



Best of Java 11 bis 18

<https://github.com/Michaeli71/Best-Of-Java-11-18>

Michael Inden

Freiberuflicher Consultant, Buchautor und Trainer

Speaker Intro



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

E-Mail: michael.inden@hotmail.com

Blog: <https://jaxenter.de/author/minden>

Kurse: **Bitte sprecht mich an!**

<https://github.com/Michaeli71/Best-Of-Java-11-18>





Agenda

Workshop Contents



- **Vorbemerkungen / Build Tools & IDEs**
 - **PART 1:** Syntax-Erweiterungen & Neuheiten und Änderungen in Java 9 bis 11
 - **PART 2:** Weitere Neuheiten und Änderungen in JDK 9 bis 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
-
- **Separat:** Modularisierung im Kurzüberblick

Workshop Contents



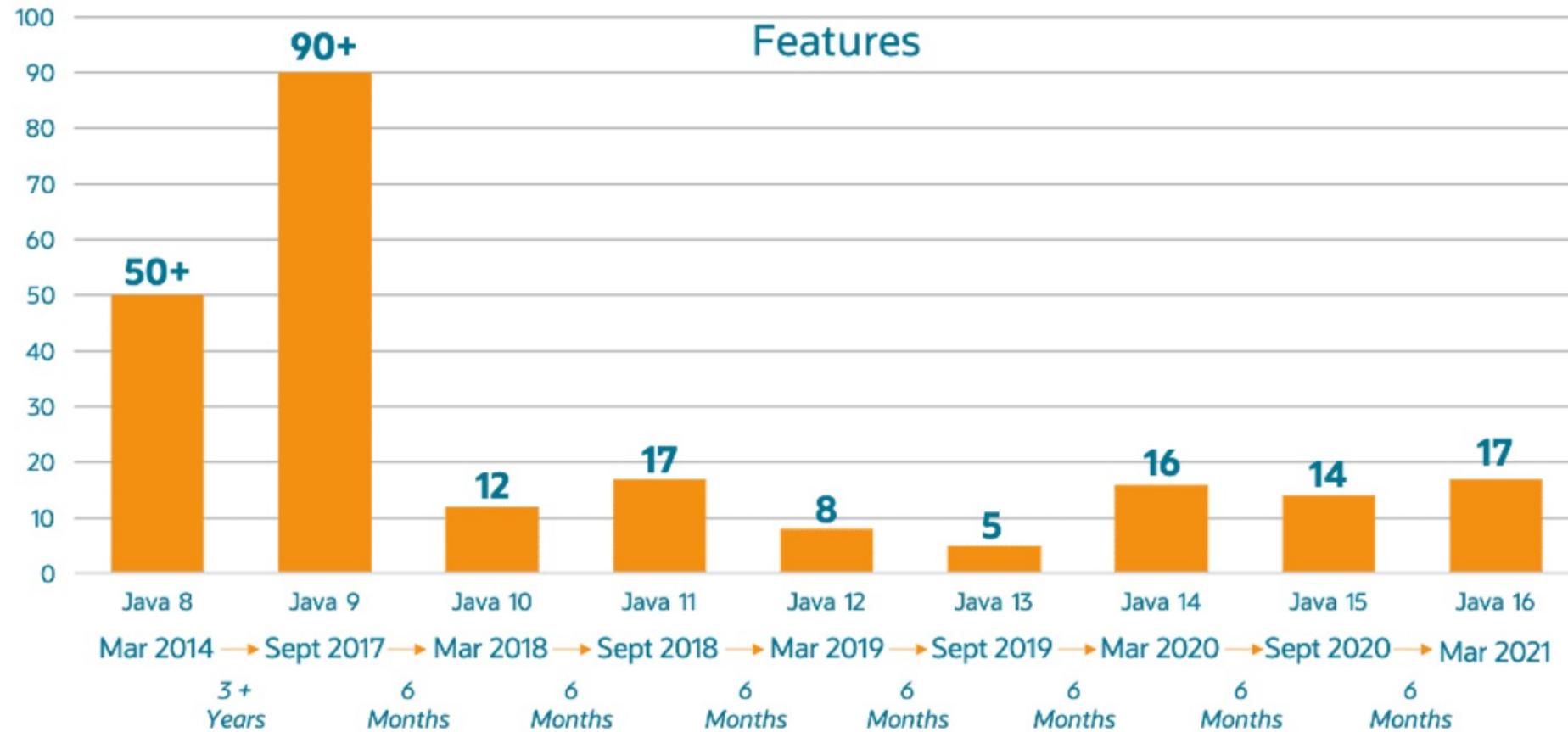
JDK	Release-Datum	Entwicklungszeit	LTS	Workshop
Oracle JDK 8	3 / 2014	-	Ja, <i>nun kommerziell</i>	-/-
Oracle JDK 9	9 / 2017	3,5 Jahre	-	Part 1, 2
Oracle JDK 10	3 / 2018	6 Monate	-	Part 1, 2
Oracle JDK 11	9 / 2018	6 Monate	Ja, <i>kommerziell</i>	Part 1, 2
Oracle JDK 12	3 / 2019	6 Monate	-	Part 3, 4, 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, <i>kommerziell, aber gelockerte Lizenz</i>	Part 3, 4, 5
Oracle JDK 18	3 / 2022	6 Monate	-	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
 - aktuelles LTS-Release ist Java 17 (September 2021)
 - **Seit Java 17 wieder ALLE 2 Jahre UND FOR FREE ☺**
- **andere Versionen sind "nur" Zwischenversionen**
 - erhalten nur 6 Monate Updates
 - Previews
 - Ideal um neue Features kennenzulernen und zum Experimentieren (vor allem privat)

Einordnung 6 monatiger Releasezyklus





Build-Tools und IDEs



IDE & Tool Support für Java 17



- Aktuelle IDEs & Tools grundsätzlich gut
- Eclipse: Version 2021-09 mit zusätzlichem Plugin
- IntelliJ: Version 2021.2.1
<https://blog.jetbrains.com/idea/2021/09/java-17-and-intellij-idea/>
- Maven: 3.8.5, Compiler-Plugin: 3.8.1
- Gradle: 7.4.1
- Aktivierung von Preview-Features nötig
 - In Dialogen
 - Im Build-Skript



Maven™

 **Gradle**

IDE & Tool Support Java 17



- Eclipse 2021-09 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.

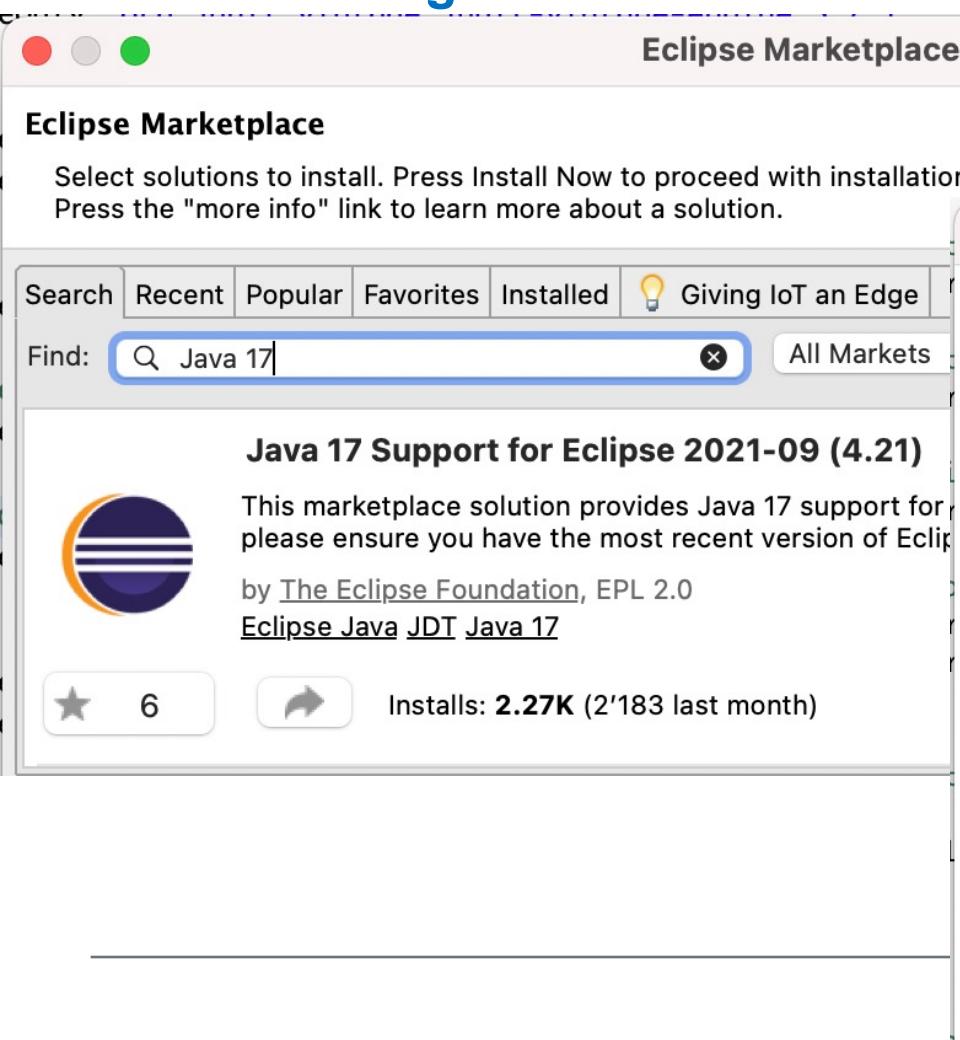
Search Recent Popular Favorites Installed Giving IoT an Edge

Find: All Markets

Java 17 Support for Eclipse 2021-09 (4.21)

This marketplace solution provides Java 17 support for please ensure you have the most recent version of Eclipse by [The Eclipse Foundation](#), EPL 2.0 [Eclipse Java JDT Java 17](#)

6 Installs: 2.27K (2'183 last month)



Eclipse Marketplace

Properties for Best_of_Java_9_to_17

Java Compiler

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

JDK Compliance

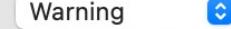
Use compliance from execution environment on the 'Java Build Path' 

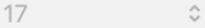
Compiler compliance level: 

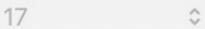
Use '--release' option

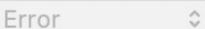
Use default compliance settings

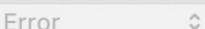
Enable preview features for Java 17 

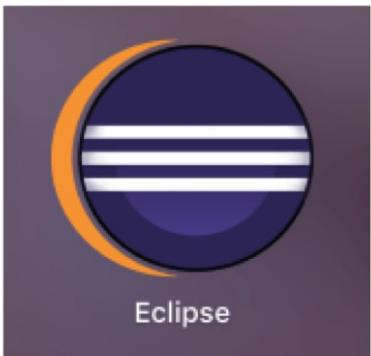
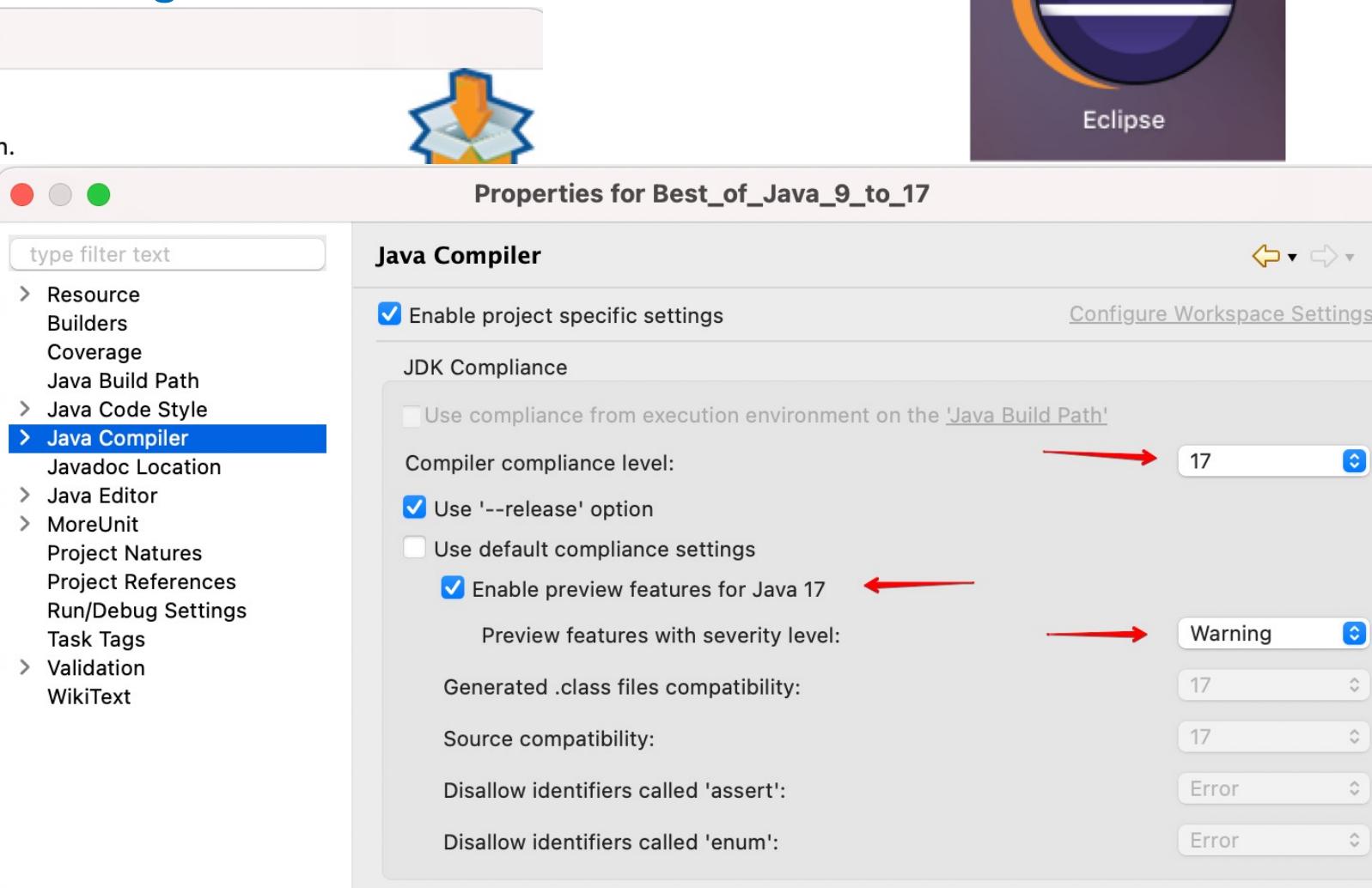
Preview features with severity level: 

Generated .class files compatibility: 

Source compatibility: 

Disallow identifiers called 'assert': 

Disallow identifiers called 'enum': 



IDE & Tool Support



- Aktivierung von Preview-Features nötig

Project Structure

Project name: Java17Examples

Project SDK:
This SDK is default for all project modules.
A module specific SDK can be configured for each of the modules as required.
17 version 17 Edit

Project language level:
This language level is default for all project modules.
A module specific language level can be configured for each of the modules as required.
SDK default (17 - Sealed types, always-strict floating-point semantics)

Project compiler output:
This path is used to store all project compilation results.
A directory corresponding to each module is created under this path.
This directory contains two subdirectories: Production and Test for production code and test sources, respectively.
A module specific compiler output path can be configured for each of the modules as required.
/Users/michaeli/Java17Examples/out

Project language level:

This language level is default for all project modules.

A module specific language level can be configured for each of the modules as required.





- Aktivierung von Preview-Features nötig

```
sourceCompatibility=17  
targetCompatibility=17
```

```
// Aktivierung von Switch Expressions Preview  
tasks.withType(JavaCompile) {  
    options.compilerArgs += ["--enable-preview"]  
}
```



- Java 18

```
sourceCompatibility=18  
targetCompatibility=18
```

```
// Aktivierung von Switch Expressions Preview  
tasks.withType(JavaCompile) {  
    options.compilerArgs += ["--enable-preview"]  
}
```



- Aktivierung von Preview-Features nötig

```
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.8.1</version>
    <configuration>
      <source>17</source>
      <target>17</target>
      <!-- Wichtig für Java Syntax-Neuerungen -->
      <compilerArgs>--enable-preview</compilerArgs>
    </configuration>
  </plugin>
</plugins>
```

```
          <plugins>
            <plugin>
              <groupId>org.apache.maven.plugins</groupId>
              <artifactId>maven-compiler-plugin</artifactId>
              <version>3.8.1</version>
              <configuration>
                <source>18</source>
                <target>18</target>
                <!-- Wichtig für Java Syntax-Neuerungen -->
                <compilerArgs>--enable-preview</compilerArgs>
              </configuration>
            </plugin>
          </plugins>
```





PART 1: **Syntax-Erweiterungen und API- Änderungen in Java 9 bis 11**



Syntax-Erweiterungen



Anonyme innere Klassen und der Diamond Operator



```
final Comparator<String> byLength = new Comparator<String>()
{
    ...
};
```

```
final Comparator<String> byLength = new Comparator<>()
{
    ...
};
```



@Deprecated-Annotation



- **@Deprecated** dient zum Markieren von obsolem 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

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

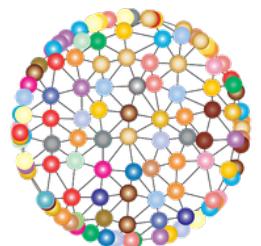
```
// var => Set<Map.Entry<String, Long>>
var entries = personAgeMapping.entrySet();
```

```
// 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».

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

- Wollte man diesen Fehler vermeiden, so müsste man folgenden Cast einfügen:
- **Insgesamt sieht man, dass var für Lambda-Ausdrücke eher ungeeignet ist.**

Local Variable Type Inference Fallstrick

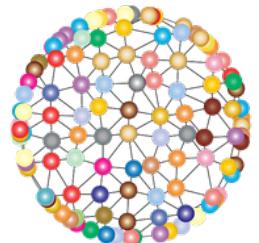


- Manchmal ist man versucht, ohne viel Nachdenken die Typangabe auf der linken Seite direkt mit var zu ersetzen:

```
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);
```



**Kompiliert 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);
```

Local Variable Type Inference Beschränkungen



- Rekapitulieren wir kurz: var ist für **lokale Variablen** gedacht, die direkt initialisiert werden.
var zur Deklaration von Attributen, Parametern oder Rückgabetypen wünschenswert?
=> Das geht jedoch nicht, weil hier der Typ vom Compiler nicht eindeutig ermittelt werden kann.

- **Weitere Dinge, die zu Kompilierfehlern führen:**

```
var justDeclaration;      // keine Wertangabe / Definition
var numbers = {0, 1, 2}; // fehlende Typangabe
var appendSpace = str -> str + " "; // Typ unklar
```

- Beim Einsatz von var wird immer der **exakte** Typ verwendet wird und nicht ein Basistyp, wie 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 in Java 9 bis 11

- Stream-API
- Optional<T>
- Collection Factory Methods
- Strings (JDK 11)



Stream API





Das umfangreiche **Stream-API** war eine **der wesentlichen Neuerungen in Java 8**

`takeWhile(...)`

`dropWhile(...)`

`ofNullable(...)`

`iterate(..., ..., ...)`



`takeWhile(...)`

`dropWhile(...)`

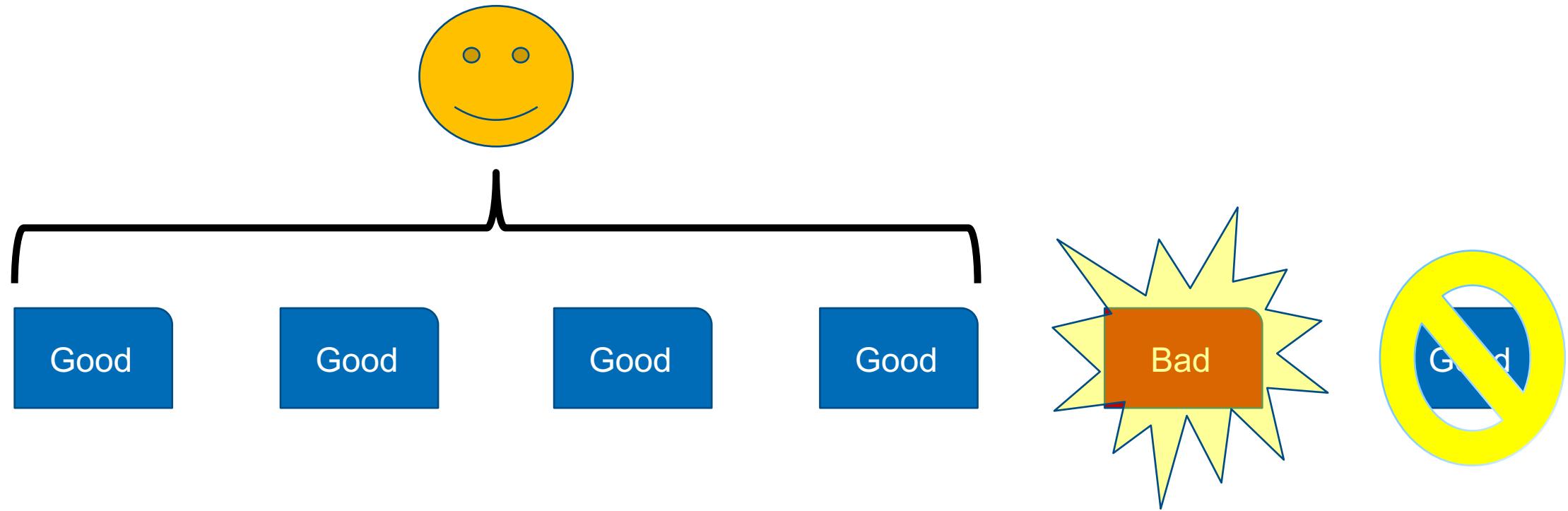
`takeWhile(Predicate<T>)` – Verarbeitet Elemente des Streams, solange die übergebene Bedingung erfüllt ist.

`ofNullable(...)`

`iterate(..., ..., ...)`



Stream – Product Scenario





Stream – Filter

JDK8

```
public class StrictCustomer {  
  
    public static void main(String[] args) {  
  
        Stream<String> deliveredProductsQuality = Stream.of("1. Good",  
                "2. Good",  
                "3. Good",  
                "4. Good",  
                "5. Bad",  
                "6. Good");  
  
        deliveredProductsQuality.  
            filter(productQuality -> productQuality.contains("Good")).  
            forEach(System.out::println);  
    }  
}
```

1. Good
2. Good
3. Good
4. Good
6. Good|



Stream – Filter

JDK8

```
public class StrictCustomer {  
  
    public static void main(String[] args) {  
  
        Stream<String> deliveredProductsQuality = Stream.of("1. Good",  
                                                "2. Good",  
                                                "3. Good",  
                                                "4. Good",  
                                                "5. Bad",  
                                                "6. Good");  
  
        deliveredProductsQuality.  
            filter(productQuality -> productQuality.contains("Good")).  
            forEach(System.out::println);  
    }  
}
```

1. Good
2. Good
3. Good
4. Good
5. Good



So ? What do we do ?

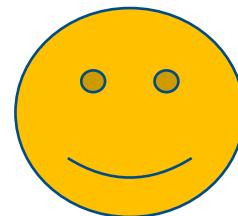


Google

Google-Suche

Auf gut Glück!

Google angeboten in: English Français Italiano Rumantsch



takeWhile(...)



```
public class StrictCustomer {  
  
    public static void main(String[] args) {  
  
        Stream<String> deliveredProductsQuality = Stream.of("1. Good",  
                                                "2. Good",  
                                                "3. Good",  
                                                "4. Good",  
                                                "5. Bad",  
                                                "6. Good");  
  
        deliveredProductsQuality.  
            arrow [ ] takeWhile(productQuality -> productQuality.contains("Good")).  
            forEach(System.out::println);  
    }  
}
```

- 1. Good
- 2. Good
- 3. Good
- 4. Good

Stream API in JDK 9



`takeWhile(...)`

`dropWhile(...)`

`dropWhile(Predicate<T>)` – Überspringt Elemente des Streams, solange die übergebene Bedingung erfüllt ist

`ofNullable(...)`

`iterate(..., ..., ...)`

dropWhile(...)



```
public class StrictCustomer {  
  
    public static void main(String[] args) {  
  
        Stream<String> deliveredProductsQuality = Stream.of("1. Bad",  
                                                "2. Bad",  
                                                "3. Bad",  
                                                "4. Good",  
                                                "5. Good",  
                                                "6. Good");  
  
        deliveredProductsQuality.  
            dropWhile(productQuality -> productQuality.contains("Bad")).  
            forEach(System.out::println);  
  
    }  
  
}
```

dropWhile(...)



```
public class StrictCustomer {  
  
    public static void main(String[] args) {  
  
        Stream<String> deliveredProductsQuality = Stream.of("1. Bad",  
                                                "2. Bad",  
                                                "3. Bad",  
                                                "4. Good",  
                                                "5. Good",  
                                                "6. Good");  
  
        deliveredProductsQuality.  
            dropWhile(productQuality -> productQuality.contains("Bad")).  
            forEach(System.out::println);  
    }  
}
```

4. Good
5. Good
6. Good

Kombination der beiden Methoden des Stream APIs



- Kombination der beiden Methoden zur Extraktion von Daten:

```
public static void main(String[] args)
{
    Stream<String> words = Stream.of("ab", "BLA", "Xy", "<START>",
                                      "WELCOME", "T0", "JAX", "Online",
                                      "<END>", "x", "y", "z");

    Stream<String> extracted = words.dropWhile(str -> !str.startsWith("<START>"))
                                    .skip(1)
                                    .takeWhile(str -> !str.startsWith("<END>"));

    extracted.forEach(System.out::println);
}
```

```
WELCOME
TO
JAX
Online
```



Stream API – Spezielle Kollektoren





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 aufzubereiten. 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.



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.

⇒ Schauen wir zunächst auf einfache Beispiele ...



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

Beispiel zum Einstieg:

```
final Set<String> result1 = programming1.filter(name -> name.contains("Java")).  
                                collect(toSet());
```

Mit Filtering Collector:

```
final Set<String> result2 = programming2.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



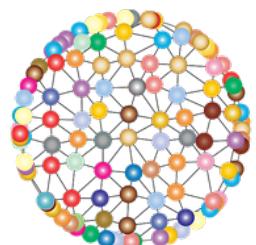
Beispiel zum Einstieg:

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}
```

Stream API Collectors in JDK 9



Die Neuerung `Collectors.flatMapping()` mit Analogie `Collectors.mapping()`

Alle Hobbies als (eine) Menge:

```
final Stream<Set<String>> lotsOfHobbies = Stream.of(Set.of("Java", "Movies"),
                                                       Set.of("Skating", "Tennis"),
                                                       Set.of("Karate", "Movies"),
                                                       Set.of("Music", "Movies"));
final Set<Set<String>> result =
    lotsOfHobbies.collect(mapping(entry -> entry, toSet()));
System.out.println("Hobbies: " + result);
```

Als Ergebnis:

Hobbies: [[Skating, Tennis], [Music, Movies], [Karate, Movies], [Java, Movies]]



Alle Hobbies als eine Menge:

```
final Stream<Set<String>> lotsOfHobbies = Stream.of(Set.of("Java", "Movies"),
                                                       Set.of("Skating", "Tennis"),
                                                       Set.of("Karate", "Movies"),
                                                       Set.of("Music", "Movies"));

final Set<String> result = lotsOfHobbies.collect(
    flatMapping(value -> value.stream(), toSet()));
System.out.println("Hobbies: " + result);
```

Als Ergebnis:

Hobbies: [Java, Tennis, Karate, Music, Movies, Skating]

Erste Variante liefert ungewünschte Schachtelung!

Zweite Variante weniger intuitiv wegen stream(). Erneut Vorteil erst mit groupingBy()



Geänderte Anforderung und praxisrelevanteres Beispiel:

Aus einer Menge von Personen mit Hobbies, all diejenigen gleichen Vornamens gruppieren und eine Sammlung der Hobbies erstellen:

```
final Stream<Map.Entry<String, Set<String>>> personsToHobbies =  
    Stream.of(Map.entry("Peter", Set.of("Groovy", "Movies")),  
             Map.entry("Peter", Set.of("Java", "Skating")),  
             Map.entry("Mike", Set.of("Python", "Sophie")),  
             Map.entry("Mike", Set.of("Java")));  
  
final Map<String, Set<String>> collected = personsToHobbies.collect(  
    groupingBy(entry -> entry.getKey(),  
               flatMapping(entry -> entry.getValue().stream(),  
                           toSet())));  
  
System.out.println(collected);
```

Als Ergebnis:

```
{Mike=[Java, Sophie, Python], Peter=[Java, Movies, Groovy, Skating]}
```



Optional<T>



Die Klasse Optional<T>



- Mit Java 8 eingeführt und erleichtert die Behandlung und Modellierung optionaler Werte.
- Bereits gutes API:



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, Optional<U>>) <U> : Optional<U>`
- `get() : T`
- `△ hashCode() : int`
- `ifPresent(Consumer<? super T>) : void`
- `isPresent() : boolean`
- `map(Function<? super T, ? extends U>) <U> : Optional<U>`
- `orElse(T) : T`
- `orElseGet(Supplier<? extends T>) : T`
- `orElseThrow(Supplier<? extends X>) <X extends Throwable> : T`
- `△ toString() : String`

Optional<T> in JDK8 / JDK9



▼ C F Optional<T>

- S empty() <T> : void
- 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
- isPresent() : boolean
- map(Function<? super T, ? extends U>) <U> : Optional<U>
- orElse(T) : T
- orElseGet(Supplier<? extends T>) : T
- orElseThrow(Supplier<? extends X>) <X extends Throwable> : T
- ▲ toString() : String

▼ 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

Die Klasse Optional<T>



- **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.

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



why ifPresentOrElse(...) ?



JDK8

```
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");  
    }  
  
}
```

why 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



why 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

or(...)



-

JDK 9

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



```
optBalane.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

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



Die Klasse Optional<T> (Recap)



- Mit Java 8 eingeführt und erleichtert die Behandlung und Modellierung optionaler Werte.
- Mit Java 9 nochmals um drei wertvolle Methoden erweitert.

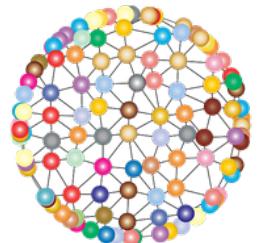
▼ **G F** `Optional<T>`

- `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`





**Was ist potenziell das
Problem an der Methode
get()?**



Die Klasse Optional<T>



- Die Methode `get()` zum Zugriff auf den Wert eines `Optional<T>` sieht **zu** harmlos aus!
- Mitunter wird `get()` ohne vorherige Prüfung auf Existenz eines Werts mit `isPresent()`
- Das führt dann aber bei einem nicht vorhandenen Wert zu einer `NoSuchElementException`.
- **Normalerweise erwartet man von einer `get()`-Methode allerdings nicht unbedingt, dass diese eine Exception auslöst.**
- **NEU IN JDK 10:**
- `orElseThrow()` als Alternative zu `get()`, um diesen Sachverhalt im API direkt auszudrücken

Die Klasse Optional<T>



- Experimentieren wir ein wenig in der JShell

```
jshell> Optional<String> optValue = Optional.of("ABC");
optValue ==> Optional[ABC]
```

```
jshell> String value = optValue.orElseThrow();
value ==> "ABC"
```

```
jshell> Optional<String> empty = Optional.empty();
empty ==> Optional.empty
```

```
jshell> empty.orElseThrow();
| java.util.NoSuchElementException thrown: No value present
|     at Optional.orElseThrow (Optional.j
```

Die Klasse `java.util.Optional<T>`



- Bis (*einschliesslich*) Java 10 immer wieder sinnvoll ergänzt.
- In Java 11 noch eine weitere Methode, nämlich `isEmpty()`
- API damit analog zu Collections und String bei Prüfungen, aber inkonsistent
- Vermeidet die Negation von `isPresent()`

```
final Optional<String> optEmpty = Optional.empty();  
  
if (!optEmpty.isPresent())  
    System.out.println("check for empty JDK 10 style");  
  
if (optEmpty.isEmpty())  
    System.out.println("check for empty JDK 11 style");
```



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
```

Collection Literals



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

**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)
```



#JavaYoungPups

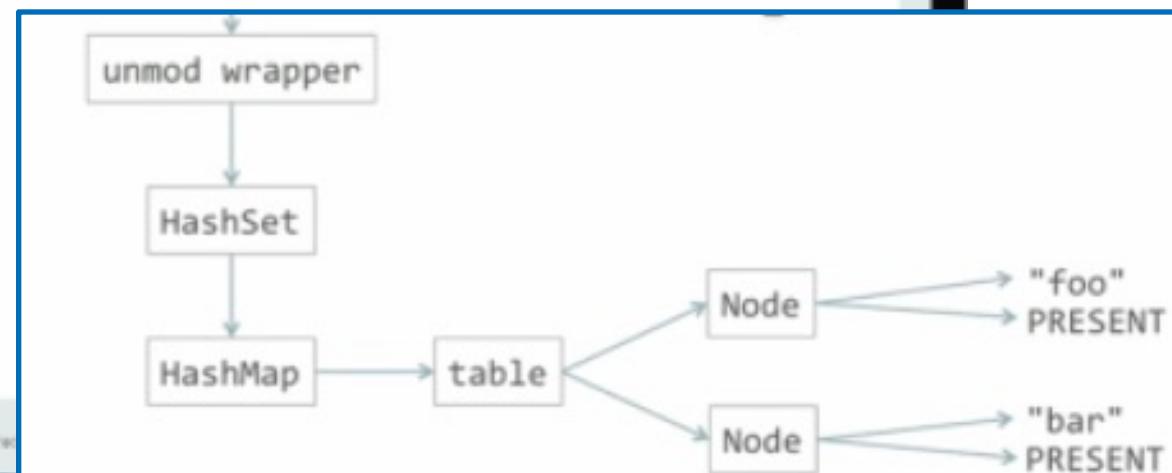
Space Efficiency

- Consider an unmodifiable set containing two strings

```
Set<String> set = new HashSet<>(3); // 3 is the number of buckets
set.add("foo");
set.add("bar");
set = Collections.unmodifiableSet(set);
```

- How much space does this take? Count objects.

- 1 unmodifiable wrapper
- 1 HashSet
- 1 HashMap
- 1 Object[] table of length 3
- 2 Node objects, one for each element





#JavaYoungPups

Space Efficiency

- Proposed field-based set implementation
`Set<String> set = Set.of("foo", "bar");`
- One object, two fields
 - 20 bytes, compared to 152 bytes for conventional structure
- Efficiency gains
 - lower fixed cost: fewer objects created for a collection of any size
 - lower variable cost: fewer bytes overhead per collection element





Erweiterung in der Klasse String in Java 11



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 mühsam oder mithilfe von externen Bibliotheken möglich, zu prüfen, ob diese nur Whitespaces 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));
}

=>

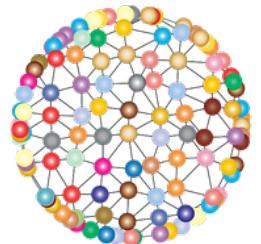
*****  
-*_- -*_- -*_- -*_- -*_- -*_-
```

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



```
"ERROR".repeat(-1);
```

```
"ERROR".repeat(Integer.MAX_VALUE);
```



Was passiert?

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



```
"ERROR".repeat(-1);
```

Exception in thread "main" `java.lang.IllegalArgumentException`: count is negative: -1
at `java.base/java.lang.String.repeat(String.java:3149)`
at `Java11Examples/snippet.Snippet.main(Snippet.java:16)`

```
"ERROR".repeat(Integer.MAX_VALUE);
```

Exception in thread "main" `java.lang.OutOfMemoryError`: Required length exceeds implementation limit
at `java.base/java.lang.String.repeat(String.java:4425)`
at `api.StringNewMethodsExample.main(StringNewMethodsExample.java:16)`

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



```
"ERROR".repeat(Integer.MAX_VALUE);
```

Exception in thread "main" java.lang.OutOfMemoryError: Required length exceeds implementation limit

at java.base/java.lang.String.repeat([String.java:4425](#))
at api.StringNewMethodsExample.main([StringNewMethodsExample.java:16](#))

```
if (Integer.MAX_VALUE / count < len) {  
    throw new OutOfMemoryError("Required length exceeds implementation limit");  
}
```

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-11-18>



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

- Date API
- InputStream und Reader
- Files
- Verschiedenes
- HTTP/2
- Direct Compilation
- JShell



Date API in Java 9



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

Klasse Duration



- **divideBy()** – teilen durch die übergebene Einheit
- **truncateTo()** – abschneiden auf übergebene Einheit

```
public static void main(String[] args)
{
    final Duration tenDaysSevenHoursThirdMinutes = Duration.ofDays(10).
                                                plusHours(7).
                                                plusMinutes(30);

    // JDK 9: divideBy(Duration)
    final long wholeDays = tenDaysSevenHoursThirdMinutes.dividedBy(Duration.ofDays(1));
    final long wholeHours = tenDaysSevenHoursThirdMinutes.dividedBy(Duration.ofHours(1));
    final long wholeMinutes = tenDaysSevenHoursThirdMinutes.dividedBy(Duration.ofMinutes(15));
    System.out.println("wholeDays: " + wholeDays);
    System.out.println("wholeHours: " + wholeHours);
    System.out.println("whole15Minutes: " + wholeMinutes);

    // JDK 9: truncatedTo(TemporalUnit)
    System.out.println("truncatedTo(DAYS): " + tenDaysSevenHoursThirdMinutes.truncatedTo(ChronoUnit.DAYS));
    System.out.println("truncatedTo(HOURS): " + tenDaysSevenHoursThirdMinutes.truncatedTo(ChronoUnit.HOURS));
    System.out.println("truncatedTo(MINUTES): " + tenDaysSevenHoursThirdMinutes.truncatedTo(ChronoUnit.MINUTES));
}
```

Klasse Duration



```
public static void main(String[] args)
{
    final Duration tenDaysSevenHoursThirdMinutes = Duration.ofDays(10).
                                                plusHours(7).
                                                plusMinutes(30);

    // JDK 9: divideBy(Duration)
    final long wholeDays = tenDaysSevenHoursThirdMinutes.dividedBy(Duration.ofDays(1));
    final long wholeHours = tenDaysSevenHoursThirdMinutes.dividedBy(Duration.ofHours(1));
    final long wholeMinutes = tenDaysSevenHoursThirdMinutes.dividedBy(Duration.ofMinutes(15));
    System.out.println("wholeDays: " + wholeDays);
    System.out.println("wholeHours: " + wholeHours);
    System.out.println("whole15Minutes: " + wholeMinutes);

    // JDK 9: truncatedTo(TemporalUnit)
    System.out.println("truncatedTo(DAYS): " + tenDaysSevenHoursThirdMinutes.truncatedTo(ChronoUnit.DAYS));
    System.out.println("truncatedTo(HOURS): " + tenDaysSevenHoursThirdMinutes.truncatedTo(ChronoUnit.HOURS));
    System.out.println("truncatedTo(MINUTES): " + tenDaysSevenHoursThirdMinutes.truncatedTo(ChronoUnit.MINUTES));
}
```

```
wholeDays:          10
wholeHours:         247
howMany15Minutes:  990
truncatedTo(DAYS): PT240H
truncatedTo(HOURS): PT247H
truncatedTo(MINUTES): PT247H30M
```



java.io.InputStream / Reader



Sonstiges – Die Klasse InputStream



- Die Klasse `InputStream` wurde um einige praktische Methoden für gebräuchliche Anwendungsfälle erweitert:
 - `public long transferTo(OutputStream out) throws IOException`
 - `public byte[] readAllBytes() throws IOException`
 - `public int readNBytes(byte[] b, int off, int len) throws IOException`

Klasse `java.io.Reader` (JDK 10)



- `long transferTo(Writer)`

Es werden alle Zeichen aus dem Reader in den übergebenen Writer übertragen – diese Funktionalität existiert analog in der Klasse `InputStream` bereits seit Java 9.

```
var sr = new StringReader("Hello");
var sw = new StringWriter();

sr.transferTo(sw);

System.out.println("Sw: " + sw.toString());
```

=>

Sw: Hello



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
```

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

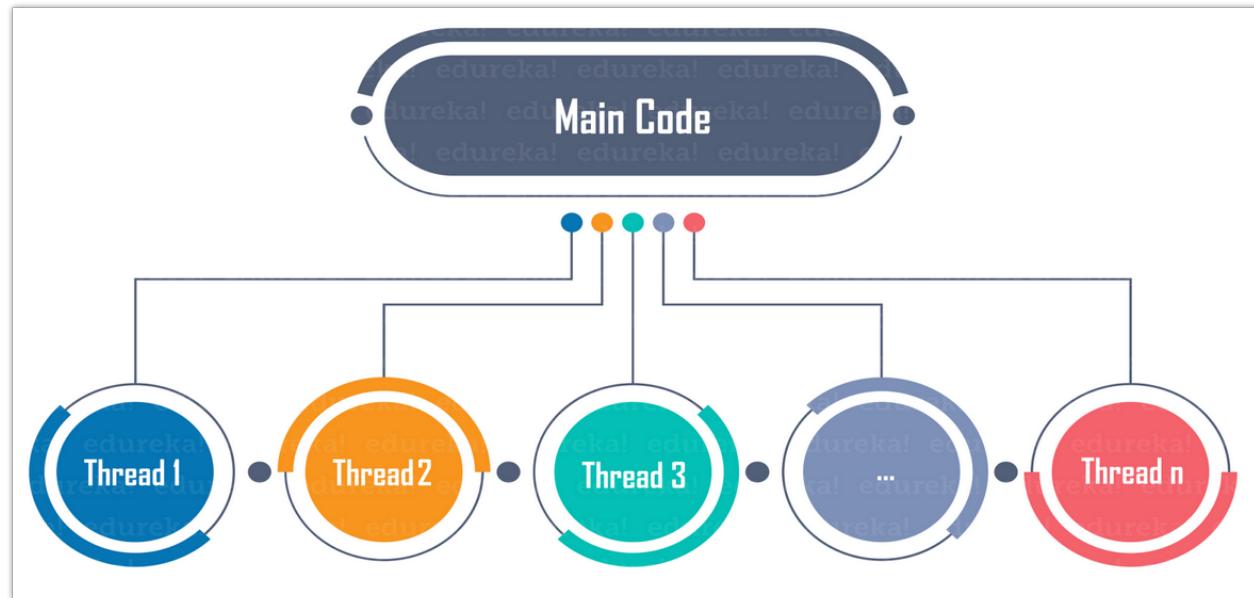
```
final String content = Files.readString(destPath);
content.lines().forEach(System.out::println);
```

=>

```
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, parallel Ausführungen ermöglichen
- Aktionen auf höherer semantischer Ebene als mit Runnable oder Callable<T>

Einstieg CompletableFuture<T>



- **Basisschritte**

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

- **Beispiel**

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

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

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

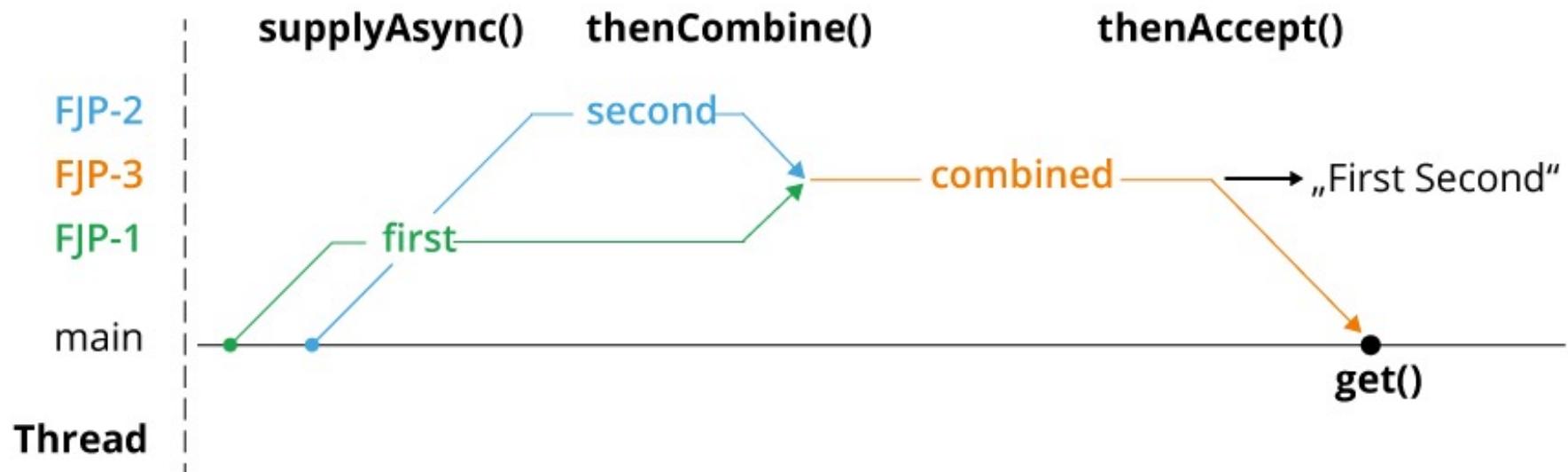
Einführung CompletableFuture<T>



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

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

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



Multi-Threading und die Klasse CompletableFuture<T>

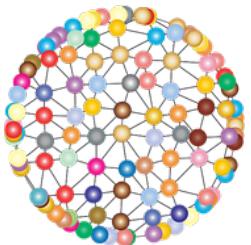


Beispiel: Es sollen folgende Aktionen stattfinden:

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



Wie könnte eine erste
Realisierung aussehen?



Multi-Threading und die Klasse CompletableFuture<T>



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

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

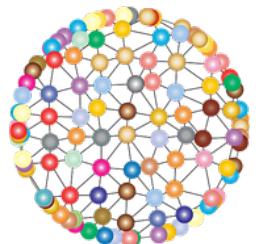
Multi-Threading und die Klasse CompletableFuture<T>



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

Die Berechnungen des Beispiels sind allerdings sehr vereinfacht ...

- **keine Parallelität**
 - **keine Abstimmung der Aufgaben (funktioniert, weil synchron)**
 - **kein Exception-Handling**
-
- **Wir ersparen uns die Mühen und kaum verständliche und 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>?**

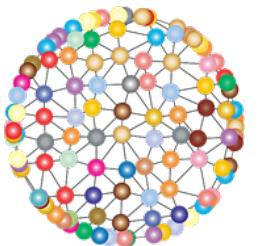
Multi-Threading und die Klasse CompletableFuture<T>



```
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()**

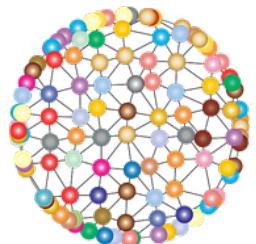
- Fallback-Wert bereitstellen, wenn die Daten nicht ermittelt werden können:

```
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 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 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





- **URL + openStream()**

```
public static void main(final String[] args) throws Exception
{
    final URL oracleUrl = new URL("https://www.oracle.com/index.html");

    final String contents = readContent(oracleUrl.openStream());
    System.out.println(contents);
}
```

- **URL + URLConnection + openConnection() + getInputStream()**

```
public static void main(final String[] args) throws Exception
{
    final URL oracleUrl = new URL("https://www.oracle.com/index.html");

    final URLConnection urlConnection = oracleUrl.openConnection();
    System.out.println(readContent(urlConnection.getInputStream()));
}
```

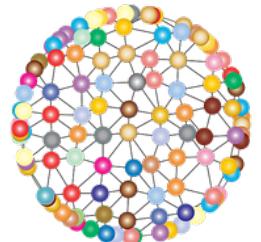


- **Content als String lesen**

```
public static String readContent(final InputStream is) throws IOException
{
    try (final InputStreamReader isr = new InputStreamReader(is);
         final BufferedReader br = new BufferedReader(isr))
    {
        final StringBuilder content = new StringBuilder();

        String line;
        while ((line = br.readLine()) != null)
        {
            content.append(line + "\n");
        }

        return content.toString();
    }
}
```



**Was sagt ihr zu diesem
Code? Was tut er nicht?**

HTTP/2 API Intro



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

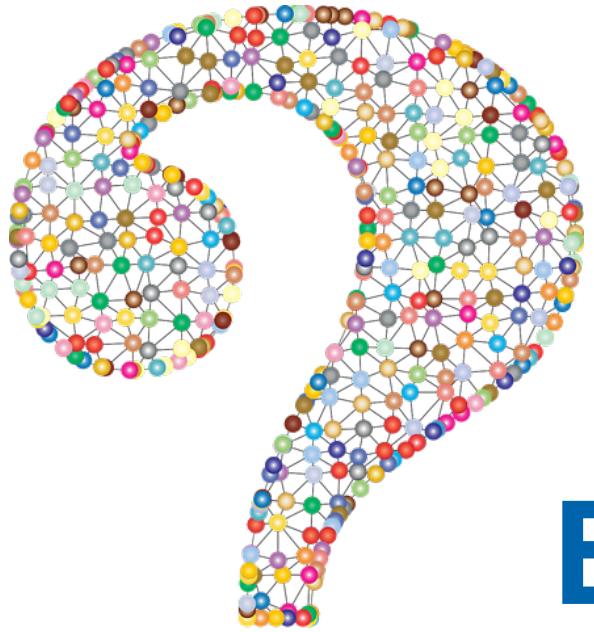
final HttpRequest request = HttpRequest.newBuilder(uri).GET().build();
final BodyHandler<String>asString = HttpResponse.BodyHandlers.ofString();

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

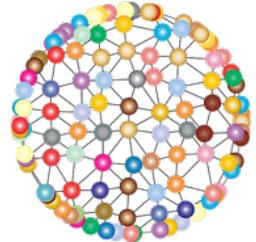
printResponseInfo(response);

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);
}
```



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



HTTP/2 API Async I



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

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

final HttpClient 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

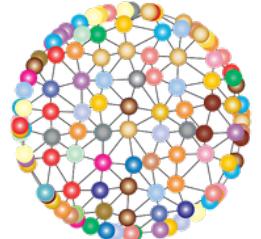


```
static void wait2secForCompletion(final CompletableFuture<?> asyncResponse)
    throws InterruptedException
{
    int i = 0;
    while (!asyncResponse.isDone() && i < 10)
    {
        System.out.println("Step " + i);
        i++;

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



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



HTTP/2 API Async II



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

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



- **HTTPRequest**

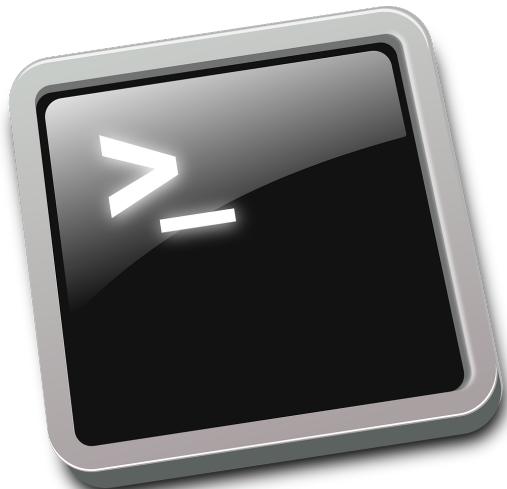
```
.timeout(Duration.ofSeconds(2))  
.header("Content-Type", "application/json")
```

- **HTTPClient**

```
.followRedirects(Redirect.SAME_PROTOCOL)  
.proxy(ProxySelector.of(new InetSocketAddress("www-proxy.com", 8080)))  
.authenticator(Authenticator.getDefault())
```



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 man 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 File



```
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 – REST Call mit HTTP/2 API



```
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse.BodyHandlers;

public class PerformGetWithHttpClient
{
    public static void main(String[] args) throws Exception
    {
        var client = HttpClient.newBuilder().build();
        URI uri = URI.create("https://reqres.in/api/unknown/2");
        var request = HttpRequest.newBuilder().GET().uri(uri).build();

        var response = client.send(request, BodyHandlers.ofString());
        System.out.printf("Response code is: %d %n", response.statusCode());
        System.out.printf("The response body is:%n %s %n", response.body());
    }
}
```

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);
    }
}
```

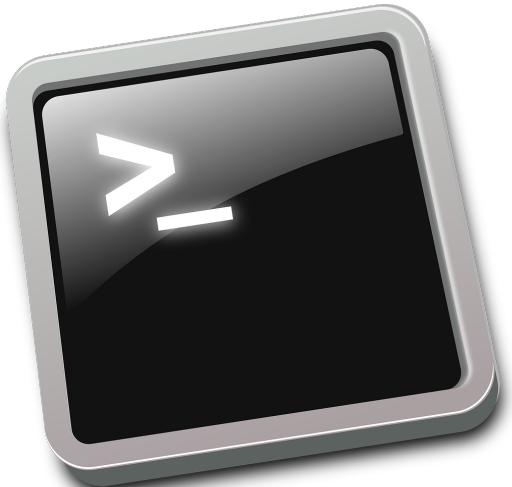
- Datei darf nicht mit ‘.java’ enden
- Dateiname UNABHÄNGIG von Klassennamen
- Datei muss executable (chmod +x) sein



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
```



- 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
- **jshell** verlassen: **/exit**



- Befehlshistorie
 - **/!** wiederholt den letzten Befehl
 - **/list** Liste aller eingegebenen Befehle
 - **/<nr>** Ausführen des Befehls mit der angegebenen Nummer
 - **/reset** löscht Historie
 - **/methods** zeigt Methoden
 - **/imports** zeigt imports
 - **/help** zeigt Hilfe



DEMO

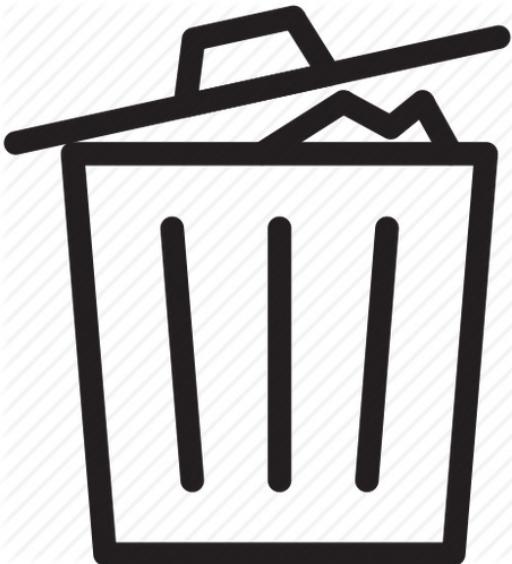
JShell – Was haben wir bisher gelernt?



- 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





- 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**



DEMO

JAXBExample



- **Observer und Observable (java.util)**
- **Thread.destroy() und Thread.stop()**
- **Object.finalize()**
- Konstruktoren der Wrapperklassen, z.B. `new Integer()`, `new Long()`
 - stattdessen `valueOf()` oder `parseXXX()` verwenden
- Applets (`java.awt.Applet`, `javax.swing.JApplet`)



- neues Tool **jdeprscan**: findet Verwendungen von als deprecated markierten Klassen und Methoden
- **jdeprscan --list** zeigt alle als deprecated markierten Teile des JDK
- auch eigene Projekte können überprüft werden, ob sie deprecated APIs verwenden:
 - **jdeprscan target/classes**
 - **jdeprscan myapp.jar**



Übungen PART 2

<https://github.com/Michaeli71/Best-Of-Java-11-18>



PART 3: Syntax-Erweiterungen in Java 12 bis 17

- Syntaxerweiterungen bei `switch`
- Text Blocks
- Records
- Syntaxerweiterung bei `instanceof`
- Local Enums und Interfaces
- 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 numLetters = -1;  
  
switch (day)  
{  
    case MONDAY, FRIDAY, SUNDAY -> numLetters = 6;  
    case TUESDAY -> numLetters = 7;  
    case THURSDAY, SATURDAY -> numLetters = 8;  
    case WEDNESDAY -> numLetters = 9;  
};
```

Switch Expressions als Abhilfe



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

Return-
Value

```
DayOfWeek day = DayOfWeek.FRIDAY;  
int numLetters = 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 Monat auf deren Namen ...

```
// ACHTUNG: Mitunter ganz übler Fehler: default mitten zwischen den cases
String monthString = "";
switch (month)
{
    case JANUARY:
        monthString = "January";
        break;
    default:
        monthString = "N/A"; // hier auch noch Fall Through
    case FEBRUARY:
        monthString = "February";
        break;
    case MARCH:
        monthString = "March";
        break;
    case JULY:
        monthString = "July";
        break;
}
System.out.println("OLD: " + month + " = " + monthString); // 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 switchBreakReturnsValue(Color color)
{
    int numOfChars = switch (color)
    {
        case RED: yield 3;
        case GREEN: yield 5;
        case YELLOW, ORANGE: yield 6;
        default: yield -1;
    };

    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 SQL-Befehlen, regulären Ausdrücken oder der Definition von JavaScript in Java-Sourcecode.
- 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



- <https://openjdk.java.net/jeps/326>

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



- NEU

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

```
String multiLineStringWithPlaceHolders = """
    SELECT %s
    FROM %
    WHERE %
""".formatted(new Object[]{"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 splitted \  
    in several smaller \  
    strings.\  
    """";
```

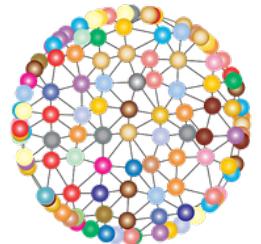
```
System.out.println(text);
```

This is a string splitted in 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 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 sowie equals() und hashCode() automatisch und vor allem kontraktkonform

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]),
              Integer.parseInt(values.split(",")[1]));
    }

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

```
var topLeft = new MyPoint(17, 19);
System.out.println(topLeft);
System.out.println(topLeft.shortForm());
```

MyPoint[x=10, y=10]
[10, 10]

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>

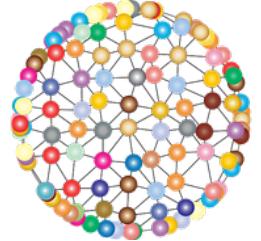


- Was ist an diesem self made Pair falsch?

```
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 Pair, 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



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;
    }
}
```

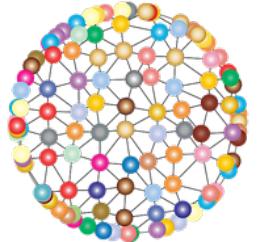
Records mit Gültigkeitsprüfung (Kurzschreibweise)



```
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);
        }
    }
}
```



**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);
    }
}
```



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(obj.getClass());  
}
```

Pattern Matching bei instanceof



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



Lokale Enums und Interfaces



Lokale Enums und Interfaces



```
public class LocalEnumsAndInterfacesExamples
{
    public static void main(String[] args)
    {
        // Erst ab Java 15, vorher Compile-Error:
        // Local enums are not supported at language level '14'
        enum LocalEnumState
        {
            BAD, GOOD, UNKNOWN
        }

        // Erst ab Java 15, vorher Compile-Error:
        // Local interfaces are not supported at language level '14'
        interface Evaluationable
        {
            LocalEnumState evaluate(String info);
        }
        ...
    }
}
```

Lokale Enums und Interfaces



```
public class LocalEnumsAndInterfacesExamples
{
```

```
...
```

```
class AlwaysBad implements Evaluationable
{
    @Override
    public LocalEnumState evaluate(String info)
    {
        return LocalEnumState.BAD;
    }
}
```

```
System.out.println(new AlwaysBad().evaluate("DOES NOT MATTER"));
```

```
}
```

```
}
```



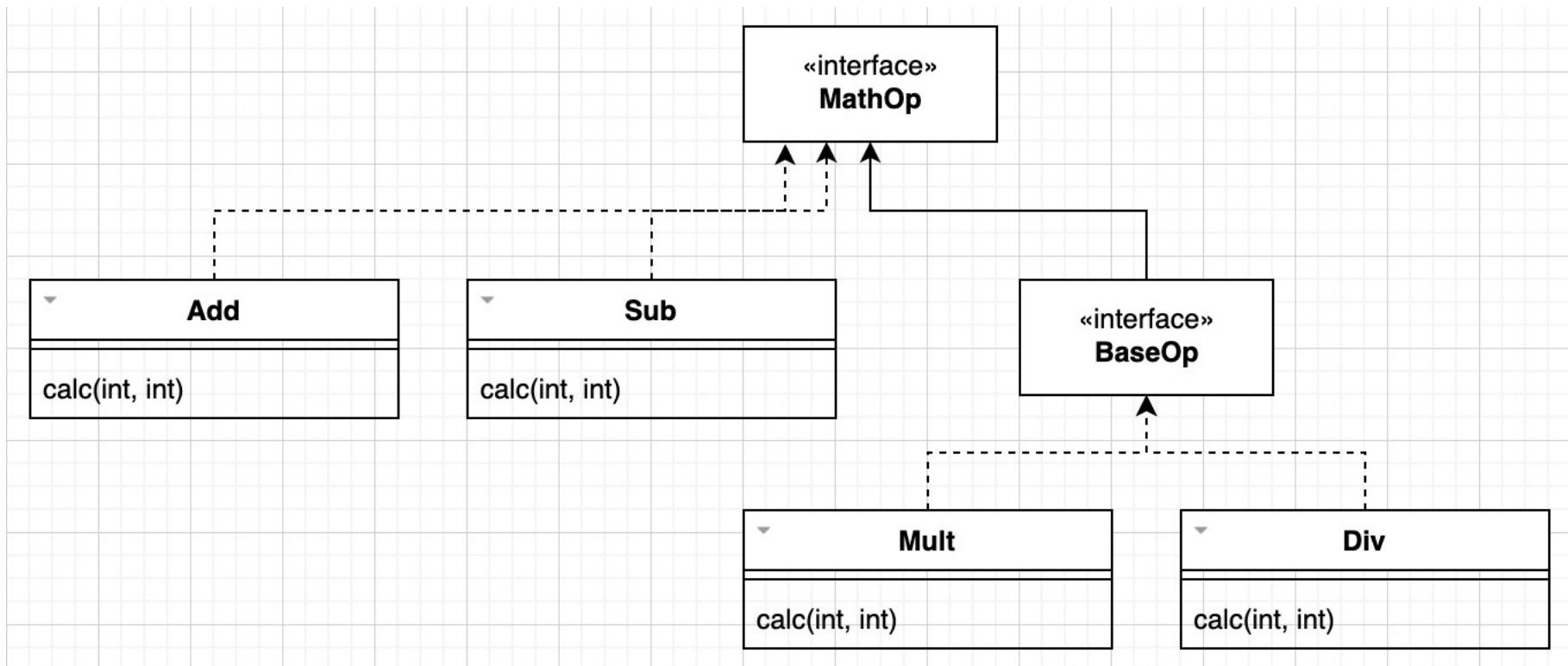
Sealed Types



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



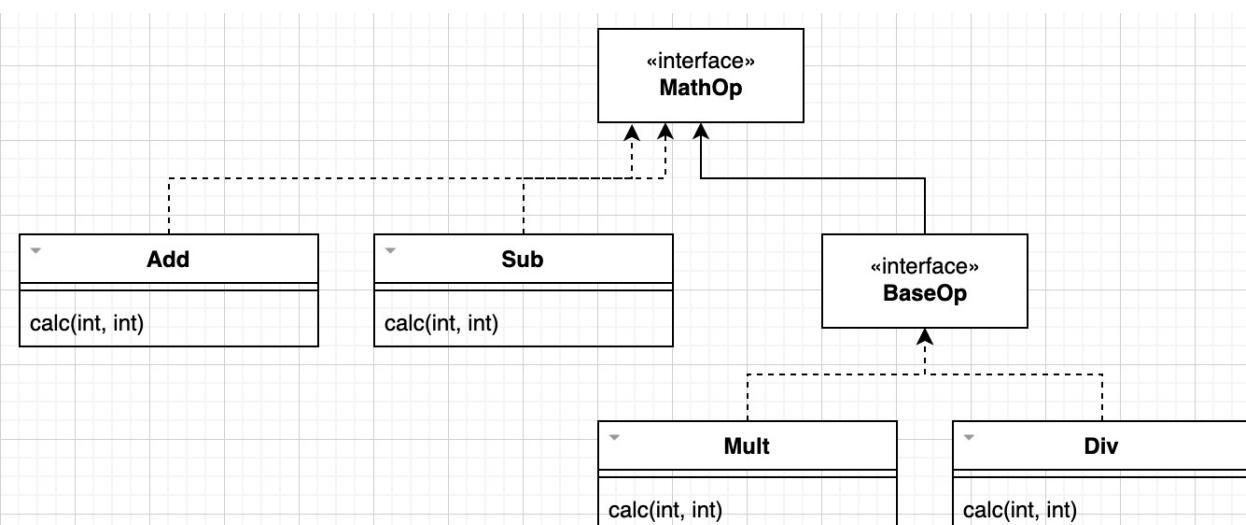
- Spezifizieren, welche anderen Klassen oder Interfaces davon Subtypen bilden dürfen.

```
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.

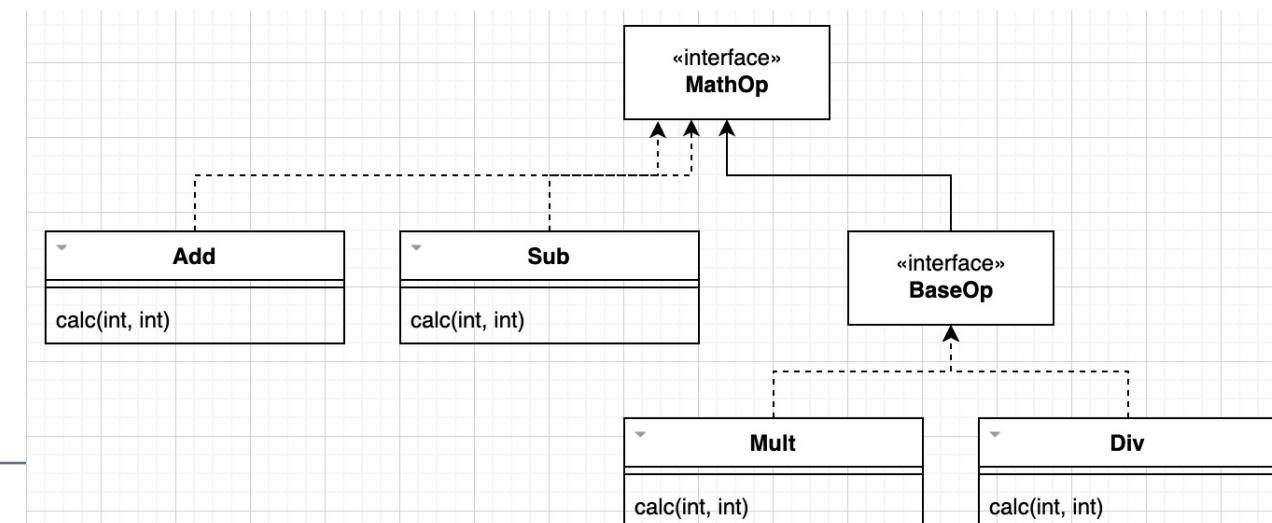


Sealed Types



```
..  
// 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  
{  
    ...  
}  
// Ableitung aus Basisklasse muss final sein  
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.



Wissenswerts zu Sealed Types



- **festlegen, welche andere Klassen oder Interfaces davon Subtypen bilden dürfen.**
 - **Sealed Types können bei der Entwicklung von Bibliotheken: Verhalten über Interfaces exponieren, aber Kontrolle über mögliche Implementierungen**
 - **Sealed Types schränken bezüglich der Erweiterbarkeit von Klassenhierarchien ein und sollten daher mit Bedacht verwendet werden. Bei der Implementierung nicht vorhergesehene Flexibilität kann nachträglich störend sein.**
-



Übungen PART 3

<https://github.com/Michaeli71/Best-Of-Java-11-18>



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

- `CompactNumberFormat`
- `Files`
- `Teeing()-Kollektor`
- `Stream.toList() / mapMulti()`
- `Deprecations`



Erweiterung CompactNumberFormat



Utility-Klasse CompactNumberFormat



- CompactNumberFormat ist eine Subklasse von NumberFormat
 - Formattiert eine Dezimalzahl in kompakter Schreibweise, also 10K statt 10.000
 - Beachtet Locales
 - Es gibt zwar nen Konstruktor, aber einfacher durch Factory-Methode

```
NumberFormat compactFormat =  
    NumberFormat.getCompactNumberInstance(Locale.US,  
        NumberFormat.Style.SHORT);
```

CompactNumberformat Style



```
public static void main(final String args[])
{
    var shortFormat = getUsCompactNumberFormat(DateFormat.Style.SHORT);
    formatNumbers("SHORT", shortFormat);

    var longFormat = getUsCompactNumberFormat(DateFormat.Style.LONG);
    formatNumbers("LONG", longFormat);
}

private static DateFormat getUsCompactNumberFormat(DateFormat.Style style)
{
    return DateFormat.getCompactNumberInstance(Locale.US, style);
}

private static void formatNumbers(final String style,
                                 final DateFormat shortFormat)
{
    System.out.println("\nNumberFormat " + style);
    System.out.println("Result: " + shortFormat.format(10_000));
    System.out.println("Result: " + shortFormat.format(123_456));
    System.out.println("Result: " + shortFormat.format(1_234_567));
    System.out.println("Result: " + shortFormat.format(1_950_000_000));
}
```

CompactNumberformat Style



```
public static void main(final String args[])
{
    var shortFormat = getUsCompactNumberFormat(NumberFormat.Style.SHORT);
    formatNumbers("SHORT", shortFormat);

    var longFormat = getUsCompactNumberFormat(NumberFormat.Style.LONG);
    formatNumbers("LONG", longFormat);
}

private static NumberFormat getUsCompactNumberFormat(NumberFormat.Style style)
{
    return NumberFormat.getCompactNumberInstance(Locale.US, style);
}

private static void formatNumbers(final String style,
                                 final NumberFormat shortFormat)
{
    System.out.println("\nNumberFormat " + style);
    System.out.println("Result: " + shortFormat.format(10_000));
    System.out.println("Result: " + shortFormat.format(123_456));
    System.out.println("Result: " + shortFormat.format(1_234_567));
    System.out.println("Result: " + shortFormat.format(1_950_000_000))
}
```

NumberFormat SHORT	Result: 10K
Result: 123K	Result: 1M
Result: 2B	NumberFormat LONG
Result: 10 thousand	Result: 123 thousand
Result: 1 million	Result: 2 billion

Utility-Klasse CompactNumberformat



```
public static void main(String[] args) throws ParseException
{
    System.out.println("US/SHORT parsing:");

    var shortFormat = NumberFormat.getCompactNumberInstance(Locale.US, Style.SHORT);
    System.out.println(shortFormat.parse("1 K")); // ACHTUNG
    System.out.println(shortFormat.parse("1K"));
    System.out.println(shortFormat.parse("1M"));
    System.out.println(shortFormat.parse("1B"));

    System.out.println("\nUS/LONG parsing:");

    var longFormat = NumberFormat.getCompactNumberInstance(Locale.US, Style.LONG);
    System.out.println(longFormat.parse("1 thousand"));
    System.out.println(longFormat.parse("1 million"));
    System.out.println(longFormat.parse("1 billion"));
}
```

Utility-Klasse CompactNumberformat



```
public static void main(String[] args) throws ParseException
{
    System.out.println("US/SHORT parsing:");

    var shortFormat = NumberFormat.getCompactNumberInstance(Locale.US, Style.SHORT);
    System.out.println(shortFormat.parse("1 K")); // ACHTUNG
    System.out.println(shortFormat.parse("1K"));
    System.out.println(shortFormat.parse("1M"));
    System.out.println(shortFormat.parse("1B"));

    System.out.println("\nUS/LONG parsing:");

    var longFormat =
        NumberFormat.getCompactNumberInstance(Locale.US,
                                              Style.LONG);
    System.out.println(longFormat.parse("1 thousand"));
    System.out.println(longFormat.parse("1 million"));
    System.out.println(longFormat.parse("1 billion"));
}
```

US/SHORT parsing:

1
1000
1000000
1000000000

US/LONG parsing:

1000
1000000
1000000000

Utility-Class CompactNumberformat – Rundung



```
System.out.println(compactNumberFormat.format(990L)); // 990  
System.out.println(compactNumberFormat.format(999L)); // 999
```

```
System.out.println(compactNumberFormat.format(1_499L)); // 1k  
System.out.println(compactNumberFormat.format(1_500L)); // 2k  
System.out.println(compactNumberFormat.format(1_999L)); // 2k
```

```
System.out.println(compactNumberFormat.format(1_499_999)); // 1m  
System.out.println(compactNumberFormat.format(1_500_000)); // 2m  
System.out.println(compactNumberFormat.format(1_567_890)); // 2m
```

Utility-Class CompactNumberformat – Rundung anpassen



```
compactNumberFormat.setMinimumFractionDigits(1);
```

```
System.out.println(compactNumberFormat.format(990L)); // 990  
System.out.println(compactNumberFormat.format(999L)); // 999
```

```
System.out.println(compactNumberFormat.format(1_499L)); // 1,5k  
System.out.println(compactNumberFormat.format(1_500L)); // 1,5k  
System.out.println(compactNumberFormat.format(1_999L)); // 2,0k
```

```
System.out.println(compactNumberFormat.format(1_499_999)); // 1,5m  
System.out.println(compactNumberFormat.format(1_500_000)); // 1,5m  
System.out.println(compactNumberFormat.format(1_567_890)); // 1,6m
```

Utility-Klasse CompactNumberformat – Anpassungen



```
final String[] compactPatterns = { "", "", "",  
"0 [KB]", "00 [KB]", "000 [KB]", "0 [MB]", "00 [MB]", "000 [MB]",  
"0 [GB]", "00 [GB]", "000 [GB]", "0 [TB]", "00 [TB]", "000 [TB]" };  
  
final DecimalFormat decimalFormat = (DecimalFormat)  
NumberFormat.getNumberInstance(Locale.GERMANY);  
  
final CompactNumberFormat customCompactNumberFormat = new  
CompactNumberFormat(decimalFormat.toPattern(),  
decimalFormat.getDecimalFormatSymbols(), compactPatterns);  
  
customCompactNumberFormat.setMinimumFractionDigits(1);  
System.out.println(customCompactNumberFormat.format(990L));  
System.out.println(customCompactNumberFormat.format(999L));  
System.out.println(customCompactNumberFormat.format(1_499L));  
System.out.println(customCompactNumberFormat.format(1_500L));  
System.out.println(customCompactNumberFormat.format(1_999L));  
System.out.println(customCompactNumberFormat.format(1_499_999));  
System.out.println(customCompactNumberFormat.format(1_500_000));  
System.out.println(customCompactNumberFormat.format(1_567_890));
```

990
999
1,5 [KB]
1,5 [KB]
2,0 [KB]
1,5 [MB]
1,5 [MB]
1,6 [MB]



Erweiterung in der Klasse Files



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



- In Java 9 wurden Methoden zum Vergleichen von Arrays eingeführt
- In Java 11 wurde die Verarbeitung von Strings im Zusammenhang mit Dateien erleichtert.
- In Java 12 und im nachfolgenden Beispiel sind diese kombiniert

```
Path filePath1 = Files.createTempFile("test1", ".txt");
Path filePath2 = Files.createTempFile("test2", ".txt");
Path filePath3 = Files.createTempFile("test3", ".txt");

Files.writeString(filePath1, "Same Content");
Files.writeString(filePath2, "Same Content");
Files.writeString(filePath3, "Same Start / Different Content");

long mismatchPos1 = Files.mismatch(filePath1, filePath2);
System.out.println("File1 mismatch File2 = " + mismatchPos1);

long mismatchPos2 = Files.mismatch(filePath1, filePath3);
System.out.println("File1 mismatch File3 = " + mismatchPos2);
```

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



```
Path filePath1 = Files.createTempFile("test1", ".txt");
Path filePath2 = Files.createTempFile("test2", ".txt");
Path filePath3 = Files.createTempFile("test3", ".txt");

Files.writeString(filePath1, "Same Content");
Files.writeString(filePath2, "Same Content");
Files.writeString(filePath3, "Same Start / Different Content");

long mismatchPos1 = Files.mismatch(filePath1, filePath2);
System.out.println("File1 mismatch File2 = " + mismatchPos1);

long mismatchPos2 = Files.mismatch(filePath1, filePath3);
System.out.println("File1 mismatch File3 = " + mismatchPos2);
```

```
File1 mismatch File2 = -1
File1 mismatch File3 = 5
```

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



```
Path fileEnc1 = Files.createTempFile("enc1", ".latin1");
Path fileEnc2 = Files.createTempFile("enc2", ".utf8");

Files.writeString(fileEnc1, "Zürich is beautiful. Mainz too",
                  StandardCharsets.ISO_8859_1);
Files.writeString(fileEnc2, "Zürich is beautiful. Mainz too",
                  StandardCharsets.UTF_8);

var mismatchPos = Files.mismatch(fileEnc1, fileEnc2);
System.out.println("enc1 mismatch enc2 = " + mismatchPos);

var oneFile = Files.createTempFile("oneFile", ".txt");

var mismatchPosSameFile = Files.mismatch(oneFile, oneFile);
System.out.println("oneFile mismatch oneFile = " + mismatchPosSameFile);
```

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



```
Path fileEnc1 = Files.createTempFile("enc1", ".latin1");
Path fileEnc2 = Files.createTempFile("enc2", ".utf8");

Files.writeString(fileEnc1, "Zürich is beautiful. Mainz too",
                  StandardCharsets.ISO_8859_1);
Files.writeString(fileEnc2, "Zürich is beautiful. Mainz too",
                  StandardCharsets.UTF_8);

var mismatchPos = Files.mismatch(fileEnc1, fileEnc2);
System.out.println("enc1 mismatch enc2 = " + mismatchPos);

var oneFile = Files.createTempFile("oneFile", ".txt");

var mismatchPosSameFile = Files.mismatch(oneFile, oneFile);
System.out.println("oneFile mismatch oneFile = " + mismatchPosSameFile);
```

enc1 mismatch enc2 = 1
oneFile mismatch oneFile = -1



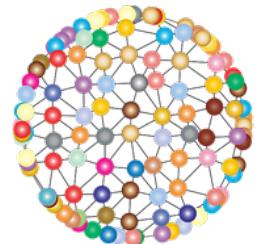
Teeing()-Kollektor





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

- `toCollection()`, `toList()`, `toSet()` ... – Sammlung der Elemente des Stream in die entsprechende Collection.
- Mit Java 10 kamen auch noch `toUnmodifiableXyz()` hinzu.



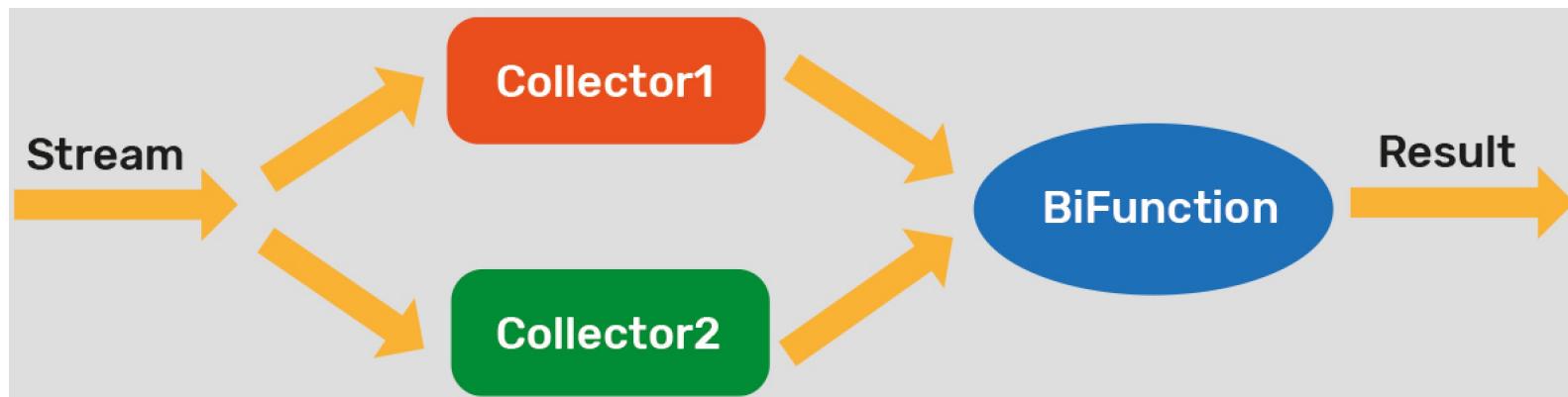
Was fehlt denn noch?



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



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

Ausflug Pair<T> (Simpelste Form, nicht allgemeingültig)



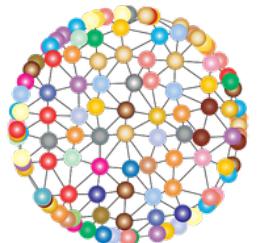
```
static class Pair<T>
{
    public T first;
    public T second;

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

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



NICHT SO GUT!! Warum?



Für Pair in modernem Java besser Records nutzen!



```
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 + "]";
    }
}
```



```
record LongPair(long count, long sum)
{}
```



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();
```



mapMulti()

Warum mapMulti()?



// OLD

```
Stream.of(Optional.of("0"), Optional.empty(), Optional.of("1"),
         Optional.empty(), Optional.of("2"), Optional.empty(), Optional.of("3"))
    .flatMap(Optional::stream)
    .forEach(System.out::print);
```

// NEW

```
Stream.of(Optional.of("0"), Optional.empty(), Optional.of("1"),
         Optional.empty(), Optional.of("2"), Optional.empty(), Optional.of("3"))
    .mapMulti(Optional::ifPresent). // !!!
    .forEach(System.out::print);
```

Warum mapMulti()?



// NEW

```
Stream.of(Optional.of("0"), Optional.empty(), Optional.of("1"),
         Optional.empty(), Optional.of("2"), Optional.empty(), Optional.of("3"))
    .mapMulti(Optional::ifPresent). // !!!
    forEach(System.out::print);
```

// Mehr der Gedanke wie imperative for-Schleife

```
var elements = List.of(Optional.of("0"), Optional.empty(), Optional.of("1"),
                      Optional.empty(), Optional.of("2"), Optional.empty(), Optional.of("3"));

for (Optional optElem : elements )
{
    optElem.ifPresent(System.out::print);
}
```

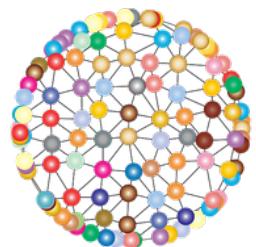


Wie bauen wir das?

```
Stream.of(List.of(1, 2, 3),  
         "ABC", null,  
         Set.of("X", "Y", "Z"));
```

=>

```
[1, 2, 3, ABC, null, Z, Y, X]
```



Warum mapMulti()? Beispiel expandIterables()



```
// OLD  
var stream = Stream.of(List.of(1, 2, 3), "ABC", null, Set.of("X", "Y", "Z"));
```

```
Stream<Object> expandedStream = stream.flatMap(e -> {  
    if (e instanceof Iterable<?> iterable) {  
        // Trick to convert Iterable to Stream  
        return StreamSupport.stream(iterablespliterator(), false);  
    }  
    return Stream.of(e);  
});
```

```
System.out.println(expandedStream.toList());
```



[1, 2, 3, ABC, null, Z, Y, X]

Warum mapMulti()? Beispiel expandIterables()



```
// NEW
var stream2 = Stream.of(List.of(1, 2, 3), "ABC", null, Set.of("X", "Y", "Z"));

Stream<Object> expandedStream2 = stream2.mapMulti(MapMultiExample2::expandIterable);

System.out.println(expandedStream2.toList());
```

```
static void expandIterable(Object e, Consumer<Object> c) {
    if (e instanceof Iterable<?> iterable) {
        for (Object elem : iterable) {
            expandIterable(elem, c);
        }
    } else {
        c.accept(e);
    }
}
```

[1, 2, 3, ABC, null, Z, Y, X]



Deprecations



Deprecations



- Konstruktion von Wrappern:

```
/Users/michaelinden/java8-
workspace/Java17Examples/src/main/java/primitives/PrimitiveCtorDeprecati
onExample.java:6: warning: [removal] Long(long) in Long has been
deprecated and marked for removal
```

```
    Long myLong = new Long(1234L);
               ^
```

```
/Users/michaelinden/java8-
workspace/Java17Examples/src/main/java/primitives/PrimitiveCtorDeprecati
onExample.java:8: warning: [removal] Integer(int) in Integer has been
deprecated and marked for removal
```

```
    Integer myInt = new Integer(1234);
               ^
```



Übungen PART 4

<https://github.com/Michaeli71/Best-Of-Java-11-18>



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(); }
}

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 und die Differenz zwischen den Werten ermittelt

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



- Wiederholte Start-/Stop-Messungen

```
// 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: Micobenchmark Suite / JMH als Unterstützung



- **JEP 230 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



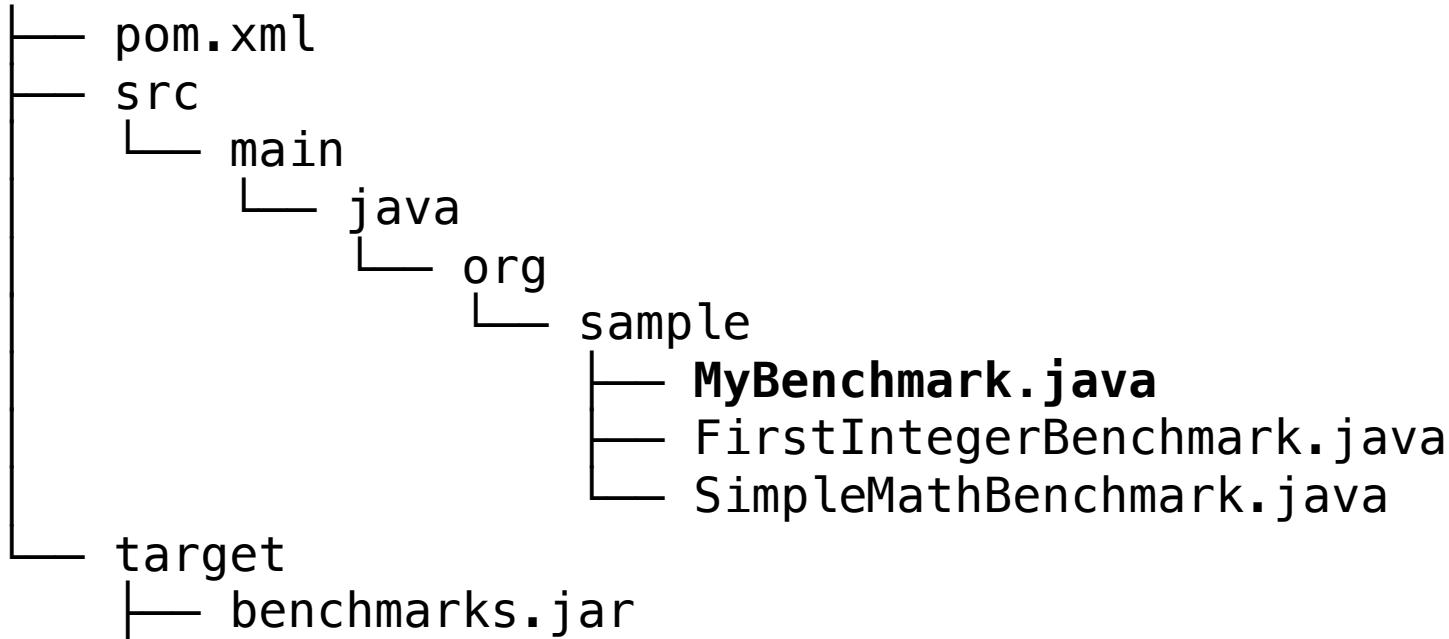
- Als Grundgerüst wird eine Klasse MyBenchmark erzeugt:

```
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 eigen 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

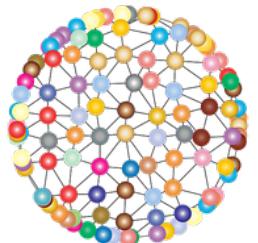
 main(String[]) : void

► src

 build.gradle

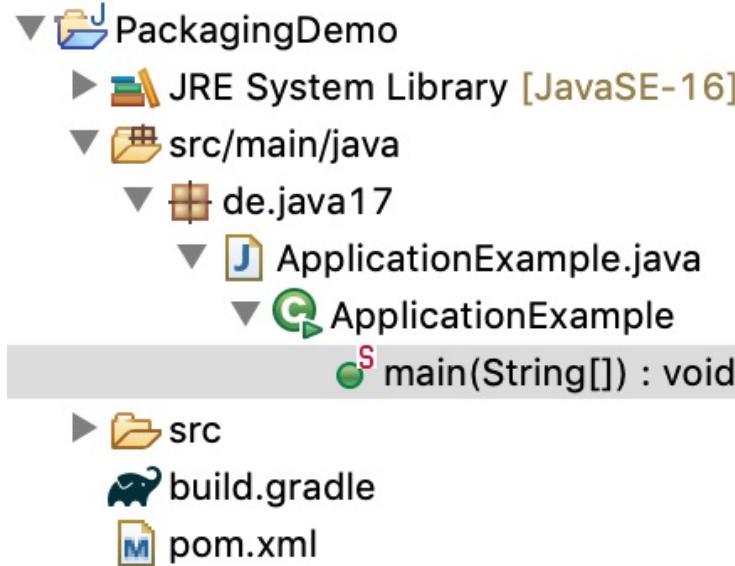
 pom.xml

```
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)  
    }  
}
```

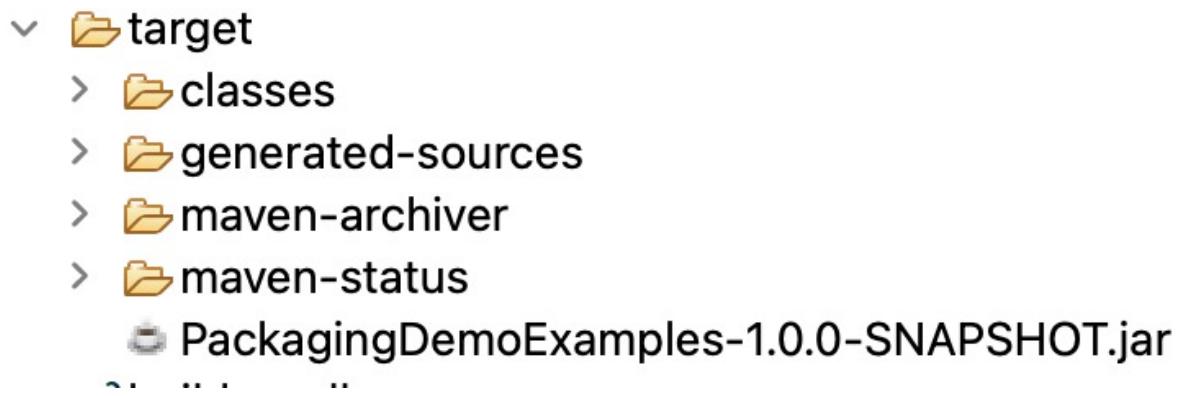


Wie bauen wir das?

JPackage



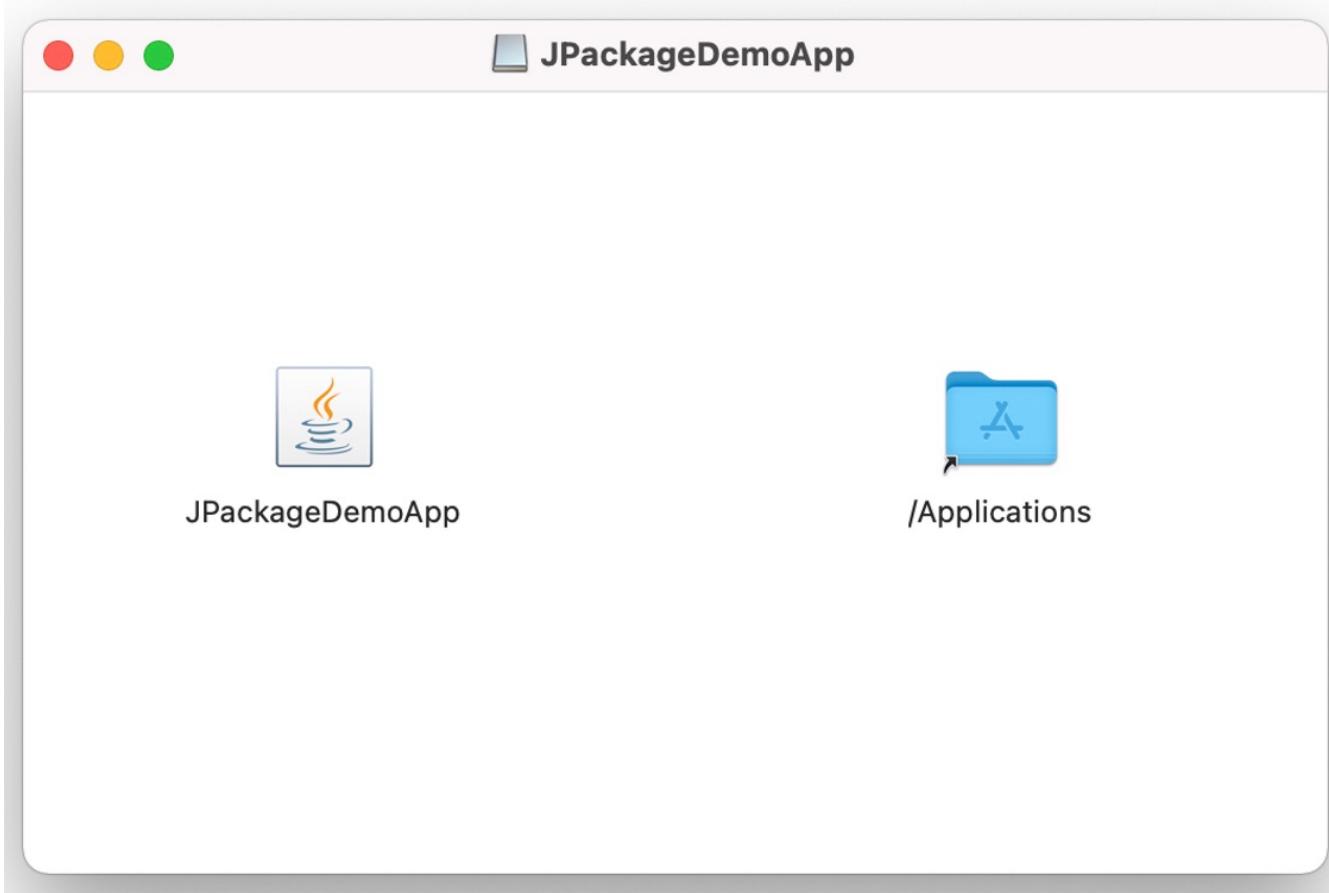
`mvn clean install`



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?**

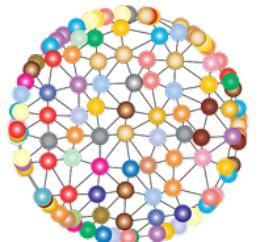
JPackage



```
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 den 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

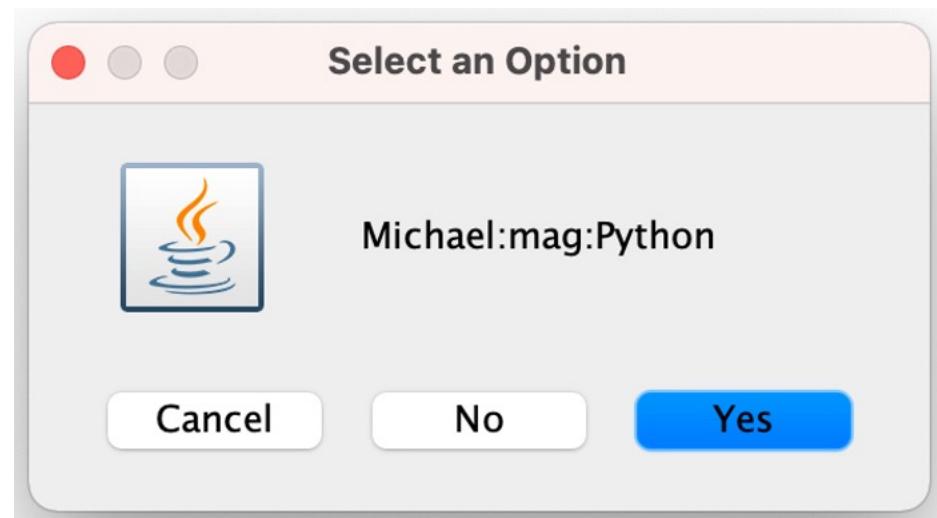


- ▼ target
 - > classes
 - > generated-sources
 - > maven-archiver
 - > maven-status
 - ⌚ checker-qual-3.12.0.jar
 - ⌚ error_prone_annotations-2.7.1.jar
 - ⌚ failureaccess-1.0.1.jar
 - ⌚ guava-31.0.1-jre.jar
 - ⌚ j2objc-annotations-1.3.jar
 - ⌚ jsr305-3.0.2.jar
 - ⌚ listenablefuture-9999.0-empty-to-avoid-conflict-with-guav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



```
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);
}
```

```
Snippet:VariableKey(name)#1-var name = "Mike";
"Mike"
variable: 'name' / type: String' / value: "Mike"
```

JShell-API



```
try (JShell js = JShell.create())
{
    // Achtung: Hier ist das Semikolon nötig, sonst inkorrekte Auswertung
    String valA = js.eval("int a = 42;").get(0).value();
    System.out.println("valA = " + valA);
    String valB = js.eval("int b = 7;").get(0).value();
    System.out.println("valB = " + valB);
    String result = js.eval("int result = a / b;").get(0).value();
    System.out.println("Result = " + result);

    js.variables().map(varSnippet -> varSnippet.name() + " => " +
        varSnippet.source()).forEach(System.out::println);
}
```

```
valA = 42
valB = 7
result = 6
a => int a = 42;
b => int b = 7;
result => int result = a / b;
```



DEMO



Übungen PART 5

<https://github.com/Michaeli71/Best-Of-Java-11-18>



Recap: Neuerungen in Java 17

- Überblick: Was ist drin?
 - Syntaxerweiterungen bei switch (PREVIEW)
-

Java 17 – Was ist drin?



- JEP 306: [Restore Always-Strict Floating-Point Semantics](#)
- JEP 356: [Enhanced Pseudo-Random Number Generators](#)
- JEP 382: [New macOS Rendering Pipeline](#)
- JEP 391: [macOS/AArch64 Port](#)
- JEP 398: [Deprecate the Applet API for Removal](#)
- JEP 403: [Strongly Encapsulate JDK Internals](#)
- JEP 406: [Pattern Matching for switch \(Preview\)](#)
- JEP 407: [Remove RMI Activation](#)
- JEP 409: [Sealed Classes](#)
- JEP 410: [Remove the Experimental AOT and JIT Compiler](#)
- JEP 411: [Deprecate the Security Manager for Removal](#)
- JEP 412: [Foreign Function & Memory API \(Incubator\)](#)
- JEP 414: [Vector API \(Second Incubator\)](#)
- JEP 415: [Context-Specific Deserialization Filters](#)

Was ist davon wirklich wichtig für uns?

- JEP 406: [Pattern Matching for switch \(Preview\)](#)
 - ?? JEP 409: [Sealed Classes](#) ??
 - ?? JEP 391: [macOS/AArch64 Port](#) ??
-



The Java Version Almanac

Collection of information about the history and future of Java.

Details	Status	Documentation	Download	Compare API to									
Java 18	DEV	API 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											



Pattern Matching bei switch (PREVIEW)



Pattern Matching bei switch



- Mit Java 16 wurde Pattern Matching für instanceof eingeführt. Damit ist es möglich, ein sogenanntes Typmuster zu akzeptieren und einen Mustervergleich durchzuführen. Dadurch kann man sich lästige Casts sparen.
- Diese Syntax-Neuerung ist mit Java 17 auch für switch möglich
- Allerdings ist das Ganze zunächst in Form von Preview Features umgesetzt und zum Nachvollziehen müssen Sie diese geeignet aktivieren, etwa wie folgt in der JShell:

```
$ jshell --enable-preview
| Welcome to JShell -- Version 17-ea
| For an introduction type: /help intro
```

Pattern Matching bei switch



- Nehmen wir an, wir müssten eine generische Methode zum Formatieren von Werten schreiben. Sofern wir Java 16 nutzen, können wir dies mithilfe von Pattern Matching und instanceof einigermaßen lesbar wie folgt schreiben -- das Ganze wäre ohne Java 16 deutlich schlimmer, da wir für jeden Typ eine Zeile mit einem Cast ergänzen müssten:

```
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;  
}
```

Pattern Matching bei switch



- Bislang 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");  
    }  
}
```

Pattern Matching bei switch



- Analog zu instanceof sind in den cases nun 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");  
    }  
}
```



PART 6: Neuerungen in Java 18

18



Build-Tools und IDEs



IDE & Tool Support für Java 18



- Aktuelle IDEs & Tools grundsätzlich gut
- Eclipse: Version 2022-03 mit zusätzlichem Plugin
- IntelliJ: Version 2022.1
- Maven: 3.8.5, Compiler-Plugin: 3.8.1 (besser 3.10)
- Gradle: 7.4.2
- Aktivierung von Preview-Features nötig
 - In Dialogen
 - Im Build-Skript



Maven™



IDE & Tool Support Java 18



- Eclipse 2022-03 mit Plugin
- Aktivierung von Preview-Features nötig

Eclipse Marketplace

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

Search Recent Popular Favorites Installed Giving IoT an Edge

Find: Search All Markets All Categories Go

Featured

Java 18 Support for Eclipse 2022-03 (4.23)

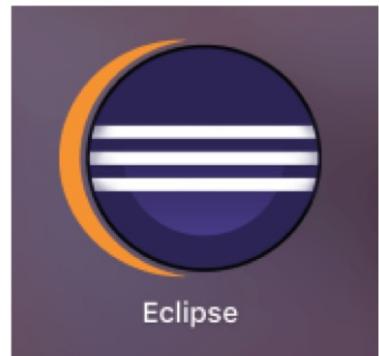
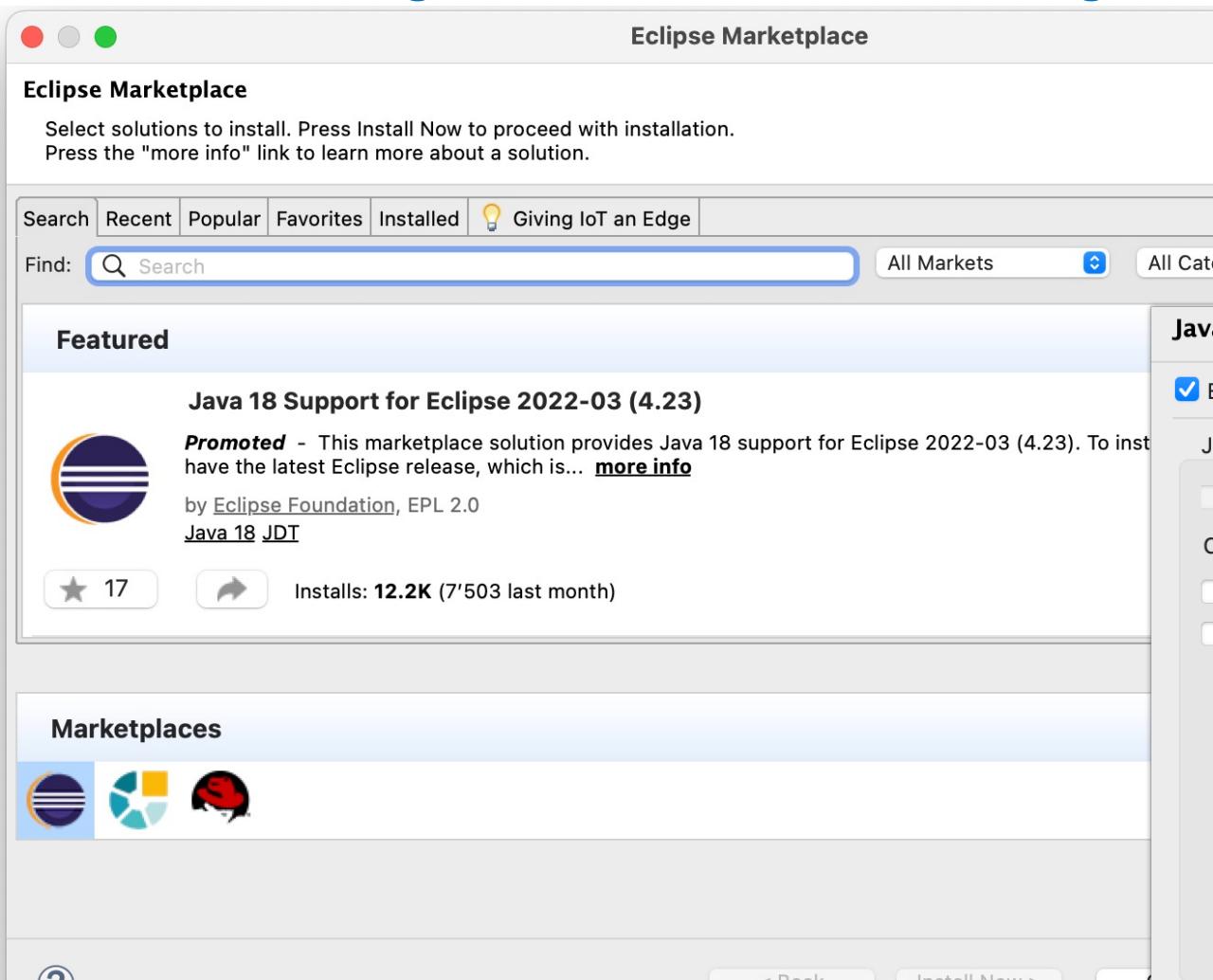
Promoted - This marketplace solution provides Java 18 support for Eclipse 2022-03 (4.23). To install have the latest Eclipse release, which is... [more info](#)

by Eclipse Foundation, EPL 2.0 [Java 18 JDT](#)

17 installs: 12.2K (7'503 last month)

Marketplaces

② Back Install Now



Java Compiler

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

JDK Compliance

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

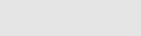
Compiler compliance level: **18** 

Use '--release' option

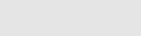
Use default compliance settings

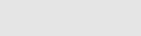
Enable preview features for Java 18 

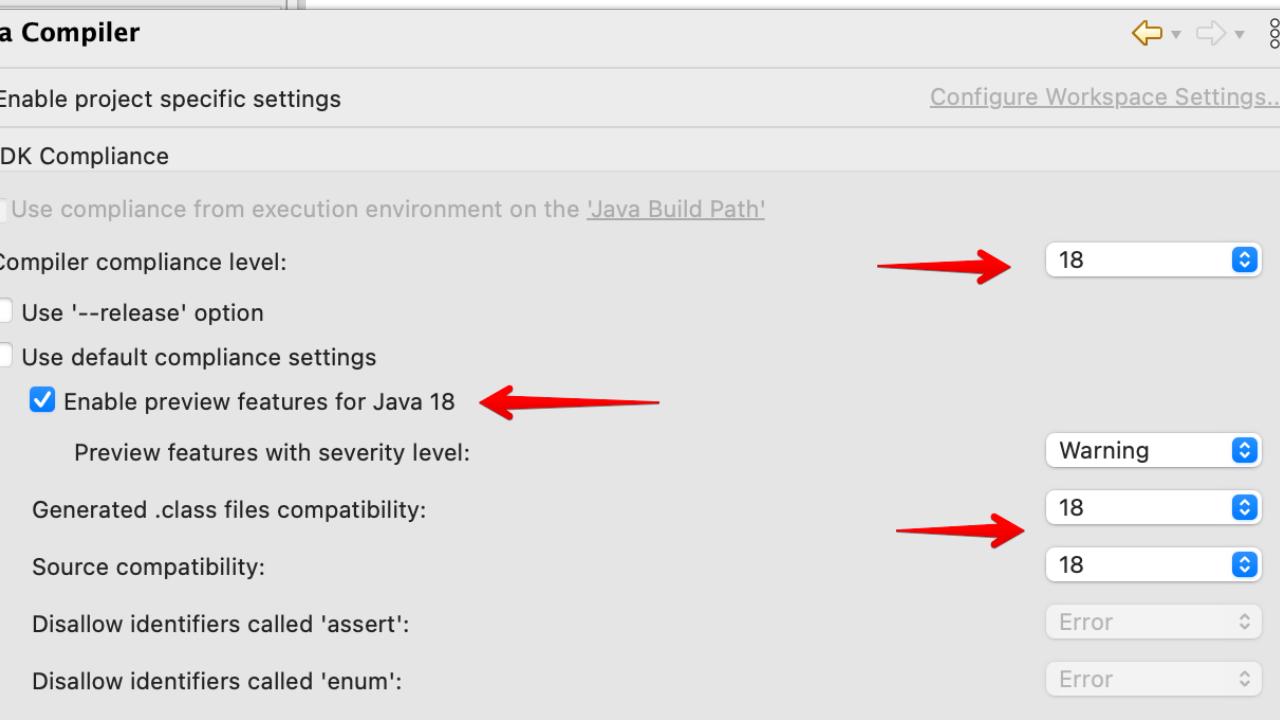
Preview features with severity level:

Generated .class files compatibility: **Warning** 

Source compatibility: **18** 

Disallow identifiers called 'assert': **18** 

Disallow identifiers called 'enum': **Error** 



IDE & Tool Support



- Aktivierung von Preview-Features nötig

Project Structure

← →

Project Settings

- Project
- Modules
- Libraries
- Facets
- Artifacts

Platform Settings

- SDKs
- Global Libraries

Project

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

Name: Java18Examples

SDK: 18 Oracle OpenJDK version 18.0.1 Edit

Language level: 18 (Preview) - Pattern matching for switch (second preview)

Compiler output: ~/java8-workspace/Java18Examples/out

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



IDE & Tool Support Java 18



- Aktivierung von Preview-Features nötig

```
sourceCompatibility=18  
targetCompatibility=18
```

```
// Aktivierung von Switch Expressions Preview  
tasks.withType(JavaCompile) {  
    options.compilerArgs += ["--enable-preview"]  
}
```



IDE & Tool Support



- Aktivierung von Preview-Features nötig

```
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.10</version>
    <configuration>
      <source>18</source>
      <target>18</target>
      <!-- Wichtig für Java Syntax-Neuerungen -->
      <compilerArgs>--enable-preview</compilerArgs>
    </configuration>
  </plugin>
</plugins>
```

Maven™

```
<release>18</release>
```



Java

18



- 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**](#)
-



[Feedback on this page?](#)

The Java Version Almanac

Collection of information about the history and future of Java.

Details	Status	Documentation	Download	Compare API to										
Java 19	DEV	API Notes	JDK JRE	18	17	16	15	14	13	12	11	10	9	...
Java 18	DEV	API 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												



Java 18

Status	In Development
Release Date	2022-03-15
EOL Date	2022-09-15
Bytecode Version	62.0
API Changes	Compare to 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](#)

New Features

JVM

- UTF-8 by Default ([JEP 400](#))

Language

- Pattern Matching for switch (Second Preview) [Preview](#) ([JEP 420](#))

API

- Reimplement Core Reflection with Method Handles ([JEP 416](#))
- Internet-Address Resolution SPI ([JEP 418](#))
- Deprecate Finalization for Removal ([JEP 421](#))

Tools

- Simple Web Server ([JEP 408](#))
- Code Snippets in Java API Documentation ([JEP 413](#))



Sandbox

Instantly compile and run Java 18 snippets without a local Java installation.

Java18.java

▶ Run

18+36-2087

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

public class Java18 {

    public static void main(String[] args) {
        System.out.println("Runtime required for this: " + SourceVersion.RELEASE_18.runtimeVersion());

        switchSupportingNull(null);
        switchSupportingNull("JAVA");
    }

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



Sandbox

Instantly compile and run Java 18 snippets without a local Java installation.

Java18.java ► Run

18+36-2087

```
Java18.java:14: warning: [preview] null in switch cases is a preview feature and may be removed in a future re
      case null -> System.out.println("null is allowed in preview");
                         ^
Runtime required for this: 18
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "String.toUpperCase()" because "<par
  at Java18.switchSupportingNull(Java18.java:13)
  at Java18.main(Java18.java:8)
```



JEP 408: Simple Web Server





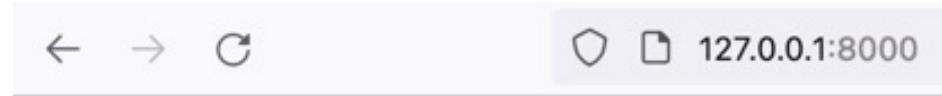
- Ziel bei diesem JEP war es, einen minimalistischen Webserver ohne viel Konfiguration ins JDK zu integrieren, der lediglich statische Dateien bereitstellen kann. Dies erleichtert die Erstellung von Prototypen, das File Sharing oder kann auch für einfache Tests hilfreich sein.

```
michaelinden@MBP-von-Michael ~ % jwebserver
Binding to loopback by default. For all interfaces use "-b 0.0.0.0" or "-b ::".
Serving /Users/michaelinden and subdirectories on 127.0.0.1 port 8000
URL http://127.0.0.1:8000/
```

```
jshell> import com.sun.net.httpserver.SimpleFileServer;
...> import com.sun.net.httpserver.SimpleFileServer.OutputLevel;

jshell> var server = SimpleFileServer.createFileServer(new
InetSocketAddress(8000), Path.of("/Users/michaelinden"), OutputLevel.VERBOSE)
server ==> sun.net.httpserver.HttpServerImpl@6659c656

jshell> server.start()
```



Directory listing for /

- [Music/](#)
- [JavaDateAndTimeOverview_RO.astar](#)
- [spring-boot-test-workspace/](#)
- [JDK_SET.txt](#)
- [spacerocks.dat](#)
- [java8-workspace/](#)
- [h2](#)
- [spring-boot-persistence-data-workspace/](#)
- [Pictures/](#)
- [zshrc.textClipping](#)
- [JavaDateAndTimeOvervie_IFs.png](#)
- [Desktop/](#)
- [Library/](#)
- [eclipse-workspace/](#)
- [mongodb-macos-x86_64-5.0.3/](#)
- [spring-framework-intro-workspace/](#)
- [Sites/](#)
- [splash.png](#)
- [my.properties](#)
- [PycharmProjects/](#)



DEMO & Hands on



JEP 413: Code Snippets in Java API Documentation



JEP 413: Code Snippets in Java API Documentation



- Bislang gibt es keinen Standard, um Codefragmente in die mit Java-Doc generiert HTML-Dokumentation aufzunehmen. Im Rahmen dieses JEPs wird das Inline-Tag @snippet eingeführt. Damit lassen sich Codefragmente in einen Java-Doc-Kommentar einfügen:

```
/**  
 * The following code shows how to use {@code Optional.isPresent}:  
 * {@snippet :  
 * if (optValue.isPresent())  
 * {  
 *     System.out.println("value: " + optValue.get());  
 * }  
 * }  
 */  
  
public static void method1(String[] args) {  
    //  
    //  
    //  
    //  
}  
}
```

 **void java18.misc.JavaDocExample.method1(String[] args)**

The following code shows how to use `Optional.isPresent`:

```
if (optValue.isPresent())  
{  
    System.out.println("value: " + optValue.get());  
}
```



- **Hervorherbungen**

```
/**  
 * The following code shows how to use {@code Optional.isPresent} and  
 * {@code Optional.get} in combination  
 *  
 * {@snippet :  
 * if (optValue.isPresent()) // @highlight substring="isPresent"  
 * {  
 *     System.out.println("value: " + optValue.get()); // @highlight substring="get"  
 * } }  
 */  
  
public static void newJavaDocExample(String[] args) {  
    //  
    //  
    //  
    //  
}  
}
```

 **void java18.misc.JavaDocExample.newJavaDocExample(String[] args)**

The following code shows how to use `Optional.isPresent` and `Optional.get` in combination

```
if (optValue.isPresent())  
{  
    System.out.println("value: " + optValue.get());  
}
```

Parameters:
args



JEP 418: Internet-Address Resolution SPI



JEP 418: Internet-Address Resolution SPI



- Im Rahmen dieses JEP wurde ein SPI (Service Provider Interface) für die Auflösung von Host- und Namensadressen implementiert.
- Die Adressauflösung verwendet bislang eine hartcodierte Umsetzung innerhalb der Klasse `java.net.InetAddress`.

```
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.util.Arrays;

public class InetAddressExmaple
{
    public static void main(String[] args) throws UnknownHostException
    {
        InetAddress[] addresses = InetAddress.getAllByName("www.dpunkt.de");
        System.out.println("addresses = " + Arrays.toString(addresses));
    }
}
```

JEP 418: Internet-Address Resolution SPI



```
public class OwnInetAddressResolver implements InetAddressResolver
{
    @Override
    public Stream<InetAddress> lookupByName(String host, LookupPolicy lookupPolicy)
        throws UnknownHostException
    {
        if (host.equals("www.dpunkt.de"))
            return Stream.of(InetAddress.getByAddress(new byte[] { 127, 0, 0, 1 }));

        throw new UnsupportedOperationException();
    }

    @Override
    public String lookupByAddress(byte[] addr)
    {
        throw new UnsupportedOperationException();
    }
}
```

JEP 418: Internet-Address Resolution SPI



```
import java.net.spi.InetAddressResolver;
import java.net.spi.InetAddressResolverProvider;

public class OwnInetAddressResolverProvider extends InetAddressResolverProvider
{
    @Override
    public InetAddressResolver get(Configuration configuration) {
        return new OwnInetAddressResolver();
    }

    @Override
    public String name() {
        return "Own Internet Address Resolver Provider";
    }
}
```





DEMO & Hands on



JEP 420: Pattern Matching bei switch (PREVIEW II)



JEP 420: Pattern Matching bei switch



- JEP 420 führt zu zwei Änderungen bei der Auswertung der case innerhalb switch beim Kompilieren:
 - zum einen ändert sich die sogenannte Dominanzprüfung,
 - zum anderen wurde die Vollständigkeitsanalyse korrigiert.
- Allerdings ist das Ganze wieder in Form eines Preview Features umgesetzt und zum Nachvollziehen müssen Sie diese geeignet aktivieren, etwa wie folgt in der JShell:

```
michaelinden@MBP-von-Michael ~ % jshell --enable-preview
| Welcome to JShell -- Version 18-ea
| For an introduction type: /help intro
```

JEP 420: Pattern Matching bei switch: Dominanzprüfung



- Es können mehrere Pattern auf eine Eingabe matchen. Dasjenige, das «am besten» passt, wird dominierendes Pattern genannt. Im Beispiel dominiert das kürzere Pattern String s das längere davor angegebene.

```
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 && 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 -> {}  
    }  
}
```



- Es können mehrere Pattern auf eine Eingabe matchen. Dasjenige, das «am besten» passt, wird dominierendes Pattern genannt. Im Beispiel dominiert das kürzere Pattern String s das längere davor angegebene.

```
static void dominaceExample(Object obj) {  
    switch (obj) {  
        case null -> System.out.println("null");  
        case String s  
        case String s && s.length() > 5 -> System.out.println(s.toLowerCase());  
        case Integer i  
        default -> {}  
    }  
}
```

A screenshot of an IDE showing Java code. A tooltip is displayed over the line 'case String s && s.length() > 5 ->'. The tooltip contains the text: '✖ This case label is dominated by one of the preceding case label' and 'Press 'F2' for focus'. The code uses the new pattern matching syntax introduced in Java 17.

- Mit Java 17 gab das noch keinen Fehler!



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

```
static void dominaceExampleWithConstant(Object obj) {  
    switch (obj.toString()) {  
        case String s && s.length() > 5 -> System.out.println(s.toUpperCase());  
        case "Sophie" -> System.out.println("My lovely daughter");  
        default -> System.out.println("FALLBACK");  
    }  
}
```

A screenshot of an IDE showing a Java code editor. The code is identical to the one above, but the second case statement ('case "Sophie"') is highlighted with a light blue background. A tooltip window appears over the code, containing the text 'This case label is dominated by one of the preceding case label' with a red 'X' icon, and 'Press 'F2' for focus' at the bottom right.

- Korrektur:

```
static void dominaceExampleWithConstant(Object obj) {  
    switch (obj.toString()) {  
        case "Sophie" -> System.out.println("My lovely daughter");  
        case String s && s.length() > 5 -> System.out.println(s.toUpperCase());  
        default -> System.out.println("FALLBACK");  
    }  
}
```



- Bug Fix im Bereich der Vollständigkeitsanalyse = der Prüfung, ob alle möglichen Pfade von den cases im switch abgedeckt werden
- mitunter fälschlicherweise 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")  
    }  
}
```



JEP 420: Pattern Matching bei switch: Vollständigkeitsanalyse



- Bug Fix im Bereich der Vollständigkeitsanalyse = der Prüfung, ob alle möglichen Pfade von den cases im switch abgedeckt werden

```
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



JEP 421: Deprecate Finalization for Removal



JEP 421: Deprecate Finalization for Removal



- Ursprünglich sollte die Finalization als Sicherheitsnetz dienen und Aufräumarbeiten oder Ressourcenfreigaben beim Ableben eines Objekts ermöglichen.
- Dazu wird während der Garbage Collection der jeweilige Finalizer in Form einer Methode `finalize()` aufgerufen, bevor der Speicher des Objekts zurückgegeben wird.
- Allerdings gibt es dabei diverse Hürden und Schwierigkeiten:
 - nicht definiert, wann der Aufruf genau geschieht (möglicherweise einige Zeit nachdem das Objekt schon längst nicht mehr referenziert wird).
 - das Objekt können wieder in einen aktiven Zustand versetzt wird, quasi wiederbelebt werden.
- Im Rahmen dieses JEPs wird die Finalization als deprecated sowie forRemoval. Zwar bleibt die Finalization standardmäßig noch aktiv, lässt sich nun aber auch deaktivieren.
- Die Hintergründe diesen JEPs beleuchtet Nicolai Parlog ausführlich online unter <https://www.youtube.com/watch?v=eDgBnjOid-g>



Other Deprecations



Other Deprecations



- Durch den JEP 421 sind bereits alle `finalize()`-Methoden deprecated.
- In der Klasse `Runtime` wurden diverse überladene `exec()`-Methoden deprecated. Ebenso die Methode `runFinalization()`.
- In der Klasse `Thread` ist nun die Methode `stop()` nicht nur deprecated, sondern `forRemoval=true`.



Fazit



Positives



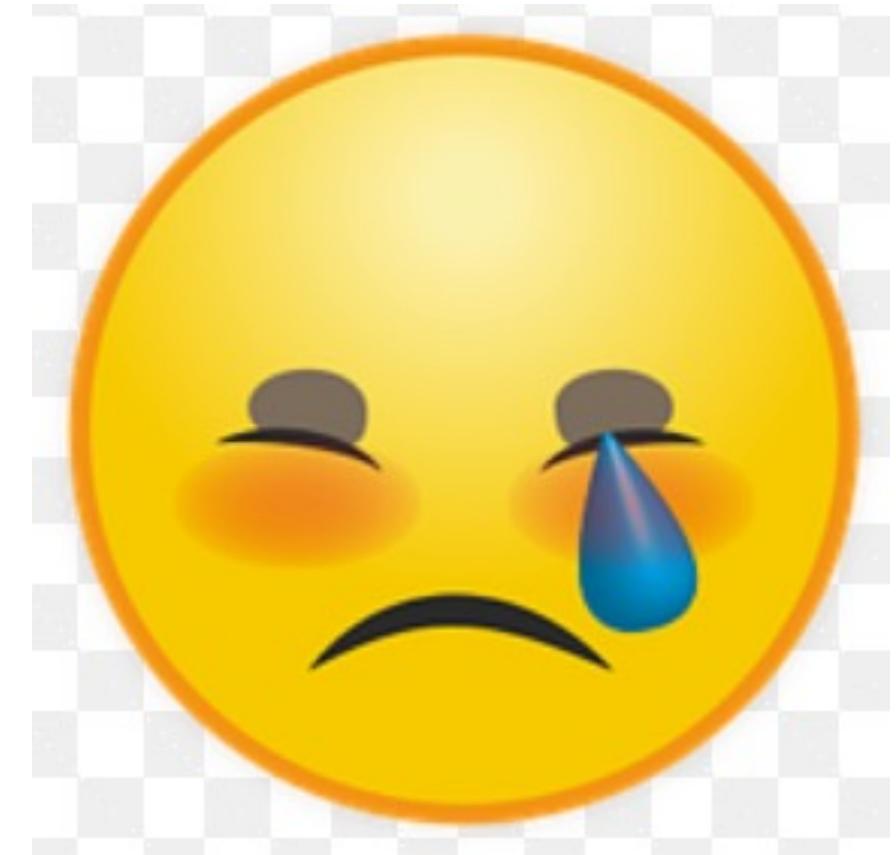
- eine paar Dinge aus Project COIN
- Switch / Records
- diverse praktische Erweiterungen in den APIs
- JPackage
- HTTP 2 (Java 11)
- Modularisierung mit Project JIGSAW (Java 9)--
aber leider (immer noch) kein wirklich gutes Tooling



Negatives



- **Mehrmalige Verschiebung von Java 9
Sept. 2016 => März 2017 => Juli 2017 => Sept. 2017**
- **Folge-Release waren zwar pünktlich, aber manchmal (außer Java 14) ziemlich dünn bezüglich wichtigen Neuerungen**
- **Immer Mal wieder weniger als geplant**
 - **keine Versionierung bei JIGSAW**
 - **kein JSON-Support**
 - **statt Collection-Literals nur Convenience-Methods**
 - **Statt ZIP nur TEE (ing)-Kollektor**







Questions?



Thank You