
Best of Java 9 – 15

<https://github.com/Michaeli71/JAX-London-Best-of-Java9-15.git>

Michael Inden

Independent SW consultant and author



- Michael Inden, Year of Birth 1971
- Diploma Computer Science, C.v.O. Uni Oldenburg
- ~8 ¼ Years @ Heidelberger Druckmaschinen AG in Kiel
- ~6 ¾ Years @ IVU Traffic Technologies AG in Aachen
- ~4 ¼ Years @ Zühlke Engineering AG in Zürich
- ~3 Years @ Direct Mail Informatics / ASMIQ in Zürich
- Independent Consultant and Trainer
- Author @ dpunkt.verlag

E-Mail: michael.inden@asmiq.ch

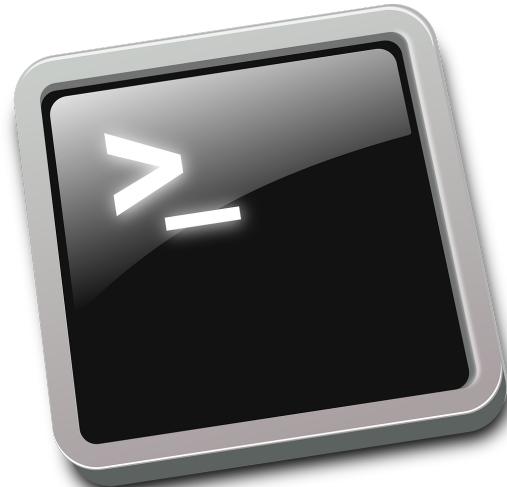
Blog: <https://jaxenter.de/author/minden>



Agenda

- **PART 1:** Syntax Enhancements & News and API-Changes in Java 9 to 11
 - **PART 2:** Multi-Threading with CompletableFuture and Reactive Streams
 - **PART 3:** Additional News and Changes in JDK 9 to 11
-
- **PART 4:** What's new in Java 12
 - **PART 5:** What's new in Java 13 und 14
 - **Outlook:** Java 15
-
- **Separate:** Java Modularization

New License Model



- If you are planning to or already distribute your software commercially, you should definitely consider the new release policy of Oracle when downloading Java 11!
- The Oracle JDK is now unfortunately NOT free for some scenarios - but during development it can still be used free of charge.
- Alternatives: OpenJDK (<https://openjdk.java.net/>) or 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

Workshop Contents

| JDK | Release Date | Development Time | LTS | Coverage in Workshop |
|----------------------|-----------------|------------------|---|----------------------|
| Oracle JDK 8 | 3 / 2014 | - | yes, <i>in the mean time commercial</i> | -/- |
| Oracle JDK 9 | 9 / 2017 | 3,5 years | - | Part 1, 2, 3 |
| Oracle JDK 10 | 3 / 2018 | 6 months | - | Part 1, 2, 3 |
| Oracle JDK 11 | 9 / 2018 | 6 months | yes, <i>commercial</i> | Part 1, 2, 3 |
| Oracle JDK 12 | 3 / 2019 | 6 months | - | Part 4 |
| Oracle JDK 13 | 9 / 2019 | 6 months | - | (Part 5) |
| Oracle JDK 14 | 3 / 2020 | 6 month | - | Part 5 |
| <i>Oracle JDK 15</i> | <i>9 / 2020</i> | <i>6 months</i> | - | <i>(Outlook)</i> |

PART 1:

Syntax Enhancements &

News and API-Changes in

Java 9 to 11

Syntax Enhancements in Java 9



Anonymous inner classes and the diamond operator

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

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



- **@Deprecated** mark obsolete sourcecode
- JDK 8: no parameters
- JDK 9: two parameter **@since** and **@forRemoval**

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

- `_` is **no** valid identifier any longer (semantically it has never been ;-))
- `final String _ = "Underline";`

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

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

FORGET ANYTHING YOU KNOW ABOUT...



JAVA INTERFACES!

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

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

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

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

Syntax Enhancements in Java 10 / 11



- Local Variable Type Inference
- New reserved word var for defining local variables, instead of explicit type specification

```
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
```

- possible if the concrete type for a local variable can be determined by the compiler using the definition on the right side of the assignment.

- Especially useful in the context of generics spelling abbreviations:

```
// 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);
```

- Especially if the type specifications include several generic parameters, **var** can make the source code significantly shorter and sometimes more readable.

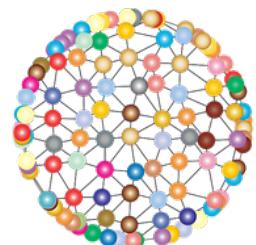
```
// var => Set<Map.Entry<String, Long>>
var entries = personAgeMapping.entrySet();
```

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

- But we have to use these Lambdas above:

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

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



**Wouldn't it be nice to
use var here too?**

- Yes!!!

- But the compiler cannot determine the concrete type purely based on these Lambdas
- Thus no conversion into var is possible, but leads to the error message>
«Lambda expression needs an explicit target-type».
- To avoid this mistake, the following cast could be inserted:

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

- Overall, we see that var is not suitable for Lambda expressions.

- Sometimes you may be tempted to just change the type with var:

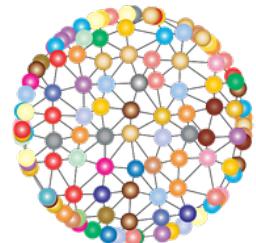
```
final List<String> names = new ArrayList<>();
names.add("Expected");
// names.add(42); // Compile error
```

- **Let's replace the concrete type with var and see what happens if we uncomment the line**

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



**Does this compile?
If yes, why?**



- **This will compile without problems? Why?**
- We use the Diamond Operators which has no clue of the types, so the compiler uses Object as Fallback:

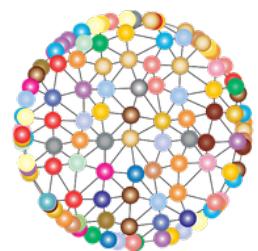
```
// var => ArrayList<Object>
final var mixedContent = new ArrayList<>();
mixedContent.add("Strange with var");
mixedContent.add(42);
```

- Short recapitulation: var is intended for **local variables** that are initialized directly.
- **More things that cause compilation errors:**

```
var justDeclaration;      // no declaration of value/definition
var numbers = {0, 1, 2}; // missing declaration of type
var appendSpace = str -> str + " "; // type not clear
```

- When using var, the **exact** type is always used and not a basic type

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



Is it desirable or possible to use var to declare attributes, parameters or return types?

YES ... BUT this is NOT possible because the type can't be determined unambiguously by the compiler.

- Java 10: For Lambdas var not supported, just the following:

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

- Java 11: var can be used in Lambdas

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

- What is the advantage?

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

News and API-Changes in Java 9 to 11

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

Process API



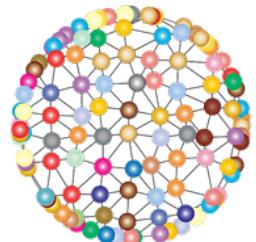
- Limited control and management of operating system processes prior to Java 9
- Example **Read PID: Often a different implementation is needed for each platform**

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

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



**What do you think about
this code? What is it not
doing?**



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

ProcessHandle can be used to **gather various other information** about processes besides the PID. The following methods are available for this purpose:

- **current()** – determines the current process as a ProcessHandle.
- **info()** – provides information about the process in the form of the internal interface ProcessHandle.Info, for example about user, command, etc.
- **info().command()** – returns the command as Optional<String>.
- **info().user()** – returns user as Optional<String> zurück
- **info().totalCpuDuration()** – extracts cpu time

In addition to information about the current process, information for all processes and all subprocesses for a process can be determined as follows:

- **allProcesses()** – Returns all processes as Stream<ProcessHandle>.
- **children()** – Determines for a process all its (direct) subprocesses as Stream<ProcessHandle>.
- **descendants()** – Determines for a process all its subprocesses as Stream<ProcessHandle>.

Stream API



The comprehensive Stream API was one of the major new features in Java 8.

takeWhile(...)

dropWhile(...)

ofNullable(...)

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

takeWhile(...)

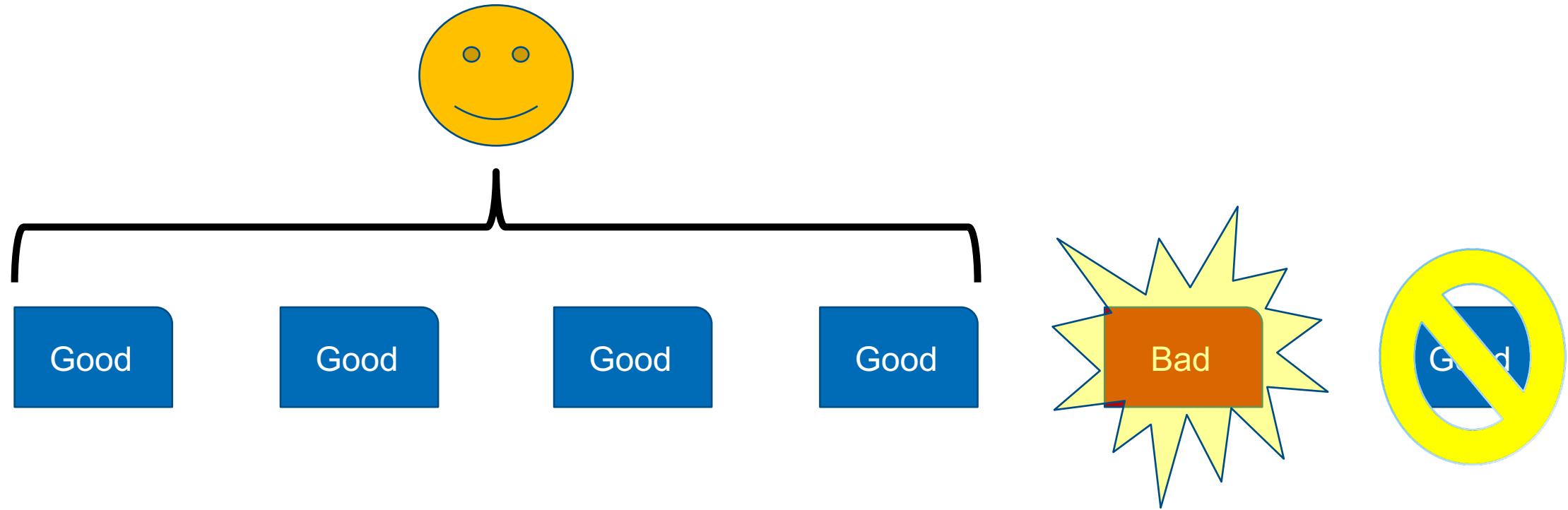
dropWhile(...)

`takeWhile(Predicate<T>)` – processes elements as long
as the condition is met

ofNullable(...)

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

Stream – Scenario



Stream – Filter

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

JDK 8

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

Stream – Filter

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

JDK 8

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

So, what do we do?

Google

Google-Suche

Auf gut Glück!

Google angeboten in: English Français Italiano Rumantsch



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

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

takeWhile(...)

dropWhile(...)

`dropWhile(Predicate<T>)` – skips/drops elements as long as the condition is met

ofNullable(...)

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

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

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

4. Good
5. Good
6. Good

- Combination of the two methods for extracting data:

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

takeWhile(...)

dropWhile(...)

ofNullable(...)

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

`ofNullable(T)` – returns a `Stream<T>` with one element if the passed element is not null. Otherwise an empty stream is generated.

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

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

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

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

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

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

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

```
<terminated> FirstNullableExample [Java Application] /Library/Java/JavaVirtualMachines/jd
1
Exception in thread "main" java.lang.NullPointerException
    at java.base/java.util.Objects.requireNonNull(Objects.java:369)
    at java.base/java.util.Optional.of(Optional.java:111)
    at java.base/java.util.stream.FindOps$FindSink$OfRef.get(FindOps.java:111)
```

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

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

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

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

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

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

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

0

No element



takeWhile(...)

dropWhile(...)

ofNullable(...)

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

- **iterate(T, Predicate<? super T>, UnaryOperator<T>)** – Creates a stream<T> with the given start value. The following values are calculated by the UnaryOperator<T> as long as the passed predicate<T> is fulfilled.

```
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
- Rather analog to for-loop :

```
for (int n = 1; n < 10; n++)  
    iterate(1, n -> n < 10, n -> n + 1);
```
- Since it's a stream, there are a variety of other possibilities.

Optional<T>



- Was introduced with Java 8 and facilitates the processing and modelling of optional values

▼ C F Optional<T>

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

▼ C F Optional<T>

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

▼ C F Optional<T>

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

- Initially with sound API, but **3 weak points** for the following usages:
 - The execution of actions even in a negative case,
 - The combination of the results of several calculations that provide Optional<T>.
 - Conversion into a Stream<T>, for compatibility with the stream API, e.g. for frameworks working on streams

- The enhancements to the `Optional<T>` class in JDK 9 address all three of the vulnerabilities listed before. The following methods are used for this purpose:
 - `ifPresentOrElse(Consumer<? super T>, Runnable)` – Allows an action to be executed in a positive or negative case.
 - `or(Supplier<Optional<T>> supplier)` – Elegantly combines multiple calculations.
 - `stream()` – converts the `Optional<T>` into a `Stream<T>`.

▼ C F Optional<T>

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



JDK 8

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

why ifPresentOrElse(...) ?

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

JDK 8

With Java 9 and the method `ifPresentOrElse()` the result evaluation of searches / actions can often be simplified:

JDK 9

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

▼ C F Optional<T>

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

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

JDK 8

JDK 9

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



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

- **or(Supplier<Optional<T>> supplier)** seems insignificant at first glance
- But call chains can be described with fallback strategies in a readable and understandable way, as the above example impressively shows.

▼ C F Optional<T>

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



JDK 8

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

- `Optional<T> => Stream<T>`: This is useful if you have a stream of optional values and want to [keep only those entries with valid values](#) (combination of methods `flatMap()` and `stream()`).
- Example of a stream consisting of `Optional<String>` elements, for example as a result of a [parallel search](#). At the end, the [results](#) should be [consolidated](#):

JDK 9

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

Optional<T> in JDK 10 & 11



- Introduced with Java 8, facilitates the handling and modeling of optional values.
- Java 9 add three valuable methods

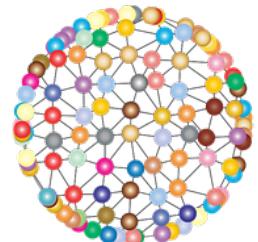
▼   Optional<T>

-   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





**What is potentially
problematic with `get()`?**



- The `get()` method for accessing a value seems to be harmless
- Sometimes `get()` get's called without check for existance with `isPresent()`
- If a value is missing this will result in a `NoSuchElementException`.
- **Normally a `get()` method is not necessarily expected to throw an exception.**
- **NEW IN JDK 10:**
- `orElseThrow()` as an alternative to `get()`, to express this fact directly in the API

- Let's do some experiments in 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
```

- Was updated up to Java 9 / 10 with valuable additions.
- Java 11 includes another new method: `isEmpty()`
- This makes the API analogous to that of collections and String
- Using this method you do not need to negate `isPresent()`

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

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

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

Collection Factory Methods



- Creating collections for a (smaller) set of predefined values can be a bit inconvenient in Java:

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

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

- Languages like Groovy or Python offer a special syntax for this, so-called collection literals ...



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

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



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

**Already in 2009, people thought about such things for Java.
Unfortunately this was not realized until now...**

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

- Behavior quite intuitive for lists ...

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

```
MAX
MORITZ
MIKE
```

- Behavior quite strange for 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 -- Specialities for construction



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

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

...

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

#JavaYoungPups

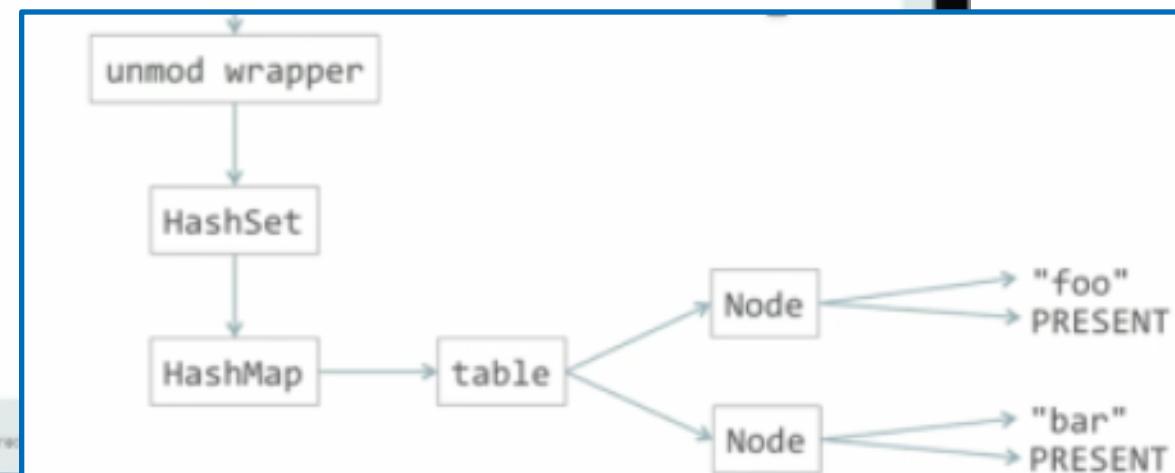
Space Efficiency

- Consider an unmodifiable set containing two strings

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

- How much space does this take? Count objects.

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



#JavaYoungPups

Space Efficiency

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



```
graph LR; set --> FieldBasedSet2; FieldBasedSet2 --> foo["foo"]; FieldBasedSet2 --> bar["bar"]
```

set → FieldBasedSet2 → "foo"
"bar"

 JavaOne 2015

Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

31

Unmodifiable Collections Copies and Collectors



- Java 9 brought Collection Factory Methods to create unmodifiable collections in a readable short manner

- **But there was no way to create unchangeable copies of any collections!**

- **Remedy 1**

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

- **Remedy 2**

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

```
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
```

Stream-API offers a lot of collectors:

- `toCollection()`, `toList()`, `toSet()` ... – all of them collect the elements of a stream into the mentioned corresponding collection.

But there was no way to create unchangeable copies of any collections!

- The new methods of class `java.util.stream.Collectors` provide exactly this:
 - `toUnmodifiableList()`,
 - `toUnmodifiableSet()` and
 - `toUnmodifiableMap()`

```
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
```

- Interestingly the behaviour of the collector differs from the result of the of() method, which raises an exception when passing duplicates

Stream API – Special Collectors



The extensive Stream API has a lot of collectors:

- `toCollection()`, `toList()`, `toSet()` ... – Collects the elements of the stream into the corresponding collection.
- `joining()` – Merge elements into a string
- `groupingBy()` – to prepare groupings. for example: histograms. Further collectors could also be used there (downstream collectors)

In the context of `groupingBy()`, however, there are some special use cases for which there was no collector before Java 9.

Two newly available collectors

- **filtering()** – Filtering the elements of the stream
- **flatMapting()** – Mapping and merging elements

⇒ Both are especially useful in the context of `groupingBy()`.

⇒ Let's first look at simple examples ...

The new `Collectors.filtering()` with analogy `filter()`

Example for the entry:

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

With new collector:

```
final Set<String> result2 = programming2.collect(  
                                filtering(name -> name.contains("Java")), toSet());
```

As a result:

[JavaFX, Java, JavaScript]

Second variant less intuitive and understandable than the first. Advantage only with `groupingBy()`

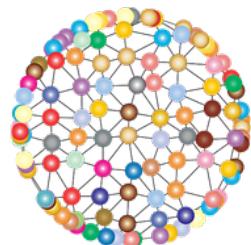
Example for the entry:

Suppose we wanted to create some kind of histogram based on the programming languages and frameworks. Let's start with a Java-8 implementation:

```
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);
```

As a result:

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



What do we do if during histogram preparation entries are also of interest that do not meet the condition?

Changed requirement: If entries that do not meet the condition are also of interest during histogram preparation, they would be lost if they were filtered beforehand. For our application, we must first group and then filter and only count those that are relevant:

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

As a result:

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

The new `Collectors.flatMapping()` with analogy `Collectors.mapping()` / `Stream.flatMap()`

Goal: All Hobbies as (one) set:

```
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);
```

As a result:

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

This first variant delivers unwanted nesting!

Goal: All Hobbies as one set:

```
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);
```

As a result:

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

This second variant is a bit less intuitive because of the call to stream(). Again advantage only with groupingBy()

Changed requirement and more practical example:

From a set of people with hobbies, we should group all those with the same firstname and create a collection of hobbies for each firstname:

```
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);
```

As a result:

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

Extensions in the Class String



- Class `String` exists since JDK 1.0 with only a few changes and additions
- Java 11 contains the following new methods:
 - `isBlank()`
 - `lines()`
 - `repeat(int)`
 - `strip()`
 - `stripLeading()`
 - `stripTrailing()`

- For strings, it was previously difficult or only possible with the help of external libraries to check whether they only contain whitespaces.
- Java 11 offers the method `isBlank()` which is based on `Character.isWhitespace(int)`

```
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());
}
```

- All of them print out true.

- When processing data from files, information often has to be broken down into individual lines. There is a method called `Files.lines(Path)` for this purpose.
- However, if the data source is a string, this functionality did not exist until now. JDK 11 provides the method `lines()`, which returns a `Stream<String>`:

```
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!
```

- Every now and then you are faced with the task of repeating an existing string n times.
- Up to now, this has required auxiliary methods of their own or those from external libraries. With Java 11 you can use the method `repeat(int)` instead:

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

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

=>

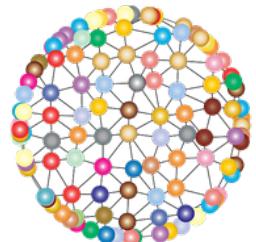
```
*****
-* - * - * - * - * -
```

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

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



What happens?



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

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

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

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

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

Exception in thread "main"

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

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

- The methods `strip()`, `stripLeading()` und `stripTrailing()` remove leading, trailing whitespace or both:

```
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'
```

Exercises PART 1

<https://github.com/Michaeli71/JAX-London-Best-of-Java9-15.git>

PART 2: Multi-Threading and Reactive Streams

- CompletableFuture<T>
 - Flow-API
-

CompletableFuture

- Since Java 5 there is the interface Future<T> in the JDK, but it often leads to blocking code by its method get().
- Since JDK 8 CompletableFuture<T> helps with the definition of asynchronous calculations
- Describing processes, enabling parallel execution
- Actions on higher semantic level than with Runnable or Callable<T>

- Basic steps
 - **supplyAsync(Supplier<T>)** => Define calculations
 - **thenApply(Function<T,R>)** => Process result of calculation
 - **thenAccept(Consumer<T>)** => Process result, but without return
 - **thenCombine(...)** => Merge processing steps

- Example

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

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

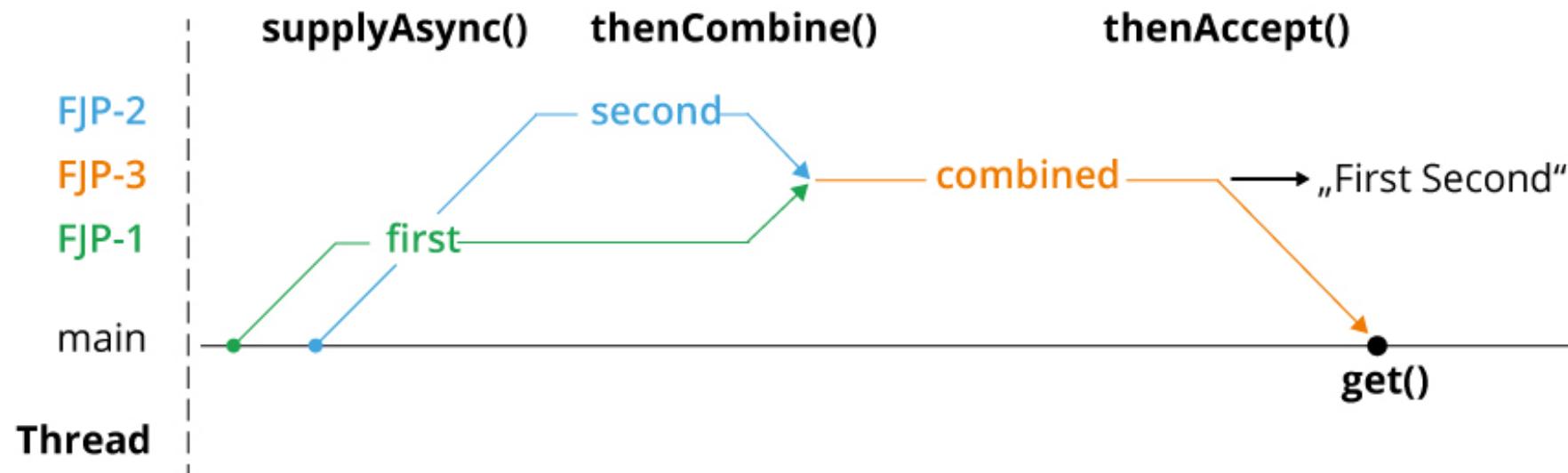
```
combined.thenAccept(System.out::println);  
System.out.println(combined.get());
```

Introduction to 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());
```

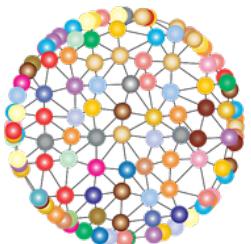


Example: The following actions are to take place:

- Read data from the server
- Calculate evaluation 1
- Calculate evaluation 2
- Calculate evaluation 3
- Merge results into a dashboard



**How could a first
realization look like?**



- **Read data from the server => retrieveData()**
- **Calculate evaluation 1 => processData1()**
- **Calculate evaluation 2 => processData2()**
- **Calculate evaluation 3 => processData3()**
- **Combine results in the form of a dashboard => calcResult()**
- **Simplifications: Data => List of strings, calculations => Result long => String**

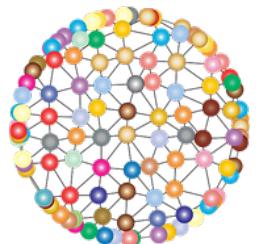
```
public void execute()
{
    final List<String> data = retrieveData();
    final Long value1 = processData1(data);
    final Long value2 = processData2(data);
    final Long value3 = processData3(data);
    final String result = calcResult(value1, value2, value3);
    System.out.println("result: "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

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

However, the calculations in the example are very simplified ...

- **no parallelism**
- **no coordination of tasks (works because it is synchronous)**
- **no exception handling**

- We avoid the trouble and hardly understandable and unmaintainable variants with **Runnable, Callable<T>, Thread-Pools, ExecutorService etc., because this itself is often still complicated and error-prone.**



**What must be changed for
parallel processing with
CompletableFuture<T>?**

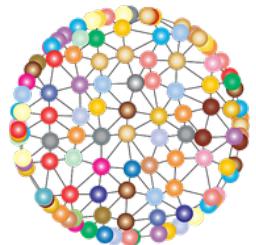
Multi-Threading and the Class CompletableFuture<T>



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

    // execute processing in parallel
    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));

    // combine results
    final String result = calcResult(cFValue1.get(), cFValue2.get(), cFValue3.get());
    System.out.println("result: " + result);
}
```



**How do we reflect
real-world errors?**

- In the real world, exceptions sometimes occur
- Errors occur from time to time: Network problems, file system access, etc.
- Assumption: An IllegalStateException would be triggered during data retrieval:

```
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)
```

- Consequently, all processing would be interrupted and disrupted!
- A simple integration of exception handling into the process flow is desirable.
- As well as less complicated handling than thread pools or ExecutorService

- The class **CompletableFuture<T>** provides the method **exceptionally()**

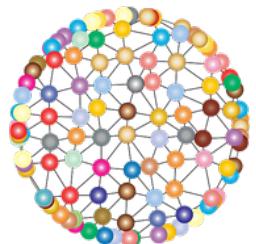
- Provide a fallback value if the data cannot be determined:

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

- Provide a fallback value if a calculation fails:

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

- **calculation can be continued even if one or more steps fail.**



**How do delays reflect
the real world?**

- In the real world, access to external resources sometimes causes delays
- In the worst case, an external partner does not answer at all and you might wait indefinitely if a call is blocked.
- Desirable: Calculations should be able to be aborted with time-out
- Assumption: The data retrieveData() sometimes takes a few seconds.
- Consequently, the entire processing would be disturbed!

Since JDK 9: The class CompletableFuture<T> provides the methods

- `public CompletableFuture<T> orTimeout(long timeout, TimeUnit unit)`
=> Processing is terminated with an exception if the result was not calculated within the specified time span.
- `public CompletableFuture<T> completeOnTimeout(T value, long timeout, TimeUnit unit)`
=> If the result was not calculated within the specified time span, a default value can be specified.

Assumption: The data determination sometimes takes several seconds, but should be aborted after 1 second at the latest:

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

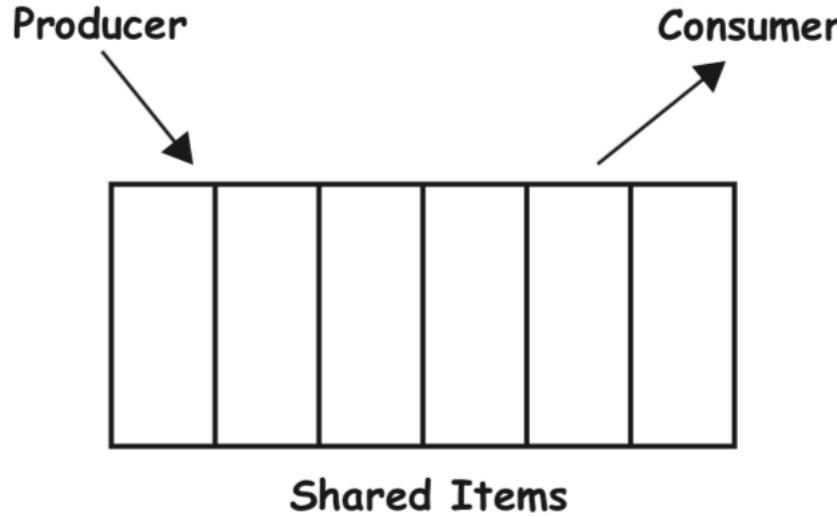
=> The processing can be continued promptly (without much delay or even blocking) even if the data determination should take longer.

Assumption: The calculation sometimes takes several seconds - it should be aborted after 2 seconds at the latest and deliver the result 7:

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

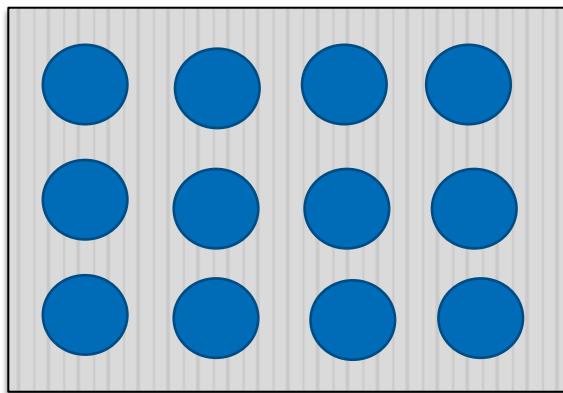
=> The calculation can be continued with a fallback value, even if one or more steps take longer.

Reactive Streams

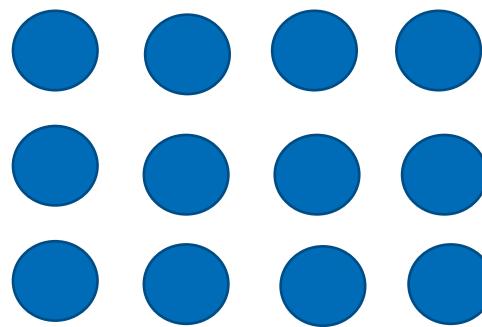


- **Coordination needed**
 - PUSH Producer pushes, Consumer processes directly
 - POLL Consumer polls, Produces creates on demand
 - **Buffering** needed

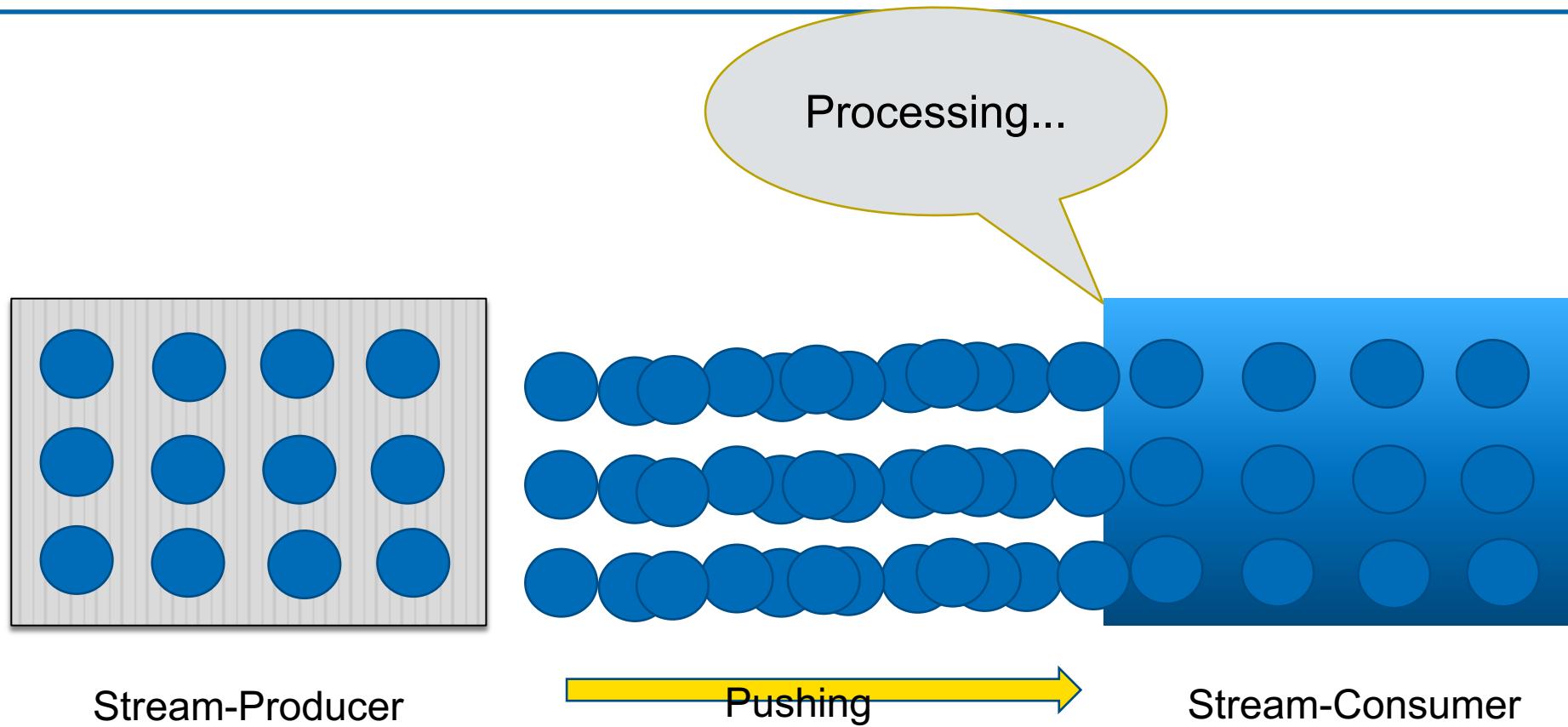
Scenario 1 - Push

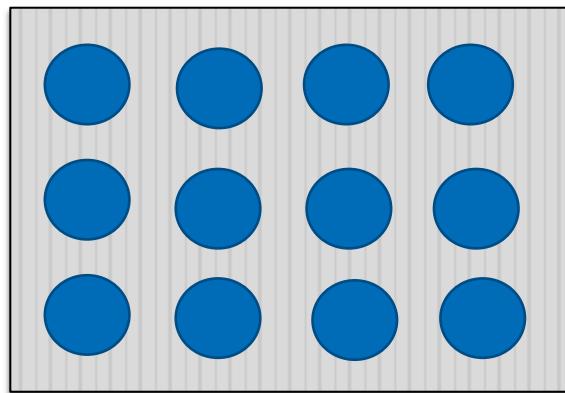


Stream-Producer

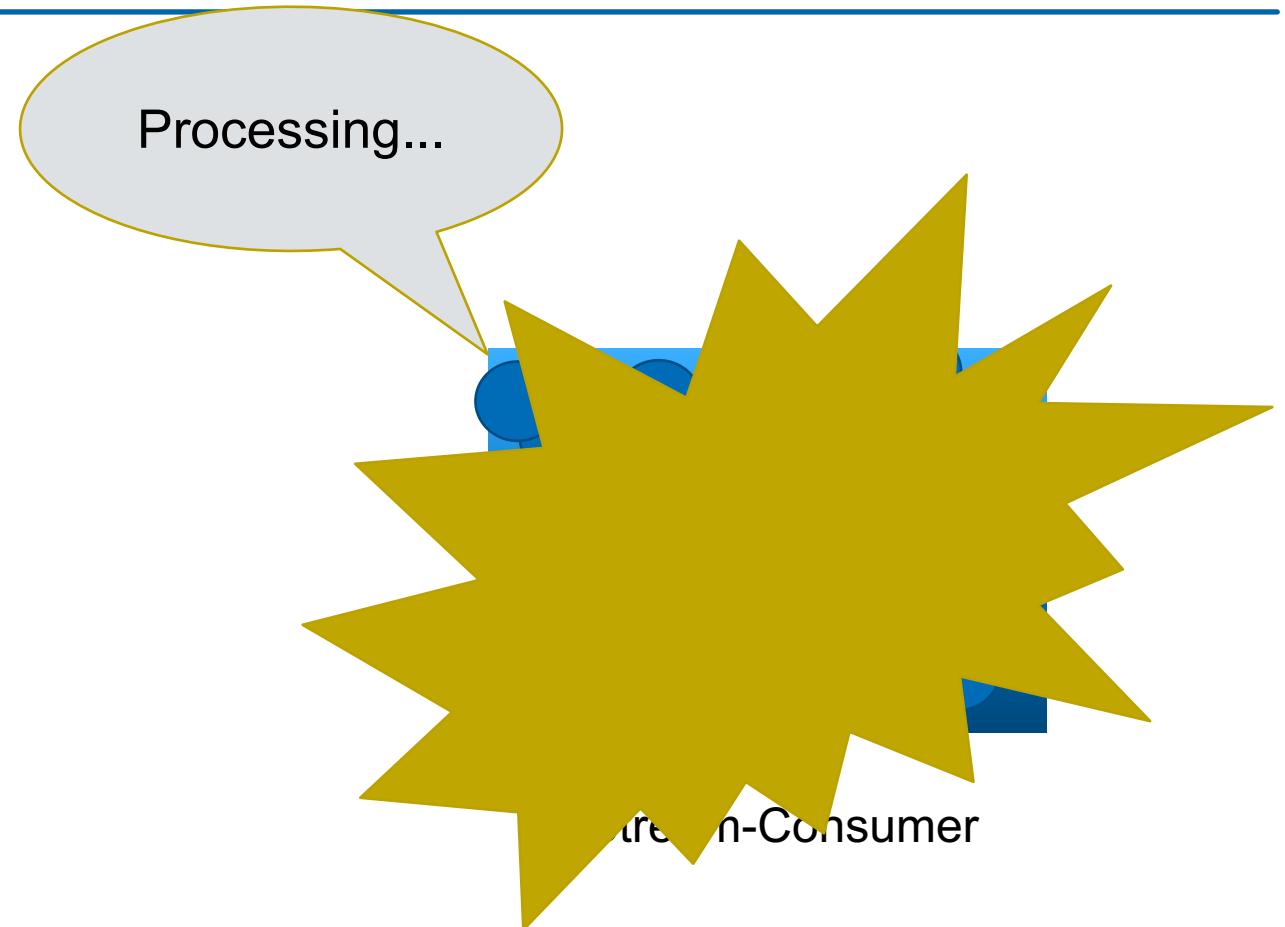


Stream-Consumer



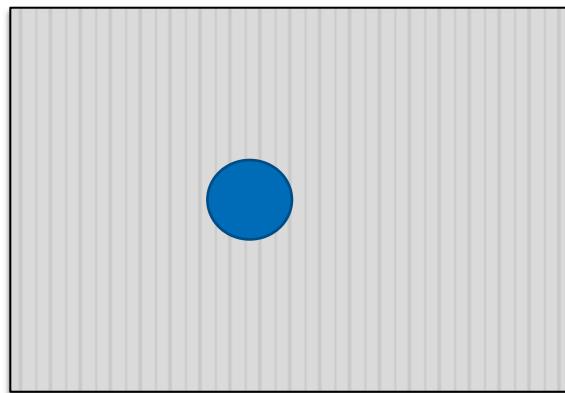


Stream-Producer



Stream-Consumer

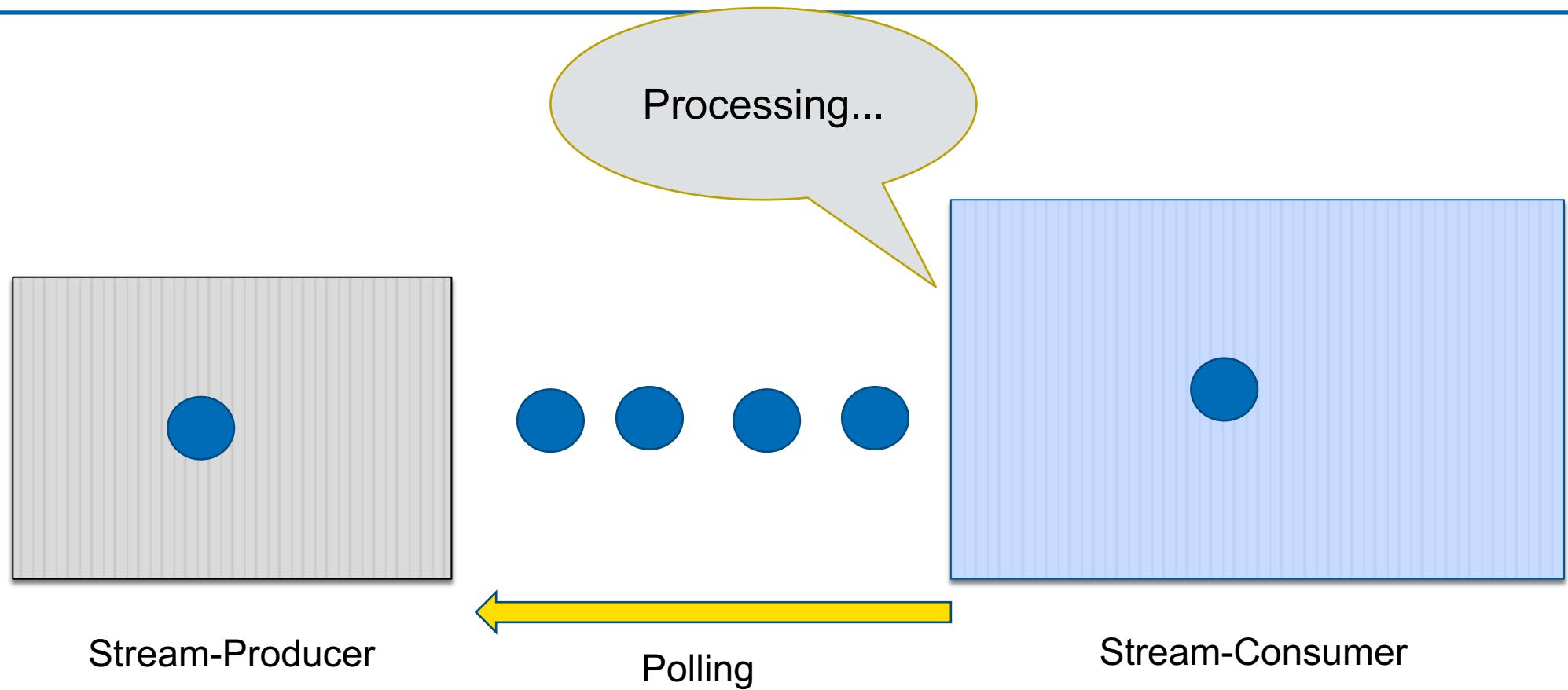
Scenario 2 - Polling

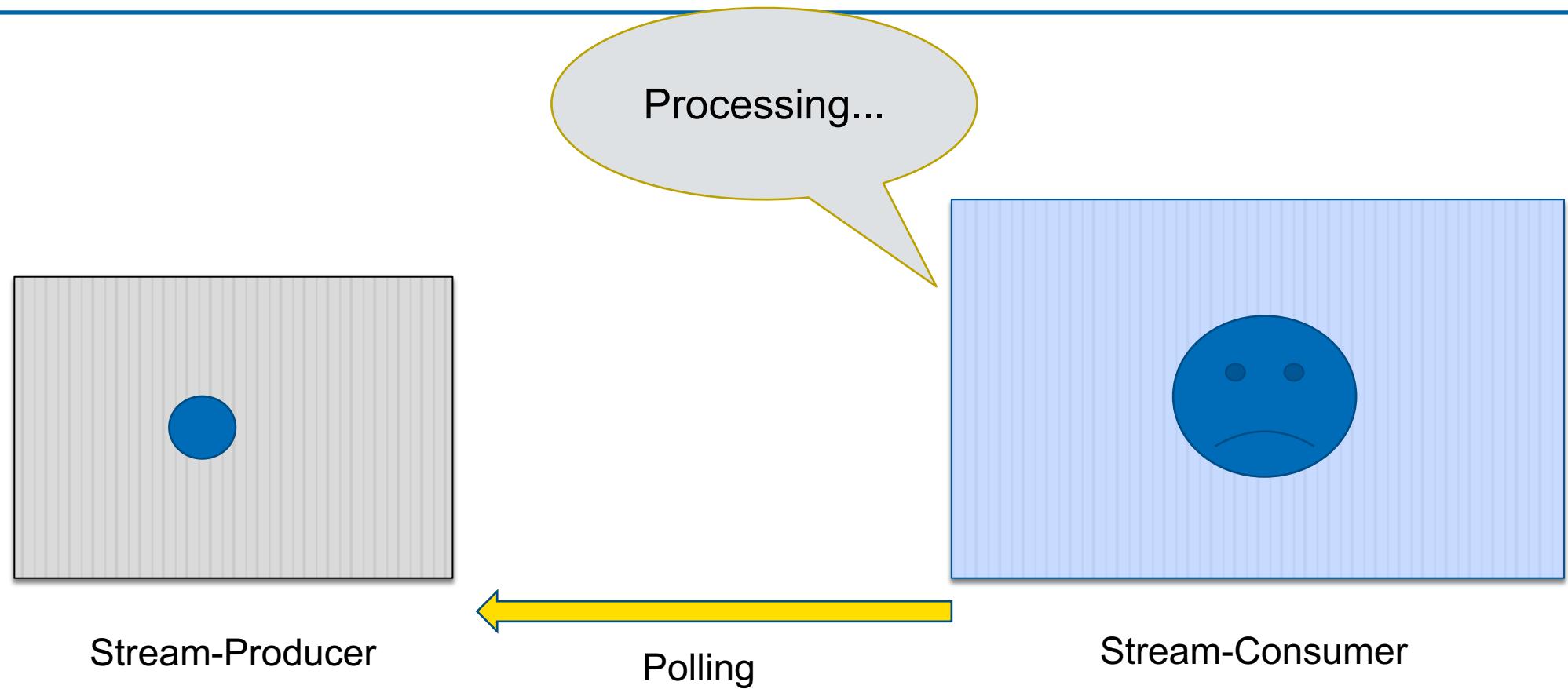


Stream-Producer



Stream-Consumer





PUSH

- 1) consumer has to process very fast in sync with the fast producer
- 2) Or consumer has to buffer at the consumer side
- 3) Or Producer has to slow down, but how does he knows this?

POLL

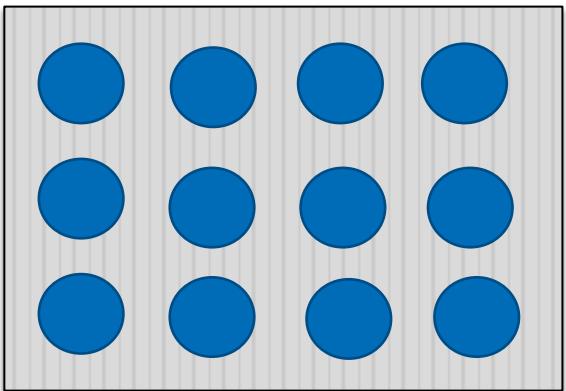
- 1) Fast consumer has to wait for slower producer
- 2) Slow consumer forces the producer to pause or to buffer data

✓ One-way communication

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

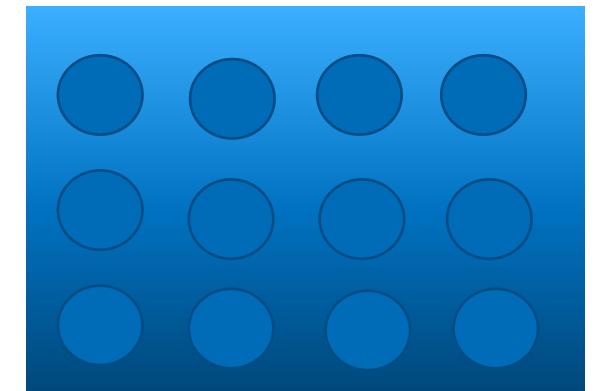
What if ?

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

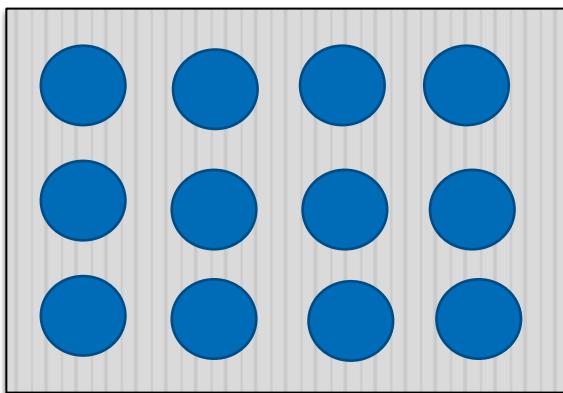


Stream-Producer

Hey I can handle only 12 at a time !

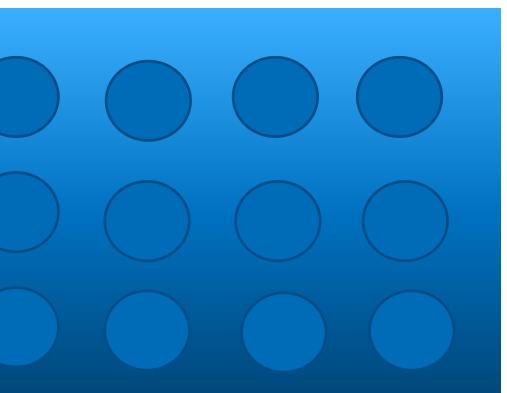
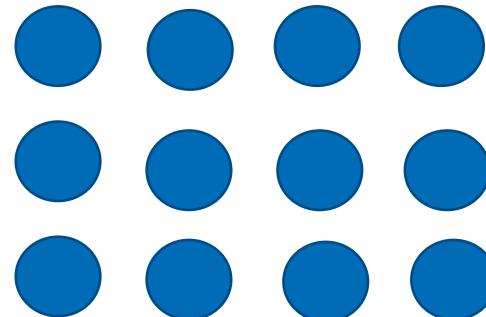


Stream-Consumer



Stream-Producer

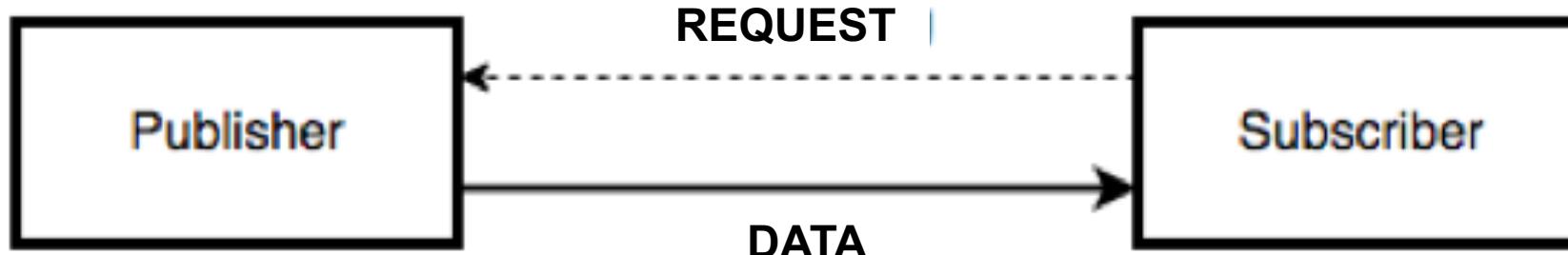
**Here you go! But ask me if you want
more with your new capabilities !**



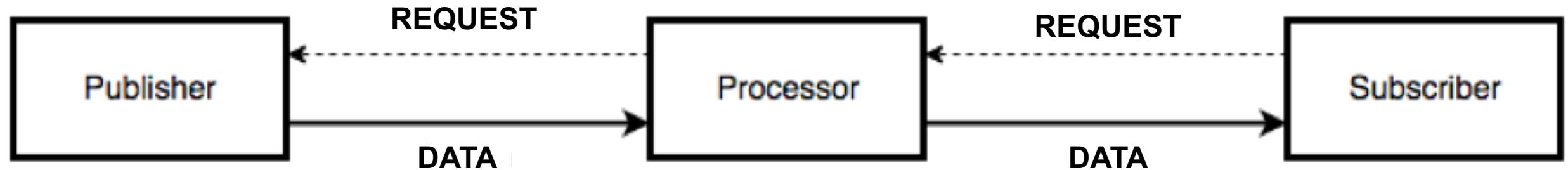
Stream-Consumer

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

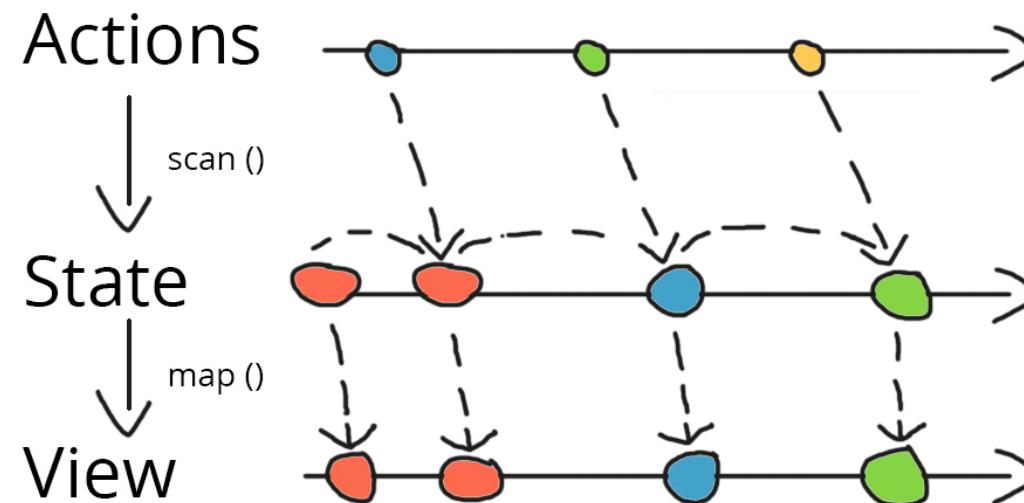
- Publishers are publishing data and subscribers are processing data**



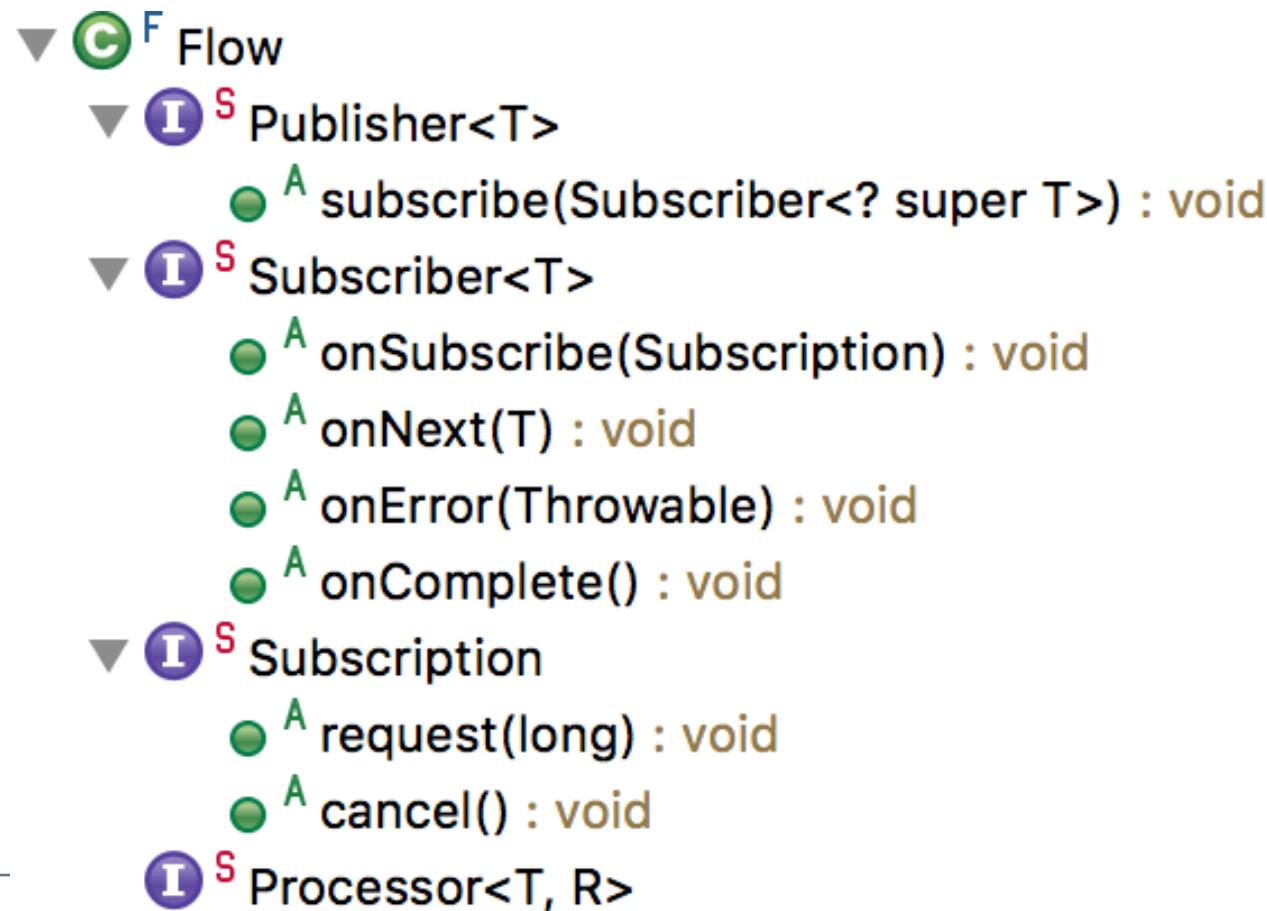
- Expansion into a processing chain, similar to a relocation a chain of helpers**

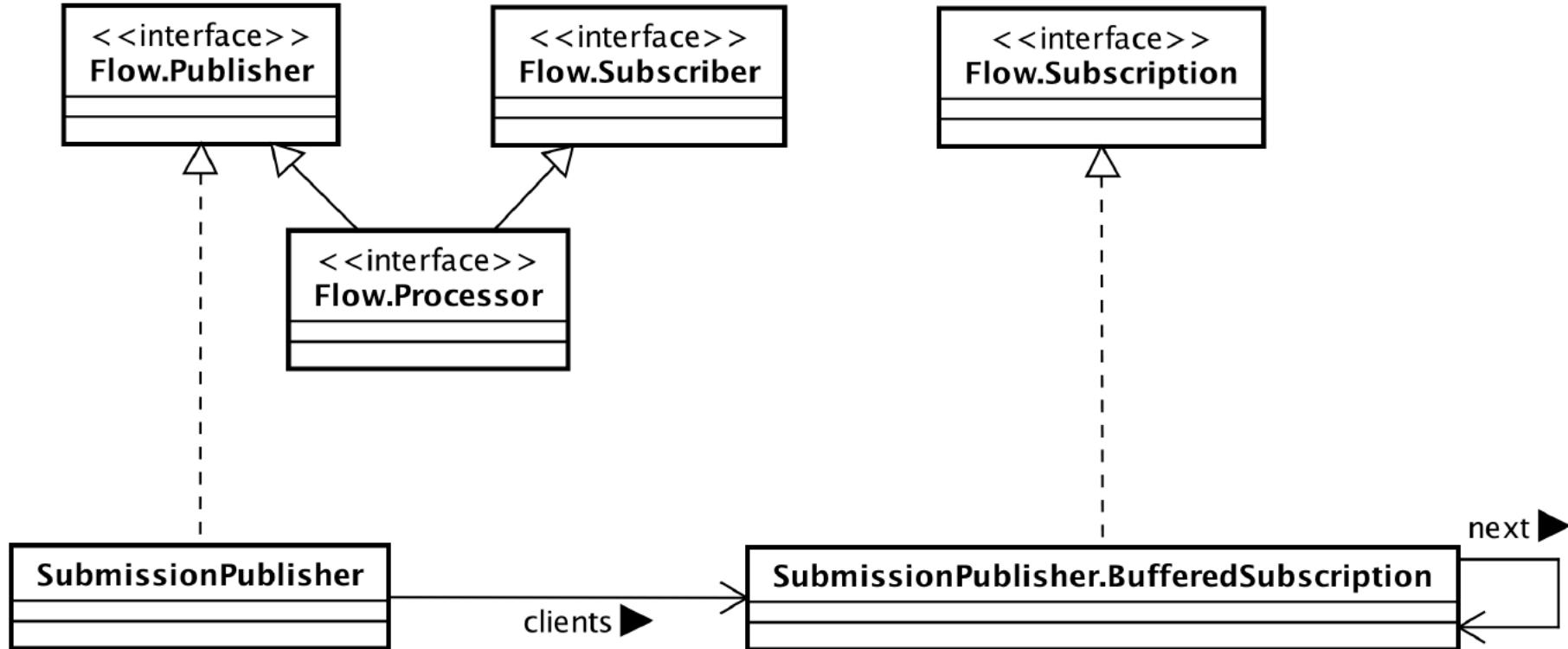


Reactive Streams in the JDK



- Publish Subscribe Framework based on the Flow class and a SubmissionPublisher utility class





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

Simple Subscriber

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

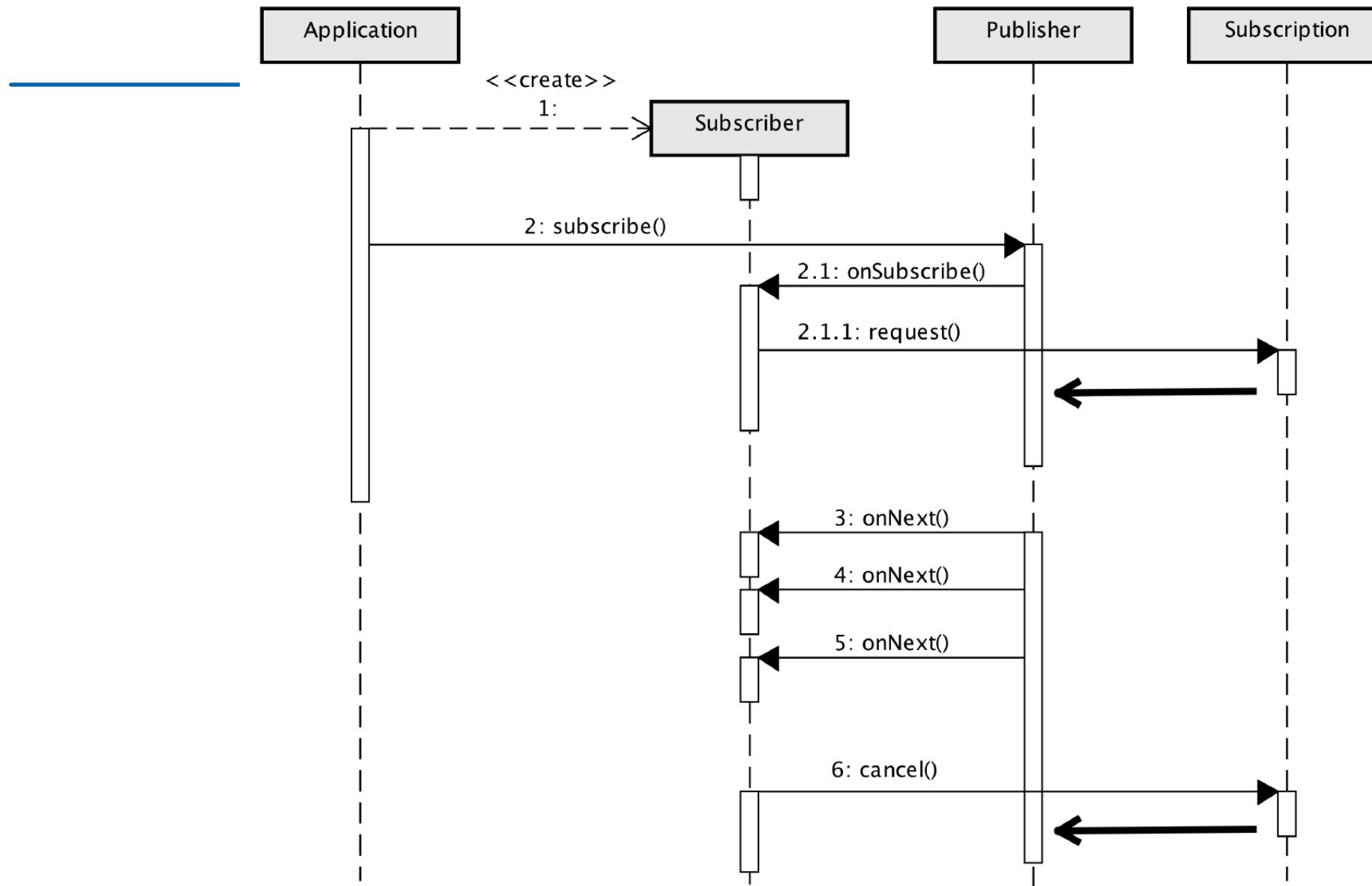
    @Override
    public void onSubscribe(Flow.Subscription subscription)
    {
        System.out.println("onSubscribe(): " + subscription);
        this.subscription = subscription;
        subscription.request(1);
    }

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

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

Submitting items...

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



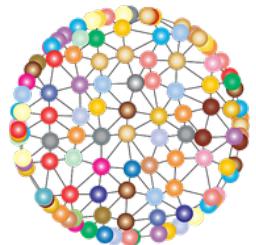
Subscription - Cancel

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



The screenshot shows a Java IDE's console window. The title bar includes tabs for 'Console' (selected), 'Progress', and others. The main area displays the output of the code execution, which consists of the text 'Received-->' followed by a sequence of integers from 186 to 200, each on a new line. This indicates that the code is printing every integer value it receives, starting from 186 up to 200.

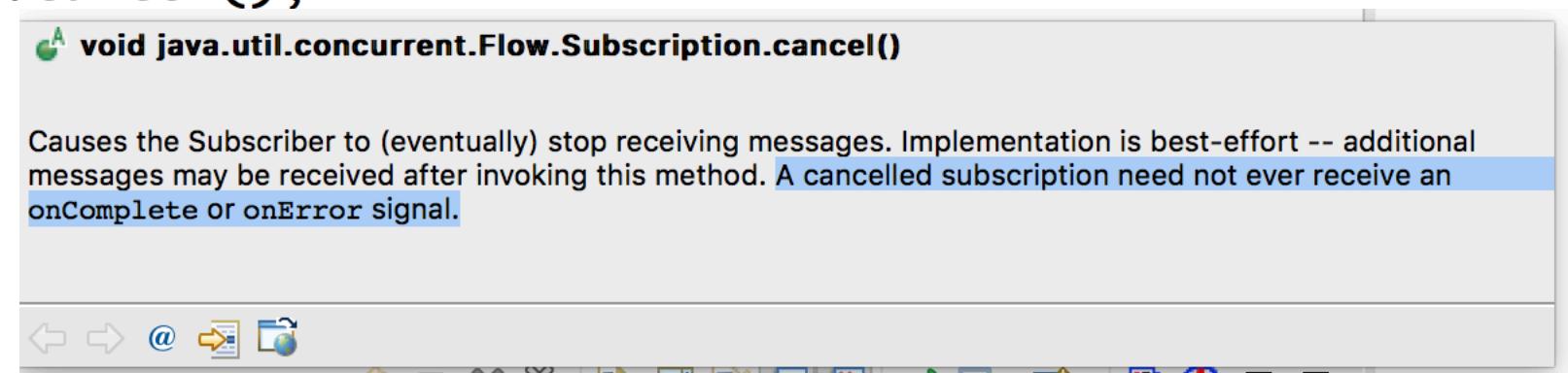
```
Received-->186  
Received-->187  
Received-->188  
Received-->189  
Received-->190  
Received-->191  
Received-->192  
Received-->193  
Received-->194  
Received-->195  
Received-->196  
Received-->197  
Received-->198  
Received-->199  
Received-->200
```

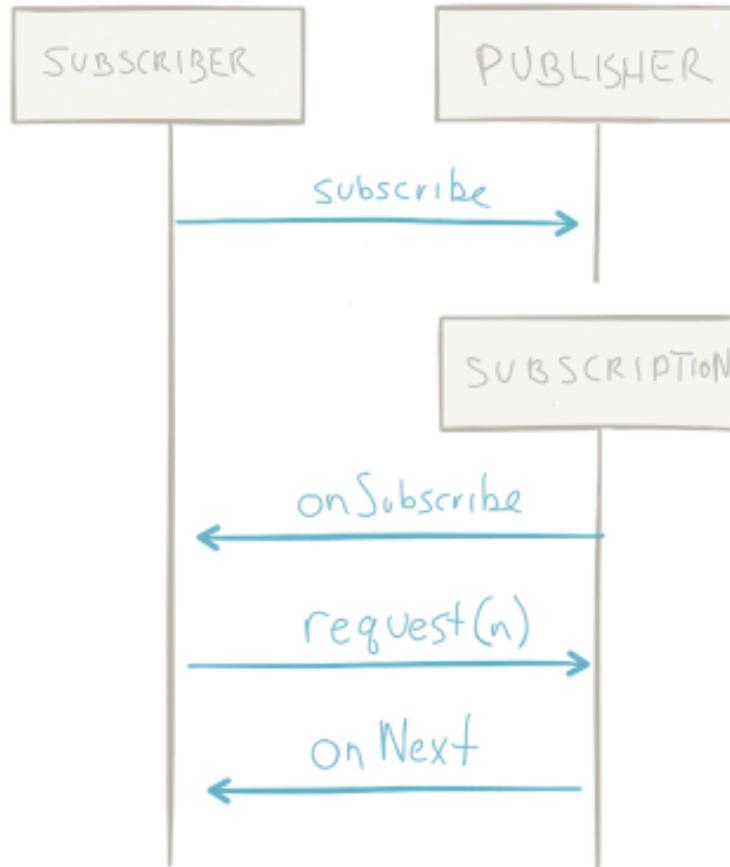


**Where's the
"Completed!"?**

Flow-API - <Interface> Subscription

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





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*.

DEMO

Exercises PART 2

<https://github.com/Michaeli71/JAX-London-Best-of-Java9-15.git>

PART 3: Other News and Changes in APIs

- Date API
 - java.util.Arrays
 - java.io.InputStream and Reader
 - Files
 - Misc
 - HTTP/2
 - Direct Compilation
 - JShell
-

Date API Enhancements



- **datesUntil()** – creates a Stream<LocalDate> between two LocalDate instances and allows you to optionally specify a step size:

```
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);
}
```

- Start February 7th => Skip 150 days into the future => July 7th
- **Day-Stream:** Per day iteration limited to 4
- **Month-Stream:** Monthly iteration limited to 3
=> Specification of an alternative step size, here 3 months:

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

- **divideBy()** – divides by given unit
- **truncateTo()** – truncates to given unit

```
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));
}
```

```
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 |

- **toXXX()** – converts into given unit
- **toXXXPart()** – extracts the part of the given unit

```
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());
}
```

```
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.Arrays



- **Enhancements in `java.util.Arrays`:**
 - `equals()` – Compares arrays to equality (related to ranges)
 - `compare()` – Compares arrays (also related to ranges)
 - `mismatch()` – Determines the first difference in array (also related to ranges)

- **Arrays.equals()** existed for a long time in JDK, but ...
 - Unfortunately, it was not possible to limit the comparison to specific ranges

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

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

- `Arrays.compare()` new in JDK
- Compares according to `Comparator<T>` – it is also possible to limit the comparison to specific ranges

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

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

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

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

- **Arrays.mismatch()** new in JDK
 - Detects differences / mismatches in two arrays – it is also possible to do this just for specific ranges

java.io.InputStream / Reader



- **There are some new helpful methods in class InputStream – some of them are long awaited:**
 - public long **transferTo**(OutputStream out) throws IOException
 - public byte[] **readAllBytes**() throws IOException
 - public int **readNBytes**(byte[] b, int off, int len) throws IOException

- **long transferTo(Writer)**

All characters are transferred from the reader to the given writer - this functionality exists in the `InputStream` class since Java 9. Why get reader improved only in Java 10?

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

sr.transferTo(sw);

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

=>

Sw: Hello

Extensions in the Class Files



- In Java 11, the processing of strings related to files has been made easier.
- Now it is easy to write strings into and to read them from a file.
- The utility class `Files` provides the methods `writeString()` und `readString()` for that.

```
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
```

- **Correction 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
```

- **Correction 2:** Read string only once

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

=>

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

Misc

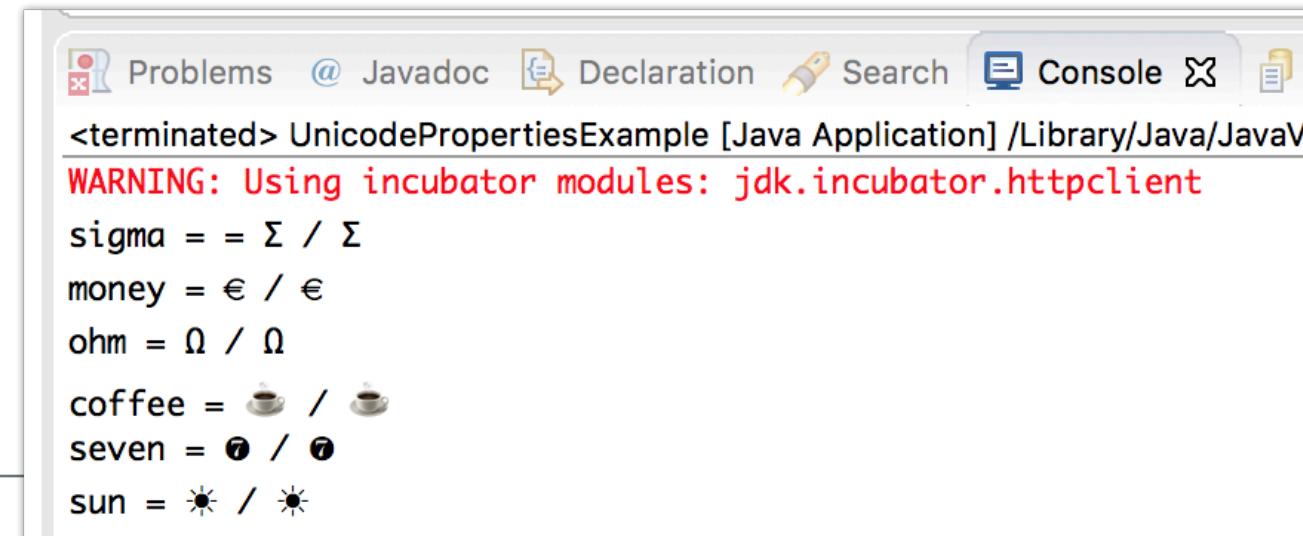


UTF-8 Resource Bundles:

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

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

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

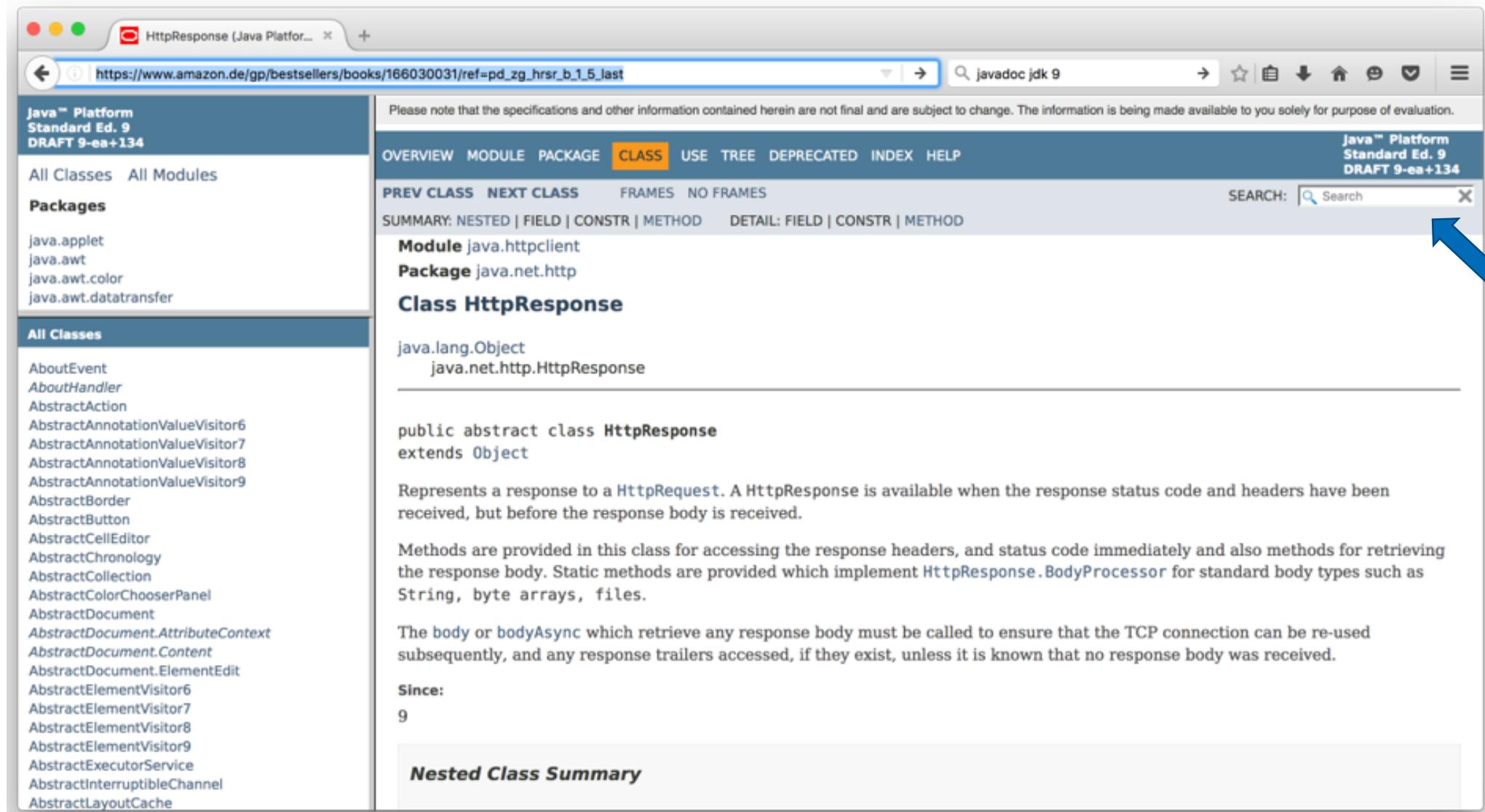


- The **Predicate<T>** class is very useful for expressing filter conditions for stream processing.
- Besides the combination with **and()** and **or()** a negation could be expressed with **negate()**. This was a bit cumbersome and difficult to read:

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

- With Java 11 this becomes more readable and no need for an additional (artificial) Explaining variable **isEmpty**

```
// JDK 11 style
final Predicate<String> notEmptyJdk11 = Predicate.not(String::isEmpty);
// with static import
final Predicate<String> notEmptyJdk11 = not(String::isEmpty);
```



The screenshot shows a Java documentation page for the `HttpServletResponse` class. The URL in the browser is https://www.amazon.de/gp/bestsellers/books/166030031/ref=pd_zg_hrsr_b_1_5_1est. The page is part of the "Java™ Platform Standard Ed. 9 DRAFT 9-ea+134" documentation.

The left sidebar contains links for "All Classes" and "All Modules". Under "All Classes", there is a list of various Java classes, with "All Classes" currently selected. Other visible class names include `AboutEvent`, `AboutHandler`, `AbstractAction`, `AbstractAnnotationValueVisitor6`, `AbstractAnnotationValueVisitor7`, `AbstractAnnotationValueVisitor8`, `AbstractAnnotationValueVisitor9`, `AbstractBorder`, `AbstractButton`, `AbstractCellEditor`, `AbstractChronology`, `AbstractCollection`, `AbstractColorChooserPanel`, `AbstractDocument`, `AbstractDocument.AttributeContext`, `AbstractDocument.Content`, `AbstractDocument.ElementEdit`, `AbstractElementVisitor6`, `AbstractElementVisitor7`, `AbstractElementVisitor8`, `AbstractElementVisitor9`, `AbstractExecutorService`, `AbstractInterruptibleChannel`, and `AbstractLayoutCache`.

The main content area displays the `HttpServletResponse` class documentation. The navigation bar at the top includes links for OVERVIEW, MODULE, PACKAGE, CLASS (which is highlighted in orange), USE, TREE, DEPRECATED, INDEX, and HELP. Below the navigation bar, there are links for PREV CLASS, NEXT CLASS, FRAMES, and NO FRAMES. The SUMMARY section lists NESTED, FIELD, CONSTR, and METHOD. The DETAIL section lists FIELD, CONSTR, and METHOD. The Module is listed as `java.httpclient` and the Package as `java.net.http`.

Class `HttpServletResponse`

`java.lang.Object
 java.net.http.HttpResponse`

`public abstract class HttpServletResponse
extends Object`

Represents a response to a `HttpRequest`. A `HttpResponse` is available when the response status code and headers have been received, but before the response body is received.

Methods are provided in this class for accessing the response headers, and status code immediately and also methods for retrieving the response body. Static methods are provided which implement `HttpResponse.BodyProcessor` for standard body types such as `String`, byte arrays, files.

The `body` or `bodyAsync` which retrieve any response body must be called to ensure that the TCP connection can be re-used subsequently, and any response trailers accessed, if they exist, unless it is known that no response body was received.

Since:
9

Nested Class Summary

HTTP/2 API



- **URL + openStream()**

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

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

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

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

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

- **Read content as String**

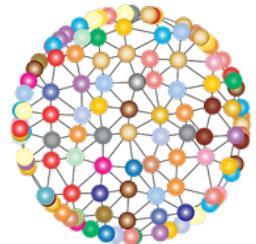
```
public static String readContent(final InputStream is) throws IOException
{
    try (final InputStreamReader isr = new InputStreamReader(is);
         final BufferedReader br = new BufferedReader(isr))
    {
        final StringBuilder content = new StringBuilder();

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

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



**What do you say about
this code? What does it
not do?**



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

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

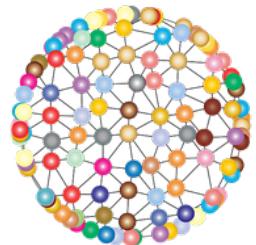
final HttpClient httpClient = HttpClient.newHttpClient();
final HttpResponse<String> response = httpClient.send(request, asString);

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

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



**The PO enters the scene:
He likes to have it
asynchronous!**



```
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");
}
```

```
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);
    }
}
```



**One moment please:
Isn't that old school?
What can we improve on
waiting?**

```
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);

// Wait and process just with CompletableFuture
asyncResponse.thenAccept(response -> printResponseInfo(response)).
    orTimeout(2, TimeUnit.SECONDS).
    exceptionally(ex ->
{
    asyncResponse.cancel(true);
    System.err.println("timeout");
    return null;
});

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

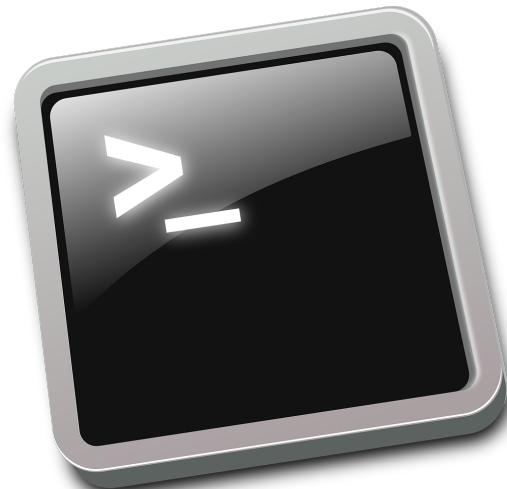
- **HTTPRequest**

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

- **HTTPClient**

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

Direct Compilation Launch Single-File Source- Code Programs



- Allows you to compile and run single file Java applications directly in one go.
- saves you work and you don't have to know anything about bytecode and .class files.
- especially useful for executing smaller Java files as scripts and for getting started with Java

```
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 – Two Public Classes 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);
    }
}
```

```
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());
    }
}
```

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

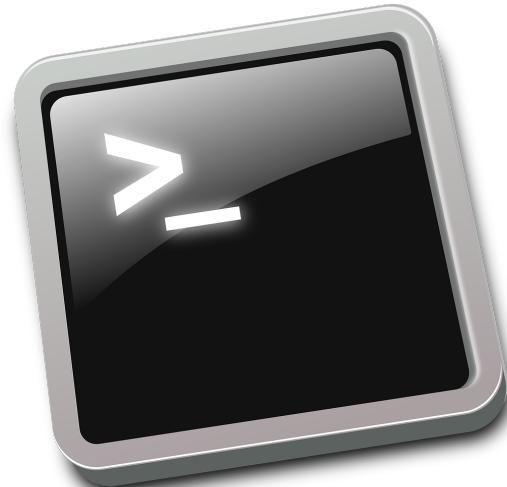
public class DirectoryLister
{
    public static void main(String[] args) throws Exception
    {
        var dirName = ".";
        if (args == null || args.length < 1)
        {
            System.err.println("Using current directory as fallback");
        }
        else
        {
            dirName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

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

- File must not end with ‘.java’
- File name independent of class name
- File must be executable (chmod +x)

DEMO

JShell



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

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

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

```
jshell> import java.time.*
```

```
jshell> boolean isSunday(LocalDate date)
| ...> {
| ...>     return date.
```

```
adjustInto(      atStartOfDay(    atTime(        compareTo(
get(              getChronology()  getClass()      getDayOfMonth()
getLong(          getMonth()       getMonthValue()   getYear()
isEqual(          isLeapYear()     isSupported()    lengthOfMonth()
minusMonths(      minusWeeks()    minusYears()    notify()
plusMonths(      plusWeeks()     plusYears()    query()
jshell> boolean isSunday(LocalDate date)(           with(
```

```
    ...> {
    ...>     return date.getDayOfWeek() == DayOfWeek.SUNDAY;
    ...> }
| created method isSunday(LocalDate)
```

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

```
datesUntil(      equals(        format(
getDayOfYear()  getDayOfWeek()  getEra()
hashCode()       isAfter(       isBefore(
lengthOfYear()  minus(        minusDays(
notifyAll()      plus(        plusDays(
range(          toEpochDay()  toEpochSecond(
withDayOfMonth( withDayOfYear( withMonth(
```

DEMO

- Perform calculations on the fly
- Define variables
- Define methods
- Practical forward referencing: a method can call methods that are not yet defined
- Practical for **SMALL** experiments
- Since Java 14: editor is much more comfortable, before that rather catastrophic regarding multi-line editing
- 3rd Party & Preview-Features
 - `jshell --class-path myOwnClassPath --enable-preview`

- Create own instances of JShell programmatically (`create()`)
- Execute snippets of code (`eval()`)
- Perform dynamic calculations
- Use it as a kind of replacement for JavaScript-Engine

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

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

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

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

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

    // unfortunately no (direct) access on value, use varValue() instead
    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"
```

```
try (JShell js = JShell.create())
{
    // ATTENTION: we need the ';' otherwise not processed correctly
    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

Exercises PART 3

<https://github.com/Michaeli71/JAX-London-Best-of-Java9-15.git>

PART 4:

What's new in Java 12

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

Enhancements in class String



- The **String class exists since JDK 1.0 and has only undergone a few API changes in the meantime.**
- With Java 11 this changed. 6 new methods were introduced.
- In Java 12 the following two methods are added:
 - **indent()** - Adjusts the indentation of a string
 - **transform()** - Allows actions to transform a string

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

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

- Es sind aber auch negative Werte erlaubt:

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

Enhancement in java.lang.String

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

```
'    Test
'
9
```

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

```
'6789
'
5
```

```
var str1 = "first_line\nsecond_line\nlast_line";
System.out.println(str1.length());
```

```
32
33
```

```
var str2 = str1.indent(0);
System.out.println(str2.length());
```

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

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

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

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

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

- Example: Transform all chars of a string

1. convert to UPPERCASE
2. Remove the letter 'T'
3. split it up

- Until now you can do it as follows:

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

- Example: Transform all chars of a string

1. convert to UPPERCASE
2. Remove the letter 'T'
3. split it up

- Until now you can do it as follows:

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

[HIS, IS, A, ES]

- Usage of `transform()` to process all chars of a string

1. convert to UPPERCASE
2. Remove the letter 'T'
3. split it up

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

```
System.out.println(Arrays.asList(result)); [HIS, IS, A, ES]
```

- Analogy `map()` in Streams
- Rather theoretically practical 😊

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

Addition CompactNumberFormat



- **CompactNumberFormat** is a subclass of **NumberFormat**
 - formats a decimal number in a compact form, Locale sensitive
 - NumberFormat is the abstract base class for all number formats which provides the interface for formatting and parsing numbers.
 - An example of a SHORT compact form would be writing 10,000 as 10K,
 - There is a constructor, but easier to use factory method:

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

CompactNumberformat Style



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

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

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

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

CompactNumberformat Style



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

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

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

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

NumberFormat SHORT
Result: 10K
Result: 123K
Result: 1M
Result: 2B

NumberFormat LONG
Result: 10 thousand
Result: 123 thousand
Result: 1 million
Result: 2 billion

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

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

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

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

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

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

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

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

US/SHORT parsing:

1
1000
1000000
1000000000

US/LONG parsing:

1000
1000000
1000000000

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

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

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

```
compactNumberFormat.setMinimumFractionDigits(1);

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

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

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

Utility-Class CompactNumberformat – Customization

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

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

```
final DecimalFormat currencyFormat = (DecimalFormat)
    NumberFormat.getCurrencyInstance();
final NumberFormat cnf = CompactNumberFormat.getCompactNumberInstance(Locale.GERMAN,
    NumberFormat.Style.SHORT);
cnf.setMinimumFractionDigits(2);
```

```
System.out.println(currencyFormat.format(1234.455));
System.out.println(currencyFormat.format(1234.445));
System.out.println(currencyFormat.format(12345));
System.out.println(currencyFormat.format(1234567));
```

```
CHF 1'234.45
CHF 1'234.44
CHF 12'345.00
CHF 1'234'567.00
```

```
System.out.println(cnf.format(1234.455));
System.out.println(cnf.format(1234.445));
System.out.println(cnf.format(12345));
System.out.println(cnf.format(1234567));
```

```
1.234
1.234
12.345
1,23 Mio.
```

Enhancements in class Files



- Methods for comparing arrays were introduced in Java 9.
- In Java 11, the processing of strings related to files has been made easier.
- In Java 12 and in the following example, these are combined in method **mismatch()**

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

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

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

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

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

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

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

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

File1 mismatch File2 = -1
File1 mismatch File3 = 5

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

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

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

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

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

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

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

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

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

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

enc1 mismatch enc2 = 1
oneFile mismatch oneFile = -1

- Return values consistent to expectation with same encoding
- Therefore ...

| File 1 | File 2 | Encoding | Result |
|--------|--------|-----------|---|
| ABCD | ABCD | same | -1 |
| ABCDEF | ABCDXY | same | 4 |
| Zürich | Zürich | different | Positive value, first character with umlaut or encoding deviation |

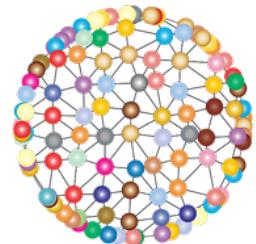
- If the paths point to the same file, it is of course also the same

Teeing() Collector



The powerful Stream API provides a set of predefined collectors:

- `toCollection()`, `toList()`, `toSet()` ... – Group the items of the stream into the appropriate collection.
- Java 10 also added `toUnmodifiableXyz()`!

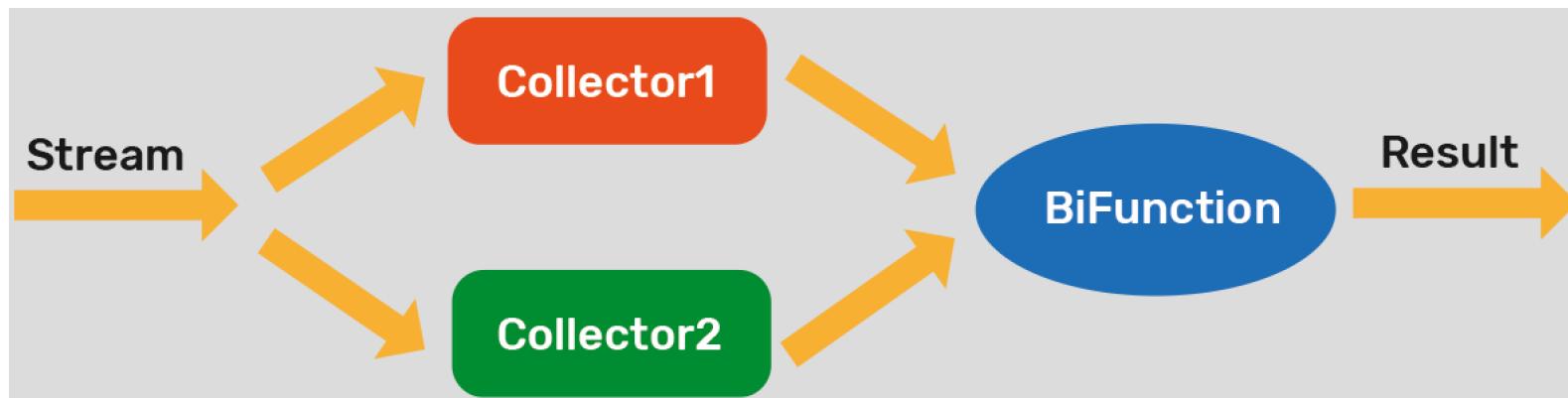


What is missing?

What about combining streams?

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

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



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

    var primeNumbers = Stream.of(1, 3, 5, 7, 11, 13, 17);
    System.out.println(calcCountAndSum(primeNumbers));
}

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

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

    var primeNumbers = Stream.of(1, 3, 5, 7, 11, 13, 17);
    System.out.println(calcCountAndSum(primeNumbers));
}

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

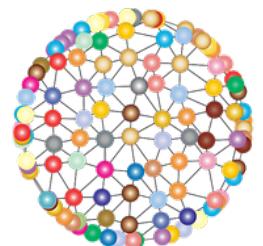


```
Pair<T> [first=6, second=21]
Pair<T> [first=7, second=57]
```

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

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

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

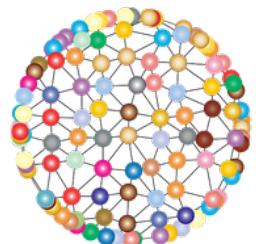


**How about Map.Entry as
a replacement for your
own pair?**



Not so good!!

**Why? Map.Entry is intended for maps
and should only be used in their
context.**



Task: From a stream of strings, different elements should be filtered out and then the results should be combined.

- This can be achieved combining collectors `filtering()` and `teeing()`:

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

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

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

Task: From a stream of strings, different elements should be filtered out and then the results should be combined.

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

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

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

Task: From a stream of strings, different elements should be filtered out and then the results should be combined.

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

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

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

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

Task: From a stream of strings, different elements should be filtered out and then the results should be combined.

```
var names = Stream.of("Michael", "Tim", "Tom", "Mike", "Bernd");  
  
final Predicate<String> startsWithMi = text -> text.startsWith("Mi");  
final Predicate<String> endsWithM = text -> text.endsWith("m");  
  
var result = names.collect(teeing(filtering(startsWithMi, toList()),  
                           [Michael, Mike]  
                           filtering(endsWithM, toList()),  
                           [Tim, Tom]  
                           (list1, list2) -> List.of(list1, list2));  
  
System.out.println(result);
```

JMH



- Sometimes some parts of the software are not as performant as needed.
- There are various tools and levels for optimizing performance
- In general, one should first measure thoroughly and optimize only with caution.
- Why?
 - There are already various optimizations built into the JVM
 - Optimization is not trivial, since the measurements should be performed under the same conditions (CPU load, memory consumption, etc.), for comparable results
 - purely on the basis of assumptions one is often wrong
- by no means optimize only based on assumptions, make sure it is based on measurements:
 - simple start/stop measurements
 - More sophisticated processes with several runs are more recommendable.

- **JEP 230** add a basic suite of microbenchmarks to the JDK source code, and make it easy for developers to run existing microbenchmarks and create new ones.
- Based on [Java Microbenchmark Harness \(JMH\)](#)
- Framework for creating microbenchmark tests
- Microbenchmarking = optimization on the level of single or fewer statements
- Considers various external interferences and fluctuations
- Comprehensive, but mostly easy to configure
- Makes writing benchmarks almost as easy as unit testing with JUnit

- Simple start/stop measurements with `System.currentTimeMillis()`

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

someCodeToMeasure();

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

- surround the program part to be measured with `System.currentTimeMillis()`
- Use as a kind of stopwatch by determining and the difference between the values

- Repeated Start-/Stop measurements

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

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

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

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

- Less susceptible to system load fluctuations or other interference due to multiple runs and averaging.
- It is also quite easy to determine minimum and maximum duration or standard deviation.

- **settling effects:** Only after a certain number of runs does functionality show its optimum runtime:

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

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

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

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

- JMH can create a performance test environment with the following Maven command:

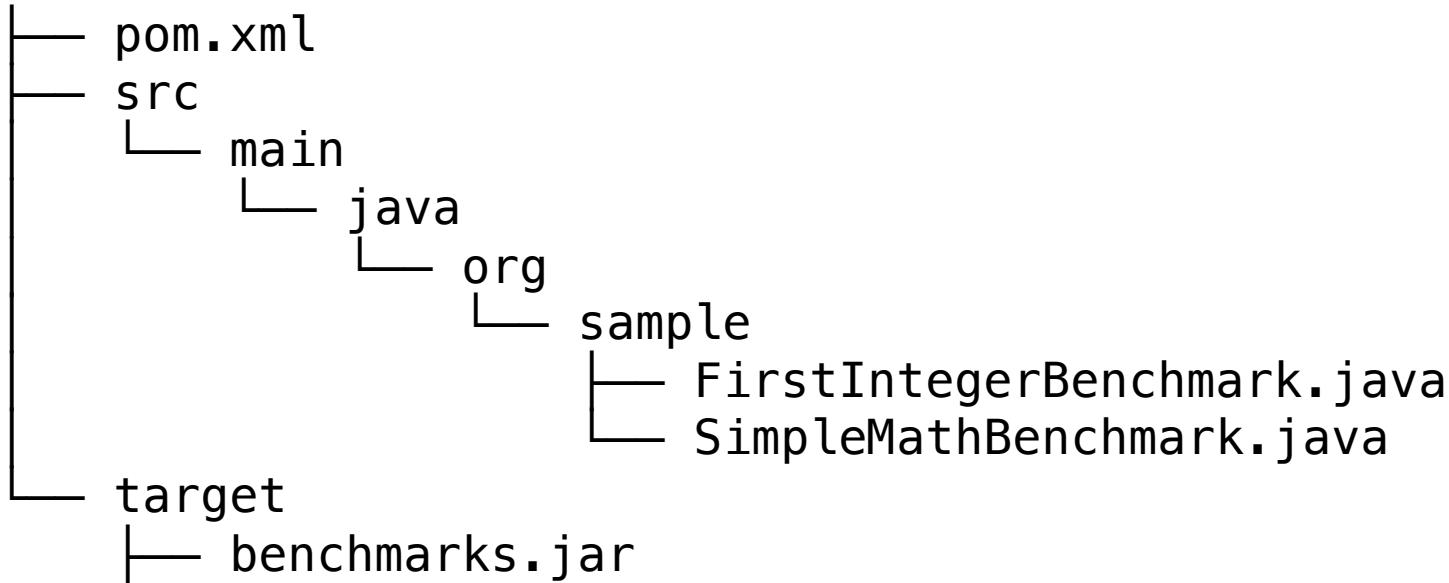
```
mvn archetype:generate \  
  > -DinteractiveMode=false \  
  > -DarchetypeGroupId=org.openjdk.jmh \  
  > -DarchetypeArtifactId=jmh-java-benchmark-archetype \  
  > -DgroupId=org.sample \  
  > -DartifactId=jmh-test \  
  > -Dversion=1.0-SNAPSHOT
```

- A **MyBenchmark** class is generated as the skeleton:

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

- JMH works with annotations and integrates different measurements based on them.

- You can create your own benchmark classes based on the skeleton:



- In 2 steps to a benchmark
 - 1) mvn clean package
 - 2) java -jar target/benchmarks.jar

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

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

Own Microbenchmark with JMH



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

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

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

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

Own Microbenchmark with JMH



```
@State(Scope.Benchmark)
public static class MyBenchmarkState {
    @Param({ "100000", "1000000" })
    public int value;
}

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

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

Own Microbenchmark with JMH



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

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

- POM adjustment for java 14:

<!-- Java source/target to use for compilation. -->

<javac.target>1.8</javac.target>

=>

<javac.target>14</javac.target>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.8.0</version>

=>

<version>3.8.1</version>

What do we want to measure?

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

Example Results

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

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

| Benchmark | value) | Mode | Cnt | Score | Error | Units |
|--|---------------|-------------|------------|--------------|---------------|--------------|
| StringJoinBenchmark.concatUsingStringBuilder | 100000 | avgt | 25 | 1.003 | ± 0.145 | ms/op |
| StringJoinBenchmark.stringConcatABC | 100000 | avgt | 25 | 2010.486 | ± 101.550 | ms/op |
| StringJoinBenchmark.stringPlusABC | 100000 | avgt | 25 | 1882.743 | ± 20.801 | ms/op |

PART 5: News and Changes in Java 13 and 14

- Build-Tools and IDEs
- Syntax enhancement switch (FINAL)
- Helpful NullPointerExceptions (FINAL)
- Syntax enhancement Text Blocks (PREVIEW II)
- Syntax enhancement Records (PREVIEW)
- Syntax enhancement instanceof (PREVIEW)

Build-Tools and IDEs



- Current IDEs & Tools are fundamentally good
- Eclipse: Version 2020-09
- IntelliJ: Version 2020.2
- Maven: 3.6.3 (latest 3.6.3_1), Compiler-Plugin
- Gradle: 6.6
- Activation of preview features required
 - In dialogs
 - In the build script



Maven™

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

- Eclipse 2020-09: Java 14 built in / Java 15 with plugin
- Activation of preview features required



Eclipse Marketplace

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

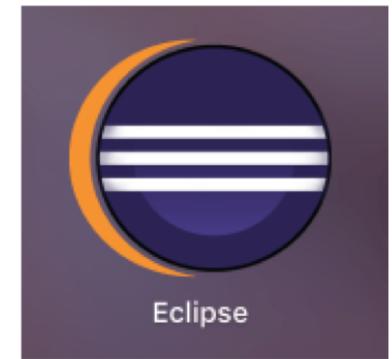
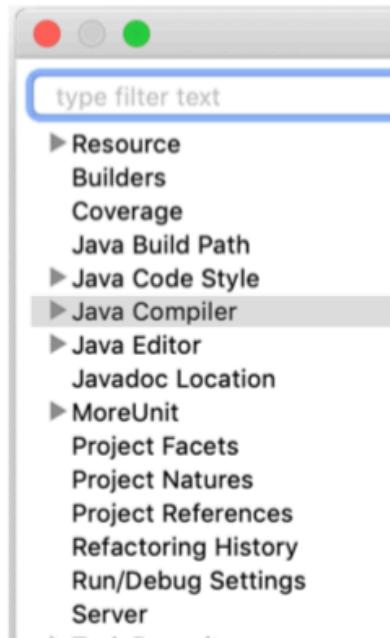
Search Recent Popular Favorites Installed Giving IoT an Edge

Find: Java 14 All Markets All Categories Go

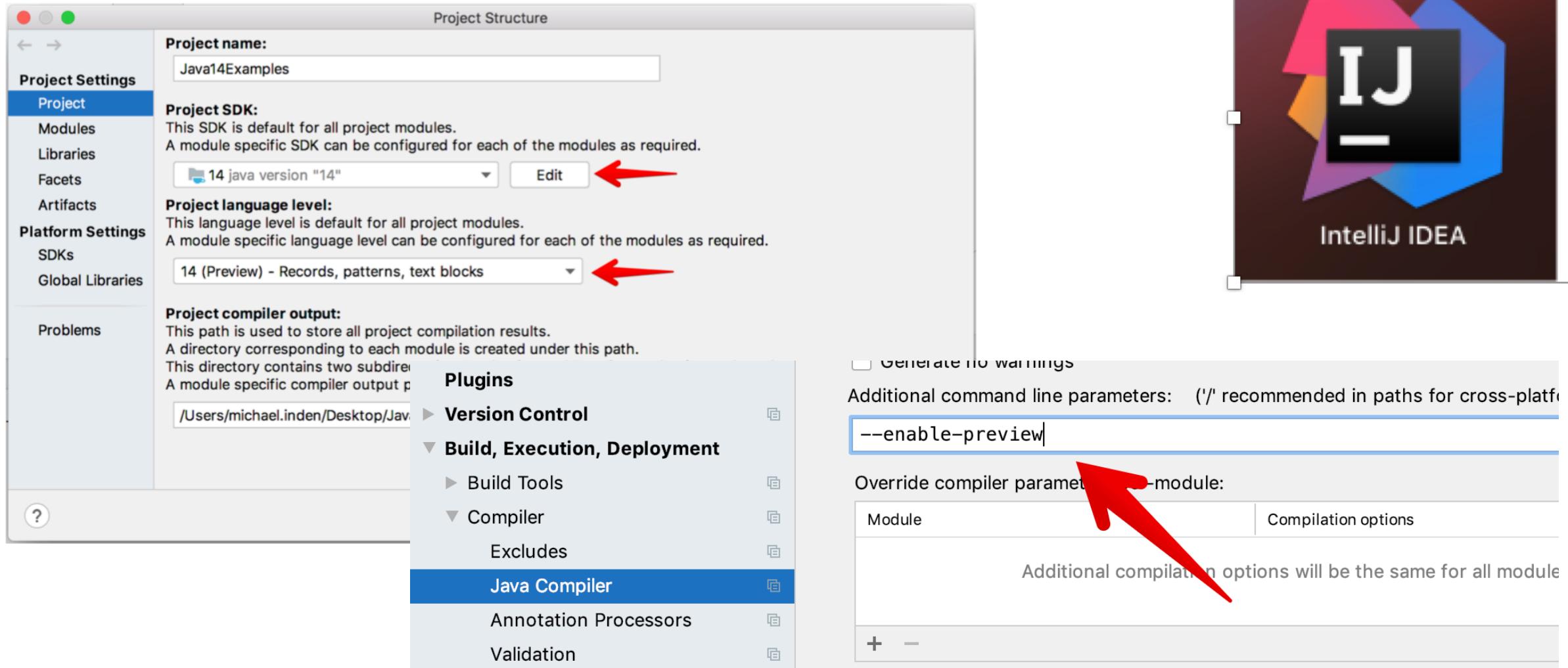
Java 14 Support for Eclipse 2020-03 (4.15)

This marketplace solution provides Java 14 support for Ec
ensure you have the following version or a more... [more in](#)
by [The Eclipse Foundation](#), EPL 2.0
[Eclipse Java JDT Java 14](#)

0 Installs: 4 (4 last month)



- Activation of preview features required



The screenshot shows the IntelliJ IDEA Project Structure dialog. The left sidebar has 'Project Settings' selected. Under 'Project', the 'Project SDK' dropdown is set to '14 java version "14"' with an 'Edit' button highlighted by a red arrow. The 'Project language level' dropdown is set to '14 (Preview) - Records, patterns, text blocks' with a red arrow pointing to it. In the main pane, under 'Build, Execution, Deployment', 'Java Compiler' is selected. At the bottom, the 'Additional command line parameters' field contains '--enable-preview', which is also highlighted by a red arrow. A large red arrow points from the 'Override compiler parameters --module:' section towards the 'Additional command line parameters' field.

Project name: Java14Examples

Project SDK:
This SDK is default for all project modules.
A module specific SDK can be configured for each of the modules as required.
14 java version "14" Edit

Project language level:
This language level is default for all project modules.
A module specific language level can be configured for each of the modules as required.
14 (Preview) - Records, patterns, text blocks

Project compiler output:
This path is used to store all project compilation results.
A directory corresponding to each module is created under this path.
This directory contains two subdirec...
A module specific compiler output p
/Users/michael.inden/Desktop/Jav

Plugins

Version Control

Build, Execution, Deployment

Build Tools

Compiler

Excludes

Java Compiler

Annotation Processors

Validation

Generate NO warnings

Additional command line parameters: (/ recommended in paths for cross-platf
--enable-preview

Override compiler parameters --module:

Module Compilation options

Additional compilation options will be the same for all module

- Activation of preview features required

```
sourceCompatibility=14  
targetCompatibility=14
```

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



- Activation of preview features required

```
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.8.1</version>
    <configuration>
      <source>14</source>
      <target>14</target>
      <!-- Needed for Java 12 Syntax-Neuerungen -->
      <compilerArgs>--enable-preview</compilerArgs>
    </configuration>
  </plugin>
</plugins>
```



Syntax Enhancements



- switch-case-expression still at the ancient roots from the early days of Java
- Compromises in language design that should make the transition easier for C++ developers.
- Relicts like the break and in the absence of such the Fall-Through
- Error prone, mistakes were made again and again
- In addition, the case was quite limited in the specification of the values.
- This all changes fortunately with Java 12 / 13. The syntax is slightly changed and now allows the specification of one expression and several values in the case

- Mapping of weekdays to their length...

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

- Mapping of weekdays to their length... elegantly with Java 13 / 14:

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

- Mapping of weekdays to their length... elegantly with Java 13 / 14:

Return-
Value

```
Day0fWeek day = Day0fWeek.FRIDAY;  
int numLetters = switch (day)  
{  
    case MONDAY, FRIDAY, SUNDAY -> 6;  
    case TUESDAY -> 7;  
    case THURSDAY, SATURDAY -> 8;  
    case WEDNESDAY -> 9;  
};
```

- Mapping of weekdays to their length... elegantly with Java 13 / 14:

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

- More elegance using case:
 - Besides the obvious arrow instead of the colon also several values allowed
 - No break necessary, no case-through either
 - switch can now return a value, avoids artificial auxiliary variables

- Mapping of month to their names...

```
// ATTENTION: Sometimes very bad error: default in the middle of cases
String monthString = "";
switch (month)
{
    case JANUARY:
        monthString = "January";
        break;
    default:
        monthString = "N/A"; // here HIDDEN Fall Through
    case FEBRUARY:
        monthString = "February";
        break;
    case MARCH:
        monthString = "March";
        break;
    case JULY:
        monthString = "July";
        break;
}
System.out.println("OLD: " + month + " = " + monthString); // February
```

- Mapping months to their names... elegantly with Java 13 / 14:

```
static String monthToName(final Month month)
{
    return switch (month)
    {
        case JANUARY -> "January";
        default -> "N/A"; // here is NO Fall Through
        case FEBRUARY -> "February";
        case MARCH -> "March";
        case JULY -> "JULY";
    };
}
```

- here is an enumeration of colors:

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

- Let's map them to the number of letters:

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

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

With Java 12 / 13 / 14 will again all be clear and easy:

```
public static void switchYieldReturnsValue(Color color)
{
    int numOfChars = switch (color)
    {
        case RED: yield 3;
        case GREEN: yield 5;
        case YELLOW, ORANGE: yield 6;
        default: yield -1;
    };

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

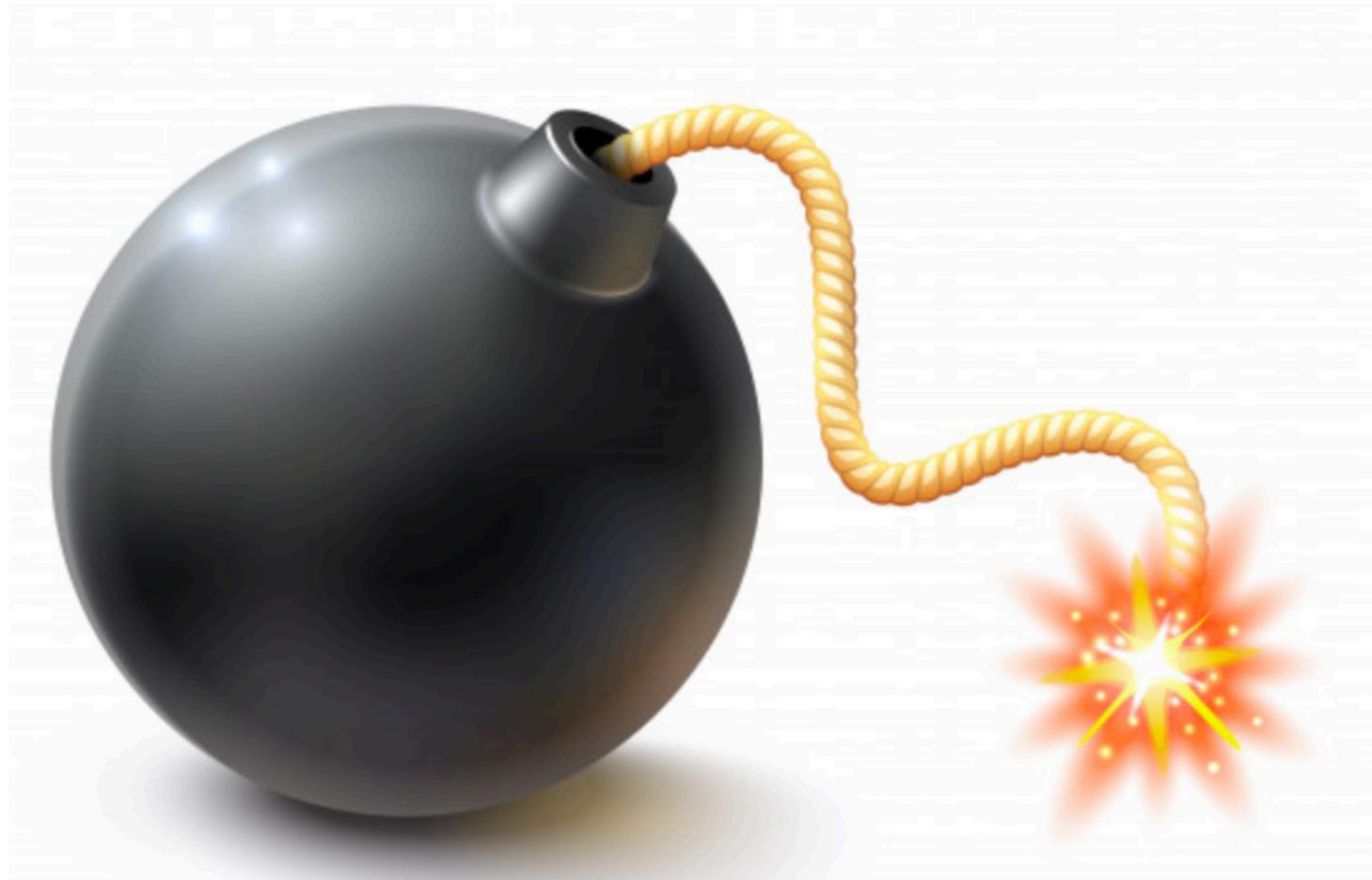
Switch Expressions: **yield** with return value



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

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

SUNDAY is FUN DAY
6



N P E

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

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

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

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

-XX:+ShowCodeDetailsInExceptionMessages

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

```
public static void main(final String[] args)
{
    try
    {
        final String[] stringArray = { null, null, null };
        final int errorPos = stringArray[2].lastIndexOf("ERROR");
    }
    catch (final NullPointerException e) { e.printStackTrace(); }
    try
    {
        final Integer value = null;
        final int sum = value + 3;
    }
    catch (final NullPointerException e) { e.printStackTrace(); }
}
```

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

`java.lang.NullPointerException: Cannot invoke
"java.lang.Integer.intValue()" because "value" is null
at java14.NPE_Second_Example.main(NPE_Second_Example.java:20)`

Text Blocks



- Long-awaited extension, namely to be able to define multiline strings without laborious links and to dispense with error-prone escaping.
- Facilitates, among other things, the handling of SQL commands, or the definition of JavaScript in Java source code.
- OLD

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

- NEW

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

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

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

Traditional String Literals

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

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

- NEW

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

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

- NEW

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

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

This is a string splitted in several smaller strings.

Records





**Wouldn't it be cool to
define DTOs etc. in a
simple way?**

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

- simplified form of classes for simple data containers
- Very short, compact notation
- API is derived implicitly from the attributes defined as constructor parameters

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

- Implementations of accessor methods as well as equals() and hashCode() automatically generated and adhering to contract

Enhancement Record

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

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

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

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

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

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

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

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

    // Zugriffsmethoden auf x und y
}
```

Records additional constructors and methods



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

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

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

MyPoint[x=17, y=19]
[17, 19]

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

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

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

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

Records for Complex Return Types or Parameters



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

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

```
IntStringReturnValue calculateTheAnswer()
{
    // Some complex stuff here
    return new IntStringReturnValue(42, "the answer");
}

IntListReturnValue calculate(CompoundKey inputKey)
{
    // Some complex stuff here
    return new IntListReturnValue(201,
        List.of("This", "is", "a", "complex", "result"));
}
```

- What's wrong with this self made pair?

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

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

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



**What is actually missing
there? What is perhaps
disturbing there?**

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

- **Extremely little writing effort**
- **Very practical for Pair, Tuples etc.**
- **Records work great with primitive types**
- **Implementations of accessor methods as well as equals() and hashCode() automatically and adhering to contracts**

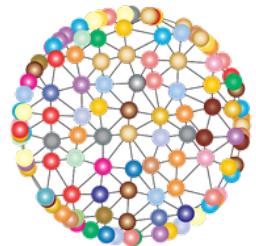


**That's cool ... BUT: How
can I integrate validity
checks?**

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

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

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



**Last question for records:
Is this all combinable?**

Example: All in

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

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

Pattern Matching instanceof



- **OLD STYLE**

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



**Always these casts...
Couldn't it be easier?**

- **OLD STYLE**

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

- **NEW STYLE**

```
if (obj instanceof Person person)
{
    // here is is possible to access variable person directly
}
```

```
Object obj2 = "Hello Java 14";  
  
if (obj2 instanceof String str)  
{  
    // here it is possible to use str without Cast  
    System.out.println("Length: " + str.length());  
}  
else  
{  
    // here we have no access to str  
    System.out.println(obj.getClass());  
}
```

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

Outlook: News and Changes in Java 15

- Syntax enhancement Text Blocks (FINAL)
- Local Enums and Interfaces (FINAL?)
- Syntax enhancement Records (PREVIEW II)
- Syntax enhancement instanceof (PREVIEW II)
- Sealed Types (PREVIEW)

- Current IDEs & Tools normally with good support
- Eclipse: Version 2020-09, needs a Plugin
- IntelliJ: Version 2020.2.1
- Maven: 3.6.3 (latest 3.6.3_1), Compiler-Plugin: 3.8.1
- Gradle: 6.6.1
- Activation of preview features needed
 - In dialogs
 - In build scripts

The logo for Maven, consisting of the word "Maven" in a bold, black, sans-serif font, with a stylized orange and red feather icon integrated into the letter "a".The logo for Gradle, featuring a stylized blue elephant icon to the left of the word "Gradle" in a bold, blue, sans-serif font.

```
public class LocalEnumsAndInterfacesExamples
{
    public static void main(String[] args)
    {
        // Works starting with Java 15, Compile-Error in Java 14 and before
        // Local enums are not supported at language level '14'
        enum LocalEnumState
        {
            BAD, GOOD, UNKNOWN
        }

        // Works starting with Java 15, Compile-Error in Java 14 and before
        // Local interfaces are not supported at language level '14'
        interface Evaluationable
        {
            LocalEnumState evaluate(String info);
        }
        ...
    }
}
```

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

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

- Control of inheritance and specify which classes can extend a base class

```
public class SealedTypesExamples
```

```
{  
    // A sealed class must have subclasses  
    sealed interface MathOp  
        permits BaseOp, Add, Sub // <= permitted sub types  
    {  
        int calc(int x, int y);  
    }  
}
```

sealed allows to seal an inheritance hierarchy and to accept only explicitly noted types. These have to be sealed, non-sealed or final.

```
// non-sealed allows to provide base class(es)
```

```
non-sealed class BaseOp implements MathOp // <= base class not sealed  
{  
    @Override  
    public int calc(int x, int y)  
    {  
        return 0;  
    }  
}  
...
```

```
..  
// direct implementation must be final  
final class Add implements MathOp  
{  
    @Override  
    public int calc(int x, int y)  
    {  
        return x + y;  
    }  
}  
final class Sub implements MathOp  
{  
    ...  
}  
// derived from base class must be final  
final class Mult extends BaseOp  
{  
}  
final class Div extends BaseOp  
{  
}
```

- A sealed class must have subclasses which in turn are listed after permits
- A non-sealed class can act as base class and subtypes can be derived
- A **final** class is – as usual – an endpoint in a class hierarchy

- **define which other classes or interfaces may form subtypes**
- **Sealed Types can expose behavior via interfaces when developing libraries, but keep control over possible implementations**
- **Sealed Types restrict the extensibility of class hierarchies and should be used with care. Flexibility not foreseen during implementation can be disturbing later. (e.g. Unit Testing and Extract and Override)**

Exercises PART 4 & 5

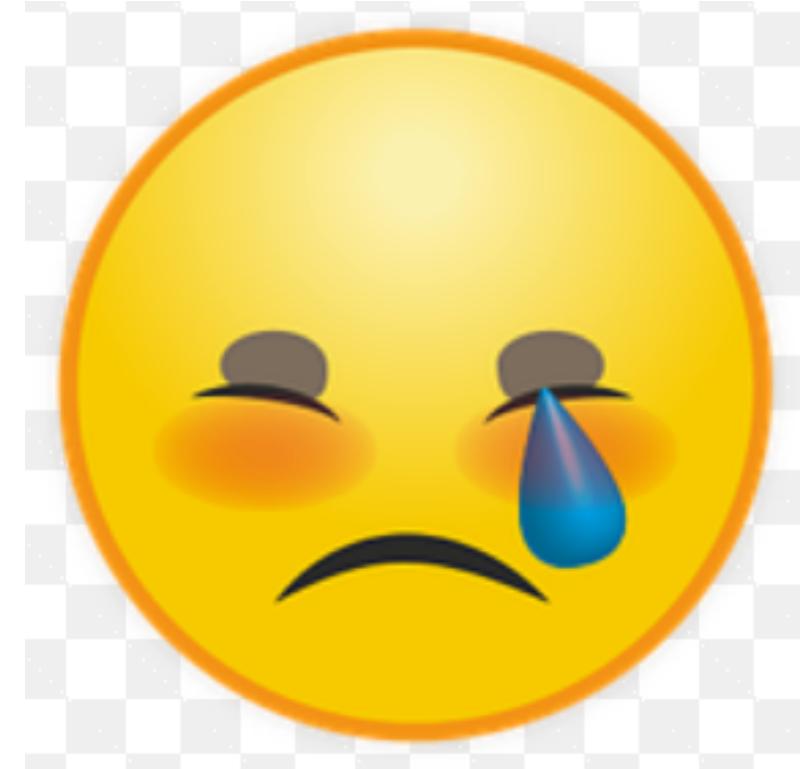
<https://github.com/Michaeli71/JAX-London-Best-of-Java9-15.git>

Conclusion

- Some parts of project coin (syntax)
- Switch / Records
- A lot of practical extensions in the APIs
- HTTP 2
- Reactive Streams
- Modularization with Project JIGSAW -- but until now a lot of room for improvement concerning tools



- **Multiple delay of Java 9**
Sept. 2016 => March 2017 => July 2017 => Sept. 2017
- **Next releases were on time, but sometimes bringing just a few new features (except Java 14)**
- **Often less than planned**
 - **no versioning with JIGSAW**
 - **no JSON support**
 - **instead of collection literals only convenience methods**
 - **Instead of ZIP only TEE (ing) collector**





Questions?

Thank You