Java Update 9 bis 11

https://github.com/Michaeli71/GLS-Update-Java-8-16.git

Michael Inden

Freiberuflicher Consultant, Buchautor und Trainer

Speaker Intro

- Michael Inden, Jahrgang 1971
- Diplom-Informatiker, C.v.O. Uni Oldenburg

- ~8 ¼ Jahre **SSE** bei Heidelberger Druckmaschinen AG in Kiel
- ~6 ¾ Jahre **TPL, SA** bei IVU Traffic Technologies AG in Aachen
- ~4 ¼ Jahre **LSA / Trainer** bei Zühlke Engineering AG in Zürich
- ~3 Jahre **TL / CTO** bei Direct Mail Informatics / ASMIQ in Zürich
- **Freiberuflicher Consultant, Trainer und Konferenz-Speaker**
- Autor und Gutachter beim dpunkt.verlag

E-Mail: michael.inden@hotmail.ch
Blog: https://jaxenter.de/author/minden

Kurse: Bitte sprecht mich an!

https://github.com/Michaeli71/GLS-Update-Java-8-16.git

Agenda

Workshop Contents

- **Vorbemerkungen**
- **PART 1:** Syntax-Erweiterungen & Neuheiten und Änderungen in Java 9 bis 11
- **PART 2:** Multi-Threading mit CompletableFuture und Reactive Streams
- **PART 3:** Weitere Neuheiten und Änderungen in JDK 9 bis 11

- **Separat: Modularisierung im Kurzüberblick**

- **Separat: Ausblick Java 12 - 16**
  - PART 1: Syntax-Erweiterungen & Neuheiten und Änderungen in Java 12 bis 16
  - PART 2: Weitere Neuheiten und Änderungen in Java 12 bis 16

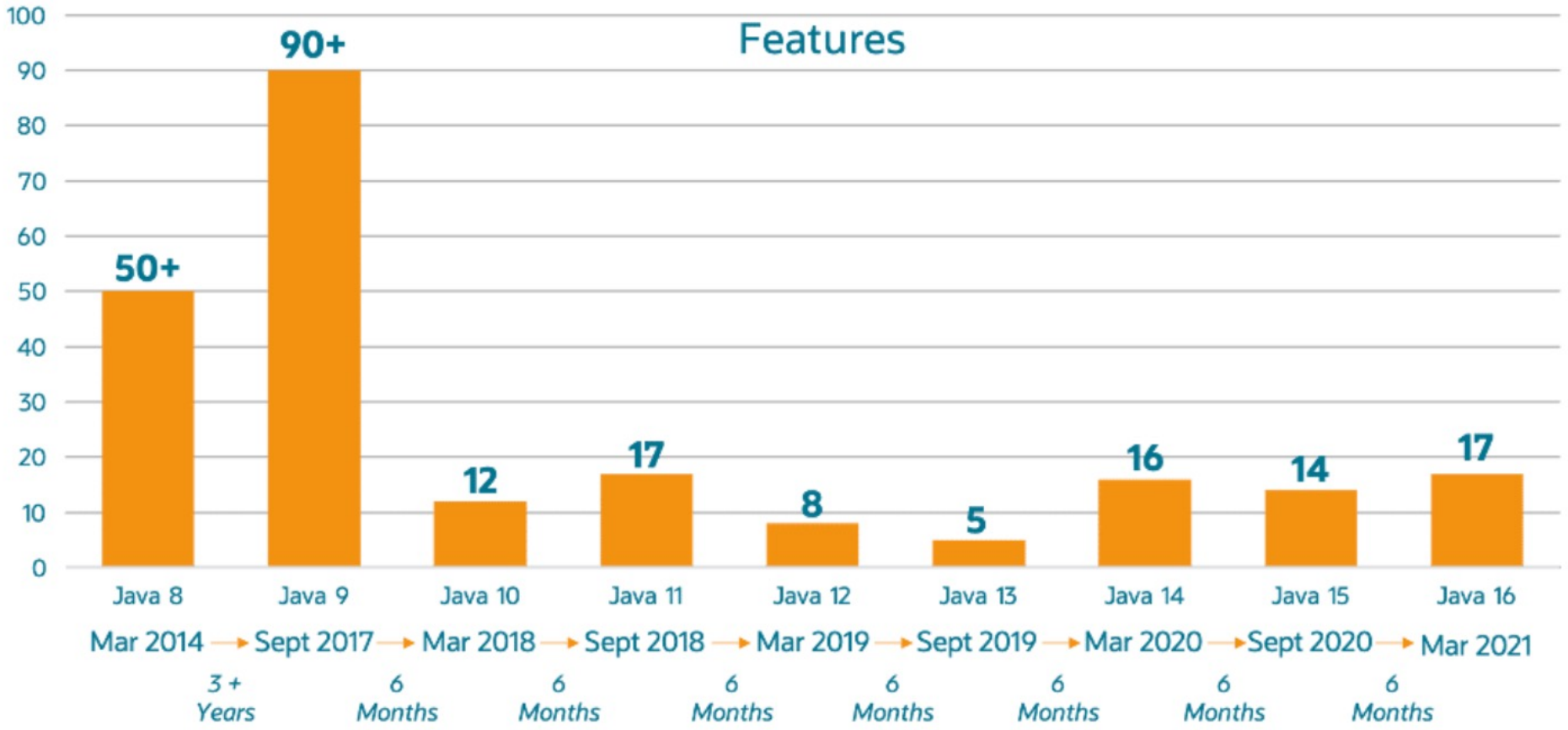
---

https://github.com/Michaeli71/GLS-Update-Java-8-16.git

Workshop Contents

JDK	Release-Datum	Enwicklungszeit	LTS	Workshop
Oracle JDK 8	**3 / 2014**	**-**	**Ja, *mittlerweile auch kommerziell***	**-/-**
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, *kommerziell***	**Part 1, 2, 3**
Oracle JDK 12	3 / 2019	6 Monate	-	separat
Oracle JDK 13	9 / 2019	6 Monate	-	separat
Oracle JDK 14	3 / 2020	6 Monate	-	separat
Oracle JDK 15	9 / 2020	6 Monate	-	separat
Oracle JDK 16	**3 / 2021**	**6 Monate**	**-**	separat

Einordnung 6 monatiger Releasezyklus



https://www.techzine.eu/news/devops/57122/oracle-releases-version-16-of-java/

Neues Lizenzmodell

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.**
- **Alternativen: OpenJDK (https://openjdk.java.net/) oder Adopt Open JDK (https://adoptopenjdk.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

Anonyme innere Klassen und der Diamond Operator

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

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

@Deprecated-Annotation

- @Deprecated dient zum Markieren von 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()
{
   // ...
}
```

@Deprecated-Annotation

- @Deprecated dient zum Markieren von obsoletem Sourcecode

- JDK 8: keine Parameter
- JDK 9: Zwei Parameter @since und @forRemoval

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

Effectivly final in ARM

- ARM erlaubt die automatische Ressourcenverwaltung
- Für jedes Object vom Typ `java.lang.AutoCloseable`
- Java 8 erfordert die Definition von Variablen

```
final BufferedInputStream bufferedIs = ...;
BufferedOutputStream bufferedOs = ...

try (final BufferedInputStream bis = bufferedIs;
     final BufferedOutputStream bos = bufferedOs)
{
    ...
```

Underscore als Identifier

- _ 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();
                  ^
```

Wieso?

Underscore als Identifier

- **Idee: Kennzeichnung unbenutzter Parameter in Lambdas für die Zukunft ermöglichen**
- **Unter anderem Python erlaubt Ähnliches**
- **Syntax-Vorschlag, bisher aber nicht umgesetzt:**

```
(x, y, z) -> x + y                    (x, y, _) -> x + y

(x, y, z) -> x * z          →         (x, _, z) -> x * z

(person, str) -> person.getAge()      (person, _) -> person.getAge()
```

Private Methoden in Interfaces

Private Methoden in 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 => var

- Local Variable Type Inference

- **Neues reserviertes Wort `var` zur Definition lokaler Variablen, statt expliziter Typangabe**

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

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

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

Local Variable Type Inference => var

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

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

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


// var => Map<String, Long>

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

Local Variable Type Inference => var

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

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

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

- **Dazu nutzen wir diese Lambdas:**

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

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

Wäre es nicht schön, auch hier var zu nutzen?

Local Variable Type Inference => var

- Ja!!!

- Aber der Compiler kann rein basierend auf diesen Lambdas den konkreten Typ nicht ermitteln

- Somit ist keine Umwandlung in `var` möglich, sondern führt zur Fehlermeldung «Lambda expression needs an explicit target-type“.

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

- Wollte man diesen Fehler vermeiden, so müsste man folgenden Cast einfügen:

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

Local Variable Type Inference Fallstrick

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

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

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

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

Kompiliert das? Und wenn ja, wieso?

Local Variable Type Inference Fallstrick

- **Tatsächlich produziert das Ganze keinen Kompilierfehler**. Wie kommt das?

- Aufgrund des Diamond Operators, bzw. der nicht vorhandenen Typangabe, stehen dem Compiler nicht ausreichend Typinformationen zur Verfügung:

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

Local Variable Type Inference Beschränkungen

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

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

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

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

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

Wäre es nicht auch schön var für Attribute, Parameters oder Return-Types zu nutzen?

JA … Aber es ist NICHT möglich, weil der Typ nicht eindeutig vom Compiler ermittelt werden kann!

Neuheiten und Änderungen in Java 9 bis 11

- Process API
- Stream-API
- Optional<T>
- Collection Factory Methods
- Stream-API Kollektoren
- Strings
- Predicate

Process API

Process Handling Java 8

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

```
private static long getPidOldSytle() 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?

Vereinfachung mit Java 9

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

Das Interface ProcessHandle

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.

Das Interface ProcessHandle

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

Stream API in JDK9

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

takeWhile(…)

dropWhile(…)

ofNullable(…)

iterate(… , …, …)

Stream API in JDK9

takeWhile(…)

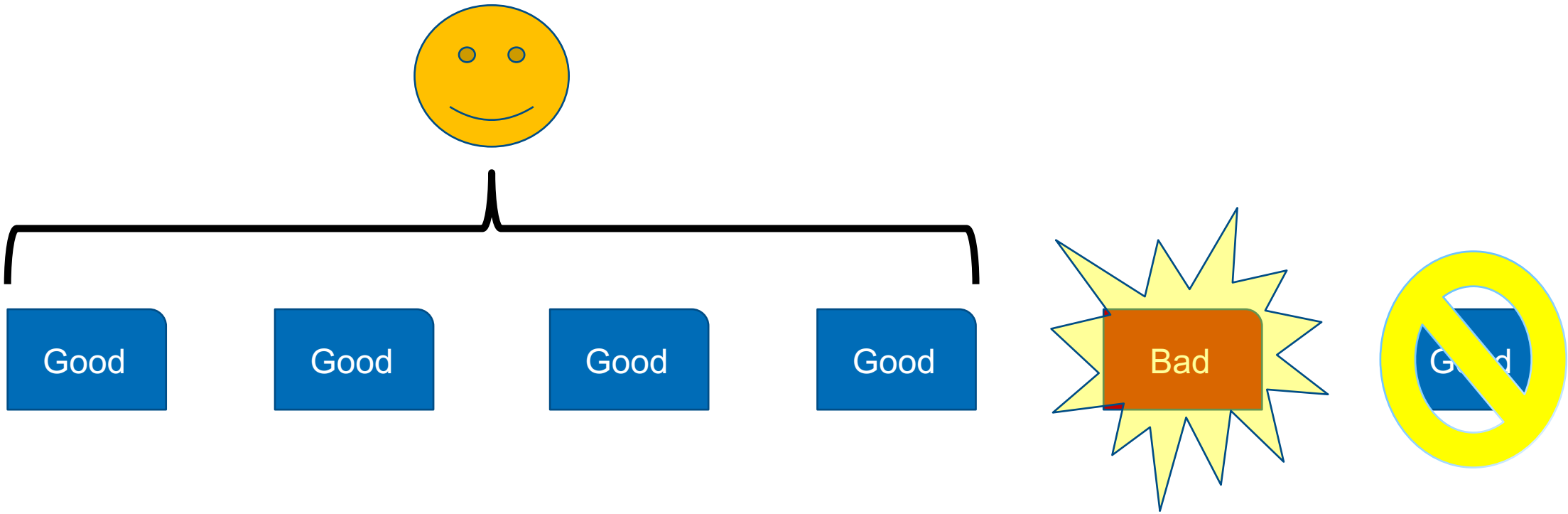
dropWhile(…)

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

ofNullable(…)

iterate(… ,…, …)

Stream – Product Scenario



How could we select only the Good till the first Bad one occurs?

Stream – Filter

JDK8

```
public class StrictCustomer {

    public static void main(String[] args) {

        Stream<String> deliveredProductsQuality = Stream.of("1. Good",
                                                            "2. Good",
                                                            "3. Good",
                                                            "4. Good",
                                                            "5. Bad",
                                                            "6. Good");

        deliveredProductsQuality.
            filter(productQuality -> productQuality.contains("Good")).
            forEach(System.out::println);
    }
}
```

```
1. Good
2. Good
3. Good
4. Good
6. Good
```

Stream – Filter

JDK8

```
public class StrictCustomer {

    public static void main(String[] args) {

        Stream<String> deliveredProductsQuality = Stream.of("1. Good",
                                                            "2. Good",
                                                            "3. Good",
                                                            "4. Good",
                                                            "5. Bad",
                                                            "6. Good");


        deliveredProductsQuality.
            filter(productQuality -> productQuality.contains("Good")).
            forEach(System.out::println);

    }

}
```

```
1. Good
2. Good
3. Good
4. Good
6. Good
```

So ? What do we do ?

Google

Google-Suche | Auf gut Glück!

Google angeboten in: English Français Italiano Rumantsch

takeWhile(…)

```
public class StrictCustomer {

    public static void main(String[] args) {

        Stream<String> deliveredProductsQuality = Stream.of("1. Good",
                                                            "2. Good",
                                                            "3. Good",
                                                            "4. Good",
                                                            "5. Bad",
                                                            "6. Good");

        deliveredProductsQuality.
                    takeWhile(productQuality -> productQuality.contains("Good")).
                    forEach(System.out::println);

    }

}
```

```
1. Good
2. Good
3. Good
4. Good
```

Stream API in JDK 9

takeWhile(…)

dropWhile(…)

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

ofNullable(…)

iterate(… ,… ,…)

dropWhile(…)

```
public class StrictCustomer {

    public static void main(String[] args) {

        Stream<String> deliveredProductsQuality = Stream.of("1. Bad",
                                                            "2. Bad",
                                                            "3. Bad",
                                                            "4. Good",
                                                            "5. Good",
                                                            "6. Good");


        deliveredProductsQuality.
                    dropWhile(productQuality -> productQuality.contains("Bad")).
                    forEach(System.out::println);

    }

}
```

dropWhile(…)

```
public class StrictCustomer {

    public static void main(String[] args) {

        Stream<String> deliveredProductsQuality = Stream.of("1. Bad",
                                                            "2. Bad",
                                                            "3. Bad",
                                                            "4. Good",
                                                            "5. Good",
                                                            "6. Good");


        deliveredProductsQuality.
                    dropWhile(productQuality -> productQuality.contains("Bad")).
                    forEach(System.out::println);
    }
}
```

```
4. Good
5. Good
6. Good
```

Kombination der beiden Methoden des Stream APIs

- **Kombination der beiden Methoden zur Extraktion von Daten:**

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

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

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

```
WELCOME
TO
JAX
Online
```

Stream API in JDK9

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.

ofNullable(T)

```
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;
    }
}
```

Achtung: Rückgabe von null sieht man leider in älteren Sourcecode (Legacycode) häufiger als man es sich wünscht!

ofNullable(T)

```
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
        at java.base/java.util.Optional.of(Optional.java:111)
        at java.base/java.util.stream.FindOps$FindSink$OfRef.get(F
```

ofNullable(T)

```
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;
    }
}
```

ofNullable(T)

```
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/JavaVirtualMachi
0
No element
```

Stream API in JDK9

takeWhile(…)

dropWhile(…)

ofNullable(…)

iterate(…, … , …)

Stream API

- `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

Was macht?

```
IntStream.iterate(1, x -> x + 1).filter(n -> n < 10).forEach(System.out::println)
```

Vermutung: 1 2 3 4 5 6 7 8 9?

Und was?

```
IntStream.iterate(1, x -> x + 1).limit(9).forEach(System.out::println)
```

Vermutung: 1 2 3 4 5 6 7 8 9?

Und was?

```
IntStream.iterate(1, x -> x < 10, x -> x + 1).forEach(System.out::println)
```

Vermutung: 1 2 3 4 5 6 7 8 9?

Stream API

Was macht?

```
IntStream.iterate(1, x -> x + 1).filter(n -> n < 10).forEach(System.out::println)
```

Und was?

```
IntStream.iterate(1, x -> x + 1).limit(9).forEach(System.out::println)
```

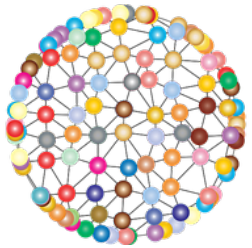
=> 1 2 3 9

Und was?

```
IntStream.iterate(1, x -> x < 10, x -> x + 1).forEach(System.out::println)
```

=> 1 2 3 9

DEMO mit JShell



Wie finden wir alle geraden und durch 3 teilbaren Zahlen inklusive 30?

Stream API

Mit `filter()` passt es wieder nicht ...

```
IntStream.iterate(0, x -> x + 2).
          filter(n -> n <= 30 && n % 3 == 0).forEach(System.out::println)
```

Erst recht passt das mit dem limit() nicht mehr ... wie schreiben wir hier die Bedingung?

```
IntStream.iterate(0, x -> x + 2).limit(???).forEach(System.out::println)
```

Stream API

JDK 9 `iterate()` als Abhilfe:

```
IntStream.iterate(0, n -> n <= 30, n -> n + 2).
          filter(n -> n % 3 == 0).forEach(System.out::println)
```

=>

```
0
6
12
18
24
30
```

Jede Stream-Methode wird gemäss ihres Sinns verwendet ... Kaum macht man es richtig, schon funktioniert es ☺

Optional<T>

Die Klasse Optional<T>

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

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

Optional<T> in JDK8 / JDK9

Optional<T>
- empty() <T> :
- of(T) <T> : Op
- ofNullable(T)
- equals(Object
- filter(Predicat
- flatMap(Funct
- get() : T
- hashCode() : i
- ifPresent(Con
- isPresent() : b
- map(Function
- orElse(T) : T
- orElseGet(Sup
- orElseThrow(S
- toString() : Str

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

Die Klasse Optional<T>

- **Gutes API, aber 3 Schwachstellen bei folgenden Aufgabenstellungen:**
  - **Das Ausführen von Aktionen auch im Negativfall.**
  - **Die Verknüpfung der Resultate mehrerer Berechnungen, die `Optional<T>` liefern.**
  - **Die Umwandlung in einen `Stream<T>`, für eine Kompatibilität mit dem Stream-API z. B. für Frameworks, die auf Streams arbeiten,**

Die Klasse Optional<T>

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

Optional<T> in JDK 9

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

why ifPresentOrElse(…) ?

JDK8

```
public class Example1 {

    public static void main(String[] args) {

        Optional<String> welcomeString = getWelcomeString();

        welcomeString.ifPresent(System.out::println);

    }

    private static Optional<String> getWelcomeString() {
        return Optional.of("Hi JUG Zürich");
    }

}
```

why ifPresentOrElse(…) ?

JDK 8

```
public class Example1 {

    public static void main(String[] args) {

        Optional<String> welcomeString = getWelcomeString();

        if(welcomeString.isPresent()) {
            System.out.println(welcomeString.get());
        }else {
            System.out.println("Hi JUG Default ");
        }

    }

    private static Optional<String> getWelcomeString() {
        return Optional.of("Hi JUG Zürich");
    }

}
```

Optional<T>::ifPresentOrElse() mit Java 8

- **Die Funktionalität von `ifPresentOrElse()` lässt sich in Java 8 nur umständlich realisieren => Kombination von `isPresent()` / `get()` und Negativfall**

```
final Optional<String> optCustomer = findCustomer("UNKNOWN");
if (optCustomer.isPresent())
{
    System.out.println("found: " + optCustomer.get());
}
else
{
    System.out.println("not found“);
}
```

ifPresentOrElse(Consumer<? super T>, Runnable)

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

JDK 9

```
public class Example1 {

public static void main(String[] args) {

        Optional<String> welcomeString = getWelcomeString();

        welcomeString.ifPresentOrElse(System.out::println, () -> System.out.println("Hi JUG XXX"));

    }

    private static Optional<String> getWelcomeString() {
        return Optional.of("Hi JUG Zürich");
    }


}
```

ifPresentOrElse(…) im JDK

```
public void ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction) {
    if (value != null) {
        action.accept(value);
    } else {
        emptyAction.run();
    }
}
```

Optional<T> in JDK 9

- 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

why or(…) ?

JDK 8

```
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");

    }
```

or(…)

-

JDK 9

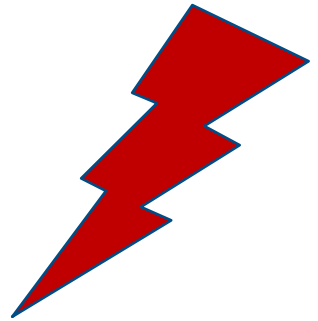
```
Optional<Double> optBalance = getDebitCardBalance().
                              or(()->getCreditCardBalance()).
                              or(()->getPayPalBalance());
```

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

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

or(…) im JDK

```
public Optional<T> or(Supplier<? extends Optional<? extends T>> supplier) {
    Objects.requireNonNull(supplier);
    if (isPresent()) {
        return this;
    } else {
        @SuppressWarnings("unchecked")
        Optional<T> r = (Optional<T>) supplier.get();
        return Objects.requireNonNull(r);
    }
}
```



NullPointerException if the supplying function is null or produces a null result

Optional<T> in JDK 9

- 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

why stream() ?

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);

    }

}
```

why stream() ?

JDK8

```
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).
    }

}
```

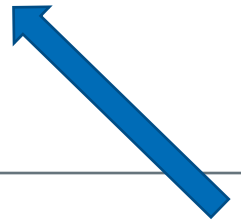
stream()

- `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);
    }

}
```



stream() im JDK

```
public Stream<T> stream() {
    if (!isPresent()) {
        return Stream.empty();
    } else {
        return Stream.of(value);
    }
}
```

Optional<T> in JDK 10 & 11

Die Klasse Optional<T> (Recap)

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

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

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

Die Klasse Optional<T>

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

Die Klasse Optional<T>

- **Experimentieren wir ein wenig in der JShell**

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

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

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

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

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

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

```
final Optional<String> optEmpty = Optional.empty();

if (!optEmpty.isPresent())
      System.out.println("check for empty JDK 10 style");

if (optEmpty.isEmpty())
      System.out.println("check for empty JDK 11 style");
```

Übungen PART 1 – Aufgabe 1 bis 8

https://github.com/Michaeli71/GLS-Update-Java-8-16.git

Collection Factory Methods

Collection Factory Methods Intro

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

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

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

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

Collection Factory Methods Intro

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

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

Collection Literals


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

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

Collection Factory Methods

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

```
final List<String> names = List.of("Tim", "Tom", "Mike");
final Set<Integer> numbers = Set.of(1, 3, 4, 2);

final Map<Integer, String> mapping = Map.of(5, "five", 6, "six", 7, "seven");
final Map<Integer, String> mapping2 = Map.ofEntries(entry(5, "five"),
                                                    entry(6, "six"),
                                                    entry(7, "seven"));
```

Collection Factory Methods

- **Verhalten recht intuitiv für Listen ...**

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

MAX
MORITZ
MIKE
```

Collection Factory Methods

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

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


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

Collection Factory Methods -- Besonderheiten bei der Konstruktion

```
static <E> List<E> of() {
    return ImmutableCollections.List0.instance();
}

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

...

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

Collection Factory Methods -- Besonderheit Space Efficiency

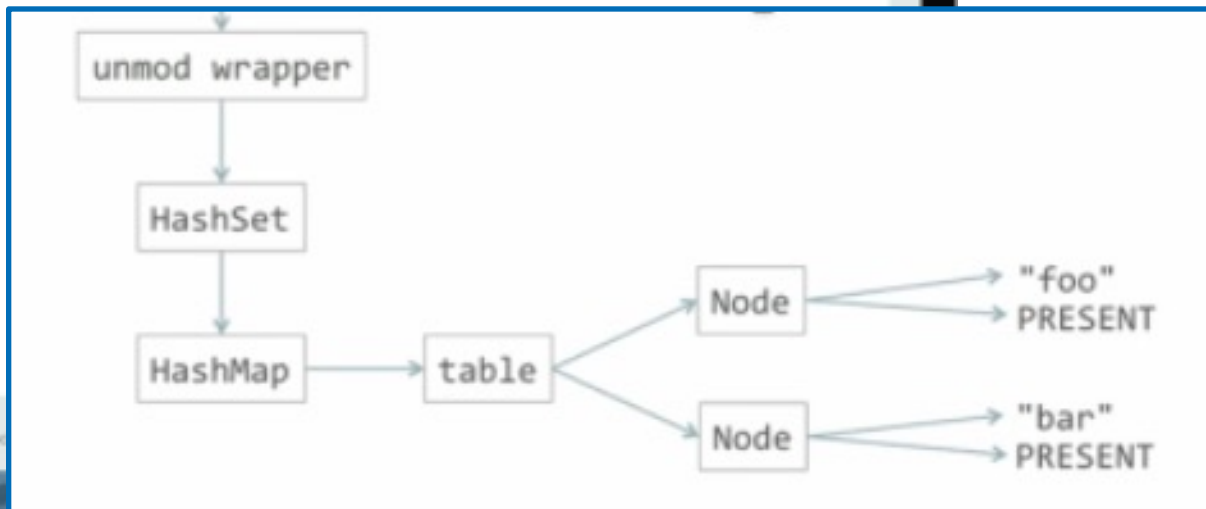
#JavaYoungPups

Space Efficiency

- Consider an unmodifiable set containing two strings

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

- How much space does this take? Count objects.
  - 1 unmodifiable wrapper
  - 1 HashSet
  - 1 HashMap
  - 1 Object[] table of length 3
  - 2 Node objects, one for each element



Collection Factory Methods -- Besonderheit Space Efficiency

#JavaYoungPups

Space Efficiency

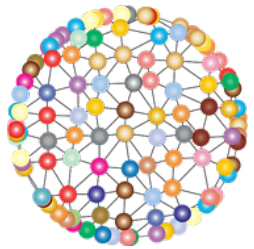
- Proposed field-based set implementation

```
Set<String> set = Set.of("foo", "bar");
```

- One object, two fields
  - 20 bytes, compared to 152 bytes for conventional structure
- Efficiency gains
  - lower fixed cost: fewer objects created for a collection of any size
  - lower variable cost: fewer bytes overhead per collection element

Java ORACLE

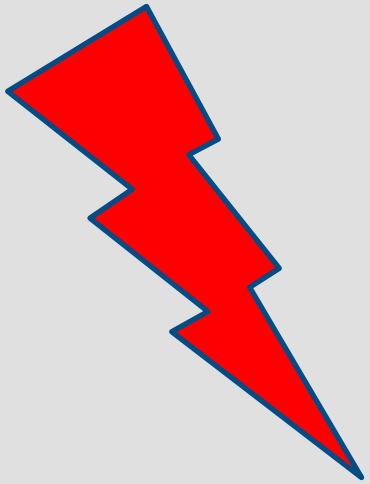
https://www.youtube.com/watch?v=OJrIMv4dAek



Was ist mit Sortierung? Wie erhalte ich ein TreeSet<E>?

CFM – Helper

```
public interface TypedCollections
{
    @SafeVarargs
    static <E> ArrayList<E> arrayListOf(E... values)
    {
        return new ArrayList<>(List.of(values));
    }

    @SafeVarargs
    static <E> LinkedList<E> linkedListOf(E... values)
    {
        return new LinkedList<>(List.of(values));
    }

    @SafeVarargs
    static <E> HashSet<E> hashSetOf(E... values)
    {
        return new HashSet<>(Set.of(values));
    }

    @SafeVarargs
    static <E> SortedSet<E> sortedSetOf(E... values)
    {
        return new TreeSet<>(Set.of(values));
    }

    static <K,V> HashMap<K,V> hashMapOf(Map<K,V> origMap)
    {
        return new HashMap<>(origMap);
    }

    static <K,V> SortedMap<K,V> sortedMapOf(Map<K,V> origMap)
    {
        return new TreeMap<>(origMap);
    }
}
```



Unmodifiable Collections
Kopien und Kollektoren

Unmodifiable Collections – Kopien

- Mit Java 9 wurden die Collection Factory Methods eingeführt, die es erlauben, auf lesbare Art unveränderliche Collections zu erzeugen

- **Es gab aber keine Möglichkeit, unveränderliche Kopien von beliebigen Collections zu erzeugen!**

- **Abhilfe 1**

```
final List<String> newCopyOfCollection = new ArrayList<>(names);
```

- **Abhilfe 2**

```
final Set<String> names = new HashSet<>();
names.add("Tim");
names.add("Tom");
names.add("Mike");
final Set<String> immutableNames = Collections.unmodifiableSet(names);
```

Unmodifiable Collections – Kopien

```
final var names = List.of("Tim", "Tom", "Mike", "Peter");
final List<String> copyOfNames = List.copyOf(names);
System.out.println("copyOfList: " +copyOfNames.getClass());

final var colors = Set.of("Red", "Green", "Blue");
final Set<String> copyOfColors = Set.copyOf(colors);
System.out.println("copyOfSet:  " + copyOfColors.getClass());

final var personAgeMapping = Map.of("Tim", 47L, "Tom", 12L, "Mike", 47L, "Max", 25L);

final Map<String, Long> copyOfMap = Map.copyOf(personAgeMapping);
System.out.println("copyOfMap:  " + copyOfMap.getClass());
```

=>

```
copyOfList:   class java.util.ImmutableCollections$ListN
copyOfSet:    class java.util.ImmutableCollections$SetN
copyOfMap:    class java.util.ImmutableCollections$MapN
```

Unmodifiable Collections – Stream API Collectors in JDK 10

Das umfangreiche Stream-API besitzt eine Menge an Kollektoren:

- **`toCollection()`, `toList()`, `toSet()` … – Sammlung der Elemente des Stream in die entsprechende Collection.**

Es gab aber keine Möglichkeit, unveränderliche Kopien von beliebigen Collections zu erzeugen!

- **Die Klasse java.util.stream.Collectors ermöglicht dies mit:**
  - toUnmodifiableList(),
  - toUnmodifiableSet() und
  - toUnmodifiableMap()

Unmodifiable Collections – Kopien

```
final var names = new ArrayList<>(List.of("Tim", "Tom", "Mike",
                                          "Peter", "Tom", Tim"));
final var immutableNames = names.stream().
                                 collect(Collectors.toUnmodifiableList());
System.out.println("immutableNames type: " + immutableNames.getClass());

final var uniqueImmutableNames = names.stream().
                                   collect(Collectors.toUnmodifiableSet());
System.out.println("uniqueImmutableNames content: " + uniqueImmutableNames);
System.out.println("uniqueImmutableNames type: " + uniqueImmutableNames.getClass());
```

=>

```
immutableNames type: class java.util.ImmutableCollections$ListN
uniqueImmutableNames content: [Peter, Mike, Tim, Tom]
uniqueImmutableNames type: class java.util.ImmutableCollections$SetN
```

- **Interessanterweise verhält sich hier der Kollektor anders als die Methode of() aus dem Interface Set, diese würde nämlich bei Duplikaten eine Exception auslösen.**

Stream API – Spezielle Kollektoren

Stream API Collectors in JDK 9

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.

Stream API Collectors in JDK 9

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

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

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

⇒ Schauen wir zunächst auf einfache Beispiele …

Stream API Collectors in JDK 9

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

Beispiel zum Einstieg:

```
final Set<String> result1 = programmming1.filter(name -> name.contains("Java")).
                                          collect(toSet());
```

Mit Filtering Collector:

```
final Set<String> result2 = programmming2.collect(
                            filtering(name -> name.contains("Java"), toSet()));
```

Als Ergebnis:

```
[JavaFX, Java, JavaScript]
```

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

Stream API Collectors in JDK 9

Beispiel zum Einstieg:

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

```
final List<String> programming = List.of("Java", "JavaScript", "Groovy", "JavaFX",
                                          "Spring", "Java");

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

Als Ergebnis:

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

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

Stream API Collectors in JDK 9

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

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

Als Ergebnis:

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

Stream API Collectors in JDK 9

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

Alle Hobbies als (eine) Menge:

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

Als Ergebnis:

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

Stream API Collectors in JDK 9

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()`

Stream API Collectors in JDK 9

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]}
```

Erweiterung in der Klasse `String`

Erweiterung in `java.lang.String`

- **Die Klasse String existiert seit JDK 1.0 und hat zwischenzeitlich nur wenige API-Änderungen erfahren.**

- **Mit Java 11 ändert sich das. Es wurden folgende Methoden neu eingeführt:**

  - `isBlank()`
  - `lines()`
  - `repeat(int)`
  - `strip()`
  - `stripLeading()`
  - `stripTrailing()`

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

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

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

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

- **Alle geben `true` aus.**

Erweiterung in java.lang.String: lines()

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

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

=>

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

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

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

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

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

=>

******************************
 -*-  -*-  -*-  -*-  -*-  -*-
```

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

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

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

Was passiert?

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

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

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

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

```
Exception in thread "main"
java.lang.OutOfMemoryError: 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)
```

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

```
"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.");
}
```

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

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

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

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

=>

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

Erweiterung in `Predicate<T>`

Das Interface `java.util.function.Predicate<T>`

- **Die Interface `Predicate<T>` ist 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);
```

Übungen PART 1 – Aufgaben 9 bis 13

https://github.com/Michaeli71/GLS-Update-Java-8-16.git

PART 2:
Multi-Threading mit CompletableFuture und Reactive Streams

Multi-Threading und die Klasse CompletableFuture<T>

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

Einstieg CompletableFuture<T>

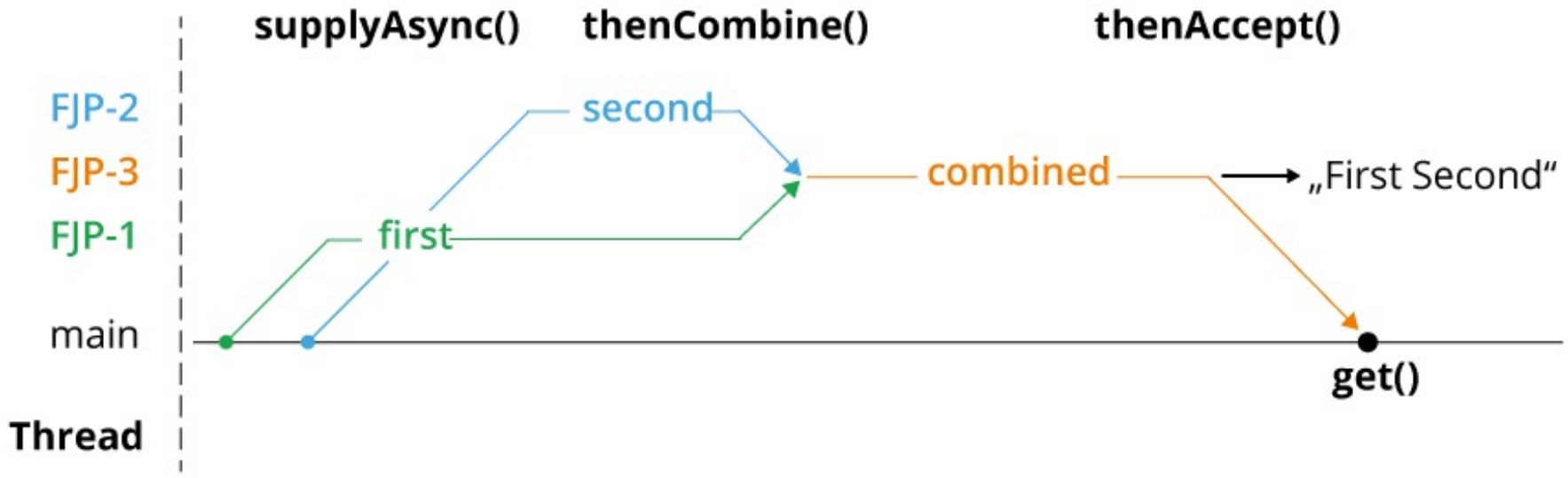
- Basisschritte
  - **supplyAsync**(Supplier<T>) => Berechnung definieren
  - **thenApply**(Function<T,R>) => Ergebnis der Berechnung verarbeiten
  - **thenAccept**(Consumer<T>) => Ergebnis verarbeiten, aber ohne Rückgabe
  - **thenCombine**(…) => Verarbeitungsschritte zusammenführen

- Beispiel

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

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

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

Einführung CompletableFuture<T>

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

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

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



Multi-Threading und die Klasse CompletableFuture<T>

Beispiel: Es sollen folgende Aktionen stattfinden:

- **Daten vom Server lesen**

- **Auswertung 1 berechnen**
- **Auswertung 2 berechnen**
- **Auswertung 3 berechnen**

- **Ergebnisse in Form eines Dashboards zusammenführen**

Wie könnte eine erste Realisierung aussehen?

Multi-Threading und die Klasse CompletableFuture<T>

- **Daten vom Server lesen** => `retrieveData()`
- **Auswertung 1 berechnen** => `processData1()`
- **Auswertung 2 berechnen** => `processData2()`
- **Auswertung 3 berechnen** => `processData3()`
- **Ergebnisse in Form eines Dashboards zusammenführen =>** `calcResult()`

- **Vereinfachungen: Daten => Liste von Strings, Berechnungen => Ergebnis `long` => `String`**

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

Multi-Threading und die Klasse CompletableFuture<T>

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

Die Berechnungen des Beispiels sind allerdings sehr vereinfacht …

- **keine Parallelität**
- **keine Abstimmung der Aufgaben (funktioniert, weil synchron)**
- **kein Exception-Handling**

- **Wir ersparen uns die Mühen und kaum verständliche und unwartbare Varianten mit `Runnable`, `Callable<T>`, Thread-Pools, `ExecutorService` usw., weil das selbst damit oft doch noch kompliziert und fehlerträchtig ist**

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

Multi-Threading und die Klasse CompletableFuture<T>

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

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

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

Wie bilden wir Fehler der realen Welt ab?

Multi-Threading und die Klasse CompletableFuture<T>

- In der realen Welt kommt es mitunter zu Exceptions
- Treten Fehler ab und an auf: Netzwerkprobleme, Zugriff aufs Dateisystem usw.

- **Annahme: Während der Datenermittlung würde eine `IllegalStateException` ausgelöst:**

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

- **Folglich würde die gesamte Verarbeitung unterbrochen und gestört!**

- Wünschenswert ist eine einfache Integration des Exception Handlings in den Ablauf
- Sowie weniger komplizierte Behandlung als bei Thread-Pools oder `ExecutorService`

Multi-Threading und die Klasse CompletableFuture<T>

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

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

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

- **Fallback-Wert bereitstellen, wenn eine Berechnung fehlschlägt:**

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

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

Wie bilden Verzögerungen der realen Welt ab?

Multi-Threading und die Klasse `CompletableFuture<T>`

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

Multi-Threading und die Klasse `CompletableFuture<T>`

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

- `public CompletableFuture<T> orTimeout(long timeout, TimeUnit unit)`

  => Die Verarbeitung wird mit einer Exception abgebrochen, falls das Ergebnis nicht innerhalb der angegebenen Zeitspanne berechnet wurde.

- `public CompletableFuture<T> completeOnTimeout(T value, long timeout, TimeUnit unit)`

  => Falls das Ergebnis nicht innerhalb der angegebenen Zeitspanne berechnet wurde, kann ein Defaultwert vorgegeben werden.

Multi-Threading und die Klasse `CompletableFuture<T>`

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

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

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

Multi-Threading und die Klasse `CompletableFuture<T>`

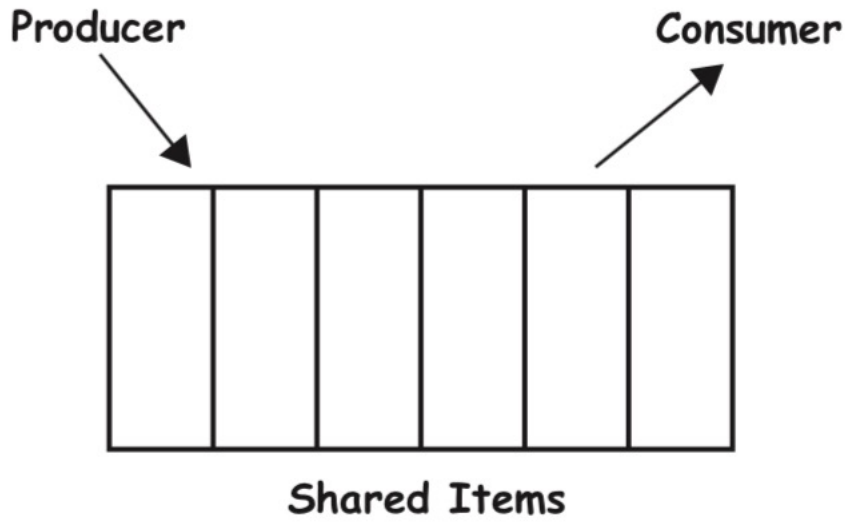
Annahme: Die Berechnung dauert mitunter mehrere Sekunden – diese soll spätestens nach 2 Sekunden abgebrochen werden und das Ergebnis 7 liefern:

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

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

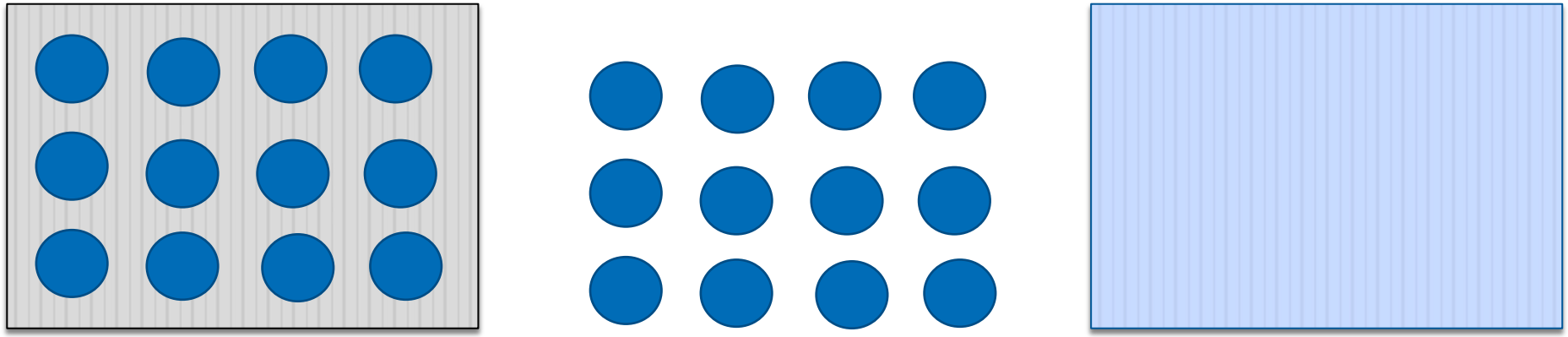
Reactive Streams

Producer - Consumer



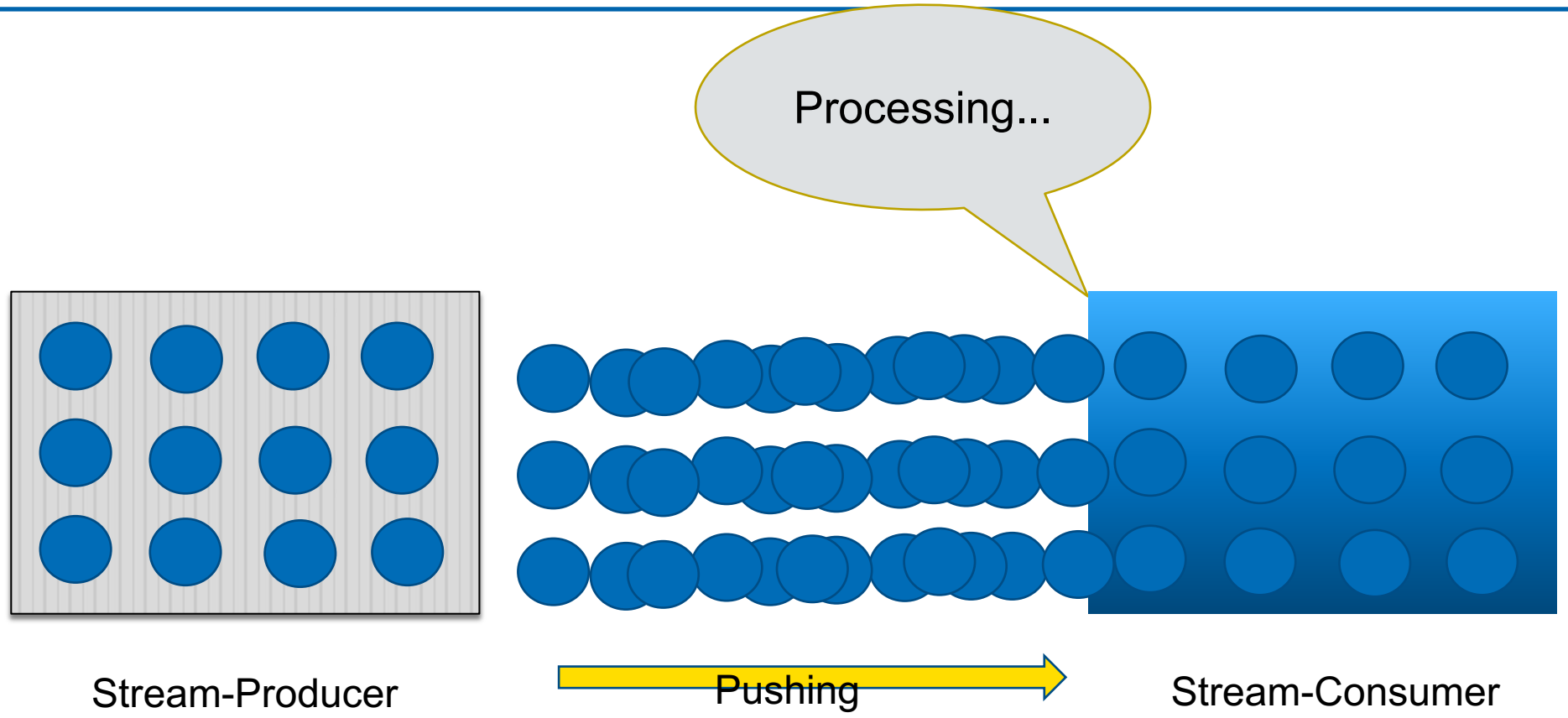
- **Coordination needed**
  - PUSH Producer pushes, Consumer processes directly
  - POLL Consumer polls, Produces creates on demand
  - Buffering needed

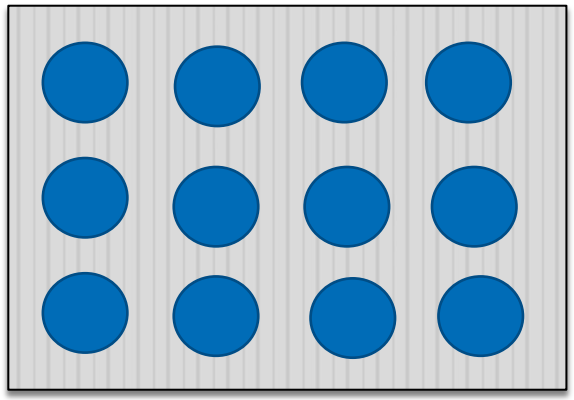
Scenario 1 - Push



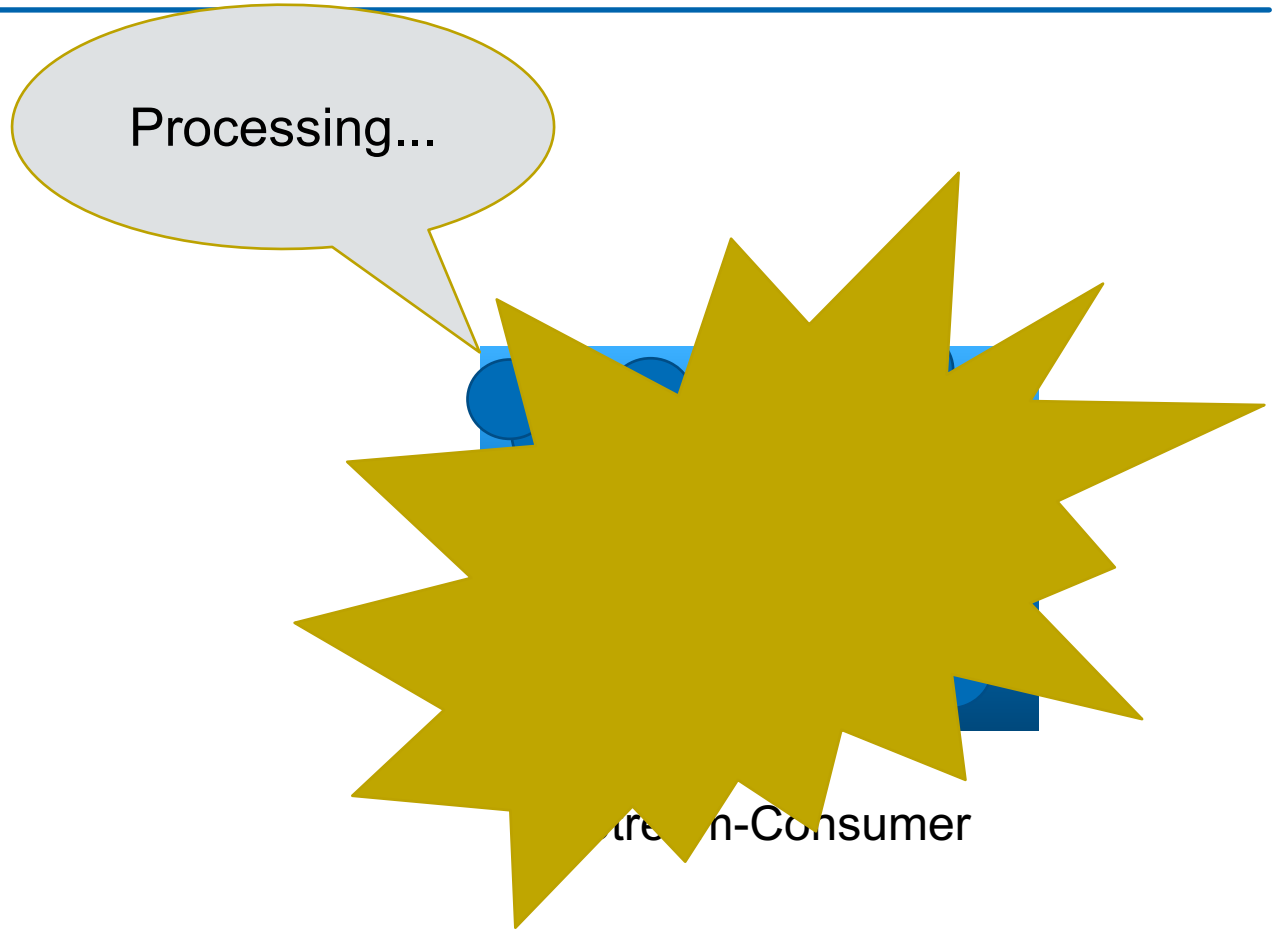
Stream-Producer

Stream-Consumer


Processing...
Stream-Producer
Pushing
Stream-Consumer

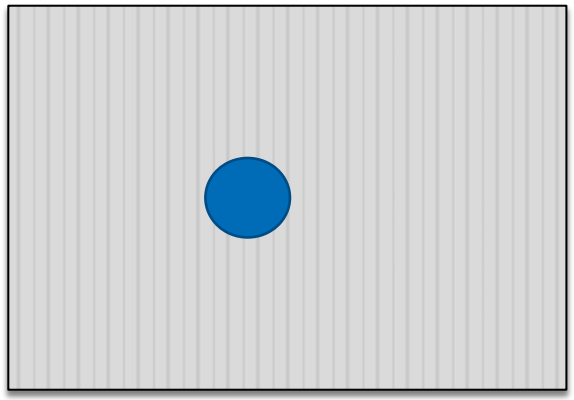


Stream-Producer



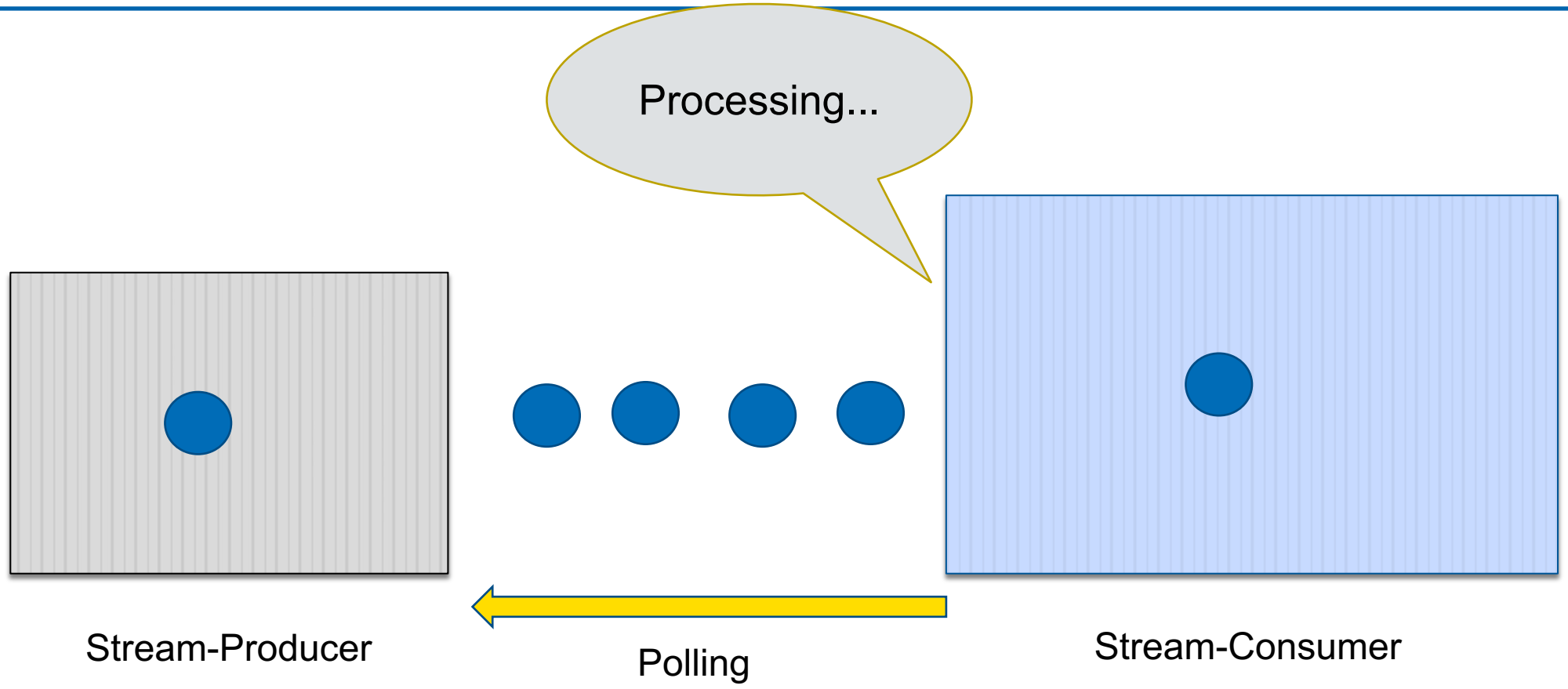
tre m-Consumer

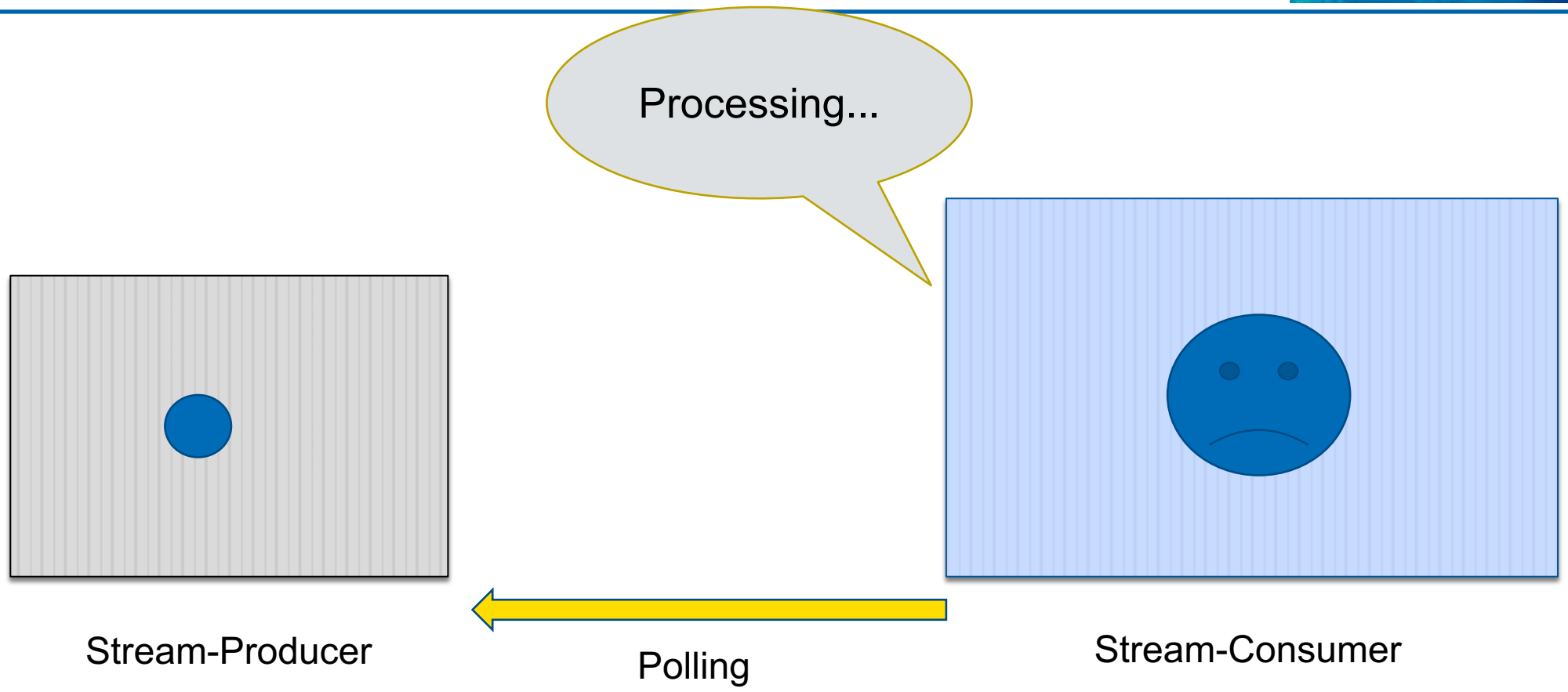
Scenario 2 - Polling



Stream-Producer

Stream-Consumer


Processing...
Stream-Producer
Polling
Stream-Consumer


Processing...
Stream-Producer
Polling
Stream-Consumer

Consequences

PUSH

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

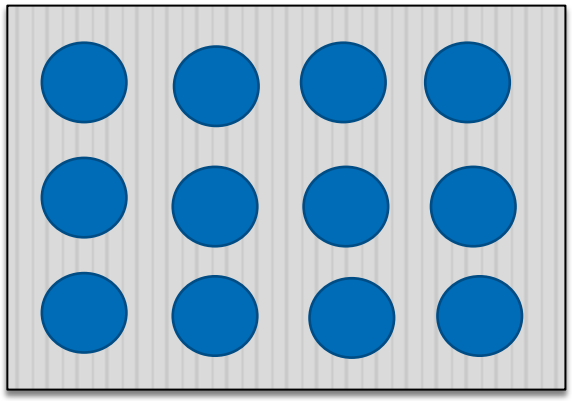
POLL

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)

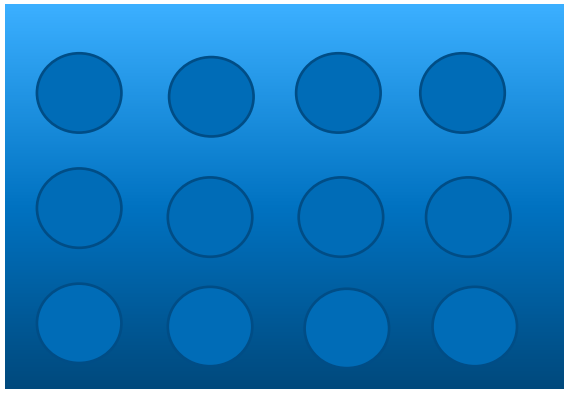
What if ?

Consumer can inform to the producer about it‘s capabilties and Producer can react according to Consumer‘s capabilties and needs.

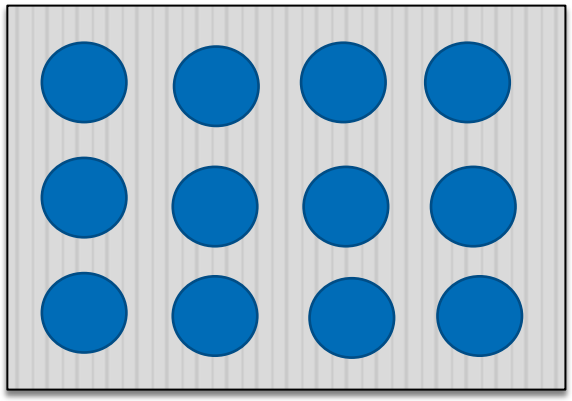


Stream-Producer

Hey I can handle only 12 at a time !

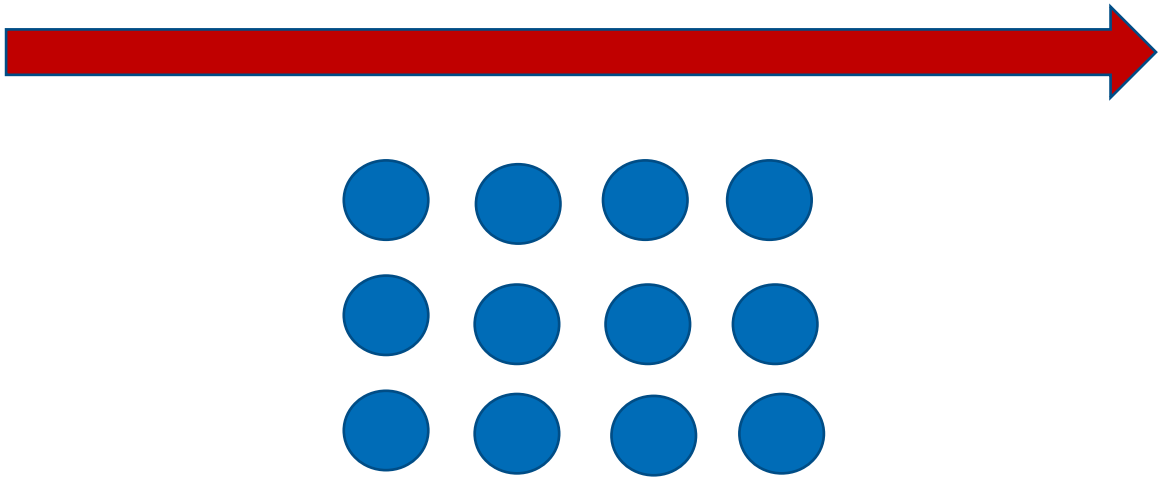


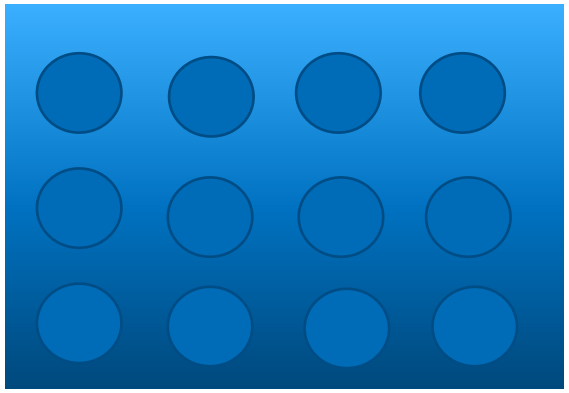
Stream-Consumer



Stream-Producer

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



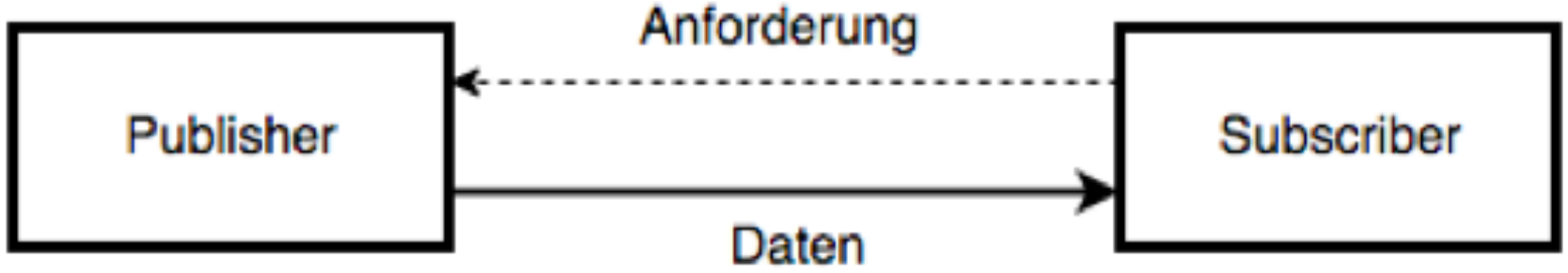


Stream-Consumer

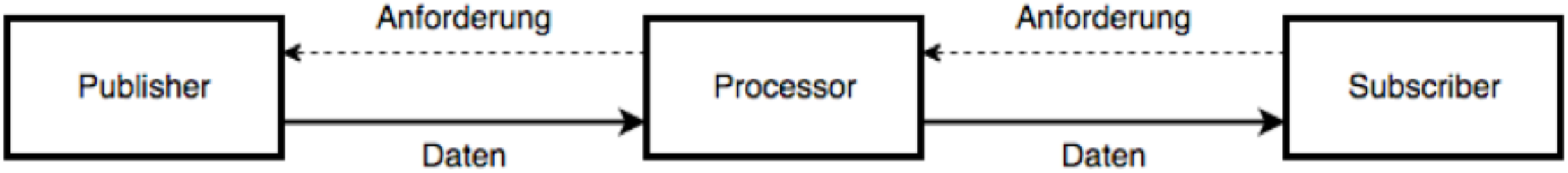
"… provide a standard for asynchronous stream processing with non-blocking back pressure. … aimed at runtime environments (JVM and JavaScript) .. "

https://www.reactive-streams.org

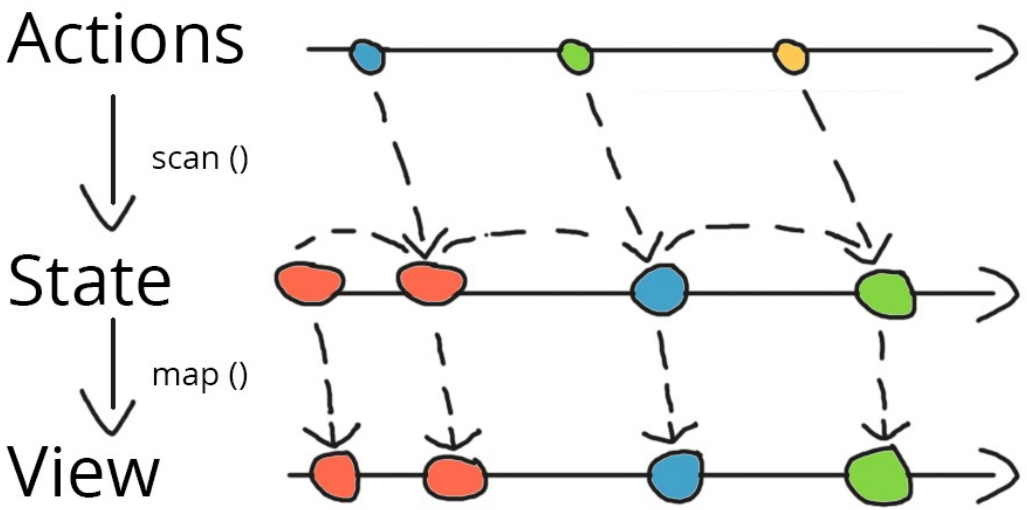
- Publisher veröffentlichen Daten und Subscriber verarbeitet Daten



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



Reactive Streams



Was haben wir gelernt?

- **Traditionell**: Verarbeitung oft mit **Engpässen** bei Kommunikation

- wenn ein Publisher/Subscriber zu langsam oder zu schnell arbeitet.
  => Gegenpart teilweise ausgelastet oder überlastet

  - **Eingangspuffer** als Abhilfe, aber größenbeschränkt
  - **Drosselung** des Senders => verschwendet potentiell Performance

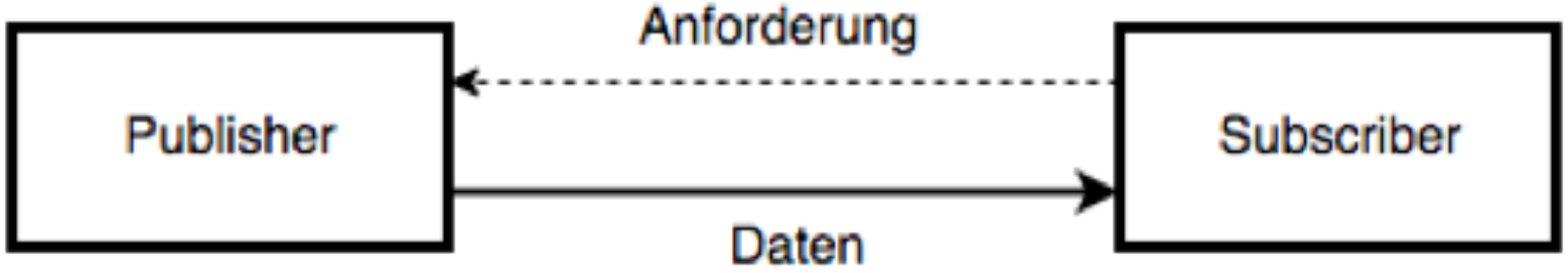
Reactive Streams

- **Standard** für **asynchrone Stream-Verarbeitung mit *Backpressure***
- Streams = eine Folge von Events, die im Laufe der Zeit erzeugt werden
- adressieren die Engpässe im Datenfluss

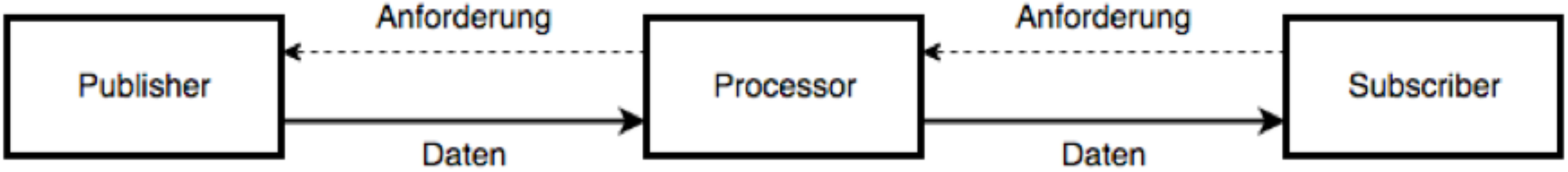
- von einander unabhängig ausgeführte Verarbeitungseinheiten kommunizieren darüber => asynchroner Flow von Daten und Anforderungen

- Frameworks wie Akka, RxJava, Vert.x und Reactor
- JDK 9: Publish-Subscribe-Framework durch die Klasse **Flow** sowie eine Utility-Klasse **SubmissionPublisher**

- Publisher veröffentlichen Daten und Subscriber verarbeitet Daten



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



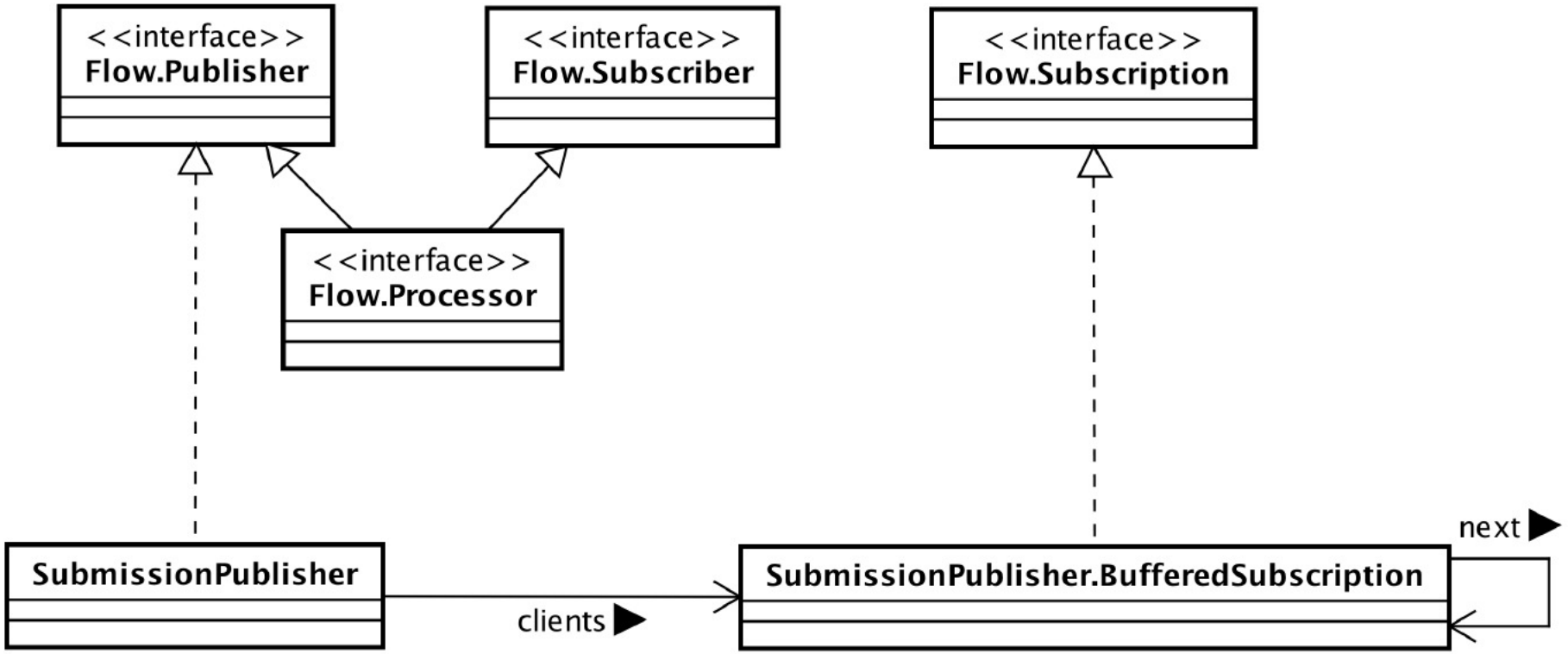
Publisher und Subscriber im Einsatz

- Die Interfaces in Flow ...

package java.util.concurrent;

- Flow
  - Publisher<T>
    - subscribe(Subscriber<? super T>) : void
  - Subscriber<T>
    - onSubscribe(Subscription) : void
    - onNext(T) : void
    - onError(Throwable) : void
    - onComplete() : void
  - Subscription
    - request(long) : void
    - cancel() : void
  - Processor<T, R>

Flow API / Reactive Streams API in JDK9



*Java – die Neuerungen in Version 9 bis 12 - Michael Inden

Subscriber

```
import java.util.concurrent.Flow.Subscriber;
import java.util.concurrent.Flow.Subscription;

public class ConsoleOutSubscriber implements Subscriber<Integer>
{
    private Subscription subscription;

    @Override
    public void onSubscribe(Subscription subscription) {

        this.subscription = subscription;
        subscription.request(1);
    }

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

    @Override
    public void onError(Throwable error)
    {
        System.out.println("onError(): Error occurred: " + error.getMessage());
    }

    @Override
    public void onComplete()
    {
        System.out.println("onComplete()");
    }
}
```

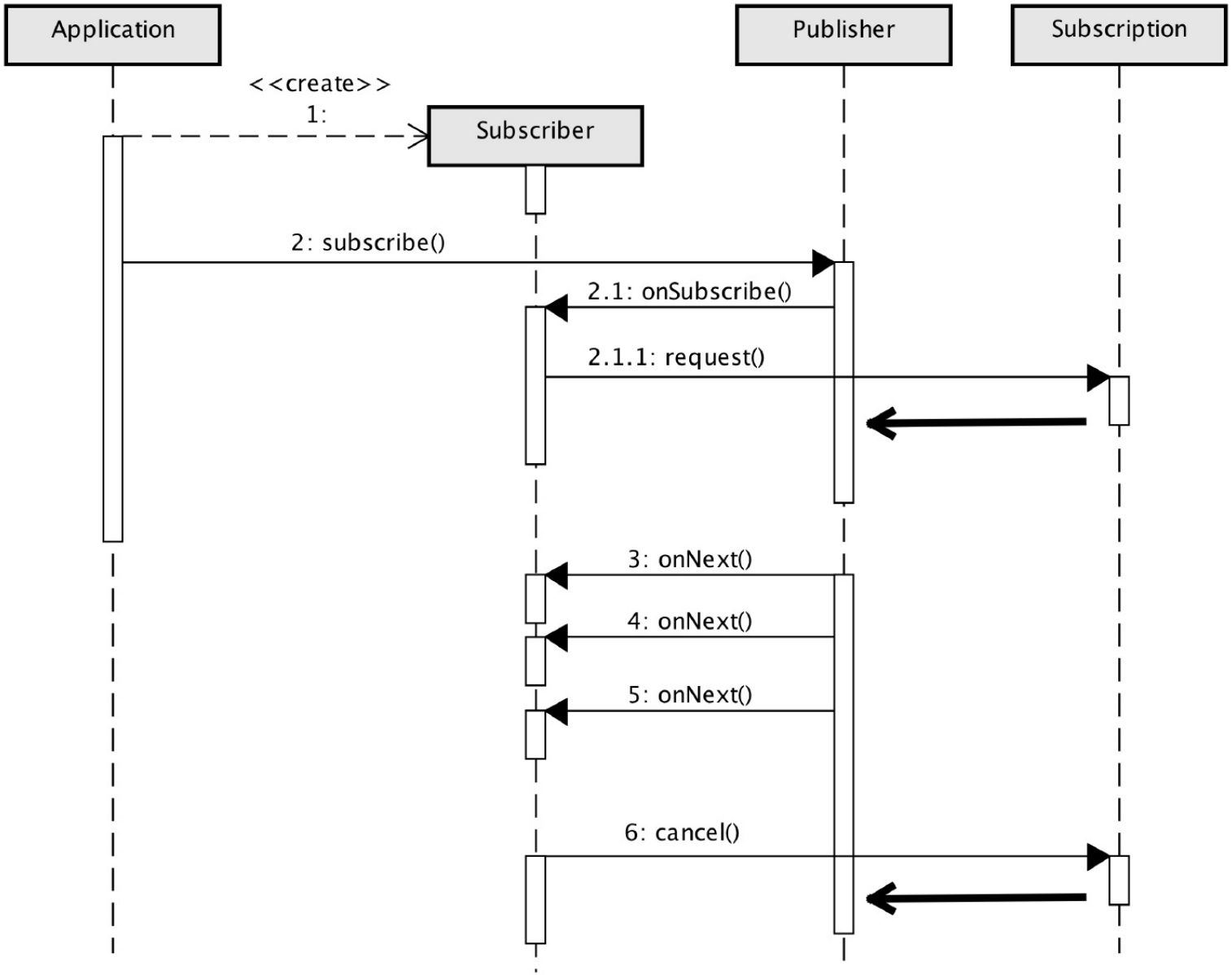
Simple Example

```
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()
```


Application
Publisher
Subscription
<<create>>
1:
Subscriber
2: subscribe()
2.1: onSubscribe()
2.1.1: request()
3: onNext()
4: onNext()
5: onNext()
6: cancel()

Subscription - Cancel

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

Console Progress

<terminated> ReactiveProgramming

```
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!»?

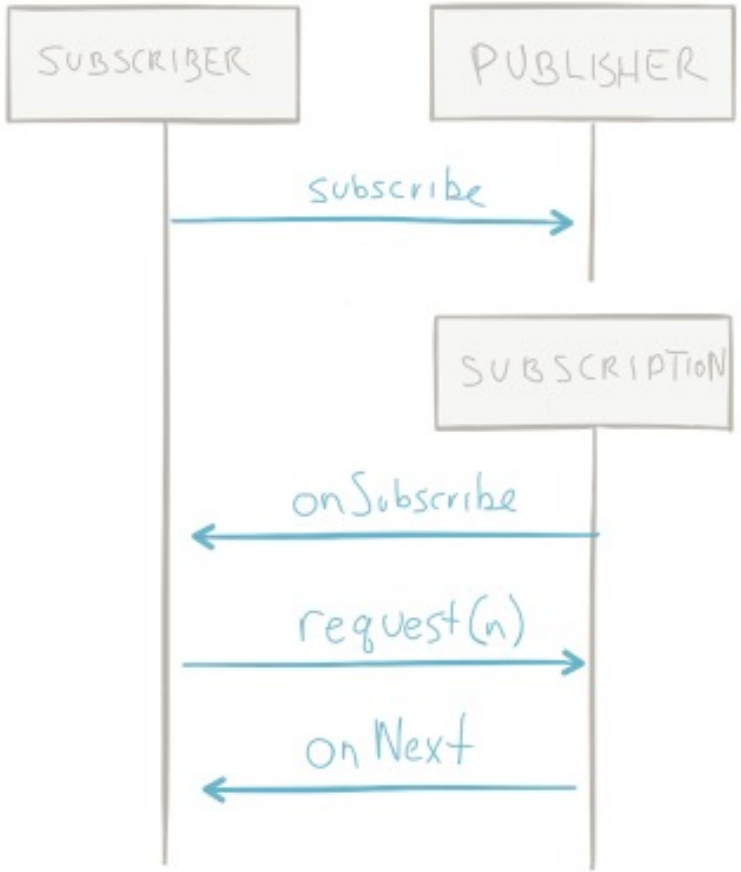
Flow-API - <Interface> Subscription

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

void java.util.concurrent.Flow.Subscription.cancel()

Causes the Subscriber to (eventually) stop receiving messages. Implementation is best-effort -- additional messages may be received after invoking this method. A cancelled subscription need not ever receive an `onComplete` or `onError` signal.

Publisher und Subscriber im Einsatz



Of particular interest is **request(long n)** in the **Subscription** interface. This is how a subscriber signals demand for work. When a subscriber signals demand by invoking **request(1)**, that's effectively a *pull*. When a subscriber signals demand by requesting more elements than the publisher is ready to emit, e.g, **request(100)**, that changes the flow to *push*.

https://blog.redelastic.com/a-journey-into-reactive-streams-5ee2a9cd7e29

Übungen PART 2

https://github.com/Michaeli71/GLS-Update-Java-8-16.git

PART 3: Weitere Neuerungen und Änderungen in den APIs und der JVM

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

Date API

Klasse LocalDate

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

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

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

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

Klasse LocalDate

- Start 7. Februar => Sprung um 150 Tage in die Zukunft => 7. Juli

- **Day-Stream**: **Tageweise Iteration** begrenzt auf 4
- **Month-Stream**: **Monatsweise Iteration** begrenzt auf 3
  => Vorgabe einer alternativen Schrittweite, hier Monate:

```
Day-Stream
1971-07-07
1971-07-08
1971-07-09
1971-07-10

3-Month-Stream
1971-02-07
1971-05-07
1971-08-07
```

Klasse Duration

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

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

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

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

Klasse Duration

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

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

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

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

Klasse Duration

- **toXXX() – wandelt in die entsprechende Einheit**
- **toXXXPart() – extrahiert den Teil der entsprechenden Einheit**

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

        // JDK 9:toXXX() und toXXXPart()
        System.out.println("toDays():         " + tenDaysSevenHoursThirdMinutes.toDays());
        System.out.println("toDaysPart():     " + tenDaysSevenHoursThirdMinutes.toDaysPart());
        System.out.println("toHours():        " + tenDaysSevenHoursThirdMinutes.toHours());
        System.out.println("toHoursPart():    " + tenDaysSevenHoursThirdMinutes.toHoursPart());
        System.out.println("toMinutes():      " + tenDaysSevenHoursThirdMinutes.toMinutes());
        System.out.println("toMinutesPart(): " + tenDaysSevenHoursThirdMinutes.toMinutesPart());
}
```

Klasse Duration

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


        // JDK 9:toXXX() und toXXXPart()
        System.out.println("toDays():         " + tenDaysSevenHoursThirdMinutes.toDays());
        System.out.println("toDaysPart():     " + tenDaysSevenHoursThirdMinutes.toDaysPart());
        System.out.println("toHours():        " + tenDaysSevenHoursThirdMinutes.toHours());
        System.out.println("toHoursPart():    " + tenDaysSevenHoursThirdMinutes.toHoursPart());
        System.out.println("toMinutes():      " + tenDaysSevenHoursThirdMinutes.toMinutes());
        System.out.println("toMinutesPart():  " + tenDaysSevenHoursThirdMinutes.toMinutesPart());
}
```

```
toDays():         10
toDaysPart():     10
toHours():        247
toHoursPart():    7
toMinutes():      14850
toMinutesPart():  30
```

java.util.Objects

Sonstiges – Objects und Null-Prüfungen (Recap)

- **Old-Style bei Null-Prüfungen**

```
public void withNullCheckOldStyle(Person person)
{
        if (person == null)
        {
                throw new NullPointerException("person cannot be null!");
        }
        …
```

- **Objects für Null-Prüfungen**

```
public void withNullCheckNewStyle(Person person)
{
        Objects.requireNonNull(person, "person cannot be null!");
        …
```

Sonstiges – Objects.requireNonNullElse…

- **Erweiterungen in Objects für Null-Prüfungen**
  - `requireNonNullElse(T obj, T fallbackValue)`
  - `requireNonNullElseGet(T obj,Supplier<? extends T> fallbackValueSupplier)`
- **Beide Methoden erleichtern `null`-Prüfungen und erlauben die Bereitstellung von Fallback-Werten**
- **Die Variante mit `Supplier<T>` kann vorteilhaft sein, wenn das Bereitstellen oder Berechnen des Fallback-Werts aufwendig ist. Mit `Supplier<T>` wird die Berechnung nur dann ausgeführt, wenn sie wirklich notwendig ist.**

Sonstiges – Einfache Fallback-Werte

- **Ausgangslage I: Spezielles Handling durch API, dass `null` liefert**

```
String media = request.getHeader("Accept");
media = media != null ? media : "Default Value";
```

Sonstiges – Einfache Fallback-Werte

- **Ausgangslage I: Spezielles Handling durch API, dass `null` liefert**

```
String media = request.getHeader("Accept");
media = media != null ? media : "Default Value";
```

- **Elvis-Operator (Proposal)**

```
String media = request.getHeader("Accept") ?: "Default Value";
```

- **JDK 9-Erweiterung**

```
String media = requireNonNullElse(request.getHeader("Accept"),
                                  "Default Value");
```

Sonstiges – Einfache Fallback-Werte

- **Ausgangslage II: Öffentliches API ohne `null`-Prüfung**

```
public long countPersonsOlderThan(final List<Person> persons, final int age)
{
   return persons.stream().
                  filter(person -> person.getAge() > age).
                  count();
}
```

Sonstiges – Einfache Fallback-Werte

- **Erweiterungen in Objects für `null`-Prüfungen mit Fallback => leere Collection**

```
public long countPersonsOlderThan(final List<Person> persons,
                                  final int age)
{
    final List<Person> adjustedPersons =
                       Objects.requireNonNullElse(persons,
                                                  Collections.emptyList());

    return adjustedPersons.stream().
                           filter(person -> person.getAge() > age).
                           count();
}
```

java.util.Arrays

Sonstiges – Die Klasse 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)

Sonstiges – Die Klasse Arrays

- **`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

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

Sonstiges – Die Klasse Arrays

- **`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
```

Sonstiges – Die Klasse Arrays

- **`Arrays.mismatch()` neu im JDK**

- **Prüft auf Abweichungen im Array – kann man auch auf spezielle Bereiche einschränken**

```
//                          0123401234567890123 4
final byte[] string1 = "ABC-DEF-GHIJKLMNOPQT".getBytes();
final byte[] string2 = "ABC-dEF-GHIJKLMNOPqT".getBytes();

System.out.println(Arrays.mismatch(string1, string2));    => 4

System.out.println(Arrays.mismatch(string1, 5, 20,
                                   string2, 5, 20));      => 13
```

java.io.InputStream / Reader

Sonstiges – Die Klasse InputStream

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

Sonstiges – Die Klasse InputStream

- **JDK 8: Alle Bytes aus einem Stream in einen anderen übertragen**

```
public static void copyStreamContent(final InputStream is,
                                      final OutputStream os) throws IOException
{
    final byte[] buffer = new byte[2048];
    int length;
    while ((length = is.read(buffer)) > 0)
    {
        os.write(buffer, 0, length);
    }
    os.flush();
}
```

- **JDK 9:** `is.transferTo(os)`

Sonstiges – Die Klasse InputStream

- **JDK 8: Alle Bytes aus einem Stream einlesen**

```
public static byte[] readAllBytesJdk8(final InputStream is) throws IOException
{
    try (final ByteArrayOutputStream os = new ByteArrayOutputStream())
    {
        copyStreamContent(is, os);
        return os.toByteArray();
    }
}
```

- **JDK 9:** `is.readAllBytes()`

Klasse java.io.Reader (JDK 10)

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

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

sr.transferTo(sw);

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

=>

SW: Hello
```

Erweiterung in der Klasse `Files`

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

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

```
final Path 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
```

Utility-Klasse java.nio.file.Files

- **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);
```

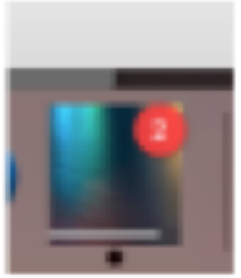
```
=>
```

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

Verschiedenes … Dies und das

Sonstiges – Taskbar-Support

```
if (Taskbar.isTaskbarSupported())
{
    final Taskbar taskbar = Taskbar.getTaskbar();
    taskbar.setIconImage(image);
    taskbar.setIconBadge(text);
    taskbar.setProgressValue(i);
    // ....
}
```



DEMO
TaskbarExample

Sonstiges – Graphisches

- **TIFF-Support**

https://www.ece.rice.edu/~wakin/images/

Sonstiges – Graphisches

- **HiDPI Support mit `java.awt.image.MultiResolutionImage`**

Sonstiges – Resource Bundles

UTF-8 Resource Bundles:

```
public static void main(final String[] args) throws Exception
{
    try (final InputStream propertyFile =
          new FileInputStream(".../unicode.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 = ❼ / \u277c
ohm = Ω / \u2126
sigma = = Σ / \u03a3
```

Problems ■ Javadoc ■ Declaration ■ Search

<terminated> UnicodeProperties [Java Application] /L

```
sigma = = Σ / Σ
money = € / €
ohm = Ω / Ω
coffee = ☕ / ☕
seven = ❼ / ❼
sun = ☀ / ☀
```

Sonstiges – HTML 5 Java Doc

https://docs.oracle.com/en/java/javase/14/docs/api/java.net.http/java/net/http/HttpResponse.html

Meistbesucht Erste Schritte Apple iCloud Yahoo Bing Google Wikipedia Facebook Twitter LinkedIn Wetter Online Yelp TripAdvisor

OVERVIEW MODULE PACKAGE CLASS USE TREE DEPRECATED INDEX HELP — Java SE 14 & JDK 14

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD SEARCH: Search

Module java.net.http
Package java.net.http

Interface HttpResponse<T>

Type Parameters:
T - the response body type

```
public interface HttpResponse<T>
```

An HTTP response.

An `HttpResponse` is not created directly, but rather returned as a result of sending an `HttpRequest`. An `HttpResponse` is made available when the response status code and headers have been received, and typically after the response body has also been completely received. Whether or not the `HttpResponse` is made available before the response body has been completely received depends on the `BodyHandler` provided when sending the `HttpRequest`.

This class provides methods for accessing the response status code, headers, the response body, and the `HttpRequest` corresponding to this response.

The following is an example of retrieving a response as a String:

```
HttpResponse<String> response = client
  .send(request, BodyHandlers.ofString());
```

The class `BodyHandlers` provides implementations of many common response handlers. Alternatively, a custom `BodyHandler` implementation can be used.

Since:
11

Nested Class Summary

Nested Classes

Modifier and Type	Interface	Description
static interface	`HttpResponse.BodyHandler<T>`	A handler for response bodies.
static class	`HttpResponse.BodyHandlers`	Implementations of `BodyHandler` that implement various useful handlers, such as handling the response body as a String, or streaming the response body to a file.
static	`HttpResponse.BodySubscriber<T>`	A `BodySubscriber` consumes response body bytes and converts them into a higher-level Java type.

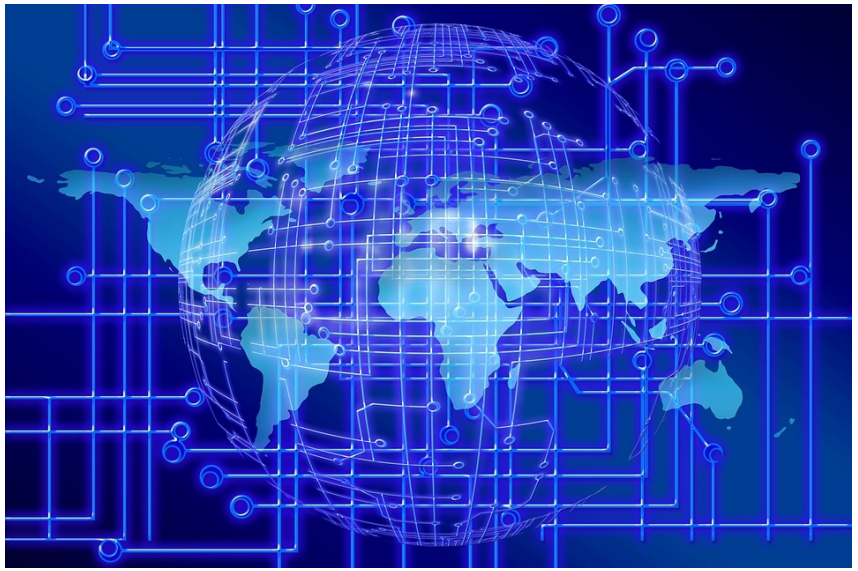
Sonstiges

- Kein Browser-Plugin mehr => Good bye Applets
- G1 wird neuer Standard GC
- Unicode 8 – Support 
- Deprecation der Typen `Observer` und `Observable`
- Performance-Optimierung in String „COMPACT-STRINGS“
- Reflection und Unsafe: Die Klassen java.lang.invoke.`MethodHandle` und java.lang.invoke.`VarHandle`
- diverse weitere Änderungen

Übungen PART 3 – Aufgaben 1 bis 12

https://github.com/Michaeli71/GLS-Update-Java-8-16.git

HTTP/2 API



HTTP-Zugriff mit Java 8

- **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()));
}
```

HTTP-Zugriff mit Java 8

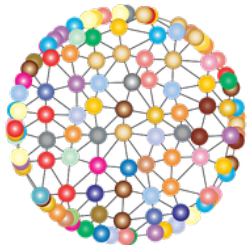
- **Content als String lesen**

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

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

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

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

HTTP/2 API Intro

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

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

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

printResponseInfo(response);


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

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



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

HTTP/2 API Async I

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

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

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

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

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

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

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

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

HTTP/2 API Async II

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

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

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

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

HTTP/2 API Weiteres

- **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())
```

Was macht HTTP/2 überhaupt?

HTTP/2 Background

- **HTTP/2 basiert auf Google SPDY**
- **binäre Datenübertragung statt der textorientierten bei HTTP 1.1**
- **trotzdem bleibt die Kompatibilität zu HTTP 1.1 gewahrt.**
- **beträchtliche Geschwindigkeitsverbesserungen von 10 bis 50 %**

HTTP/2 Background

- **Geschwindigkeit ist extrem wichtig für Nutzerzufriedenheit**
- **3 zentrale Faktoren**
  - **Verarbeitung im Server**
  - **Übertragung über das Netzwerk**
  - **Darstellung im Browser**
- **Übertragung konnten Entwickler bislang kaum beeinflussen**
- **HTTP/2 setzt einen klaren Fokus auf Performance und gibt Entwicklern verschiedene Möglichkeiten zur Optimierung der Seitenladezeit an die Hand**

⇒ **https://www.heise.de/ix/heft/WWW-Beschleuniger-3948333.html**

HTTP/2 Background

- **zentrale Limitierung von HTTP/1.1: nur ein Request pro TCP-Verbindung zum Server**
- **Browser öffnen üblicherweise sechs parallele Verbindungen, sodass maximal sechs Ressourcen gleichzeitig übertragen werden können. Alle übrigen Requests müssen auf eine freie Verbindung warten.**

Direct Compilation Launch Single-File Source-Code Programs

Direct Compilation

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

```
package direct.compilation;

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

```
java ./HelloWorld.java
```

```
Hello Execute After Compile
```

Direct Compilation – Zwei Public Klassen in 1 File

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

        System.out.printf("'%s' is a palindrome? => %b %n",
                  args[0], StringUtils.isPalindrome(args[0]));
   }
}
public class StringUtils
{
    public static boolean isPalindrome(String word)
    {
      return new StringBuilder(word).reverse().toString().equalsIgnoreCase(word);
    }
}
```

Direct Compilation – REST Call mit HTTP/2 API

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

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

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

Direct Compilation – Shebang Execution

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

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

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

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

DEMO

JShell

```
michaeli — java ◂ jshell — 138×40
Michaels-MBP-2:~ michaeli$ jshell
|  Welcome to JShell -- Version 15
|  For an introduction type: /help intro

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

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

jshell> import java.time.*

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

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

DEMO

JShell – Was haben wir bisher gelernt?

- Berechnungen on the fly ausführen
- Variablen definieren
- Methoden definieren
- Praktisch forward referencing: Methode kann noch nicht definierte Methoden aufrufen

- Praktisch für ***KLEINE*** Experimente
- Editor seit Java 14 deutlich komfortabler, davor ziemlich katastrophal bezüglich Multi-Line-Editierung

- 3rd Party & Preview-Features
  - `jshell --class-path myOwnClassPath --enable-preview`

JShell-API

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

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

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

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

JShell-API

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

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

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

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

JShell-API

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

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

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

DEMO

Multi Release JARs

Multi Release JARs

- JAR-Dateien dienen bekanntlich dazu, Klassendefinitionen in Form eines ZIP-Archivs zu bündeln und anderen Applikationen bereitstellen zu können.
- jeweils genau eine Version einer kompilierten Java-Datei enthalten
- **mit JDK 9: Innerhalb eines JARs kann man nun eine Substruktur aufbauen und .class-Dateien bereitstellen, die für verschiedene, bestimmte Java- Versionen gedacht sind.**
- Erlaubt es, Programmcode zu hinterlegen, der je nach Java-Version auf neue / spezielle Funktionalitäten zugreift oder aber alternativen Sourcecode nutzt
- spezielles Property im Manifest namens `Multi-Release`

Multi Release JARs

- Ein `MANIFEST.txt` sowie spezielle Source-Verzeichnisse pro Version erforderlich:

```
├── MANIFEST.txt
├── src
│   └── com
│       └── inden
│           ├── Application.java
│           └── Generator.java
└── srcjdk9
    └── com
        └── inden
            └── Generator.java
```

Multi Release JARs

- Pro Version muss ein Compile-Durchgang erfolgen:

```
javac -d build --release 8 src/com/inden/*.java

javac -d build/META-INF/versions/9 --release 9 \
         srcjdk9/com/inden/*.java

javac -d build/META-INF/versions/15 --release 15 srcjdk15/com/inden/*.java
```

- Multi-Release JAR erzeugen:

```
jar --create --file multireleaseexample.jar --manifest MANIFEST.txt \
    --main-class=com.inden.Application -C build .
```

Multi Release JARs

- Als Multi-Release-JAR besitzt das JAR etwa folgende Struktur, in der die Versionen in einem Unterverzeichnis `versions` im Verzeichnis `META-INF` abgelegt werden:

```
├── META-INF
│   └── versions
│       ├── 9
│       │   └── com
│       │       └── inden
│       │           └── Generator.class
│       └── 15
│           └── ...
...
└── com
    └── inden
        ├── Application.class
        └── Generator.class
```

Multi Release JARs

- Applikationsstart mit Java 8

```
$ setJdk8
$ java -jar multireleaseexample.jar
Generated strings: [Java, 8]
```

- Applikationsstart mit Java 9

```
$ setJdk9
$ java -jar multireleaseexample.jar
Generated strings: [New, JDK 9, Collection, Factory, Methods]
```

- Applikationsstart mit Java 15

```
$ setJdk15
$ java -jar multireleaseexample.jar
THIS IS A
MULTI LINE
STRING
Generated strings: [New, JDK 9, Collection, Factory, Methods]
```

Übungen PART 3 – Aufgaben 13 bis 17

https://github.com/Michaeli71/GLS-Update-Java-8-16.git

Fazit

Positives

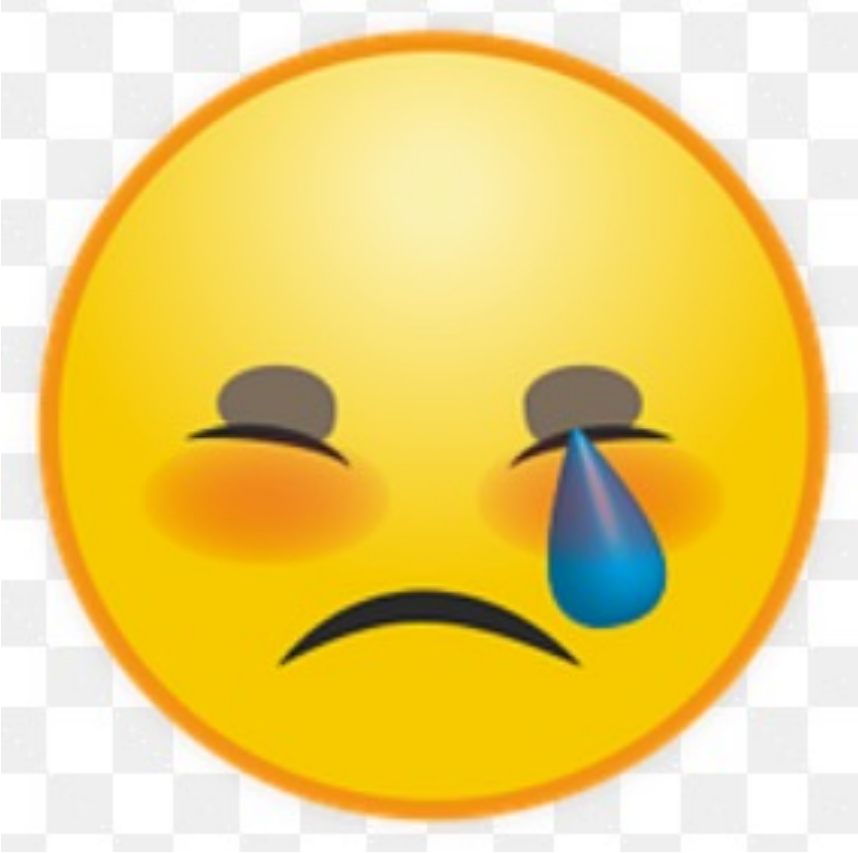
- eine paar Dinge aus Project COIN
- Switch / Records
- diverse praktische Erweiterungen in den APIs
- HTTP 2
- Modularisierung mit Project JIGSAW --
  aber leider (immer noch) kein wirklich gutes Tooling

Negatives

- **Mehrmalige Verschiebung von Java 9**
  Sept. 2016 => März 2017 => Juli 2017 => Sept. 2017

- **Folge-Release waren zwar pünktlich, aber manchmal (außer Java 14) ziemlich dünn bezüglich wichtigen Neuerungen**

- **Immer Mal wieder weniger als geplant**
  - keine Versionierung bei JIGSAW
  - kein JSON-Support
  - statt Collection-Literals nur Convenience-Methods
  - Statt ZIP nur TEE (ing)-Kollektor



Hilfe


HELP
IS ON THE
WAY


Java
Michael Inden
Challenge
Fit für das Job-Interview und die Praxis –
mit mehr als 100 Aufgaben
und Musterlösungen
dpunkt.verlag


Michael Inden
Java
Die Neuerungen in
Version 9 bis 14
Modularisierung, Syntax- und
API-Erweiterungen
dpunkt.verlag


5.
Auflage
Michael Inden
Der Weg zum
Java-Profi
Konzepte und Techniken für die
professionelle Java-Entwicklung
dpunkt.verlag

Questions?

Thank You