



---

# JUGS – Best of Modern Java 21

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



**Michael Inden**

**Head of Development, freiberuflicher Buchautor und Trainer**

# Speaker Intro



E-Mail: [michael\\_inden@hotmail.com](mailto: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**
- **Freiberuflicher Consultant, Trainer und Konferenz-Speaker**
- **Seit Januar 2022 Head of Development bei Adcubum in Zürich**
- **Autor und Gutachter beim dpunkt.verlag / O'Reilly / APress**



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



# Agenda

# Workshop Agenda

---



- **Vorbemerkungen / Build Tools & IDEs**
- **«Normale Features»**
  - JEP 431: Sequenced Collections
  - JEP 440: Record Patterns
  - JEP 441: Pattern Matching for switch
  - JEP 444: Virtual Threads
- **Preview Features und Incubators**
  - JEP 430: String Templates (Preview)
  - JEP 443: Unnamed Patterns and Variables (Preview)
  - JEP 445: Unnamed Classes and Instance Main Methods (Preview)
  - JEP 448: Vector API (Sixth Incubator)
  - JEP 453: Structured Concurrency (Preview)





---

# Build-Tools und IDEs



# Aktuelles Java 21 installiert

---



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

```
$ java -version
openjdk version "21" 2023-09-19
OpenJDK Runtime Environment (build 21+35-2513)
OpenJDK 64-Bit Server VM (build 21+35-2513, 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

---



- Eclipse: Version 2023-09 mit Plugin
- IntelliJ: Version 2023.2.3
- Maven: 3.9.5, Compiler Plugin: 3.11.0
- Gradle: 8.4
- Aktivierung von Preview-Features / Incubator nötig
  - In Dialogen
  - Im Build Scripts



**Maven™**

 **Gradle**

# IDE & Tool Support Java 21



- Eclipse 2023-09 mit Plugin
- Aktivierung von Preview-Features nötig



Eclipse Marketplace

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

Search Recent Popular Favorites Installed Research at the top

Find:

**Java 21 Support for Eclipse 2023-09 (4.1.0)**

This marketplace solution provides Java 21 support... [more info](#)

by [Eclipse Foundation](#), EPL 2.0

0 installs: 209 (209 last month)

**Eclipse Marketplace**

type filter text

- > Resource Builders Coverage Java Build Path
- > Java Code Style
- > **Java Compiler**
- Javadoc Location
- > Java Editor Project Facets Project Natures Project References Refactoring History Run/Debug Settings
- > Task Repository Task Tags Validation
- > WikiText

**Java Compiler**

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

**JDK Compliance**

Use compliance from execution environment 'JavaSE-20' on the [Java Build Path](#)

Compiler compliance level:  →

Use '--release' option

Use default compliance settings

Enable preview features for Java 21 ←

Preview features with severity level:  →

Generated .class files compatibility:  →

Source compatibility:  →

Disallow identifiers called 'assert':

Disallow identifiers called 'enum':

# 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:  (arrow pointing to this field)

SDK:  (arrow pointing to this dropdown) Edit

Language level:  ▾

Compiler output:

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

**Project Settings**

- Project (selected)
- Modules
- Libraries
- Facets
- Artifacts

**Platform Settings**

- SDKs
- Global Libraries

Problems





- 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



- 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>
```



# IDE & Tool Support Java 21 für Vector API Incubator



Run Configurations

Create, manage, and run configurations  
Run a Java application

Name: VectorApiExample

Main Arguments JRE Dependencies Source Environment Common Prototype

Program arguments:

VM arguments:

--add-modules jdk.incubator.vector

Use the -XstartOnFirstThread argument when launching with SWT

Use the -XX:+ShowCodeDetailsInExceptionMessages argument when launching

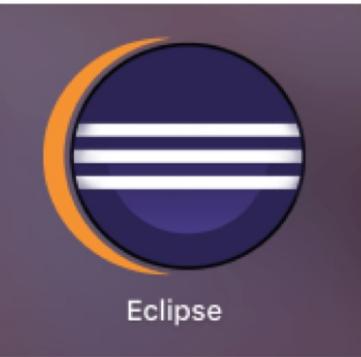
Use @argfile when launching

Working directory:

Default: \${workspace\_loc:Java18Examples}

Other: \_\_\_\_\_

Show Command Line Revert Apply



# IDE & Tool Support für Vector API Incubator



Screenshot of the IntelliJ IDEA Preferences dialog showing Java Compiler settings for the Vector API Incubator.

The left sidebar shows the navigation tree:

- Appearance & Behavior
- System Settings
- File Colors
- Scopes
- Notifications
- Quick Lists
- Path Variables
- Presentation Assistant
- Keymap
- Editor
- Plugins
- Version Control
- Build, Execution, Deployment
- Build Tools
- Compiler
- Excludes
- Java Compiler** (selected)
- Annotation Processors
- Validation
- RMI Compiler
- Groovy Compiler

The main panel displays the "Java Compiler" preferences:

- Use compiler: Javac
- Checkmark: 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
Java21Examples	21
- Javac Options:
  - Checkmarks: Use compiler from module target JDK when possible, Generate debugging info, Report use of deprecated features
  - unchecked: Generate no warnings
- Additional command line parameters: ('/' recommended in paths for cross-platform configurations)  
Value: `--enable-preview --add-modules jdk.incubator.vector`
- Override compiler parameters per-module:

Module	Compilation options
Java21Examples	--enable-preview --add-modules jdk.incubator.vector

Buttons at the bottom: ? (Help), Cancel, Apply, OK.

To the right of the dialog is the IntelliJ IDEA logo.

# IDE & Tool Support für Vector API Incubator



The screenshot shows the IntelliJ IDEA interface for configuring a run/debug session. The configuration is set up for an 'Application' named 'VectorApiExample'. A red arrow points to the 'Program arguments' field, which contains the command-line options: `--enable-preview --add-modules jdk.incubator.vector`. This configuration is intended to enable the Java Vector API incubator modules.

**Run/Debug Configurations**

Name: VectorApiExample  Store as project file

**Build and run**

java 21 SDK of 'Java21E' `--enable-preview --add-modules jdk.incubator.vector`

api.VectorApiExample

Program arguments

Working directory: /Users/michaelinden/java8-workspace/Java21Examples

Environment variables:

Separate variables with semicolon: VAR=value; VAR1=value1

Open run/debug tool window when started

Code Coverage  Modify

Packages and classes to include in coverage data

Run Apply Cancel OK

Edit configuration templates...

IntelliJ IDEA



# JEPs in Java 18, 19, 20 und 21

# 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



- 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 – 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

Total: 15

Normal: 8

Preview: 6

Incubator: 1



---

# Normale Features in Java 21

- 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, Set und Map
- Was fehlt, ist so etwas wie eine geordnete Reihenfolge der Elemente
- Wir beobachten, dass einige Sammlungen eine Begegnungsreihenfolge haben, d. h. es ist definiert, in welcher Reihenfolge die Elemente durchlaufen werden
  - Von vorne nach hinten, indexbasiert für Listen
  - HashSet hat keine Begegnungsreihenfolge
  - TreeSet definiert sie indirekt durch Comparable oder übergebenen Comparator
  - LinkedHashSet behält die Einfügereihenfolge bei

# Sequenced Collections – Motivation

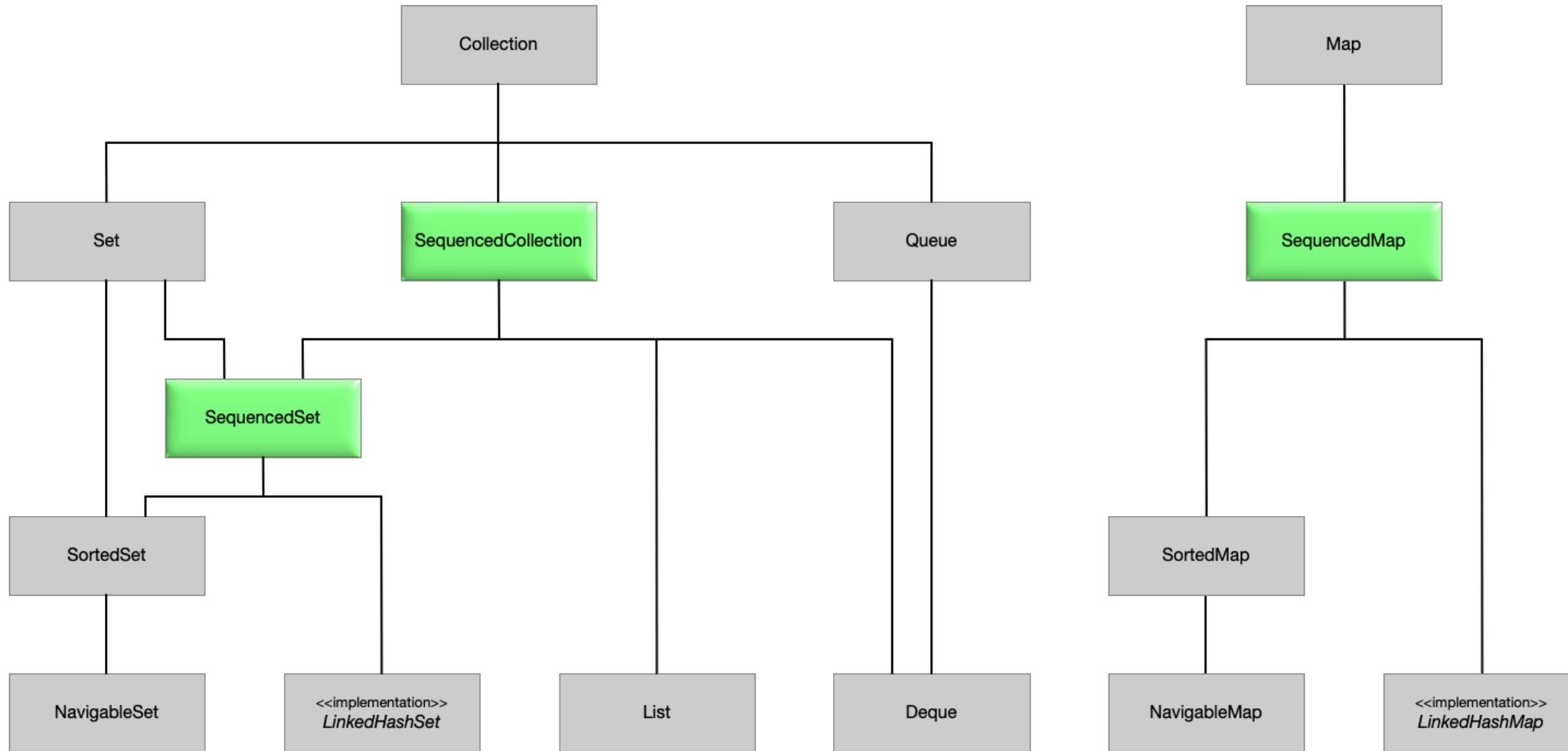


- In der Vergangenheit gab es mehrere Möglichkeiten, auf das erste oder letzte Element zuzugreifen

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

- Unübersichtlich und fehleranfällig
- Als Abhilfe gibt es nun "Sequenced Collections", die eine Collection repräsentieren, deren Elemente eine bestimmte Reihenfolge haben.

# Sequenced Collections – Integration bestehende Typenhierarchie



# 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 – 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();  
}
```

SequencedMap API does not fit that well. It uses NavigableMap as a base, so instead of `getFirstEntry()` it offers `firstEntry()`, and instead of `removeLastEntry()` it defines `pollLastEntry()`. As mentioned these names are not according to SequencedCollection. But trying to do this would have caused NavigableMap to get four new methods that do the same thing as four other methods it already has.

## Sequenced Collection – In Aktion



```
public static void sequenceCollectionExample() {  
    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 sequenceSetExample() {  
    // Plain Sets do not have encounter order ... run multiple time to see variation  
    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 has order  
    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



---

# JEP 440: Record Patterns



# 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: %d y: %d, sum: %d".formatted(x, y, 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 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 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 p, Color c) {}
record Rectangle(ColoredPoint upperLeft, ColoredPoint lowerRight) {}
```

```
static void printColorOfUpperLeftPoint(Rectangle rect)
{
    if (rect instanceof Rectangle(ColoredPoint(Point p, Color c),
                                  ColoredPoint lr))
    {
        System.out.println(c);
    }
}
```



---

# DEMO & Hands on

Jep405\_RecordPatternsExample.java

Jep405\_InstanceofRecordMatchingAdvanced.java

---



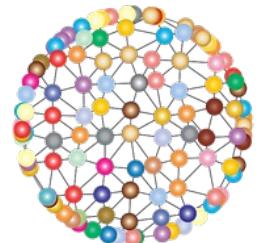
- Record Patterns können für Eleganz sorgen:

```
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!



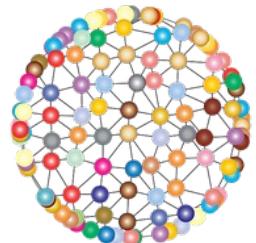
**Wie kann man das Ganze noch  
eleganter gestalten?**



- **Sauberer 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?**



- 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 from,  
                         City destination) {  
}
```



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

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

            if (reservation.destination() != null)
            {
                LocalDate birthday = person.birthday();
                String languageCode = destination.languageCode();

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

Jep405\_FlighReservationExample.java



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

# JEP 405/440: Record Patterns



```
boolean checkAgeAndDestinationLanguageCorrectNew(Object obj)
{
    if (obj instanceof FlightReservation(
        Person(String firstname, String lastname, LocalDate birthday),
        Phone phoneNumber, City from,
        City(String name, String country, String languageCode)))
    {
        if (birthday != null && languageCode != null)
        {
            long years = ChronoUnit.YEARS.between(LocalDate.now(), birthday);
            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.**



Insgesamt bot Java 19 die folgenden drei Möglichkeiten, Pattern Matching für Records durchzuführen:

- 1) Pattern Matching – Zugriff über Variable und Methoden
- 2) Record Pattern – Dekomposition in Einzelbestandteile
- 3) Named Record Pattern – Kombination aus 1) und 2)

```
record StringIntPair(String name, int value) {}

Object obj = new StringIntPair("Michael", 52);

// 1. Pattern Matching
if (obj instanceof StringIntPair pair) {
    System.out.println("object is a StringIntPair, name = " +
                       pair.name() + ", value = " + pair.value());
}

// 2. Record Pattern
if (obj instanceof StringIntPair(String name, int value)) {
    System.out.println("object is a StringIntPair, " +
                       "name = " + name + ", value = " + value);
}

// 3. Named Record Pattern
if (obj instanceof StringIntPair(String name, int value) pair) {
    System.out.println("object is a StringIntPair, name = " +
                       pair.name() + ", value = " + pair.value() +
                       "// name = " + name + ", value = " + value);
}
```

Java 20



- Keine Unterstützung für Named Record Patterns mehr

```
if (obj instanceof Person(var firstname, var lastname,  
                         var dateOfBirth, var eyeColor) person)
```

- Warum? Diese Schreibweise führt zu der «Inkonsistenz» / Doppeldeutigkeit, dass man über mehrere Wege auf die Variablen zugreifen kann, etwa auf den Vornamen wie folgt:

- `person.firstname()` – mit der benannten Variable und der Accessor-Methode
- `firstname` – auf Basis der Dekonstruktion

- Für mehr Stringenz ist dies mit Java 20 nicht mehr erlaubt und führt zu einem Kompilierfehler. Es wird nur noch folgende syntaktisch klarere Variante unterstützt:

```
if (obj instanceof Person(var firstname, var lastname,  
                         var dateOfBirth, var eyeColor))
```



---

# JEP 441: 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 Pattern 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 when str.length() > 5 -> System.out.println(str.strip());  
        case Integer i -> System.out.println(i);  
        default -> {  
    }  
}
```

A screenshot of an IDE showing the Java code above. A tooltip is displayed over the third case label 'case String str when str.length() > 5 ->'. The tooltip contains the text 'This case label is dominated by one of the preceding case labels' with a red 'X' icon, and 'Press 'F2' for focus' at the bottom right.

- 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!



- 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 box appears over the 'default' branch of the switch statement, containing the message 'Switch label '"Sophie"' is unreachable' followed by a colon and three dots. Below the message are two buttons: 'Remove switch branch'"Sophie"' and 'More actions...'. A screenshot of the Eclipse IDE interface shows the code in a editor window.

- 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 dem positiven Wertebereich und
- danach dem verbliebenen Rest:

```
Integer value = calcValue();

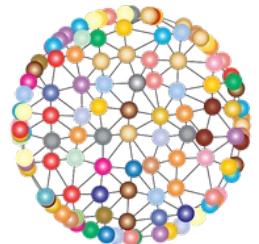
switch (value) {
    case -1, 0, 1 -> System.out.println("Handle special cases -1, 0 or 1");
    case Integer i && i >= 1 ->
        System.out.println("Handle positive integer cases i >= 1");
    case Integer i -> System.out.println("Handle all the remaining integers");
}
```

# 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



- **Vollständigkeitsanalyse = Prüfung, ob alle möglichen Pfade von den cases im switch abgedeckt werden**
- In früheren Java 17 Versionen kam es fälschlicherweise zu der Fehlermeldung «switch statement does not cover all possible input values»

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



- Mit Java 17.0.6 ist das nicht mehr der Fall und der Fix wurde backgeportet.

# JEP 441: Pattern Matching bei switch: Vollständigkeitsanalyse



- 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");
    }
}

SwitchWhen.java
```



- Mit Java 19 musste man beim Pattern Matching noch die Typen in <> angeben:

```
record MyPair<T1, T2>(T1 first, T2 second) { }

static void recordInferenceJdk19(MyPair<String, Integer> pair)
{
    switch (pair) {
        case MyPair<String, Integer>(var text, var count)
            when text.contains("Michael") ->
                System.out.println(text + " is " + count + " years old");
        case MyPair<String, Integer>(var text, var count)
            when count > 5 && count < 10 ->
                System.out.println("repeated " + text.repeat(count));
        case MyPair<String, Integer>(var text, var count) ->
                System.out.println(text + count);
        default -> System.out.println("NOT HANDLED");
    }
}
```

# JEP 441: Pattern Matching bei switch: Typinferenz mit Patterns



- Analog zu der Änderung bei instanceof kann man mit Java 20 auch in switch auf die konkreten Typangaben für generische Record Patterns verzichten (nur ab JDK 20.0.1\*)

```
static void recordInferenceJdk20(MyPair<String, Integer> pair)
{
    switch (pair)
    {
        // var geht hier nicht, wenn man auf die typspezifischen
        // Methode zugreifen möchte,
        case MyPair(String text, var count) when text.contains("Michael") ->
            System.out.println(text + " is " + count + " years old");
        case MyPair(String text, Integer 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");
    }
}
```

# JEP 441: Pattern Matching bei switch: Typinferenz mit Patterns



- Mit Java 21 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

---



---

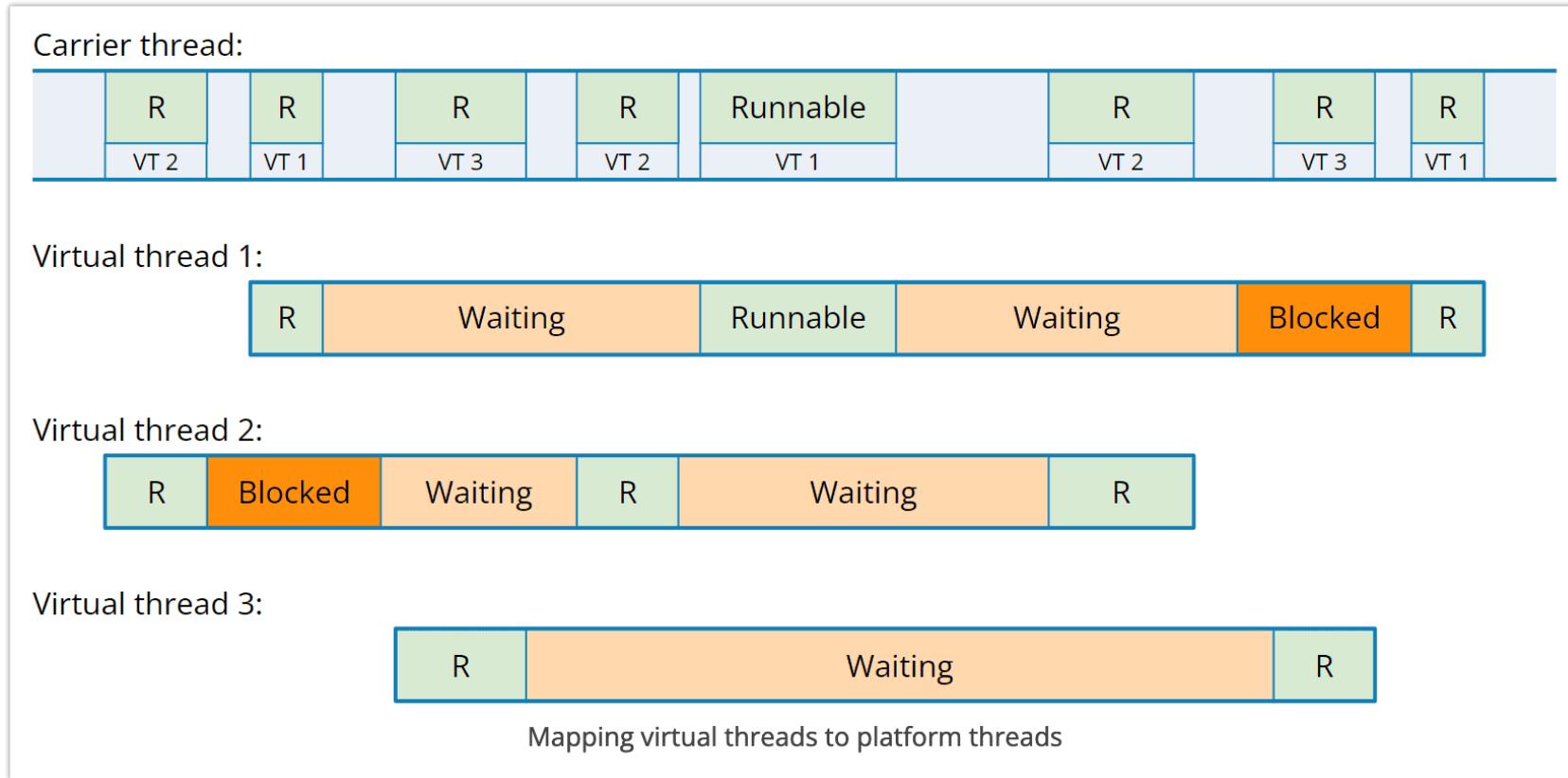
# JEP 444: Virtual Threads

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





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





- Dieses Preview-Feature führt das Konzept **leichtgewichtiger virtueller Threads ein, die nicht direkt auf Threads des Betriebssystems abgebildet werden.**
- Besser noch: Bereits vorhandener Code, der das bisherige Thread-API verwendet, lässt sich mit minimalen Änderungen auf **virtuelle Threads umstellen.**
- Mit **virtuellen Threads** kann man im Bereich der Serverprogrammierung wieder mit jeweils einem separaten Thread pro Anfrage arbeiten und leichtgewichtiger einen **asynchronen Programmierstil unterstützen.**
- Über Factory-Methoden wie `newVirtualThreadPerTaskExecutor()` kann man wählen, ob **virtuelle Threads oder Plattform-Threads (z.B. mit Executors.newCachedThreadPool()) verwendet werden sollen.**

# JEP 444: Virtual Threads

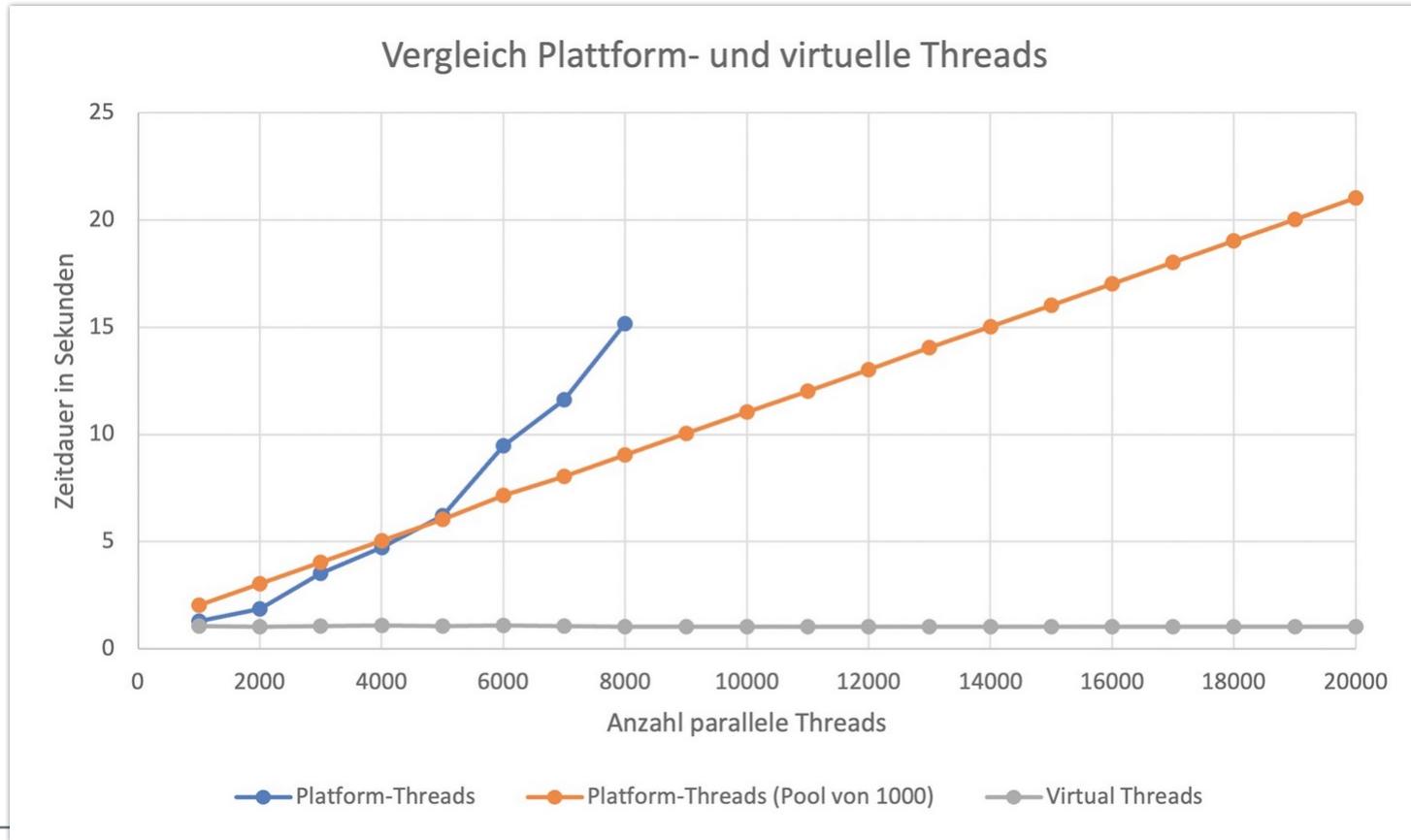


```
public static void main(String[] args)
{
    System.out.println("Start");

    try (var executor = Executors.newVirtualThreadPerTaskExecutor())
    {
        for (int i = 0; i < 10_000; i++)
        {
            final int pos = i;
            executor.submit(() -> {
                Thread.sleep(Duration.ofSeconds(1));
                return pos;
            });
        }
    }
    // executor.close() is called implicitly, and waits
    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.**





---

# DEMO & Hands on

PlatformThreads.java  
VirtualThreads.java

---



---

# Übungen PART 1

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





---

# 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)
  - JEP 448: Vector API (Sixth Incubator)
-



---

# 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}`

# String Templates



- **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

- **Beispiel 2**

```
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";
```

# 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 showing the generated HTML output. The browser interface includes a tab bar with three colored dots (red, yellow, green), a search bar with 'localhost', and navigation buttons. The main content area displays the heading 'My Hobbies:' followed by an ordered list: '• 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);
```

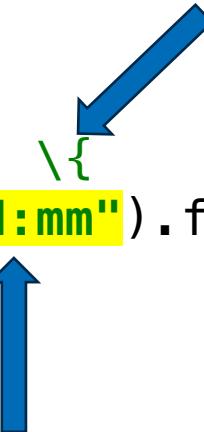
- => 10 + 20 = 30

```
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
```

## 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 = 52;
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 52 years old.

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

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

# String Templates – Eigene String Processors





---

# 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 p, Color c) {}
```



- Was beobachten Sie bei der Verwendung von Record Patterns?

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

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

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

- Nur ein paar Teile 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



- Dieses JEP befasst sich damit, dass verschiedene Elemente in einem Ausdruck oder einer Variablen durch ein einzelnes `_` ersetzt werden können. Ziel ist es, eine Musterkomponente 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.

# 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);
```



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

- =>

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

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



- **Unnamed pattern variable III**

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

- =>

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

- **Das Gleiche gilt für case.**

# JEP 443: Unnamed Patterns and Variables (Preview)



- **Unnamed pattern**

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

- =>

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

- **Das Gleiche gilt für case.**

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





---

# 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!");
    }
}
```

- 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 und ablenkende Wörter und Konzepte, die von der eigentlichen Aufgabe ablenken.

# JEP 445: Unnamed classes and instance main() method

---



- The goal is to develop Java in the direction of simpler initial learning.
- 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.



## 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 I: Instance main()

- Wie bisher können diese Klassen wie ein normales Java-Programm kompiliert und gestartet werden, wobei die direkte Komplilierung 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 statische 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!");  
}
```



## 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 kann der Aufwand auf ein Minimum reduziert werden, 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.



- 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)");  
    }  
}
```



---

# JEP 453: Structured Concurrency (Preview)

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





- 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.



- **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.



- **Weil die Teilaufgaben parallel ausgeführt werden, können sie unabhängig voneinander erfolgreich sein oder fehlschlagen. 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.**



- 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 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.



## Vorteile beim Einsatz der Klasse StructuredTaskScope:

- **Task und Subtasks bilden eine in sich geschlossene Einheiten**
- **Es werden kein ExecutorService und Threads aus einem Thread-Pool genutzt. Jede Teilaufgabe wird in einem neuen virtuellen Thread ausgeführt.**
- **Fehler in einer der Teilaufgaben => alle anderen Teilaufgaben werden abgebrochen.**
- **Wird der aufrufende Thread abgebrochen, werden auch die Teilaufgaben abgebrochen.**
- **Aufrufhierarchie (aufrufenden Thread und Teilaufgaben) im Thread-Dump gut zu erkennen.**
- Beim ExecutorService sieht man im Thread-Dump nur etwa die Thread-Namen "pool-X-thread-Y". Die Zuordnung von Pool-Thread zu aufrufenden Thread für Teilaufgaben ist kaum möglich.



---

# DEMO & Hands on

StructuredConcurrency.java

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



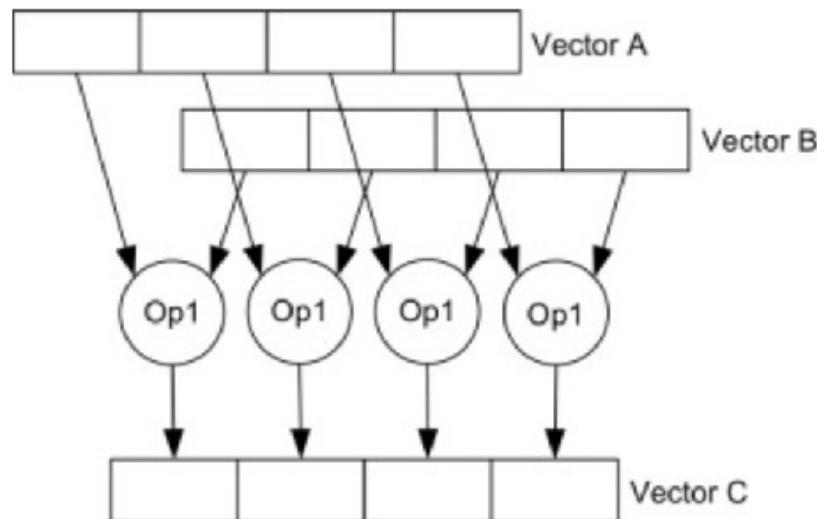
---

# Incubator Features in Java 21



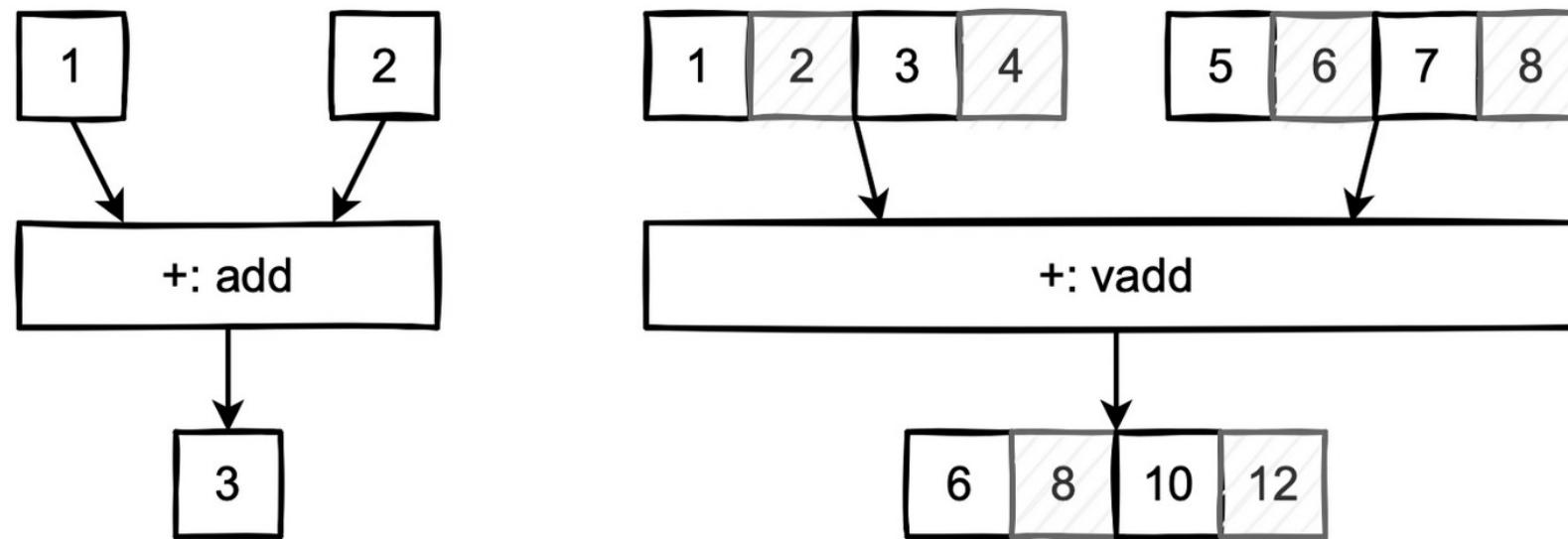
# JEP 448: Vector API (Sixth Incubator in Java 21)

<https://openjdk.org/jeps/448>





- Dieser JEP hat nichts mit der Klasse `java.util.Vector` zu tun! Vielmehr geht es um eine plattformunabhängige Unterstützung von sogenannten Vektorberechnungen.
- Moderne Prozessoren können beispielsweise eine Addition oder Multiplikation nicht nur für zwei Werte, sondern für eine Vielzahl von Werten ausführen. Man spricht dabei auch von **Single Instruction Multiple Data (SIMD)**. Die folgende Grafik visualisiert das Grundprinzip:





- Das Vector API zielt darauf ab, die Leistung von Vektorberechnungen zu verbessern.
- Eine Vektorberechnung besteht aus einer Folge von Operationen mit Vektoren. Dabei kann man sich einen Vektor wie ein Array von primitiven Werten vorstellen.
- Wollte man nun zwei Vektoren respektive Arrays mit einer mathematischen Operation verknüpfen, so würde man dazu herkömmlich eine Schleife über alle Werte nutzen.

```
int[] a = { 1, 2, 3, 4, 5, 6, 7, 8 };
int[] b = { 1, 2, 3, 4, 5, 6, 7, 8 };

var c = new int[a.length];
for (int i = 0; i < a.length; i++)
{
    c[i] = a[i] + b[i];
}
```



- Mit dem **Vector API** lassen sich die Besonderheiten und Optimierungen in modernen Prozessoren ausnutzen. Dazu gibt man gewisse Hilfestellungen für die Berechnungen vor.

```
int[] a = { 1, 2, 3, 4, 5, 6, 7, 8 };
int[] b = { 1, 2, 3, 4, 5, 6, 7, 8 };
```

```
var c = new int[a.length];
var vectorA = IntVector.fromArray(IntVector.SPECIES_256, a, 0);
var vectorB = IntVector.fromArray(IntVector.SPECIES_256, b, 0);
var vectorC = vectorA.add(vectorB);
// var vectorC = vectorA.mul(vectorB);
vectorC.intoArray(c, 0);
```

- Das steuernde Element ist hier die Größe des Vektors, den wir durch das Argument `IntVector.SPECIES_256` auf 256 Bit festlegen.
- Danach kann dann die passende Aktion erfolgen, hier `add()` oder `mul()`.



- Bei nicht perfekt passenden und größeren Ausgangsdaten müssen die Aktionen passend aufgeteilt werden.
- Betrachten wir das Beispiel aus JEP 417 und die Skalarberechnung als Ausgangsbasis:

```
void scalarComputation(float[] a, float[] b, float[] c)
{
    for (int i = 0; i < a.length; i++)
    {
        c[i] = (a[i] * a[i] + b[i] * b[i]) * -1.0f;
    }
}
```

- Algorithmus ist absolut verständlich und leicht nachvollziehbar

# JEP 448: Vector API



```
static final VectorSpecies<Float> SPECIES = FloatVector.SPECIES_PREFERRED;

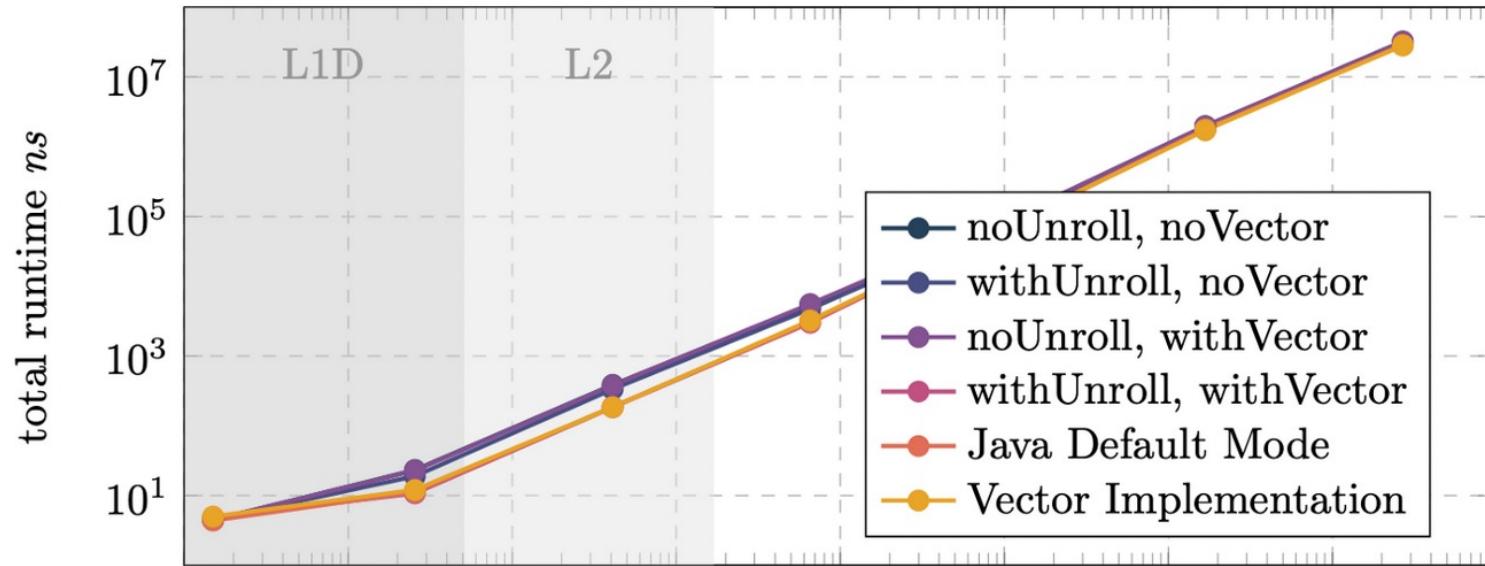
void vectorComputation(float[] a, float[] b, float[] c)
{
    int i = 0;
    int upperBound = SPECIES.loopBound(a.length);
    for (; i < upperBound; i += SPECIES.length())
    {
        var va = FloatVector.fromArray(SPECIES, a, i);
        var vb = FloatVector.fromArray(SPECIES, b, i);
        var vc = va.mul(va)
                  .add(vb.mul(vb))
                  .neg();
        vc.intoArray(c, i);
    }

    for (; i < a.length; i++)
    {
        c[i] = (a[i] * a[i] + b[i] * b[i]) * -1.0f;
    }
}
```

# JEP 448: Vector API Benchmarking



Benchmark:  $c[n] = a[n] + b[n]$



Method	Peak Speed-Up
No Unroll, No Vector	1.00x
With Unroll, No Vector	1.22x
No Unroll, With Vector	1.01x
With Unroll, With Vector	2.15x
Java Default	2.06x
Vector Implementation	2.08x



---

# Übungen PART 2

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





# Fazit



## On the positive side

---



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

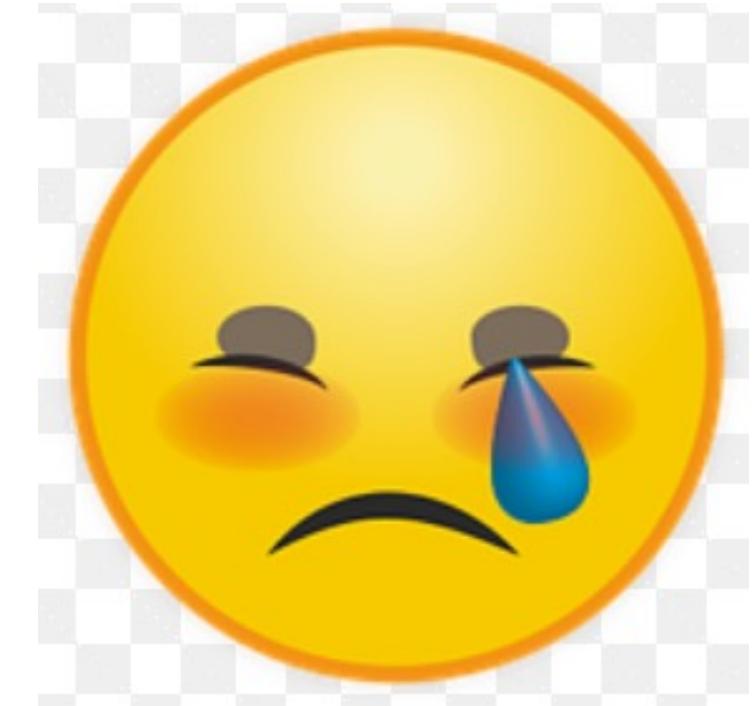


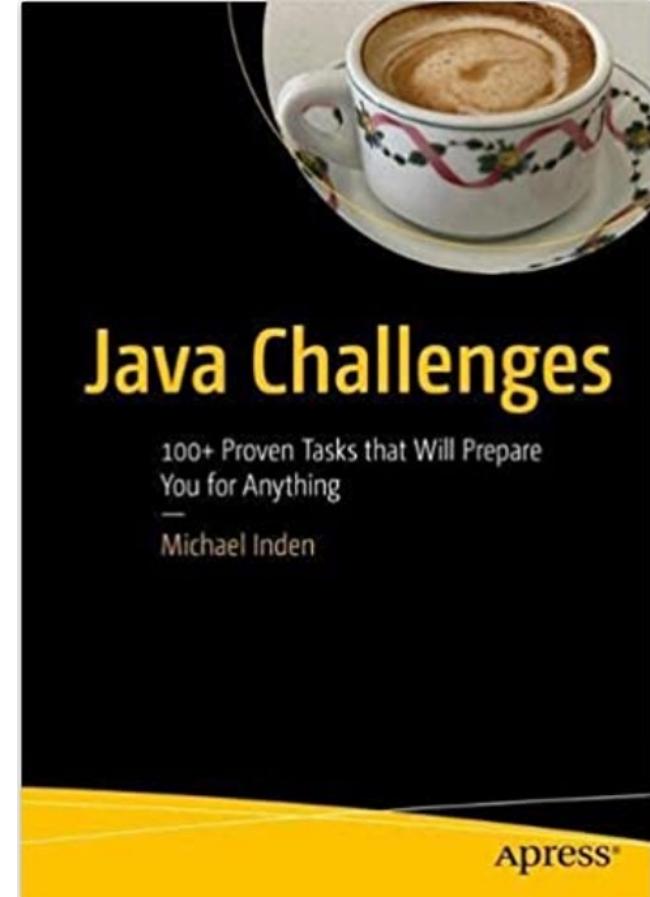
## Negatives

---



- Releases waren zwar pünktlich, aber manchmal (außer Java 14) ziemlich dünn bezüglich wichtigen 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 noch 2 Jahre warten, bis die netten Unnamed Classes und Variables für den Gebrauch verfügbar sind
- Warum ist die Syntax von Pattern Matching bei instanceof und switch inkonstant?







# Questions?



# Thank You