



JUGS – Best of Modern Java 21 – 24

Meine Lieblingsfeatures

<https://github.com/Michaeli71/JUGS-Best-of-Modern-Java-21-24>



Michael Inden

Head of Development, freiberuflicher Buchautor und Trainer

Speaker Intro



michael_inden@hotmail.com



- **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
 - **F**
 - **S**
 - **A**
- Open To Work**
Adcubum closed P&C dept.



<https://github.com/Michaeli71/JUGS-Best-of-Modern-Java-21-24>



Agenda

All Time Favorites + Meine Top 10 aus Java 21 LTS bis 24 (chronologisch)



All Time Favorites:

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

Meine Top 10 aus Java 21 LTS ...

1. Sequenced Collections (Java 21)
2. Record Patterns (Java 21)
3. Pattern Matching bei switch (Java 21)
4. Virtual Threads (Java 21)
5. Structured Concurrency (*Preview*) (Java 21/24)

... sowie Java 22, 23 und 24:

6. Unnamed Variables & Patterns (Java 22)
 7. Markdown Comments (Java 23)
 8. Stream Gatherers (Java 24)
 9. *Flexible Constructor Bodies (Preview)* (Java 24)
-
10. *Simple Source Files and Instance Main Methods (Preview)* (Java 24)





Build-Tools, IDEs und Sandbox



IDE & Tool Support für Java 24



- **Eclipse: Version 2025-03 (mit Plugin)**
- **IntelliJ: Version 2024.3.1**
- **Maven: 3.9.9, Compiler Plugin: 3.13.0**
- **Gradle: 8.14 (aber erst seit dem 25.04.2025 verfügbar)**
- **Aktivierung von Preview Features erforderlich**
 - In Dialogen
 - In Build Scripts



Maven™

 **Gradle**

The Gradle logo consists of a stylized blue elephant icon followed by the word "Gradle" in a bold, sans-serif font.

IDE & Tool Support Java 24



- Eclipse 2025-03 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 Installed Research at the Eclipse

Find: ←

Java 24 Support for Eclipse 2025-03 (4.35)

This marketplace solution provides Java 24 support for Eclipse 2025-03. Ensure you have the latest Eclipse release, which is... [more](#)

by [Eclipse Foundation](#), EPL 2.0

Installs: 457 (422 last month)

Properties for Java24Examples

Java Compiler

Enable project specific settings

JDK Compliance

Use compliance from execution environment on the [Java Build Path](#)

Compiler compliance level: → 24 (spin)

Use '--release' option

Use default compliance settings

Enable preview features for Java 24 ←

Preview features with severity level: → Info (spin)

Generated .class files compatibility: 24 (spin)

Source compatibility: 24 (spin)

Classfile Generation

Resource Builders Coverage Java Build Path Java Code Style Java Compiler Javadoc Location Java Editor Maven Project Facets Project Natures Project References Run/Debug Settings Server Task Repository Task Tags Validation WikiText

IDE & Tool Support Java 24



- IntelliJ 2024.3.1 / IntelliJ 2025.1
- Aktivierung von Preview Features erforderlich

Project Structure

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

Name:

SDK: 

Language level: 

Compiler output: 

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

Cancel Apply OK





- Aktivierung von Preview Features erforderlich

```
java {  
    toolchain {  
        languageVersion = JavaLanguageVersion.of(24)  
    }  
}  
  
tasks.withType(JavaCompile).configureEach {  
    options.compilerArgs += ["--enable-preview"]  
}  
  
tasks.withType(JavaCompile).configureEach {  
    options.compilerArgs += ["--enable-preview",  
                           "--add-modules", "jdk.incubator.vector"]  
}
```





- Aktivierung von Preview Features erforderlich

```
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.13.0</version>
    <configuration>
      <source>24</source>
      <target>24</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>24</release>
    <compilerArgs>
      <arg>--enable-preview</arg>
      <arg>--add-modules</arg>
      <arg>jdk.incubator.vector</arg>
    </compilerArgs>
  </configuration>
</plugin>
```



The Java Version Almanac

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



Details	Status	Documentation	Download	Compare API to											
Java 25	DEV	API Notes	JDK JRE	24	23	22	21	20	19	18	17	16	15	...	
Java 24	REL	API Lang VM Notes	JDK JRE	23	22	21	20	19	18	17	16	15	14	...	
Java 23	EOL	API Lang VM Notes	JDK JRE	22	21	20	19	18	17	16	15	14	13	...	
Java 22	EOL	API Lang VM Notes	JDK JRE	21	20	19	18	17	16	15	14	13	12	...	
Java 21	LTS	API Lang VM Notes	JDK JRE	20	19	18	17	16	15	14	13	12	11	...	
Java 20	EOL	API Lang VM Notes	JDK JRE	19	18	17	16	15	14	13	12	11	10	...	
Java 19	EOL	API Lang VM Notes	JDK JRE	18	17	16	15	14	13	12	11	10	9	...	
Java 18	EOL	API Lang VM Notes	JDK JRE	17	16	15	14	13	12	11	10	9	8	...	
Java 17	LTS	API Lang VM Notes	JDK JRE	16	15	14	13	12	11	10	9	8	7	...	
Java 16	EOL	API Lang VM Notes	JDK JRE	15	14	13	12	11	10	9	8	7	6	...	
Java 15	EOL	API Lang VM Notes	JDK JRE	14	13	12	11	10	9	8	7	6	5	...	
Java 14	EOL	API Lang VM Notes	JDK JRE	13	12	11	10	9	8	7	6	5	1.4	...	
Java 13	EOL	API Lang VM Notes	JDK JRE	12	11	10	9	8	7	6	5	1.4	1.3	...	
Java 12	EOL	API Lang VM Notes	JDK JRE	11	10	9	8	7	6	5	1.4	1.3	1.2	...	
Java 11	LTS	API Lang VM Notes	JDK JRE	10	9	8	7	6	5	1.4	1.3	1.2	1.1	...	
Java 10	EOL	API Lang VM Notes	JDK JRE	9	8	7	6	5	1.4	1.3	1.2	1.1	1.0	...	
Java 9	EOL	API Lang VM Notes	JDK JRE	8	7	6	5	1.4	1.3	1.2	1.1	1.0			
Java 8	LTS	API Lang VM Notes	JDK JRE	7	6	5	1.4	1.3	1.2	1.1	1.0				
Java 7	EOL	API Lang VM Notes	JDK JRE	6	5	1.4	1.3	1.2	1.1	1.0					
Java 6	EOL	API Lang VM Notes	JDK JRE	5	1.4	1.3	1.2	1.1	1.0						
Java 5	EOL	API Lang VM Notes		1.4	1.3	1.2	1.1	1.0							
Java 1.4	EOL	API		1.3	1.2	1.1	1.0								
Java 1.3	EOL	API		1.2	1.1	1.0									
Java 1.2	EOL	API Lang		1.1	1.0										
Java 1.1	EOL	API		1.0											
Java 1.0	EOL	API Lang VM													
Pre 1.0	EOL														

Data Source

Sandbox – <https://javaalmanac.io/>



Sandbox

Instantly compile and run Java 24 snippets without a local Java installation.

Java24.java ▶ Run 24+36

```
1 import java.lang.reflect.ClassFileFormatVersion;
2
3 void main() {
4     var v = ClassFileFormatVersion.latest();
5     System.out.printf("Hello Java bytecode version %s!", v.major());
6 }
```

Sandbox

Instantly compile and run Java 24 snippets without a local Java installation.

No Support
for Preview
Features!

Java24.java ▶ Run 24+36

```
Hello Java bytecode version 68!
```



All Time Favorites





ATF 1: Switch Expressions / Text Blocks / Records



Switch Expressions



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

Return-
Value

```
Day0fWeek day = Day0fWeek.FRIDAY;  
int num0fLetters = switch (day)  
{  
    case MONDAY, FRIDAY, SUNDAY -> 6;  
    case TUESDAY -> 7;  
    case THURSDAY, SATURDAY -> 8;  
    case WEDNESDAY -> 9;  
};
```

- Elegantere Schreibweise beim case:

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

Switch Expressions



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

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

Switch Expressions



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

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

SUNDAY is FUN DAY
6



Text Blocks



Text Blocks



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

Text Blocks



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

    result += createTable();
    result += createWordList();

    result += """
                    </body>
                </html>
        """;
}

return result;
}
```

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

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



Records



Enhancement Record

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



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

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

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

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

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

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

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

    // Zugriffsmethoden auf x und y
}
```

Records für komplexere Rückgabewerte und Parameter



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

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

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

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

Records für Pairs und Tupel



```
record IntIntPair(int first, int second) {};  
  
record StringIntPair(String name, int age) {};  
  
record Pair<T1, T2>(T1 first, T2 second) {};  
  
record Top3Favorites(String top1, String top2, String top3) {};  
  
record CalcResultTuple(int min, int max, double avg, int count) {};
```

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



ATF 2: Hilfreiche NullPointerExceptions / Pattern Matching bei instanceof



Hilfreiche NullPointerExceptions



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

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



Hilfreiche NullPointerExceptions



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

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

-XX:+ShowCodeDetailsInExceptionMessages

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

Hilfreiche NullPointerExceptions



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

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



Pattern Matching bei instanceof



Pattern Matching bei instanceof



- **ALT**

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

- **NEU**

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

Pattern Matching bei instanceof



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



Meine Top 10





Platz 1: Sequenced Collections



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();  
}
```

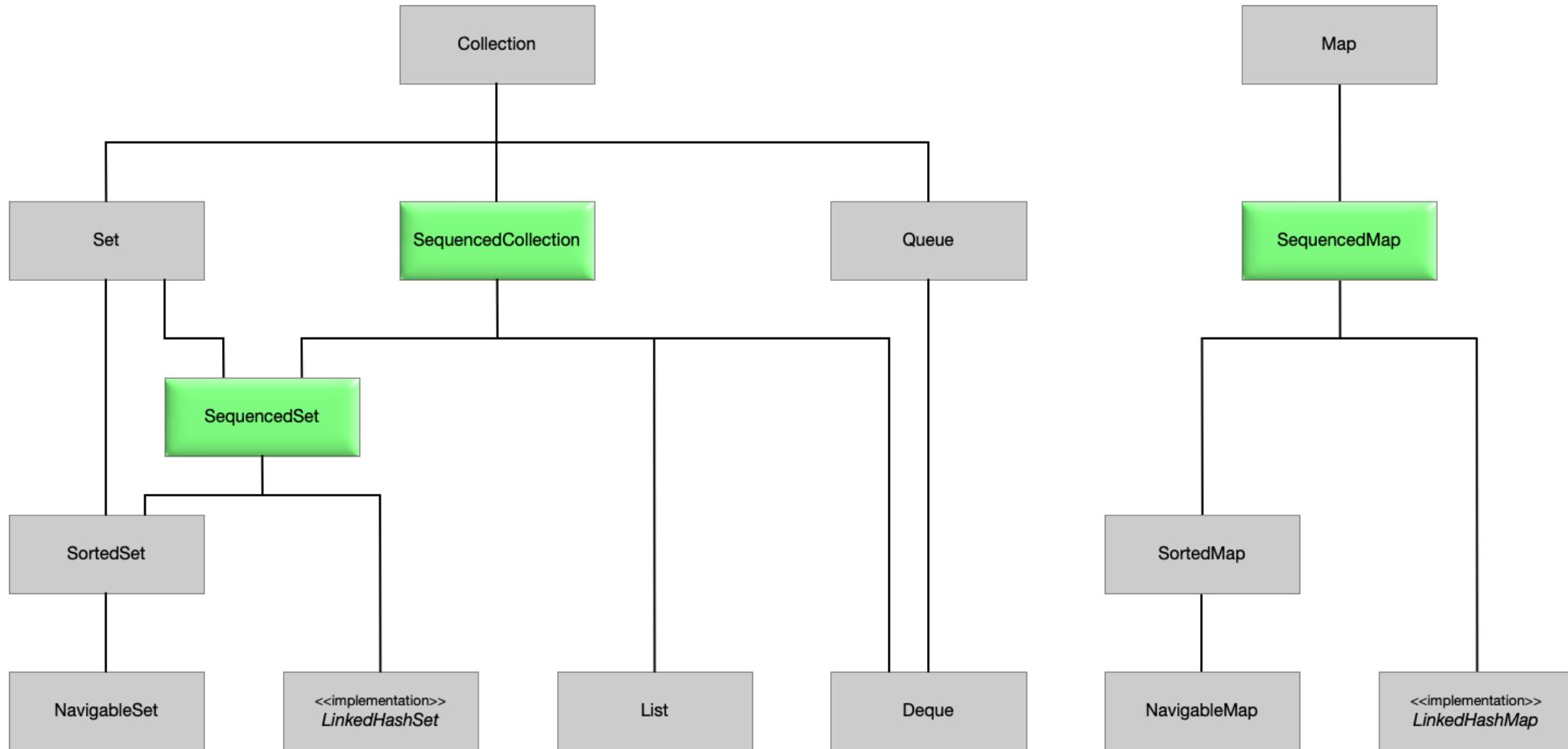
SequencedCollection defines the modifying methods:

- 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 – 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() die Methode firstEntry() anbietet und anstelle von removeLastEntry() die Methode 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

IHFEDCBA

Processing letterSequence with tree set

A / D

DCBA



Platz 2: 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.



- 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));
}
```

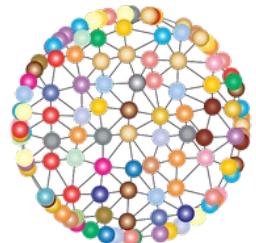


- Record Patterns können auch verschachtelt werden, um eine deklarative, leistungsfähige und kombinierbare Form der Datennavigation und -verarbeitung zu ermöglichen.

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

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

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



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



- Nehmen wir einmal folgende Records als Datenmodell an:

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

```
record Phone(String areaCode, String number) {  
}
```

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

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

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;
}
```



- 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

FlightReservationExample.java



Platz 3: Pattern Matching bei switch



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



- **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.



- 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 a Java code editor. The code is a switch statement with several case blocks. The third case block, which contains a condition 'str.length() > 5', is highlighted with a red underline, indicating a syntax error. A tooltip appears over this line, stating: 'Label is dominated by a preceding case label 'String str''. Another part of the tooltip says: 'Move switch branch 'String str && str.length() > 5' before 'String str'' and '© java.lang.String'. This illustrates that the pattern matching engine has detected that the third case is unreachable because it is dominated by the second case.

- 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.



- Probleme mit Konstanten (wurde mit Java 17 LTS 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 screenshot of an IDE showing a Java code editor. The code is a switch statement with two cases: one for strings longer than 5 characters and one for the constant "Sophie". A tooltip appears over the "default" branch, stating "Switch label '"Sophie"' is unreachable". Below the tooltip are buttons for "Remove switch branch '"Sophie"'", "More actions...", and a vertical ellipsis.

- 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());
```



- 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");  
}
```

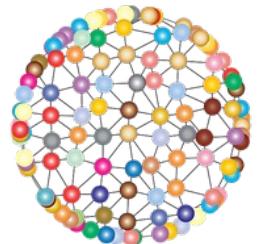
Problematik wird vom Compiler nicht erkannt

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



DEMO & Hands on

`SwitchMultiPatternExample.java`
`SwitchSpecialCasesExample.java`
`SwitchDominanceExample.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");
    }
}
```



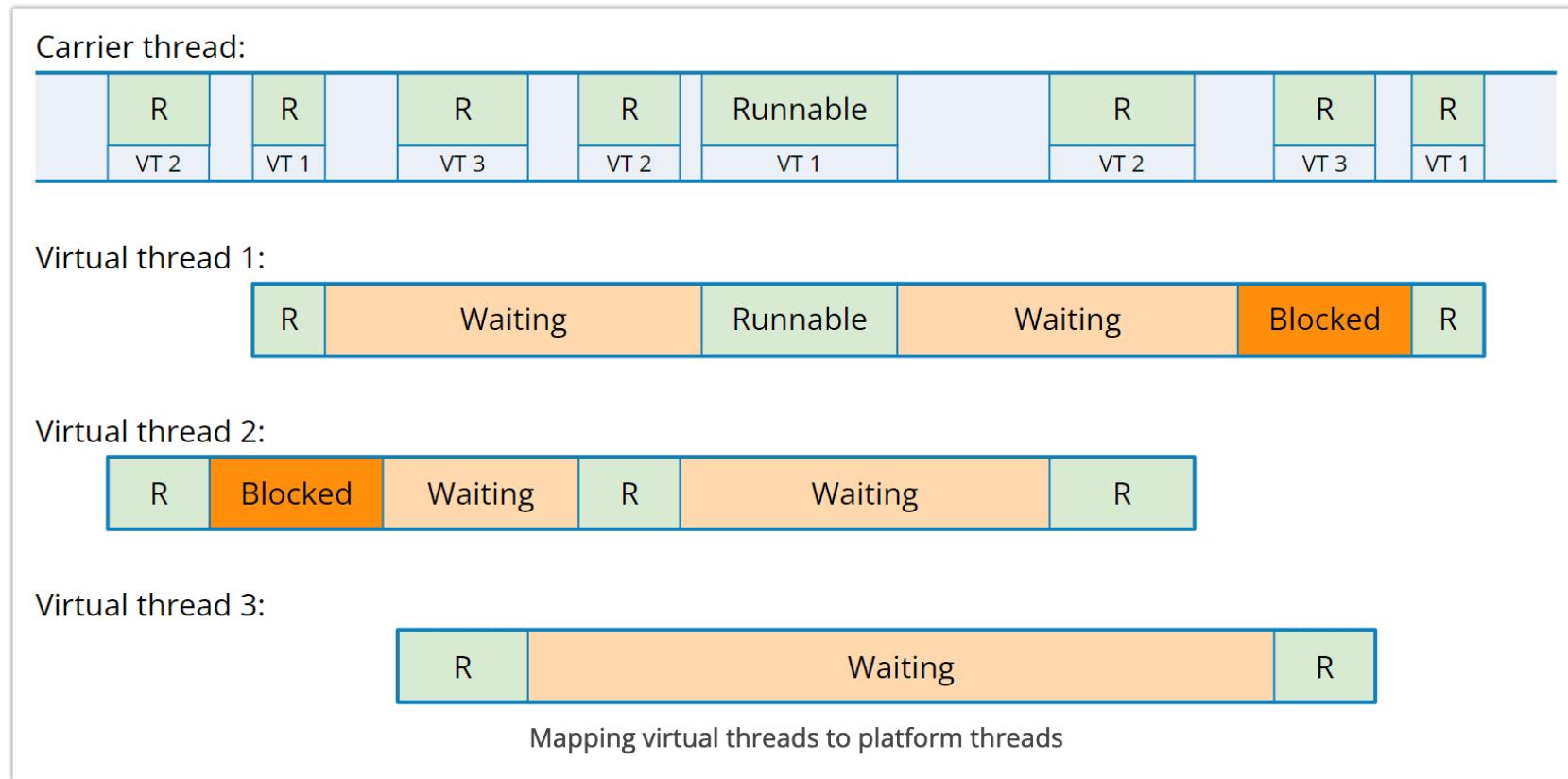
Platz 4: Virtual Threads



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.





- 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.



- 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");
}
```



- 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.

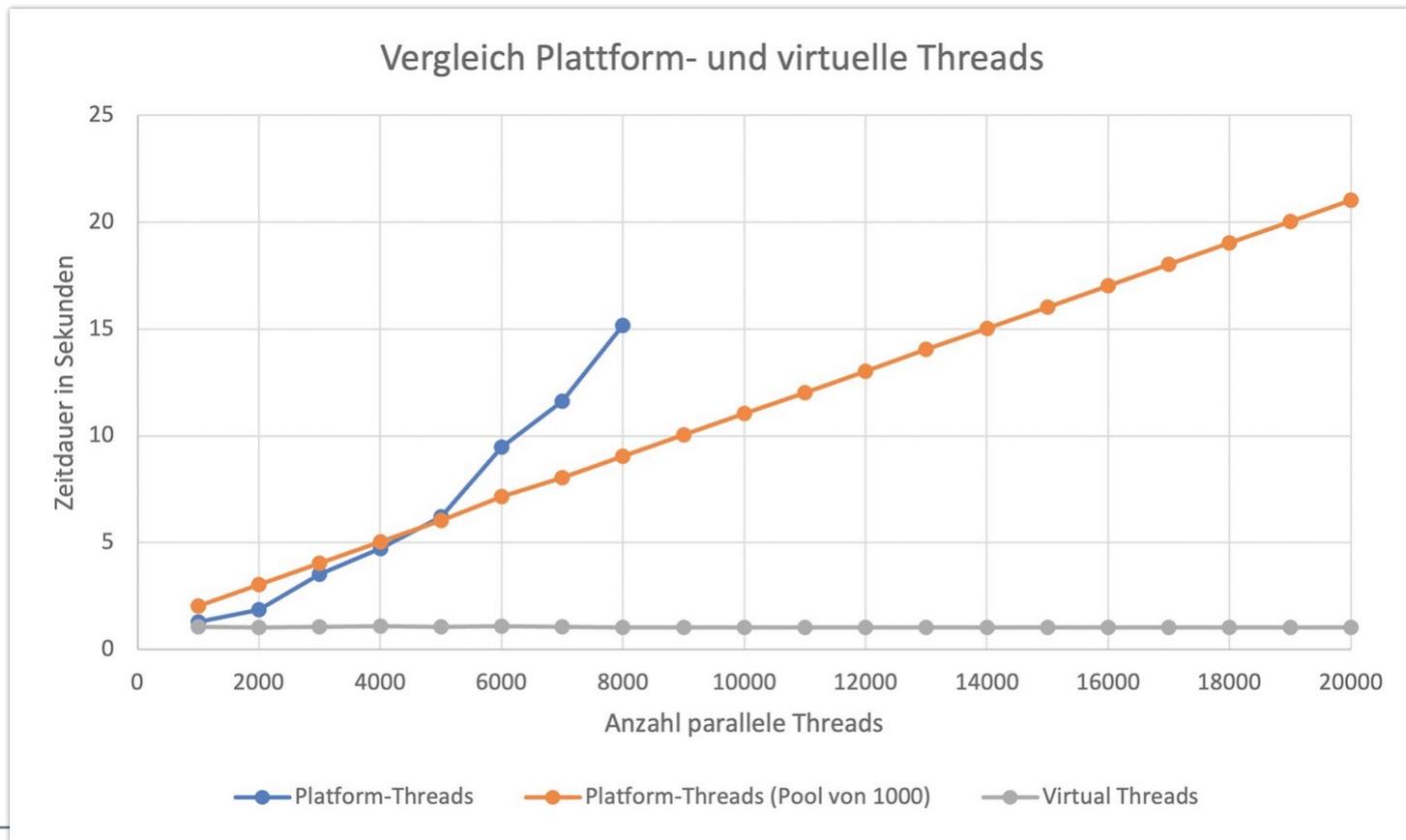
```
private static void submit1000Threads(ExecutorService executor) {  
    for (int i = 0; i < 1_000; i++) {  
        final int pos = i;  
        executor.submit(() -> {  
            Thread.sleep(Duration.ofSeconds(5));  
            return pos;  
        });  
    }  
}
```

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

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



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





DEMO & Hands on

FirstVirtualThreadsExample.java

PlatformThreadsExample.java

VirtualThreadsExample.java

VirtualThreadsPoolingExample.java



Platz 5: Structured Concurrency





- Dieser JEP bringt die Structured Concurrency als Vereinfachung von Multithreading.
 - 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.
 - Wenn eine Aufgabe aus mehreren Teilaufgaben besteht, die parallel verarbeitet werden können, ermöglicht Structured Concurrency, dies auf besonders lesbare und wartbare Weise umzusetzen.
-



- **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.**



```
static Response handleOldStyle(Long userId) throws ExecutionException,  
                                         InterruptedException  
{  
    var executorService = Executors.newCachedThreadPool();  
  
    var userFuture = executorService.submit(() -> findUser(userId));  
    var ordersFuture = executorService.submit(() -> fetchOrders(userId));  
  
    var user = userFuture.get();           // Join findUser  
    var orders = ordersFuture.get();      // Join fetchOrders  
  
    return new Response(user, orders);  
}
```

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



- Umsetzung mit Structurd Concurrency in Form der Klasse StructuredTaskScope:

```
static Response handle(Long userId) throws ExecutionException,  
                                         InterruptedException  
{  
    try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {  
        var userSubtask = scope.fork(() -> findUser(userId));  
        var ordersSubtask = scope.fork(() -> fetchOrder(userId));  
  
        scope.join();           // Join both forks  
        scope.throwIfFailed(); // ... and propagate errors  
  
        // 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.



Die Klasse `StructuredTaskScope` besitzt zwei Spezialisierungen:

- `ShutdownOnFailure` – fängt die erste `Exception` ab und beendet den `StructuredTaskScope`. Diese Strategie 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.



- 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("PS " + result1.state() + "/FS " + result2.state() +
                       "/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)

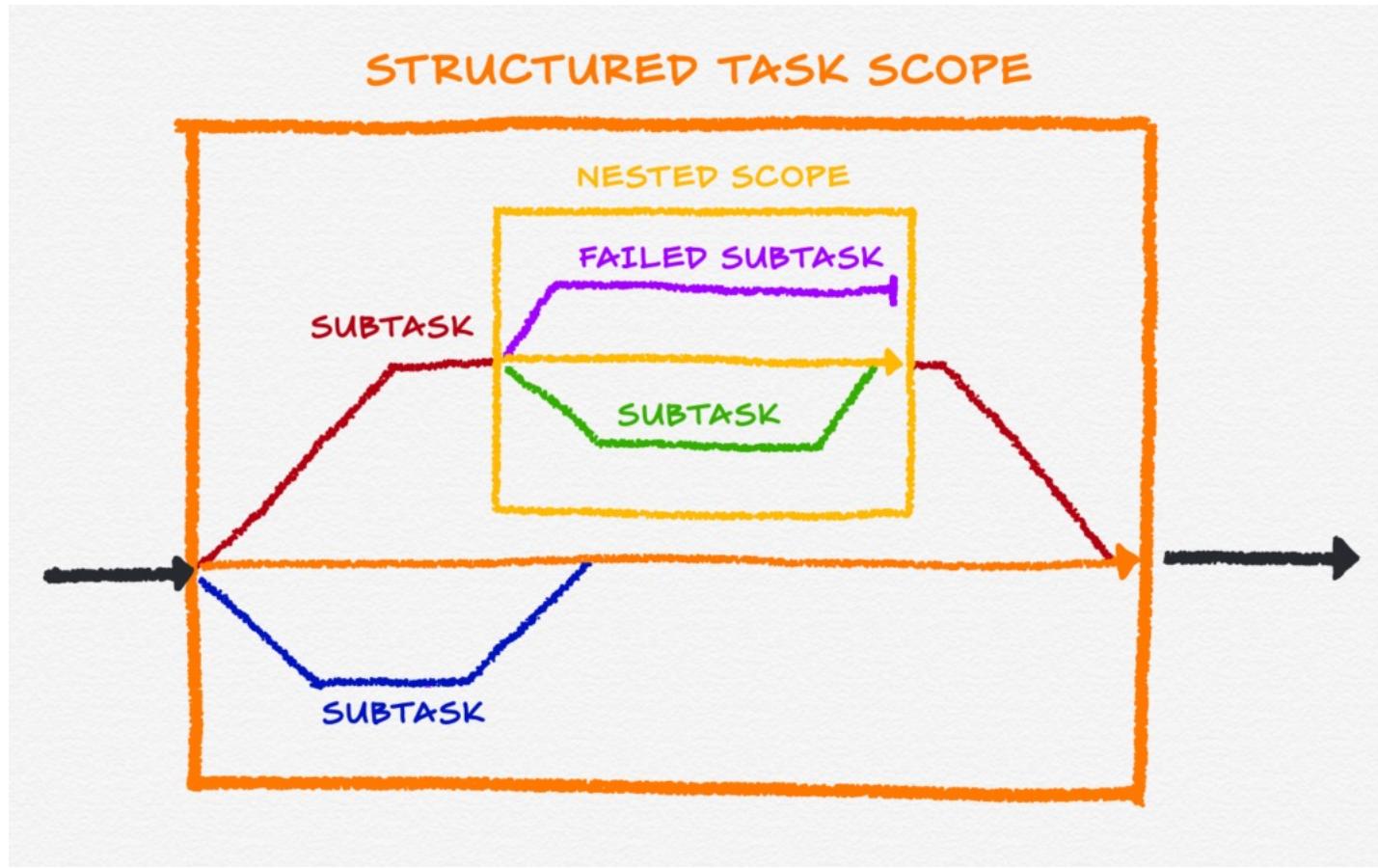


DEMO & Hands on

StructuredConcurrency.java



- Structured Concurrency kann man auch verschachteln:



JEP 499: Structured Concurrency



```
private static Proposal performCityTripProposals(String city, LocalDate startDate)
    throws InterruptedException, ExecutionException
{
    try (var scope = new StructuredTaskScope.ShutdownOnFailure())
    {
        var hotelSubtask = scope.fork(() -> findHotels(city, startDate));
        var carRentalSubtask = scope.fork(() -> fetchCarRentals(city, startDate));

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

        // Here, both forks have succeeded, so compose their results
        return new Proposal(hotelSubtask.get(), carRentalSubtask.get());
    }
}
```



- **Erster Versuch: Finde n Vorschläge für ein Car Rental:**

```
private static List<String> fetchCarRentals(String city, LocalDate startDate)
    throws InterruptedException, ExecutionException
{
    // Achtung: ShutdownOnSuccess muss passend den generischen Parameter bekommen
    try (var scope = new StructuredTaskScope.ShutdownOnSuccess<String>())
    {
        var proposal1 = scope.fork(() -> findProposalEuropcar(city, startDate));
        var proposal2 = scope.fork(() -> findProposalAvis(city, startDate));
        var proposal3 = scope.fork(() -> findProposalHertz(city, startDate));

        scope.join();

        // Hier kommt der Wunsch nach FirstNSuccessful auf ...
        return List.of(scope.result());
    }
}
```



- **Einsatz eines eigenen TaskScopes:**

```
private static List<String> fetchCarRentals(String city, LocalDate startDate)
    throws InterruptedException, ExecutionException
{
    try (var scope =
        new StructuredConcurrentOwnTaskScopeExample.FirstNSuccessful<String>(2))
    {
        var proposal1 = scope.fork(() -> findProposalEuropcar(city, startDate));
        var proposal2 = scope.fork(() -> findProposalAvis(city, startDate));
        var proposal3 = scope.fork(() -> findProposalHertz(city, startDate));

        scope.join();

        return scope.result();
    }
}
```



DEMO & Hands on

NestedStructuredConcurrencyExample.java
StructuredConcurrentTaskScopeExample.java



Übungen PART 1

<https://github.com/Michaeli71/JUGS-Best-of-Modern-Java-21-24>





Platz 6: Unnamed Patterns and Variables



JEP 456: Unnamed Patterns and Variables



- 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) { }
```



- 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 456: Unnamed Patterns and Variables



- 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



- 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
    // unbekannte 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.

*(Hinweis Python)



- 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)



- **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.



- **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 + "'");
}
```



- **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 _)**



- **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 _)**



- **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.**



- **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 456: Unnamed Patterns and Variables



```
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;
}
```



Platz 7: Markdown 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;
}
```



- **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



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

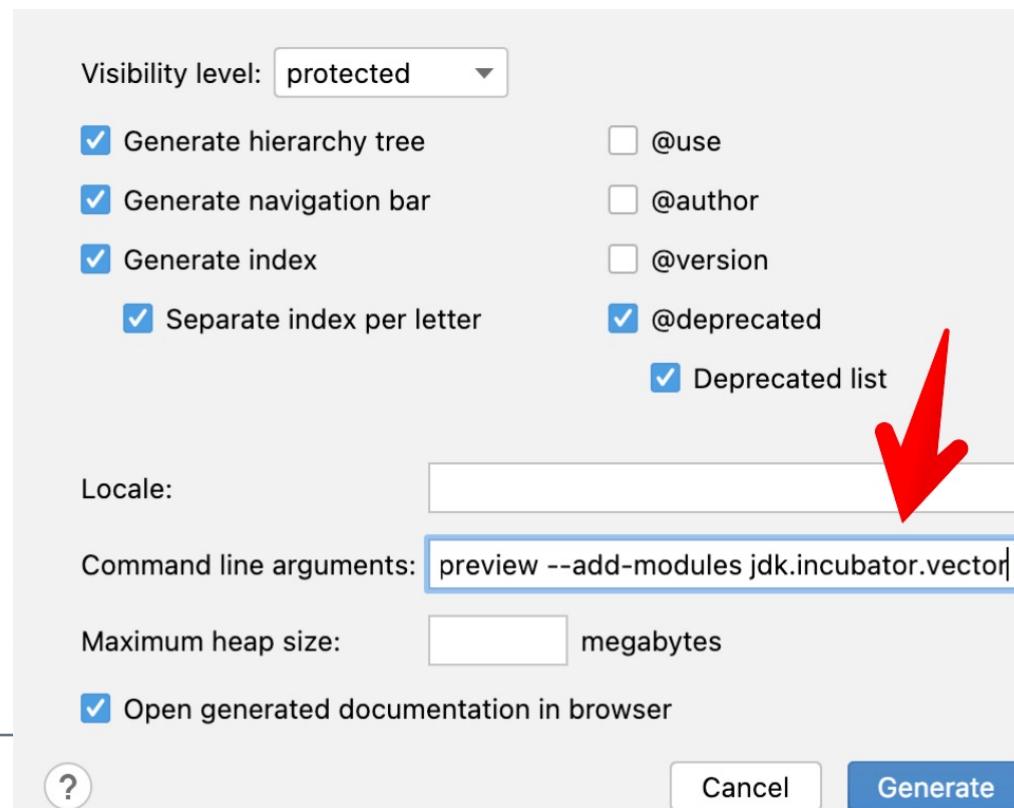
```
/// / Latein / Griechisch /
/// /-----/-----/
/// / a      / &alpha; (alpha) /
/// / b      / &beta; (beta) /
/// / c      / &gamma; (gamma) // &Gamma; /
/// / ...    / ... /
/// / z      / &omega; (omega) /
```

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**



▼	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`.



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



Platz 8: Stream Gatherers





- Die in Java 8 eingeführten Streams waren bereits von Anfang an recht mächtig.
- Danach 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.
- Wünschenswert, um Aufgabenstellungen realisieren zu können, die zuvor nicht ohne Weiteres oder nur mit Tricks sowie eher umständlich umzusetzen waren.



- 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 485: Stream Gatherers Motivation



- 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());  
    }  
}
```



- 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 nur in spezifischen Kontexten sinnvoll. Im Allgemeinen würden diese das Stream-API ziemlich aufblähen und den Einstieg in das (ohnehin schon umfangreiche) API (weiter) erschweren.



- 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 485: 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 485: 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]]
```



- 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 485: Stream Gatherers – fold()



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

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

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

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



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

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

- Als Ausgabe ergibt sich die Folgende:

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



- Sollen die Elemente eines Streams zu neuen Kombinationen zusammengeführt werden, sodass jeweils immer ein Element dazukommt, dann ist dies 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, 1223334444555556666667777777]
```



DEMO & Hands on



- Transformation – Mit Stream Gatherers lassen sich die Elemente in verschiedener Weise in Ausgabeelemente umwandeln: 1:1, 1:n, n:1 oder n:m. Es lassen sich aus einem Eingabeelement ein Ausgabeelement oder mehrere produzieren, oder aber aus mehreren Eingabeelementen ein oder mehrere Ausgabeelement(e) erzeugen.
 - n:m – Window fixed/sliding
 - n:1 – Fold
 - 1:1 – Scan
 - 1:n -- ??
- Zustandsinformationen – Mitunter erfordert die Verarbeitung von Daten Informationen zu zuvor verarbeiteten Elementen. Bislang war es in Streams nur über Tricks möglich, Zustandsinformationen zu verarbeiten. Praktischerweise erlauben es Stream Gatherers, Zustand nutzen zu können. Damit lassen sich Informationen zu Anzahl, Inhalt o. ä. sammeln, um die Transformation nachfolgender Elemente zu beeinflussen, etwa abhängig von der Position oder wenn eine Bedingung erfüllt ist.



- Neben den besprochenen vordefinierten Gatherers gibt es die Methode `mapConcurrent()`, die funktional wie die bereits seit den Anfangstagen des Stream-APIs existierende Methode `map()` arbeitet, allerdings im Unterschied zu dieser Eingabeelemente mithilfe von Virtual Threads parallel gemäß der angegebenen Aktion umwandelt.

```
Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10).  
    gather(Gatherers.mapConcurrent(5, x -> x * x)).  
    toList();
```

- =>

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

- Herkömmlich:

```
Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10).  
    parallel().  
    map(x -> x * x).  
    toList();
```



- **Herkömmlicher Ansatz:**

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

- => **Wartezeit von ca. 30 Sekunden, wenn `lookup()` ca. 1 Sekunde benötigt**
- **Vorteile durch Steuerung der Parallelität, Umstellung auf Virtual Threads reduziert Wartezeit auf ca. 2 Sekunden:**

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

- **Man mit der Anzahl der Parallelität experimentierenm 500 reduziert auf 1 Sekunde**
-



DEMO & Hands on



Eigener Gatherer

Bisher Beispiele für 1:1, n:1 und n:m.

Der Fall, dass aus einem Eingabeelement n Ausgabeelemente entstehen, wird nun in Form eines selbstgeschriebenen Triple-Gatherer präsentiert, der Werte verdreifacht, also in die Kategorie 1:n gehört.



- Um eigene Stream Gatherers zu realisieren, müssen diese das Interface Gatherer implementieren. Dabei gibt es vier elementare Steuermöglichkeiten, die in Form von gleichnamigen Methoden das Verhalten je nach Bedarf anzupassen:

```
public interface Gatherer<T, A, R>
{
    default Supplier<A> initializer() {
        return defaultInitializer();
    }

    Integrator<A, T, R> integrator();

    default BinaryOperator<A> combiner() {
        return defaultCombiner();
    }

    default BiConsumer<A, Downstream<? super R>> finisher() {
        return defaultFinisher();
    }

    ...
}
```



- **integrator()** – Bildet das Herzstück und definiert die auszuführenden Aktionen zur Verarbeitung
- **initializer()** – Sofern Zustandsinformationen benötigt werden, erfolgt hier die Bereitstellung eines initialen Zustands
- **finisher()** – Ermöglicht zum Abschluss der Aktionen weitere Berechnungen oder Auswertungen
- **combiner()** – Sofern Stream Gatherers in Parallel-Streams verwendet werden, dann ist es Aufgabe dieser Methode festzulegen, wie die einzelnen Resultate von parallelen Berechnungsschritten miteinander verknüpft werden sollen.

```
public interface Gatherer<T, A, R>
{
    default Supplier<A> initializer() {
        return defaultInitializer();
    };

    Integrator<A, T, R> integrator();

    default BinaryOperator<A> combiner() {
        return defaultCombiner();
    }

    default BiConsumer<A, Downstream<? super R>> finisher() {
        return defaultFinisher();
    }
}
```



- Errinnern wir uns: Die zentrale Stelle für eigene Aktionen ist die Methode `integrator()`:

```
public interface Gatherer<T, A, R>
{
    ...
    Integrator<A, T, R> integrator();
    ...
}
```

- Schauen wir auf das entsprechende Interface:

```
interface Integrator<A, T, R>
{
    boolean integrate(A state, T element,
                      Downstream<? super R> downstream);
}
```

Für jedes Element des Streams wird `integrate()` mit Zustandsinformationen vom Typ A, dem aktuellen Element mit Typ T und einem Verweis auf einen Stream vom Typ R, um dorthin verarbeitete Elemente weiterzugeben, aufgerufen. Der Rückgabewert signalisiert, ob nach dem aktuellen Element noch weitere Elemente verarbeitet werden sollen.

JEP 485: Stream Gatherers – Eigene Gatherer



```
<T> Gatherer<T, Object, T> triple()
{
    return new Gatherer<T, Object, T>()
    {
        @Override
        public Integrator<Object, T, T> integrator()
        {
            return new Integrator<Object, T, T>()
            {
                @Override
                public boolean integrate(Object unusedState, T element,
                                         Downstream<? super T> downstream) {
                    downstream.push(element);
                    downstream.push(element);
                    downstream.push(element);
                    return true;
                }
            };
        }
    };
}
```



- Definieren wir folgende main()-Methode, die zeigt, dass dieser eigene Stream Gatherer auch mit unterschiedlichen Typen arbeitet:

```
void main()
{
    var result = Stream.of("This", "is", "a", "test").
                    gather(triple()).
                    toList();
    println(result);

    var result2 = Stream.of(1, 2, 3, 4, 5).
                    gather(triple()).
                    toList();
    println(result2);
}
```

- =>

```
[This, This, This, is, is, is, a, a, a, test, test, test]
[1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 5, 5, 5]
```



- Weitere Möglichkeiten durch Zustandshaltung, etwa eines Zählers:

```
public Supplier<AtomicInteger> initializer()
{
    return () -> new AtomicInteger(0);
}
```

- Komfort durch Konstruktionsmethoden und Nutzung von Lambdas

```
<T> Gatherer<T, AtomicInteger, T> every2ndTripleAgain()
{
    return Gatherer.ofSequential(
        AtomicInteger::new,
        (state, element, downstream) ->
    {
        if (state.getAndIncrement() % 2 == 0)
        {
            downstream.push(element);
            downstream.push(element);
            downstream.push(element);
        }
        return true;
    },
    Gatherer.defaultFinisher()
);
}
```

Zwar ist es zum Gewinnen von tieferen Einblicken und Einsichten sinnvoll, gewisse Funktionalitäten einmal (prototypisch) selbst zu implementieren, allerdings sollte man das Rad nicht neu erfinden, wenn es schon vorgefertigte, gut getestete Bibliotheken gibt. Für Stream Gatherers gibt es folgende:

- <https://github.com/pivovarit/more-gatherers>
- <https://github.com/tginsberg/gatherers4j>



Platz 9: Flexible Constructor Bodies



JEP 492: 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 492: 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 492: Statements before super(...) / Flexible Constructor Bodies



```
public class PositiveBigIntegerOld1 extends BaseInteger
{
    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 492: Statements before super(...) / Flexible Constructor Bodies



- **Herkömmliche Abhilfe: statische Hilfsmethode**

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

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

        return value;
    }
}
```

JEP 492: Statements before super(...) / Flexible Constructor Bodies



- Die Argumentprüfung wird deutlich besser lesbar und verständlich, wenn die Validierungslogik direkt im Konstruktor noch vor dem Aufruf von `super()` geschieht.
- Mittlerweile 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 492: Statements before super(...) / Flexible Constructor Bodies



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

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

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

JEP 492: Statements before super(...) / Flexible Constructor Bodies



- 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());
    }
}
```



- 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 492: Statements before super(...) / 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 492: Statements before super(...) / 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
-



Platz 10: Simple Source Files and Instance Main Methods



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.



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!");
    }
}
```



Vereinfachung II: Unnamed class

- Wenn nur einfache dann kann man mit 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!");  
}
```

- 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!");  
}
```



- 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



- **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



- 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("Static Main");  
    }  
}
```

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

c jep477_Implicitly_Declared_Classes.InstanceMainMethodExample3

public static void main()

Java23Examples





- 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



- 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!'");
}
```



- **Konsolen-Interaktion ist in der Regel umständlich und kompliziert, vor allem 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.*;
```



- 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.*;
```



- 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  
    }  
}
```



Übungen PART 2

<https://github.com/Michaeli71/JUGS-Best-of-Modern-Java-21-24>





Ausblick (Bisherige) JEPs in Java 25 LTS

Java 25 LTS – Was wird inkludiert sein?



Targeted

- JEP 502: **Stable Values (Preview)**
- JEP 503: **Remove the 32-bit x86 Port**
- JEP 505: **Structured Concurrency (Fifth Preview)**
- JEP 511: **Module Import Declarations**
- JEP 512: **Compact Source Files and Instance Main Methods**
- JEP 513: **Flexible Constructor Bodies**

Proposed to Target

- JEP 506: **Scoped Values**
- JEP 508: **Vector API (Tenth Incubator)**
- JEP 510: **Key Derivation Function API**

...

Java 25 LTS

Java 25 LTS – Was könnte inkludiert sein?



Candidate

- JEP 504: Remove the Applet API
- JEP 507: Primitive Types in Patterns, instanceof, and switch (Third Preview)
- JEP 509: JFR CPU-Time Profiling (Experimental)
- JEP 514: Ahead-of-Time Command-Line Ergonomics
- JEP 515: Ahead-of-Time Method Profiling
- JEP 516: Ahead-of-Time Object Caching with Any GC
- JEP 517: HTTP/3 for the HTTP Client API
- JEP 518: JFR Cooperative Sampling

Java 25 LTS



Fazit



Positives



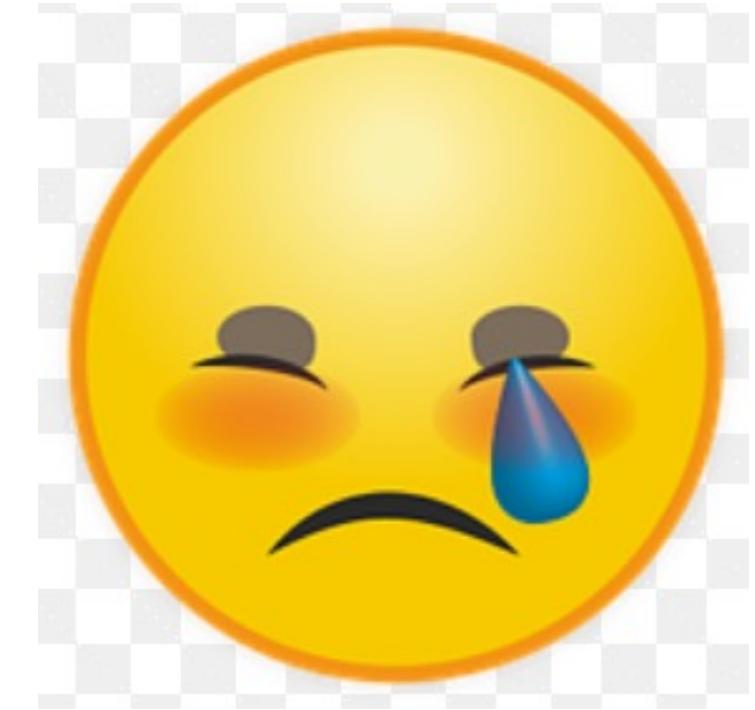
- **Stabile und zuverlässige 6-monatige Release-Zyklen und alle 2 Jahre LTS-Versionen**
- **Java wird einfacher und attraktiver**
- **Viele schöne Verbesserungen in Syntax und APIs wie switch, Records, Text Blocks usw.**
- **Pattern Matching und Record Patterns final in Java 21**
- **HTTP/2, virtuelle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



Negatives



- **Releases waren zwar pünktlich, aber manchmal eher 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**
- **Warum ist die Syntax von Pattern Matching bei instanceof und switch inkonsistent?**







Questions?



Thank You