



Best of Java 11 bis 21

<https://github.com/Michaeli71/Best-Of-Java-11-21>



Michael Inden

Head of Development, freiberuflicher Buchautor und Trainer

Speaker Intro



E-Mail: 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/Best-Of-Java-11-21>

Co-Speaker Intro



- **Michael Kulla, Jahrgang 1972**
- **Sun Certified Java Trainer**
- **Oracle University Delivery Instructor Java**
- Autor diverser Videotrainings u. a. zu Java, Java EE und Entwurfsmustern
- lange Zeit **freiberuflicher Softwareentwickler** und Berater
- Seit 2017 festangestellt bei **GEDOPLAN GmbH** in Bielefeld als **Trainer, Berater und Softwareentwickler Java und Jakarta EE**
- **Proofreader** u.a. der Bücher des Herrn Inden ☺

E-Mail: michael.kulla@gedoplan.de





Agenda



- **Vorbemerkungen / Build Tools & IDEs**
- **PART 1:** Syntax-Erweiterungen & Neuheiten und Änderungen bis Java 11
- **PART 2:** Weitere Neuheiten und Änderungen bis JDK 11
- **PART 3:** Syntax-Erweiterungen & Neuheiten und Änderungen in Java 12 bis 17
- **PART 4:** Neuheiten und Änderungen in den APIs in Java 12 bis 17
- **PART 5:** Neuheiten und Änderungen in der JVM in Java 12 bis 17
- **PART 6:** Neuheiten in Java 18 bis 21



Separat: Modularisierung im Kurzüberblick

Workshop Contents



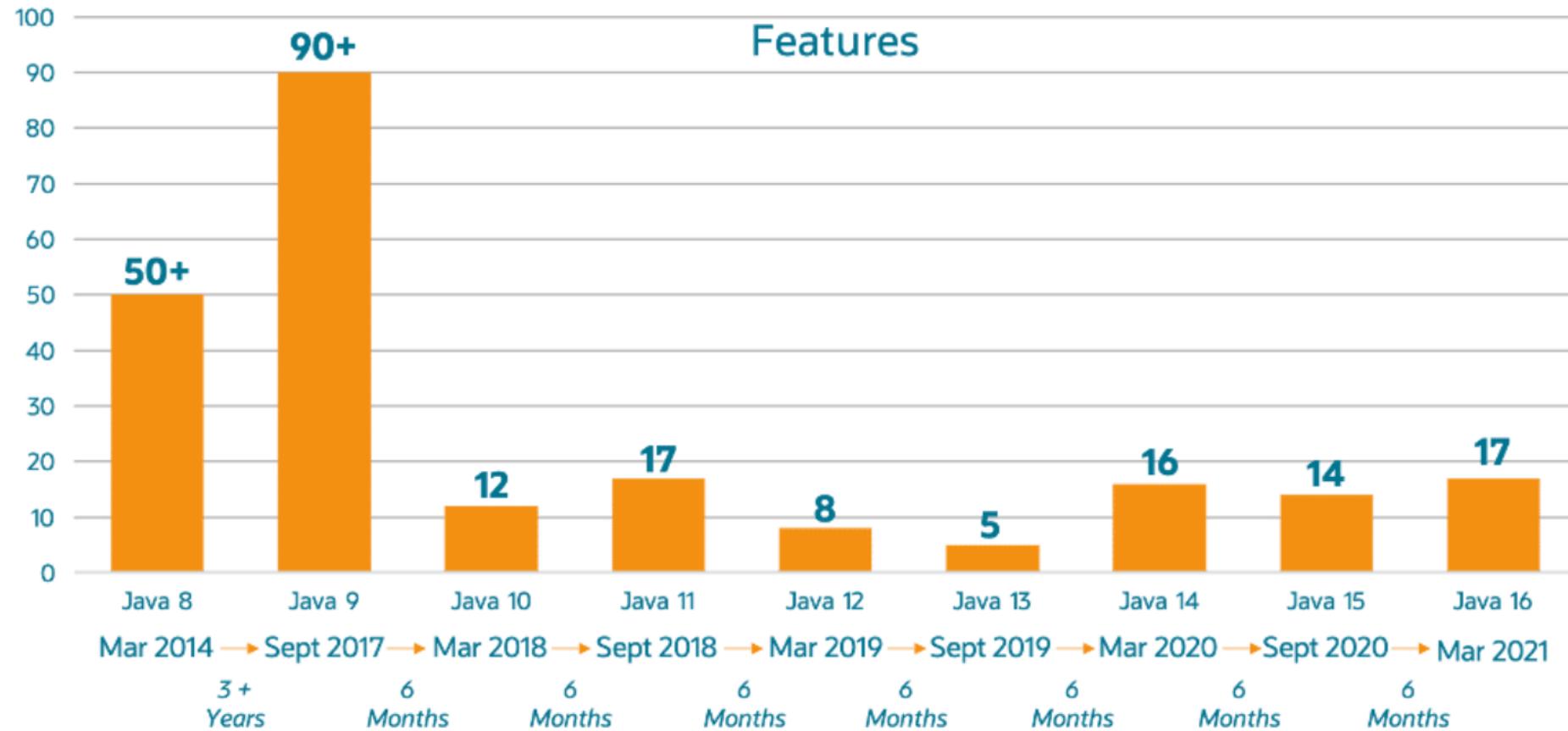
JDK	Release-Datum	Entwicklungszeit	LTS	Workshop
Oracle JDK 9	9 / 2017	3,5 Jahre	-	Part 1, 2
Oracle JDK 10	3 / 2018	6 Monate	-	Part 1, 2
Oracle JDK 11	9 / 2018	6 Monate	Ja, <i>kommerziell</i>	Part 1, 2
Oracle JDK 12	3 / 2019	6 Monate	-	Part 3, 4, 5
Oracle JDK 13	9 / 2019	6 Monate	-	Part 3, 4, 5
Oracle JDK 14	3 / 2020	6 Monate	-	Part 3, 4, 5
Oracle JDK 15	9 / 2020	6 Monate	-	Part 3, 4, 5
Oracle JDK 16	3 / 2021	6 Monate	-	Part 3, 4, 5
Oracle JDK 17	9 / 2021	6 Monate	Ja, „frei“, aber Lizenz	Part 3, 4, 5
Oracle JDK 18	3 / 2022	6 Monate	-	Part 6
Oracle JDK 19	9 / 2022	6 Monate	-	Part 6
Oracle JDK 20	3 / 2023	6 Monate	-	Part 6
Oracle JDK 21	9 / 2023	6 Monate	Ja, „frei“, aber Lizenz	Part 6

Long-Term Support-Modell



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

Einordnung des 6-monatigen Releasezyklus‘



Für Java 17 und weitere ähnlich



Build-Tools und IDEs



Aktuelles Java 21 installiert



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

```
$ java -version
java version "21.0.1" 2023-10-17 LTS
Java(TM) SE Runtime Environment (build 21.0.1+12-LTS-29)
Java HotSpot(TM) 64-Bit Server VM (build 21.0.1+12-LTS-29, 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

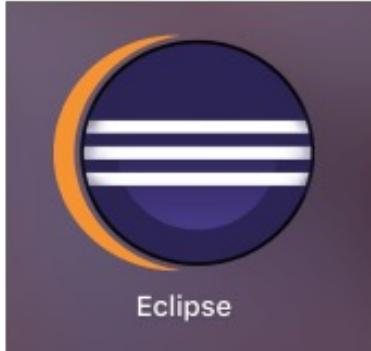
import javax.lang.model.SourceVersion;

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 2023.2.4
- Maven: 3.9.5, Compiler Plugin: 3.11.0
- Gradle: 8.4
- Aktivierung von Preview-Features / Incubator nötig
 - In Dialogen
 - In Build Scripts



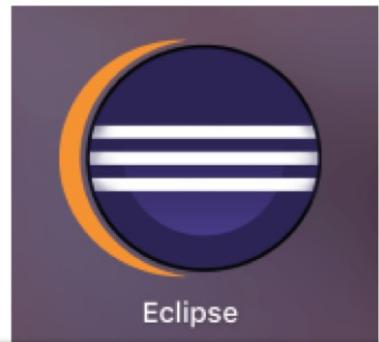
Maven™

 **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



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

- Name: Java21Examples (with a red arrow pointing to it)
- SDK: 21 java version "21" (with a red arrow pointing to the dropdown button)
- Language level: 21 (Preview) - String templates, unnamed classes and instance main methods etc. (with a red arrow pointing to the dropdown button)
- Compiler output: (with a folder icon and a red arrow pointing to it)

A note at the bottom states: "Used for module subdirectories, Production and Test directories for the corresponding sources."



- 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

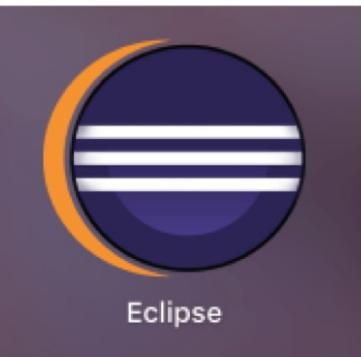
Use @argfile when launching

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



Run/Debug Configurations

Name: VectorApiExample Store as project file

Build and run

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

api.VectorApiExample

Program arguments

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

Working directory: /Users/michaelindl/Desktop/Vorträge/SPECIAL/JUGS-Java21/BestOfModernJava21Exam

Environment variables:

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

Open run/debug tool window when started

Code Coverage

Packages and classes to include in coverage data

- + api.*

Edit configuration templates...

?

Run Apply Cancel OK





PART 1: Syntax-Erweiterungen und API- Änderungen bis Java 11



Syntax-Erweiterungen



@Deprecated-Annotation



- **@Deprecated** dient zum Markieren von obsoletem Sourcecode
- **JDK 8:** keine Parameter
- **JDK 9:** Zwei Parameter **@since** und **@forRemoval**

```
/**  
 * @deprecated this method is replaced by someNewMethod() which is  
more stable  
*/  
@Deprecated(since = "7.2", forRemoval = true)  
private static void someOldMethod()  
{  
    // ...  
}
```

Local Variable Type Inference => var



- Local Variable Type Inference
- Neues reserviertes Wort var zur Definition lokaler Variablen, statt expliziter Typangabe

```
var name = "Peter";           // var => String
var chars = name.toCharArray(); // var => char[]

var mike = new Person("Mike", 47); // var => Person
var hash = mike.hashCode();      // var => int
```

- möglich, sofern der konkrete Typ für eine lokale Variable anhand der Definition auf der rechten Seite der Zuweisung vom Compiler ermittelt werden kann

 **var justDeclaration; // keine Wertangabe / Definition**
var numbers = {0, 1, 2}; // fehlende Typangabe

Local Variable Type Inference => var



- Besonders im Kontext von Generics zur Schreibweisen-Abkürzung nützlich:

```
// var => ArrayList<String> names
```

```
ArrayList<String> namesOldStyle = new ArrayList<String>();  
var names = new ArrayList<String>();  
names.add("Tim"); names.add("Tom");
```

```
// var => Map<String, Long>
```

```
Map<String, Long> personAgeMappingOldStyle = Map.of("Tim", 47L, "Tom", 12L,  
                                                 "Michael", 47L);  
var personAgeMapping = Map.of("Tim", 47L, "Tom", 12L, "Michael", 47L);
```

Local Variable Type Inference => var



- Insbesondere wenn die Typangaben mehrere generische Parameter umfassen, kann `var` den Sourcecode deutlich kürzer und mitunter lesbarer machen

```
// var => Set<Map.Entry<String, Long>>
var entries = personAgeMapping.entrySet();
```

```
// var => Map<Character, Set<Map.Entry<String, Long>>>
var filteredPersons = personAgeMapping.entrySet().stream().
    collect(groupingBy(firstChar,
                      filtering(isAdult, toSet())));
```

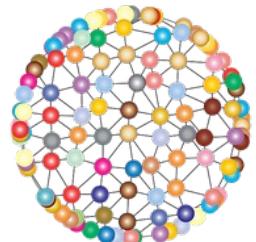
- Dazu nutzen wir diese Lambdas:

```
final Function<Map.Entry<String, Long>, Character> firstChar =
    entry -> entry.getKey().charAt(0);
```

```
final Predicate<Map.Entry<String, Long>> isAdult =
    entry -> entry.getValue() >= 18;
```



**Wäre es nicht schön,
auch hier var zu nutzen?**





- Ja!!!

- Aber der Compiler kann rein basierend auf diesen Lambdas den konkreten Typ nicht ermitteln
- Somit ist keine Umwandlung in var möglich, sondern führt zur Fehlermeldung «Lambda expression needs an explicit target-type».
- Wollte man diesen Fehler vermeiden, so müsste man folgenden Cast einfügen:

```
var isAdultVar = (Predicate<Map.Entry<String, Long>>)
    entry -> entry.getValue() >= 18;
```

- Insgesamt sieht man, dass var für Lambda-Ausdrücke eher ungeeignet ist.

Local Variable Type Inference Fallstrick

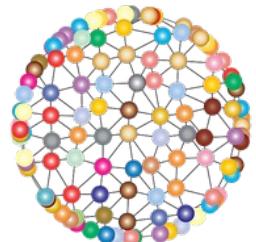


- Manchmal ist man versucht, ohne viel darüber nachzudenken, die Typangabe auf der linken Seite direkt mit var zu ersetzen:

```
final List<String> names = new ArrayList<>();
names.add("Expected");
// names.add(42); // Compile error
```

- **Ersetzen wir den Typ durch var und kommentieren die untere Zeile ein:**

```
final var mixedContent = new ArrayList<>();
mixedContent.add("Strange with var");
mixedContent.add(42);
```



**Kompliert das? Und
wenn ja, wieso?**

Local Variable Type Inference Fallstrick



- Tatsächlich produziert das Ganze keinen Kompilierfehler. Wie kommt das?
- Aufgrund des Diamond Operators, bzw. der nicht vorhandenen Typangabe, stehen dem Compiler nicht ausreichend Typinformationen zur Verfügung:

```
// var => ArrayList<Object>
final var mixedContent = new ArrayList<>();
mixedContent.add("Strange with var");
mixedContent.add(42);
```

- Beim Einsatz von var wird immer der **exakte** Typ verwendet wird und nicht ein Basistyp, wie man es getreu dem Paradigma «Program against interfaces» sehr gerne macht:

```
// var => ArrayList<String> und nicht List<String>
var names = new ArrayList<String>();
names = new LinkedList<String>(); // error
```



Neuheiten und Änderungen bis Java 11

- Optional<T>
 - Collection Factory Methods
 - Strings
-



Optional<T>



Die Klasse Optional<T>



- Mit Java 8 eingeführt und erleichtert die Behandlung und Modellierung optionaler Werte.
- Bereits gutes API
- Aber **3 Schwachstellen** bei folgenden Aufgabenstellungen:
 - Das Ausführen von Aktionen auch im Negativfall.
 - Die Verknüpfung der Resultate mehrerer Berechnungen, die Optional<T> liefern.
 - Die Umwandlung in einen Stream<T>, für eine Kompatibilität mit dem Stream-API z. B. für Frameworks, die auf Streams arbeiten,

Die Klasse Optional<T>



- Durch die Erweiterungen der Klasse Optional<T> in JDK 9 wurden alle der drei zuvor aufgelisteten Schwachstellen adressiert. Dazu dienen folgende Methoden:
 - `ifPresentOrElse(Consumer<? super T>, Runnable)` – Erlaubt die Ausführung einer Aktion im Positiv- oder im Negativfall.
 - `or(Supplier<Optional<T>> supplier)` – Ermöglicht auf elegante Weise die Verknüpfung mehrerer Berechnungen.
 - `stream()` – Wandelt das Optional<T> in einen Stream<T> um.

Motivation ifPresentOrElse(...)



JDK 8

```
public class Example1 {  
  
    public static void main(String[] args) {  
  
        Optional<String> welcomeString = getWelcomeString();  
  
        ➔ welcomeString.ifPresent(System.out::println);  
  
    }  
  
    private static Optional<String> getWelcomeString() {  
        return Optional.of("Hi JUG Zürich");  
    }  
  
}
```

Motivation ifPresentOrElse(...)



JDK 8

```
public class Example1 {  
  
    public static void main(String[] args) {  
  
        Optional<String> welcomeString = getWelcomeString();  
  
        ➔ if(welcomeString.isPresent()) {  
            System.out.println(welcomeString.get());  
        }else {  
            System.out.println("Hi JUG Default ");  
        }  
  
    }  
  
    private static Optional<String> getWelcomeString() {  
        return Optional.of("Hi JUG Zürich");  
    }  
}
```

ifPresentOrElse(Consumer<? super T>, Runnable)



Mit Java 9 und der Methode `ifPresentOrElse()` lässt sich die Ergebnisauswertung von Suchen/ Aktionen oftmals vereinfachen:

JDK 9

```
public class Example1 {  
  
    public static void main(String[] args) {  
  
        Optional<String> welcomeString = getWelcomeString();  
  
        → welcomeString.ifPresentOrElse(System.out::println, () -> System.out.println("Hi JUG XXX"));  
  
    }  
  
    private static Optional<String> getWelcomeString() {  
        return Optional.of("Hi JUG Zürich");  
    }  
  
}
```

Optional<T> in JDK 9



▼ C F Optional<T>

- S empty() <T> : Optional<T>
- S of(T) <T> : Optional<T>
- S ofNullable(T) <T> : Optional<T>
- ▲ equals(Object) : boolean
- filter(Predicate<? super T>) : Optional<T>
- flatMap(Function<? super T, ? extends Optional<? extends U>>) <U> : Optional<U>
- get() : T
- ▲ hashCode() : int
- ifPresent(Consumer<? super T>) : void
- ifPresentOrElse(Consumer<? super T>, Runnable) : void
- isPresent() : boolean
- map(Function<? super T, ? extends U>) <U> : Optional<U>
- or(Supplier<? extends Optional<? extends T>>) : Optional<T>
- orElse(T) : T
- orElseGet(Supplier<? extends T>) : T
- orElseThrow(Supplier<? extends X>) <X extends Throwable> : T
- stream() : Stream<T>
- ▲ toString() : String



Motivation or(...)



```
public class PaymentProcessor {  
  
    public static void main(String[] args) {  
        Double balance = 0.0;  
        Optional<Double> debitCardBalance = getDebitCardBalance();  
        Optional<Double> creditCardBalance = getCreditCardBalance();  
        Optional<Double> paypalBalance = getPayPalBalance();  
  
        → if (debitCardBalance.isPresent()) {  
  
            balance = debitCardBalance.get();  
  
        } else if (creditCardBalance.isPresent()) {  
  
            balance = creditCardBalance.get();  
  
        } else if (paypalBalance.isPresent()) {  
  
            balance = paypalBalance.get();  
  
        }  
  
        if(balance > 0)  
            System.out.println("Payment will be processed...");  
        else  
            System.out.println("Sorry insufficient balance");  
    }  
}
```

JDK 8

Starke Vereinfachung ...

wenn zwei present sind, das erste mit balance ≤ 0 , aber das zweite > 0 , kommt trotzdem insufficient balance...

or(...)



-

JDK 9

```
Optional<Double> optBalance = getDebitCardBalance().  
                           or(()->getCreditCardBalance()).  
                           or(()->getPayPalBalance());
```



```
optBalance.ifPresentOrElse(value -> System.out.println("Payment " + value +  
                                         " will be processed ..."),  
                           () -> System.out.println("Sorry, insufficient balance"));
```

- **or(Supplier<Optional<T>> supplier)** wirkt unscheinbar
- Aber es lassen sich **Aufrufketten** mit **Fallback-Strategien** auf **lesbare** und **verständliche** Art beschreiben, wie es das obige Beispiel **eindrucksvoll** zeigt



Collection Factory Methods



Collection Factory Methods Intro



- Das Erzeugen von Collections für eine (kleinere) Menge vordefinierter Werte ist in Java mitunter etwas umständlich:

```
// Variante 1
final List<String> oldStyleList1 = new ArrayList<>();
oldStyleList1.add("item1");
oldStyleList1.add("item2");

// Variante 2
final List<String> oldStyleList2 = Arrays.asList("item1", "item2");
```

- Sprachen wie Groovy oder Python bieten dafür eine spezielle Syntax, sogenannte Collection-Literale ...

Collection Factory Methods Intro



```
List<String> names = ["MAX", "Moritz", "Tim"];  
Map<String, Integer> nameToAge = { "Tim" : 45, "Tom" : 23 };
```

```
newStyleList[0] = "newValue";  
final String valueAtPos0 = newStyleList[0]; // => newValue
```

Bereits 2009 hat man auch für Java über Derartiges nachgedacht.
Leider wurde dies nicht realisiert ...

Collection Factory Methods



Collection Literals **LIGHT** a.k.a Collection Factory Methods

Collection Factory Methods



- Verhalten recht intuitiv für Listen ...

```
List<String> names = List.of("MAX", "MORITZ", "MIKE");  
names.forEach(name -> System.out.println(name));
```

```
MAX  
MORITZ  
MIKE
```

Collection Factory Methods



- **Verhalten recht merkwürdig für Sets ...**

```
Set<String> names2 = Set.of("MAX", "MORITZ", "MAX");
names2.forEach(name -> System.out.println(name));
```

```
Exception in thread "main" java.lang.IllegalArgumentException:
duplicate element: MAX
    at java.util.ImmutableCollections$SetN.<init>(java.base@9-ea/
ImmutableCollections.java:329)
    at java.util.Set.of(java.base@9-ea/Set.java:500)
    at jdk9example.Jdk9Example.main(Jdk9Example.java:31)
```



Erweiterung in der Klasse String



Erweiterung in `java.lang.String`



- Die Klasse `String` existiert seit JDK 1.0 und hat zwischenzeitlich nur wenige API-Änderungen erfahren.
- Mit Java 11 ändert sich das. Es wurden folgende Methoden neu eingeführt:
 - `isBlank()`
 - `lines()`
 - `repeat(int)`
 - `strip()`
 - `stripLeading()`
 - `stripTrailing()`

Erweiterung in `java.lang.String`: `isBlank()`



- Für Strings war es bisher nur mühsam oder mithilfe von externen Bibliotheken möglich zu prüfen, ob diese nur Whitespace enthalten.
- Dazu wurde mit Java 11 die Methode `isBlank()` eingeführt, die sich auf `Character.isWhitespace(int)` abstützt.

```
private static void isBlankExample()
{
    final String exampleText1 = "";
    final String exampleText2 = "      ";
    final String exampleText3 = " \n \t ";

    System.out.println(exampleText1.isBlank());
    System.out.println(exampleText2.isBlank());
    System.out.println(exampleText3.isBlank());
}
```

- Alle geben true aus.
-

Erweiterung in `java.lang.String`: `lines()`



- Beim Verarbeiten von Daten aus Dateien müssen des Öfteren Informationen in einzelne Zeilen aufgebrochen werden. Dazu gibt es etwa die Methode `Files.lines(Path)`.
- Ist die Datenquelle allerdings schon ein `String`, gab es diese Funktionalität bislang noch nicht. JDK 11 bietet die Methode `lines()`, die einen `Stream<String>` zurückliefert:

```
private static void linesExample()
{
    final String exampleText = "1 This is a\n2 multi line\r" +
                               "3 text with\r\n4 four lines!";
    final Stream<String> lines = exampleText.lines();
    lines.forEach(System.out::println);
}
```

=>

```
1 This is a
2 multi line
3 text with
4 four lines!
```

Erweiterung in `java.lang.String`: `repeat()`



- Immer mal wieder steht man vor der Aufgabe, einen String mehrmals aneinanderzureihen, also einen bestehenden String n-Mal zu wiederholen.
- Dazu waren bislang eigene Hilfsmethoden oder solche aus externen Bibliotheken nötig. Mit Java 11 kann man stattdessen die Methode `repeat(int)` nutzen:

```
private static void repeatExample()
{
    final String star = "*";
    System.out.println(star.repeat(30));

    final String delimiter = " -* ";
    System.out.println(delimiter.repeat(6));
}

=>

*****  
-*_- -*_- -*_- -*_- -*_- -*_-
```

Erweiterung in `java.lang.String`: `strip()`/`-Leading()`/`-Trailing()`



- Die Methoden `strip()`, `stripLeading()` und `stripTrailing()` dienen dazu, führende und nachfolgende Leerzeichen (Whitespaces) aus einem String zu entfernen:

```
private static void stripExample()
{
    final String exampleText1 = " abc ";
    final String exampleText2 = "\t XYZ \t ";

    System.out.println("'" + exampleText1.strip() + "'");
    System.out.println("'" + exampleText2.strip() + "'");
    System.out.println("'" + exampleText2.stripLeading() + "'");
    System.out.println("'" + exampleText2.stripTrailing() + "'");
}

=>

'abc'
'XYZ'
'XYZ
'
'XYZ'
```



Übungen PART 1

<https://github.com/Michaeli71/Best-Of-Java-11-21>





PART 2: Weitere Neuerungen und Änderungen in den APIs und der JVM

- Date and Time API
- Files
- Multi-Threading mit CompletableFuture
- HTTP/2
- Direct Compilation
- JShell



Date and Time API



Klasse LocalDate



- **datesUntil()** – erzeugt einen Stream<LocalDate> zwischen zwei LocalDate-Instanzen und erlaubt es, optional eine Schrittweite vorzugeben:

```
public static void main(final String[] args)
{
    final LocalDate myBirthday = LocalDate.of(1971, Month.FEBRUARY, 7);
    final LocalDate christmas = LocalDate.of(1971, Month.DECEMBER, 24);

    System.out.println("Day-Stream");
    final Stream<LocalDate> daysUntil = myBirthday.datesUntil(christmas);
    daysUntil.skip(150).limit(4).forEach(System.out::println);

    System.out.println("\n3-Month-Stream");
    final Stream<LocalDate> monthsUntil =
        myBirthday.datesUntil(christmas, Period.ofMonths(3));
    monthsUntil.limit(3).forEach(System.out::println);
}
```

Klasse LocalDate



- Start 7. Februar => Sprung um 150 Tage in die Zukunft => 7. Juli
- **Day-Stream:** Tageweise Iteration begrenzt auf 4
- **Month-Stream:** Monatsweise Iteration begrenzt auf 3
=> Vorgabe einer alternativen Schrittweite, hier Monate:

Day-Stream

1971-07-07

1971-07-08

1971-07-09

1971-07-10

3-Month-Stream

1971-02-07

1971-05-07

1971-08-07



Erweiterung in der Klasse Files



Utility-Klasse `java.nio.file.Files`



- In Java 11 wurde die Verarbeitung von Strings im Zusammenhang mit Dateien erleichtert.
- Es ist nun einfach möglich, Strings in eine Datei zu schreiben bzw. daraus zu lesen.
- Dazu bietet die Utility-Klasse `Files` die Methoden `writeString()` und `readString()`.

```
final Path destPath = Path.of("ExampleFile.txt");
```

```
Files.writeString(destPath, "1: This is a string to file test\n");
Files.writeString(destPath, "2: Second line");
```

```
final String line1 = Files.readString(destPath);
final String line2 = Files.readString(destPath);
```

```
System.out.println(line1);
System.out.println(line2);
```

=>

```
2: Second line
2: Second line
```

Intuitive Verwendung
... aber leider falsch ...
Woran liegt es?

Utility-Klasse `java.nio.file.Files`



- **Korrektur 1:** APPEND-Mode

```
Files.writeString(destPath, "2: Second line", StandardOpenOption.APPEND);
```

=>

```
1: This is a string to file test  
2: Second line  
1: This is a string to file test  
2: Second line
```

- **Korrektur 2:** String nur einmal lesen

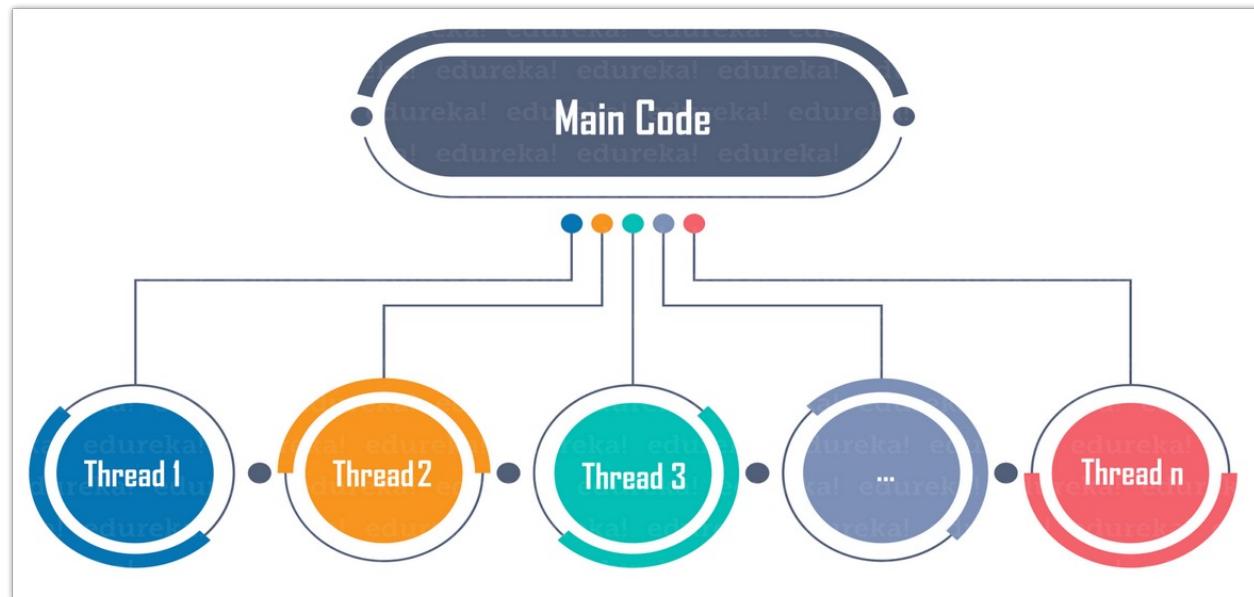
```
final String content = Files.readString(destPath);  
content.lines().forEach(System.out::println);
```

=>

```
1: This is a string to file test  
2: Second line
```



Multi-Threading mit CompletableFuture



Multi-Threading und die Klasse CompletableFuture<T>



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

Einstieg CompletableFuture<T>



- **Basisschritte**

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

- **Beispiel**

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

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

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

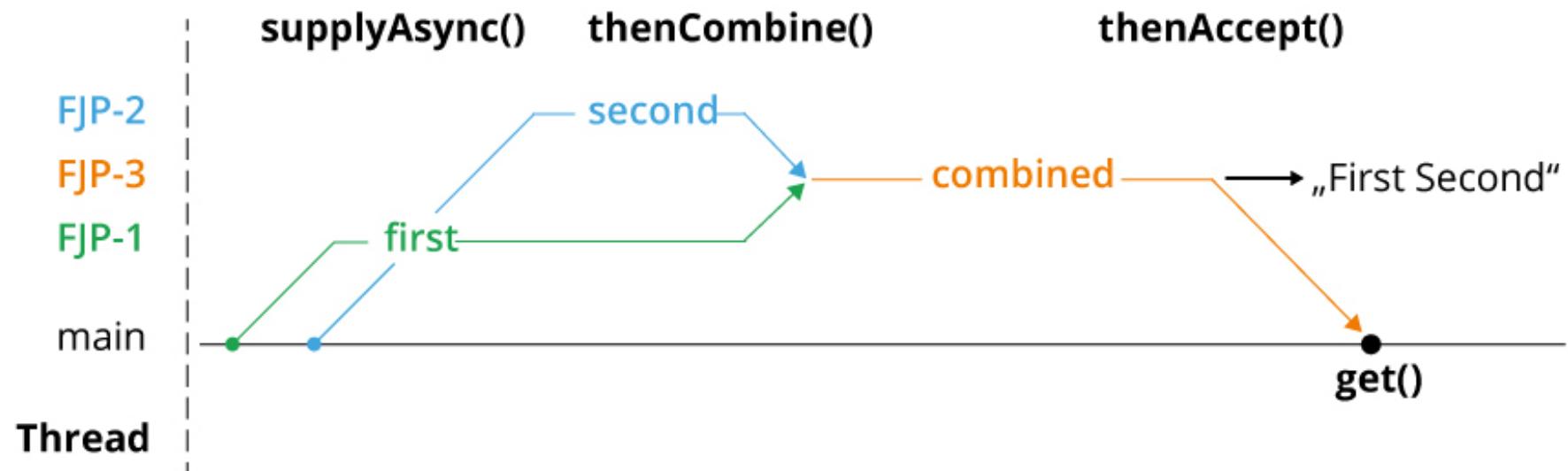
Einführung CompletableFuture<T>



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

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

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



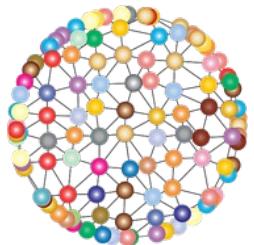


Beispiel: Es sollen folgende Aktionen stattfinden:

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



Wie könnte eine erste
Realisierung aussehen?



Multi-Threading und die Klasse CompletableFuture<T>



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

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

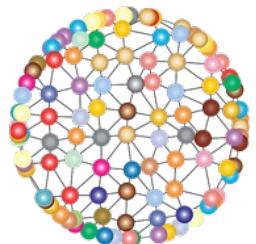
Multi-Threading und die Klasse CompletableFuture<T>



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

Die Berechnungen des Beispiels sind allerdings sehr vereinfacht ...

- **keine Parallelität**
 - **keine Abstimmung der Aufgaben (funktioniert, weil synchron)**
 - **kein Exception-Handling**
-
- **Wir ersparen uns die Mühen und kaum verständliche und unerwartbare Varianten mit Runnable, Callable<T>, Thread-Pools, ExecutorService usw., weil das selbst damit oft doch noch kompliziert und fehlerträchtig ist**



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

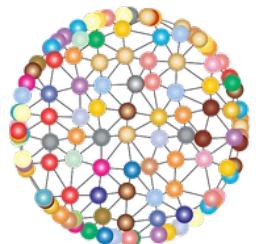
Multi-Threading und die Klasse CompletableFuture<T>



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

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

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



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

Multi-Threading und die Klasse CompletableFuture<T>



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

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

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

Multi-Threading und die Klasse CompletableFuture<T>



- Die Klasse CompletableFuture<T> bietet die Methode **exceptionally()**

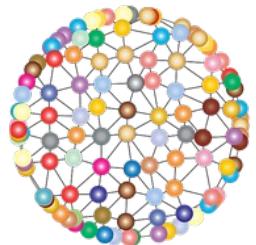
- Fallback-Wert bereitstellen, wenn die Daten nicht ermittelt werden können:

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

- Fallback-Wert bereitstellen, wenn eine Berechnung fehlschlägt:

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

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



**Wie bilden Verzögerungen
der realen Welt ab?**

Multi-Threading und die Klasse CompletableFuture<T>



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

Multi-Threading und die Klasse CompletableFuture<T>



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

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

Multi-Threading und die Klasse CompletableFuture<T>



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

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

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

Multi-Threading und die Klasse CompletableFuture<T>



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

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

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



HTTP/2 API



HTTP/2 API Intro



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

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

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

printResponseInfo(response);

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

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

HTTP/2 API Intro (var shines here)



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

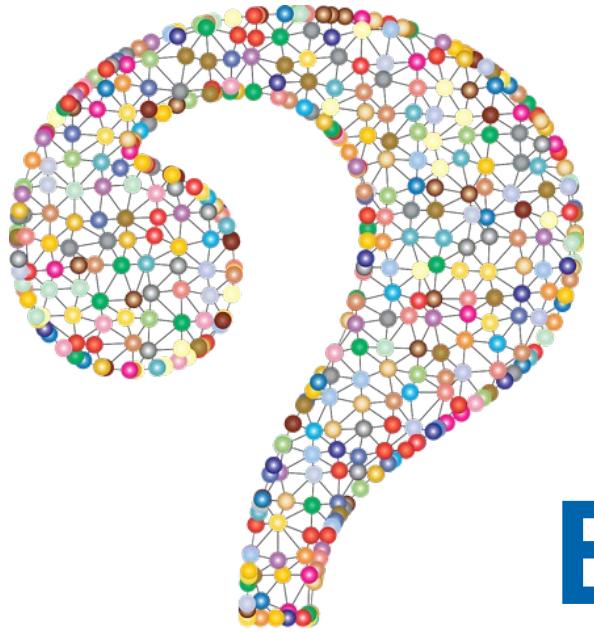
var request = HttpRequest.newBuilder(uri).GET().build();
var asString = HttpResponse.BodyHandlers.ofString();

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

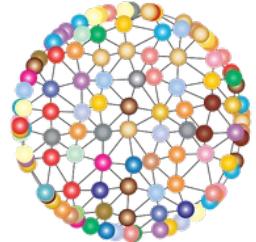
printResponseInfo(response);

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

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



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



HTTP/2 API Async I



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

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

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

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

HTTP/2 API Async I – geschickt warten mit vorzeitigem Abbruch

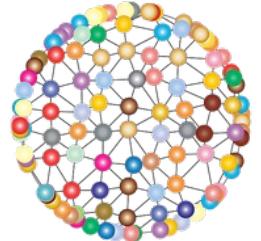


```
static void wait2secForCompletion(final CompletableFuture<?> asyncResponse)
    throws InterruptedException
{
    int i = 0;
    while (!asyncResponse.isDone() && i < 10)
    {
        System.out.println("Step " + i);
        i++;

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



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



HTTP/2 API Async II



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

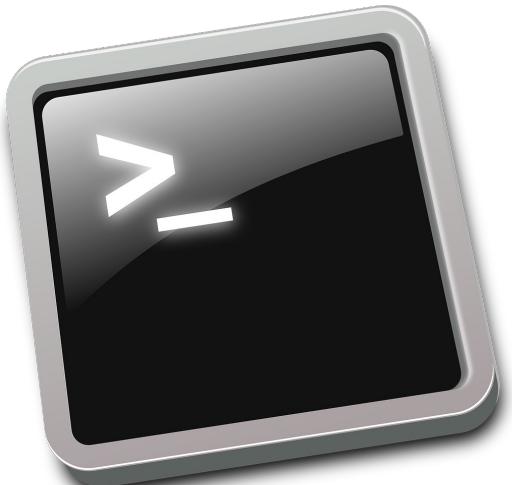
var request = HttpRequest.newBuilder(uri).GET().build();
var asString = HttpResponse.BodyHandlers.ofString();
var httpClient = HttpClient.newHttpClient();
var asyncResponse = httpClient.sendAsync(request, asString);

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

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



Direct Compilation Launch Single-File Source-Code Programs



Direct Compilation



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

```
package direct.compilation;

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

java ./HelloWorld.java



Hello Execute After Compile

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



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

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

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

Direct Compilation – Shebang Execution



```
#!/usr/bin/java --source 11
import java.nio.file.*;

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

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

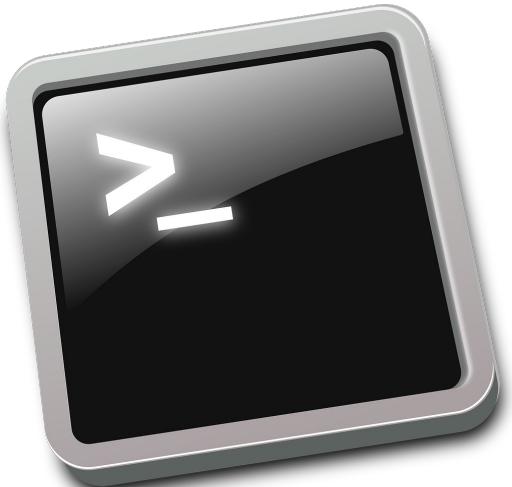
- Datei darf nicht mit ‘.java’ enden
- Dateiname UNABHÄNGIG von Klassennamen
- Datei muss executable (chmod +x) sein



DEMO



JShell



```
Michaels-MBP-2:~ michaeli$ jshell
| Welcome to JShell -- Version 15
| For an introduction type: /help intro

jshell> 2 * 3 * 5 * 7
[$1 ==> 210

jshell> int add(int val1, int val2)
| ...> {
| ...>     return val1 + val2;
| ...> }
| created method add(int,int)

jshell> import java.time.*

jshell> boolean isSunday(LocalDate date)
| ...> {
| ...>     return date.
adjustInto(      atStartOfDay(    atTime(        compareTo(      datesUntil(    equals(        format(
get(            getChronology() getClass()      getDayOfMonth()  getDayOfWeek()  getDayOfYear()  getEra()
getLong(         getMonth()       getMonthValue()  getYear()       hashCode()      isAfter(       isBefore(
isEqual(        isLeapYear()     isSupported()   lengthOfMonth()  lengthOfYear()  minus(        minusDays(
minusMonths(    minusWeeks()    minusYears()    notify()        notifyAll()    plus(         plusDays(
plusMonths(    plusWeeks()     plusYears()    query()        range(        toEpochDay()  toEpochSecond(
jshell> boolean isSunday(LocalDate date){
| ...> {
| ...>     return date.getDayOfWeek() == DayOfWeek.SUNDAY;
| ...> }
| created method isSunday(LocalDate)

jshell> isSunday(LocalDate.of(1971, 2, 7))
[$5 ==> true
```

Java-Kommandozeile jsshell



- Codeausführung ohne Klassen- und Methodendeklaration
- Semikolon am Zeilenende kann (teilweise) entfallen
- (teilweise) kein Exception Handling nötig
- für jeden Befehl wird automatisch eine Variable für den Rückgabewert angelegt (\$1, \$2, ...)
- Deklaration von Methoden und Klassen möglich
- Hinzufügen von Modulen möglich beim Start
- **jsshell** verlassen: **/exit**



DEMO

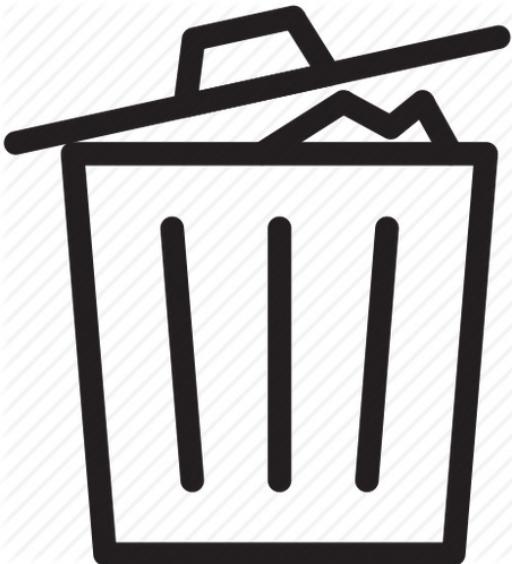
JShell – Was haben wir bisher gelernt?



- Berechnungen on the fly ausführen
- Variablen definieren
- Methoden definieren
- Praktisch forward referencing: Methode kann noch nicht definierte Methoden aufrufen
- Praktisch für **KLEINE** Experimente
- Editor seit Java 14 deutlich komfortabler, davor ziemlich katastrophal bezüglich Multi-Line-Editierung
- 3rd Party & Preview-Features
 - `jshell --class-path myOwnClassPath --enable-preview`



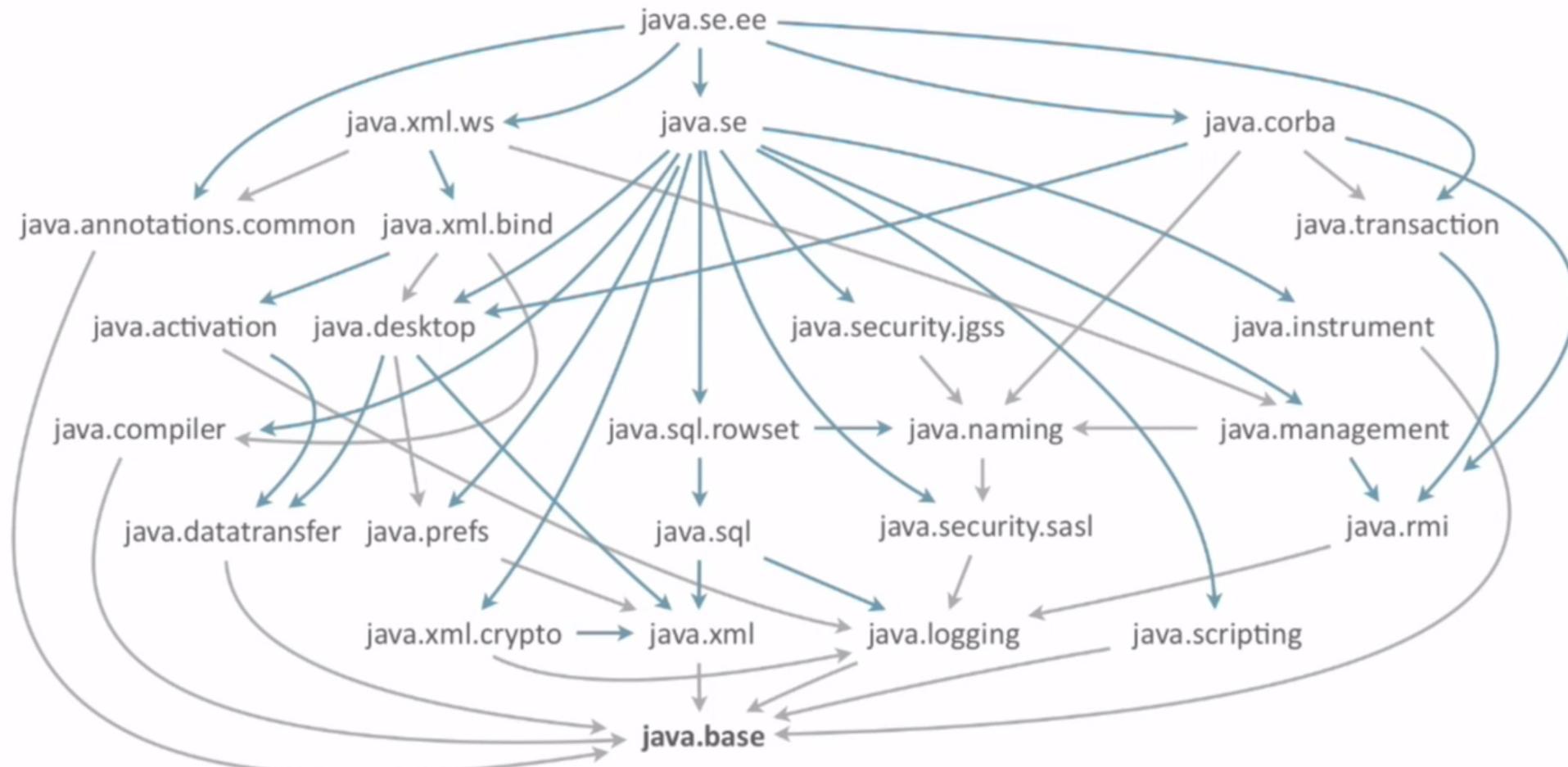
Deprecations



Entfernte APIs und Bibliotheken



- Modularisierung des JDK ermöglicht das Erkennen von Abhängigkeiten





- Modularisierung des JDK ermöglicht das Entfernen einzelner Module
- Doppelungen mit den Java-EE-Spezifikationen wurden entfernt:
 - **java.activation** JAF
 - **java.corba** CORBA
 - **java.transaction** JTA
 - **java.xml.bind** JAXB
 - **java.xml.ws** JAX-WS, SAAJ
 - **java.xml.ws.annotation** Common Annotations
 - **java.se.ee** Aggregator-Modul für diese Module

Entfernte APIs und Bibliotheken



- mit entfernten Modulen zusammenhängende Tools wurden entfernt
- **jdk.xml.ws** (JAX-WS)
 - **wsgen**
 - **wsimport**
- **jdk.xml.bind** (JAXB)
 - **schemagen**
 - **xjc**
- **java.corba** (CORBA)
 - **idlj, orbd, servertool, tnamesrv**



Ab Java 11 für JAXB: Externe Dependencies nötig

JAXBExample_JDK8
JAXBExample_JDK11

```
dependencies
{
    implementation "javax.xml.bind:jaxb-api:2.3.0"
    implementation "com.sun.xml.bind:jaxb-core:2.3.0"
    implementation "com.sun.xml.bind:jaxb-impl:2.3.0"
    runtimeOnly "javax.activation:activation:1.1.1"
}
```



Übungen PART 2

<https://github.com/Michaeli71/Best-Of-Java-11-21>





PART 3: Syntax-Erweiterungen in Java 12 bis 17

- Syntaxiserweiterungen bei switch
- Text Blocks
- Records
- Syntaxiserweiterung bei instanceof
- Sealed Types



Syntax-Erweiterungen



Switch Expressions



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

Switch Expressions: Blick zurück



- Abbildung von Wochentagen auf deren Länge ...

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

Switch Expressions als Abhilfe



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

```
DayOfWeek day = DayOfWeek.FRIDAY;  
int num0fLetters = -1;  
  
switch (day)  
{  
    case MONDAY, FRIDAY, SUNDAY    -> num0fLetters = 6;  
    case TUESDAY                   -> num0fLetters = 7;  
    case THURSDAY, SATURDAY        -> num0fLetters = 8;  
    case WEDNESDAY                 -> num0fLetters = 9;  
};
```

Switch Expressions als Abhilfe



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

Return-
Value

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

- Elegantere Schreibweise beim case:

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

Switch Expressions: Blick zurück ... Fallstricke



- Abbildung von Monaten auf deren Namen ...

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

Switch Expressions als Abhilfe



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

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



- Gegeben sei eine Aufzählung von Farben:

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

- Bilden wir diese auf die Anzahl an Buchstaben ab:

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

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

Switch Expressions: `yield` mit Rückgabe



Mit modernem Java wird wieder alles sehr klar und einfach:

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

Switch Expressions



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

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

SUNDAY is FUN DAY
6



Text Blocks



Text Blocks



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

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

Text Blocks



- NEU

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

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

Text Blocks



Traditional String Literals

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

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

Text Blocks



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

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

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

return result;
}
```

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

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

Text Blocks



- NEU

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

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

Text Blocks



- NEU

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

Besonderheit Ausrichtung bei Text Blocks



```
jshell> var multiLine = """"  
...>           | Eins  
...>           | Zwei  
...>           | Drei  
...>           | Vier  
...>           """"  
multiLine ==> "| Eins\n| Zwei\n| Drei\n| Vier\n"
```

```
jshell> var multiLine = """"  
...>           - Eins  
...>           - Zwei  
...>           - Drei  
...>           - - - Vier  
...>           """"  
multiLine ==> "_ Eins\n_ Zwei\n - Drei\n _ - - Vier\n"
```

Besonderheit bei Text Blocks



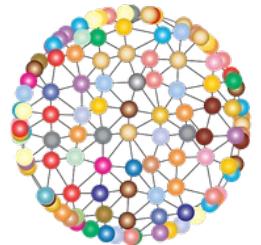
```
String text = """  
    This is a string split \  
    into several smaller \  
    strings.\ \  
    """;  
  
System.out.println(text);
```

This is a string split into several smaller strings.



Records





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

Erweiterung Record



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

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

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

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

Erweiterung Record

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

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

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

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

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

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

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

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

    // Zugriffsmethoden auf x und y
}
```

Records und zusätzliche Konstruktoren und Methoden



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

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

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

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

Records für DTO / Parameter Value Objects



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

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

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

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

Records für komplexere Rückgabewerte und Parameter



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

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

```
IntStringReturnValue calculateTheAnswer()
{
    // 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 Tupel? – Ausflug Pair<T>

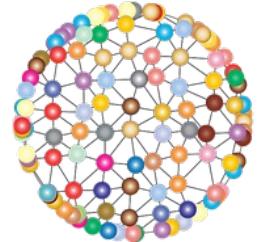


- Was ist an diesem self made Pair falsch?

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

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

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



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

Records für Pairs und Tupel



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

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



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

Records mit Gültigkeitsprüfung



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

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

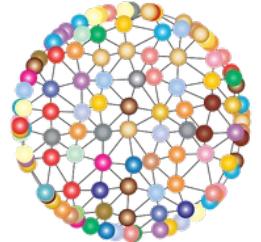
Records mit Gültigkeitsprüfung (Kurzschreibweise)



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



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



All in Beispiel



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

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



Records

Beyond the Basics



Speicherung in verschiedenen Sets



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

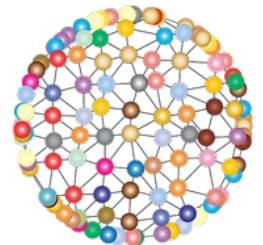
```
record SimplePerson(String name, int age, String city) {}
```

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

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

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

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



Das sollte das Ergebnis sein, oder?

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

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

Speicherung in verschiedenen Sets



```
[SimplePerson[name=Michael, age=51, city=Zürich], SimplePerson[name=Anton, age=42, city=Aachen]]  
Exception in thread "main" java.lang.ClassCastException: class
```

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

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

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

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

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

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

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

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

Records und Interfaces



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

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

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

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



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

Records und Interfaces + Comparator



- Korrigieren wir die Definition des Comparators:

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

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



Builder ...



Builder like



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

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

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

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

Builder like



- Problemfeld, viele Attribute

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

- Aber: ComplexPersonBuilder müsste man selbst implementieren => 😞



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

Builder like



- **Records für einzelne Bestandteile definieren**

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

record BodyInfo(int height, int weight) {}

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

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

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



Date Range ... Immutability

A large, stylized digital clock displaying the date 12.12 in red and white on a black background. The numbers are bold and have a three-dimensional, slightly rounded appearance. The red outline is thick and prominent against the black background.

Record für Datumsbereich



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

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

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

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

- Gültigkeitsprüfung für Invariante `start < end` einbauen

Record für Datumsbereich



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

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

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

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

Record für Datumsbereich



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

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

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

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

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

Record für Datumsbereich



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

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

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

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

Records



DEMO & Hands on



Pattern Matching bei instanceof



Pattern Matching bei instanceof



- ALT

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



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

Pattern Matching bei instanceof



- **ALT**

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

- **NEU**

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

Pattern Matching bei instanceof



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

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

Pattern Matching bei instanceof



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



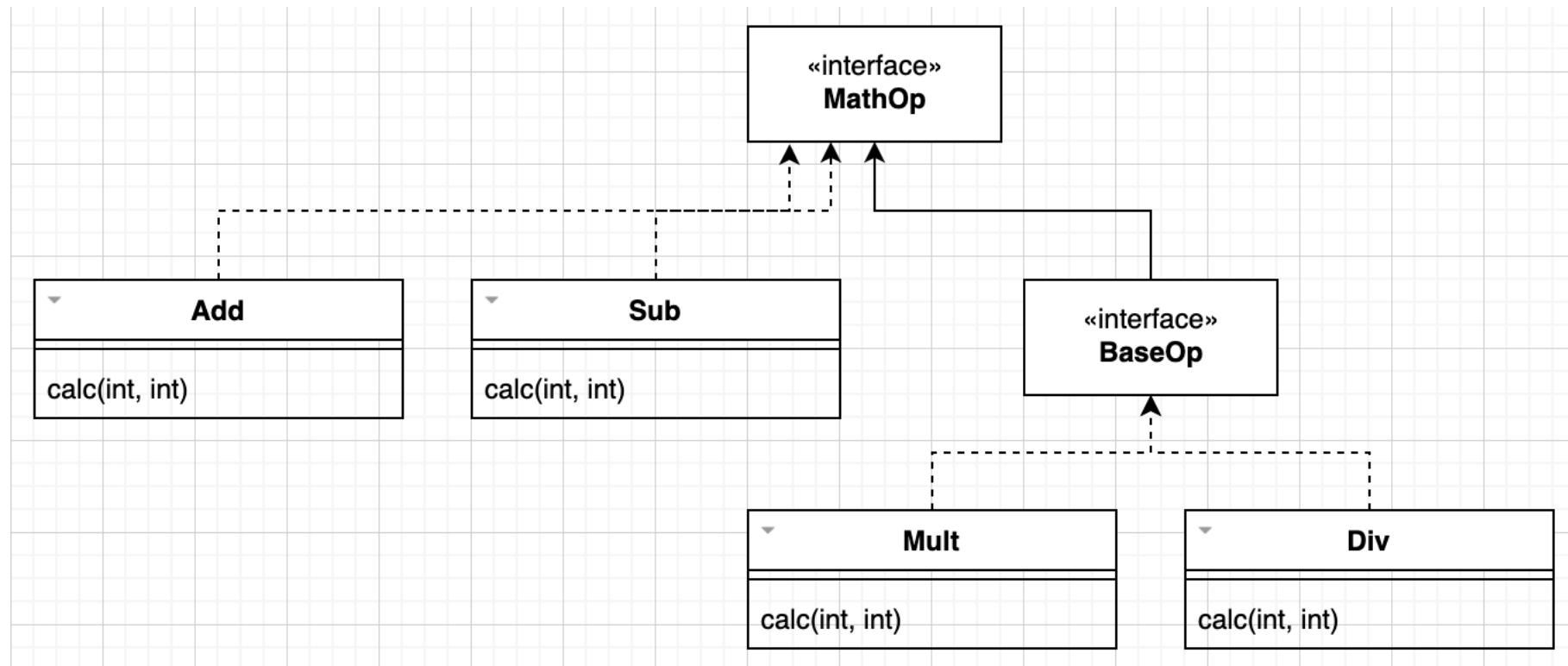
Sealed Types



Sealed Types



- **Vererbung steuern** und spezifizieren, welche Klassen eine Basisklasse erweitern können, also welche anderen Klassen oder Interfaces davon Subtypen bilden dürfen.



Sealed Types – Vererbung steuern



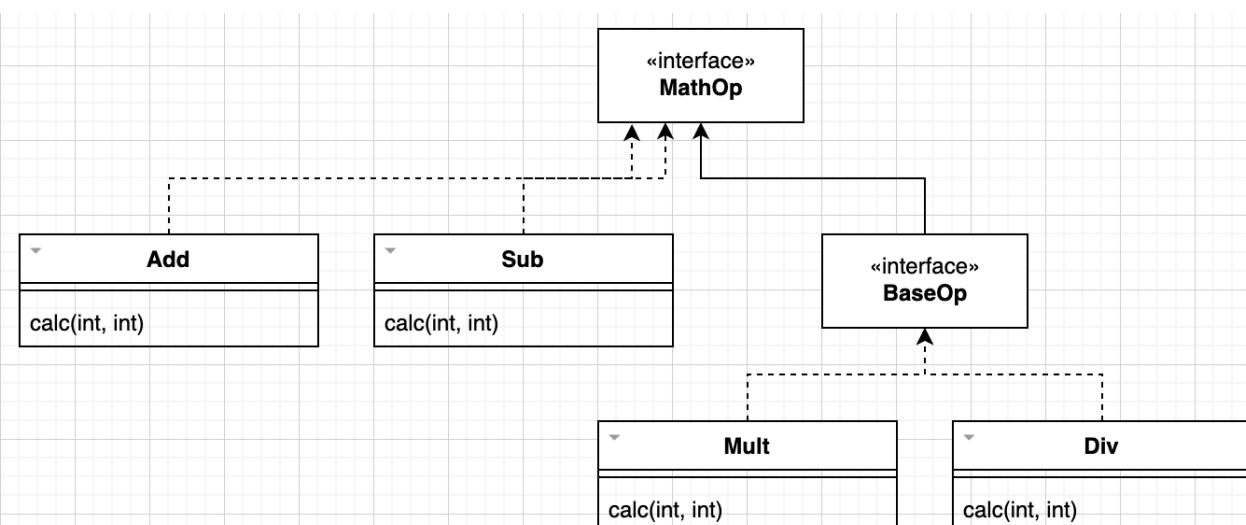
- Spezifizieren, welche anderen Klassen oder Interfaces davon Subtypen bilden dürfen.

```
public class SealedTypesExamples
{
    sealed interface MathOp
        permits BaseOp, Add, Sub // <= erlaubte Subtypen
    {
        int calc(int x, int y);
    }
}
```

// Mit non-sealed kann man innerhalb der Vererbungshierarchie Basisklassen bereitstellen

```
non-sealed class BaseOp implements MathOp // <= Basisklasse nicht versiegeln
{
    @Override
    public int calc(int x, int y)
    {
        return 0;
    }
}
...
```

Mit sealed können wir eine Vererbungshierarchie versiegeln und nur die explizit angegebenen Typen erlauben. Diese müssen sealed, non-sealed oder final sein.

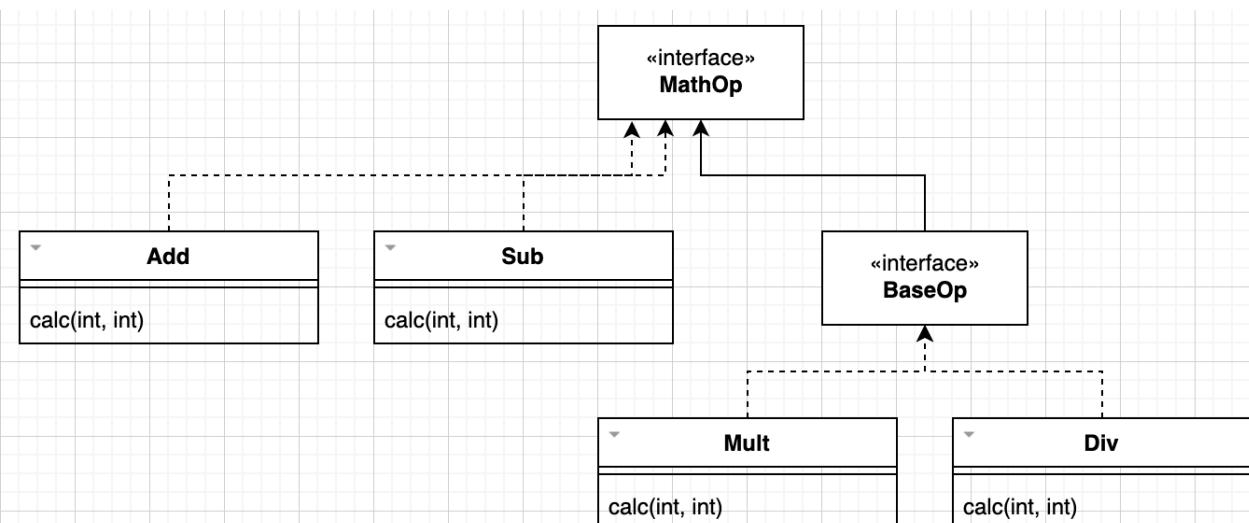


Sealed Types



```
..  
// direkte Implementierung muss final sein  
final class Add implements MathOp  
{  
    @Override  
    public int calc(int x, int y)  
    {  
        return x + y;  
    }  
}  
final class Sub implements MathOp  
{  
    ...  
}  
final class Mult extends BaseOp  
{  
}  
final class Div extends BaseOp  
{  
}
```

- Eine als sealed markierte Klasse muss Subklassen besitzen, die wiederum hinter permits aufgeführt werden
- Eine als non-sealed markierte Klasse kann als Basisklasse fungieren und von dieser können Klassen abgeleitet werden.
- Eine als final markierte Klasse bildet – wie gewohnt – den Endpunkt einer Ableitungshierarchie.
- **Gleiches gilt auch für Interfaces**



Wissenswerts zu Sealed Types



- **festlegen, welche andere Klassen oder Interfaces davon Subtypen bilden dürfen.**
 - **Sealed Types können bei der Entwicklung von Bibliotheken helfen:
Verhalten über Interfaces exponieren, aber Kontrolle über mögliche Implementierungen behalten**
 - **Sealed Types schränken bezüglich der Erweiterbarkeit von Klassenhierarchien ein
und sollten daher mit Bedacht verwendet werden.**
-



Übungen PART 3

<https://github.com/Michaeli71/Best-Of-Java-11-21>





PART 4: Neuerungen und Änderungen in den APIs in Java 12 bis 17

- String APIs
- Teeing()-Kollektor
- Stream.toList()



Erweiterungen in der Klasse String



Erweiterung in `java.lang.String`



- Die Klasse `String` existiert seit JDK 1.0 und hat zwischenzeitlich nur wenige API-Änderungen erfahren.
- Mit Java 11 änderte sich das. Es wurden 6 neue Methoden eingeführt.
- In Java 12 werden folgende zwei ergänzt:
 - `indent()` – Passt die Einrückung eines Strings an
 - `transform()` – Ermöglicht Aktionen zur Transformation eines Strings

Erweiterung in `java.lang.String`: `indent()`



- Diese Methode fügt 'n' Leerzeichen (U+00200) vor jeder Zeile an und fügt dann einen Zeilenvorschub "\n" hinzu.

```
var test = "Test".indent(4);
System.out.println("  " + test + "  ");
System.out.println(test.length());
```

```
'      Test
'
9
```

- Es sind aber auch negative Werte erlaubt:

```
var removeTest = "      6789".indent(-5);
System.out.println("  " + removeTest + "  ");
System.out.println(removeTest.length());
```

```
'6789
'
5
```

Erweiterung in `java.lang.String`: `transform()`



- **Einsatz von `transform()` für eine Verarbeitungskette**

- In UPPERCASE wandeln
- Dann T entfernen
- Zum Schluss aufspalten

```
var text = "This is a Test";
```

```
// chaining of operations
```

```
var result = text.transform(String::toUpperCase).
            transform(str -> str.replaceAll("T", "")).
            transform(str -> str.split(" "));
```

```
System.out.println(Arrays.asList(result));
```

Sieht jemand den potenziellen Vorteil?

[HIS, IS, A, ES]

- Analog zu `map()` in Streams, zum Hintereinanderschalten von Transformationen

- Eher theoretisch praktisch ☺

```
public <R> R transform(Function<? super String,
                      ? extends R> f)
{
    return f.apply(this);
}
```



Teeing()-Kollektor





Das umfangreiche **Stream-API** besitzt eine Menge an Kollektoren:

- `toCollection()`, `toList()`, `toSet()` ... – Sammlung der Elemente des Stream in die entsprechende Collection.
- `joining()` – Zusammenfügen von Elementen als String
- `groupingBy()` – Gruppierungen aufbereiten. Beispiel: Histogramme
Zudem konnte man dort weitere Kollektoren übergeben.

Im Kontext von `groupingBy()` gibt es allerdings einige spezielle Anwendungsfälle, für die es vor Java 9 keinen Kollektor gab.



Einschub Stream API – Spezieller Kollektor





Das umfangreiche **Stream-API** besitzt eine Menge an Kollektoren und wurde um folgende zwei erweitert:

- **filtering()** – Filtern der Elemente des Stream
- **flatMapping()** – Mappen und Zusammenfügen von Elementen

⇒ Beide sind vor allem im Kontext von `groupingBy()` nützlich.



Die Neuerung `Collectors.filtering()` mit der Analogie `filter()`

Beispiel zum Einstieg:

```
var programming = Stream.of("Java", "JavaScript", "Groovy", "JavaFX", "Spring", "Java");

final Set<String> result1 = programming.filter(name -> name.contains("Java"))
                                         .collect(toSet());
```

Mit Filtering Collector:

```
final Set<String> result2 = programming.collect(
    filtering(name -> name.contains("Java")), toSet());
```

Als Ergebnis:

[JavaFX, Java, JavaScript]

Zweite Variante weniger intuitiv und verständlich als die erste. Vorteil erst mit `groupingBy()`

Stream API Collectors in JDK 9



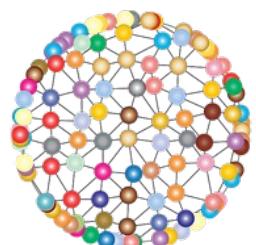
Beispiel zum Einstieg:

Nehmen wir an, wir wollten basierend auf den Nennungen eine Art Histogramm erstellen. Beginnen wir mit einer Umsetzung mit Java-8-Bordmitteln:

```
final List<String> programming = List.of("Java", "JavaScript", "Groovy", "JavaFX",
                                         "Spring", "Java");
final Map<String, Long> result = programming.stream().
    filter(name -> name.contains("Java")).
    collect(groupingBy(name -> name, counting()));
System.out.println(result);
```

Als Ergebnis:

{JavaFX=1, Java=2, JavaScript=1}



**Was machen wir, wenn bei
der Histogrammaufbereitung
auch Eingaben von Interesse
sind, die der Bedingung
nicht entsprechen?**

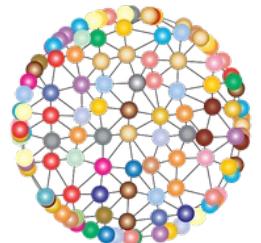


Geänderte Anforderung: Wenn bei der Histogrammaufbereitung **auch Eingaben** von Interesse sind, die der **Bedingung nicht entsprechen**, würden diese durch eine vorherige Filterung verloren gehen. Für unseren Anwendungsfall müssen wir zunächst gruppieren und danach filtern und nur diejenigen zählen, die relevant sind:

```
final Map<String, Long> result = programming.stream().  
    collect(groupingBy(name -> name,  
                      filtering(name -> name.contains("Java")),  
                      counting()));  
  
System.out.println(result);
```

Als Ergebnis:

```
{JavaFX=1, Java=2, JavaScript=1, Spring=0, Groovy=0}
```



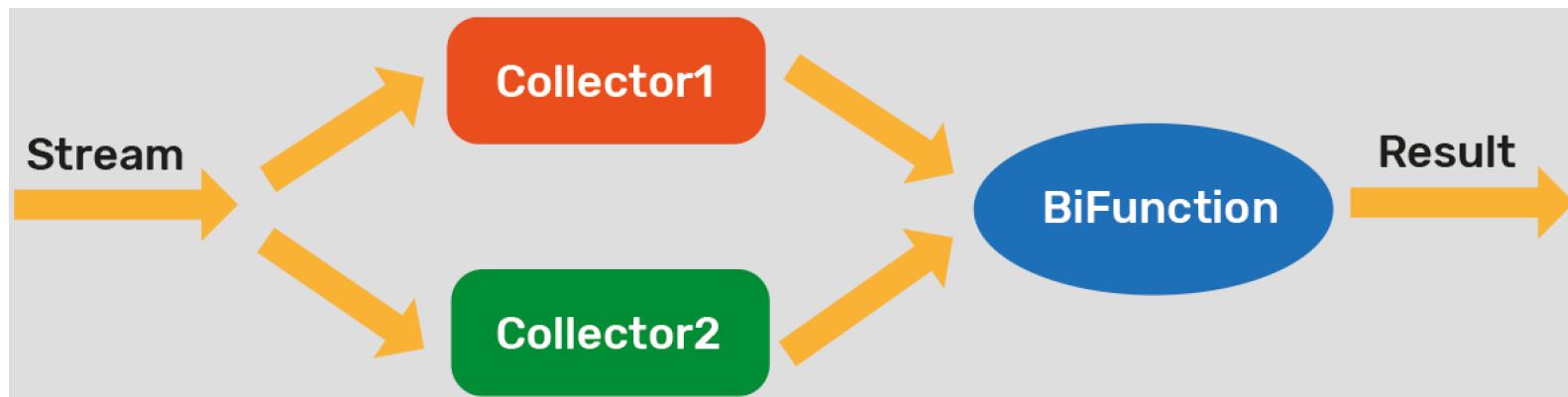
Was fehlt denn noch?



Wie sieht es denn mit dem Zusammenfassen von Streams aus?

"...returns a Collector that is a composite of two downstream collectors. Every element passed to the resulting collector is processed by both downstream collectors, then their results are merged using the specified merge function into the final result."

```
static <T, R1, R2, R> Collector<T, ?, R> teeing(Collector<? super T, ?, R1> downstream1,  
                                         Collector<? super T, ?, R2> downstream2,  
                                         BiFunction<? super R1, ? super R2, R> merger)
```



Kollektoren



```
public static void main(String[] args)
{
    var firstSixNumbers = Stream.of(1, 2, 3, 4, 5, 6);
    System.out.println(calcCountAndSum(firstSixNumbers));

    var primeNumbers = Stream.of(2, 3, 5, 7, 11, 13, 17);
    System.out.println(calcCountAndSum(primeNumbers));
}

private static LongPair calcCountAndSum(Stream<Integer> numbers)
{
    return numbers.collect(Collectors.teeing(
        Collectors.counting(),
        Collectors.summingLong(n -> n),
        // (count, sum) -> new LongPair(count, sum))
        LongPair ::new));
}
```

Kollektoren



```
public static void main(String[] args)
{
    var firstSixNumbers = Stream.of(1, 2, 3, 4, 5, 6);
    System.out.println(calcCountAndSum(firstSixNumbers));

    var primeNumbers = Stream.of(2, 3, 5, 7, 11, 13, 17);
    System.out.println(calcCountAndSum(primeNumbers));
}

private static LongPair calcCountAndSum(Stream<Integer> numbers)
{
    return numbers.collect(Collectors.teeing(
        Collectors.counting(),
        Collectors.summingLong(n -> n),
        LongPair::new));
}
```



```
Pair<T> [first=6, second=21]
Pair<T> [first=7, second=58]
```



Aufgabe: Aus einem Stream von Strings sollen unterschiedliche Elemente herausgefiltert und dann die Ergebnisse kombiniert werden.

- Hier hilft der `filtering()`-Kollektor aus Java 9 sowie `teeing()`:

```
var names = Stream.of("Michael", "Tim", "Tom", "Mike", "Bernd");

final Predicate<String> startsWithMi = text -> text.startsWith("Mi");
final Predicate<String> endsWithM = text -> text.endsWith("m");
final BiFunction<List<String>, List<String>, List<List<String>>> combineLists =
    (list1, list2) -> List.of(list1, list2);

var result = names.collect(teeing(filtering(startsWithMi, toList()),
                                filtering(endsWithM, toList()),
                                // (list1, list2) -> List.of(list1, list2)
                                combineLists));
System.out.println(result);
```



Aufgabe: Aus einem Stream von Strings sollen unterschiedliche Elemente herausgefiltert und dann die Ergebnisse kombiniert werden.

```
var names = Stream.of("Michael", "Tim", "Tom", "Mike", "Bernd");

final Predicate<String> startsWithMi = text -> text.startsWith("Mi");
final Predicate<String> endsWithM = text -> text.endsWith("m");

var result = names.collect(teeing(filtering(startsWithMi, toList()),
    [Michael, Mike]
    filtering(endsWithM, toList())),
    (list1, list2) -> List.of(list1, list2));

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



Aufgabe: Aus einem Stream von Strings sollen unterschiedliche Elemente herausgefiltert und dann die Ergebnisse kombiniert werden.

```
var names = Stream.of("Michael", "Tim", "Tom", "Mike", "Bernd");

final Predicate<String> startsWithMi = text -> text.startsWith("Mi");
final Predicate<String> endsWithM = text -> text.endsWith("m");

var result = names.collect(teeing(filtering(startsWithMi, toList()),
    [Michael, Mike]
    filtering(endsWithM, toList())),
    [Tim, Tom]
    (list1, list2) -> List.of(list1, list2));

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



Aufgabe: Aus einem Stream von Strings, sollen unterschiedliche Elemente herausgefiltert und dann die Ergebnisse kombiniert werden.

```
var names = Stream.of("Michael", "Tim", "Tom", "Mike", "Bernd");

final Predicate<String> startsWithMi = text -> text.startsWith("Mi");
final Predicate<String> endsWithM = text -> text.endsWith("m");

var result = names.collect(teeing(filtering(startsWithMi, toList()),
    [Michael, Mike]
    filtering(endsWithM, toList())),
    [Tim, Tom]
    (list1, list2) -> List.of(list1, list2));

System.out.println(result);
    [[Michael, Mike], [Tim, Tom]]
```



Java 16



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



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

FINALLY... `toList()`



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



Übungen PART 4

<https://github.com/Michaeli71/Best-Of-Java-11-21>





PART 5: Neuerungen in der JVM in Java 12 bis 17

- Hilfreiche NullPointerExceptions
 - JMH (Microbenchmarks)
 - JPackage
 - ~~JavaScript Engine~~
-



N P E

Hilfreiche NullPointerExceptions



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

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

Hilfreiche NullPointerExceptions



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

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

-XX:+ShowCodeDetailsInExceptionMessages

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

Hilfreiche NullPointerExceptions



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

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

Don't do this in production!

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

Hilfreiche NullPointerExceptions



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

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

Hilfreiche NullPointerExceptions



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

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

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



JMH



Benchmarking Intro



- Mitunter sind einige Teile der Software nicht so performant, wie benötigt.
- Zur Optimierung der Performance gibt es verschiedene Hilfsmittel und Ebenen
- Generell sollte man zunächst gründlich messen und nur mit Bedacht optimieren.
- Wieso?
 - bereits diverse Optimierungen in die JVM eingebaut
 - nicht trivial, da die Messungen unter möglichst gleichen Bedingungen (CPU-Last, Speicherverbrauch usw.) geschehen sollten, für vergleichbare Resultate
 - rein auf Basis von Vermutungen liegt man häufig falsch
- keinesfalls nur aufgrund von Vermutungen, sondern basierend auf Messungen:
 - einfache Start-/Stop-Messungen
 - empfehlenswerter sind ausgeklügeltere Verfahren mit mehreren Durchläufen

Einfache Start-/Stop-Messungen (Bitte nicht machen!!)



- Einfache Start-/Stop-Messungen mit `System.currentTimeMillis()`

```
// ACHTUNG: keine gute Variante
final long startTimeMs = System.currentTimeMillis();

someCodeToMeasure();

final long stopTimeMs = System.currentTimeMillis();
final long duration = stopTimeMs - startTimeMs;
```

- den zu messenden Programmteil mit `System.currentTimeMillis()` umklammern
- Als Art Stoppuhr nutzen, indem man die Differenz zwischen den Werten ermittelt
- Genauer – aber nicht wirklich besser – wird es mit `System.nanoTime()`

Wiederholte Start-/Stop-Messungen (Möglichst vermeiden!)



- Wiederholte Start-/Stop-Messungen und besserer (genauerer) Timer

```
// bessere Variante
final long startTimeNs = java.lang.System.nanoTime();

for (int i = 0; i < MAX_ITERATIONS; i++)
{
    someCodeToMeasure();
}

final long stopTimeNs = java.lang.System.nanoTime();

final long avgDurationNs = (stopTimeNs - startTimeNs) / MAX_ITERATIONS;
```

- Durch mehrere Durchläufe und Durchschnittsbildung weniger anfällig für Systemlastschwankungen oder sonstige Störeinflüsse.
- recht einfach auch Minimal- und Maximaldauer oder Standardabweichung zu ermitteln.

Wiederholte Start-/Stop-Messungen mit Warm-up (Wird kompliziert)



- Einschwing-Effekte: Erst nach einer gewissen Anzahl an Durchläufen zeigt eine Funktionalität ihre optimale Laufzeit:

```
// noch bessere Variante der Messung durch Warm-up
for (int i = 0; i < MAX_WARMUP_ITERATIONS; i++)
{
    someCodeToMeasure();
}

// eigentliche Messung nach Warm-up
final long startTimeNs = java.lang.System.nanoTime();

for (int i = 0; i < MAX_ITERATIONS; i++)
{
    someCodeToMeasure();
}
final long stopTimeNs = java.lang.System.nanoTime();

final long avgDurationNs = (stopTimeNs - startTimeNs) / MAX_ITERATIONS;
```



- **JEP 230 goal:** add a basic suite of microbenchmarks to the JDK source code, and make it easy for developers to run existing microbenchmarks and create new ones.
 - Basiert auf [Java Microbenchmark Harness \(JMH\)](#)
 - Framework zum Erstellen von Microbenchmark-Tests
 - Microbenchmarking = Optimierungsebene einzelner bzw. weniger Anweisungen
 - Berücksichtigt verschiedene externe Störeinflüsse und Schwankungen
 - Umfangreich, aber meistens einfach konfigurierbar
 - Macht das Schreiben von Benchmarks fast so einfach wie Unit Testing mit JUnit
-

Microbenchmarks mit JMH



- Eine Performance-Test-Umgebung kann JMH mit folgendem Maven-Kommando erzeugen:

```
mvn archetype:generate \
  -DinteractiveMode=false \
  -DarchetypeGroupId=org.openjdk.jmh \
  -DarchetypeArtifactId=jmh-java-benchmark-archetype \
  -DgroupId=org.sample \
  -DartifactId=jmh-test \
  -Dversion=1.0-SNAPSHOT
```

Microbenchmarks mit JMH



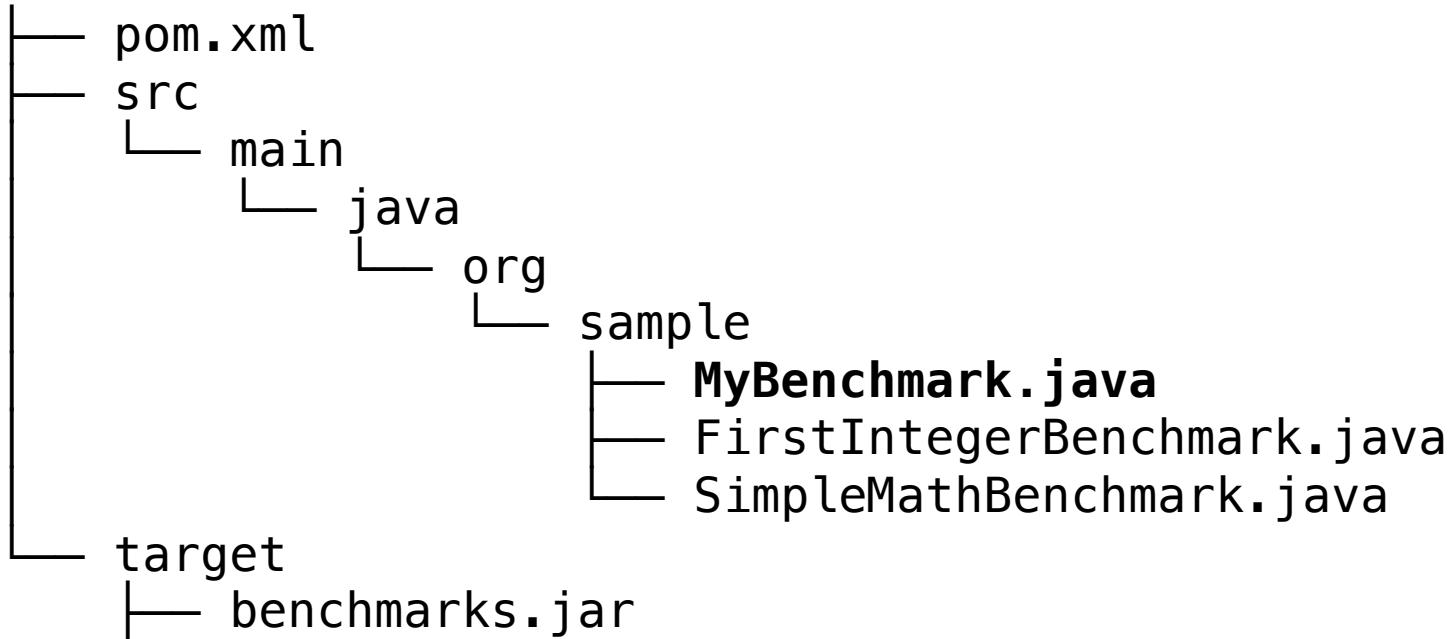
- Als Grundgerüst wird eine Klasse MyBenchmark erzeugt:

```
public class MyBenchmark
{
    @Benchmark
    public void testMethod()
    {
        // This is a demo/sample template for building your JMH benchmarks. Edit
        // as needed.
        // Put your benchmark code here.
    }
}
```

- JMH arbeitet mit Annotations und integriert basierend darauf verschiedene Messungen.
-



- Basierend auf dem Grundgerüst kann man eigene Benchmark-Klassen erstellen:



- Mit 2 Schritten zum Benchmark
 - 1) mvn clean package
 - 2) java -jar target/benchmarks.jar

Eigener Microbenchmark mit JMH



```
@Measurement(iterations = 3, time = 1000, timeUnit = TimeUnit.MILLISECONDS)
@Warmup(iterations = 7, time = 1000, timeUnit = TimeUnit.MICROSECONDS)
@Fork(4)
public class FirstIntegerBenchmark
{
    @Benchmark
    public String numberAsHexString()
    {
        int dec = 123456789;
        return Integer.toHexString(dec);
    }

    @Benchmark
    public String numberAsBinaryString()
    {
        int dec = 123456789;
        return Integer.toBinaryString(dec);
    }
}
```

Eigene Microbenchmarks mit JMH



```
// Based on https://www.retit.de/continuous-benchmarking-with-jmh-and-junit-2/
public class SearchBenchmark {
    @State(Scope.Thread)
    public static class SearchState {
        public String text = "ABCDEFGHIJKLMNOPQRSTUVWXYZ__abcdefghijklmnopqrstuvwxyz";
    }

    @Benchmark
    public int testIndex0f(SearchState state) {
        return state.text.indexOf("M");
    }

    @Benchmark
    public int testIndex0fChar(SearchState state) {
        return state.text.indexOf('M');
    }

    @Benchmark
    public boolean testContains(SearchState state) {
        return state.text.contains("M");
    }
}
```

Eigene Microbenchmarks mit JMH



```
@BenchmarkMode(Mode.AverageTime)
@Fork(2)
@State(Scope.Benchmark)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
public class SimpleStringJoinBenchmark
{
    private String from = "Michael";
    private String to = "Participants";
    private String subject = "Benchmarking with JMH";

    @Benchmark
    public String stringPlus(Blackhole blackhole)
    {
        String result = "From: " + from + "\nTo: " + to + "\nSubject: " + subject;
        blackhole.consume(result);
        return result;
    }
}
```

Inspiriert von <http://alblue.bandlem.com/2016/04/jmh-stringbuffer-stringbuilder.html>,
aber hier mit Blackhole und leicht abgewandelt

Eigene Microbenchmarks mit JMH



```
@Benchmark
public String stringPlusEqual(Blackhole blackhole)
{
    String result = "From: " + from;
    result += "\nTo: " + to;
    result += "\nSubject: " + subject;

    blackhole.consume(result);
    return result;
}

@Benchmark
public String builderAppendChained(Blackhole blackhole)
{
    String result = new StringBuilder().append("From: ").append(from).
                                         append("\nTo: ").append(to).
                                         append("\nSubject: ").append(subject).
                                         toString();

    blackhole.consume(result);
    return result;
}
```

ACHTUNG – Eigene Microbenchmarks mit JMH



```
@State(Scope.Benchmark)
public static class MyBenchmarkState {
    @Param({ "10000", "100000" })
    public int value;
}

@Benchmark
public String stringPlusABC(MyBenchmarkState state, Blackhole blackhole) {
    String result = "";
    for (int i = 0; i < state.value; i++) {
        result += "ABC";
    }

    blackhole.consume(result);
    return result;
}
```

ACHTUNG – Eigene Microbenchmarks mit JMH



```
@Benchmark
public String stringConcatABC(MyBenchmarkState state, Blackhole blackhole) {
    String result = "";
    for (int i = 0; i < state.value; i++) {
        result = result.concat("ABC");
    }
    blackhole.consume(result);
    return result;
}
```

```
@Benchmark
public String concatUsingStringBuilder(MyBenchmarkState state, Blackhole blackhole) {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < state.value; i++) {
        sb.append("ABC");
    }
    String result = sb.toString();
    blackhole.consume(result);
    return result;
}
```



- **POM anpassen:**

```
<!-- Java source/target to use for compilation. -->
<javac.target>1.8</javac.target>
=>
<javac.target>17</javac.target>

<groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.8.0</version>
=>
<version>3.8.1</version>
```



Was wollen wir messen?

- `indexOf(String)`, `indexOf(char)`, `contains(String)`
 - `for`, `forEach`, `while`, `Iterator`
 - `String +=`, `String.concat()`, `StringBuilder.append()`
-



Beispielergebnisse

Benchmark	Mode	Cnt	Score	Error	Units
LoopBenchmark.loopFor	avgt	10	5.039 ± 0.134	ms/op	
LoopBenchmark.loopForEach	avgt	10	5.308 ± 0.322	ms/op	
LoopBenchmark.loopIterator	avgt	10	5.466 ± 0.528	ms/op	
LoopBenchmark.loopWhile	avgt	10	5.026 ± 0.218	ms/op	

Benchmark	Mode	Cnt	Score	Error	Units
SearchBenchmark.testContains	avgt	15	7.712 ± 0.241	ns/op	
SearchBenchmark.testIndexOf	avgt	15	7.797 ± 0.475	ns/op	
SearchBenchmark.testIndexOfChar	avgt	15	7.046 ± 0.070	ns/op	



Beispielergebnisse

Benchmark	Mode	Cnt	Score	Error	Units
SimpleStringJoinBenchmark.builderAppendChained	avgt	10	46.308	± 7.950	ns/op
SimpleStringJoinBenchmark.builderMultipleAppend	avgt	10	127.312	± 14.935	ns/op
SimpleStringJoinBenchmark.stringConcat	avgt	10	83.888	± 18.979	ns/op
SimpleStringJoinBenchmark.stringPlus	avgt	10	38.871	± 2.823	ns/op
SimpleStringJoinBenchmark.stringPlusEqual	avgt	10	39.346	± 3.015	ns/op



Beispielergebnisse

Benchmark		Mode	Cnt	Score	Error	Units
SimpleStringJoinBenchmark.builderAppendChained		avgt	10	46.308	± 7.950	ns/op
SimpleStringJoinBenchmark.builderMultipleAppend		avgt	10	127.312	± 14.935	ns/op
SimpleStringJoinBenchmark.stringConcat		avgt	10	83.888	± 18.979	ns/op
SimpleStringJoinBenchmark.stringPlus		avgt	10	38.871	± 2.823	ns/op
SimpleStringJoinBenchmark.stringPlusEqual		avgt	10	39.346	± 3.015	ns/op

Benchmark	(value)	Mode	Cnt	Score	Error	Units
StringJoinBenchmark.concatUsingStringBuilder	10000	avgt	10	0.016	± 0.001	ms/op
StringJoinBenchmark.concatUsingStringBuilder	100000	avgt	10	0.172	± 0.009	ms/op
StringJoinBenchmark.stringConcatABC	10000	avgt	10	9.634	± 0.812	ms/op
StringJoinBenchmark.stringConcatABC	100000	avgt	10	662.953	± 4.029	ms/op
StringJoinBenchmark.stringPlusABC	10000	avgt	10	7.926	± 0.149	ms/op
StringJoinBenchmark.stringPlusABC	100000	avgt	10	662.343	± 2.912	ms/op



JPackage



JPackage



▼ PackagingDemo

► JRE System Library [JavaSE-16]

▼ src/main/java

 ▼ de.java17

 ▼ ApplicationExample.java

 ▼ ApplicationExample

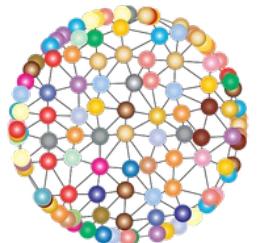
 main(String[]) : void

► src

 build.gradle

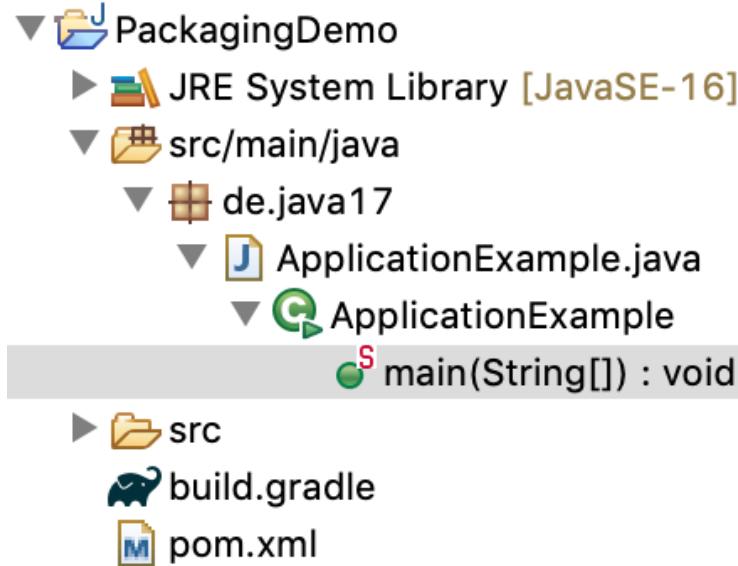
 pom.xml

```
public class ApplicationExample {  
    public static void main(String[] args) {  
        System.out.println("First JPackage");  
  
        JOptionPane.showConfirmDialog(null, "Generated by jpackage", "DEMO",  
            JOptionPane.OK_CANCEL_OPTION, JOptionPane.INFORMATION_MESSAGE, null)  
    }  
}
```

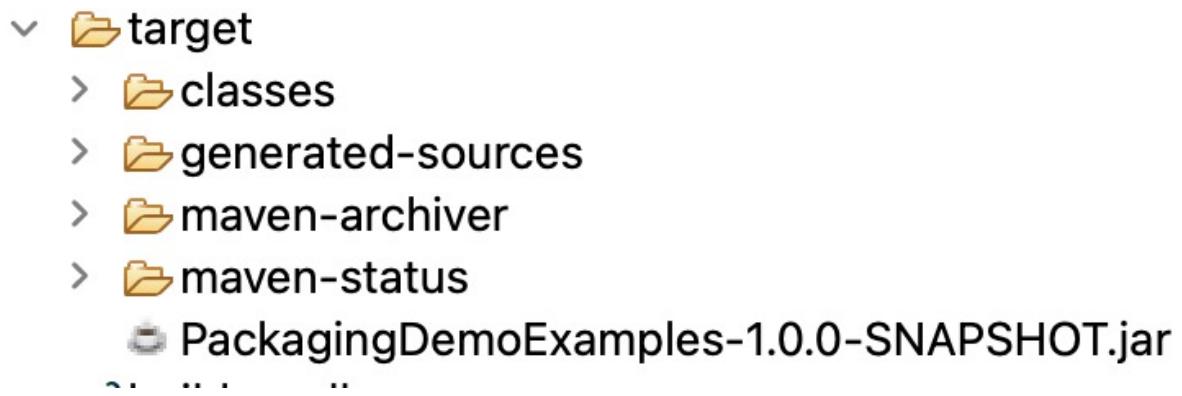


Wie bauen wir das?

JPackage



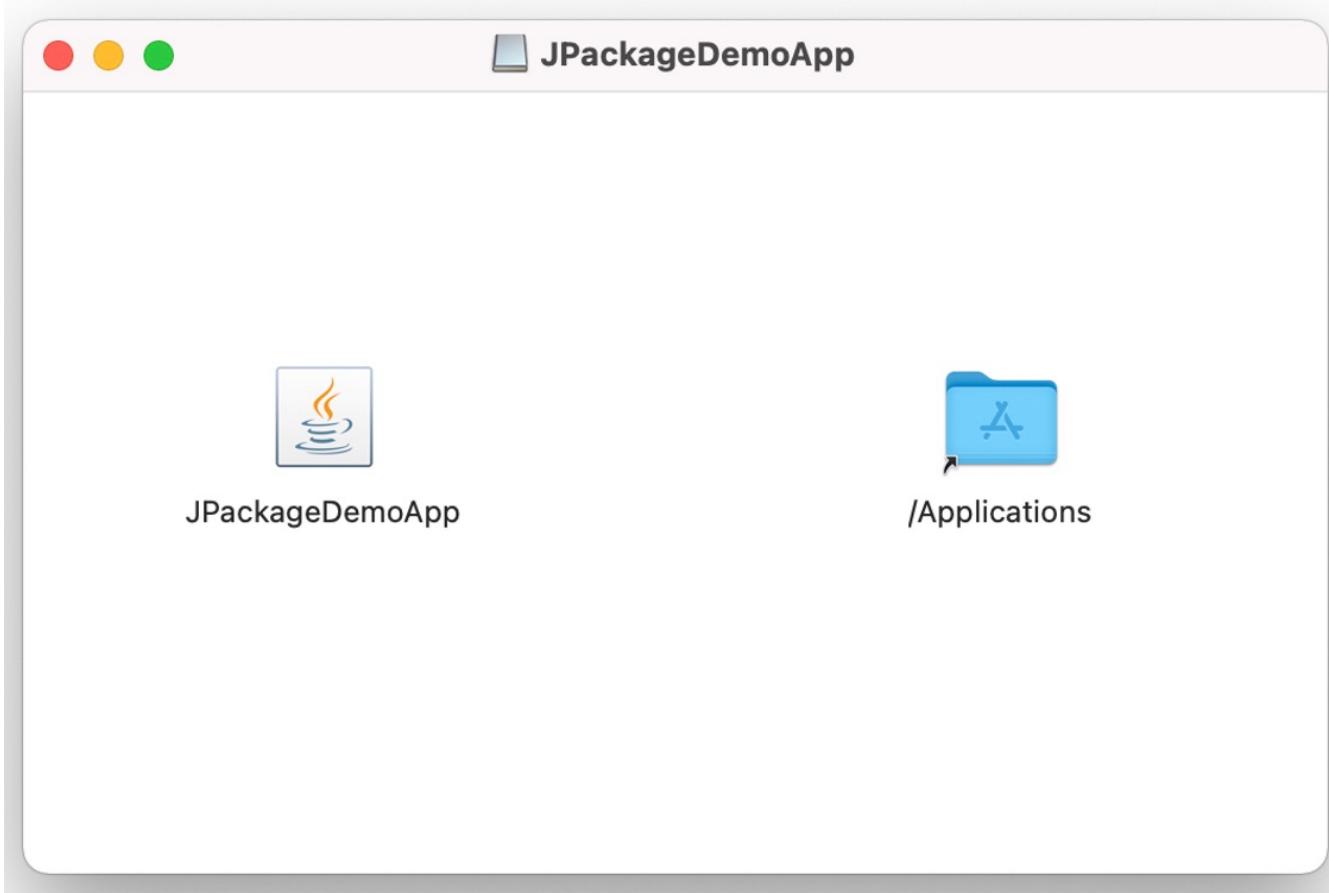
`mvn clean install`

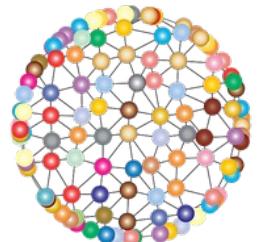


JPackage



```
jpackage --input target/ --name JPackageDemoApp --main-jar PackagingDemoExamples-1.0.0-SNAPSHOT.jar --main-class de.java17.ApplicationExample --type dmg --java-options '--enable-preview'
```





**Aber was machen wir mit 3rd
Party Libraries?**

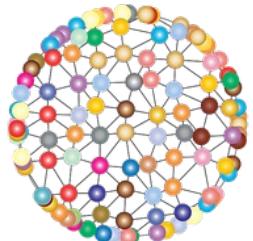
JPackage



```
public class ApplicationExample {  
  
    public static void main(String[] args) {  
        System.out.println("First JPackage");  
  
        Joiner joiner = Joiner.on(":");  
        String result = joiner.join(List.of("Michael", "mag", "Python"));  
        System.out.println(result);  
        JOptionPane.showConfirmDialog(null, result);  
    }  
}  
  
<!-- https://mvnrepository.com/artifact/com.google.guava/guava -->  
<dependency>  
  <groupId>com.google.guava</groupId>  
  <artifactId>guava</artifactId>  
  <version>31.1-jre</version>  
</dependency>
```



Wie wird die Bibliothek in die Anwendung eingebunden?





```
<plugin>
  <artifactId>maven-dependency-plugin</artifactId>
  <executions>
    <execution>
      <phase>process-sources</phase>
      <goals>
        <goal>copy-dependencies</goal>
      </goals>

      <configuration>
        <outputDirectory>target</outputDirectory>
        <includeScope>runtime</includeScope>
        <excludeScope>test</excludeScope>
      </configuration>
    </execution>
  </executions>
</plugin>
```

JPackage

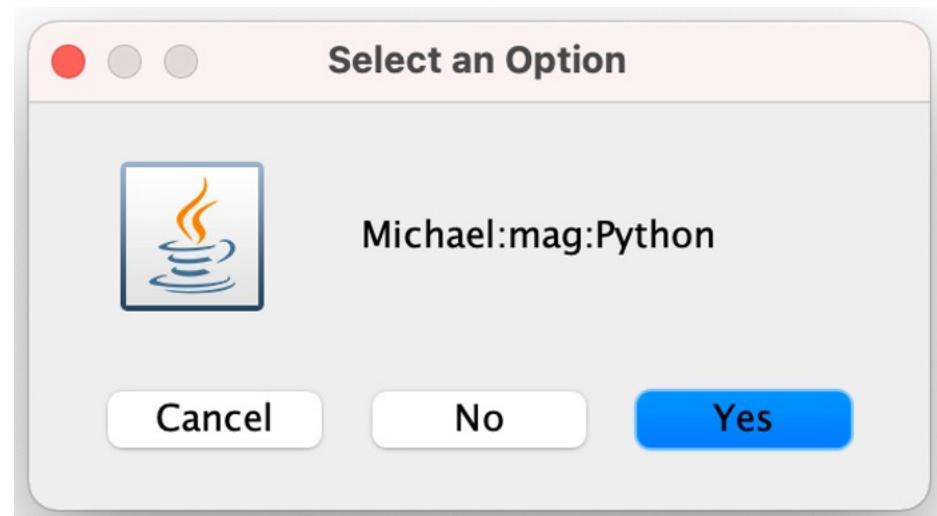


- ▼ target
 - > classes
 - > generated-sources
 - > maven-archiver
 - > maven-status
 - ⌚ checker-qual-3.12.0.jar
 - ⌚ error_prone_annotations-2.7.1.jar
 - ⌚ failureaccess-1.0.1.jar
 - ⌚ guava-31.0.1-jre.jar
 - ⌚ j2objc-annotations-1.3.jar
 - ⌚ jsr305-3.0.2.jar
 - ⌚ listenablefuture-9999.0-empty-to-avoid-conflict-with-guav
 - ⌚ PackagingDemoExamples-1.0.0-SNAPSHOT.jar

JPackage



IntelliJ IDEA 2022.1	13.04.22, 22:26	3.22 GB Programm
IntelliJ IDEA CE	02.04.22, 19:08	2.39 GB Programm
iPhoto	13.01.22, 23:36	1.7 GB Programm
JPackageDemoApp	Heute, 15:03	132.4 MB Programm
JPackageDemoAppV2	Heute, 15:07	135.6 MB Programm
Kalender	26.02.22, 08:05	15.6 MB Programm





DEMO & Hands on



Nashorn Java Script Engine

(seit Java 11 deprecated, mit Java 15 entfernt)



JShell-API als Abhilfe für dynamische Berechnungen



- Eigene Instanzen der JShell programmatisch erzeugen (`create()`)
- Code-Schnipsel automatisiert ausführen (`eval()`)
- Dynamische Berechnungen durchführen und somit als Ablösung für JavaScript-Engine nutzbar

```
final JShell jshell = JShell.create();
final List<SnippetEvent> result1 = jshell.eval("var name = \"Mike\";");

for (final SnippetEvent se : result1)
{
    System.out.println(se.snippet());
    System.out.println(se.value());

    // leider kein (direkter) Zugriff auf Wert, nur über varValue()
    jshell.variables().map(varSnippet -> "variable: '" + varSnippet.name() + "' / " +
                                         "type: '" + varSnippet.typeName() + "' / " +
                                         "value: " + jshell.varValue(varSnippet))
        .forEach(System.out::println);
}
```



DEMO



Übungen PART 5

<https://github.com/Michaeli71/Best-Of-Java-11-21>





Pattern Matching bei switch (PREVIEW in Java 17 ... später abgewandelt)



Pattern Matching bei switch / instanceof (Recap)



- Folgende generische Methode zum Formatieren von Werten kann man seit Java 16 mithilfe von Pattern Matching und instanceof einigermaßen lesbar schreiben:

```
static String formatterJdk16instanceof(Object obj) {  
    String formatted = "unknown";  
    if (obj instanceof Integer i) {  
        formatted = String.format("int %d", i);  
    } else if (obj instanceof Long l) {  
        formatted = String.format("long %d", l);  
    } else if (obj instanceof Double d) {  
        formatted = String.format("double %f", d);  
    } else if (obj instanceof String s) {  
        formatted = String.format("String %s", s);  
    }  
    return formatted;  
}
```

- ohne Java 16 müssten wir für jeden Typ eine Zeile mit einem Cast ergänzen!

Pattern Matching bei switch



- Diese Syntax-Neuerung ist mit Java 17 auch für switch (zunächst in Form von Preview Features) möglich:

```
static String formatterJdk17switch(Object obj) {  
    String formatted = switch (obj)  
    {  
        case Integer i -> String.format("int %d", i);  
        case Long l -> String.format("long %d", l);  
        case Double d -> String.format("double %f", d);  
        case String s -> String.format("String %s", s);  
        default -> "unknown";  
    };  
    return formatted;  
}
```

- Auf diese Weise lässt sich mit einer switch-Anweisung die Typzugehörigkeit eines Objekts prüfen und mit einem Wert befüllen.

Pattern Matching bei switch



- Bis Java 17 war es nicht möglich, in den cases eines switchs den Wert null zu behandeln, sondern dazu war folgende Spezialbehandlung nötig:

```
static void switchSpecialNullSupport(String str) {  
    if (str == null) {  
        System.out.println("special handling for null");  
        return;  
    }  
  
    switch (str) {  
        case "Java", "Python" -> System.out.println("cool language");  
        default -> System.out.println("everything else");  
    }  
}
```



Pattern Matching bei switch



- Mit Java 17 und aktivierten Preview Features können wir nun auch null-Werte angeben und das ganze Konstrukt vereinheitlichen:

```
static void switchSupportingNull(String str) {  
    switch (str) {  
        case null -> System.out.println("null is allowed in preview"); ←  
        case "Java", "Python" -> System.out.println("cool language");  
        default -> System.out.println("everything else");  
    }  
}
```

- Zum Nachvollziehen muss man Preview Features geeignet aktivieren, etwa wie folgt in der JShell:

```
jshell --enable-preview  
| Welcome to JShell -- Version 17.0.6  
| For an introduction type: /help intro
```

Pattern Matching bei switch



- Analog zu instanceof sind in den cases auch Abfragen wie die folgenden möglich:

```
static void processData(Object obj) {  
    switch (obj) {  
        case String str && str.startsWith("V1") -> System.out.println("Processing V1");  
        case String str && str.startsWith("V2") -> System.out.println("Processing V2");  
        case Integer i && i > 10 && i < 100 -> System.out.println("Processing ints");  
        default -> throw new IllegalArgumentException("invalid input");  
    }  
}
```



Ändert sich mit Java 18

- Die Angabe zusätzlicher Bedingungen nach der Typprüfung ist eine praktische syntaktische Neuerung, um Abfragen kompakt zu formulieren, wie oben die Versionsunterscheidung V1 und V2 oder die Wertebereiche des Integers.



PART 6: Neuerungen in Java 18 bis 21



Overview: JEPs in Java 18, 19, 20 und 21



- 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)

19

- 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)

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

The Java Version Almanac

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

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

Java 21

Info – <https://javaalmanac.io/>

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

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

New Features

JVM

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

Language

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

API

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



Sandbox

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

Java21.java ▶ Run

21+35-2513

```
1 import java.lang.reflect.ClassFileVersion;
2
3 public class Java21 {
4
5     public static void main(String[] args) {
6         var v = ClassFileVersion.latest();
7         System.out.printf("Hello Java bytecode version %s!", v.major());
8     }
9
10 }
```

No Support
for Preview
Features!

Java21.java ▶ Run

21+35-2513

```
Hello Java bytecode version 65!
```

Sandbox – <https://javaalmanac.io/>



```
public class Java21 {

    public static void main(String[] args) {
        multiMatch("Python");
        multiMatch(null);
        multiMatch(7);

        record Person(String name, int age) {}
        multiMatch(new Person("Michael", 52));
    }

    static void multiMatch(Object obj) {
        switch (obj) {
            case null -> System.out.println("null");
            case String str when str.length() > 5 -> System.out.println(str.toUpperCase());
            case String str -> System.out.println(str.toLowerCase());
            case Integer i -> System.out.println(i * i);
            default -> throw new IllegalArgumentException("Unsupported type " + obj.getClass());
        }
    }
}
```

Sandbox – <https://javaalmanac.io/>



Java21.java ▶ Run 21+35-2513

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

Java21.java ▶ Run 21+35-2513

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



Finale 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

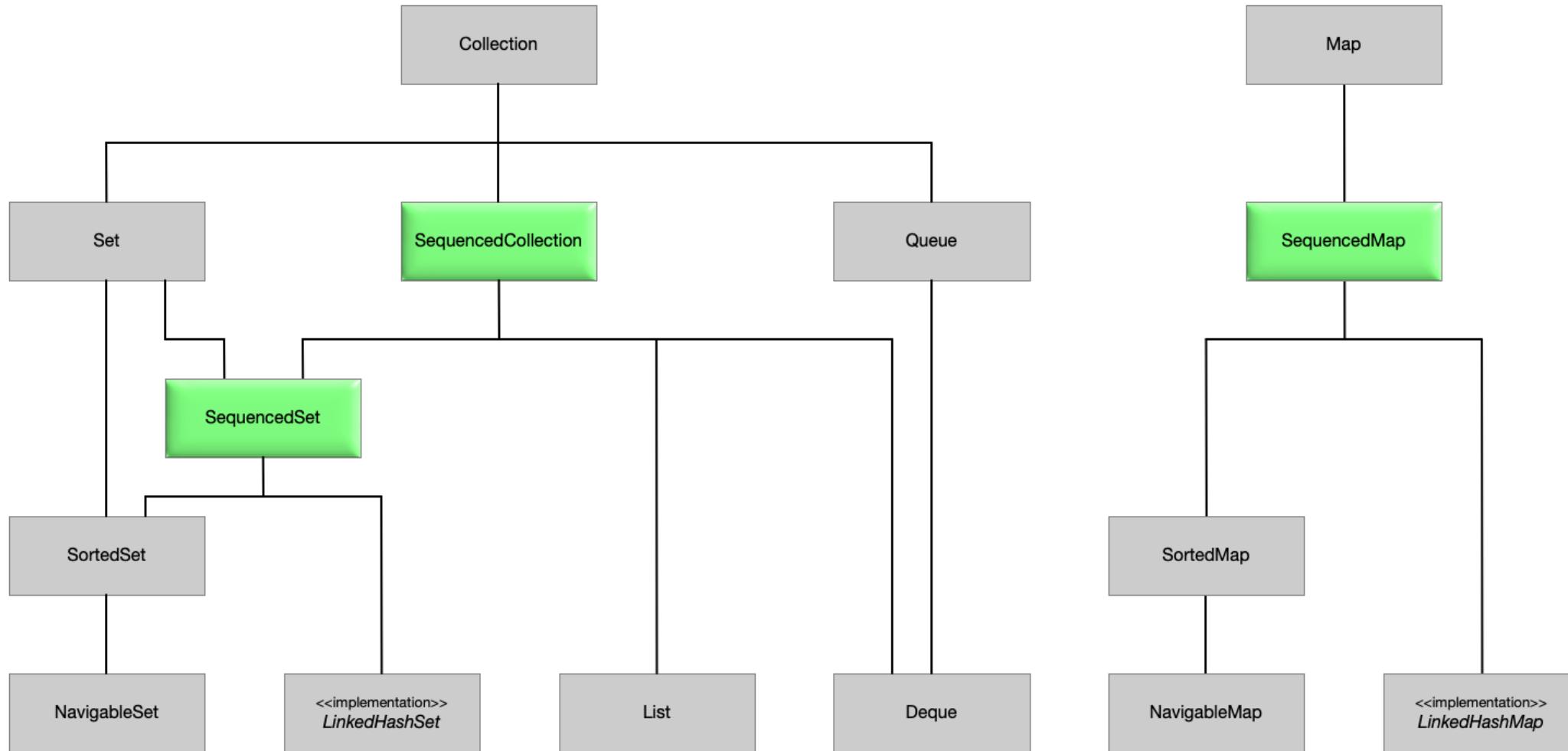


- In der Vergangenheit gab es mehrere Möglichkeiten, auf das erste oder letzte Element zuzugreifen

	First element	Last element
List	list.get(0)	list.get(list.size() - 1)
Deque	deque.getFirst()	deque.getLast()
SortedSet	sortedSet.first()	sortedSet.last()
LinkedHashSet	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 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 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 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 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



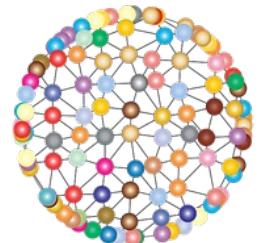
- Record Patterns können für Eleganz sorgen:

```
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!



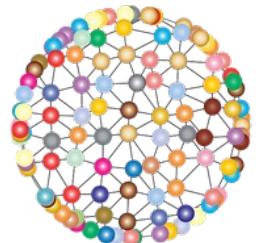
**Wie kann man das Ganze noch
eleganter gestalten?**



- **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 origin,  
                         City destination) {  
}
```



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

```
boolean checkAgeAndDestinationLanguageCorrectOld(Object obj) {  
    if (obj instanceof FlightReservation reservation) {  
        if (reservation.person() != null) {  
            Person person = reservation.person();  
            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(LocalDate.now(), birthday);  
                    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(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 origin,
        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 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 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' and 'Press 'F2' for focus'. The IDE interface shows syntax highlighting and code completion suggestions.

- 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 den positiven Wertebereich und
- danach den verbliebenen Rest:

```
Integer value = calcValue();

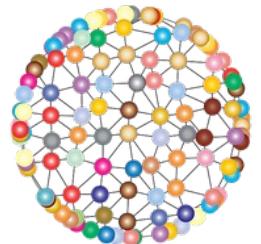
switch (value) {
    case -1, 0, 1 -> System.out.println("Handle special cases -1, 0 or 1");
    case Integer i when i >= 1 ->
        System.out.println("Handle positive integer cases i >= 1");
    case Integer i -> System.out.println("Handle all the remaining integers");
}
```

JEP 441: Pattern Matching bei switch: Dominanzprüfung



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

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



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

JEP 441: Pattern Matching bei switch – Spezialfälle I



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

JEP 441: Pattern Matching bei switch – Spezialfälle II



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

SPEZIALFALL IN DER ABFRAGE
=> DOMINANZ



- **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 dort nachgezogen (Backport).

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");
    }
}
```

JEP 441: Pattern Matching bei switch: Typinferenz mit Patterns



- 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>(String text, var count)
            when text.contains("Michael") ->
                System.out.println(text + " is " + count + " years old");
        case MyPair<String, Integer>(var text, Integer 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
        // Methoden 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

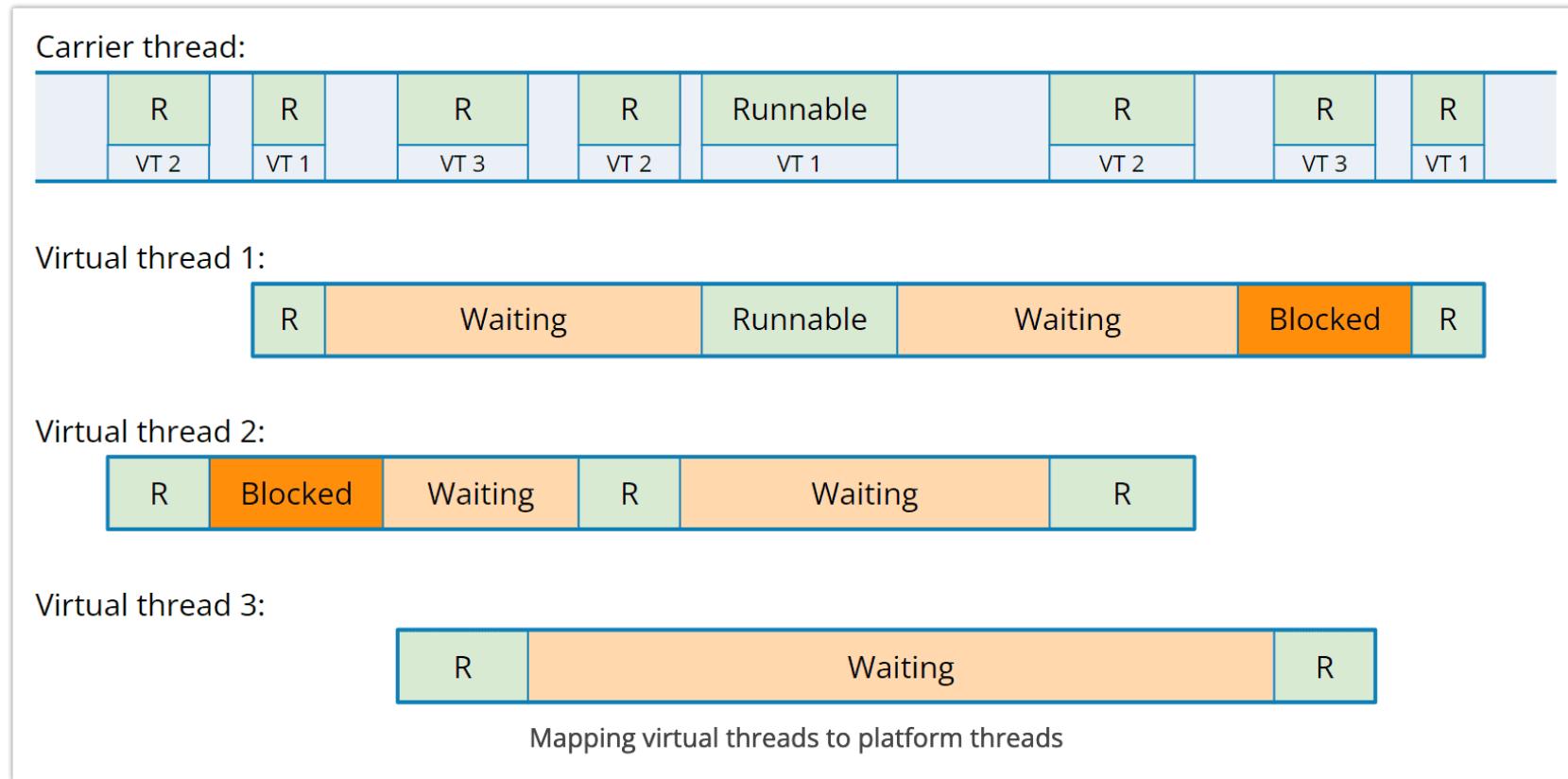
<https://openjdk.org/jeps/444>



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.





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

JEP 444: Virtual Threads



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

    try (var executor = Executors.newVirtualThreadPerTaskExecutor())
    {
        for (int i = 0; i < 10_000; i++)
        {
            final int pos = i;
            executor.submit(() -> {
                Thread.sleep(Duration.ofSeconds(1));
                return pos;
            });
        }
    }
    // executor.close() is called implicitly,
    // and waits until all tasks are completed
    System.out.println("End");
}
```



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

Concurrency Issues

Why is it bad to block?

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





- 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.

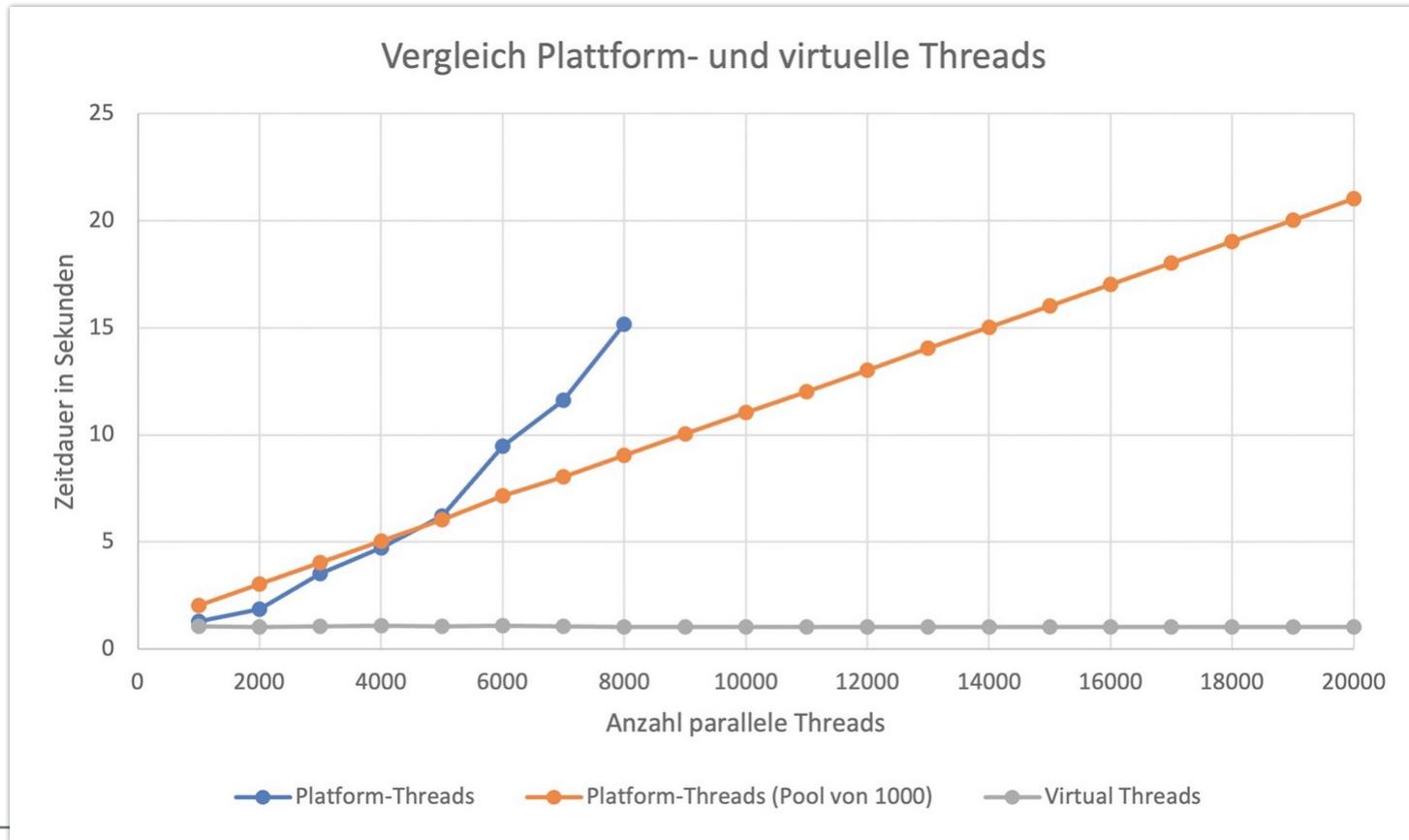
```
private static void submit1000Threads(ExecutorService executor) {  
    for (int i = 0; i < 1_000; i++) {  
        final int pos = i;  
        executor.submit(() -> {  
            Thread.sleep(Duration.ofSeconds(1));  
            return pos;  
        });  
    }  
}
```

- 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 eine Sekunde, um den Zugriff auf eine externe Schnittstelle zu simulieren. Am Ende wird die Gesamtzeit gemessen.**





DEMO & Hands on

PlatformThreads.java

VirtualThreads.java

VirtualThreadsPoolingExample.java



Übungen PART 6

<https://github.com/Michaeli71/Best-Of-Java-11-21>





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 430: String Templates (Preview)

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



Stringkonkatenation



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

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

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

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

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

- Alle haben ihre besonderen Stärken und vor allem Schwächen

String Interpolation



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

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

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

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

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

String Templates



- **Beispiel aus alt mach neu**

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

- =>

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

- **Beispiel**

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

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

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

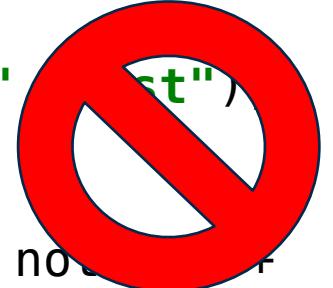
Michael Inden
Inden, Michael

String Templates



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

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

A large red circular 'no' symbol with a diagonal slash through it, indicating that the code snippets shown are incorrect or should be avoided.

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

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

String Templates und Text Blocks



- **Beispiel**

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

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

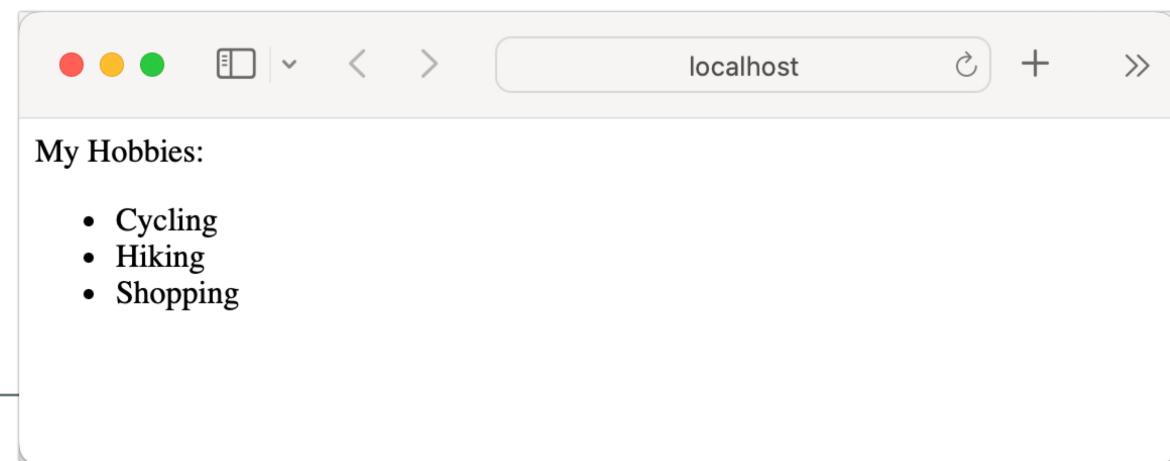
```
{
    "statusCode": 201,
    "msg": "CREATED"
}
```

String Templates und Text Blocks



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

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



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



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



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 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 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
    // unbenannte Muster, lokale Variablen, Ausnahmeparameter oder
    // Lambda-Parameter zu deklarieren
    //_.printStackTrace();
}
```

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



- 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 (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);
}
```

- **Das Gleiche gilt für case.**

JEP 443: Unnamed Patterns and Variables (Preview)



- **Unnamed pattern**

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

- =>

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

- **Das Gleiche gilt für case.**

instanceof _
instanceof _(int x, int y)



JEP 443: Unnamed Patterns and Variables (Preview)



- **ÜBRIGENS mit Java 21.0.0**

Fehler: Beim Laden der Klasse preview.syntax.JEP443_UnnamedVarsAndPatterns ist ein LinkageError aufgetreten

java.lang.ClassFormatError: Illegal field name "" in class
preview/syntax/JEP443_UnnamedVarsAndPatterns



- **DOWNLOAD Java 21.0.1 ... löst <https://bugs.openjdk.org/browse/JDK-8313323>**

x: 23 y: 11, sum: 34

x: 23 y: 11, sum: 34

x = 3

x = 3

Expected number, but was: 'E605'

...



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

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



JEP 445: Unnamed classes and instance main() method



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

```
package preview;

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

- Mit Python reduziert auf das Wesentliche:

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

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

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

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

JEP 445: Unnamed classes and instance main() method



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



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



Vereinfachung II: Unnamed class

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

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

- Die resultierende Unnamed Class hat (natürlich) keine Klassendefinition, kann aber Attribute und Methoden angeben. Außerdem wird sie automatisch in ein Unnamed Package eingeordnet.

- Ausführen mit (wenn der Dateiname SimplerHelloWorld.java lautet)

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

JEP 445: Unnamed classes and instance main() method



- **PAST**

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

- **PRESENT**

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

- **PRESENT OPTIMIZED**

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



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.



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



- 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 (aufrufender Thread und Teilaufgaben) im Thread-Dump etwas besser zu erkennen.**

JEP 453: Structured Concurrency



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

Caused by: java.lang.IllegalStateException: JUST TO PROVIDE EX

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

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

Caused by: java.lang.IllegalStateException: JUST TO PROVIDE EX

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



DEMO & Hands on

StructuredConcurrency.java

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



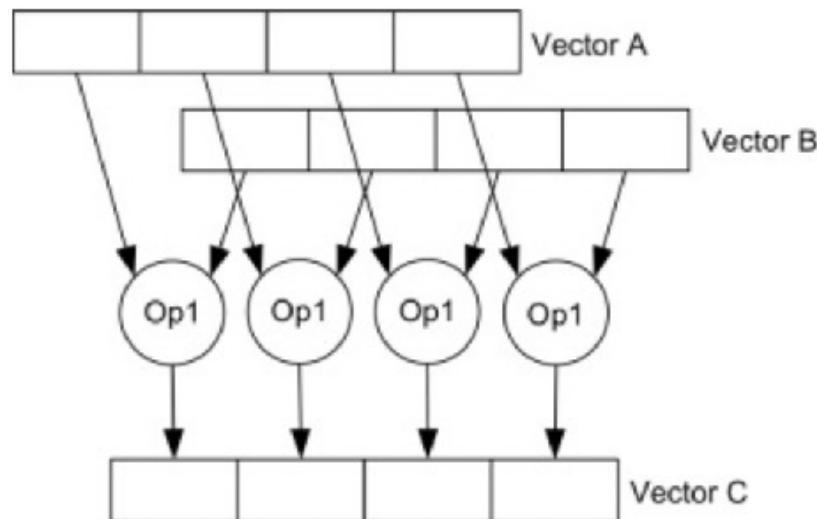
Incubator Features in Java 21

- JEP 448: Vector API (Sixth Incubator)
-



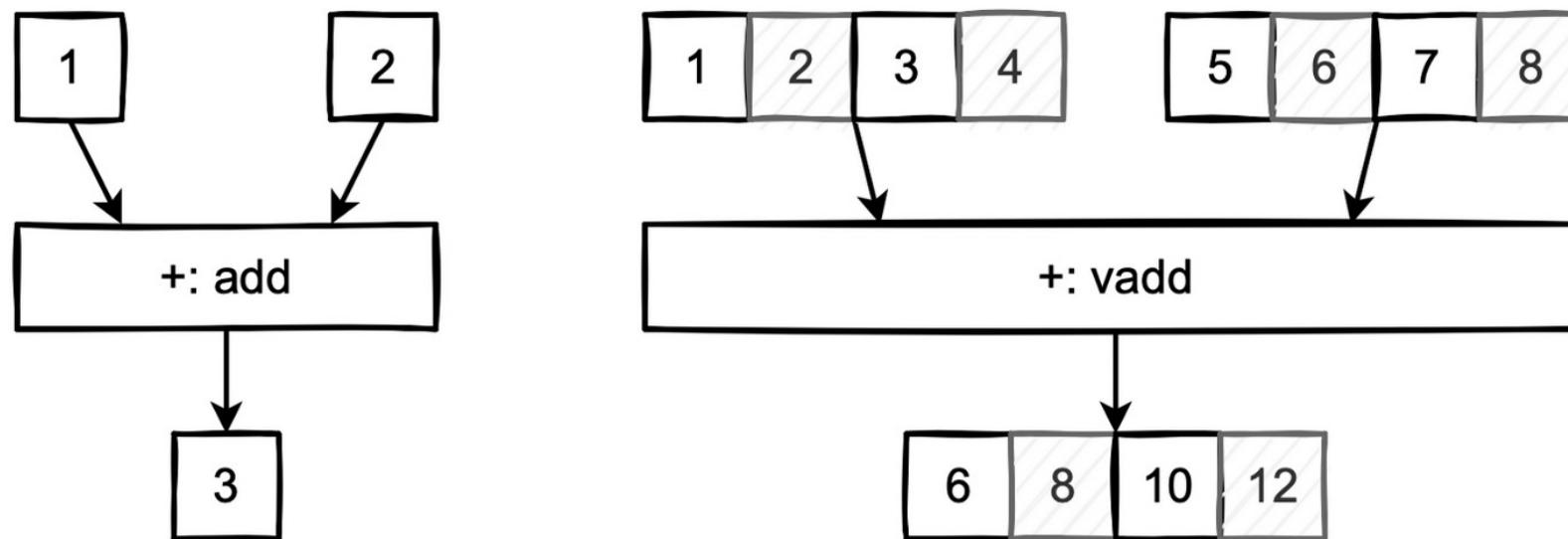
JEP 448: Vector API (Sixth Incubator in Java 21)

<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 ein Beispiel aus JEP 448 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;

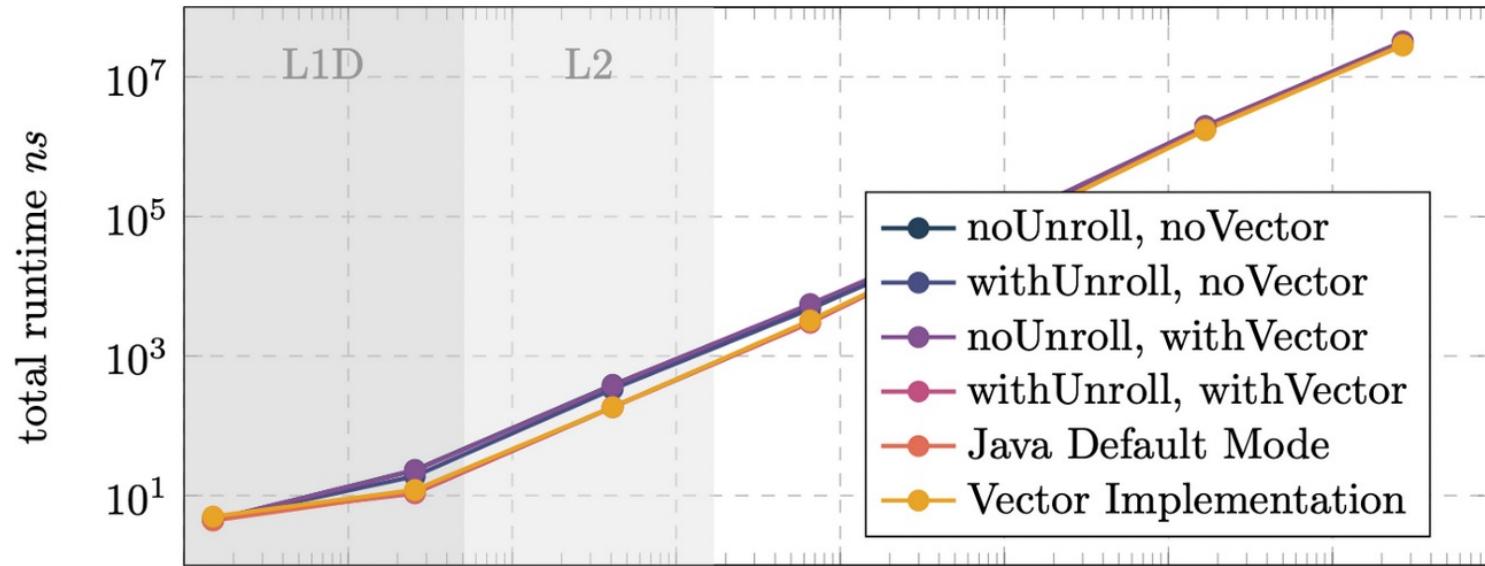
void vectorComputation(float[] a, float[] b, float[] c)
{
    int i = 0;
    int upperBound = SPECIES.loopBound(a.length);
    for (; i < upperBound; i += SPECIES.length())
    {
        var va = FloatVector.fromArray(SPECIES, a, i);
        var vb = FloatVector.fromArray(SPECIES, b, i);
        var 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 6

<https://github.com/Michaeli71/Best-Of-Java-11-21>





Fazit



On the positive side



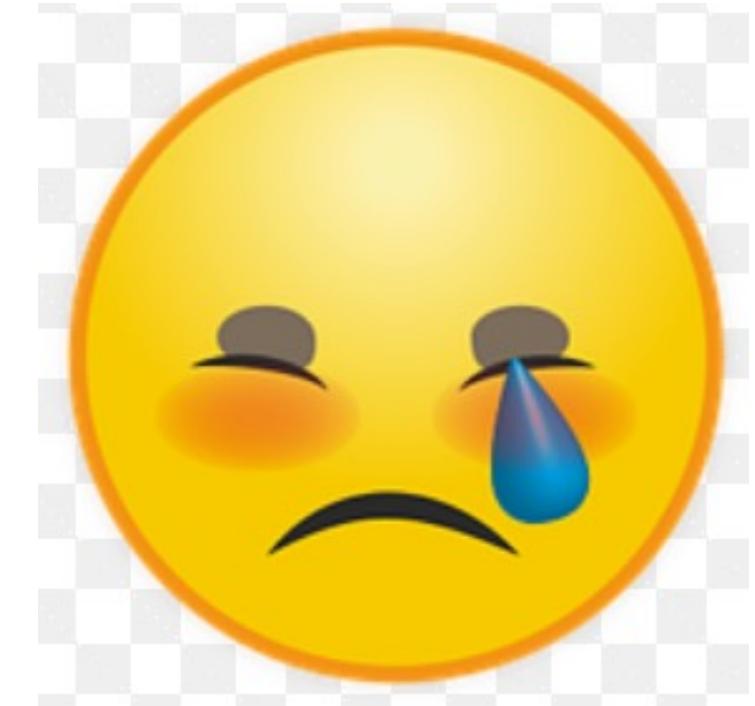
- **Stabile und zuverlässige 6-monatige Release-Zyklen und alle 2 Jahre LTS-Versionen**
- **Java wird einfacher und attraktiver**
- **Viele schöne Verbesserungen in Syntax und APIs wie switch, Records, Text Blocks**
- **Pattern Matching und Record Patterns final in Java 21**
- **JPackage und HTTP/2**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

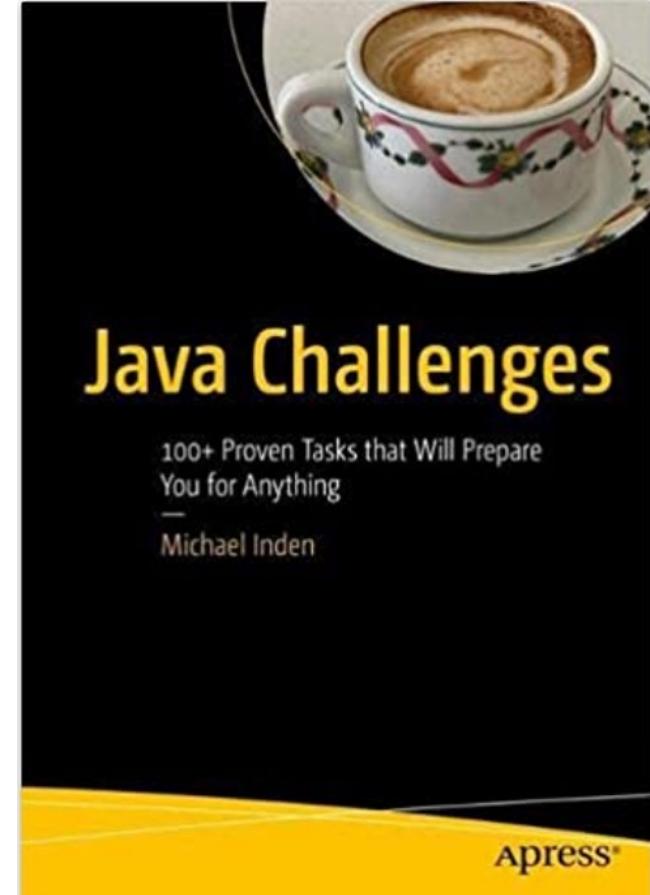


Negatives



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







Questions?



Thank You