



Coole neue Java Features – Java 17 bis 23

<https://github.com/Michaeli71/Best-Of-Java-17-23>



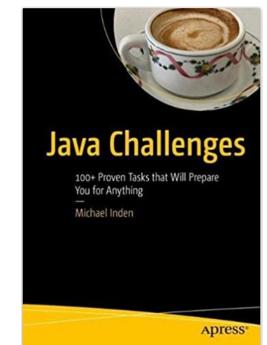
Michael Inden
Head of Development, freiberuflicher Buchautor und Trainer

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 bei dpunkt.verlag, O'Reilly und APress**

E-Mail: michael_inden@hotmail.com



<https://github.com/Michaeli71/Best-Of-Java-17-23>



Agenda

Zeitplan

9:00 – 10:30	Teil 1
10:30 – 11:00	KAFFEE
11:00 – 12:30	Teil 2
12:30 – 13:30	MITTAG
13:30 – 15:00	Teil 3
15:00 – 15:30	KAFFEE
15:30 – 17:00	Teil 4

Workshop Contents



- **Vorbemerkungen / Build Tools & IDEs**
- **PART 1:** Syntax-Erweiterungen bis Java 17 LTS
- **PART 2:** API- und JVM-Neuheiten und Änderungen bis Java 17 LTS
- **PART 3:** Neuheiten in Java 18 bis 21 LTS
- **PART 4:** Neuheiten in Java 22 und 23

Separat: Modularisierung im Kurzüberblick



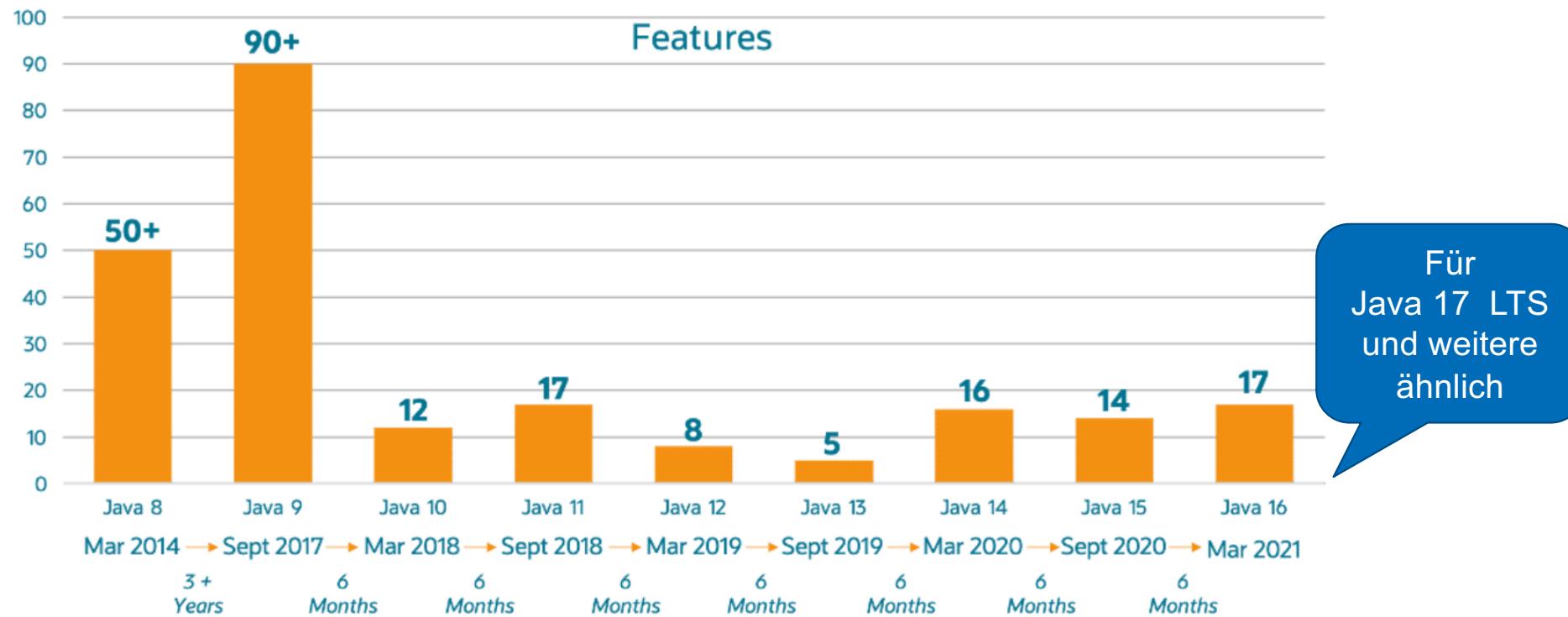
<https://github.com/Michaeli71/Best-Of-Java-17-23>

Long-Term Support-Modell



- **alle drei -> zwei Jahre Long Term Support (LTS) Release**
 - erhalten über längere Zeit Updates
 - Produktionsversionen
 - derzeit Java 8 LTS, 11 LTS und 17 LTS sowie neu Java 21 LTS
 - aktuelles LTS-Release ist Java 21 (September 2023)
 - **Seit Java 17 wieder ALLE 2 Jahre UND FOR FREE ☺ (mit spezieller Einschränkung)**
- **andere Versionen sind "nur" Zwischenversionen**
 - erhalten nur 6 Monate Updates
 - Previews – inkludiert Features, die noch nicht fertiggestellt sind, um Feedback zu erhalten
 - Ideal um neue Features kennenzulernen und zum Experimentieren (vor allem privat)
 - Incubators – noch rudimentärer als Previews, sind noch im Stadium, wo sie ggf. komplett wieder aufgegeben werden

Einordnung des 6-monatigen Releasezyklus'





Build-Tools und IDEs



Aktuelles Java 21 LTS installiert



- Laden Sie die **neueste Version von Java 21 LTS herunter**

```
$ java --version
java 21.0.4 2024-07-16 LTS
Java(TM) SE Runtime Environment (build 21.0.4+8-LTS-274)
Java HotSpot(TM) 64-Bit Server VM (build 21.0.4+8-LTS-274, mixed mode, sharing)
```

- **Aktivierung von Preview-Features nötig beim Arbeiten auf der Konsole**

```
% java --enable-preview --source 21 src/main/java/HelloJava21.java
Hello Java 21
```

```
import javax.lang.model.SourceVersion;

public class HelloJava21 {
    public static void main(String[] args) {
        System.out.println("Hello Java " +
                           SourceVersion.RELEASE_21.runtimeVersion());
    }
}
```

IDE & Tool Support für Java 21 LTS



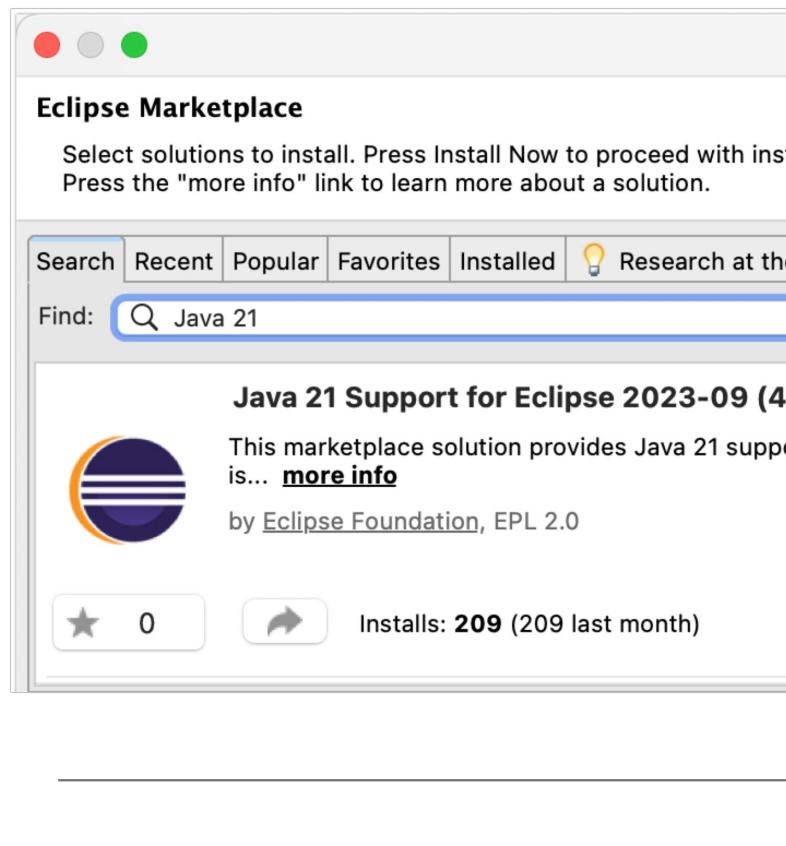
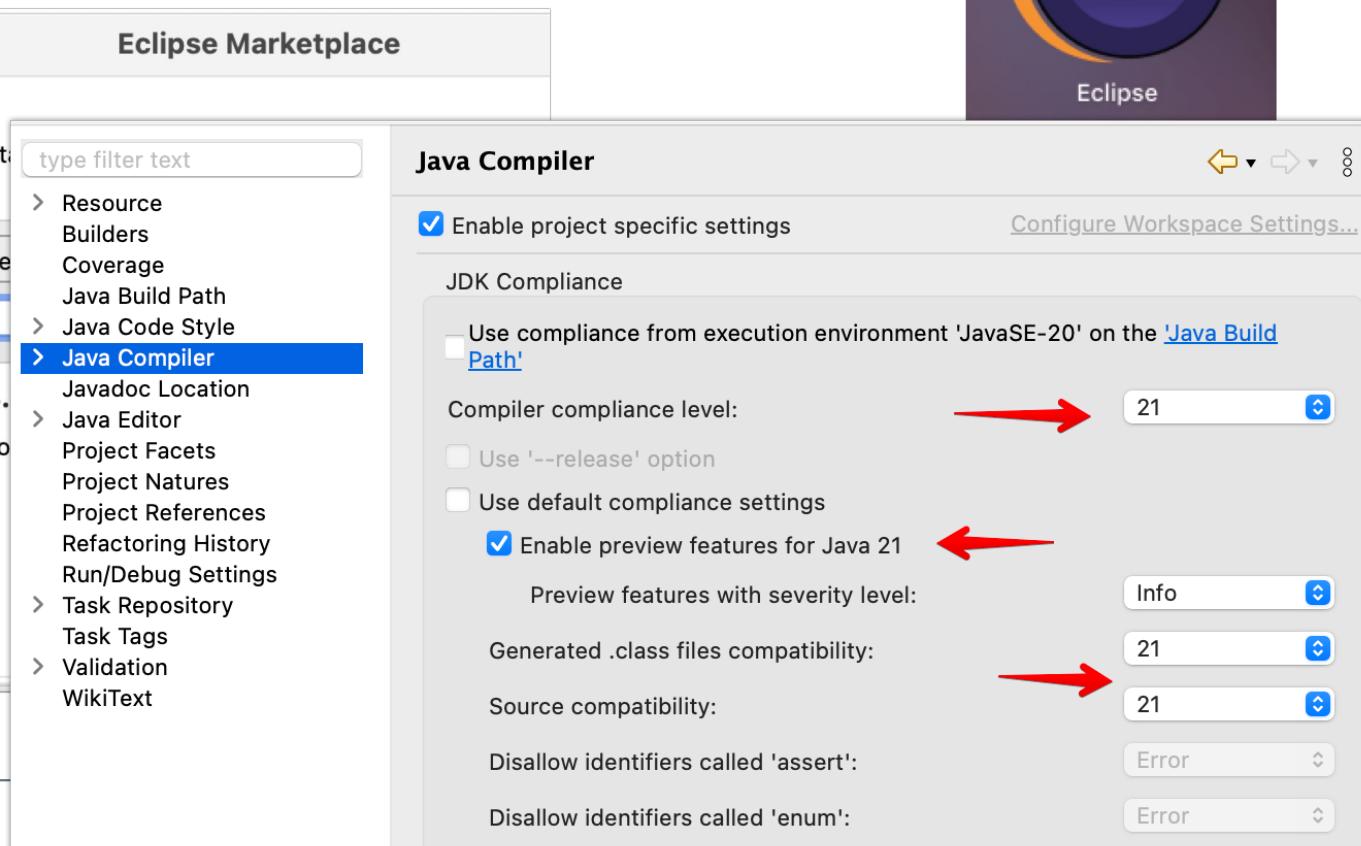
- Eclipse: Version 2023-12 (nicht alle Previews supported)
- IntelliJ: Version 2023.3.x
- Maven: 3.9.6, Compiler Plugin: 3.11.0
- Gradle: 8.5
- Aktivierung von Preview-Features / Incubator nötig
 - In Dialogen
 - In Build Scripts



IDE & Tool Support Java 21 LTS



- Eclipse 2023-09 mit Plugin / Eclipse 2023-12 ohne Plugin
- Aktivierung von Preview-Features nötig

The screenshot shows the Eclipse Marketplace interface. A search bar at the top contains the text "Java 21". Below it, a card for "Java 21 Support for Eclipse 2023-09 (4.1.0)" is displayed, stating "This marketplace solution provides Java 21 support is... more info" and "by Eclipse Foundation, EPL 2.0". It shows 0 installs and was last updated 209 installs last month.The screenshot shows the Java Compiler settings in the Eclipse preferences. A red arrow points to the "Compiler compliance level" dropdown set to "21". Another red arrow points to the "Enable preview features for Java 21" checkbox, which is checked. Red arrows also point to the "Generated .class files compatibility" and "Source compatibility" dropdowns, both set to "21".

IDE & Tool Support



- Aktivierung von Preview-Features nötig



Project Structure

Project

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

Name: Java21Examples

SDK: 21 java version "21"

Language level: 21 (Preview) - String templates, unnamed classes and instance main methods etc.

Compiler output:

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

IDE & Tool Support Java 21 LTS



- Aktivierung von Preview-Features / Incubator nötig



```
sourceCompatibility=21  
targetCompatibility=21
```

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

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

IDE & Tool Support Java 21 LTS



- 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>21</source>
      <target>21</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>
```





PART 1: Syntax-Erweiterungen bis Java 17 LTS

- Switch Expressions
 - Text Blocks
 - Records
 - Syntaxerweiterung bei instanceof
-



Switch Expression



Switch Expressions



- **switch-case-Konstrukt noch bei den uralten Wurzeln aus der Anfangszeit von Java**
 - **Kompromisse im Sprachdesign, die C++-Entwicklern den Umstieg erleichtern sollten.**
 - **Relikte wie das break und beim Fehlen des solchen das Fall-Through**
 - **Flüchtigkeitsfehler kamen immer wieder vor**
 - **Zudem war man beim case recht eingeschränkt bei der Angabe der Werte.**
 - **Das alles ändert sich glücklicherweise mit Java 13. Dazu wird die Syntax leicht verändert und erlaubt nun die Angabe einer Expression sowie mehrerer Werte beim case:**
-

Switch Expressions: Blick zurück



- Abbildung von Wochentagen auf deren Länge ...

```
DayOfWeek day = DayOfWeek.FRIDAY;  
int numOfLetters = -1;  
  
switch (day)  
{  
    case MONDAY:  
    case FRIDAY:  
    case SUNDAY:  
        numOfLetters = 6;  
        break;  
    case TUESDAY:  
        numOfLetters = 7;  
        break;  
    case THURSDAY:  
    case SATURDAY:  
        numOfLetters = 8;  
        break;  
    case WEDNESDAY:  
        numOfLetters = 9;  
        break;  
}
```

Switch Expressions als Abhilfe



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

```
DayOfWeek day = DayOfWeek.FRIDAY;  
int numLetters = -1;  
  
switch (day)  
{  
    case MONDAY, FRIDAY, SUNDAY -> numLetters = 6;  
    case TUESDAY -> numLetters = 7;  
    case THURSDAY, SATURDAY -> numLetters = 8;  
    case WEDNESDAY -> numLetters = 9;  
}
```

Switch Expressions als Abhilfe



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

Return-
Value

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

- 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: Blick zurück ... Fallstricke



- Abbildung von Monaten auf deren Namen ...

```
Month month = Month.APRIL;
```

```
// ACHTUNG: Mitunter ganz übler Fehler: default mitten zwischen den cases
String monthName = "";
switch (month)
{
    case JANUARY:
        monthName = "January";
        break;
    default:
        monthName = "N/A"; // hier auch noch Fall Through
    case FEBRUARY:
        monthName = "February";
        break;
    case MARCH:
        monthName = "March";
        break;
    case JULY:
        monthName = "July";
        break;
}
```

```
System.out.println("OLD: " + month + " = " + monthName); // February
```

Switch Expressions als Abhilfe



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

```
static String monthToString(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: switch-old Fallstrick mit break



- Gegeben sei eine Aufzählung von Farben:

```
enum Color { RED, GREEN, BLUE, YELLOW, ORANGE }
```

- Bilden wir diese auf die Anzahl an Buchstaben ab:

```
Color color = Color.GREEN;
int numChars;

switch (color)
{
    case RED: numChars = 3; break;
    case GREEN: numChars = 5; /* break; UPS: FALL-THROUGH */;
    case YELLOW: numChars = 6; break;
    case ORANGE: numChars = 6; break;
    default: numChars = -1;
}
```

Switch Expressions: `yield` mit Rückgabe



Mit modernem Java wird wieder alles sehr klar und einfach:

```
public static void switchYieldReturnsValue(Color color)
{
    int num0fChars = switch (color)
    {
        case RED: yield 3;
        case GREEN: yield 5;
        case YELLOW, ORANGE: yield 6;
        default: yield -1;
    };
    throw new IllegalArgumentException("Unexpected color: " + color);
    System.out.println("color: " + color + " ==> " + num0fChars);
}
```

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



- **langersehnte Erweiterung, nämlich mehrzeilige Strings ohne mühselige Verknüpfungen definieren zu können und auf fehlerträchtiges Escaping zu verzichten.**
- **Erleichtert unter anderem den Umgang mit**
 - der Definition von JavaScript in Java-Sourcecode
 - SQL-Anweisungen
 - kurzen XML- und JSON-Fragmenten
- **ALT**

```
String javaScriptCodeOld = "function hello() {\n" +  
    "    print(\"Hello World\");\n" +  
    "}\n" +  
    "\n" +  
    "hello();\n";
```

Text Blocks



- NEU

```
String javascriptCode = """  
    function hello()  
    {  
        print("Hello World");  
    }  
  
    hello();  
""";
```

```
String multiLineString = """  
THIS IS  
A MULTI  
LINE STRING  
WITH A BACKSLASH \\  
""";
```

Text Blocks



Traditional String Literals

```
String html = "<html>\n" +  
    "    <body>\n" +  
    "        <p>Hello World.</p>\n" +  
    "    </body>\n" +  
"</html>\n";
```

```
String multiLineHtml = """  
    <html>  
        <body>  
            <p>Hello, world</p>  
        </body>  
    </html>  
""";
```

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

Text Blocks



- NEU

```
String multiLineSQL = """
    SELECT `ID`, `LAST_NAME` FROM `CUSTOMER`
    WHERE `CITY` = 'ZÜRICH'
    ORDER BY `LAST_NAME`;
""";
```

```
String multiLineStringWithPlaceHolders = """
    SELECT %s
    FROM %
    WHERE %
""".formatted("A", "B", "C");
```

Text Blocks



- NEU

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

Text Blocks (Alignment)



```
jshell> var multiLine = """"  
...>     One  
...>     Two  
...>     Three"""  
multiLine ==> "One\n Two\n Three"
```

- Kein Zeilenumbruch am Ende
- Leerzeichen vor dem ersten Zeichen werden NICHT berücksichtigt
- Leerzeichen am Anfang werden entfernt

```
jshell> var multiLine = """"  
...>     - One  
...>     - Two  
...>     - Three  
...>     - Four  
...>  
multiLine ==> " _ One\n _ Two\n _ Three\n_ _ Four\n"
```

- Zeilenumbruch am Ende
- Leerzeichen vor dem ersten Zeichen werden berücksichtigt
- Leerzeichen am Anfang

- Alle Leerzeichen am Ende werden automatisch entfernt

Besonderheit bei Text Blocks



```
String text = """  
    This is a string split \  
    into several smaller \  
    strings.\ \  
    """;  
  
System.out.println(text);
```

This is a string split into several smaller strings.



Records





**Wäre es nicht cool, auf
einfache Weise DTOs usw.
zu definieren?**

Erweiterung Record



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

- simplifizierte Form von Klassen für einfache, unveränderliche* Datencontainer
- Sehr kurze, kompakte Schreibweise
- API ergibt sich implizit aus den als Konstruktorparameter definierten Attributen

```
MyPoint point = new MyPoint(47, 11);
System.out.println("point x:" + point.x());
System.out.println("point y:" + point.y());
```

- Implementierungen von Accessor-Methoden, equals() und hashCode() automatisch und vor allem kontraktkonform sowie toString() mit allen Attributangaben
-

Erweiterung Record

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



Was Sie bekommen ...

```
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 und zusätzliche Konstruktoren und Methoden



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

    public String shortForm()
    {
        return "[" + x + ", " + y + "]";
    }
}
```

```
jshell> var topLeft = new MyPoint("23, 11")
topLeft ==> MyPoint[x=23, y=11]

jshell> System.out.println(topLeft);
MyPoint[x=23, y=11]

jshell> System.out.println(topLeft.shortForm());
[23, 11]
```

Records für DTO / Parameter Value Objects



```
record TopLeftWidthAndHeight(MyPoint topLeft, int width, int height) { }

record ColorAndRgbDTO(String name, int red, int green, int blue) { }

record PersonDTO(String firstname, String lastname, LocalDate birthday) { }

record Rectangle(int x, int y, int width, int height)
{
    Rectangle(TopLeftWidthAndHeight pointAndDimension)
    {
        this(pointAndDimension.topLeft().x(), pointAndDimension.topLeft().y(),
              pointAndDimension.width(), pointAndDimension.height());
    }
}
```

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()
{
    // einige komplexe Dinge
    return new IntStringReturnValue(42, "the answer");
}
```

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

Records für Tupel? – Ausflug Pair<T>



- Was ist an diesem self made Pair falsch?

```
static class Pair<T>
{
    public T first;
    public T second;

    public Pair(T first, T second)
    {
        this.first = first;
        this.second = second;
    }

    @Override
    public String toString()
    {
        return "Pair [first=" + first + ", second=" + second + "]";
    }
}
```



**Was fehlt da eigentlich?
Was stört da vielleicht?**

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()
-



Ist ja cool ... ABER:
Wie kann ich denn
Gültigkeitsprüfungen
integrieren?

Records mit Gültigkeitsprüfung



```
record ClosedInterval(int lower, int upper)
{
    public ClosedInterval(int lower, int upper)
    {
        if (lower >= upper)
        {
            String errorMsg = String.format("invalid: %d (lower) >= %d (upper)",
                                             lower, upper);
            throw new IllegalArgumentException(errorMsg);
        }

        this.lower = lower;
        this.upper = upper;
    }
}
```

Records mit Gültigkeitsprüfung (Kurzschreibweise)



```
record ClosedInterval(int lower, int upper)
{
    public ClosedInterval
    {
        if (lower >= upper)
        {
            String errorMsg = String.format("invalid: %d (lower) >= %d (upper)",
                                             lower, upper);
            throw new IllegalArgumentException(errorMsg);
        }
    }
}
```



**(Vorerst) letzte Frage für
Records:
Ist das alles kombinierbar?**

All in Beispiel



```
record MultiTypes<K, V, T>(Map<K, V> mapping, T info)
{
    public void printTypes()
    {
        System.out.println("mapping type: " + mapping.getClass());
        System.out.println("info type: " + info.getClass());
    }
}

record ListRestrictions<T>(List<T> values, int maxSize)
{
    public ListRestrictions
    {
        if (values.size() > maxSize)
            throw new IllegalArgumentException(
                "too many entries! got: " + values.size() +
                ", but restricted to " + maxSize);
    }
}
```



Records Beyond the Basics



Speicherung in verschiedenen Sets



- Definieren wir mal einen einfachen Record und zwei verschiedene Datenspeicherungen:

```
record SimplePerson(String name, int age, String city) {}

Set<SimplePerson> speakers = new HashSet<>();
speakers.add(new SimplePerson("Michael", 51, "Zürich"));
speakers.add(new SimplePerson("Michael", 51, "Zürich"));
speakers.add(new SimplePerson("Anton", 42, "Aachen"));

System.out.print(speakers);

Set<SimplePerson> sortedSpeakers = new TreeSet<>();
sortedSpeakers.add(new SimplePerson("Michael", 51, "Zürich"));
sortedSpeakers.add(new SimplePerson("Michael", 51, "Zürich"));
sortedSpeakers.add(new SimplePerson("Anton", 42, "Aachen"));

System.out.print(sortedSpeakers);
```



Das sollte das Ergebnis sein, oder?

[SimplePerson[name=Michael, age=51, city=Zürich],
SimplePerson[name=Anton, age=42, city=Aachen]]

[SimplePerson[name=Anton, age=42, city=Aachen],
SimplePerson[name=Michael, age=51, city=Zürich]]

Speicherung in verschiedenen Sets



```
[SimplePerson[name=Michael, age=51, city=Zürich], SimplePerson[name=Anton, age=42, city=Aachen]]
```

```
Exception in thread "main" java.lang.ClassCastException: class
```

b_slides.RecordInterfaceExample\$1SimplePerson cannot be cast to class java.lang.Comparable

(b_slides.RecordInterfaceExample\$1SimplePerson is in unnamed module of loader 'app';
java.lang.Comparable is in module java.base of loader 'bootstrap')

at java.base/java.util.TreeMap.compare(TreeMap.java:1569)

at java.base/java.util.TreeMap.addEntryToEmptyMap(TreeMap.java:776)

at java.base/java.util.TreeMap.put(TreeMap.java:785)

at java.base/java.util.TreeMap.put(TreeMap.java:534)

at java.base/java.util.TreeSet.add(TreeSet.java:255)

at b_slides.RecordInterfaceExample.main(RecordInterfaceExample.java:32)

Records und Interfaces



- Korrigieren wir die Definition des einfachen Records und implementieren ein Interface:

```
record SimplePerson2(String name, int age, String city)
    implements Comparable<SimplePerson2>
{
    @Override
    public int compareTo(SimplePerson2 other)
    {
        return name.compareTo(other.name);
    }
}

Set<SimplePerson2> sortedSpeakers = new TreeSet<>();
sortedSpeakers.add(new SimplePerson2("Michael", 51, "Zürich"));
sortedSpeakers.add(new SimplePerson2("Michael", 51, "Zürich"));
sortedSpeakers.add(new SimplePerson2("Anton", 42, "Aachen"));
System.out.println(sortedSpeakers);
```

```
[SimplePerson2[name=Anton, age=42, city=Aachen],
 SimplePerson2[name=Michael, age=51, city=Zürich]]
```



**Was passiert, wenn wir
mehrere Datensätze haben,
die sich nicht nur im
Namen unterscheiden?**

Records und Interfaces + Comparator



- Korrigieren wir die Definition des Comparators:

```
record SimplePerson3(String name, int age, String city)
    implements Comparable<SimplePerson3>
{
    static Comparator<SimplePerson3> byAllAttributes = Comparator.
        comparing(SimplePerson3::name).
        thenComparingInt(SimplePerson3::age).
        thenComparing(SimplePerson3::city);

    @Override
    public int compareTo(SimplePerson3 other)
    {
        return byAllAttributes.compare(this, other);
    }
}
```



Builder ...



Builder like



```
record SimplePerson(String name, int age, String city)
{
    SimplePerson(String name)
    {
        this(name, 0, "");
    }

    SimplePerson(String name, int age)
    {
        this(name, age, "");
    }

    SimplePerson withAge(int newAge)
    {
        return new SimplePerson(name, newAge, city);
    }

    SimplePerson withCity(String newCity)
    {
        return new SimplePerson(name, age, newCity);
    }
}
```

Builder like

.



- **Problemfeld, viele Attribute**

```
record ComplexPerson(String firstname, String surname, LocalDate birthday,  
                     int height, int weight, String addressStreet,  
                     String addressNumber, String city)  
{  
    public static void main(String[] args)  
    {  
        ComplexPerson john = new ComplexPerson("John", "Smith", LocalDate.of(2010, 6, 21),  
                                              170, 90, "Fuldastrasse", "16a", "Berlin");  
  
        ComplexPerson mike = new ComplexPersonBuilder().withFirstname("Mike").  
                                         withBirthday(LocalDate.of(2021, 1, 21)).  
                                         withSurname("Peters").build();  
        System.out.println(mike);  
    }  
}
```

- **Aber: ComplexPersonBuilder müsste man selbst implementieren => 😞**



**Was wäre eine weitere
Möglichkeit zur
Komplexitätsreduktion?**

Builder like

.



- **Records für einzelne Bestandteile definieren**

```
record Address(String addressStreet, String addressNumber, String city) {}

record BodyInfo(int height, int weight) {}

record PersonDTO(String firstname, String surname, LocalDate birthday) {}

record ReducedComplexPerson(PersonDTO person, BodyInfo bodyInfo, Address address)
{
    public static void main(String[] args)
    {
        var john = new PersonDTO("John", "Smith", LocalDate.of(2010, 6, 21));
        var bodyInfo = new BodyInfo(170, 90);
        var address = new Address("Fuldastrasse", "16a", "Berlin");

        var rcp = new ReducedComplexPerson(john, bodyInfo, address);
    }
}
```



Date Range ... Immutability

A large, stylized text graphic featuring the date "12.12". The digits are rendered in a bold, white font with a thick red outline. They are set against a solid black rectangular background, which has some subtle texture or noise visible at the edges.

Record für Datumsbereich



```
public class RecordImmutabilityExample
{
    public static void main(String[] args)
    {
        record DateRange(Date start, Date end) {}

        DateRange range1 = new DateRange(new Date(71,1,7), new Date(71,2,27));
        System.out.println(range1);

        DateRange range2 = new DateRange(new Date(71,6,7), new Date(71,2,27));
        System.out.println(range2);
    }
}
```

```
DateRange[start=Sun Feb 07 00:00:00 CET 1971, end=Sat Mar 27 00:00:00 CET 1971]
DateRange[start=Wed Jul 07 00:00:00 CET 1971, end=Sat Mar 27 00:00:00 CET 1971]
```

- Gültigkeitsprüfung für Invariante `start < end` einbauen

Record für Datumsbereich



```
record DateRange(Date start, Date end)
{
    DateRange
    {
        if (!start.before(end))
            throw new IllegalArgumentException("start >= end");
    }
}

DateRange range1 = new DateRange(new Date(71,1,7), new Date(71,2,27));
System.out.println(range1);

DateRange range2 = new DateRange(new Date(71,6,7), new Date(71,2,27));
System.out.println(range2);

DateRange[start=Sun Feb 07 00:00:00 CET 1971, end=Sat Mar 27 00:00:00 CET 1971]
Exception in thread "main" java.lang.IllegalArgumentException: start >= end
at b_slides.RecordImmutabilityExample3$1DateRange.<init>(RecordImmutabilityExample3.java:21)
at b_slides.RecordImmutabilityExample3.main(RecordImmutabilityExample3.java:28)
```

Record für Datumsbereich



```
record DateRange(Date start, Date end)
{
    DateRange
    {
        if (!start.before(end))
            throw new IllegalArgumentException("start >= end");
    }
}

DateRange range1 = new DateRange(new Date(71,1,7), new Date(71,2,27));
System.out.println(range1);

range1.start.setTime(new Date(71,6,7).getTime());
System.out.println(range1);

DateRange[start=Sun Feb 07 00:00:00 CET 1971, end=Sat Mar 27 00:00:00 CET 1971]
DateRange[start=Wed Jul 07 00:00:00 CET 1971, end=Sat Mar 27 00:00:00 CET 1971]
```

- **Immutability nur für Immutable-Attribute => ACHTUNG: Referenzsemantik in Java**

Record für Datumsbereich



```
record DateRange(LocalDate start, LocalDate end)
{
    DateRange
    {
        if (!start.isBefore(end))
            throw new IllegalArgumentException("start >= end");
    }
}

DateRange range1 = new DateRange(LocalDate.of(1971,1,7), LocalDate.of(1971,2,27));
System.out.println(range1);

DateRange range2 = new DateRange(LocalDate.of(1971,6,7), LocalDate.of(1971,2,27));
System.out.println(range2);

DateRange[start=1971-01-07, end=1971-02-27]
Exception in thread "main" java.lang.IllegalArgumentException: start >= end
at b_slides.RecordImmutabilityExample4$1DateRange.<init>(RecordImmutabilityExample4.java:21)
at b_slides.RecordImmutabilityExample4.main(RecordImmutabilityExample4.java:28)
```

Records



DEMO & Hands on



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...
}
```



**Immer diese Casts...
Geht es nicht einfacher?**

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



```
Object obj2 = "Hallo Java 14";
```

```
if (obj2 instanceof String str)
{
    // Hier kann man str nutzen
    System.out.println("Länge: " + str.length());
}
else
{
    // Hier kein Zugriff auf str
    System.out.println(obj2.getClass());
}
```

Pattern Matching bei instanceof



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



Übungen PART 1

<https://github.com/Michaeli71/Best-Of-Java-17-23>



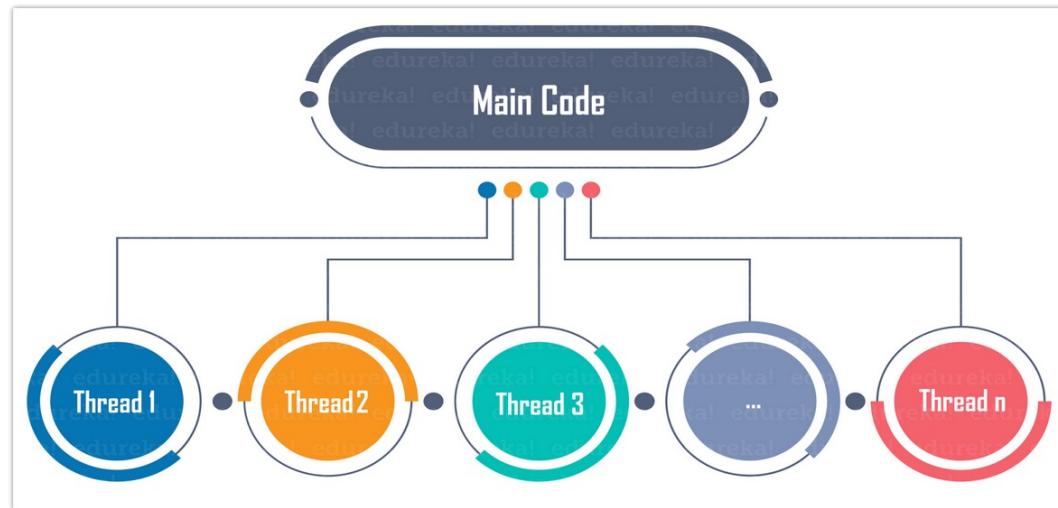


PART 2: API- und JVM- Neuerungen und Änderungen bis Java 17 LTS

- Multi-Threading mit `CompletableFuture`
 - HTTP/2
 - `Stream.toList()`
 - Direct Compilation
 - Hilfreiche `NullPointerExceptions`
-



Multi-Threading mit CompletableFuture



Multi-Threading und die Klasse CompletableFuture<T>



- Seit Java 5 gibt es im JDK das Interface Future<T> , führt aber durch seine Methode get() oft zu blockierendem Code
 - Seit JDK 8 hilft CompletableFuture<T> bei der Definition asynchroner Berechnungen
 - Abläufe beschreiben, parallele Ausführungen ermöglichen
 - Aktionen auf höherer semantischer Ebene als mit Runnable oder Callable<T>
-

Einstieg CompletableFuture<T>



- **Basisschritte**

- `supplyAsync(Supplier<T>)` => Berechnung definieren
- `thenApply(Function<T,R>)` => Ergebnis der Berechnung verarbeiten
- `thenAccept(Consumer<T>)` => Ergebnis verarbeiten, aber ohne Rückgabe
- `thenCombine(...)` => Verarbeitungsschritte zusammenführen

- **Beispiel**

```
CompletableFuture<String> firstTask = CompletableFuture.supplyAsync(() -> "First");  
CompletableFuture<String> secondTask = CompletableFuture.supplyAsync(() -> "Second");
```

```
CompletableFuture<String> combined = firstTask.thenCombine(secondTask,  
                                (f, s) -> f + " " + s);
```

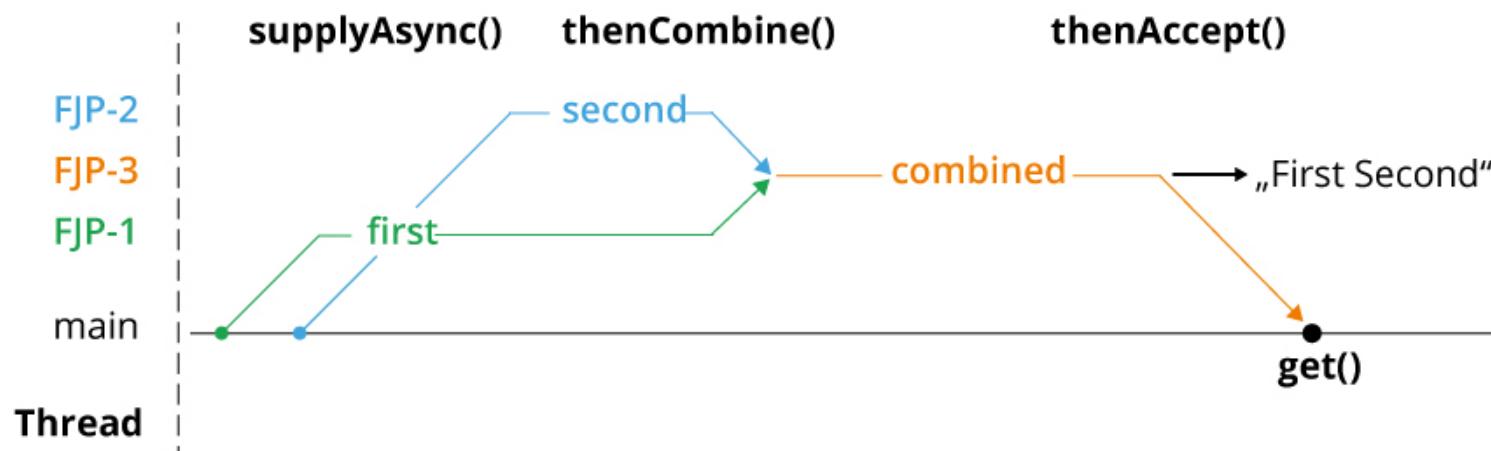
```
combined.thenAccept(System.out::println);  
System.out.println(combined.get());
```

Einführung CompletableFuture<T>



```
CompletableFuture<String> firstTask = CompletableFuture.supplyAsync(() -> "First");
CompletableFuture<String> secondTask = CompletableFuture.supplyAsync(() -> "Second");

CompletableFuture<String> combined = firstTask.thenCombine(secondTask, (f, s) -> f + " " + s);
combined.thenAccept(System.out::println);
System.out.println(combined.get());
```



Multi-Threading und die Klasse CompletableFuture<T>

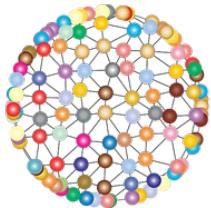


Beispiel: Es sollen folgende Aktionen stattfinden:

- **Daten vom Server lesen**
 - **Auswertung 1 berechnen**
 - **Auswertung 2 berechnen**
 - **Auswertung 3 berechnen**
 - **Ergebnisse in Form eines Dashboards zusammenführen**
-



**Wie könnte eine erste
Realisierung aussehen?**



Multi-Threading und die Klasse CompletableFuture<T>



- Daten vom Server lesen => retrieveData()
- Auswertung 1 berechnen => processData1()
- Auswertung 2 berechnen => processData2()
- Auswertung 3 berechnen => processData3()
- Ergebnisse in Form eines Dashboards zusammenführen => calcResult()
- Vereinfachungen: Daten => Liste von Strings, Berechnungen => Ergebnis long => String

```
public void execute()
{
    final List<String> data = retrieveData();
    final Long value1 = processData1(data);
    final Long value2 = processData2(data);
    final Long value3 = processData3(data);
    final String result = calcResult(value1, value2, value3);
    System.out.println("result: " + result);
}
```

Multi-Threading und die Klasse CompletableFuture<T>



```
public void execute()
{
    final List<String> data = retrieveData();
    final Long value1 = processData1(data);
    final Long value2 = processData2(data);
    final Long value3 = processData3(data);
    final String result = calcResult(value1, value2, value3);
    System.out.println("result: " + result);
}
```

Die Berechnungen des Beispiels sind allerdings sehr vereinfacht ...

- **keine Parallelität**
- **keine Abstimmung der Aufgaben (funktioniert, weil synchron)**
- **kein Exception-Handling**

- **Wir ersparen uns die Mühen und kaum verständliche und unwartbare Varianten mit Runnable, Callable<T>, Thread-Pools, ExecutorService usw., weil das selbst damit oft doch noch kompliziert und fehlerträchtig ist**



**Was muss man ändern
für eine parallele
Verarbeitung mit
CompletableFuture<T>?**

Multi-Threading und die Klasse CompletableFuture<T>



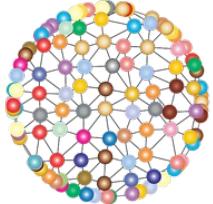
```
public void execute() throws InterruptedException, ExecutionException
{
    final CompletableFuture<List<String>> cfData =
        CompletableFuture.supplyAsync(()->retrieveData());
    // Zwischenstand ausgeben
    cfData.thenAccept(System.out::println);

    // Auswertungen parallel ausführen
    final CompletableFuture<Long> cFValue1 = cfData.thenApplyAsync(data -> processData1(data));
    final CompletableFuture<Long> cFValue2 = cfData.thenApplyAsync(data -> processData2(data));
    final CompletableFuture<Long> cFValue3 = cfData.thenApplyAsync(data -> processData3(data));

    // Einzelergebnisse als Result zusammenführen
    final String result = calcResult(cFValue1.get(), cFValue2.get(), cFValue3.get());
    System.out.println("result: " + result);
}
```



**Wie bilden wir Fehler
der realen Welt ab?**



Multi-Threading und die Klasse CompletableFuture<T>



- In der realen Welt kommt es mitunter zu Exceptions
- Treten Fehler ab und an auf: Netzwerkprobleme, Zugriff aufs Dateisystem usw.
- **Annahme: Während der Datenermittlung würde eine IllegalStateException ausgelöst:**

```
Exception in thread "main" java.util.concurrent.ExecutionException:  
java.lang.IllegalStateException: retrieveData() failed  
at java.base/java.util.concurrent.CompletableFuture.reportGet(CompletableFuture.java:395)  
at java.base/java.util.concurrent.CompletableFuture.get(CompletableFuture.java:1999)
```

- **Folglich würde die gesamte Verarbeitung unterbrochen und gestört!**
 - Wünschenswert ist eine einfache Integration des Exception Handlings in den Ablauf
 - Sowie weniger komplizierte Behandlung als bei Thread-Pools oder ExecutorService
-

Multi-Threading und die Klasse CompletableFuture<T>



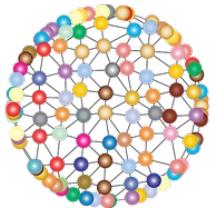
- Die Klasse **CompletableFuture<T>** bietet die Methode **exceptionally()**
- Fallback-Wert bereitstellen, wenn die Daten nicht ermittelt werden können:

```
final CompletableFuture<List<String>> cfData =  
    CompletableFuture.supplyAsync(() -> retrieveData()).  
    exceptionally((throwable) -> Collections.emptyList());
```

- Fallback-Wert bereitstellen, wenn eine Berechnung fehlschlägt:

```
final CompletableFuture<Long> cFValue2 =  
    cfData.thenApplyAsync(data -> processData2(data)).  
    exceptionally((throwable) -> 0L);
```

- Berechnung kann selbst dann noch fortgesetzt werden, wenn der eine oder andere Schritt fehlschlagen sollte



Wie bilden Verzögerungen der realen Welt ab?

Multi-Threading und die Klasse CompletableFuture<T>



- In der realen Welt kommt es bei Zugriffen auf externe Ressourcen manchmal zu Verzögerungen
 - Im schlimmsten Fall antwortet ein externer Partner auch gar nicht und man würde ggf. unendlich lange warten, wenn ein Aufruf blockierend erfolgt
 - **Wünschenswert:** Berechnungen sollten mit Time-out abgebrochen werden können
 - **Annahme:** Die Datenermittlung `retrieveData()` benötigt mitunter einige Sekunden
 - **Folglich würde die gesamte Verarbeitung gestört!**
-

Multi-Threading und die Klasse CompletableFuture<T>



Seit JDK 9: Die Klasse CompletableFuture<T> bietet die Methoden

- `public CompletableFuture<T> orTimeout(long timeout, TimeUnit unit)`
=> Die Verarbeitung wird mit einer Exception abgebrochen, falls das Ergebnis nicht innerhalb der angegebenen Zeitspanne berechnet wurde.
- `public CompletableFuture<T> completeOnTimeout(T value, long timeout, TimeUnit unit)`
=> Falls das Ergebnis nicht innerhalb der angegebenen Zeitspanne berechnet wurde, kann ein Defaultwert vorgegeben werden.

Multi-Threading und die Klasse CompletableFuture<T>



Annahme: Die Datenermittlung dauert mitunter mehrere Sekunden, soll aber nach spätestens 1 Sekunde abgebrochen werden:

```
final CompletableFuture<List<String>> cfData =  
    CompletableFuture.supplyAsync(()->retrieveData()).  
        orTimeout(1, TimeUnit.SECONDS).  
        exceptionally((throwable) -> Collections.emptyList());
```

=> Die Verarbeitung kann zeitnah (ohne viel Verzögerung oder gar Blockierung) selbst dann fortgesetzt werden, wenn die Datenermittlung länger dauern sollte

Multi-Threading und die Klasse CompletableFuture<T>



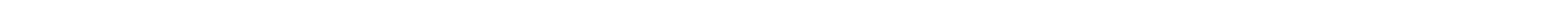
Annahme: Die Berechnung dauert mitunter mehrere Sekunden – diese soll spätestens nach 2 Sekunden abgebrochen werden und das Ergebnis 7 liefern:

```
final CompletableFuture<Long> cFValue3 =  
    cfData.thenApplyAsync(data -> processData3(data)).  
        completeOnTimeout(7L, 2, TimeUnit.SECONDS);
```

=> Die Berechnung kann mit einem Fallback-Wert fortgesetzt werden, selbst wenn der eine oder andere Schritt länger dauern sollte



HTTP/2 API



HTTP/2 API Intro



```
final URI uri = new URI("https://www.oracle.com");

final HttpRequest request = HttpRequest.newBuilder(uri).GET().build();
final HttpResponse.BodyHandler<String>asString =
    HttpResponse.BodyHandlers.ofString();

final HttpClient httpClient = HttpClient.newHttpClient();
final HttpResponse<String> response = httpClient.send(request, asString);

printResponseInfo(response);

static void printResponseInfo(final HttpResponse<String> response)
{
    final int responseCode = response.statusCode();
    final String responseBody = response.body();

    System.out.println("Status: " + responseCode);
    System.out.println("Body: " + responseBody);
}
```

HTTP/2 API Intro (var shines here)



```
var uri = new URI("https://www.oracle.com");

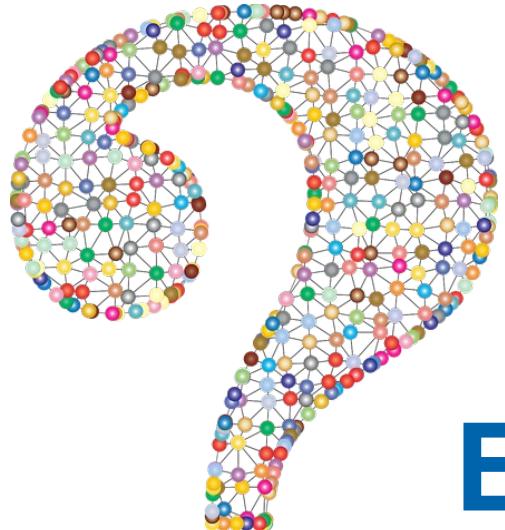
var request = HttpRequest.newBuilder(uri).GET().build();
var asString = HttpResponse.BodyHandlers.ofString();

var httpClient = HttpClient.newHttpClient();
var response = httpClient.send(request, asString);

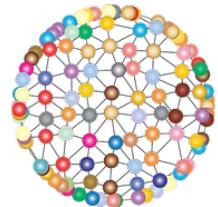
printResponseInfo(response);

static void printResponseInfo(final HttpResponse<String> response)
{
    var responseCode = response.statusCode();
    var responseBody = response.body();

    System.out.println("Status: " + responseCode);
    System.out.println("Body:    " + responseBody);
}
```



**Und nun kommt der PO:
Er hätte es gern asynchron!**



HTTP/2 API Async I



```
var uri = new URI("https://www.oracle.com/index.html");

var request = HttpRequest.newBuilder(uri).GET().build();
var asString = HttpResponse.BodyHandlers.ofString();

var httpClient = HttpClient.newHttpClient();
final CompletableFuture<HttpResponse<String>> asyncResponse =
        httpClient.sendAsync(request, asString);

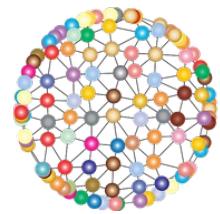
wait2secForCompletion(asyncResponse);
if (asyncResponse.isDone())
{
    printResponseInfo(asyncResponse.get());
}
else
{
    asyncResponse.cancel(true);
    System.err.println("timeout");
}
```

HTTP/2 API Async I – busy waiting mit vorzeitigem Abbruch



```
static void wait2secForCompletion(final CompletableFuture<?> asyncResponse)
    throws InterruptedException
{
    int i = 0;
    while (!asyncResponse.isDone() && i < 10)
    {
        System.out.println("Step " + i);
        i++;

        Thread.sleep(200);
    }
}
```



**Einen Moment bitte:
Ist das nicht old school?
Was können wir am Warten
verbessern?**

HTTP/2 API Async II



```
var uri = new URI("https://www.oracle.com/index.html");

var request = HttpRequest.newBuilder(uri).GET().build();
var asString = HttpResponse.BodyHandlers.ofString();
var httpClient = HttpClient.newHttpClient();
var asyncResponse = httpClient.sendAsync(request, asString);

// Warten und Verarbeitung: Variante rein mit CompletableFuture
asyncResponse.thenAccept(response -> printResponseInfo(response)).
    orTimeout(2, TimeUnit.SECONDS).
    exceptionally(ex ->
{
    asyncResponse.cancel(true);
    System.err.println("timeout");
    return null;
});

// CompletableFuture should have time to complete
Thread.sleep(2_000);
```



Java 16



Stream => List ... es war so umständlich ...



```
List<String> namesMi = Stream.of("Tim", "Tom", "Mike", "Michael").  
    filter(str -> str.startsWith("Mi")).  
collect(Collectors.toList());
```



**Wie häufig habt ihr bisher
toArray() aufgerufen?
Und wie oft
Collectors.toList()?**

1 : 1000 ?

FINALLY... `toList()`



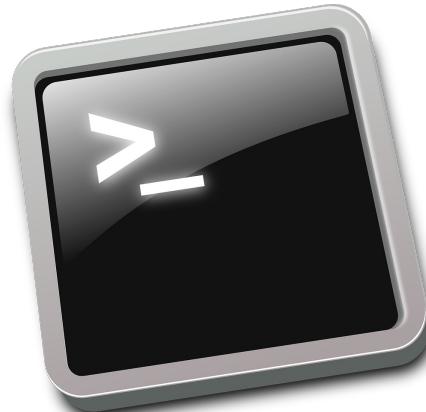
```
List<String> namesMi = Stream.of("Tim", "Tom", "Mike", "Michael").  
    filter(str -> str.startsWith("Mi")).  
    toList();
```



Direct Compilation

Launch Single-File Source-Code

Programs



Direct Compilation



- Erlaubt es, Java-Applikationen, die nur aus einer Datei bestehen, direkt in einem Rutsch zu kompilieren und auszuführen.
- erspart Arbeit und man muss nichts vom Bytecode und .class -Dateien wissen
- vor allem für die Ausführung von kleineren Java-Dateien als Skript und für den Einstieg in Java hilfreich

```
package direct.compilation;

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

java ./HelloWorld.java



Hello Execute After Compile

- Aber bis einschließlich Java 21 LTS nur für eine einzelne Java-Datei!

Direct Compilation – Zwei public Klassen in 1 Java-Datei möglich



```
public class PalindromeChecker
{
    public static void main(final String[] args)
    {
        if (args.length < 1)
        {
            System.err.println("input is required!");
            System.exit(1);
        }

        System.out.printf('%s is a palindrome? => %b %n',
                          args[0], StringUtils.isPalindrome(args[0]));
    }
}

public class StringUtils
{
    public static boolean isPalindrome(String word)
    {
        return new StringBuilder(word).reverse().toString().equalsIgnoreCase(word);
    }
}
```

Direct Compilation – Shebang Execution



```
#!/usr/bin/java --source 11
import java.nio.file.*;
public class DirectoryLister
{
    public static void main(String[] args) throws Exception
    {
        var dirName = ".";
        if (args == null || args.length < 1)
        {
            System.err.println("Using current directory as fallback");
        }
        else
        {
            dirName = args[0];
        }
        Files.walk(Paths.get(dirName)).forEach(System.out::println);
    }
}
```

- Datei darf nicht mit ‘.java’ enden
- Dateiname UNABHÄNGIG von Klassenname
- Datei muss executable (chmod +x) sein

Direct Compilation – Shebang Execution mit externer Abhängigkeit



```
#!/usr/bin/java --source 11 -cp ./guava-32.1.3-jre.jar

import java.nio.file.*;
import com.google.common.base.Joiner;

public class DirectoryLister
{
    public static void main(String[] args) throws Exception
    {
        var dirName = ".";
        if (args == null || args.length < 1)
        {
            var errorMsg = Joiner.on("++").join("Using current", "directory", "as fallback");
            System.err.println(errorMsg);
        }
        else
        {
            dirName = args[0];
        }

        Files.walk(Paths.get(dirName)).forEach(System.out::println);
    }
}
```



DEMO



N P E

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)
{
    try
    {
        final String[] stringArray = { null, null, null };
        final int errorPos = stringArray[2].lastIndexOf("ERROR");
    }
    catch (final NullPointerException e) { e.printStackTrace(); }

    try
    {
        final Integer value = null;
        final int sum = value + 3;
    }
    catch (final NullPointerException e) { e.printStackTrace(); }
}

java.lang.NullPointerException: Cannot invoke
"String.lastIndexOf(String)" because "stringArray[2]" is null
at java14.NPE_Second_Example.main(NPE_Second_Example.java:10)
java.lang.NullPointerException: Cannot invoke
"java.lang.Integer.intValue()" because "value" is null
at java14.NPE_Second_Example.main(NPE_Second_Example.java:20)
```

Don't do this in production!

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](#))

Hilfreiche NullPointerExceptions



```
public static WindowManager getWindowManager()
{
    return new WindowManager();
}

public static class WindowManager
{
    public Window getWindow(final int i)
    {
        return null;
    }
}

public static record Window(Size size) {}
public static record Size(int width, int height) {}
```



Übungen PART 2

<https://github.com/Michaeli71/Best-Of-Java-17-23>





PART 3: Neuerungen in Java 18 bis 21 LTS



JEPs in Java 18, 19, 20 und 21 LTS

Java 18 / 19 – Was ist drin?



- JEP 400: UTF-8 by Default
- JEP 408: Simple Web Server
- JEP 413: Code Snippets in Java API Documentation
- JEP 416: Reimplement Core Reflection with Method Handles
- JEP 417: Vector API (Third Incubator)
- JEP 418: Internet-Address Resolution SPI
- JEP 419: Foreign Function & Memory API (Second Incubator)
- **JEP 420: Pattern Matching for switch (Second Preview)**
- JEP 421: Deprecate Finalization for Removal

- **JEP 405: Record Patterns (Preview)**
- JEP 422: Linux/RISC-V Port
- JEP 424: Foreign Function & Memory API (Preview)
- JEP 425: Virtual Threads (Preview)
- JEP 426: Vector API (Fourth Incubator)
- **JEP 427: Pattern Matching for switch (Third Preview)**
- JEP 428: Structured Concurrency (Incubator)

18

19

Java 19 / 20 – Was ist drin?



- JEP 405: Record Patterns (Preview)
- JEP 422: Linux/RISC-V Port
- JEP 424: Foreign Function & Memory API (Preview)
- JEP 425: Virtual Threads (Preview)
- JEP 426: Vector API (Fourth Incubator)
- **JEP 427: Pattern Matching for switch (Third Preview)**
- JEP 428: Structured Concurrency (Incubator)

- JEP 429: Scoped Values (Incubator)
- **JEP 432: Record Patterns (Second Preview)**
- **JEP 433: Pattern Matching for switch (Fourth Preview)**
- JEP 434: Foreign Function & Memory API (Second Preview)
- JEP 436: Virtual Threads (Second Preview)
- JEP 437: Structured Concurrency (Second Incubator)
- JEP 438: Vector API (Fifth Incubator)

19

20

Java 21 LTS – Was ist drin?



- [JEP 430: String Templates \(Preview\)](#)
- [JEP 431: Sequenced Collections](#)
- JEP 439: Generational ZGC
- [JEP 440: Record Patterns](#)
- [JEP 441: Pattern Matching for switch](#)
- JEP 442: Foreign Function & Memory API (Third Preview)
- [JEP 443: Unnamed Patterns and Variables \(Preview\)](#)
- [JEP 444: Virtual Threads](#)
- [JEP 445: Unnamed Classes and Instance Main Methods \(Preview\)](#)
- JEP 446: Scoped Values (Preview)
- JEP 448: Vector API (Sixth Incubator)
- JEP 449: Deprecate the Windows 32-bit x86 Port for Removal
- JEP 451: Prepare to Disallow the Dynamic Loading of Agents
- JEP 452: Key Encapsulation Mechanism API
- [JEP 453: Structured Concurrency \(Preview\)](#)

Java 21
LTS

Total: 15

Normal: 8

Preview: 6

Incubator: 1

<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 22	DEV	API Notes	JDK JRE	21	20	19	18	17	16	15	14	13	12	11	...	
Java 21	LTS	API Lang VM Notes	JDK JRE	20	19	18	17	16	15	14	13	12	11	10	...	
Java 20	EOL	API Lang VM Notes	JDK JRE	19	18	17	16	15	14	13	12	11	10	9	...	
Java 19	EOL	API Lang VM Notes	JDK JRE	18	17	16	15	14	13	12	11	10	9	8	...	
Java 18	EOL	API Lang VM Notes	JDK JRE	17	16	15	14	13	12	11	10	9	8	7	...	
Java 17	LTS	API Lang VM Notes	JDK JRE	16	15	14	13	12	11	10	9	8	7	6	...	
Java 16	EOL	API Lang VM Notes	JDK JRE	15	14	13	12	11	10	9	8	7	6	5	...	
Java 15	EOL	API Lang VM Notes	JDK JRE	14	13	12	11	10	9	8	7	6	5	4	...	
Java 14	EOL	API Lang VM Notes	JDK JRE	13	12	11	10	9	8	7	6	5	4	3	...	
Java 13	EOL	API Lang VM Notes	JDK JRE	12	11	10	9	8	7	6	5	4	3	2	...	
Java 12	EOL	API Lang VM Notes	JDK JRE	11	10	9	8	7	6	5	4	3	2	1	...	
Java 11	LTS	API Lang VM Notes	JDK JRE	10	9	8	7	6	5	4	3	2	1	1	...	
Java 10	EOL	API Lang VM Notes	JDK JRE	9	8	7	6	5	4	3	2	1	1	1	...	
Java 9	EOL	API Lang VM Notes	JDK JRE	8	7	6	5	4	3	2	1	1	1	1	...	
Java 8	LTS	API Lang VM Notes	JDK JRE	7	6	5	4	3	2	1	1	1	1	1	...	
Java 7	EOL	API Lang VM Notes	JDK JRE	6	5	4	3	2	1	1	1	1	1	1	...	
Java 6	EOL	API Lang VM Notes	JDK JRE	5	4	3	2	1	1	1	1	1	1	1	...	
Java 5	EOL	API Lang VM Notes		4	3	2	1	1	1	1	1	1	1	1	...	
Java 1.4	EOL	API		3	2	1	1	1	1	1	1	1	1	1	1	...
Java 1.3	EOL	API		2	1	1	1	1	1	1	1	1	1	1	1	...
Java 1.2	EOL	API Lang		1	1	1	1	1	1	1	1	1	1	1	1	...
Java 1.1	EOL	API		1	1	1	1	1	1	1	1	1	1	1	1	...
Java 1.0	EOL	API Lang VM		1	1	1	1	1	1	1	1	1	1	1	1	...
Pre 1.0	EOL															...

Data Source

Info – <https://javaalmanac.io/>

Java 21

Status	In Development
Release Date	2023-09-15
EOL Date	2028-09
Class File Version	65.0
API Changes	Compare to 20 - 19 - 18 - 17 - 16 - 15 - 14 - 13 - 12 - 11 - 10 - 9 - 8 - 7 - 6 - 5 - 1.4 - 1.3 - 1.2 - 1.1
Documentation	Release Notes , JavaDoc
SCM	git
Data Source	

This will be the next LTS Release after [Java 17](#).

New Features

JVM

- Generational ZGC ([JEP 439](#))
- Deprecate the Windows 32-bit x86 Port for Removal ([JEP 449](#))
- Prepare to Disallow the Dynamic Loading of Agents ([JEP 451](#))

Language

- String Templates [1. Preview](#) ([JEP 430, Java Almanac](#))
- Record Patterns ([JEP 440, Java Almanac](#))
- Pattern Matching for switch ([JEP 441, Java Almanac](#))
- Unnamed Patterns and Variables [1. Preview](#) ([JEP 443](#))
- Unnamed Classes and Instance Main Methods [1. Preview](#) ([JEP 445, Java Almanac](#))

API

- Sequenced Collections ([JEP 431](#))
- Foreign Function & Memory API [3. Preview](#) ([JEP 442](#))
- Virtual Threads ([JEP 444, Java Almanac](#))
- Scoped Values [1. Preview](#) ([JEP 446](#))
- Vector API [6. Incubator](#) ([JEP 448](#))
- Key Encapsulation Mechanism API ([JEP 452](#))
- Structured Concurrency [1. Preview](#) ([JEP 453](#))

Sandbox – <https://javaalmanac.io/>



Sandbox

Instantly compile and run Java 21 snippets without a local Java installation.

Java21.java

▶ Run

21+35-2513

```
1 import java.lang.reflect.ClassFileFormatVersion;
2
3 public class Java21 {
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!

Java21.java

▶ Run

21+35-2513

Hello Java bytecode version 65!

Sandbox – <https://javaalmanac.io/>



```
public class Java21 {  
  
    public static void main(String[] args) {  
        multiMatch("Python");  
        multiMatch(null);  
        multiMatch(7);  
  
        record Person(String name, int age) {}  
        multiMatch(new Person("Michael", 52));  
    }  
  
    static void multiMatch(Object obj) {  
        switch (obj) {  
            case null -> System.out.println("null");  
            case String str when str.length() > 5 -> System.out.println(str.toUpperCase());  
            case String str -> System.out.println(str.toLowerCase());  
            case Integer i -> System.out.println(i * i);  
            default -> throw new IllegalArgumentException("Unsupported type " + obj.getClass());  
        }  
    }  
}
```

Sandbox – <https://javaalmanac.io/>



Java21.java ► Run 21+35-2513

```
1 public class Java21 {
2
3     public static void main(String[] args) {
4         multiMatch("Python");
5         multiMatch(null);
6         multiMatch(7);
7
8         record Person(String name, int age) {}
9         multiMatch(new Person("Michael", 52));
10    }
11
12    static void multiMatch(Object obj) {
13        switch (obj) {
14            case null -> System.out.println("null");
15            case String str when str.length() > 5 -> System.out.println(str.toUpperCase());
16            case String str -> System.out.println(str.toLowerCase());
17            case Integer i -> System.out.println(i * i);
18            default -> throw new IllegalArgumentException("Unsupported type " + obj.getClass());
19        }
20    }
21 }
```

Java21.java ► Run 21+35-2513

```
PYTHON
null
49
Exception in thread "main" java.lang.IllegalArgumentException: Unsupported type class Java21$1Person
at Java21.multiMatch(Java21.java:18)
at Java21.main(Java21.java:9)
```



Normalized Features in Java 21 LTS

- JEP 431: Sequenced Collections
 - JEP 440: Record Patterns
 - JEP 441: Pattern Matching for switch
 - JEP 444: Virtual Threads
-



JEP 431: Sequenced Collections

<https://openjdk.org/jeps/431>



Sequenced Collections



- Das Collections-API ist eines der ältesten und am besten konzipierten APIs im JDK.
 - Drei Haupttypen: `List<E>`, `Set<E>` und `Map<K, V>`
 - Was fehlt, ist so etwas wie Beschreibung einer geordneten Reihenfolge der Elemente
 - Wir beobachten aber, dass einige Sammlungen eine Begegnungsreihenfolge haben, d. h., es ist definiert, in welcher Reihenfolge die Elemente durchlaufen werden:
 - `List<E>` von vorne nach hinten, indexbasiert für Listen
 - `HashSet<E>` hat keine Begegnungsreihenfolge
 - `TreeSet<E>` definiert sie indirekt durch `Comparable<T>` oder übergebenen `Comparator<T>`
 - `LinkedHashSet<E>` behält die Einfügereihenfolge bei
-

Sequenced Collections – Motivation



- In der Vergangenheit gab es mehrere Möglichkeiten, auf das erste oder letzte Element zuzugreifen:

	First element	Last element
List	<code>list.get(0)</code>	<code>list.get(list.size() - 1)</code>
Deque	<code>deque.getFirst()</code>	<code>deque.getLast()</code>
SortedSet	<code>sortedSet.first()</code>	<code>sortedSet.last()</code>
LinkedHashSet	<code>linkedHashSet.iterator().next() // missing</code>	

- API-Unterschiede machen das Ganze unübersichtlich und fehleranfällig
- Als Abhilfe gibt es nun "Sequenced Collections", die eine Collection repräsentieren, deren Elemente eine bestimmte Reihenfolge haben.

Sequenced Collections – Motivation



- "Sequenced Collections" repräsentieren eine Collection, deren Elemente eine bestimmte Reihenfolge haben.

```
interface SequencedCollection<E> extends Collection<E> {  
    // new method  
    SequencedCollection<E> reversed();  
    // methods promoted from Deque  
    void addFirst(E);  
    void addLast(E);  
    E getFirst();  
    E getLast();  
    E removeFirst();  
    E removeLast();  
}
```

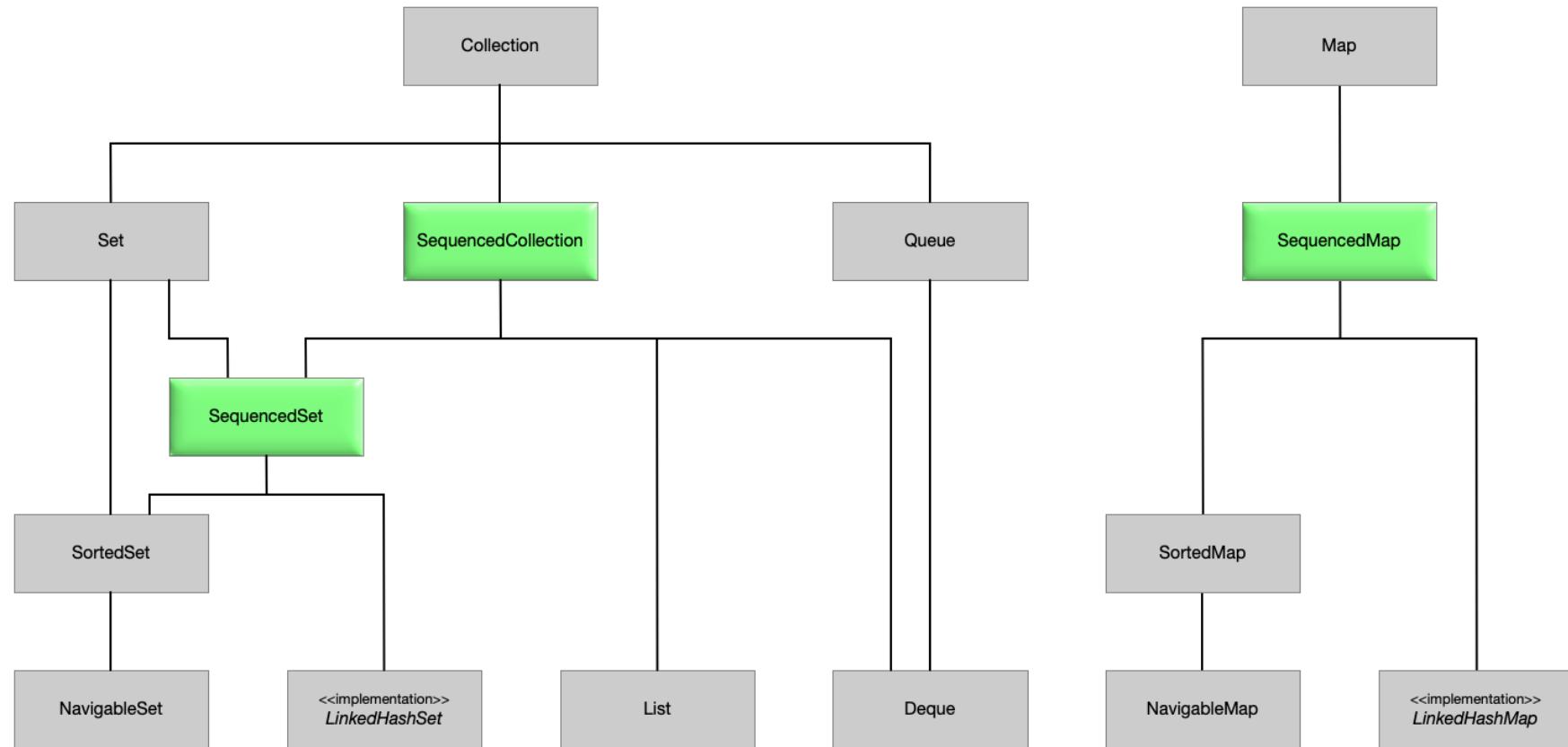
SequencedCollection defines the modifying methods:

- addFirst(E) – inserts an element at the beginning
- addLast(E) – appends an element to the end
- removeFirst() – removes the first element and returns it
- removeLast() – removes the last element and returns it

For immutable collections, all four methods throw an UnsupportedOperationException.

- Darüber hinaus bieten diese "Sequenced Collections" Methoden zum Hinzufügen, Ändern oder Löschen von Elementen am Anfang oder Ende der Collection.
- Außerdem ermöglichen sie die Verarbeitung der Elemente in umgekehrter Reihenfolge.

Sequenced Collections – Integration in bestehende Typenhierarchie



Sequenced Collections – Die Magie dahinter ... Default Methods



```
public interface SequencedCollection<E> extends Collection<E>
{
    SequencedCollection<E> reversed();

    default void addFirst(E e) {
        throw new UnsupportedOperationException();
    }

    default void addLast(E e) {
        throw new UnsupportedOperationException();
    }

    default E getFirst() {
        return this.iterator().next();
    }

    default E getLast() {
        return this.reversed().iterator().next();
    }

    ...
}

...
}

default E removeFirst() {
    var it = this.iterator();
    E e = it.next();
    it.remove();
    return e;
}

default E removeLast() {
    var it = this.reversed().iterator();
    E e = it.next();
    it.remove();
    return e;
}
```

Sequenced Collections – Sets und Maps



```
interface SequencedSet<E> extends Set<E>, SequencedCollection<E> {
    SequencedSet<E> reversed();      // covariant override
}

interface SequencedMap<K,V> extends Map<K,V> {
    // new methods
    SequencedMap<K,V> reversed();
    SequencedSet<K> sequencedKeySet();
    SequencedCollection<V> sequencedValues();
    SequencedSet<Entry<K,V>> sequencedEntrySet();
    V putFirst(K, V);
    V putLast(K, V);
    // methods promoted from NavigableMap
    Entry<K, V> firstEntry();
    Entry<K, V> lastEntry();
    Entry<K, V> pollFirstEntry();
    Entry<K, V> pollLastEntry();
}
```

Die SequencedMap-API fügt sich nicht so gut ein. Es verwendet NavigableMap als Basis, sodass es anstelle von getFirstEntry() firstEntry() anbietet und anstelle von removeLastEntry() pollLastEntry() definiert. Wie bereits erwähnt, entsprechen diese Namen nicht SequencedCollection. Aber der Versuch, dies zu tun, hätte dazu geführt, dass NavigableMap vier neue Methoden erhalten hätte, die dasselbe tun wie vier andere Methoden, die es bereits hat.

Sequenced Collection – In Aktion



```
public static void sequencedCollectionExample() {  
    System.out.println("Processing letterSequence with list");  
    SequencedCollection<String> letterSequence = List.of("A", "B", "C", "D", "E");  
    System.out.println(letterSequence.getFirst() + " / " +  
                       letterSequence.getLast());  
  
    System.out.println("Processing letterSequence in reverse order");  
    SequencedCollection<String> reversed = letterSequence.reversed();  
    reversed.forEach(System.out::print);  
    System.out.println();  
    System.out.println("reverse order stream skip 3");  
    reversed.stream().skip(3).forEach(System.out::print);  
    System.out.println();  
    System.out.println(reversed.getFirst() +  
                       " / " +  
                       reversed.getLast());  
    System.out.println();  
}
```

```
Processing letterSequence with list  
A / E  
Processing letterSequence in  
reverse order  
EDCBA  
reverse order stream skip 3  
BA  
E / A
```

Sequenced Collection – In Aktion



```
public static void sequencedSetExample() {  
    // Plain Sets haben keine Begegnungsreihenfolge ... mehrmals ausführen  
    System.out.println("Processing set of letters A–D");  
    Set.of("A", "B", "C", "D").forEach(System.out::print);  
    System.out.println();  
    System.out.println("Processing set of letters A–I");  
    Set.of("A", "B", "C", "D", "E", "F", "G", "H", "I").forEach(System.out::print);  
    System.out.println();  
  
    // TreeSet besitzt Reihenfolge  
    System.out.println("Processing letterSequence with tree set");  
    SequencedSet<String> sortedLetters = new TreeSet<>((Set.of("C", "B", "A", "D")));  
    System.out.println(sortedLetters.getFirst() + " / " + sortedLetters.getLast());  
    sortedLetters.reversed().forEach(System.out::print);  
    System.out.println();  
}
```

Processing set of letters A–D

DCBA

Processing set of letters A–I

IHGFEDCBA

Processing letterSequence with tree set

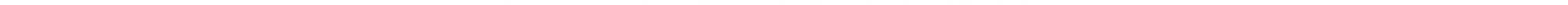
A / D

DCBA



JEP 440: Record Patterns

<https://openjdk.org/jeps/440>



JEP 440: Record Patterns



- Basis für diesen JEP und seine Vorgänger JEP 405 und JEP 432 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.

JEP 440: Record Patterns



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

- =>

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

JEP 440: Record Patterns



- Record Patterns 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);
    }
}
```

JEP 440: Record Patterns



- Record Patterns können für Eleganz sorgen:

```
record Person(String name, int age, Boolean hasDrivingLicense) { }

boolean isAllowedToDrive(Object obj) {
    if (obj instanceof Person person) {
        return person.age() >= 18 && person.hasDrivingLicense();
    }
    return false;
}

boolean isAllowedToDrive(Object obj) {
    if (obj instanceof Person(String name, int age, Boolean hasDrivingLicense)) {
        return age >= 18 && hasDrivingLicense;
    }
    return false;
}
```

- Bitte aber immer auch gutes OO-Design im Hinterkopf haben!



Wie kann man das Ganze noch eleganter gestalten?



JEP 440: Record Patterns



- Sauberes OO-Design würde die Methode im Record selbst definieren:

```
record Person(String name, int age, Boolean hasDrivingLicense) {  
    boolean isAllowedToDrive() {  
        return age(). >= 18 && hasDrivingLicense();  
    }  
}
```

In dem vorherigen Beispiel dient der Zugriff auf die Attribute lediglich dazu, das Pattern Matching zu illustrieren.



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



JEP 440: Record Patterns



- 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) {  
}
```

JEP 440: Record Patterns



- 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();
            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;
}
```

Jep405_FlighReservationExample.java

JEP 440: Record Patterns



- 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 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.



DEMO & Hands on

Jep405_FlighReservationExample.java



JEP 441: Pattern Matching bei switch

<https://openjdk.org/jeps/441>



JEP 441: Pattern Matching bei switch



- JEP 441 und seine Vorgänger JEP 420, JEP 427 und JEP 433 führen zu Änderungen bei der Auswertung der case innerhalb switch beim Kompilieren
 - zum einen ändert sich die sogenannte Dominanzprüfung,
 - zum anderen wurde die Vollständigkeitsanalyse korrigiert.
 - sowie in der Syntax bei der Angabe von Bedingungen mit when statt &&
 - die Unterstützung von Record Patterns
 - ein paar Besonderheiten
 - kein Support mehr für Klammerungen um Record Patterns:
`(if (obj instanceof (String s)))`
 - Support für qualifizierte Enum-Zugriffe
-

JEP 441: Pattern Matching bei switch: Dominanzprüfung



- **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                               -> System.out.println(s.toLowerCase());
        case Integer i                            -> System.out.println(i * 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: Dominanzprüfung



- Problematisch wird das Ganze, wenn die Reihenfolge der Patterns vertauscht wird:

```
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.out.println(i);
        default -> { }
    }
}
```

A screenshot of an IDE showing Java code. A tooltip is displayed over the line 'case String str && str.length() > 5 -> System.out.println(str.strip());'. The tooltip contains the message 'Label is dominated by a preceding case label 'String str''. It also suggests moving the branch before the 'String str' branch and shows a copyright notice for 'java.lang.String'.

- Die Dominanzprüfung deckt das Problem auf und führt seit Java 18 und «älterem» Java 17.0.6 zu einem Kompilierfehler, da das zweite case de facto unreachable code ist.
- Mit den ersten Java 17-Versionen gab das noch keinen Fehler!

JEP 441: Pattern Matching bei switch: Dominanzprüfung



- Probleme mit Konstanten (wurde mit Java 17 nicht entdeckt)

```
// Fehler im Eclipse-Compiler bei Dominance-Check, unreachable wird nicht erkannt
no usages

static void dominanceExampleWithConstant(Object obj) {
    switch (obj.toString()) {
        case String str when str.length() > 5 -> System.out.println(str.strip());
        case "Sophie" -> System.out.println("My lovely daughter");
        default -> Switch label '"Sophie"' is unreachable ...
    }
}
```

A tooltip appears over the 'case "Sophie"' branch, stating 'Switch label '"Sophie"' is unreachable' with a three-dot ellipsis icon. Below the tooltip are two buttons: 'Remove switch branch'"Sophie"' and 'More actions...'. A small icon of a person with a gear is also visible.

- Korrektur, sodass das speziellste Pattern ganz oben ist:

```
switch (obj.toString()) {
    case "Sophie" -> System.out.println("My lovely daughter");
    case String str when str.length() > 5 -> System.out.println(str.strip());
```

JEP 441: Pattern Matching bei switch: Dominanzprüfung



- Betrachten wir ein Beispiel zur Abfrage verschiedener Spezialfälle eines Integer:

- zunächst einige fixe Werte,
- dann den positiven Wertebereich und
- danach den verbliebenen Rest:

```
Integer value = calcValue();

switch (value) {
    case -1, 0, 1 -> System.out.println("Handle special cases -1, 0 or 1");
    case Integer i when i >= 1 ->
        System.out.println("Handle positive integer cases i > 1");
    case Integer i -> System.out.println("Handle all the remaining integers");
}
```

JEP 441: Pattern Matching bei switch: Dominanzprüfung



```
Integer value = calcValue();
switch (value) {
    case Integer i when i >= 1 -> System.out.println("Handle positive integer cases i > 1");
    case -1, 0, 1 -> System.out.println("Handle special cases -1, 0 or 1");
    case Integer i: Switch label '1' is unreachable
}
    : ining integers");
    Remove switch label '1' ⌂ More actions... ⌂
```

```
Integer value = calcValue();
switch (value) {
    case Integer i -> System.out.println("Handle all the remaining integers");
    case Integer i when i >= 1 -> System.out.println("Handle positive integer cases i > 1");
    case -1, 0, 1 -> System.out.println("Handle special cases -1, 0 or 1");
}
    : Label is dominated by a preceding case label 'Integer i'
    Move switch branch '1' before 'Integer i' ⌂ More actions... ⌂
```



**Was gilt es bei der
Dominanzprüfung zu beachten?**



JEP 441: Pattern Matching bei switch – Spezialfälle I



```
static void testTriangleAndString(Object obj)
{
    switch (obj)
    {
        case Triangle t when t.calculateArea() > 7000 ->
            System.out.println("Large triangle");
        case String str when str.startsWith("INFO") && str.contains("SPECIAL") ->
            System.out.println("a very special info");
        case String str when str.startsWith("INFO") ->
            System.out.println("just an info");
        // not detected ...
        case String str when str.startsWith("INFO") && str.contains("SPECIAL") ->
            System.out.println("a very special info");      DUPLIKATE IN DER ABFRAGE
        default ->
            System.out.println("Something else: " + obj);
    }
}
```

JEP 441: Pattern Matching bei switch – Spezialfälle II



```
static void testTriangleAndString(Object obj)
{
    switch (obj)
    {
        case Triangle t when t.calculateArea() > 7000 ->
            System.out.println("Large triangle");
        // case String str && str.startsWith("INFO") && str.contains("SPECIAL") ->
        //     System.out.println("a very special info");
        case String str when str.startsWith("INFO") ->
            System.out.println("just an info");
        // not detected ...
        case String str when str.startsWith("INFO") && str.contains("SPECIAL") ->
            System.out.println("a very special info");
        default ->
            System.out.println("Something else: " + obj);
    }
}
```

SPEZIALFALL IN DER ABFRAGE
=> DOMINANZ

JEP 441: Pattern Matching bei switch: Vollständigkeitsanalyse



- **Vollständigkeitsanalyse = Prüfung, ob alle möglichen Pfade von den cases im switch abgedeckt werden**
- **Java 18 bringt einen Bug Fix im Bereich der Vollständigkeitsanalyse, sodass folgender Sourcecode ohne Fehler kompiliert:**

```
static sealed abstract class BaseOp permits Add, Sub {}

static final class Add extends BaseOp {}

static final class Sub extends BaseOp {}

static void performAction(BaseOp op)
{
    switch (op) {
        case Add a -> System.out.println(a);
        case Sub s -> System.out.println(s);
    }
}
```



DEMO & Hands on

SwitchMultiPatternExample.java
SwitchSpecialCasesExample.java
SwitchDominanceExample.java
SwitchCompletenessExample.java

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");
    }
}
```

JEP 441: Pattern Matching bei switch: Typinferenz mit Patterns



- Mit Java 21 LTS wurde auch noch die Typinferenz verbessert und man kann auch in der Typangabe var verwenden:

```
static void recordInferenceJdk21(MyPair<String, Integer> pair)
{
    switch (pair)
    {
        case MyPair(var text, var count) when text.contains("Michael") ->
            System.out.println(text + " is " + count + " years old");
        case MyPair(var text, var count) when count > 5 && count < 10 ->
            System.out.println("repeated " + text.repeat(count));
        case MyPair(var text, var count) -> System.out.println(text + count);
        default -> System.out.println("NOT HANDLED");
    }
}
```



DEMO & Hands on

SwitchRecordPatternsExample.java
SwitchTypeInferenceExample.java



JEP 444: Virtual Threads

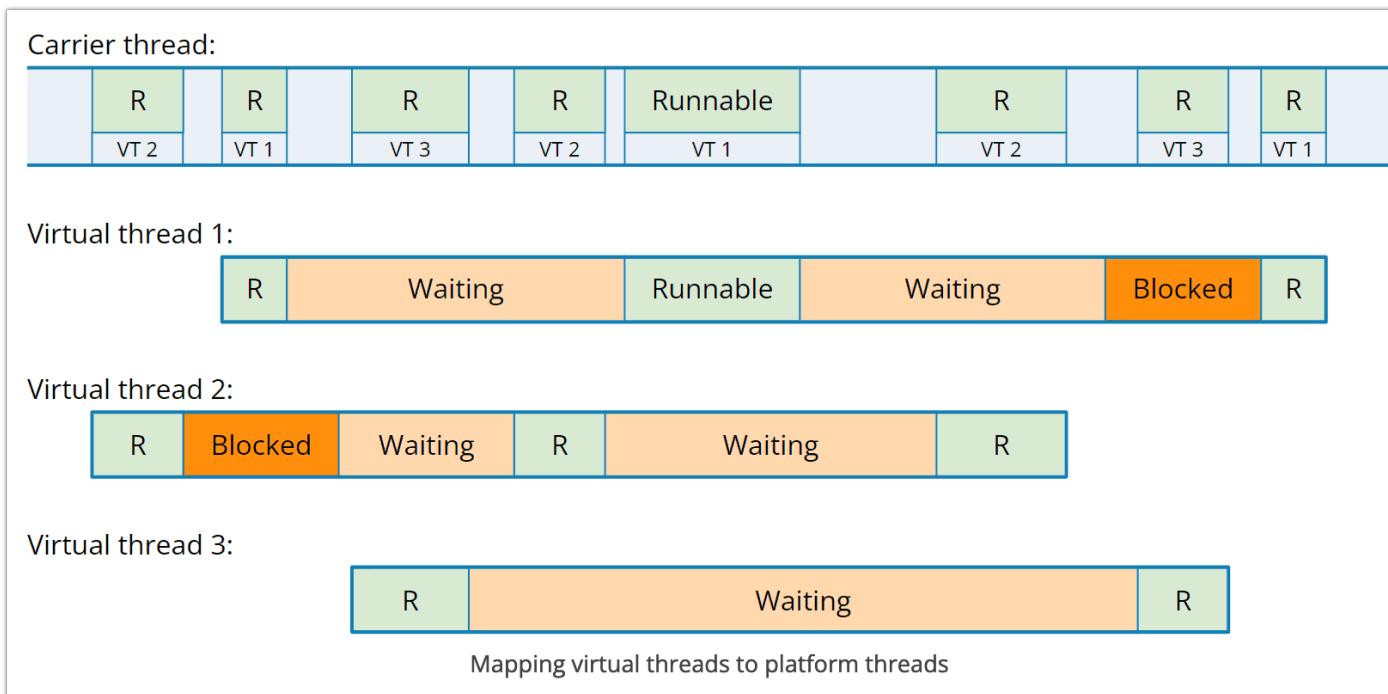
<https://openjdk.org/jeps/444>



JEP 444: Virtual Threads



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



JEP 444: Virtual Threads



- 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);
```



- 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.

https://www.youtube.com/watch?v=d_XmNicqC2I

JEP 444: Virtual Threads



- 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: Platform Threads & Virtual Threads Basics



```
Runnable action = () -> {
    var currentThread = Thread.currentThread();
    System.out.println("name: " + currentThread.getName() +
        " isVirtual(): " + currentThread.isVirtual());
};

var platformThread = Thread.ofPlatform().name("myPlatform").unstarted(action);
platformThread.start();

var virtualThread = Thread.ofVirtual().name("myFirstVirtual").unstarted(action);
virtualThread.start();

virtualThread.join();
System.out.println("is Thread? " + (virtualThread instanceof Thread));

Thread.startVirtualThread(action);
```

```
name: myPlatform isVirtual(): false
name: myFirstVirtual isVirtual(): true
is Thread? true
name: isVirtual(): true
```

JEP 444: Platform Threads => 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");
}
```

JEP 444: Virtual Threads



- Performance-Vergleich in Tausenderschritten für Plattform- und virtuelle Threads
- Die Threads warten jeweils eine Sekunde, um den Zugriff auf eine externe Schnittstelle zu simulieren. Am Ende wird die Gesamtzeit gemessen.

```
private static void submit1000Threads(ExecutorService executor) {  
    for (int i = 0; i < 1_000; i++) {  
        final int pos = i;  
        executor.submit(() -> {  
            Thread.sleep(Duration.ofSeconds(5));  
            return pos;  
        });  
    }  
}
```

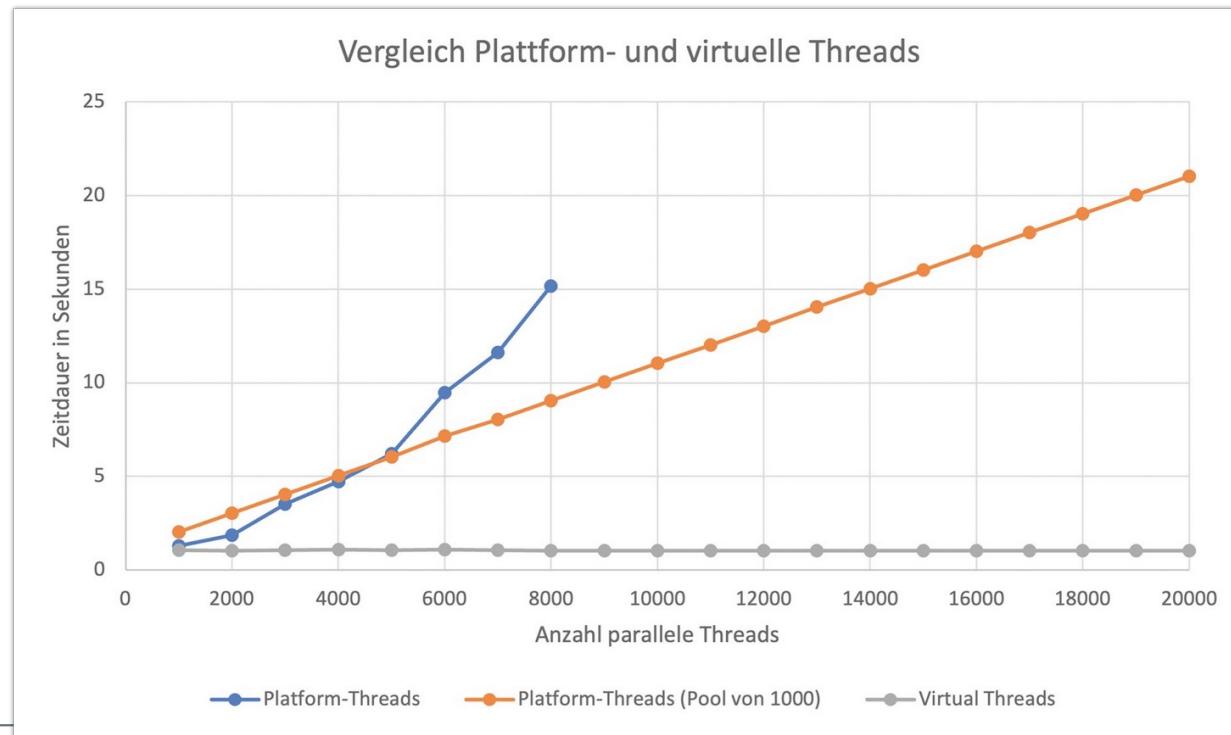
- Mehrere Durchläufe für 10k, 20k, ..., 50k

```
for (int i = 0; i < 10 * factor; i++) {  
    submit1000Threads(executor);  
}
```

JEP 444: Virtual Threads



- Performance-Vergleich in Tausenderschritten für Plattform- und virtuelle Threads
- Die Threads warten jeweils eine Sekunde, um den Zugriff auf eine externe Schnittstelle zu simulieren. Am Ende wird die Gesamtzeit gemessen.



<https://entwickler.de/java/javaneunzehn-features-virtualthreads>



DEMO & Hands on

`FirstVirtualThreadsExample.java`

`PlatformThreads.java`

`VirtualThreads.java`

`VirtualThreadsPoolingExample.java`



Übungen PART 3a

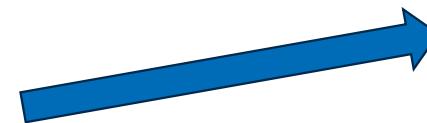
<https://github.com/Michaeli71/Best-Of-Java-17-23>





Preview Features in Java 21

- JEP 430: String Templates (Preview)
- JEP 443: Unnamed Patterns and Variables (Preview)
- JEP 445: Unnamed Classes and Instance Main Methods (Preview)
- JEP 453: Structured Concurrency (Preview)



String Templates
in Java 23 nach
der zweiten
Vorschau in Java
22 eingestellt



JEP 430: String Templates (Preview)

<https://openjdk.org/jeps/430>



Stringkonkatenation



- Um Strings, die Texte und variable Bestandteile enthalten, zu verknüpfen, bietet Java verschiedene Varianten, angefangen von der einfachen Verkettung mit + bis hin zur formatierten Umwandlung.

```
String result = "Calculation: " + x + " plus " + y + " equals " + (x + y);  
System.out.println(result);
```

```
String resultSB = new StringBuilder().  
    append("Calculation: ").append(x).append(" plus ").  
    append(y).append(" equals ").append(x + y).toString();  
System.out.println(resultSB);
```

```
System.out.println(String.format("Calculation: %d plus %d equals %d", x, y,  
                                x + y));  
System.out.println("Calculation: %d plus %d equals %d".formatted(x, y, x + y));
```

```
var messageFormat = new MessageFormat(" Calculation: {0} plus {1} equals {2}");  
System.out.println(messageFormat.format(new Object[] { x, y, x + y }));
```

- Alle haben ihre besonderen Stärken und vor allem Schwächen

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

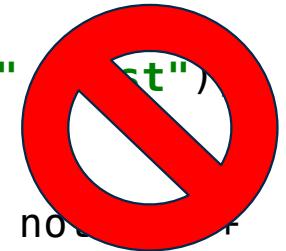
String Templates



- **Beispiel 2: Achtung, beim Splitten von Templates!!!**

```
Path filePath = Path.of("example.txt");
String infoOld = "The file " + filePath + " " +
    (filePath.toFile().exists() ? "does" : "does not" + " exist");

String infoNew = STR."The file \{filePath} " +
    STR."\{filePath.toFile().exists() ? "does " : "does not" +
    "exist";
```



- **Macht das hier wirklich Sinn? Oder sollte man die Auswertung in beiden Fällen nicht separat machen?**

```
String existInfo = filePath.toFile().exists() ? "does" : "does not";
String infoOldV2 = "The file " + filePath + " " + existInfo + " exist";
String infoNewV2 = STR."The file \{filePath} \{existInfo} exist";
```

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>
```

A screenshot of a web browser window titled "localhost". The page content is "My Hobbies:" followed by an unordered list containing three items: "• Cycling", "• Hiking", and "• Shopping".

```
My Hobbies:
• Cycling
• Hiking
• Shopping
```

String Templates – Berechnungen



```
int x = 10, y = 20;  
String calculation = STR."\{x} + \{y} = \{x + y}";  
System.out.println(calculation);
```

- => 10 + 20 = 30

```
int index = 0;  
String modifiedIndex = STR."\{index++}, \{index++}, \{index++}, \{index++}";  
System.out.println(modifiedIndex);
```

- => 0, 1, 2, 3

```
String currentTime = STR."Current time: \{  
    DateTimeFormatter.ofPattern("HH:mm").format(LocalTime.now())}\";  
System.out.println(currentTime);
```

- => Current time: 14:42

String Templates – nicht immer die beste Wahl ...



```
var sophiesBirthday = LocalDateTime.parse("2020-11-23T09:02");

var infoSophie = STR."Sophie was born on \{
    DateTimeFormatter.ofPattern("dd.MM.yyyy").format(sophiesBirthday)} at \{
    DateTimeFormatter.ofPattern("HH:mm").format(sophiesBirthday)}";
System.out.println(infoSophie);

System.out.println("Sophie was born on %s at %s".formatted(
    DateTimeFormatter.ofPattern("dd.MM.yyyy").format(sophiesBirthday),
    DateTimeFormatter.ofPattern("HH:mm").format(sophiesBirthday)));


• =>
```

```
Sophie was born on 23.11.2020 at 09:02
Sophie was born on 23.11.2020 at 09:02
```



**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
```

String Templates – Eigene String Processors



```
String name = "Michael";
int age = 53;
System.out.println(STR."Hello \{name}. You are \{age} years old.");
```

```
var myProc = new MyProcessor();
System.out.println(myProc."Hello \{name}. You are \{age} years old.");
```

```
Hello Michael. You are 53 years old.
```

```
-- process() --
info fragments:[Hello , . You are ,  years old.]
info values: [Michael, 53]
info interpolate: Hello Michael. You are 52 years old.
```

```
Hello >>Michael<<. You are >>52<< years old.
```

String Templates – Eigene String Processors



```
static class MyProcessor implements StringTemplate.Processor<String,  
                           IllegalArgumentException> {  
  
    @Override  
    public String process(StringTemplate stringTemplate)  
        throws IllegalArgumentException {  
        System.out.println("\n-- process() --");  
        System.out.println("info fragments:" + stringTemplate.fragments());  
        System.out.println("info values: " + stringTemplate.values());  
        System.out.println("info interpolate: " + stringTemplate.interpolate());  
        System.out.println();  
  
        var adjustedValues = stringTemplate.values().stream()  
            .map(str -> ">>" + str + "<<").  
            .toList();  
  
        return StringTemplate.interpolate(stringTemplate.fragments(),  
                                         adjustedValues);  
    }  
}
```



DEMO & Hands on

`JEP430_StringTemplates.java`
`JEP430_StringTemplates_Advanced.java`



JEP 443: Unnamed Patterns and Variables (Preview)

<https://openjdk.org/jeps/443>



JEP 443: Unnamed Patterns and Variables (Preview)



- Für alle, die die neuesten Java-Trends nicht verfolgen, hier eine kurze Rekapitulation
- Pattern Matching und Record Patterns haben sich in den letzten Java-Versionen massiv weiterentwickelt

```
Object obj = new Point(23, 11);

// Pattern Matching
if (obj instanceof Point point)
{
    int x = point.x();
    int y = point.y();
    System.out.println("x: " + x + " y: " + y + ", sum: " + (x + y));
}

// Record Pattern
if (obj instanceof Point(int x, int y))
{
    System.out.println("x: " + x + " y: " + y + ", sum: " + (x + y));
}
```

```
record Point(int x, int y) { }
enum Color { RED, GREEN, BLUE }
record ColoredPoint(Point point,
                     Color color) { }
```

JEP 443: Unnamed Patterns and Variables (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

JEP 443: Unnamed Patterns and Variables (Preview)



- Dieses JEP befasst sich damit, dass verschiedene Elemente in einem Ausdruck oder einer Variablen durch ein einzelnes `_` ersetzt werden können. Ziel ist es, eine Record Pattern Komponente oder Variable als unbrauchbar zu markieren und den Compiler verhindern zu lassen, dass die Variable verwendet wird, weil sie nicht dafür gedacht ist.

```
try
{
    Files.writeString(Path.of("UnnamedVars.txt"), "Underscore");
}
catch (IOException _)
{
    // Java: Ab Release 21 ist nur das Unterstrichschlüsselwort "_" zulässig, um
    // unbenannte Muster, lokale Variablen, Ausnahmeparameter oder
    // Lambda-Parameter zu deklarieren
    //_printStackTrace();
}
```

- Besonders erwähnenswert ist, dass eine mit `_` gekennzeichnete Variable weder gelesen noch geschrieben werden kann, wie es die im Kommentar implizierte Fehlermeldung zeigt.

JEP443_UnnamedVarsAndPatterns.java *(Hinweis Python)

JEP 443: Unnamed Patterns and Variables (Preview)



- Es gibt die folgenden drei Varianten:

1. **unnamed variable** – erlaubt die Verwendung von `_` zur Benennung oder Kennzeichnung nicht verwendeter 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 in einem Teil eines Record Patterns vollständig wegzulassen (und durch ein einzelnes `_` zu ersetzen)
-

JEP 443: Unnamed Patterns and Variables (Preview)



- **Unnamed variable I**

```
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.

JEP 443: Unnamed Patterns and Variables (Preview)



- **Unnamed variable II**

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

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

JEP 443: Unnamed Patterns and Variables (Preview)



- Unnamed variable III – aber ein bisschen verrückt? Warum? Lassen Sie es uns noch einmal überdenken ...

```
int LIMIT = 1000;
int total = 0;
List<Order> orders = List.of(new Order("iPhone"),
                             new Order("Pizza"), new Order("Water"));

for (var _ : orders) {
    if (total < LIMIT) {
        total++;
    }
}
System.out.println("total: " + total);
```

JEP 443: Unnamed Patterns and Variables (Preview)



- **Unnamed pattern variable I**

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

- =>

```
if (obj instanceof Point(int x, int _))  
{  
    System.out.println("x: " + x);  
}
```

- **Das Gleiche gilt für case Point(int x, int _)**

JEP 443: Unnamed Patterns and Variables (Preview)



- **Unnamed pattern variable II**

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

- =>

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

- **Das Gleiche gilt für case ColoredPoint(Point point, Color _)**

JEP 443: Unnamed Patterns and Variables (Preview)



- **Unnamed pattern variable III**

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

- =>

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

- **Das Gleiche gilt für case.**

JEP 443: Unnamed Patterns and Variables (Preview)



- Unnamed pattern

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

- =>

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

- Das Gleiche gilt für case.

instanceof _
instanceof _(int x, int y)

JEP 443: Unnamed Patterns and Variables (Preview)



```
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);
}
```

JEP 443: Unnamed Patterns and Variables (Preview)



```
static boolean checkFirstNameAndCountryCodeAgainImproved_UNNAMED_2(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;
}
```



JEP 445: Unnamed Classes and Instance Main Methods (Preview)

<https://openjdk.org/jeps/445>



JEP 445: Unnamed classes and instance main() method



- 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.

JEP 445: Unnamed classes and instance main() method



- Dieses JEP soll den Einstieg in Java erleichtern und es für kleinere Experimente so komfortabel wie möglich machen, insbesondere in Kombination mit Direct Compilation.
 - Das Ziel ist es, Java in Richtung eines **einfacheren Einstiegs** zu entwickeln.
 - Dies war in der Vergangenheit nicht der Fall und man musste immer wieder verschiedene Konzepte wie Packages, Klassen, Arrays, Sichtbarkeit erklären, die eigentlich nur im Zusammenhang mit größeren Programmen wichtig und nützlich sind, aber Anfänger zunächst verwirren.
 - Mit zunehmendem Wissen und wachsender Erfahrung können dann schrittweise fortgeschrittenere Konzepte wie Klassen, Sichtbarkeitskontrolle und statische Komponenten eingeführt werden.
 - Den Spracharchitekten bei Oracle war es wichtig, dass kein eigener Java-Dialekt entsteht oder eine separate Toolchain benötigt wird.
-

JEP 445: Unnamed classes and instance main() method



Vereinfachung I: Instance main()

- Nun ist es erlaubt, die `main()`-Methode nicht `static` und nicht `public` sowie `ohne Parameter` zu definieren, was schon in weniger Boilerplate-Code resultiert und die Verständlichkeit verbessert:

```
package preview;

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

- Sogar das `package` und die Namensangabe können weggelassen werden:

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

JEP 445: Unnamed classes and instance main() method



Vereinfachung I: Instance main()

- Wie bisher können diese Klassen wie ein normales Java-Programm kompiliert und gestartet werden, wobei die direkte Kompilierung sogar mit der skriptbasierten Ausführung von Python vergleichbar ist:

```
$ java --enable-preview --source 21 src/main/java/preview/InstanceMainMethod.java
Hello, World!
```

- Das bedeutet, dass Sie keine Sichtbarkeitsmodifikatoren oder statischen Elemente einführen müssen, um ein kleines Java-Programm zu schreiben. Außerdem ist es nicht notwendig, sich mit der Übergabe eines Parameters und dessen Array-Typ zu beschäftigen.
 - Ein guter erster Schritt, aber es geht noch weiter und wird sowohl besser als auch kürzer
-



Vereinfachung II: Unnamed class

- Wenn eine Klasse nur einfache Methoden definiert, wie es hier der Fall ist, lässt sich mit der neuen Funktion der unbenannten Klassen sogar die Klassendefinition weglassen. Jetzt haben wir ein fast so kurzes Programm wie mit dem Einzeiler in Python:

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

- Die resultierende Unnamed Class hat (natürlich) keine Klassendefinition, kann aber Attribute und Methoden angeben. Außerdem wird sie automatisch in ein Unnamed Package eingeordnet.
- Ausführen mit (wenn der Dateiname SimplerHelloWorld.java lautet)

```
$ java --enable-preview --source 21 SimplerHelloWorld.java  
Hello, World!
```

JEP 445: Unnamed classes and instance main() method



- **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!");  
}
```

JEP 445: Unnamed classes and instance main() method



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 <---
```



Fazit: Java auf dem Weg zur skriptbasierten Ausführung

- Diese neuen Funktionen erleichtern den Einstieg in Java erheblich.
- Außerdem reduziert sich der Aufwand auf ein Minimum, wenn man kleinere Tools erstellen möchte, die sich per Direct Compilation ohne vorheriges Kompilieren ausführen lassen.
- So profitieren nicht nur Anfänger, sondern auch alte Hasen, allerdings nur, wenn es um kleine Programme geht.
- Man kann sogar noch weiter gehen und möglicherweise `System.out.println()` als Vereinfachungspotenzial identifizieren, das sich allgemeiner auf Konsolenausgaben und -eingaben bezieht.
- Zumindest wird bei Oracle darüber nachgedacht. Dabei kann man etwas von Python mit den eingebauten Funktionen `print()` und `input()` lernen.

JEP 445: Unnamed classes and instance main() method



- Ausführungsreihenfolge -- Die mit diesem JEP implementierte Funktionalität vereinfacht vieles. Um die Ausführungsreihenfolge herauszufinden, wird folgender Ablauf befolgt:

1. static void main(String[] args)
2. static void main() ohne Parameter
3. void main(String[] args)
4. void main() ohne Parameter

- Mehrere Mains - raten Sie, welche ausgeführt wird😊

```
public class MultipleMains {  
    protected static void main() {  
        System.out.println("protected static void main()");  
    }  
  
    public void main(String[] args) {  
        System.out.println("public void main(String[] args)");  
    }  
}
```

```
$ java --enable-preview --source 21 src/main/java/preview/MultipleMains.java
```



JEP 453: Structured Concurrency (Preview)

<https://openjdk.org/jeps/453>



JEP 453: Structured Concurrency



- 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.
-

JEP 453: Structured Concurrency



- **Erster Versuch: Herkömmliche Umsetzung mit ExecutorService:**

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

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

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

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

- **Wir übergeben die zwei Teilaufgaben an den Executor und warten auf die Teilergebnisse. Dieser Happy Path ist schnell implementiert.**

JEP 453: Structured Concurrency



```
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 fehlgeschlagen. 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.

JEP 453: Structured Concurrency



- **Zur Erinnerung: Weil die Teilaufgaben parallel ausgeführt werden, können sie unabhängig voneinander erfolgreich sein oder fehlgeschlagen. Dann kann das Handling recht kompliziert werden.**
 - **Generelle Fragestellung: Wie gehen wir mit Exceptions um?**
 - Konkret, wenn in einer Teilaufgabe eine Exception auftritt, wie können wir die andere abbrechen?
 - Wie können wir die Abarbeitung der Teilaufgaben insgesamt stoppen, etwa, wenn wir die Ergebnisse nicht mehr benötigen?
 - **Mit einigen Tricks kann man das erreichen, der Sourcecode wird dann jedoch komplex, enthält diverse Abfragen und wird insgesamt schwierig zu verstehen und zu warten.**
-

JEP 453: Structured Concurrency



- 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  
  
        // Here, both forks have succeeded, so compose their results  
        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.

JEP 453: Structured Concurrency



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.
-

JEP 453: Structured Concurrency



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

```
try (var scope =
      new StructuredTaskScope.ShutdownOnSuccess<DeliveryService>())
{
    var result1 = scope.fork(() -> tryToGetPostService());
    var result2 = scope.fork(() -> tryToGetFallbackService());
    var result3 = scope.fork(() -> tryToGetCheapService());
    var result4 = scope.fork(() -> tryToGetFastDeliveryService());
    scope.join(); // KEIN throwIfFailed()

    System.out.println(STR."PS \{result1.state()\}/FS \{result2.state()\}" +
                       STR."/CS \{result3.state()\}/FDS \{result4.state()\}");
    System.out.println("found delivery service: " + scope.result());
}
```

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

JEP 453: Structured Concurrency



Vorteile beim Einsatz der Klasse StructuredTaskScope:

- Task und Subtasks bilden eine in sich geschlossene Einheit
- Es werden kein ExecutorService und Threads aus einem Thread-Pool genutzt.
Jede Teilaufgabe wird in einem neuen virtuellen Thread ausgeführt.
- ShutdownOnSuccess: wartet auf alle
 - Fehler in einer der Teilaufgaben => alle anderen Teilaufgaben werden abgebrochen.
- Shutdown onFailure: wartet auf einen
 - Fehler in einer der Teilaufgaben => Status FAILED
- Wird der aufrufende Thread abgebrochen, werden auch die Teilaufgaben abgebrochen.
- Aufrufhierarchie (aufrufender Thread und Teilaufgaben) im Thread-Dump etwas besser zu erkennen.

JEP 453: Structured Concurrency



Exception in thread "main" java.util.concurrent.ExecutionException: java.lang.IllegalStateException: JUST TO PROVIDE EX
at java.base/java.util.concurrent.FutureTask.report(FutureTask.java:122)
at java.base/java.util.concurrent.FutureTask.get(FutureTask.java:191)
at preview.api.StructuredConcurrency.**handleOldStyle**(StructuredConcurrency.java:48)
at preview.api.StructuredConcurrency.main(StructuredConcurrency.java:26)

Caused by: java.lang.IllegalStateException: JUST TO PROVIDE EX
at preview.api.StructuredConcurrency.**fetchOrders**(StructuredConcurrency.java:73)
at preview.api.StructuredConcurrency.lambda\$handleOldStyle\$1(StructuredConcurrency.java:43)
at java.base/java.util.concurrent.FutureTask.run(FutureTask.java:317)
at java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1144)
at java.base/java.util.concurrent.ThreadPoolExecutor\$Worker.run(ThreadPoolExecutor.java:642)
at java.base/java.lang.Thread.run(Thread.java:1583)

Exception in thread "main" java.util.concurrent.ExecutionException: java.lang.IllegalStateException: JUST TO PROVIDE EX
at java.base/java.util.concurrent.StructuredTaskScope\$ShutdownOnFailure.throwIfFailed(StructuredTaskScope.java:1318)
at java.base/java.util.concurrent.StructuredTaskScope\$ShutdownOnFailure.**throwIfFailed**(StructuredTaskScope.java:1295)
at preview.api.StructuredConcurrency.**handle**(StructuredConcurrency.java:57)
at preview.api.StructuredConcurrency.main(StructuredConcurrency.java:27)

Caused by: java.lang.IllegalStateException: JUST TO PROVIDE EX
at preview.api.StructuredConcurrency.**fetchOrders**(StructuredConcurrency.java:73)
at preview.api.StructuredConcurrency.lambda\$handle\$3(StructuredConcurrency.java:54)
at java.base/java.util.concurrent.StructuredTaskScope\$SubtaskImpl.run(StructuredTaskScope.java:889)
at java.base/java.lang.VirtualThread.run(VirtualThread.java:311)



DEMO & Hands on

StructuredConcurrency.java



Übungen PART 3b

<https://github.com/Michaeli71/Best-Of-Java-17-23>





PART 4:

Neuerungen in

Java 22 und Java 23



Java 22 – Was ist drin?



- JEP 423: Region Pinning for G1
- [JEP 447: Statements before super\(...\) \(Preview\)](#)
- [JEP 454: Foreign Function & Memory API](#)
- [JEP 456: Unnamed Variables & Patterns](#)
- JEP 457: Class-File API (Preview)
- [JEP 458: Launch Multi-File Source-Code Programs](#)
- JEP 459: String Templates (Second Preview)
- JEP 460: Vector API (Seventh Incubator)
- [JEP 461: Stream Gatherers \(Preview\)](#)
- JEP 462: Structured Concurrency (Second Preview)*
- [JEP 463: Implicitly Declared Classes and Instance Main Methods \(Second Preview\)](#)
- JEP 464: Scoped Values (Second Preview)

Java 22

Total: 12

Normal: 4

Preview: 7

Incubator: 1

*keine Änderungen, nur mehr Zeit für Feedback



Finale Features in Java 22

- JEP 454: Foreign Function & Memory API
 - JEP 456: Unnamed Variables & Patterns
 - JEP 458: Launch Multi-File Source-Code Programs
-



JEP 454: Foreign Function & Memory API

<https://openjdk.org/jeps/454>



Foreign Function & Memory API



- Mit JEP 454 zum Foreign Function & Memory API wird ein API zum Zugriff auf Funktionalitäten und Speicherbereiche, die außerhalb der JVM liegen, entwickelt.
 - Mitunter muss man aus Java-Programmen auf Funktionalitäten zugreifen, die etwa in C oder C++ geschrieben sind. Ebenso kann es notwendig sein, Speicherbereiche außerhalb der JVM zu adressieren.
 - Zwar gibt es für derartige Aufgaben seit Javas frühen Zeiten das JNI (Java Native Interface), das den Aufruf von nativem Code unterstützt, das aber auch diverse Unzulänglichkeiten besitzt: Es kann auch Probleme mit der Garbage Collection geben, wenn man Speicher in C mit malloc reserviert und dann an den Java-Aufruf zurückgeben will.
-

Foreign Function & Memory API



```
public static void main(final String[] args) throws Throwable
{
    // 1. SymbolLookup für gebräuchliche Bibliotheken ermitteln
    var linker = Linker.nativeLinker();
    final SymbolLookup stdlib = linker.defaultLookup();

    // 2. MethodHandle für "strlen" in der C Standard Library ermitteln
    var strlenMethodHandle = linker.downcallHandle(stdlib.find("strlen").orElseThrow(),
                                                    FunctionDescriptor.of(JAVA_LONG, ADDRESS));

    // 3. Java in C String umwandeln und in C-Speicher bereitstellen
    final Arena auto = Arena.ofAuto();
    var strMemorySegment = auto.allocateFrom("direct c call of strlen");

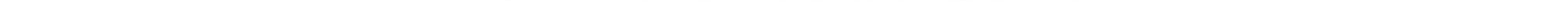
    // 4. Aufruf der "foreign function"
    final long len = (long) strlenMethodHandle.invoke(strMemorySegment);

    System.out.println("len = " + len);
    // 5. Speicher wird durch "Arena.ofAuto()" automatisch aufgeräumt
}
```



JEP 456: Unnamed Variables & Patterns

<https://openjdk.org/jeps/456>



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 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);
}
```



JEP 458: Launch Multi-File Source-Code Programs

<https://openjdk.org/jeps/458>



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.
- Bis einschließlich Java 21 LTS lediglich eine einzelne Java-Datei ausführbar.
- Je größer die Programme werden, desto mehr kommt der Wunsch auf, Funktionalitäten in verschiedene Klassen in eigenen Java-Dateien zu untergliedern. Das wird bislang für Direct Compilation nicht unterstützt.
- Mit diesem JEP wird genau dieser Umstand adressiert und eine Verbesserung beim Java-Programmstart umgesetzt. Übergang von kleinen zu größeren Programmen kann schrittweise erfolgen, ohne dafür ein Build-Tool wie Maven oder Gradle einführen zu müssen.

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.performCalculation();
        System.out.println(result);
    }
}
```

```
package jep458_Launch_MultiFile_SourceCode_Programs;

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

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

```
MainApp.java
```

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!<<
```

MainAppV2.java



DEMO & Hands on



Preview Features in Java 22

- JEP 447: Statements before super(...) (Preview)
- JEP 461: Stream Gatherers (Preview)
- JEP 463: Implicitly Declared Classes and Instance Main Methods (Second Preview)
- JEP 464: Scoped Values (Second Preview)

Alle sind
Teil von
Java 23 ...
also weiter
im Text



PART 4: Neuerungen in Java22 und Java 23

IDE & Tool Support für Java 23



- Eclipse: Version 2024-09 (mit Plugin)
- IntelliJ: Version 2024.2.2
- Maven: 3.9.9, Compiler Plugin: 3.13.0
- Gradle: 8.10
- Aktivierung von Preview Features erforderlich
 - In Dialogen
 - In Build Scripts



IDE & Tool Support Java 23



- Eclipse 2024-09 mit Plugin
- Aktivierung von Preview Features erforderlich

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

Find: Q Java 23 All Markets

Java 23 Support for Eclipse 2024-09 (4.33)
This marketplace solution provides Java 23 support for Eclipse 2024-09 (4.3). It is the latest Eclipse release, which is... [more info](#)
by [Eclipse Foundation](#), EPL 2.0
[java](#)

0 Installs: 70 (71 last month)

Resource Builders Coverage Java Build Path Java Code Style Java Compiler Javadoc Location Java Editor Maven Project Natures Project References Refactoring History Run/Debug Settings Task Repository WikiText

Eclipse

Properties for Java23Examples

Java Compiler

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

JDK Compliance

Use compliance from execution environment 'JavaSE-23' on the [Java Build Path](#) 23

Compiler compliance level: 23

Use '--release' option

Use default compliance settings

Enable preview features for Java 23 ←

Preview features with severity level: Info

Generated .class files compatibility: 23

Source compatibility: 23

Classfile Generation

IDE & Tool Support



- Aktivierung von Preview Features erforderlich

The screenshot shows the 'Project Structure' dialog in IntelliJ IDEA. The left sidebar has 'Project Settings' selected, with 'Project' highlighted. The main area is titled 'Project' and contains the following fields:

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

Two red arrows point to the 'Edit' button next to the SDK dropdown and the language level dropdown. At the bottom are 'Cancel', 'Apply', and 'OK' buttons.



IDE & Tool Support Java 23



- Aktivierung von Preview Features erforderlich

`sourceCompatibility=23`
`targetCompatibility=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 Java 23



- Aktivierung von Preview Features erforderlich

```
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



- Aktivierung von Preview Features erforderlich

```
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.13.0</version>
    <configuration>
      <source>23</source>
      <target>23</target>
      <!-- Important for Preview Features -->
      <compilerArgs>--enable-preview</compilerArgs>
    </configuration>
  </plugin>
</plugins>
```

```
<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>
```



IDE & Tool Support Java 23 für Vector API Incubator



Run Configurations

Create, manage, and run configurations

Run a Java application

Name: FirstVectorExample

Main Arguments JRE Dependencies Source Environment Common Prototype

Program arguments:

VM arguments:

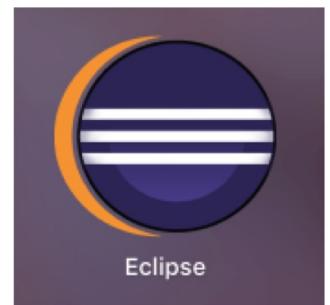
```
--enable-preview --add-modules jdk.incubator.vector
```

Variables... Variables...

Use the -XstartOnFirstThread argument when launching with SWT
 Use the -XX:+ShowCodeDetailsInExceptionMessages argument when launching
 Use @argfile when launching

Working directory:

Show Command Line Revert Apply



IDE & Tool Support für Vector API Incubator



Settings

Build, Execution, Deployment > Compiler > Java Compiler Revert changes ← →

Use compiler: Javac

Use '--release' option for cross-compilation (Java 9 and later)

Project bytecode version: Same as language level

Per-module bytecode version:

+ -

Module	Target bytecode version
Java23Examples	23

Javac Options

Use compiler from module target JDK when possible

Generate debugging info

Report use of deprecated features

Generate no warnings

Additional command line parameters:

--enable-preview --add-modules jdk.incubator.vector

'/' recommended in paths for cross-platform configurations

Override compiler parameters per-module:

+ -

Cancel Apply OK



IDE & Tool Support für Vector API Incubator



Run/Debug Configurations

Name: FirstVectorExample Store as project file

Run on: Local machine Manage targets...

Build and run

java 23 SDK of 'Java23' --enable-preview --add-modules jdk.incubator.vector

jep469_Vector_Api.FirstVectorExample

Program arguments

VM options. CLI arguments to the 'Java' command. Example: -ea -Xmx2048m.

Working directory: \VA-BEST-OF/2024/W_JAX_it_tage_Java_23_WORKSHOP_2024/Java23Exam

Environment variables: Environment variables or .env files

Separate variables with semicolon: VAR=value; VAR1=value

Open run/debug tool window when started

Code Coverage

Packages and classes to include in coverage data

- + jep469_Vector_Api.*

Edit configuration templates...





Overview: JEPs in Java 23

Java 23 – Was ist inkludiert?



- **JEP 455:** [Primitive Types in Patterns, instanceof, and switch \(Preview\)](#)+ neu: aus Tex
- JEP 466: Class-File API (Second Preview)
- **JEP 467:** [Markdown Documentation Comments](#)
- JEP 471: Deprecate the Memory-Access Methods in sun.misc.Unsafe for Removal
- **JEP 473:** [Stream Gatherers \(Second Preview\)](#)
- JEP 474: ZGC: Generational Mode by Default
- **JEP 476:** [Module Import Declarations \(Preview\)](#)
- **JEP 477:** [Implicitly Declared Classes and Instance Main Methods \(Third Preview\)](#)
- JEP 480: Structured Concurrency (Third Preview)*
- JEP 481: Scoped Values (Third Preview)
- **JEP 482:** [Flexible Constructor Bodies \(Second Preview\)](#)
- JEP 469: Vector API (Eighth Incubator)

Java 23

Total: 12

Normal: 3

Preview: 8 Incubator: 1

*keine Änderungen, nur mehr Zeit für Feedback, bereits in Java 21 Teil behandelt



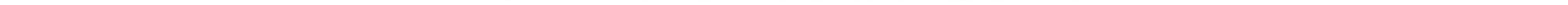
Finale Features in Java 23

- JEP 467: Markdown Documentation Comments
-



JEP 467: Markdown Documentation Comments

<https://openjdk.org/jeps/467>



JEP 467: Markdown Documentation Comments



- Als Java in den späten 1990er Jahren populär wurde, war die Wahl von HTML-Elementen als JavaDoc-Kommentarspezifikation absolut logisch.
- Allerdings wurde in den letzten Jahren Markdown für Dokumentationen immer beliebter.
- Markdown-Kommentare werden durch drei Schrägstriche (///) gekennzeichnet:

```
/// 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;
}
```

JEP 467: Markdown Documentation Comments – Formatierung



- Textpassagen sollen mitunter hervorgehoben werden

- kursiv* (*...* or ...) oder **fett** (**...**)
- Schriftart durch einen Backtick ('...) in Schreibmaschinenschrift ändern.
Fett und/oder kursiv sind dabei ebenfalls möglich.
- Mehrzeilige Quellcode-Ausschnitte mit ("..."") im Kommentar.

```
/// **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

```
public class MarkDownComment
```

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;
}
```

Java23Examples

JEP 467: Markdown Documentation Comments – Listen & Tabellen



```
/// - Punkt A
/// * Punkt B
/// - Punkt C
///
/// 1. Eintrag 1
/// 1. Eintrag 2 -- **wird automatisch nummeriert
/// 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)

JEP 467: Markdown Documentation Comments



- Mit diesem JEP können Sie bekanntlich Kommentare mit Markdown anstelle von HTML-Elementen erstellen.
- In IntelliJ über das Menü Tools > Generate JavaDoc... generieren
- Generiert JavaDoc im Ordner generated-doc

✓	Java23Examples	~/Desktop/Vorträge/A_JAVA-BE
>	.gradle	
>	.idea	
>	.settings	
>	build	
✓	generated-doc	
>	index-files	
>	jep454_Foreign_Function	
>	jep466_Class_File_Api	
>	jep469_Vector_Api	
>	jep473_Stream_Gatherers	
>	legal	
>	resource-files	
>	script-files	
>	syntax	
✓	allclasses-index.html	
✓	allpackages-index.html	
✓	DateAndTimeAPI.html	
✓	element-list	
✓	FirstGathererExample.html	
✓	help-doc.html	
✓	index.html	
✓	InitialExample.html	

JEP 467: Markdown Documentation Comments



- Markdown-Kommentare werden auch in IntelliJ direkt auf dem kommentierten Programmelement (Klasse/Methode) angezeigt.

```
/// 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;
}
```

c 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 – Referenzierungen



- Verweise auf Programmelemente (Module, Klassen, Methoden usw.) durch [`<ref>`]
- In `java.lang` definierte Typen können ohne Package-Angabe geschrieben werden, andere Typen müssen vollständig qualifiziert angegeben werden.

```
/// [java.base/] - verweist auf ein Modul \
/// [java.util] - verweist auf ein Package
///
/// Hinweis: _**`java.lang`**_ java.base✉ - verweist auf ein Modul
/// [java.lang.String] - verweist auf String✉
/// [String] - verweist auf eine Klasse Hinweis: java.lang kann man im Verweis weglassen:
/// [Integer] - verweist auf Integer✉
/// [String#chars()] - verweist auf String.chars()✉
/// [Integer#valueOf(String, i)] - verweist auf Integer.valueOf(String, int)✉
///
/// [String#CASE_INSENSITIVE_ORDER] - verweist auf String.CASE_INSENSITIVE_ORDER✉
/// [SPECIAL_ORDER][String#CASE_INSENSITIVE_ORDER] - verweist auf ein Attribut
/// [SPECIAL_ORDER] - Verweis mit Beschriftung SPECIAL_ORDER✉
```



Preview Features in Java 23

- JEP 455: Primitive Types in Patterns, instanceof, and switch (Preview)
 - JEP 473: Stream Gatherers (Second Preview)
 - JEP 476: Module Import Declarations (Preview)
 - JEP 477: Implicitly Declared Classes and Instance Main Methods (Third Preview)
 - JEP 482: Flexible Constructor Bodies (Second Preview)
-



JEP 455: Primitive Types in Patterns, instanceof, and switch (Preview)

<https://openjdk.org/jeps/455>



JEP 455: Primitive Types in Patterns, instanceof, and switch (Preview)



- With instanceof and switch, type and value checks can be performed. Conveniently, modern Java also allows record patterns to be specified there.

```
// Pattern Matching with instanceof
if (obj instanceof String str && str.length() > 7) {
    System.out.println("string " + str + " is longer than 7 characters");
} else if (obj instanceof Double d && d > 100) {
    System.out.println("double " + d + " is larger than 100");
} else {
    System.out.println("arbitrary obj: " + obj);
}

// Pattern Matching with switch
switch (obj) {
    case String str when str.length() > 7 ->
        System.out.println("string " + str + " is longer than 7 characters");
    case Double d when d > 100 ->
        System.out.println("double " + d + " is larger than 100");
    default -> System.out.println("arbitrary obj: " + obj);
}
```

- The aim of this JEP was to improve pattern matching and make it more universally valid by also supporting so-called primitive type patterns to allow primitive types.

JEP 455: Primitive Types in Patterns, instanceof, and switch (Preview)



- Beispiel

```
String msg = switch (logLevel.severity())
{
    case 0 -> "info";
    case 1 -> "warning";
    case 2 -> "error";
    default -> "unknown severity: " + logLevel.severity();
};
```

Doppelte Aufrufe,
sonst kein Zugriff auf
den Wert in der
Standardeinstellung



```
String msg = switch (logLevel.severity()) {
    case 0 -> "info";
    case 1 -> "warning";
    case 2 -> "error";
    // JEP 455
    case int severity -> "unknown severity: " + severity;
};
```

JEP 455: Primitive Types in Patterns, instanceof, and switch (Preview)



- Beispiel mit Guards

```
void evaluate(final int score)
{
    var result = switch (score)
    {
        case int value when value > 100 -> "DISQUALIFIED due to cheating";
        case int value when value >= 80 -> "excellent";
        case int value when value >= 50 -> "okalish";
        case int value when value >= 0 && value <= 50 -> "below expectations";
        default -> throw new IllegalStateException("Invalid score: " + score);
    };
    System.out.println("Your score: " + result);
}
```

JEP 455: Primitive Types in Patterns, instanceof, and switch (Preview)



- Sonderfall mit Gleitkommazahlen

```
float val = 1.0f;
switch (val) {
    case 1.0f -> System.out.println("1.0");
    case 0.99999999f -> System.out.println("0.9999..");
    default -> System.out.println("something else");
}
```

```
float val = 1.0f;
switch (val) {
    case 1.0f -> System.out.println("1.0");
    case 0.99999999f -> System.out.println("0.9999..");
    default -> Sys
```

Duplicate label '1'

⋮

Remove switch branch '0.99999999f' ↕ More actions... ↕

JEP 455: Primitive Types in Patterns, instanceof, and switch (Preview)



- Wertbereichsüberprüfungen vereinfacht

```
void checkByteAndPrintOld(int value)
{
    if (value >= -128 && value <= 127)
    {
        byte byteValue = (byte)value;
        System.out.println("byte: " + byteValue);
    }
}
```



```
void checkByteAndPrint(int value)
{
    if (value instanceof byte byteValue)
    {
        System.out.println("byte: " + byteValue);
    }
}
```



JEP 473: Stream Gatherers (Second Preview)

<https://openjdk.org/jeps/473>



JEP 461: Stream Gatherers



- Die in Java 8 eingeführten Streams waren bereits von Anfang an recht mächtig.
 - In den folgenden Java-Versionen wurden verschiedene Erweiterungen im Bereich der Terminal Operations hinzugefügt. Terminal Operations dienen dazu, die Berechnungen eines Streams abzuschließen und den Stream beispielsweise in eine Collection oder einen Ergebniswert zu überführen.
 - Mit diesem JEP wird eine Erweiterung des Stream-APIs zur Unterstützung benutzerdefinierter Intermediate Operations umgesetzt. Bisher gab es zwar diverse vordefinierte Intermediate Operations, aber keine Erweiterungsmöglichkeit.
 - Eine solche ist wünschenswert, um Aufgabenstellungen realisieren zu können, die zuvor nicht ohne Weiteres oder nur mit Tricks sowie eher umständlich umzusetzen waren.
-

JEP 461: Stream Gatherers



- 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).           // Hypothetisch  
    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();
```

JEP 461: Stream Gatherers



- Der Trick besteht in einem Record, der einen String kapselt und equals() und hashCode() speziell auf die Stringlänge ausgerichtet implementiert:

```
var result = Stream.of("Tim", "Tom", "Jim", "Mike").  
    map(DistinctByLength::new).  
    distinct().  
    map(DistinctByLength::str).  
    toList();  
  
record DistinctByLength(String str) {  
  
    @Override public boolean equals(Object obj) {  
        return obj instanceof DistinctByLength(String other)  
            && str.length() == other.length();  
    }  
  
    @Override public int hashCode() {  
        return str == null ? 0 : Integer.hashCode(str.length());  
    }  
}
```

JEP 461: Stream Gatherers



- 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).          // Hypothetisch  
    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 – Neuerung in Java 22



- Java 22 bringt analog zu `collect(Collector)` für Terminal Operationsn nun eine Methode `gather(Gatherer)` zur Bereitstellung einer benutzerdefinierten Intermediate Operation.
 - Dazu dient das Interface `java.util.stream.Gatherer`, das jedoch ein wenig herausfordernd selbst zu implementieren ist.
 - Praktischerweise gibt es in der Utility-Klasse `java.util.stream.Gatherers` diverse vordefinierte Gatherer wie:
 - `windowFixed()`
 - `windowSliding()`
 - `fold()`
 - `scan()`
-

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 (hier nicht gezeigt):

```
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 foldSum()
{
    var crossSum = Stream.of(1, 2, 3, 4, 5, 6, 7).
        gather(Gatherers.fold(() -> 0L,
                           (result, number) -> result + number)).
        findFirst();
    System.out.println("sum with fold(): " + crossSum);
}
```

- `gather()` gibt einen Stream als Ergebnis zurückgibt (hier einen einelementigen). Mit `toList()` würde man eine einelementige Liste erhalten:
`sum with fold(): [28]`
- Um den Wert auszulesen, dient der Aufruf von `findFirst()`, das liefert einen `Optional<T>`:
`sum with fold(): Optional[28]`

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]
```

JEP 461: Stream Gatherers – fold()



- 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(): [1223334444555556666667777777]
```

JEP 461: Stream Gatherers – scan()



- Sollen die Elemente eines Streams zu neuen Kombinationen zusammengeführt werden, sodass jeweils immer ein Element dazukommt, dann ist dies die Aufgabe von `scan()`.
- Die Methode arbeitet ähnlich wie `fold()`, das die Werte zu einem Ergebnis kombiniert. Bei `scan()` wird jedoch 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, 12233344445555566666777777]
```



DEMO & Hands on



JEP 476: Module Import Declarations (Preview)

<https://openjdk.org/jeps/476>



JEP 476: Module Import Declarations (Preview)



- **Mit JEP 476 ist es möglich, die von einem Modul exportierten Typen mit nur einem Import zu verwenden**
- **Genauer gesagt: alle öffentlichen Typen aus den von einem Modul exportierten Paketen**
- **Zusammenfassung:**
 - Seit den Anfängen von Java wurde bewusst entschieden, dass alle Typen aus dem `java.lang`-Package automatisch in jeder Klasse verfügbar sind und nicht explizit importiert werden müssen.
 - Der Grund dafür ist die Bequemlichkeit und Benutzerfreundlichkeit.
 - Mit diesem Mechanismus können Sie die gängigen Typen `String`, `Integer` oder `Exception` direkt in jedem Programm verwenden.

JEP 476: Module Import Declarations (Preview)



- Spätestens wenn Programme größer werden, werden häufig Typen aus anderen Packages benötigt.
- Wie z. B. die Containerklassen `ArrayList<E>`, `HashSet<E>` und `HashMap<K,V>` oder die zugehörigen Interfaces `List<E>`, `Set<E>` und `Map<K,V>`.
- Dies erfordert separate Importe wie diese:

```
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
```

- In der Java-Community gibt es kontroverse Meinungen und Debatten darüber, ob ein allgemeiner Import wie dieser besser oder schlechter ist:

```
import java.util.*;
```

JEP 476: Module Import Declarations (Preview)



- Mit Java 23 ist es einfacher, Dinge wiederzuverwenden, indem ganze Module auf einmal importiert werden.
`import module <module-name>;`
- Dadurch können sich wiederholende und ähnliche Importanweisungen vermieden werden.
- Dadurch werden alle öffentlichen Typen, d. h. Klassen, Schnittstellen und Aufzählungen, aus allen vom Modul exportierten Paketen ohne weiteren Import bereitgestellt.
- Wie bereits erwähnt, können Importe zu einer beträchtlichen Anzahl von Zeilen führen. Wenn Sie beispielsweise alle Typen analog zu denen aus dem Modul importieren möchten

`import module java.base;`

würden über 50 einzelne Importe erforderlich sein, darunter viele Typen, die in aus
`import java.io.*` and `import java.util.*` stammen.

JEP 476: Module Import Declarations (Preview)



- Dadurch werden alle öffentlichen Typen, d. h. Klassen, Schnittstellen und Aufzählungen, aus allen vom Modul exportierten Paketen ohne weiteren Import bereitgestellt.
 - Praktischerweise funktioniert dies auch transitiv. Wenn Sie schreiben

```
import module java.sql;
```
 - sind die Typen aus dem `java.xml` -Modul ebenfalls verfügbar. Dies ist praktisch, um die Modulimporte übersichtlich zu halten.
 - Es ist auch einfacher, mit Klassen im Kontext von Direct Compilation und implizit deklarierten Klassen zu beginnen und sie zu verwenden.
-

JEP 476: Module Import Declarations (Preview)



```
import java.util.List;
import java.util.stream.Stream;

public class InitialModuleImportExample {
    public static void main(final String[] args) {
        List<String> result = Stream.of("AB", "BC", "CD", "DE").
            filter(str -> str.contains("C")).
            toList();
        System.out.println(result);
    }
}

import module java.base;

public class InitialModuleImportExample2 {
    public static void main(final String[] args) {
        List<String> result = Stream.of("AB", "BC", "CD", "DE").
            filter(str -> str.contains("C")).
            toList();
        System.out.println(result);
    }
}
```

JEP 476: Module Import Declarations (Preview)



```
import javax.swing.*;
import java.io.IOException;
import java.util.List;
import java.util.stream.Stream;

public class ModuleImportExample3 {
    public static void main(String[] args) {
        List<String> result = Stream.of("AB", "BC", "CD", "DE").
            filter(str -> str.contains("C")).
            toList();
        System.out.println(result);

        IO.print("New Style IO");

        JOptionPane.showMessageDialog(null, "Info");
    }
}
```

JEP 476: Module Import Declarations (Preview)



```
import module java.base; // import java.util.* , java.util.stream.* , ...
import module java.desktop; // import javax.swing.* ; usw.

public class ModuleImportExample3 {
    public static void main(String[] args) {
        List<String> result = Stream.of( ...values: "AB" , "BC" , "CD" , "DE" ) .
            Reference to 'List' is ambiguous, both 'java.util.List' and 'java.awt.List' match
            Sy Import class More actions...
            IO.print("New Style IO");
            JOptionPane.showMessageDialog(null, "Info");
    }
}
```

- Allerdings stoßen wir jetzt auf den Fehler, dass der Typ List nicht mehr eindeutig ist. Wie die Fehlermeldung deutlich zeigt, ist der Typ List im Package java.awt und in java.util definiert.
- Grund: Da ein Modulimport alle Typen aus allen vom Modul exportierten Paketen bereitstellt, ist es möglich, dass Typen mit demselben Namen aus verschiedenen Packages existieren.

JEP 476: Module Import Declarations (Preview)



```
import module java.base; // import java.util.* , java.util.stream.* , ...
import module java.desktop; // import javax.swing.* ; usw.

import java.util.List;

public class ModuleImportExample3 {
    public static void main(String[] args) {
        List<String> result = Stream.of("AB", "BC", "CD", "DE").
            filter(str -> str.contains("C")).
            toList();
        System.out.println(result);

        I0.print("New Style I0");

        JOptionPane.showMessageDialog(null, "Info");
    }
}
```

JEP 476: Module Import Declarations (Preview)



- Bei der Date-Klasse ist es auch sehr wahrscheinlich, dass es zu Problemen mit Modulimports kommt, da Date in den Paketen `java.util` und `java.sql` vorhanden ist:

```
jshell> Date module java.base;  
  
jshell> import module java.sql;  
  
jshell> new Date(4711)  
| Fehler:  
| Referenz zu Date ist mehrdeutig  
| Sowohl Klasse java.sql.Date in java.sql als auch Klasse java.util.Date  
in java.util stimmen überein  
| new Date(4711)  
|     ^__^
```

- Ein import löst das Problem:

```
jshell> import java.util.Date;  
  
jshell> var today = new Date()  
today ==> Sun Sep 08 17:03:32 CEST 2024
```



JEP 477: Implicitly Declared Classes and Instance Main Methods (Third Preview)

<https://openjdk.org/jeps/477>



JEP 463: Implicitly Declared Classes and Instance Main Methods



- In Java 21 LTS wurden mit JEP 445 die Unnamed Classes and Instance Main Methods als Preview-Feature eingeführt, um etwa «Hello World» maximal kurz zu schreiben:

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

- Die so entstehenden impliziten Klassen dürfen keine Package-Angabe besitzen.

```
package jep477.is.not.supported;

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

JEP 463: Implicitly Declared Classes and Instance Main Methods



- Zudem wurde die Auswahl der passenden `main()`-Methode vereinfacht: Gibt es eine statische Methode, so wird diese genommen, ansonsten die andere. In Java 21 LTS gab es noch einen vierstufigen Ermittlungsprozess:

```
public class InstanceMainMethodExample
{
    public void main()
    {
        System.out.println("InstanceMainMethodExample");
    }

    public static void main(final String[] args)
    {
        System.out.println("Static main");
    }
}
```

- =>

Static main

JEP 477: Implicitly Declared Classes and Instance Main Methods



- Auswahl der geeigneten `main()`-Methode basierend auf dem Parameter:

```
public class InstanceMainMethodExample
{
    public void main(final String[] args)
    {
        System.out.println("InstanceMainMethodExample");
    }

    public static void main()
    {
        System.out.println("Static main");
    }
}
```

- =>

InstanceMainMethodExample

JEP 477: Implicitly Declared Classes and Instance Main Methods



- Zwei main()-Methoden mit der gleichen Signatur sind nicht erlaubt:

```
public class InstanceMainMethodExample3 {  
    public void main() { System.out.println("InstanceMainMethodExample"); }  
  
    public static void main() {  
        System.out.println("Main Method Example");  
    }  
}
```

'main()' is already defined in 'jep477_Implicitly_Declared_Classes.InstanceMainMethodExample3'

↳ jep477_Implicitly_Declared_Classes.InstanceMainMethodExample3
public static void main()
Java23Examples

JEP 477: Implicitly Declared Classes and Instance Main Methods



- Vereinfachte Auswahl der passenden main()-Methode:

```
public class InstanceMainMethodExample
{
    public void main()
    {
        System.out.println("InstanceMainMethodExample");
    }

    /*
    public static void main(final String[] args)
    {
        System.out.println("Static main");
    }
    */
}
```

- =>

InstanceMainMethodExample



DEMO & Hands on

JEP 477: Implicitly Declared Classes and Instance Main Methods



- Der JEP 477 aus Java 23 enthält die Neuerungen von Java 22. Darüber hinaus wurden in Java 23 zwei maßgebliche Neuerungen hinzugefügt:
 - **Interaktion mit der Konsole:** Implizit deklarierte Klassen importieren automatisch die drei statischen Methoden `print()`, `println()` und `readln()`, die in der Klasse `java.io.IO` definiert sind und die textbasierte Interaktion mit der Konsole vereinfachen.
 - **Automatischer Modulimport von `java.base`:** Implizit deklarierte Klassen importieren automatisch alle öffentlichen Klassen und Schnittstellen der vom `java.base`-Modul exportierten Packages.
- Basierend auf beiden kann die `main()`-Methode in Java 23 klarer und kürzer wie folgt geschrieben werden:

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

JEP 477: Interaction with console



- Die Interaktion mit der Konsole ist in der Regel umständlich und kompliziert, insbesondere die Eingabe
- Die Klasse `java.io.IO` dient als Workaround und bietet drei neue Methoden:

```
public static void println(Object obj);
public static void print(Object obj);
public static String readln(String prompt);
```

- Diese helfen beim Einlesen des Namens und beim Ausdrucken der Begrüßung wie folgt:

```
void main()
{
    var name = readln("Please enter your name: ");
    println("Hello " + name);
}
```

- Bei implizit deklarierten Klassen sieht das so aus:

```
import static java.io.IO.*;
```

JEP 477: Automatic module import of java.base



- Mit zunehmender Erfahrung neigt man dazu, komplexere Dinge umsetzen zu wollen, nämlich solche, die über einfache Spielereien und Interaktionen mit der Konsole hinausgehen.
- Nehmen wir zum Beispiel an, dass die Eingaben in einer Liste verwaltet oder in einer Datei für die spätere Verwendung gespeichert werden sollen.
- Das JDK bietet dafür nette Funktionalitäten, die aber in der Regel einige Importe erfordern.
- Für Anfänger ist es nicht einfach, sich Packages und deren Hierarchien zu merken.
- Oracle möchte den Einstieg in Java erleichtern, indem das Modul `java.base` für jede implizit deklarierte Klasse automatisch importiert wird.
- Dies bedeutet, dass wichtige und beliebte APIs in gängigen Paketen wie `java.io`, `java.math` und `java.util` direkt verwendet werden können.

JEP 477: Automatic module import of java.base



- Einfache Entwicklung kleinerer Programme:

```
void main()
{
    String name = readln("Please enter your name: ");

    var authors = List.of("Tim", "Tom", "Mike", "Michael");
    for (var author_name : authors)
    {
        if (name.equalsIgnoreCase(author_name))
        {
            println(name + " you are registered as author!");
        }
    }
}
```

- Funktioniert, als ob diese Importe angegeben wären:

```
import java.util.List;
import static java.io.IOException;
```

JEP 477: Evolution to a regular class



- Ein Programm, das als implizit deklarierte Klasse implementiert wird, ermöglicht es, sich intensiver auf die anstehende Aufgabe zu konzentrieren.
- Alles Unwichtige kann (zunächst) weggelassen werden. Dennoch werden alle Komponenten auf die gleiche Weise interpretiert wie in einer regulären Klasse.
- Der Übergang ist absolut einfach
 - Fügen Sie einfach eine Klassendeklaration um main() herum hinzu
 - Fügen Sie eine Package-Anweisung hinzu
 - Fügen Sie die Importe hinzu

```
package jep477_Implicitly_Declared_Classes;  
  
import java.util.List;  
  
import static java.io.IOException.*;  
  
public class GrowingClass  
{  
    void main()  
    {  
        ... // same as before  
    }  
}
```



JEP 482: Flexible Constructor Bodies (Second Preview)

<https://openjdk.org/jeps/482>



JEP 482: Statements before super(...) / Flexible Constructor Bodies



- Ein Ziel dieses JEPs ist es, den Entwicklern mehr Freiheit bei der Implementierung von Konstruktoren zu geben.
- Insbesondere werden gewisse Aktionen noch vor dem Aufruf von `super()` (und sogar `this()`) möglich, etwa zum Prüfen von Konstruktorargumenten.*
- Es sollte wie bisher garantiert sein, dass Konstruktoren während der Klasseninstanziierung in der Vererbungshierarchie von oben nach unten ausgeführt werden.
- Damit können die Anweisungen in einem Subklassenkonstruktor die Instanziierung der Basisklasse nicht beeinträchtigen.
- Zudem sollte das Ganze keine Änderungen an der JVM erfordern.

*Bislang ist das nur über Tricks wie statische Hilfsmethoden oder zusätzliche Hilfskonstruktoren möglich.

JEP 482: Statements before super(...) / Flexible Constructor Bodies



- 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 482: Statements before super(...)



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

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

- 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.

JEP 482: 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 482: 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 und den Nachfolger JEP 482 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 482: Statements before super(...)



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

```
record MyPoint0ld(int x, int y)
{
    public MyPoint0ld(final String values)
    {
        this(Integer.parseInt(values.split(",")[0].strip(),
                             Integer.parseInt(values.split(",")[1].strip())));
    }
}

record MyPoint3d0ld(int x, int y, int zy)
{
    public MyPoint3d0ld(final String values)
    {
        this(Integer.parseInt(values.split(",")[0].strip(),
                             Integer.parseInt(values.split(",")[1].strip(),
                             Integer.parseInt(values.split(",")[2].strip())));
    }
}
```

JEP 482: 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());
    }
}
```

JEP 482: Flexible Constructor Bodies – Neu in Java 23



- Im Zusammenhang mit Vererbung kann es gelegentlich zu Überraschungen kommen, wenn in Konstruktoren Methoden aufgerufen werden, die in Unterklassen überschrieben werden.

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

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

        logValues();
    }

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

JEP 482: Flexible Constructor Bodies – Neu in Java 23



```
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**

Während der Verarbeitung des Basisklassenkonstruktors ist das Attribut subClassInfo noch nicht zugewiesen, da der Aufruf von super() VOR der Zuweisung an die Variable erfolgt. Dies führt zu der oben genannten, aber unerwarteten Ausgabe.

JEP 482: Flexible Constructor Bodies – Neu in Java 23



```
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

Während der Verarbeitung des Basisklassenkonstruktors ist das Attribut subClassInfo jetzt bereits zugewiesen, da der Aufruf von super() NACH der Zuweisung an die Variable erfolgt. Dies führt zu der oben gezeigten und erwarteten Ausgabe.



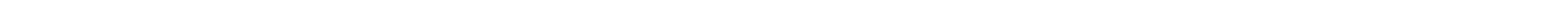
DEMO & Hands on

- SubClass
 - NewSubClass
-



Übungen PART 4

<https://github.com/Michaeli71/Best-Of-Java-17-23>





Fazit



On the positive side



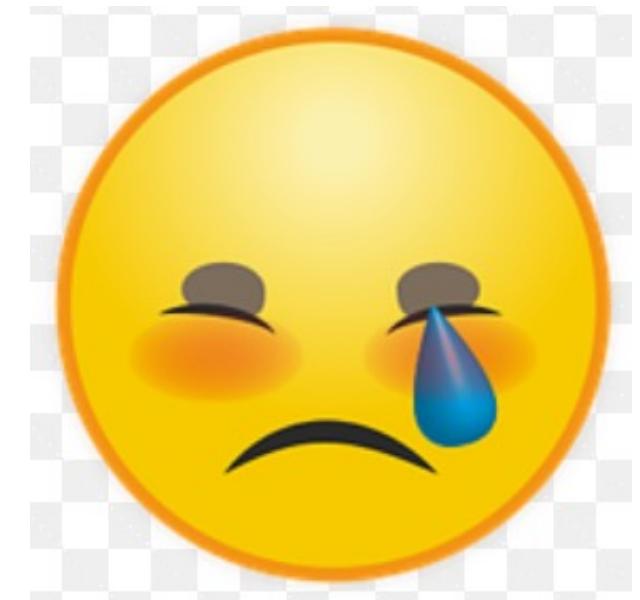
- Stabile und zuverlässige 6-monatige Release-Zyklen und alle 2 Jahre LTS-Versionen
- Java wird einfacher und attraktiver
- Viele schöne Verbesserungen in Syntax und APIs wie **switch, Records, Text Blocks**
- Pattern Matching und Record Patterns final in Java 21
- HTTP/2, virtuelle Threads & Structured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



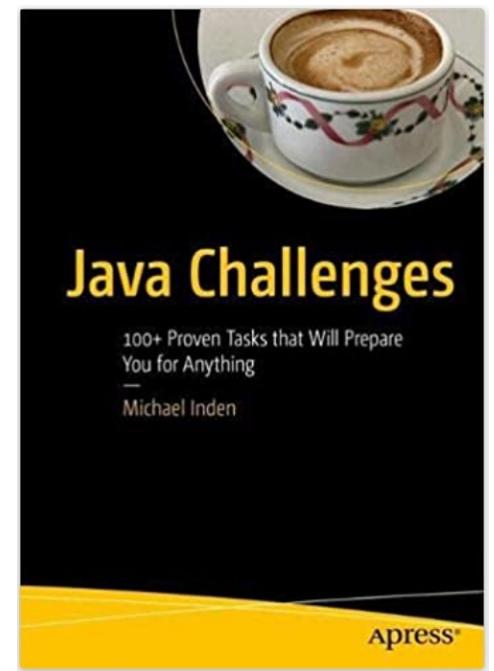
Negatives



- 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?



Hilfe





Questions?



Thank You