
Best of Java 9 bis 13

<https://github.com/Michaeli71/JUGS-Best-of-Java9-13.git>

Michael Inden
CTO@ASMIQ AG



- Michael Inden, Jahrgang 1971
- Diplom-Informatiker, C.v.O. Uni Oldenburg
- ~8 ¼ Jahre bei Heidelberger Druckmaschinen AG in Kiel
- ~6 ¾ Jahre bei IVU Traffic Technologies AG in Aachen
- ~4 ¼ Jahre bei Zühlke Engineering AG in Zürich
- Seit Juni 2017 bei **Direct Mail Informatics / ASMIQ** in Zürich
- Autor und Gutachter beim dpunkt.verlag

E-Mail: michael.inden@asmiq.ch

Kursangebot: <https://asmiq.ch/>

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

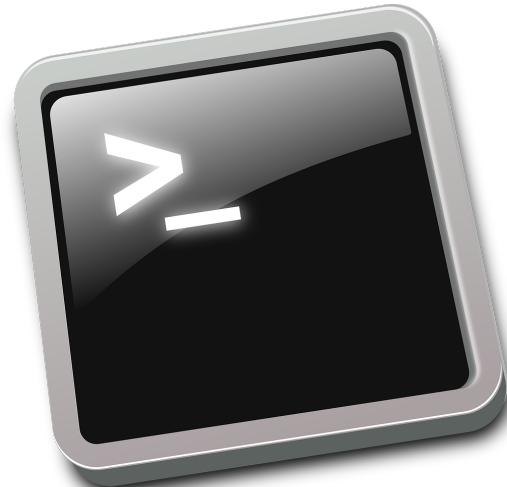


Agenda

- **PART 1:** Syntax-Erweiterungen in Java 9 bis 11
 - **PART 2:** API-Neuheiten und Änderungen in Java 9 bis 11
-
- **PART 3:** Multi-Threading mit CompletableFuture
 - **PART 4:** HTTP/2
 - **PART 5:** Neuerungen in Java 13

JDK	Release-Datum	Entwicklungszeit	LTS	Workshop
Oracle JDK 8	3 / 2014	-	Ja, <i>mittlerweile auch kommerziell</i>	-/-
Oracle JDK 9	9 / 2017	3,5 Jahre	-	Part 1, 2, 3
Oracle JDK 10	3 / 2018	6 Monate	-	Part 1, 2, 3
Oracle JDK 11	9 / 2018	6 Monate	Ja, <i>kommerziell</i>	Part 1, 2, 3
Oracle JDK 12	3 / 2019	6 Monate	-	Part 4
Oracle JDK 13	9 / 2019	6 Monate	-	Part 5

Neues Lizenzmodell



- **Sofern Sie planen, Ihre Software kommerziell vertreiben (wollen), sollten Sie beim Herunterladen von Java 11 unbedingt die neue Release-Politik von Oracle beachten!**
- **Das Oracle JDK ist nun leider für einige Szenarien kostenpflichtig – während der Entwicklung kann es allerdings weiterhin kostenfrei nutzen.**
- **Als Alternative können Sie auf das OpenJDK (<https://openjdk.java.net/>)**

Java SE Development Kit 11 Downloads

Thank you for downloading this release of the Java™ Platform, Standard Edition Development Kit (JDK™). The JDK is a development environment for building applications, and components using the Java programming language.

The JDK includes tools useful for developing and testing programs written in the Java programming language and running on the Java platform.

Important changes in Oracle JDK 11 License

With JDK 11 Oracle has updated the license terms on which we offer the Oracle JDK.

The new [Oracle Technology Network License Agreement for Oracle Java SE](#) is substantially different from the licenses under which previous versions of the JDK were offered. Please review the new terms carefully before downloading and using this product.

Oracle also offers this software under the [GPL License](#) on jdk.java.net/11

PART 1: Syntax-Erweiterungen in Java 9 bis 11

Syntax-Erweiterungen in Java 9



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

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



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

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

- `_` ist **kein** valider Bezeichner mehr (semantisch war er es aber eh nie ;-))
- `final String _ = "Underline";`

```
MyClass.java:2: error: as of release 9, '_' is a keyword, and may not be  
used as an identifier
```

```
    static Object _ = new Object();  
          ^
```

FORGET ANYTHING YOU KNOW ABOUT...



JAVA INTERFACES!

```
public interface PrivateMethodsExample
{
    public abstract int method1();

    public default int calc(int a, int b) {
        return myCalc(a, b);
    }

    public default int calc2(int a, int b) {
        return myCalc(a, b);
    }

    private int myCalc(int a, int b) {
        return a + b;
    }
}
```

Syntax-Erweiterung in JDK 10 / 11



- 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

- 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);
```

- 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())));
```

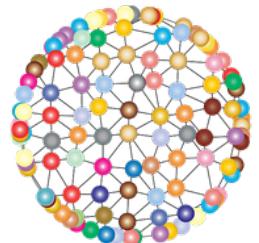
- Dazu nutzen wir diese Lambdas:

```
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.**

- 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:

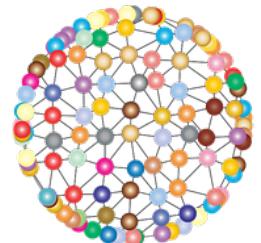
```
// var => ArrayList<String> und nicht List<String>
var names = new ArrayList<String>();
names = new LinkedList<String>(); // error
```

- 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?**

- **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);
```

- Java 10: In Lambda kein var möglich, nur folgende Varianten:

```
IntFunction<Integer> doubleItTyped = (final int x) -> x * 2;  
IntFunction<Integer> doubleItNoType = x -> x * 2;
```

- Java 11: Nun auch var in Lambda erlaubt

```
IntFunction<Integer> doubleItTyped = (var x) -> x * 2;
```

- Was ist der Vorteil?

```
Function<String, String> trimmer = (@SomeAnnotation var str) -> str.trim();
```

Neuheiten und Änderungen in Java 9 bis 11

- Process API
- Stream-API
- Optional<T>
- Collection Factory Methods
- Date API
- InputStream / Reader
- Strings (JDK 11)
- Files (JDK 11)

Process API



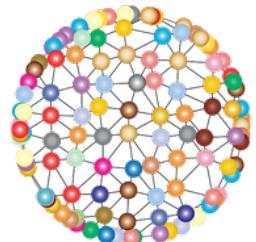
- Bisher nur begrenzte Kontrolle und Verwaltung von Betriebssystemprozessen
- Beispiel PID auslesen: Oftmals für jede Plattform eine andere Implementierung

```
private static long getPidOldStyle() throws InterruptedException, IOException
{
    final Process proc = Runtime.getRuntime().exec(new String[]{ "/bin/sh", "-c", "echo $PPID" });
    if (proc.waitFor() == 0)
    {
        final InputStream in = proc.getInputStream();
        final byte[] outputBytes = new byte[in.available()];

        in.read(outputBytes);
        final String pid = new String(outputBytes);
        return Long.parseLong(pid.trim());
    }
    throw new IllegalStateException("PID is not accessible");
}
```



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



```
long pid = ProcessHandle.current().getPid();
```

Neben der PID kann man mithilfe von ProcessHandle noch diverse weitere Informationen zu Prozessen auslesen. Dazu gibt es unter anderem folgende Methoden:

- **current()** – Ermittelt den aktuellen Prozess als ProcessHandle.
- **info()** – Stellt Infos zum Prozess in Form des inneren Interface ProcessHandle.Info bereit, etwa zu Benutzer, Kommando usw.
- **info().command()** – Liefert das Kommando als Optional<String>.
- **info().user()** – Gibt den Benutzer als Optional<String> zurück
- **info().totalCpuDuration()** – Ermittelt aus den Infos die benötigte CPU-Zeit.

Neben Informationen zum aktuellen Prozess lassen sich Informationen für alle Prozesse des Benutzers sowie alle Subprozesse zu einem Prozess wie folgt ermitteln:

- **allProcesses()** – Liefert alle Prozesse als Stream<ProcessHandle>.
- **children()** – Ermittelt zu einem Prozess alle seine (direkten) Subprozesse als Stream<ProcessHandle>.
- **descendants()** – Ermittelt zu einem Prozess alle seine Subprozesse als Stream<ProcessHandle>.

Stream API



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

takeWhile(...)

dropWhile(...)

ofNullable(...)

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

takeWhile(...)

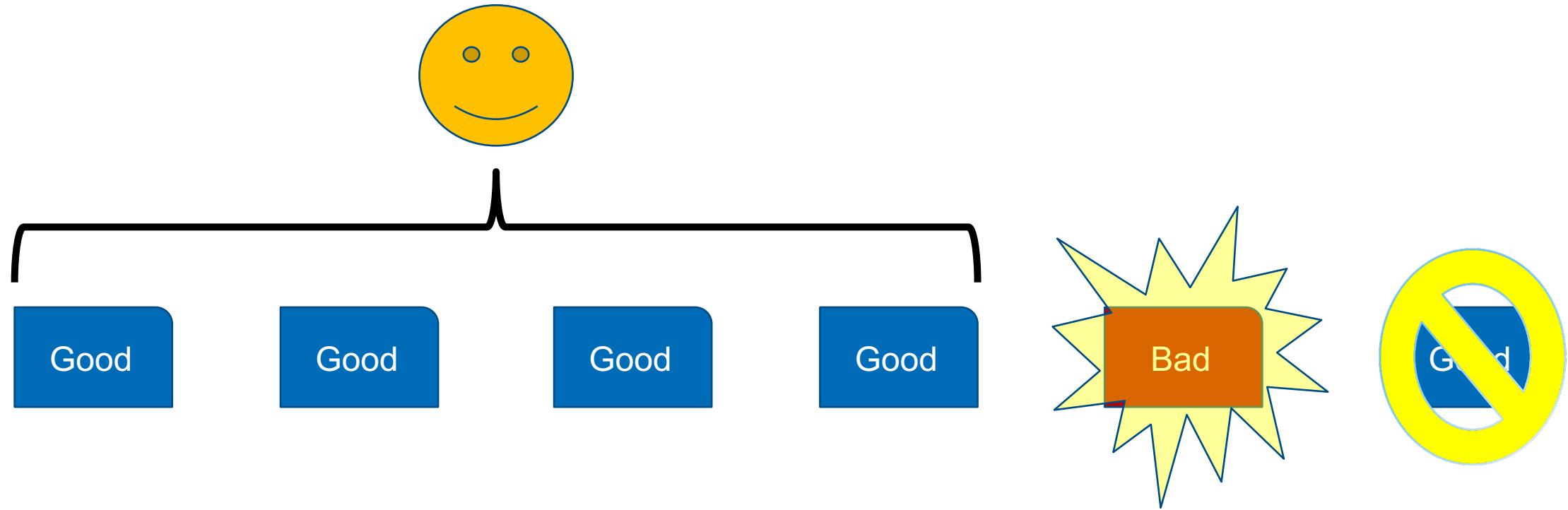
dropWhile(...)

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

ofNullable(...)

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

Stream – Product Scenario



```
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

takeWhile(...)

dropWhile(...)

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

ofNullable(...)

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

```
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);  
    }  
}
```

```
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 zur Extraktion von Daten:

```
public static void main(String[] args)
{
    Stream<String> words = Stream.of("ab", "BLA", "Xy", "<START>",
                                      "WELCOME", "T0", "JUGS", "Luzern",
                                      "<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
JUGS
Luzern

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

- **Gutes API, aber 3 Schwachstellen bei folgende 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,**

▼ 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

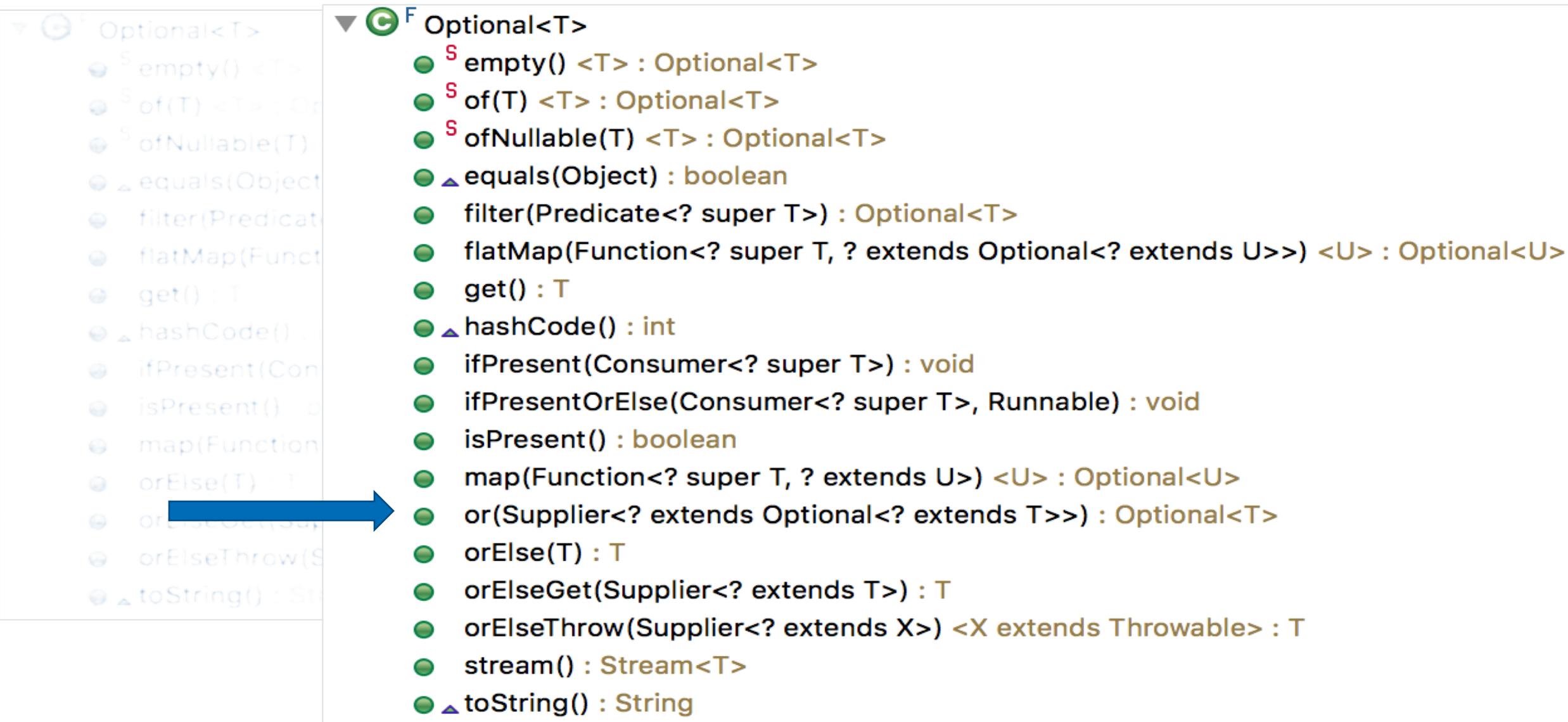
The screenshot shows the JavaDoc for the `Optional<T>` class. On the left, a list of methods from an older version of the class is shown, and on the right, a more complete list is shown, with a blue arrow pointing from the old list to the new `isPresent()` method.

Old Methods (Left):

- empty() <T>
- of(T) <T> : Optional<T>
- ofNullable(T)
- equals(Object)
- filter(Predicate)
- flatMap(Function)
- get() : T
- hashCode()
- ifPresent(Consumer)
- isPresent()
- map(Function)
- orElse(T) : T
- orElseGet(Supplier)
- orElseThrow(Supplier<? extends Throwable> : T)
- toString() : String

New Methods (Right):

- empty() <T> : Optional<T>
- of(T) <T> : Optional<T>
- 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



The screenshot shows the JavaDoc for the `Optional<T>` class, comparing its methods between Java 8 and Java 9.

JDK 8 Methods:

- empty() <T>
- of(T) <T> : Optional<T>
- ofNullable(T)
- equals(Object)
- filter(Predicate)
- flatMap(Function)
- get() : T
- hashCode()
- ifPresent(Consumer)
- isPresent() : boolean
- map(Function)
- orElse(T) : T
- orElseGet(Supplier)
- orElseThrow(Supplier)
- toString() : String

JDK 9 Methods:

- empty() <T> : Optional<T>
- of(T) <T> : Optional<T>
- 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

A large blue arrow points from the JDK 8 methods section to the JDK 9 methods section, indicating the addition of new methods in Java 9.

```
▼ G F Optional<T>
  • S empty() <T>
  • S of(T) <T> : Opt
  • S ofNullable(T)
  • ▲ equals(Object)
  • filter(Predicate)
  • flatMap(Funct
  • get() : T
  • hashCode()
  • ifPresent(Con
  • isPresent() : b
  • map(Function)
  • orElse(T) : T
  • orElseGet(Sup
  • orElseThrow(S
  • ▲ toString() : St
```

```
▼ 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
```



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(...) ?

```
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");  
    }  
}
```

JDK 8

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");  
    }  
  
}
```

▼ 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



```
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

-

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

▼ 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

Optional<T> in JDK 10 & 11



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

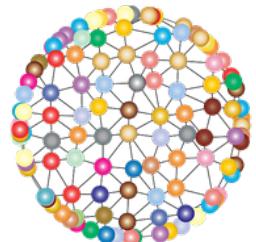
▼   **Optional<T>**



-   `empty() <T> : Optional<T>`
-   `of(T) <T> : Optional<T>`
-   `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 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

- 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
- 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



- 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 ...



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

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



```
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 Literals **LIGHT** a.k.a 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
```

- **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)
```

Date API Erweiterungen



- **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 = birthday.datesUntil(christmas);
    daysUntil.skip(150).limit(4).forEach(System.out::println);

    System.out.println("\nMonth-Stream");
    final Stream<LocalDate> monthsUntil =
        birthday.datesUntil(christmas, Period.ofMonths(1));
    monthsUntil.limit(3).forEach(System.out::println);
}
```

- 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

Month-Stream

1971-02-07

1971-03-07

1971-04-07

java.io.InputStream / Reader



- 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`

- **`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 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()`

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

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

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

- Alle geben true aus.
-

- 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!
```

- 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));

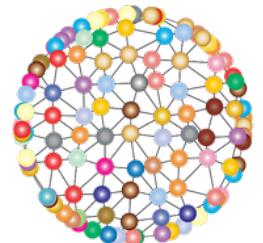
    final String delimiter = " -* ";
    System.out.println(delimiter.repeat(6));
}

=>

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

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

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



Was passiert?

```
"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`: Repeating 5 bytes String 2147483647 times will
produce a String exceeding maximum size.
at `java.base/java.lang.String.repeat(String.java:3164)`
at `Java11Examples/snippet.Snippet.main(Snippet.java:14)`

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

Exception in thread "main"

```
java.lang.OutOfMemoryError: Repeating 5 bytes String 2147483647 times will
produce a String exceeding maximum size.
at java.base/java.lang.String.repeat(String.java:3164)
at Java11Examples/snippet.Snippet.main(Snippet.java:14)
```

```
if (Integer.MAX_VALUE / count < len)
{
    throw new OutOfMemoryError("Repeating " + len + " bytes String " + count +
                               " times will produce a String exceeding maximum size.");
}
```

- 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'
```

Erweiterung in der Klasse 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 destDath = Path.of("ExampleFile.txt");
```

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

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

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

=>

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

- 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 destDath = Path.of("ExampleFile.txt");
```

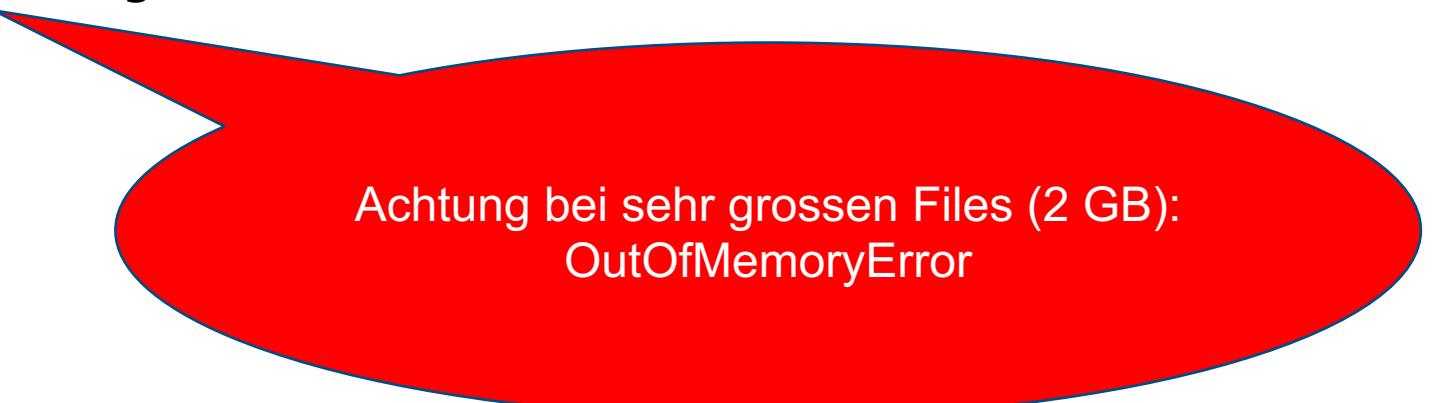
```
Files.writeString(destDath, "1: This is a string to file test\n");
Files.writeString(destDath, "2: Second line");
```

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

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

=>

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



Achtung bei sehr grossen Files (2 GB):
OutOfMemoryError

- **Korrektur 1:** APPEND-Mode

```
Files.writeString(destDath, "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(destDath);  
content.lines().forEach(System.out::println);
```

=>

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

Übungen PART 1 & 2

<https://github.com/Michaeli71/JUGS-Best-of-Java9-13.git>

PART 3:

Multi-Threading mit

CompletableFuture

- 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>

- **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"));

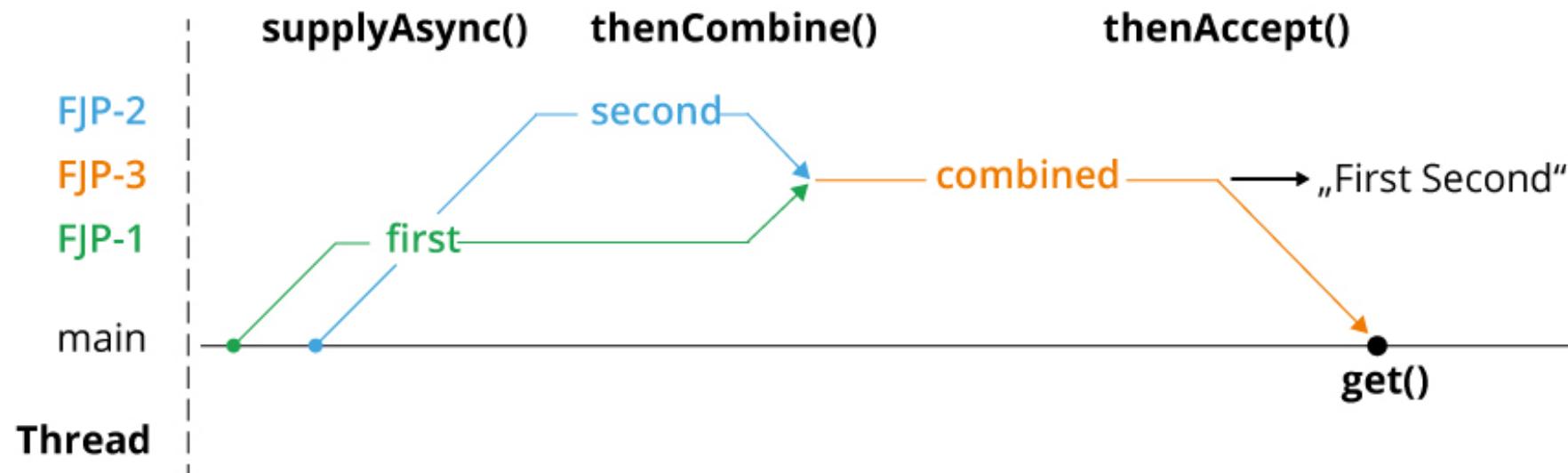
CompletableFuture<String> combined = firstTask.thenCombine(secondTask,
    (f, s) -> f + " " + s);

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

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

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

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

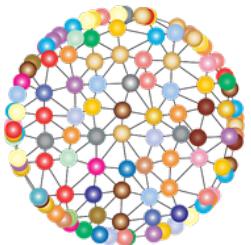


Beispiel: Es sollen folgende Aktionen stattfinden:

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



**Wie könnte eine erste
Realisierung aussehen?**



- 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);
}
```

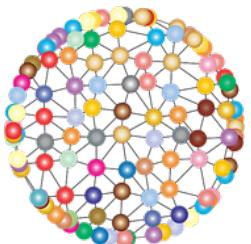
```
public void execute()
{
    final List<String> data = retrieveData();
    final Long value1 = processData1(data);
    final Long value2 = processData2(data);
    final Long value3 = processData3(data);
    final String result = calcResult(value1, value2, value3);
    System.out.println("result: "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

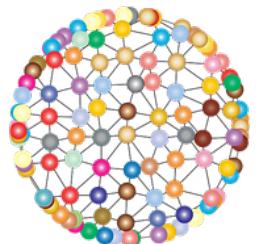


Multi-Threading und die Klasse CompletableFuture<T>

```
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?**

- 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

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

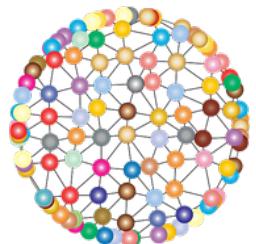
- Fallback-Wert bereitstellen, wenn die Daten nicht ermittelt werden können:

```
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?**

- 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!**

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.

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

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

Part 4: 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**

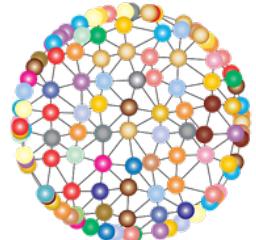
```
public static String readContent(final InputStream is) throws IOException
{
    try (final InputStreamReader isr = new InputStreamReader(is);
         final BufferedReader br = new BufferedReader(isr))
    {
        final StringBuilder content = new StringBuilder();

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

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



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



```
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);

final int responseCode = response.statusCode();
final String responseBody = response.body();

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

```
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);

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

- **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())
```

PART 5: Neuheiten und Änderungen in Java 13

- Build-Tools und IDEs
 - Syntaxerweiterungen bei switch
 - Syntaxerweiterung Text Blocks
-

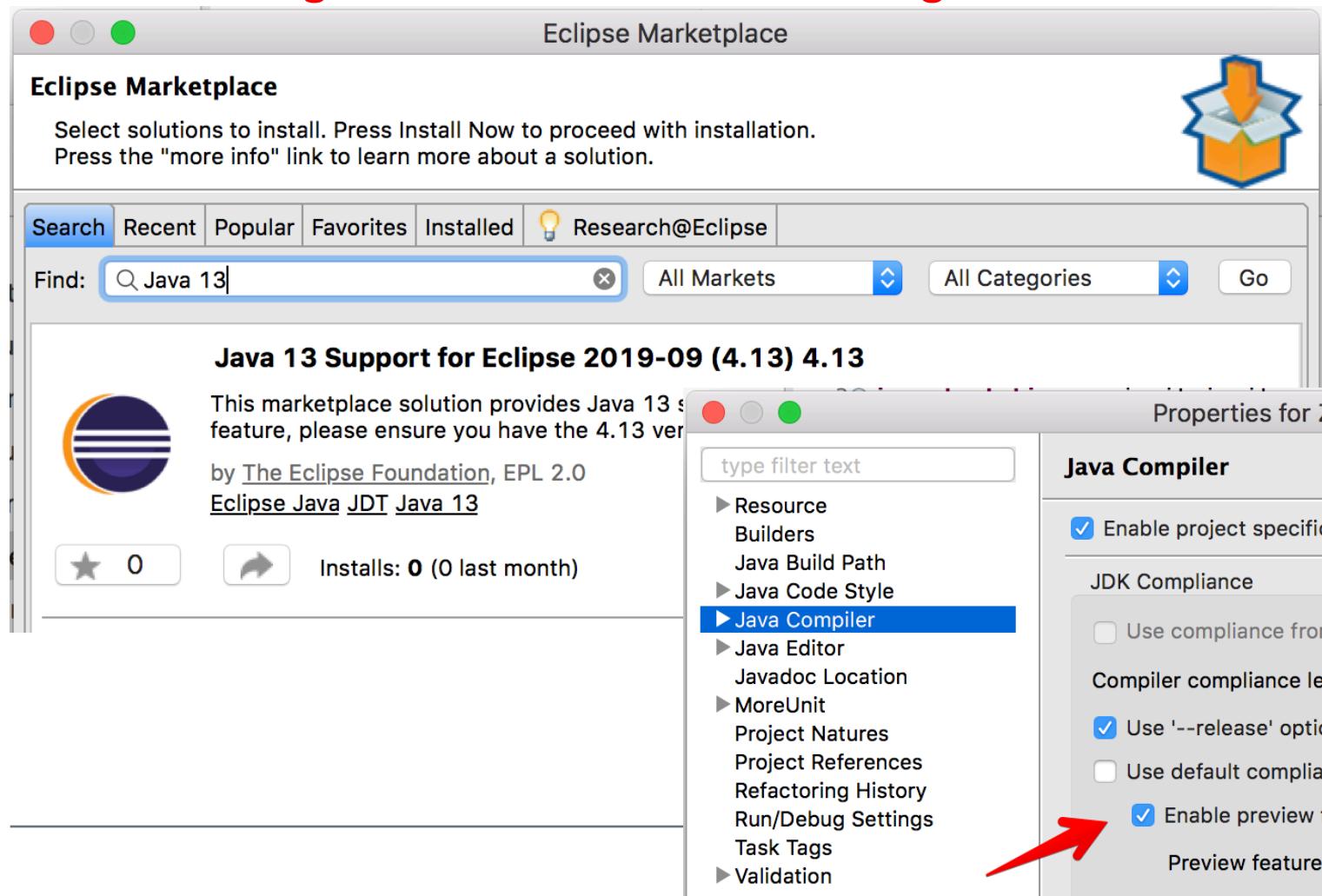
Build-Tools und IDEs



- Aktuelle IDEs & Tools grundsätzlich gut
- Eclipse: Version 2019-09 mit Plugin
- IntelliJ: Version 2019.2.3
- Maven: 3.6.2, Compiler-Plugin: 3.8.1
- Gradle: 5.6.3 😊 // Gradle 6.0
- Aktivierung von Preview-Features nötig
 - In Dialogen
 - Im Build-Skript



- Eclipse 2019-09: Installation von speziellem Plugin nötig
- Aktivierung von Preview-Features nötig



Eclipse Marketplace

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

Search Recent Popular Favorites Installed Research@Eclipse

Find: Java 13 All Markets All Categories Go

Java 13 Support for Eclipse 2019-09 (4.13) 4.13

This marketplace solution provides Java 13 support. Please note that this feature, please ensure you have the 4.13 version of Eclipse installed.

by The Eclipse Foundation, EPL 2.0
[Eclipse Java JDT Java 13](#)

0 installs: 0 (0 last month)

type filter text

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

Properties for ZZZZZZ_Oracle-Code-One

Java Compiler

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

JDK Compliance

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

Compiler compliance level:

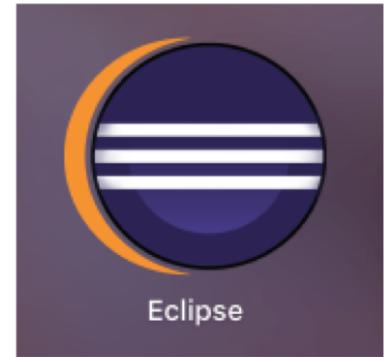
Use '--release' option

Use default compliance settings

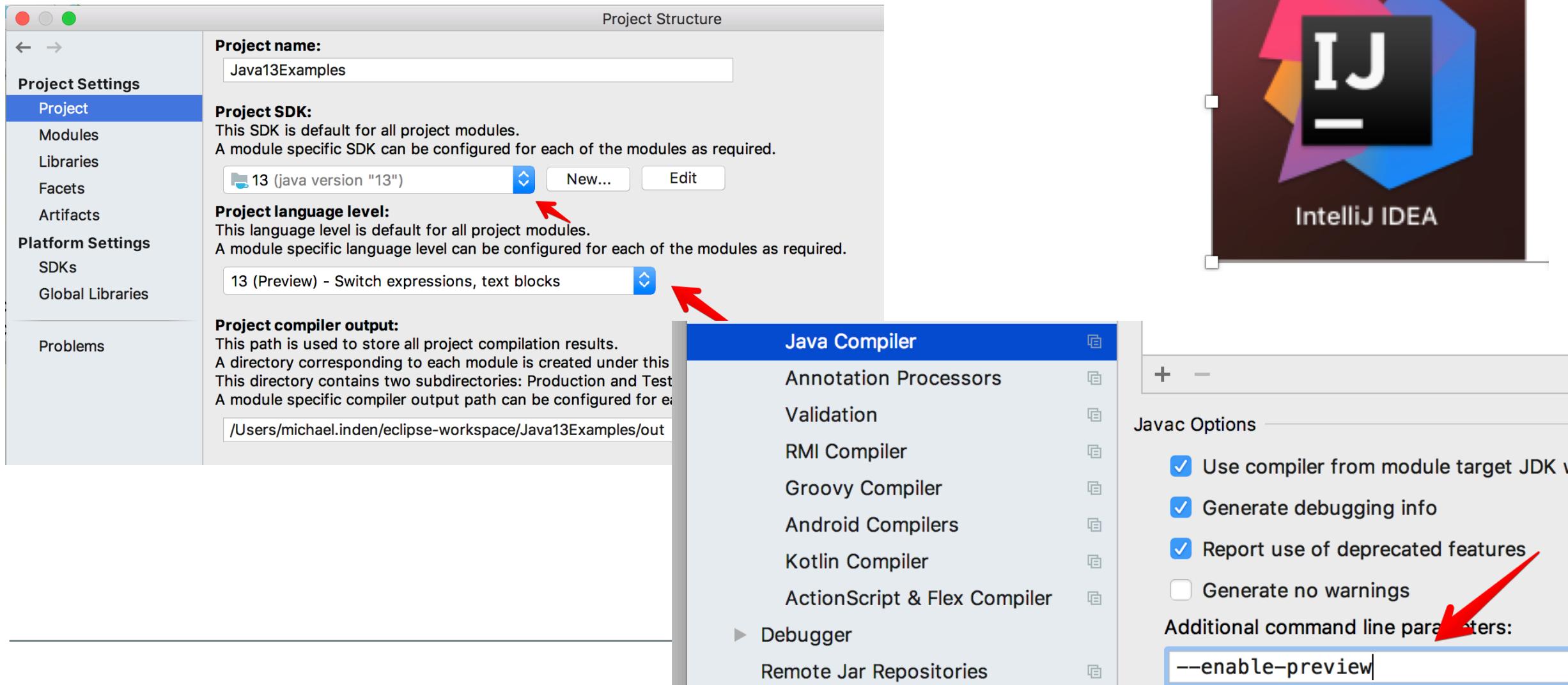
Enable preview features for Java 13

Preview features with severity level: 13

Warning



- Aktivierung von Preview-Features nötig



The screenshot shows the IntelliJ IDEA Project Structure dialog. The left sidebar has 'Project Settings' selected. In the main area, under 'Project SDK', '13 (java version "13")' is selected. Under 'Project language level', '13 (Preview) - Switch expressions, text blocks' is selected. Red arrows point from the text 'Switch expressions, text blocks' to both the 'Project language level' dropdown and the 'Java Compiler' section below. The 'Java Compiler' section lists various compiler components, and the 'Additional command line parameters' field at the bottom contains the value '--enable-preview'. To the right of the dialog, the IntelliJ IDEA logo is displayed.

- Aktivierung von Preview-Features nötig

```
sourceCompatibility=13  
targetCompatibility=13
```

```
// 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>13</source>
      <target>13</target>
      <!-- Wichtig für Java 12/13 Syntax-Neuerungen -->
      <compilerArgs>--enable-preview</compilerArgs>
    </configuration>
  </plugin>
</plugins>
```



Syntax-Erweiterungen



- **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:**

- Abbildung von Wochentagen auf deren Länge ...

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

- Abbildung von Wochentagen auf deren Länge ... elegant mit Java 13:

```
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;  
};
```



- Abbildung von Wochentagen auf deren Länge ... elegant mit Java 13:

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;  
};
```



- Abbildung von Wochentagen auf deren Länge ... elegant mit Java 13:

```
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

- 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
```

- Abbildung von Monaten auf deren Namen ... elegant mit Java 13:

```
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";
    };
}
```

- 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;
}
```

Mit Java 13 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);
}
```

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";
```

- NEU

```
String javascriptCode = """  
    void print(Object o)  
    {  
        System.out.println(Objects.toString(o));  
    }  
""";
```

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

- <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>  
""";
```

- NEU

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

```
String multiLineStringWithPlaceHolders = """  
    SELECT %s  
    FROM %s  
    WHERE %s  
""" .formatted(new Object[]{"A", "B", "C"});
```

- NEU

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

Übungen PART 3 & 4 & 5

Questions?



Thank You
