
Best of Java 9 bis 13

<https://github.com/Michaeli71/WJAX-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>

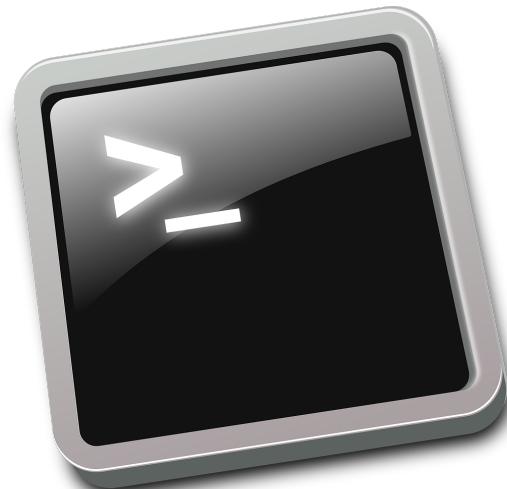


Agenda

- **PART 1:** Syntax-Erweiterungen & Neuheiten und Änderungen in Java 9 bis 11
 - **PART 2:** Multi-Threading und Reactive Streams
 - **PART 3:** Weitere Neuheiten und Änderungen in JDK 9 bis 11
-
- **PART 4:** Neuerungen in Java 12
 - **PART 5:** Neuerungen in Java 13
-
- **Separat:** Modularisierung

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 und API- Änderungen 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())));
```

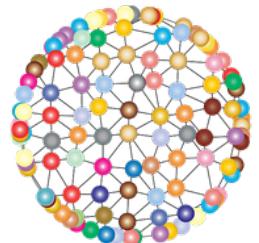
- Dazu nutzen wir diese Lambdas:

```
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:

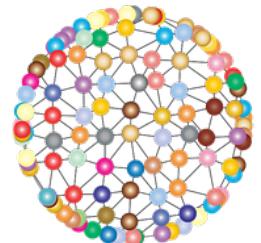
```
// var => ArrayList<String> und nicht List<String>
var names = new ArrayList<String>();
names = new LinkedList<String>(); // error
```

- 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
- Strings (JDK 11)

Process API



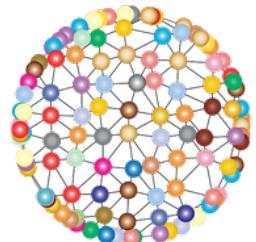
- Bisher nur begrenzte Kontrolle und Verwaltung von Betriebssystemprozessen
- Beispiel PID auslesen: Oftmals für jede Plattform eine andere Implementierung

```
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(...)

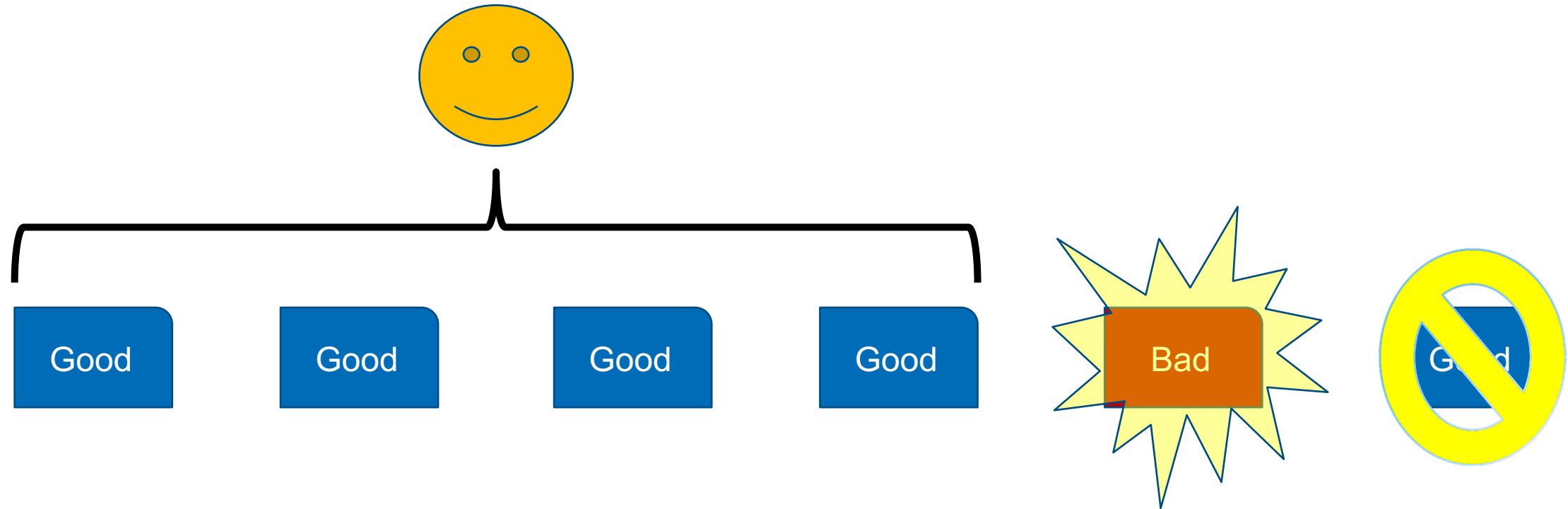
dropWhile(...)

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

ofNullable(...)

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

Stream – Product Scenario



Stream – Filter

```
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);  
    }  
}
```

JDK8

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

Stream – Filter

```
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);  
    }  
}
```

JDK8

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



```
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", "W-JAX", "München",
                                      "<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
W-JAX
München

takeWhile(...)

dropWhile(...)

ofNullable(...)

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

`ofNullable(T)` – Liefert einen Stream<T> mit einem Element, sofern das übergebene Element ungleich null ist. Ansonsten wird ein leerer Stream erzeugt.

```
public class FirstNullableExample
{
    public static void main(final String[] args)
    {
        final Stream<Customer> customerStream1 = Stream.of(findCustomer());
        System.out.println(customerStream1.count());

        final Stream<Customer> customerStream2 = Stream.of(findCustomer());
        final Optional<Customer> optFirst = customerStream2.findFirst();
        optFirst.ifPresentOrElse(System.out::println,
                               () -> System.out.println("No element"));
    }

    private static Customer findCustomer()
    {
        // complex logic for search with fallback ...
        return null;
    }
}
```

Please note: returning null is often considered bad style,
but it is not uncommon in older (legacy) code!!

```
public class FirstNullableExample
{
    public static void main(final String[] args)
    {
        final Stream<Customer> customerStream1 = Stream.of(findCustomer());
        System.out.println(customerStream1.count());

        final Stream<Customer> customerStream2 = Stream.of(findCustomer());
        final Optional<Customer> optFirst = customerStream2.findFirst();
        optFirst.ifPresentOrElse(System.out::println,
                               () -> System.out.println("No element"));
    }

    private static Customer findCustomer()
    {
        return null;
    }
}
```

```
<terminated> FirstNullableExample [Java Application] /Library/Java/JavaVirtualMachines/jd
1
Exception in thread "main" java.lang.NullPointerException
at java.base/java.util.Objects.requireNonNull(Objects.java:324)
at java.base/java.util.Optional.of(Optional.java:111)
at java.base/java.util.stream.FindOps$FindSink$OfRef.get(FindOps.java:111)
```

```
public class NullableExample
{
    public static void main(final String[] args)
    {
        final Stream<Customer> customerStream1 = Stream.ofNullable(findCustomer());
        System.out.println(customerStream1.count());

        final Stream<Customer> customerStream2 = Stream.ofNullable(findCustomer());
        final Optional<Customer> optFirst = customerStream2.findFirst();
        optFirst.ifPresentOrElse(System.out::println,
                               () -> System.out.println("No element"));
    }

    private static Customer findCustomer()
    {
        return null;
    }
}
```

```
public class NullableExample
{
    public static void main(final String[] args)
    {
        final Stream<Customer> customerStream1 = Stream.ofNullable(findCustomer());
        System.out.println(customerStream1.count());

        final Stream<Customer> customerStream2 = Stream.ofNullable(findCustomer());
        final Optional<Customer> optFirst = customerStream2.findFirst();
        optFirst.ifPresentOrElse(System.out::println,
                               () -> System.out.println("No element"));
    }

    private static Customer findCustomer()
    {
        return null;
    }
}
```

<terminated> NullableExample (1) [Java Application] /Library/Java/JavaVirtualMachine

0

No element



takeWhile(...)

dropWhile(...)

ofNullable(...)

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

- **iterate(T, Predicate<? super T>, UnaryOperator<T>)** – Erzeugt einen Stream<T> mit mit dem übergebenen Startwert. Die folgenden Werte werden durch den UnaryOperator<T> berechnet, solange das übergebene Predicate<T> erfüllt ist.

```
final IntStream stream = IntStream.iterate(1, n -> n < 10, n -> n + 1);  
System.out.println(stream.mapToObj(num -> "" + num).collect(joining(", ")));
```

- => 1, 2, 3, 4, 5, 6, 7, 8, 9
- Angaben ziemlich analog zur for-Schleife:
`for (int n = 1; n < 10; n++)
 iterate(1, n -> n < 10, n -> n + 1);`
- Da es ein Stream ist, aber mit einer Vielzahl weiterer Möglichkeiten

Stream API – 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
- **flatMapting()** – 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()`

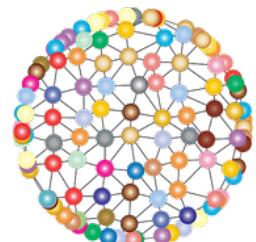
Beispiel zum Einstieg:

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}
```

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("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], Peter=[Java, Movies, Groovy, Skating]}

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

▼ 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

- **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,**

- 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.

▼ 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

JDK 8

```
public class CityPrinter {  
    public static void main(String[] args) {  
        Stream<Optional<String>> optionalCityNames = Stream.of(Optional.of("Zurich"), Optional.of("Bern"),  
                                                               Optional.of("Basel"), Optional.empty());  
 optionalCityNames.filter(Optional::isPresent).map(Optional::get).forEach(System.out::println);  
    }  
}
```

- `Optional<T> => Stream<T>`: Das ist hilfreich, wenn man einen Stream von optionalen Werten hat und dort **nur diejenigen Einträge mit gültigen Werten behalten** werden sollen (Kombination der Methoden `flatMap()` und `stream()`).
- Beispiel eines **Streams**, der aus `Optional<String>-Elementen` besteht, etwa als Folge einer parallelen Suche. Am Ende sollen die **Ergebnisse konsolidiert** werden:

JDK 9

```
public class CityPrinter {

    public static void main(String[] args) {

        Stream<Optional<String>> optionalCityNames = Stream.of(Optional.of("Zurich"), Optional.of("Bern"),
                                                               Optional.of("Basel"), Optional.empty());

        optionalCityNames.flatMap(Optional::stream).forEach(System.out::println);
    }
}
```

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.

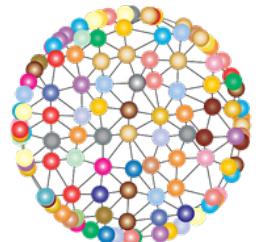
▼   **Optional<T>**



-   `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

- 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
```

- 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)
```

```
static <E> List<E> of() {
    return ImmutableList.of();
}

static <E> List<E> of(E e1) {
    return new ImmutableList.List1<E>(e1);
}

...

@SafeVarargs
@SuppressWarnings("varargs")
static <E> List<E> of(E... elements) {
    switch (elements.length) {
        case 0:
            return new ImmutableList.List0<E>();
        case 1:
            return new ImmutableList.List1<E>(elements[0]);
        case 2:
            return new ImmutableList.List2<E>(elements[0], elements[1]);
        default:
            return new ImmutableList.ListN<E>(elements);
    }
}
```

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));

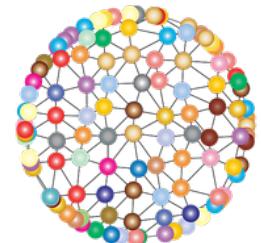
    final String delimiter = " -* ";
    System.out.println(delimiter.repeat(6));
}

=>

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

```
"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'
```

Übungen PART 1

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

PART 2: Multi-Threading und Reactive Streams

- CompletableFuture<T>
 - Flow-API
-

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"));

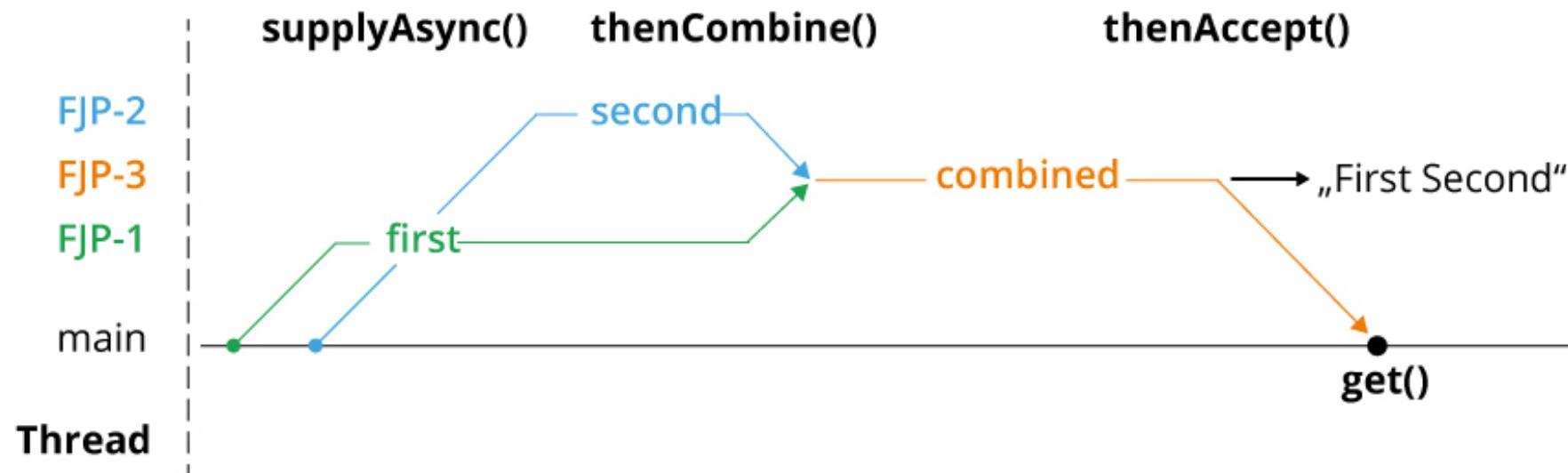
CompletableFuture<String> combined = firstTask.thenCombine(secondTask,
    (f, s) -> f + " " + s);

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());
```

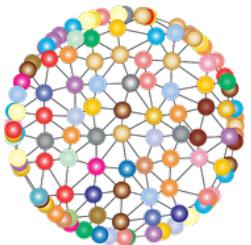


Beispiel: Es sollen folgende Aktionen stattfinden:

- **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);
}
```

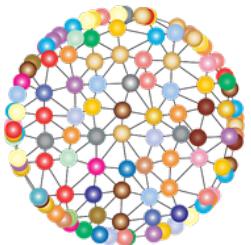
```
public void execute()
{
    final List<String> data = retrieveData();
    final Long value1 = processData1(data);
    final Long value2 = processData2(data);
    final Long value3 = processData3(data);
    final String result = calcResult(value1, value2, value3);
    System.out.println("result: "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

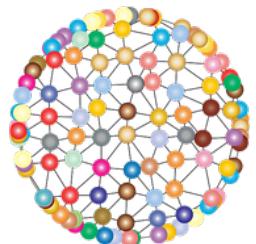


Multi-Threading und die Klasse CompletableFuture<T>

```
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()**

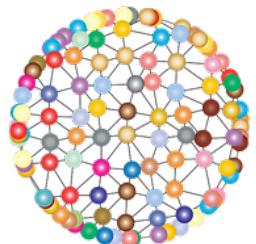
- Fallback-Wert bereitstellen, wenn die Daten nicht ermittelt werden können:

```
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());
```

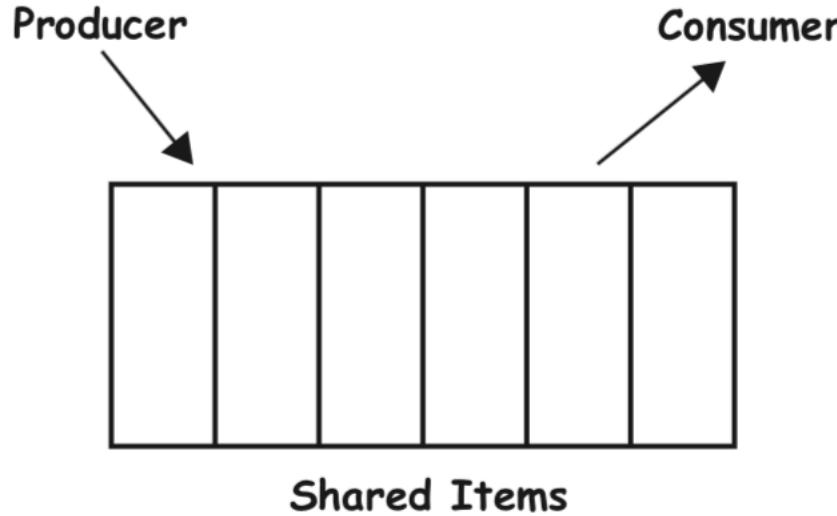
=> Die Verarbeitung kann zeitnah (ohne viel Verzögerung oder gar Blockierung) selbst dann fortgesetzt werden, wenn die Datenermittlung länger dauern sollte

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);
```

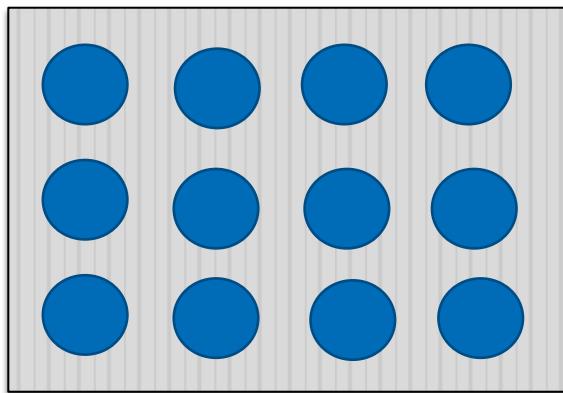
=> Die Berechnung kann mit einem Fallback-Wert fortgesetzt werden, selbst wenn der eine oder andere Schritt länger dauern sollte

Reactive Streams

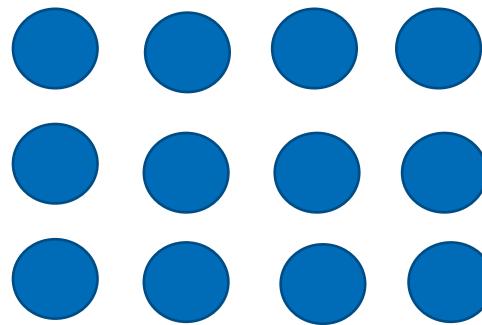


- **Coordination needed**
 - PUSH Producer pushes, Consumer processes directly
 - POLL Consumer polls, Produces creates on demand
 - Buffering needed

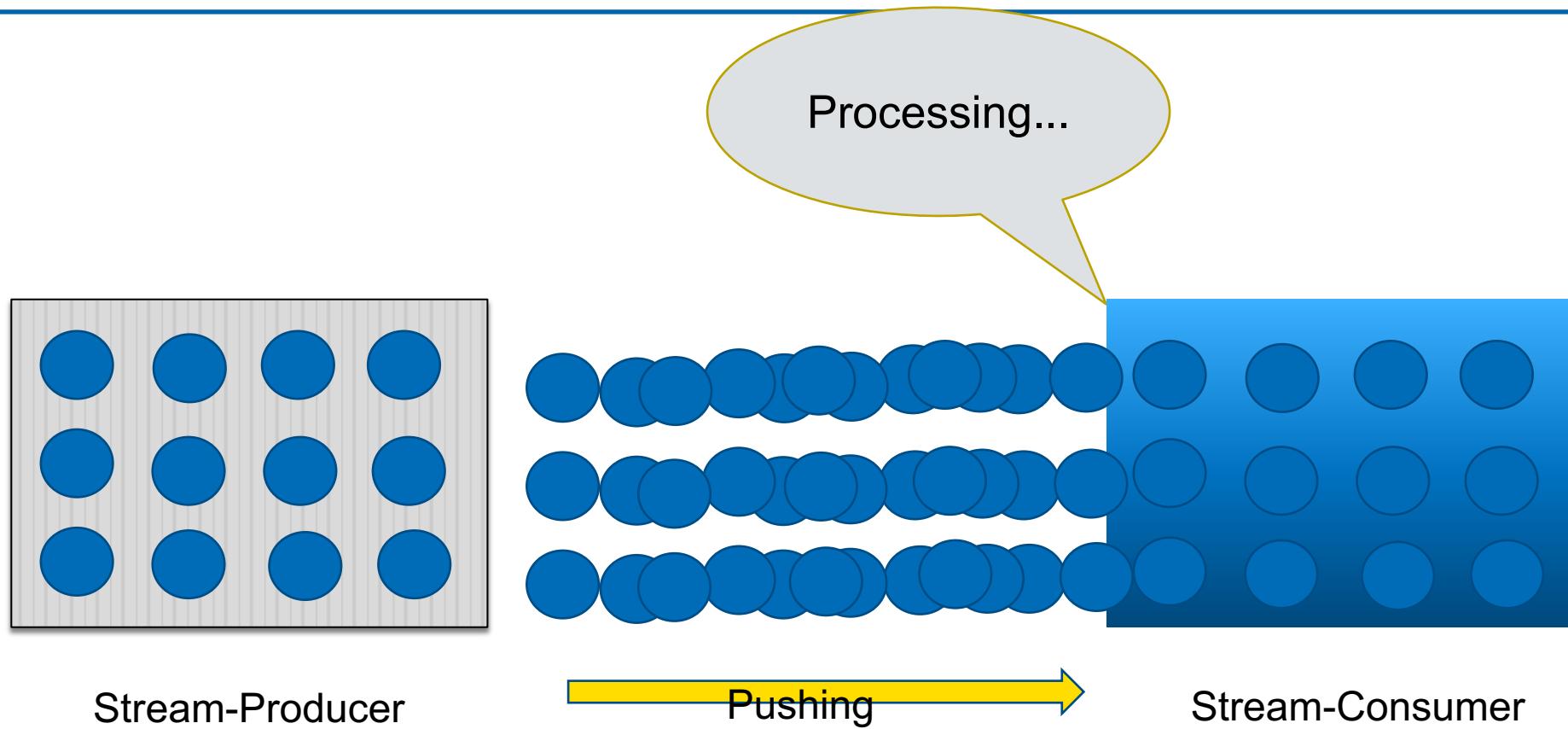
Scenario 1 - Push

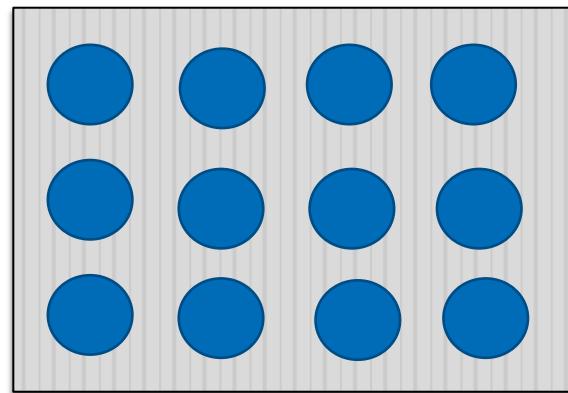


Stream-Producer

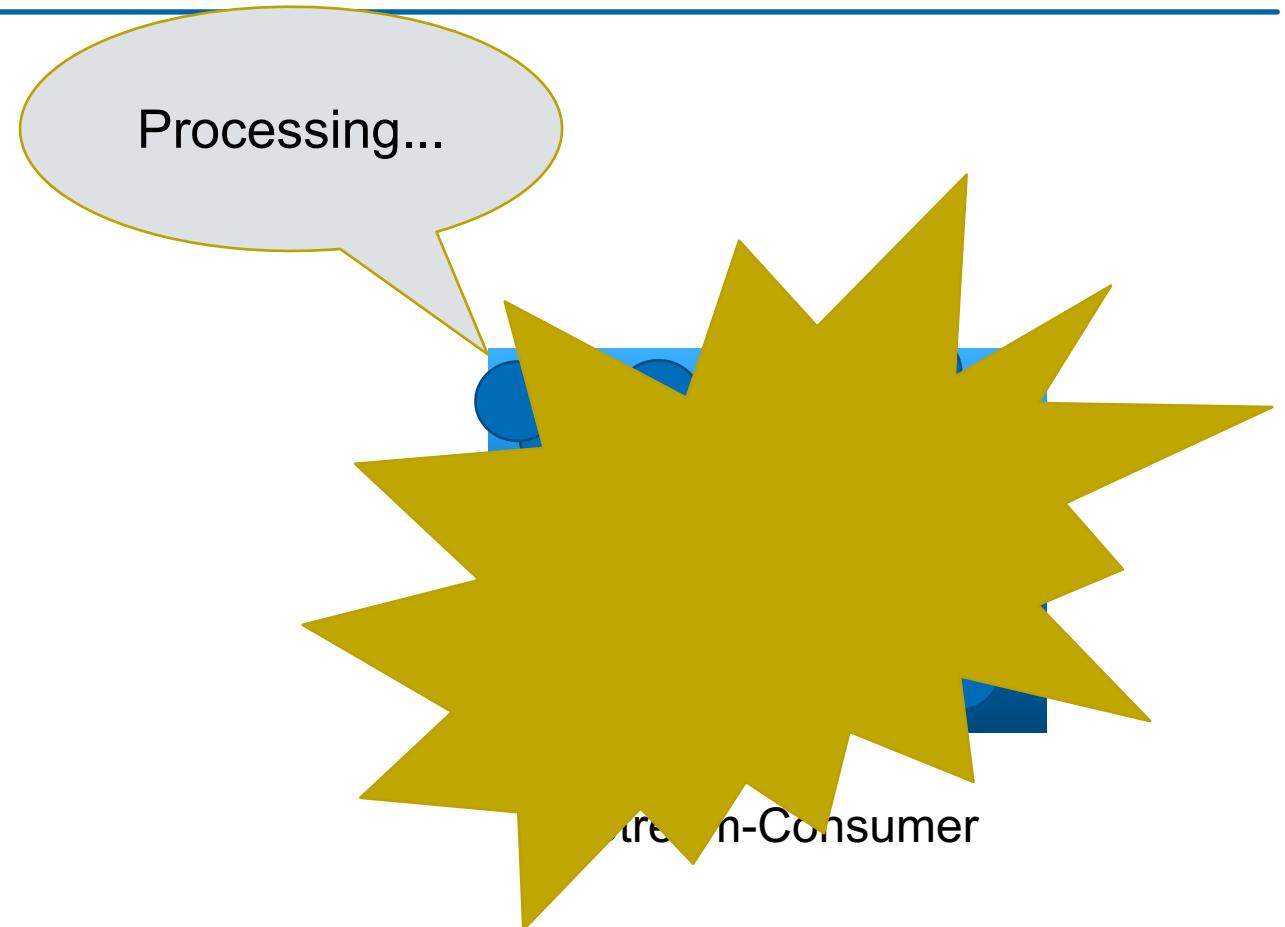


Stream-Consumer



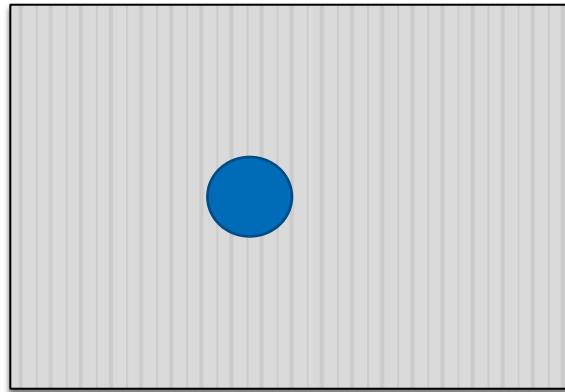


Stream-Producer



Stream-Consumer

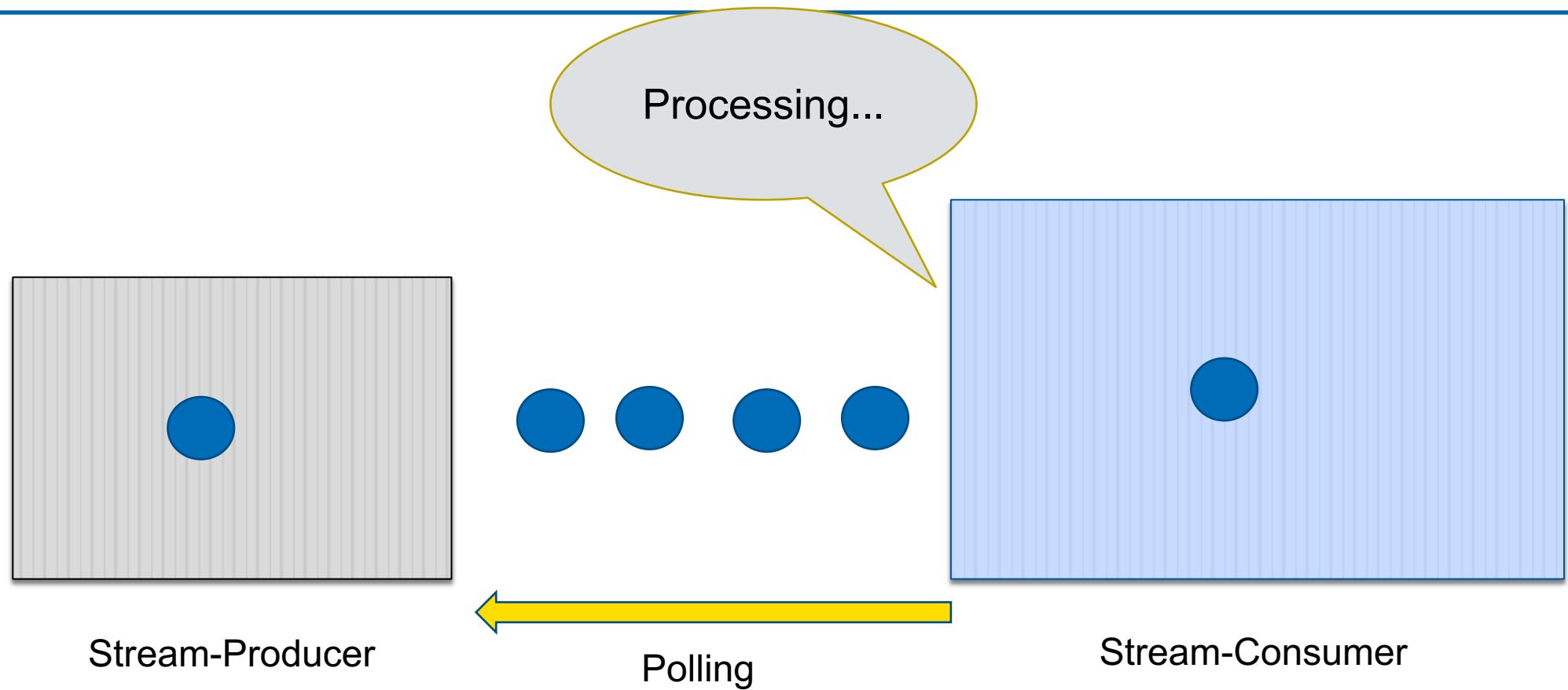
Scenario 2 - Polling

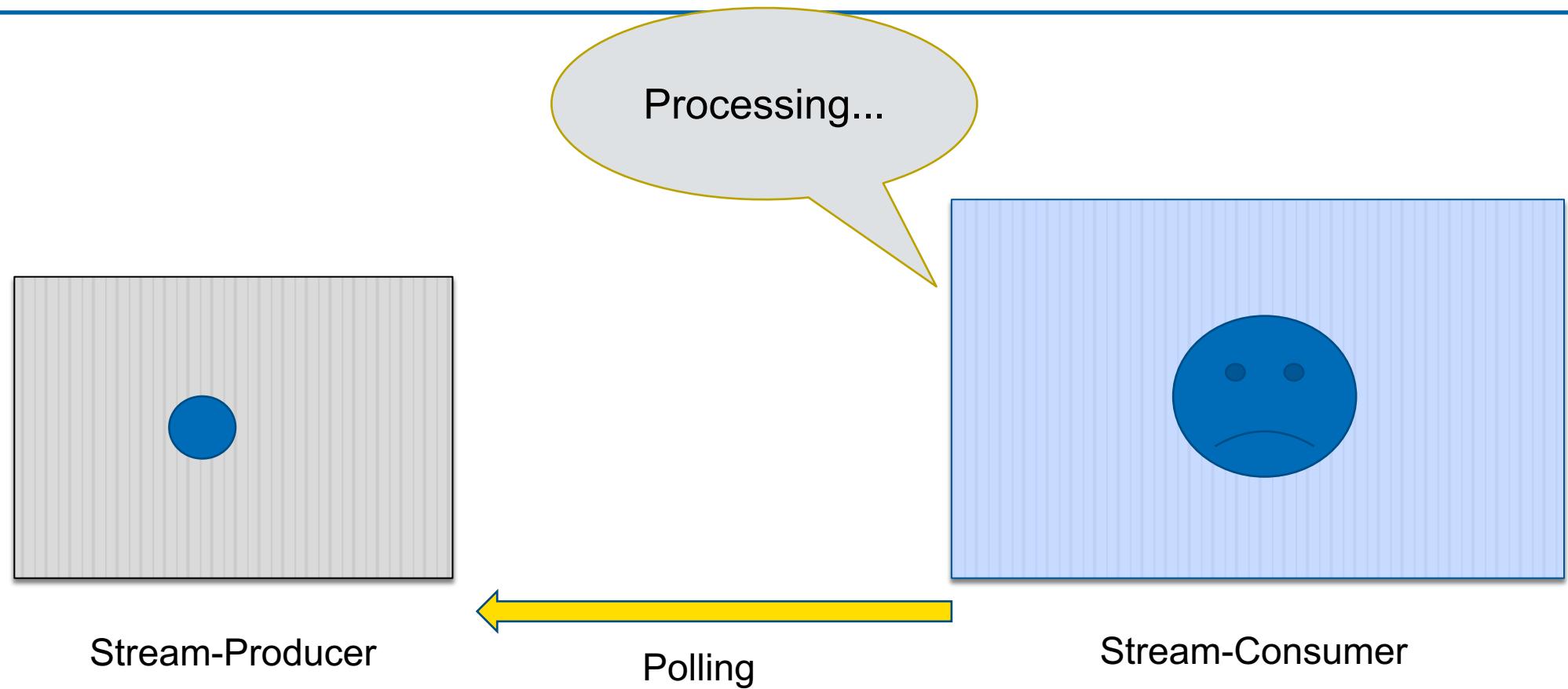


Stream-Producer



Stream-Consumer





PUSH

- 1) consumer has to process very fast in sync with the fast producer
- 2) Or consumer has to buffer at the consumer side

POLL

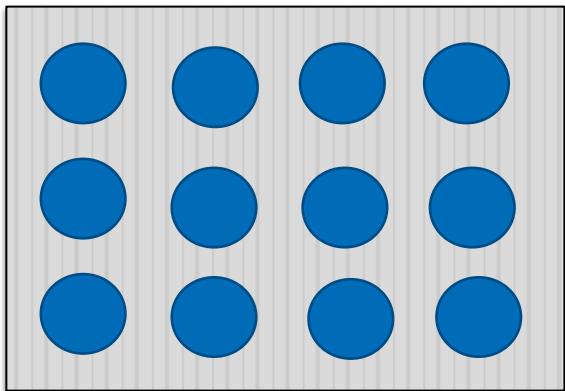
- 1) Fast consumer has to wait for producer
- 2) Slow consumer forces the producer to stop or to buffer data

✓ One-way communication

- ✓ Producer -> Consumer (PUSH)
- ✓ Consumer -> Producer (POLL)

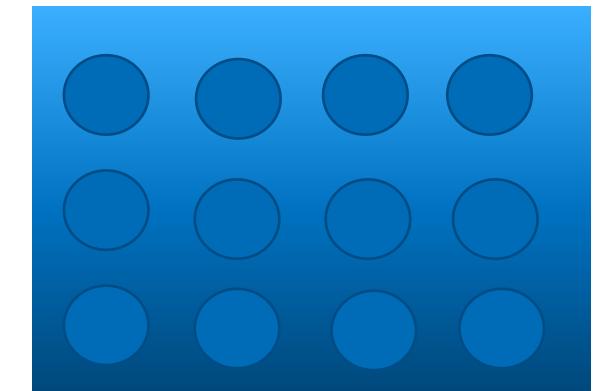
What if ?

Consumer can inform to the producer about it's capabilities and Producer can react according to Consumer's capabilities and needs.

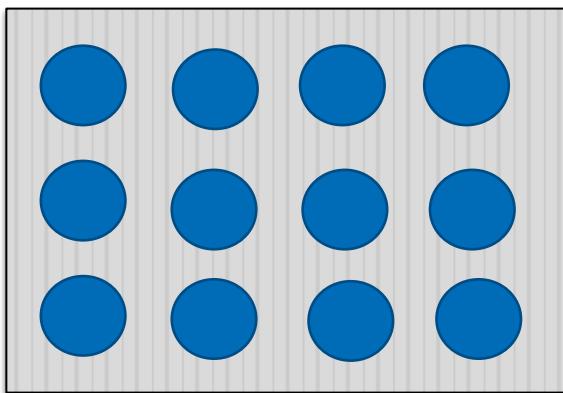


Stream-Producer

Hey I can handle only 12 at a time !

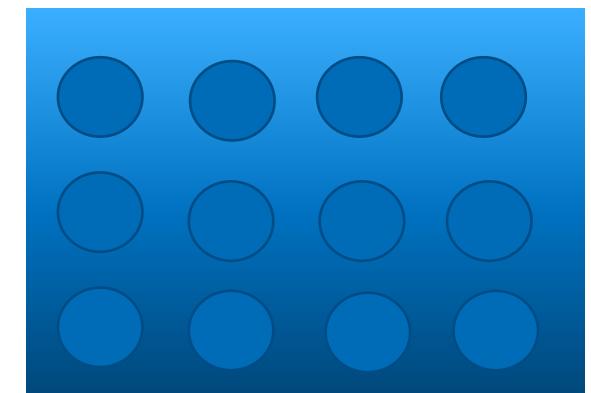
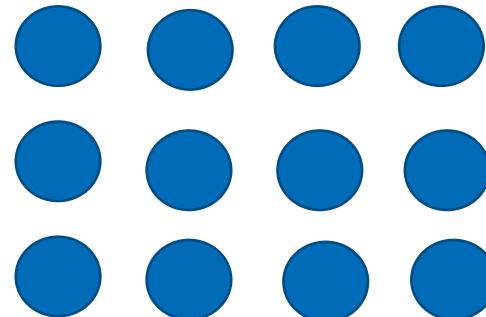


Stream-Consumer



Stream-Producer

**Here you go! But ask me if you want
more with your new capabilities !**



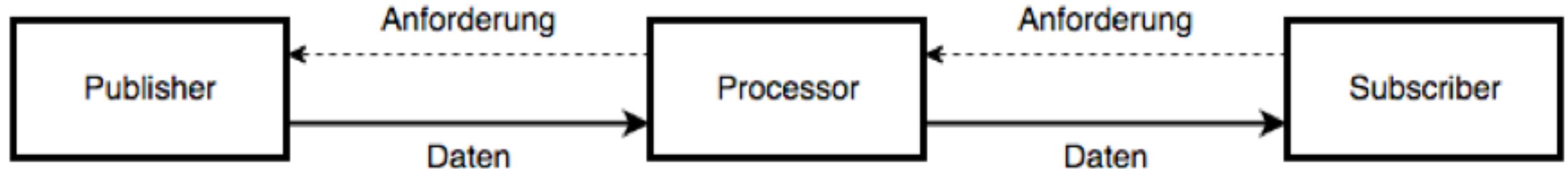
Stream-Consumer

“... provide a standard for **asynchronous stream processing** with **non-blocking back pressure**. ... aimed at runtime environments (JVM and JavaScript) ..”

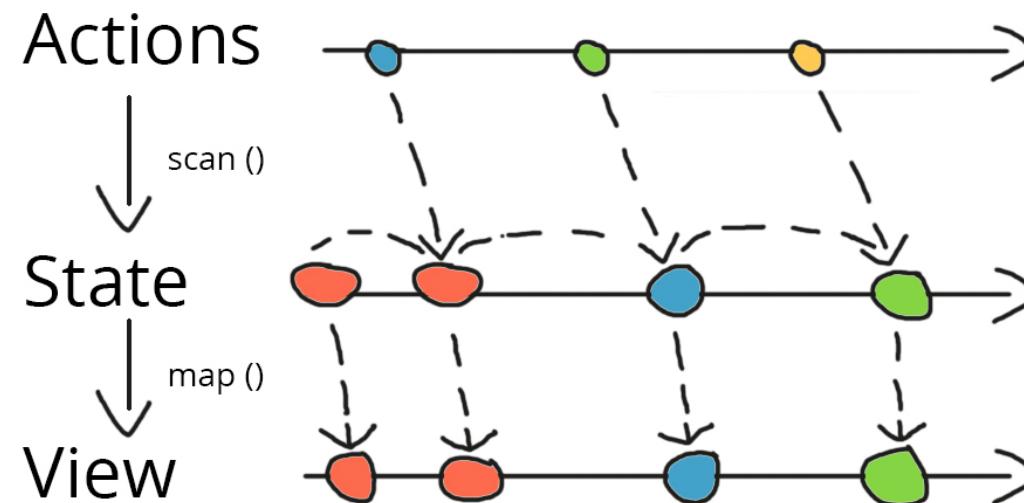
- Publisher veröffentlicht Daten und Subscriber verarbeitet Daten



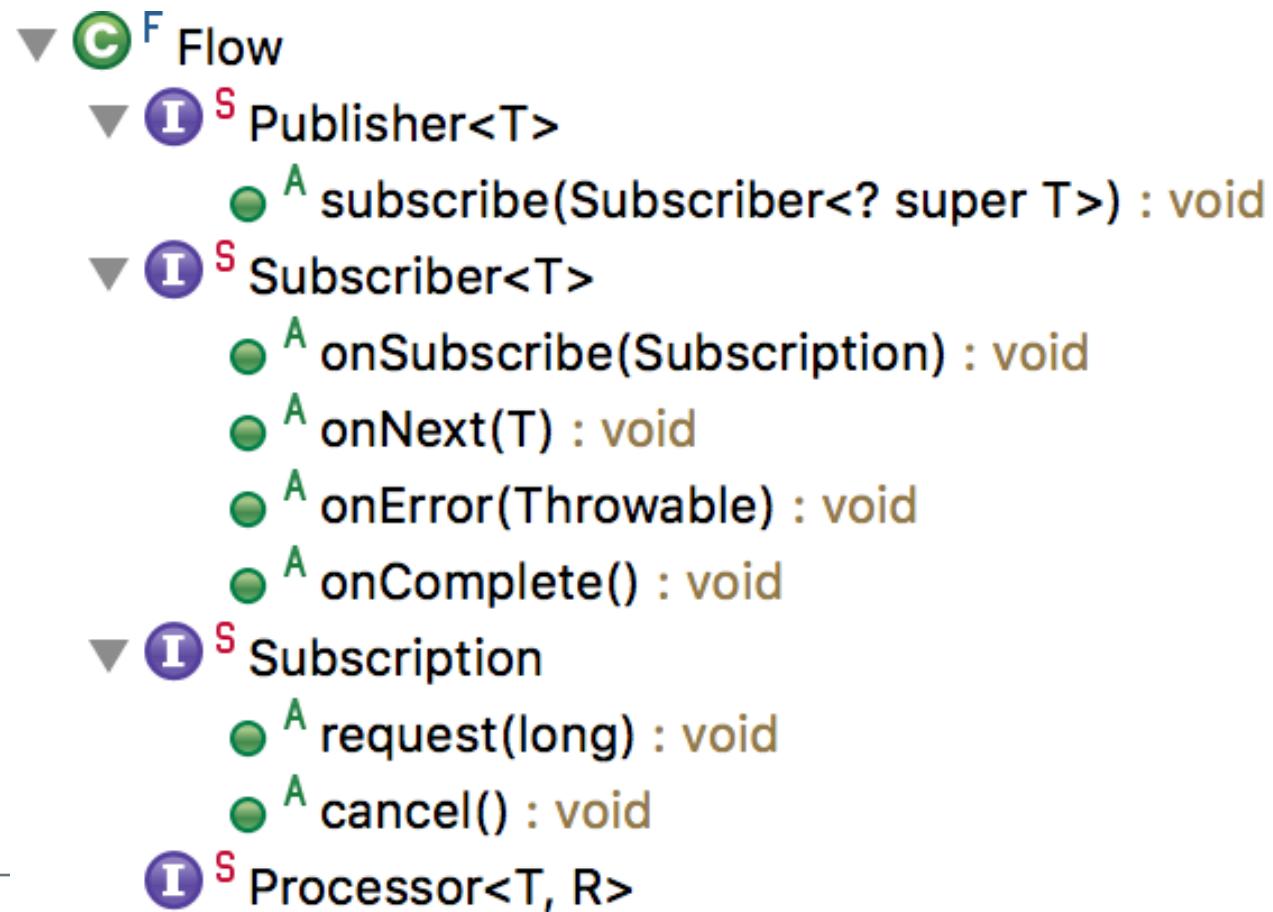
- Ausbau zu Verarbeitungskette, ähnlich wie bei einem Umzug eine Kette von Helfern

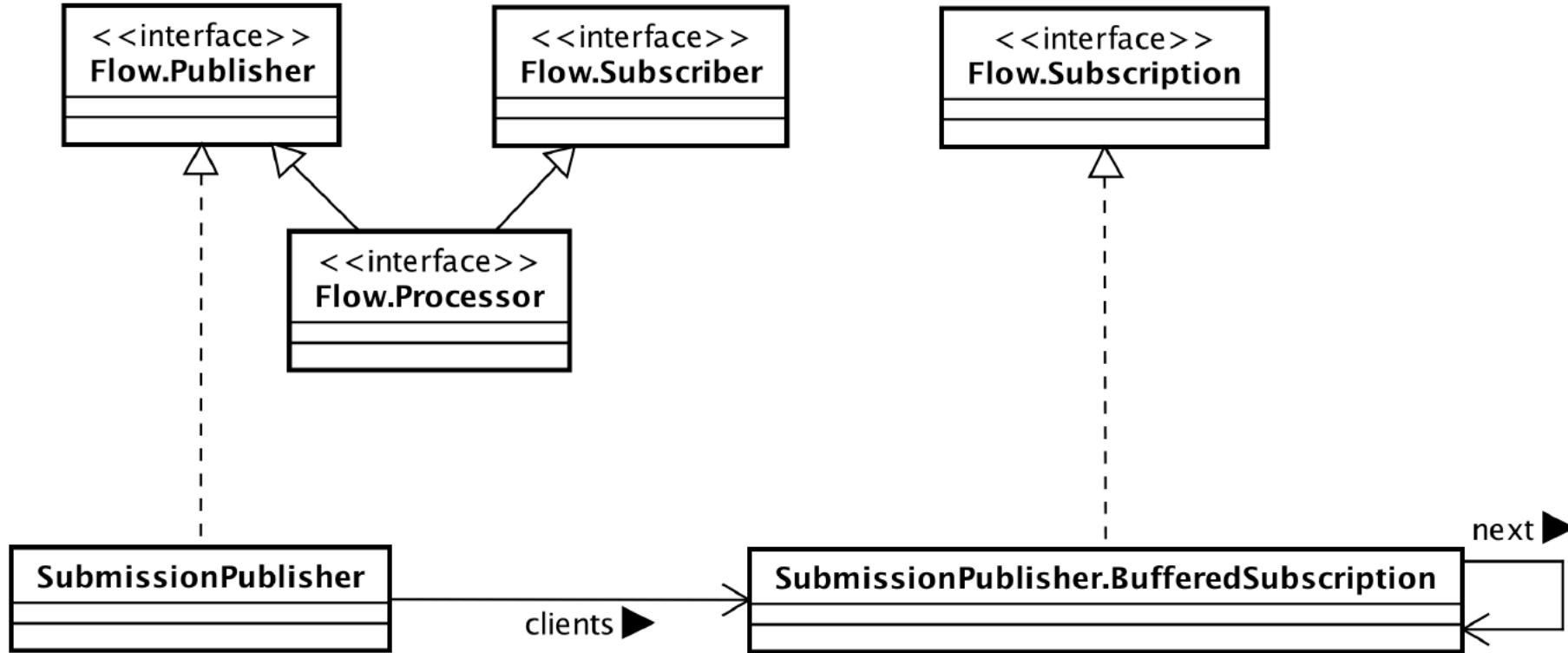


Reactive Streams im JDK



- Publish-Subscribe-Framework durch die Klasse **Flow** sowie eine Utility-Klasse **SubmissionPublisher**





```
public class SubmissionPublisherExample
{
    public static void main(String[] args) throws InterruptedException
    {
        try (SubmissionPublisher<Integer> publisher =
                new SubmissionPublisher<>())
        {
            publisher.subscribe(new ConsoleOutNumberSubscriber());
            System.out.println("Submitting items ...");
            for ( int i = 0; i < 10; i++)
            {
                publisher.submit(i);
            }
            TimeUnit.SECONDS.sleep(1);
        }
    }
}
```

Simple Subscriber

```
public class ConsoleOutNumberSubscriber implements Flow.Subscriber<Integer>
{
    private Flow.Subscription subscription;

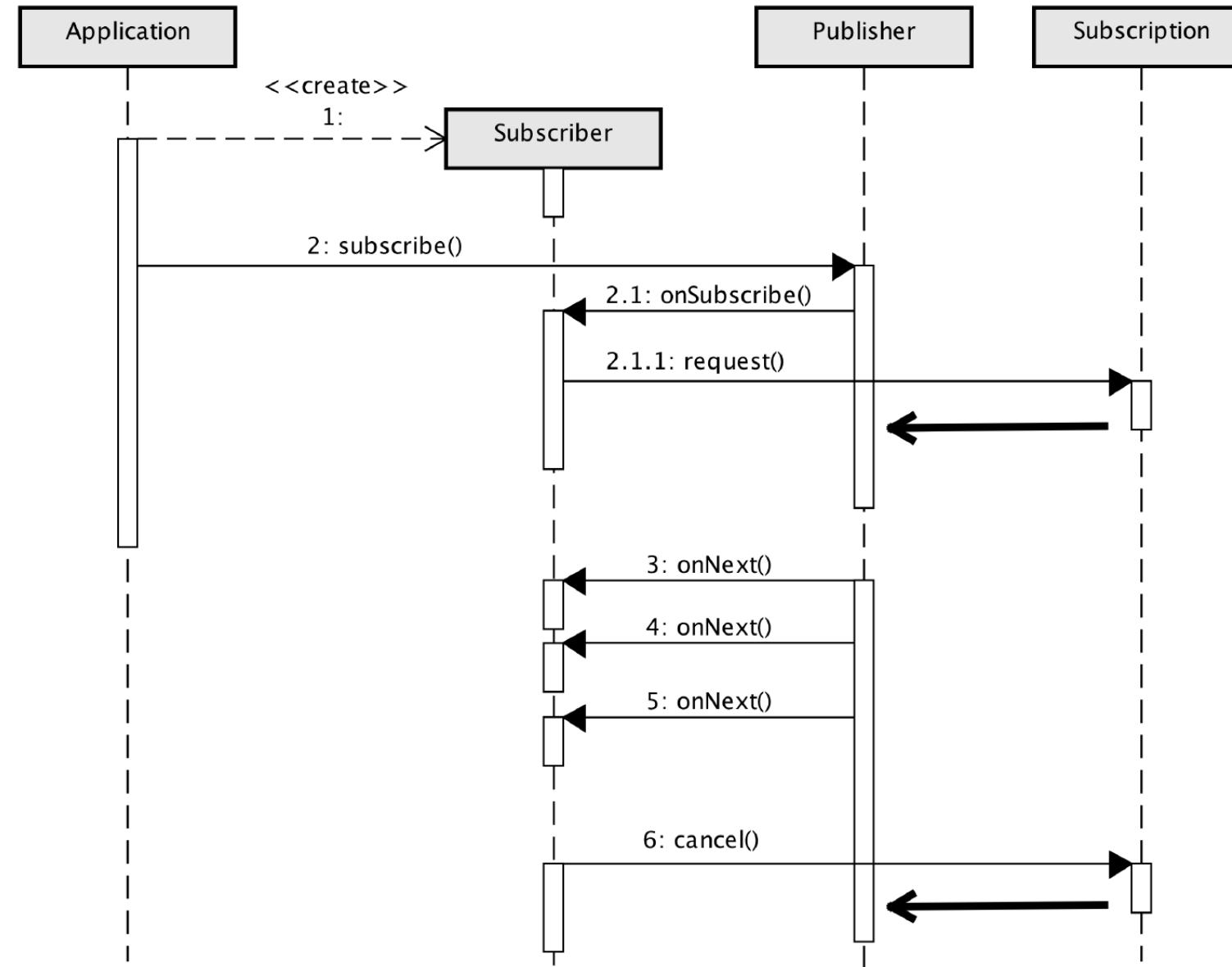
    @Override
    public void onSubscribe(Flow.Subscription subscription)
    {
        System.out.println("onSubscribe(): " + subscription);
        this.subscription = subscription;
        subscription.request(1);
    }

    @Override
    public void onNext(Integer item)
    {
        System.out.println("onNext() item: " + item);
        subscription.request(1);
    }

    // onError() / onComplete()
}
```

Submitting items...

```
onNext() received item: 0
onNext() received item: 1
onNext() received item: 2
onNext() received item: 3
onNext() received item: 4
onNext() received item: 5
onNext() received item: 6
onNext() received item: 7
onNext() received item: 8
onNext() received item: 9
onComplete()
```



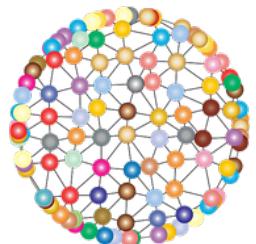
Subscription - Cancel

```
@Override  
public void onNext(Integer value) {  
    System.out.println("Received-->" + value);  
    if(value<200) {  
        subscription.request(1);  
    }else {  
        subscription.cancel();  
    }  
}
```



The screenshot shows a Java IDE's console window with the title bar "Console" and "Progress". The console output is from a terminated process named "ReactiveProgramming". The output consists of 200 lines of text, each starting with "Received-->" followed by an integer value ranging from 186 to 200.

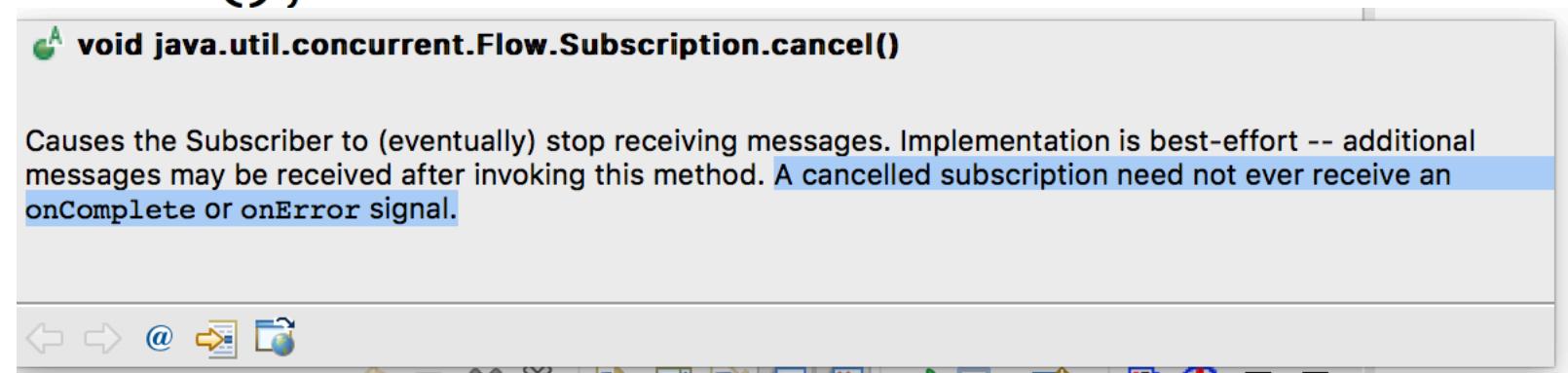
Value
Received-->186
Received-->187
Received-->188
Received-->189
Received-->190
Received-->191
Received-->192
Received-->193
Received-->194
Received-->195
Received-->196
Received-->197
Received-->198
Received-->199
Received-->200



**Wo ist denn die Ausgabe
von «Completed!»?**

Flow-API - <Interface> Subscription

```
@Override  
public void onNext(Integer value) {  
    System.out.println("Received-->" + value);  
    if(value<200) {  
        subscription.request(1);  
    }else {  
        subscription.cancel();  
    }  
}
```



DEMO

Übungen PART 2

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

PART 3: Weitere Neuerungen und Änderungen in den APIs

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

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.util.Arrays



- **Erweiterungen in `java.util.Arrays`:**

- `equals()` – Vergleicht Arrays auf Gleichheit (bezogen auf Bereiche)
- `compare()` – Vergleicht Arrays (auch bezogen auf Bereiche)
- `mismatch()` – Ermittelt die erste Differenz in Array (auch bezogen auf Bereiche)

- **Arrays.equals()** schon lange im JDK, aber ...
 - man konnte den Vergleich leider nicht auf spezielle Bereiche einschränken

```
final byte[] string1 = "ABC-DEF-GHI".getBytes();
final byte[] string2 = "DEF".getBytes();
```

```
System.out.println(Arrays.equals(string1, string2));      => false
```

- **Arrays.compare()** neu im JDK
- Vergleicht gemäss Comparator<T> – kann man auch auf spezielle Bereiche einschränken

```
final byte[] string1 = "ABC-DEF-GHI".getBytes();
final byte[] string2 = "DEF".getBytes();
```

```
System.out.println(Arrays.compare(string1, string2));      => ABC < DEF => -3
```

```
System.out.println(Arrays.compare(string1, 4, 7,
                                  string2, 0, 3));      => DEF = DEF => 0
```

```
System.out.println(Arrays.compare(string1, 8, 11,
                                  string2, 0, 3));      => GHI > DEF => 3
```

- **Arrays.mismatch()** neu im JDK
 - Prüft auf Abweichungen im Array – kann man auch auf spezielle Bereiche einschränken

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 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");
```

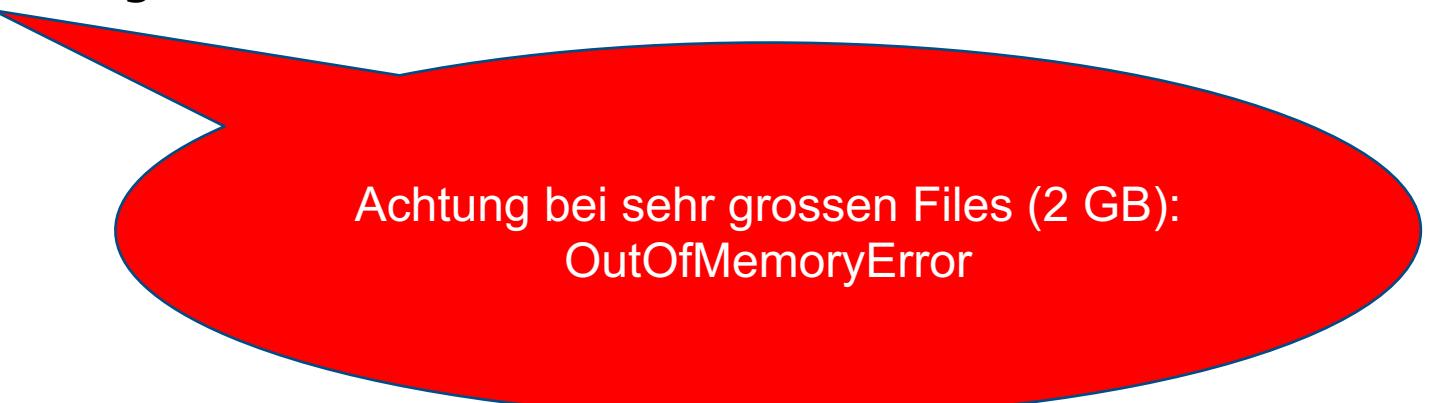
```
Files.writeString(destDath, "1: This is a string to file test\n");
Files.writeString(destDath, "2: Second line");
```

```
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
```

Verschiedenes ... Dies und das

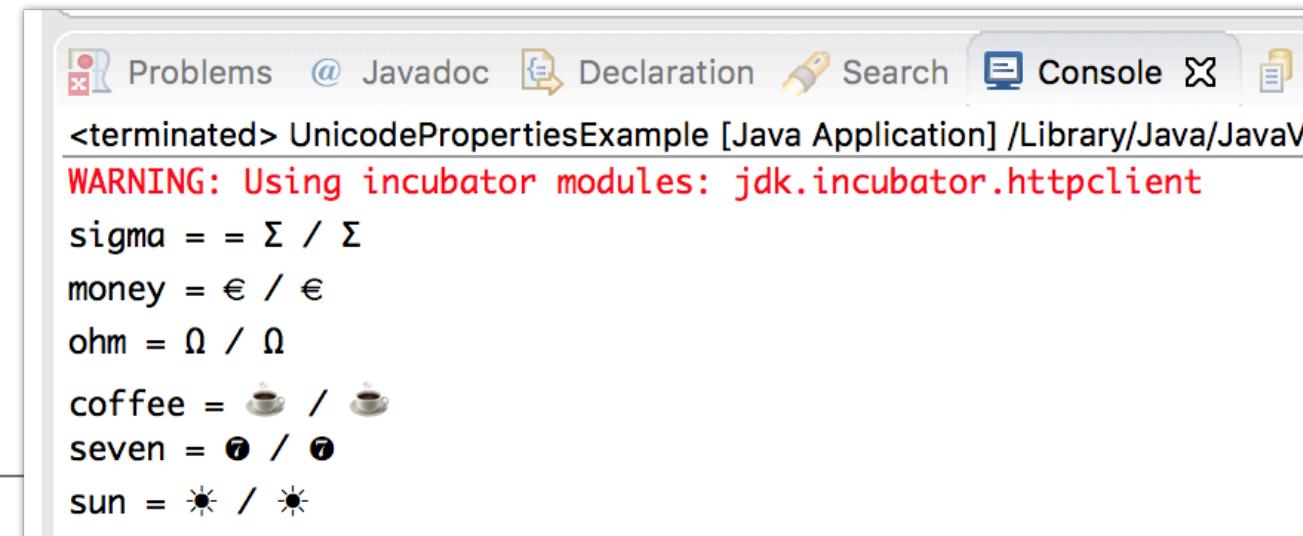


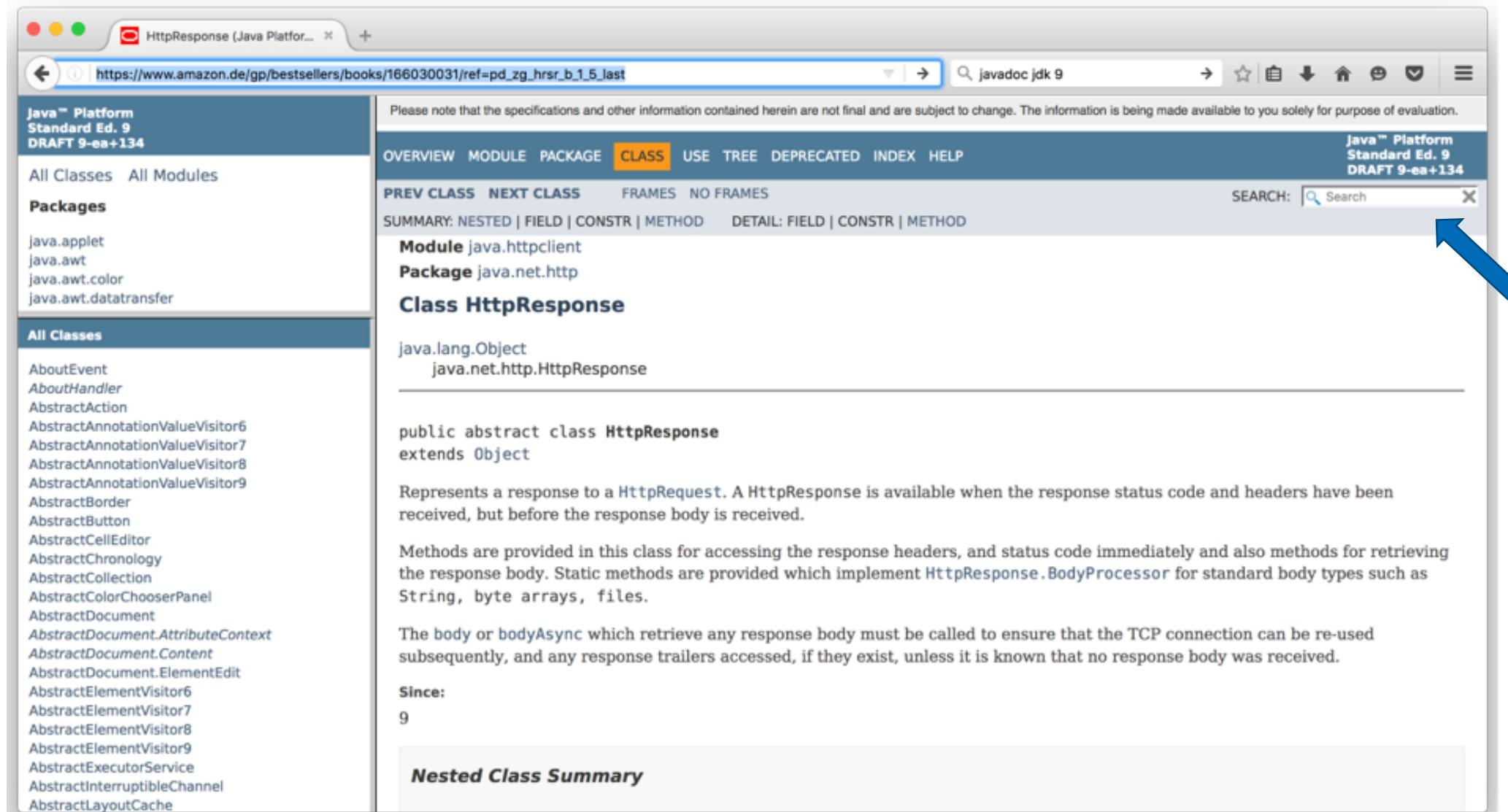
UTF-8 Resource Bundles:

```
public static void main(final String[] args) throws Exception
{
    try (final InputStream propertyFile =
        new FileInputStream("src/ch3_2_4/unicode/config.properties"))
    {
        final ResourceBundle properties = new PropertyResourceBundle(propertyFile);

        // JDK 9: Enumeration => Iterator
        properties.getKeys().asIterator().forEachRemaining(key ->
        {
            System.out.println(key + " = " + properties.getString(key));
        });
    }
}
```

```
money = € / \u20AC
coffee = ☕ / \u2615
sun = ☀ / \u2600
seven = 7 / \u277c
ohm = Ω / \u2126
sigma = Σ / \u03a3
```





The screenshot shows a Java documentation page for the `HttpServletResponse` class. The URL in the browser is https://www.amazon.de/gp/bestsellers/books/166030031/ref=pd_zg_hrsr_b_1_5_1est. The page is part of the "Java™ Platform Standard Ed. 9 DRAFT 9-ea+134" documentation. The navigation bar includes links for OVERVIEW, MODULE, PACKAGE, CLASS (which is highlighted), USE, TREE, DEPRECATED, INDEX, and HELP. Below the navigation bar, there are links for PREV CLASS, NEXT CLASS, FRAMES, and NO FRAMES. The search bar contains the text "javadoc jdk 9". The main content area is titled "Class `HttpResponse`". It shows the inheritance chain: `java.lang.Object` → `java.net.http.HttpResponse`. The class is described as a public abstract class that extends `Object`. It represents a response to a `HttpRequest`. The page also mentions methods for accessing response headers and status code, and static methods for retrieving response body. A note states that `body` or `bodyAsync` must be called to ensure TCP re-use. A "Since" section indicates the class was introduced in version 9. A "Nested Class Summary" section is at the bottom.

<https://docs.oracle.com/javase/9/docs/api/jdk/incubator/http/HttpResponse.html>

- Die Klasse `Predicate<T>` war sehr nützlich, um Filterbedingungen für die Stream-Verarbeitung ausdrücken zu können.
- Neben der Verknüpfung mit `and()` und `or()` liess sich eine Negation per `negate()` ausdrücken. Das war etwas umständlich und schwieriger lesbar:

```
// JDK 10 style
final Predicate<String> isEmpty = String::isEmpty;
final Predicate<String> notEmptyJdk10 = isEmpty.negate();
```

- Mit Java 11 lesbarer und ohne zusätzliche (künstliche) Explaining Variable `isEmpty`

```
// JDK 11 style
final Predicate<String> notEmptyJdk11 = Predicate.not(String::isEmpty);
// mit statischem Import
final Predicate<String> notEmptyJdk11 = not(String::isEmpty);
```

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**

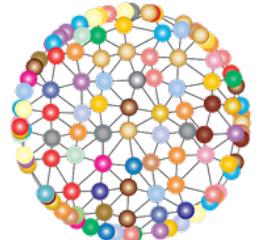
```
public static String readContent(final InputStream is) throws IOException
{
    try (final InputStreamReader isr = new InputStreamReader(is);
         final BufferedReader br = new BufferedReader(isr))
    {
        final StringBuilder content = new StringBuilder();

        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");
}
```

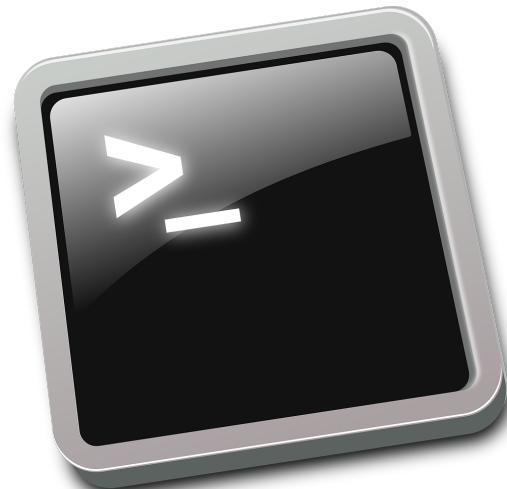
- **HTTPRequest**

```
.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())
```

Direct Compilation Launch Single-File Source- Code Programs



- 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

DEMO

Übungen PART 3

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

PART 4: Neuheiten und Änderungen in Java 12

- Syntaxerweiterungen bei switch => Java 13
- String APIs
- CompactNumberFormat
- Files
- Teeing()-Kollektor
- JMH (Microbenchmarks)

Erweiterungen in der Klasse String



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

- this method appends 'n' number of space characters (U+00200) in front of each line, then suffixed with a line feed "\n"

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

- Es sind aber auch negative Werte erlaubt:

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

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

```
'      Test
'
9
```

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

```
'6789
'
5
```

```
private static void indentHowItShouldBeUsed()
{
    var header = "Report";
    var infoMessage = "This is a message\nThis is line 2\nLine3".indent(4);

    System.out.println(header);
    System.out.println(infoMessage);
}
```

```
private static void indentHowItShouldBeUsed()
{
    var header = "Report";
    var infoMessage = "This is a message\nThis is line 2\nLine3".indent(4);

    System.out.println(header);
    System.out.println(infoMessage);
}
```

```
Report
    This is a message
    This is line 2
    Line3
```

- Beispiel: alle Zeichen eines Strings
 - In UPPERCASE wandeln
 - Dann T entfernen
 - Zum Schluss aufspalten
- Bisher realisiert man das beispielsweise wie folgt:

```
var text = "This is a Test";  
  
var upperCase = text.toUpperCase();  
var noTs = upperCase.replaceAll("T", "");  
var result = noTs.split(" ");  
  
System.out.println(Arrays.asList(result));
```

- Beispiel: alle Zeichen eines Strings
 - In UPPERCASE wandeln
 - Dann T entfernen
 - Zum Schluss aufspalten
- Bisher realisiert man das beispielsweise wie folgt:

```
var text = "This is a Test";  
  
var upperCase = text.toUpperCase();  
var noTs = upperCase.replaceAll("T", "");  
var result = noTs.split(" ");  
  
System.out.println(Arrays.asList(result));
```

[HIS, IS, A, ES]

- Einsatz von `transform()` für die Verarbeitungskette

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

```
var text = "This is a Test";  
  
// chaining of operations  
var result = text.transform(String::toUpperCase).  
    transform(str -> str.replaceAll("T", "")).  
    transform(str -> str.split(" "));
```

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

[HIS, IS, A, ES]

- Analog zu `map()` in Streams, zum Hintereinanderschalten von Transformationen
- Eher theoretisch praktisch ☺

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

Erweiterung CompactNumberFormat



- **CompactNumberFormat** is a subclass of **NumberFormat**
 - formats a decimal number in a compact form, Locale sensitive
 - NumberFormat is the abstract base class for all number formats which provides the interface for formatting and parsing numbers.
 - An example of a SHORT compact form would be writing 10,000 as 10K,
 - 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(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));
}
```

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

```
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"));
}
```

```
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

Erweiterung in der Klasse 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);
```

```
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
```

```
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);
```

```
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

- Erwartungskonform bei gleichem Encoding,
 - Gleiche Länge
 - Gleicher Inhalt (auf Byte-Ebene)
- Deshalb ...

File 1	File 2	Encoding	Ergebnis
ABCD	ABCD	Gleich	-1
ABCDEF	ABCDXY	Gleich	4
Zürich	Zürich	Abweichend	Positiver Wert, erstes Zeichen mit Umlaut oder Encoding-Abweichung

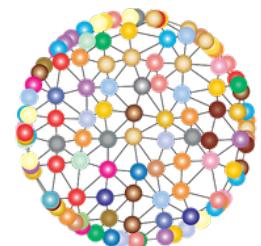
- Wenn die Pfade auf das gleich File zeigen, ist es natürlich auch gleich

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.

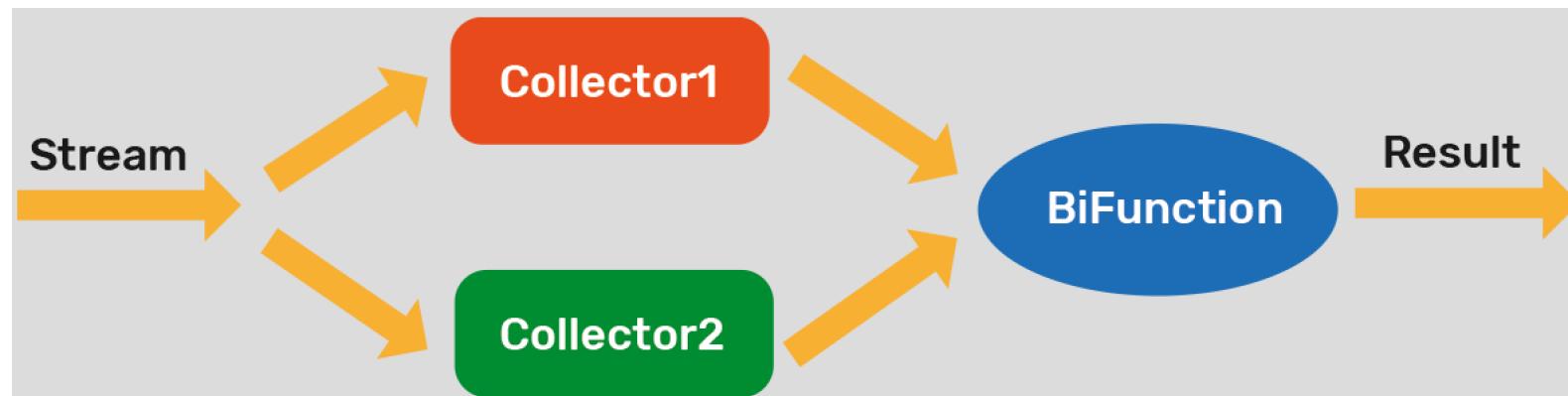


Was fehlt denn noch?

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)
```



```
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(1, 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));
}
```

```
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(1, 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));
}
```

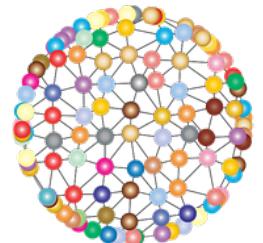


```
Pair<T> [first=6, second=21]
Pair<T> [first=7, second=57]
```

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

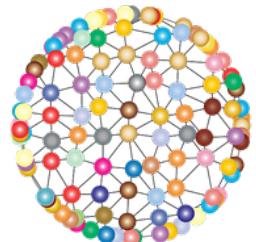


**Wie wäre es mit
Map.Entry als Ersatz für
ein eigenes Pair?**



NICHT SO GUT!!

**Warum? Map.Entry ist für Maps
gedacht und sollte nur in deren
Kontext genutzt werden.**



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]]
```

JMH



- 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

- **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**

- 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

```
// 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 Durchläufen und Durchschnittsbildung weniger anfällig für Systemlastschwankungen oder sonstige Störeinflüsse.
- recht einfach auch Minimal- und Maximaldauer oder Standardabweichung zu ermitteln.

- 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;
```

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

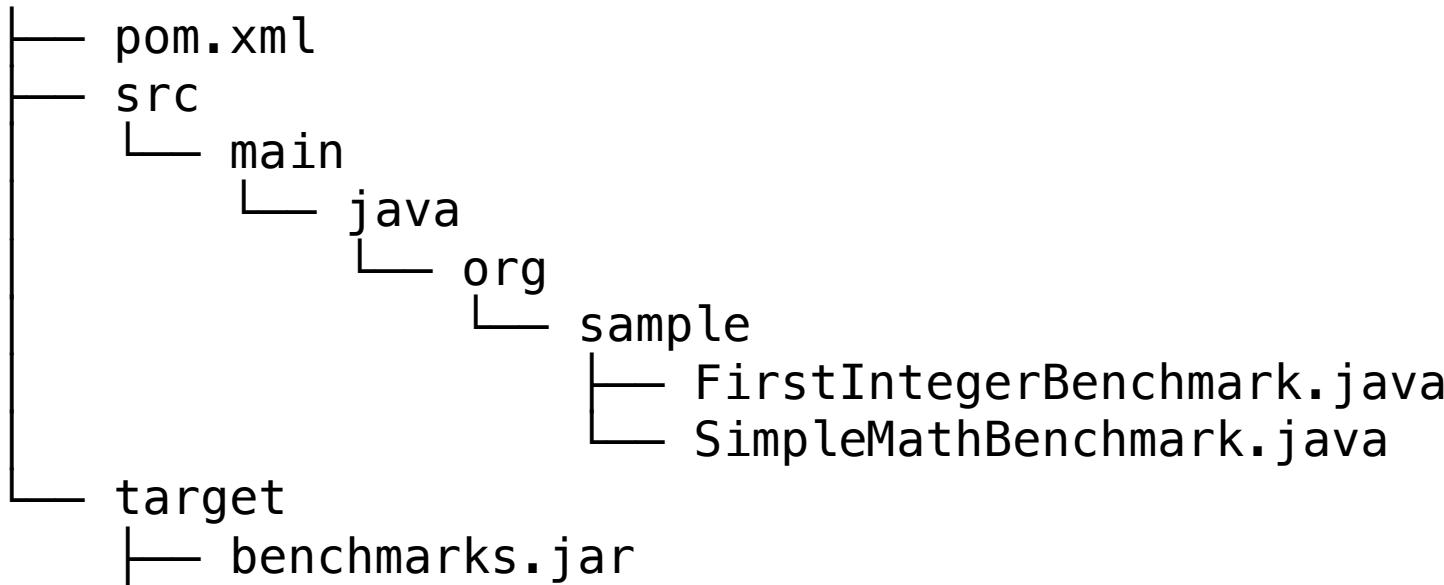
```
mvn archetype:generate \  
> -DinteractiveMode=false \  
> -DarchetypeGroupId=org.openjdk.jmh \  
> -DarchetypeArtifactId=jmh-java-benchmark-archetype \  
> -DgroupId=org.sample \  
> -DartifactId=jmh-test \  
> -Dversion=1.0-SNAPSHOT
```

- 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

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

Benchmark	Mode	Cnt	Score	Error	Units
StringAddBenchmark.stringBuilderAppend_Dump	avgt	10	22.349	± 0.091	ms/op
StringAddBenchmark.stringBuilderAppend_Multiple	avgt	10	15.872	± 0.055	ms/op
StringAddBenchmark.stringConcat	avgt	10	636655.091	± 80873.906	ms/op
StringAddBenchmark.stringPlus	avgt	10	673963.206	± 64323.287	ms/op

PART 5: Neuheiten und Änderungen in Java 13

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

Build-Tools und IDEs



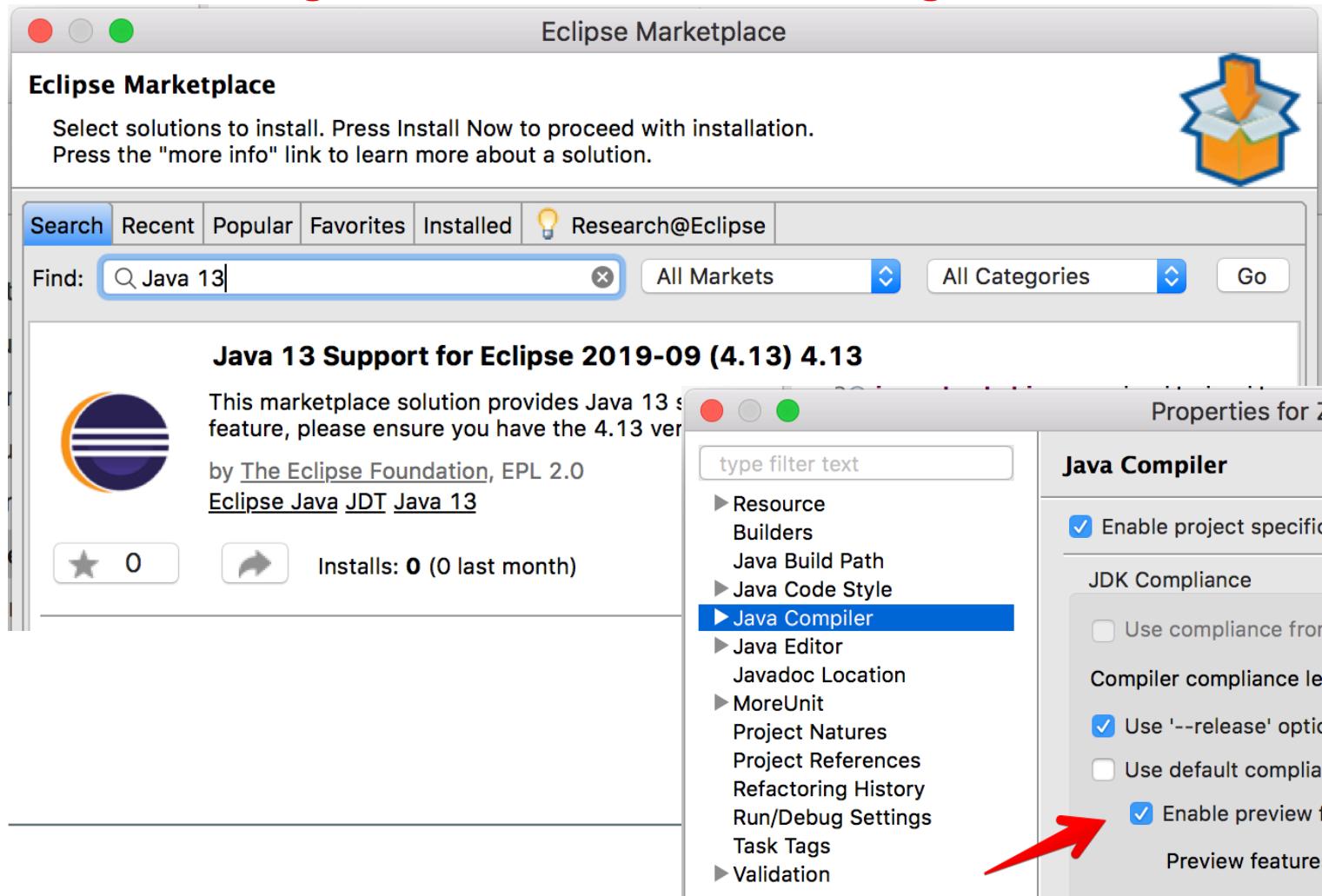
- Aktuelle IDEs & Tools grundsätzlich gut
- Eclipse: Version 2019-09 mit Plugin
- IntelliJ: Version 2019.2.4
- Maven: 3.6.2, Compiler-Plugin: 3.8.1
- Gradle: 5.6.4 (1. November 2019) ☺☺☺
- Aktivierung von Preview-Features nötig
 - In Dialogen
 - Im Build-Skript



Maven™

The logo for Gradle features a stylized blue elephant icon to the left of the word "Gradle" in a bold, blue, sans-serif font.

- 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: 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:

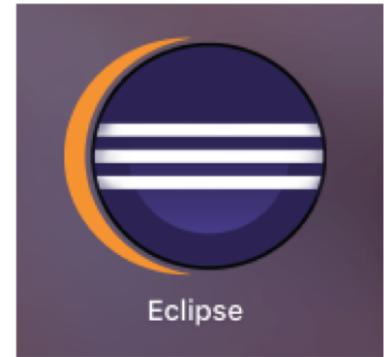
Use '--release' option

Use default compliance settings

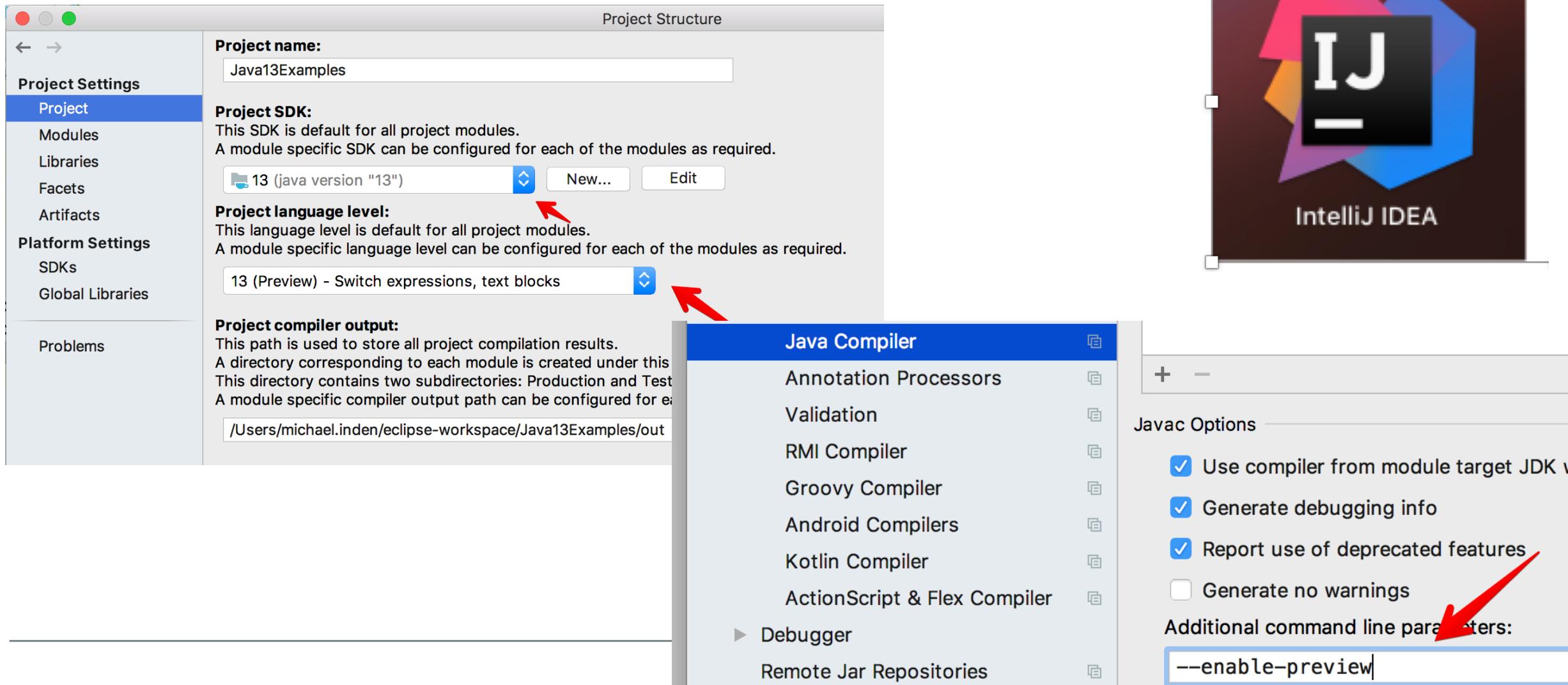
Enable preview features for Java 13

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"]  
}
```



* What went wrong:

Could not compile build file '/Users/michael.inden/Desktop/Vorträge/ZZZZZZ_JAX-London Java 9 bis 13 WORKSHOP 2019/Übungen-Teil1-APIs/Best_of_Java_9_bis_13/build.gradle'.
> startup failed:

General error during semantic analysis: Unsupported class file major version 57

java.lang.IllegalArgumentException: Unsupported class file major version 57
at groovyjarjarasm.asm.ClassReader.<init>(ClassReader.java:184)

- **Aktivierung von Preview-Features nötig**

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

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



- **Zudem muss Gradle noch mit Java 12 oder früher gestartet werden: Datei gradle.properties:**

```
// JAVA_HOME auf Java 12 oder früher in der Konsole  
// Aber zum Kompilieren auf Java 13  
org.gradle.java.home=/Library/Java/JavaVirtualMachines/jdk-13.jdk/Contents/Home
```

- 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 4 & 5

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

Questions?



Thank You
