <---
```



**Selbst ist der Mann oder die Frau ...
probier es aus :-)**

<https://github.com/Michaeli71/Best-Of-Modern-Java-21-22-My-Favorite-Features>





Fazit



Positives



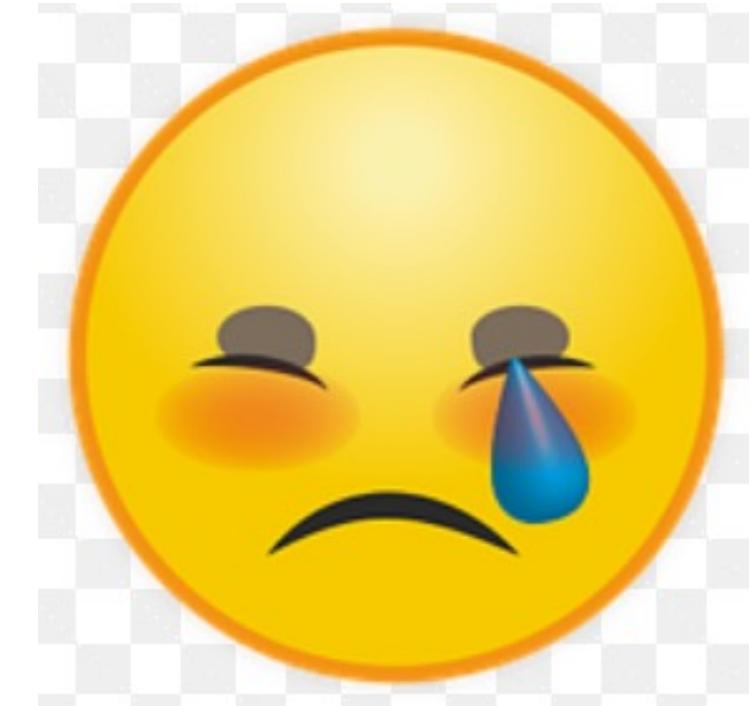
- **Stabile und zuverlässige 6-monatige Release-Zyklen und alle 2 Jahre LTS-Versionen**
- **Java wird einfacher und attraktiver**
- **Viele schöne Verbesserungen in Syntax und APIs wie switch, Records, Text Blocks**
- **Pattern Matching und Record Patterns final in Java 21**
- **Virtuelle Threads & Structured Concurrency**

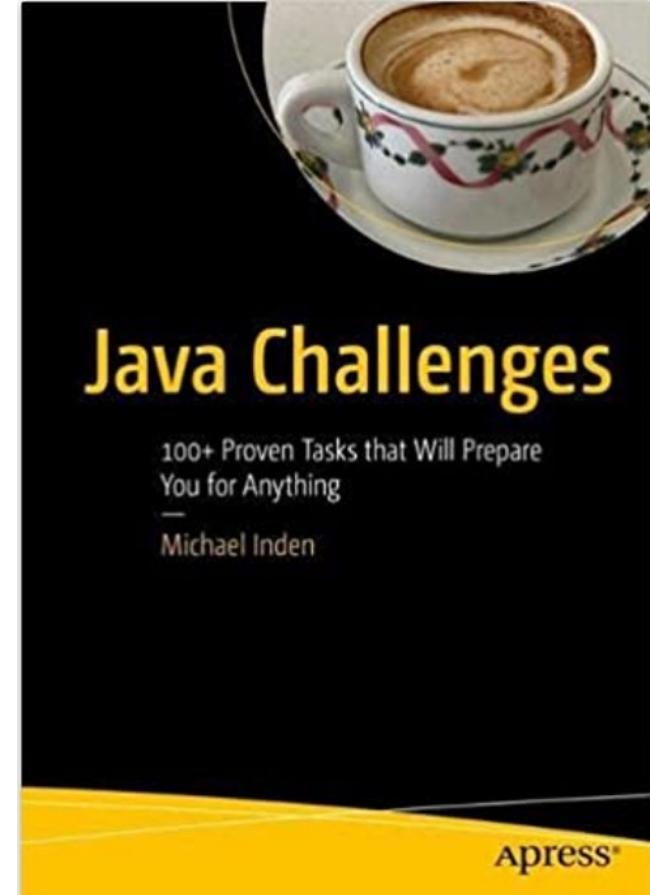


Negatives



- Releases waren zwar pünktlich, aber manchmal (außer Java 14) ziemlich dünn bezüglich wichtiger Neuerungen, manchmal sogar nur Preview Features
- Java 21 LTS enthält viele unfertige Dinge ... meiner Meinung nach sollte ein LTS nur wenige Previews und möglichst keine Incubators enthalten
- Wir müssen leider noch 2 Jahre warten, bis die netten Unnamed Classes und Instance Main Methods sowie Unnamed Patterns and Variables für den Gebrauch im nächsten LTS-Release verfügbar sind
- Warum ist die Syntax von Pattern Matching bei instanceof und switch inkonsistent?







Questions?



Thank You