ASMIQ

Best of Java 9 – 13

Michael Inden

CTO@ASMIQ AG

Speaker Intro

- 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
- Since June 2017 @ **Direct Mail Informatics / ASMIQ** in Zürich

- Author @ dpunkt.verlag

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

Agenda

Workshop Contents

ASMIQ

- **PART 1:** Syntax Enhancements in Java 9 to 12
- **PART 2:** News and API-Changes in the Java 9 to 12
- **PART 3:** Multi-Threading with CompletableFuture
- **PART 4:** HTTP/2
- **PART 5:** What's new in Java 13 (tomorrow: 2019/09/17)

PART 1:
Syntax Enhancements in Java 9 – 12

Anonymous inner classes and the diamond operator

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

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

@Deprecated-Annotation

ASMIQ

- @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()
{
   // ...
}
```

Local Variable Type Inference => var (JDK 10)

ASMIQ

- **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.**

Local Variable Type Inference => var

- **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);
```

Local Variable Type Inference => var

ASMIQ

- **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?

ASMIQ

- 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 should be inserted:

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

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

Local Variable Type Inference Beschränkungen

ASMIQ

- Short recapitulation: var is intended for **local variables** that are initialized directly.

- **`Is var to declare attributes, parameters or return types desirable?`**
  => this is not possible because the type cannot be determined unambiguously by the compiler.

- 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
```

- **More things that cause compilation errrors:**

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

PART 2: News and API-Changes in Java 9 – 12

- Stream-API
- Optional<T>
- Collection Factory Methods
- API Enhancements in String
- API Enhancements in Files

Stream API

takeWhile(…)

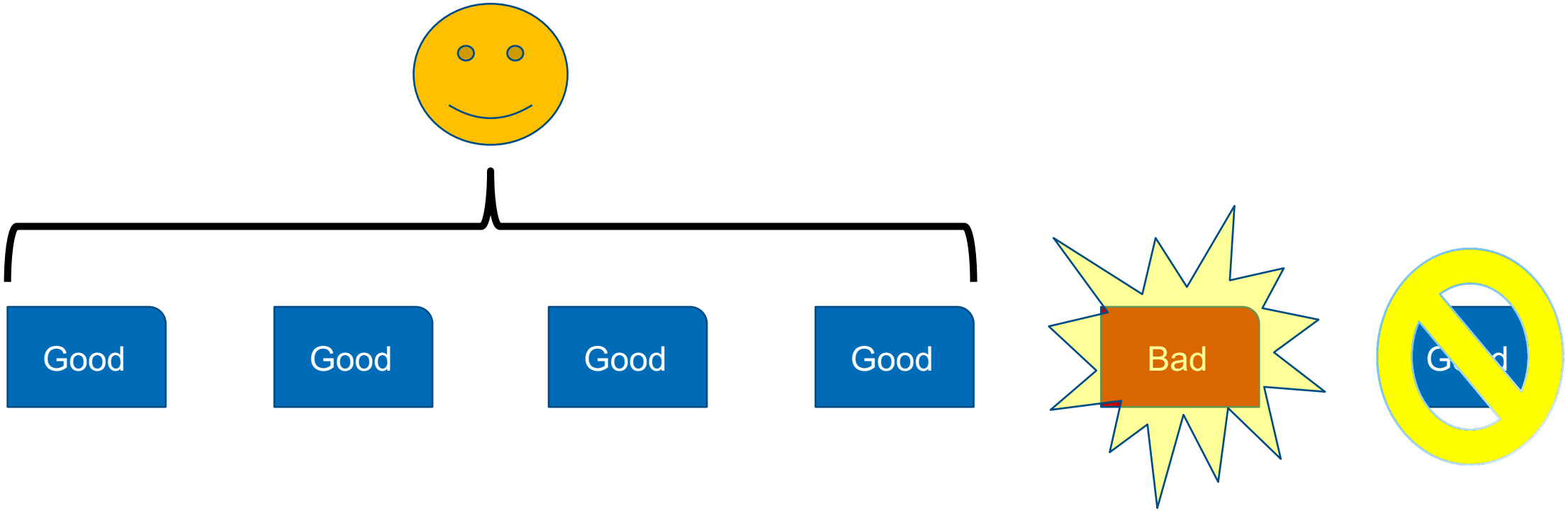
dropWhile(…)

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

ofNullable(…)

iterate(… ,…, …)

Stream – Scenario



takeWhile(…)

ASMIQ

```
public class StrictCustomer {

    public static void main(String[] args) {

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

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

    }

}
```

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

takeWhile(…)

dropWhile(…)

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

ofNullable(…)

iterate(… ,… ,…)

dropWhile(…)

ASMIQ

```
public class StrictCustomer {

    public static void main(String[] args) {

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

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

    }

}
```

dropWhile(...)

ASMIQ

```
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
```

ASMIQ

- **Combination of the two methods for extracting data:**

```
public static void main(String[] args)
{
    Stream<String> words = Stream.of("ab", "BLA", "Xy", "<START>",
                                     "WELCOME", "TO", "ORACLE", "CODE ONE",
                                     "<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
ORACLE
CODE ONE
```

Optional<T>

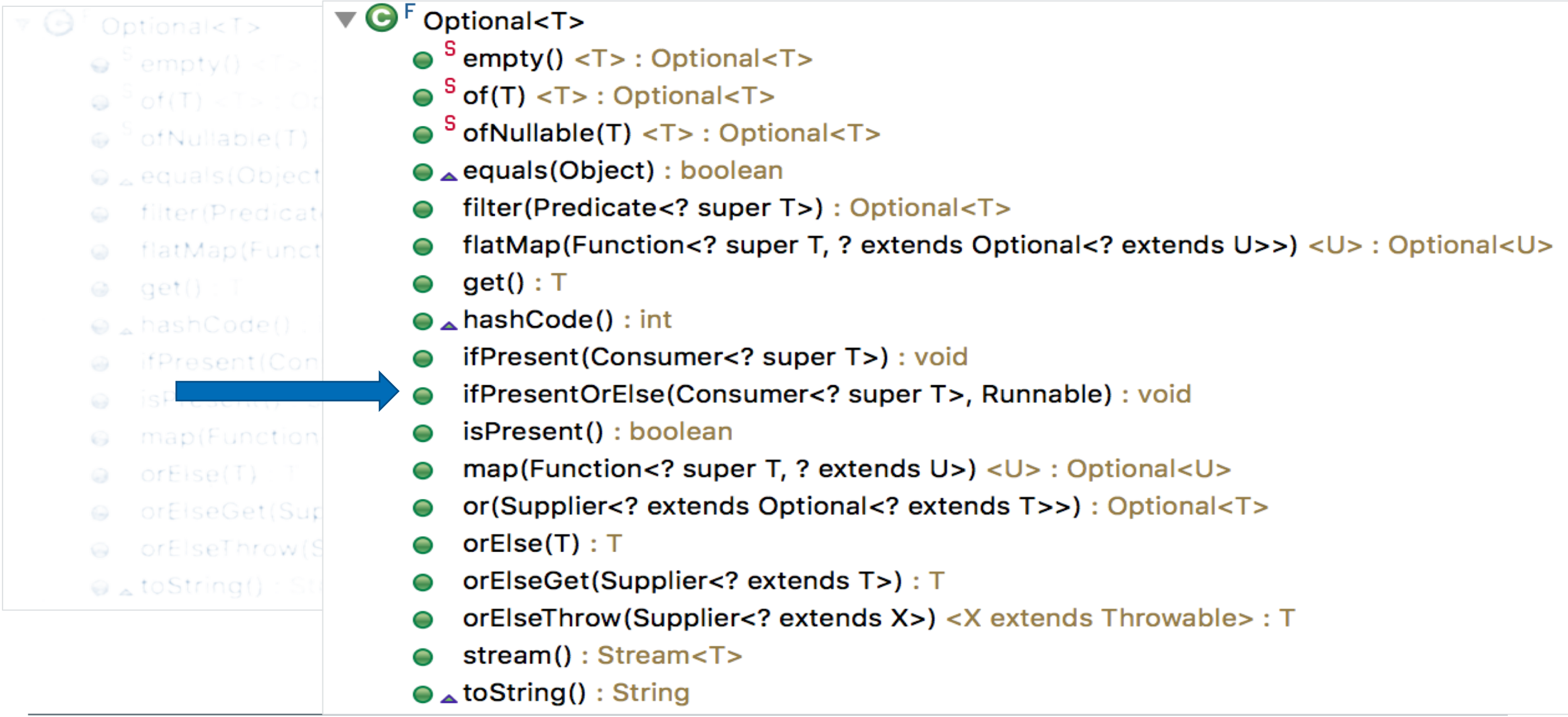
Optional<T>

ASMIQ

- Was introduced with Java 8 and facilitates the processing and modelling of optional values
- Initially with sound API, but 3 weak points in the following tasks:
  - The execution of actions even in a negative case,
  - The combination of the results of several calculations that optionally provide<T>.
  - Conversion into a stream<T>, for compatibility with the stream API, e.g. for frameworks working on streams

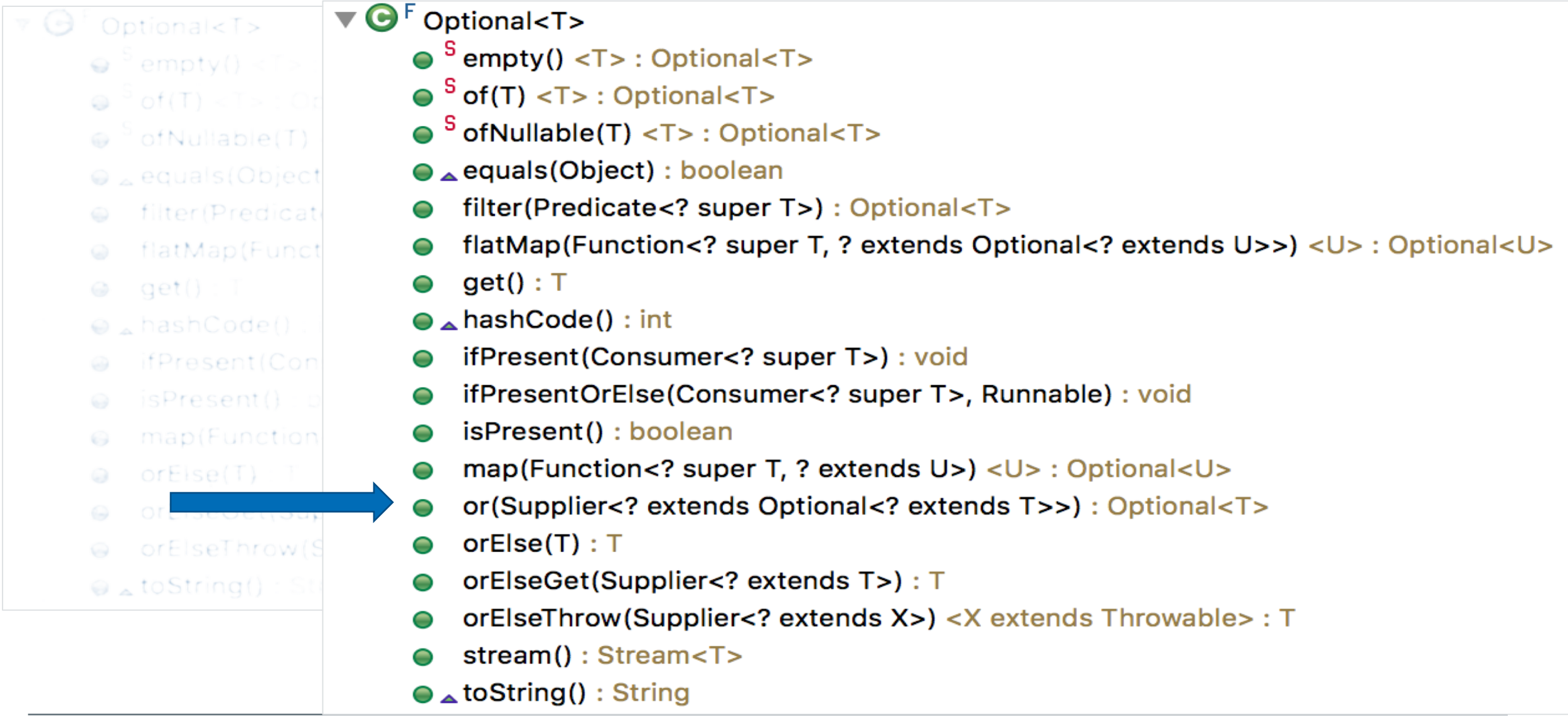
Optional<T> in JDK9

ASMIQ



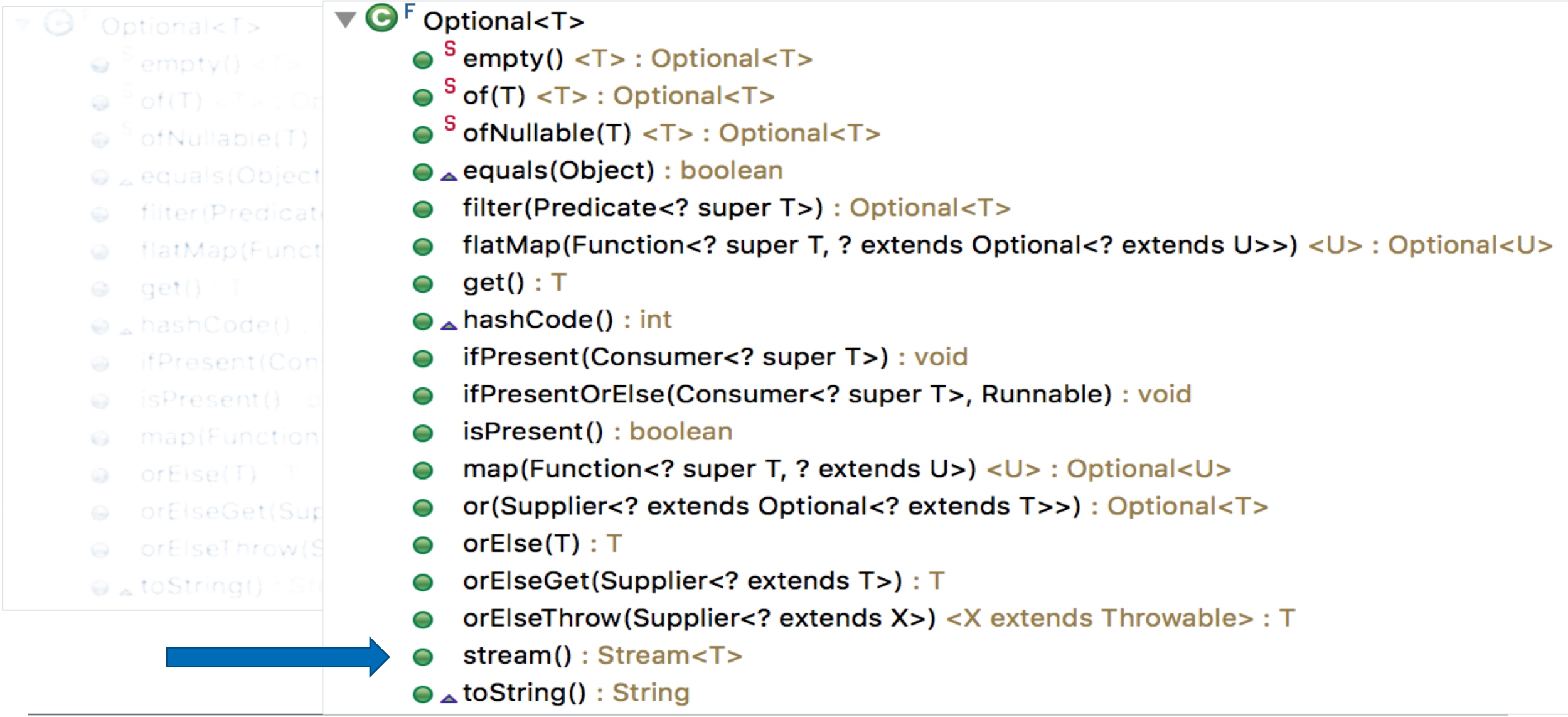
Optional<T> in JDK9

ASMIQ



Optional<T> in JDK9

ASMIQ



Why ifPresentOrElse(…) ?

ASMIQ

JDK8

```
public class Example1 {

    public static void main(String[] args) {

        Optional<String> welcomeString = getWelcomeString();

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

    }

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

}
```

ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction)

ASMIQ

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

JDK9

```
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");
    }


}
```

Why or(...) ?

ASMIQ

JDK8

```
public class PaymentProcessor {

    public static void main(String[] args) {
        Double balance = 0.0;
        Optional<Double> debitCardBalance = getDebitCardBalance();
        Optional<Double> creditCardBalance = getCreditCardBalance();
        Optional<Double> payPalBalance = getPayPalBalance();

        if (debitCardBalance.isPresent()) {

            balance = debitCardBalance.get();

        } else if (creditCardBalance.isPresent()) {

            balance = creditCardBalance.get();

        } else if (payPalBalance.isPresent()) {

            balance = payPalBalance.get();

        }

        if(balance > 0)
            System.out.println("Payment will be processed...");
        else
            System.out.println("Sorry insufficient balance");

    }
```

or(…)

JDK9

```
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.

java.util.Optional<T> (JDK 11)

ASMIQ

- In Java 11 there is another method, `isEmpty()`
- API analogous to Collections and String for checks
- Avoids the negation of `isPresent()`

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

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

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

Collection Factory Methods

Collection Factory Methods Intro

ASMIQ

- **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 ...**

Collection Factory Methods Intro

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

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

ASMIQ

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

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

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

Collection Factory Methods

ASMIQ

- **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)
```

Date API Enhacements

The Class LocalDate

ASMIQ

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

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

The Class LocalDate

ASMIQ

- Start 7. February => Jump 150 days into the future => 7 July

- **Day-Stream:** **Daily iteration limited to 4**
- **Month-Stream:** **Monthly Iteration** limited to 3
  => Specification of an alternative step size, here months:

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

Month-Stream
1971-02-07
1971-03-07
1971-04-07
```

Extensions in the Class `String`

Enhancements in java.lang.String: lines()

- 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!
```

Enhancements in java.lang.String: repeat()

- **Every now and then you are faced with the task of to string together a string several times, i.e. 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 delimeter = " -*- ";
        System.out.println(delimeter.repeat(6));
}

=>

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

Extensions in the Class `Files`

Utility-class java.nio.file.Files

ASMIQ

- 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
```

Utility-class java.nio.file.Files

ASMIQ

- **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
```

Exercises 1 – 7

PART 3: Multi-Threading with CompletableFuture

Multi-Threading and the Class CompletableFuture<T>

ASMIQ

- 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>`

Entry of CompletableFuture<T>

ASMIQ

- 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);
```

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?

Multi-Threading and the Class CompletableFuture<T>

ASMIQ

- **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);
}
```

Multi-Threading and the Class CompletableFuture<T>

ASMIQ

```
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>

ASMIQ

```
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?

Multi-Threading and the Class CompletableFuture<T>

ASMIQ

- 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`

Multi-Threading and the Class CompletableFuture<T>

ASMIQ

- 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?

Multi-Threading and the Class `CompletableFuture<T>`

ASMIQ

- 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!

Multi-Threading and the Class `CompletableFuture<T>`

ASMIQ

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.

Multi-Threading and the Class `CompletableFuture<T>`

ASMIQ

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());
```

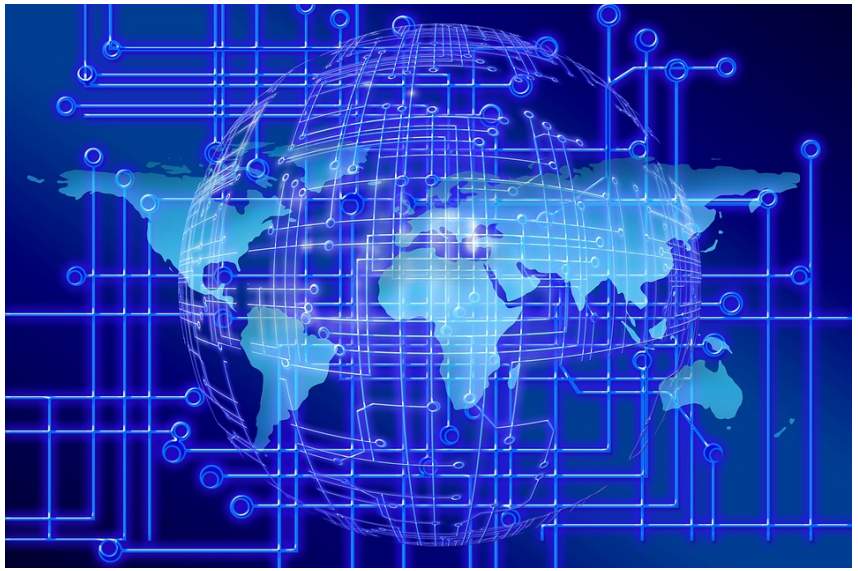
=> The processing can be continued promptly (without much delay or even blocking) even if the data determination should take longer.

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.

PART 4: HTTP/2 API



HTTP-access with Java 8

ASMIQ

- **URL + openStream()**

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

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

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

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

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

HTTP-access with Java 8

ASMIQ

- **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);
```

HTTP/2 API Async

ASMIQ

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

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

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

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

HTTP/2 API other

ASMIQ

- **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())
```

ASMIQ

PART 5:
What's new in Java 13

- Build-Tools und IDEs
- Syntax Enhancements

Build-Tools and IDEs

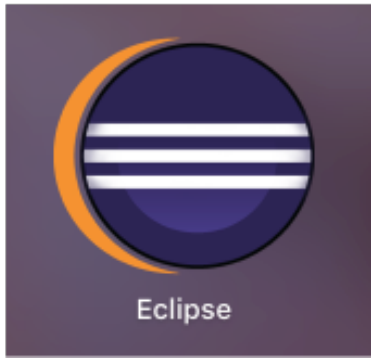
IDE & Tool Support

ASMIQ

- **Current IDEs & Tools are fundamentally good**

- **Eclipse: Version 2019-09 RC 1**
- **IntelliJ: Version 2019.2.2**
- **Maven: 3.6.1, Compiler-Plugin 3.8.1**
- **Gradle: 5.6.2**

- **Activation of preview features required**
  - In dialogues
  - In the build script

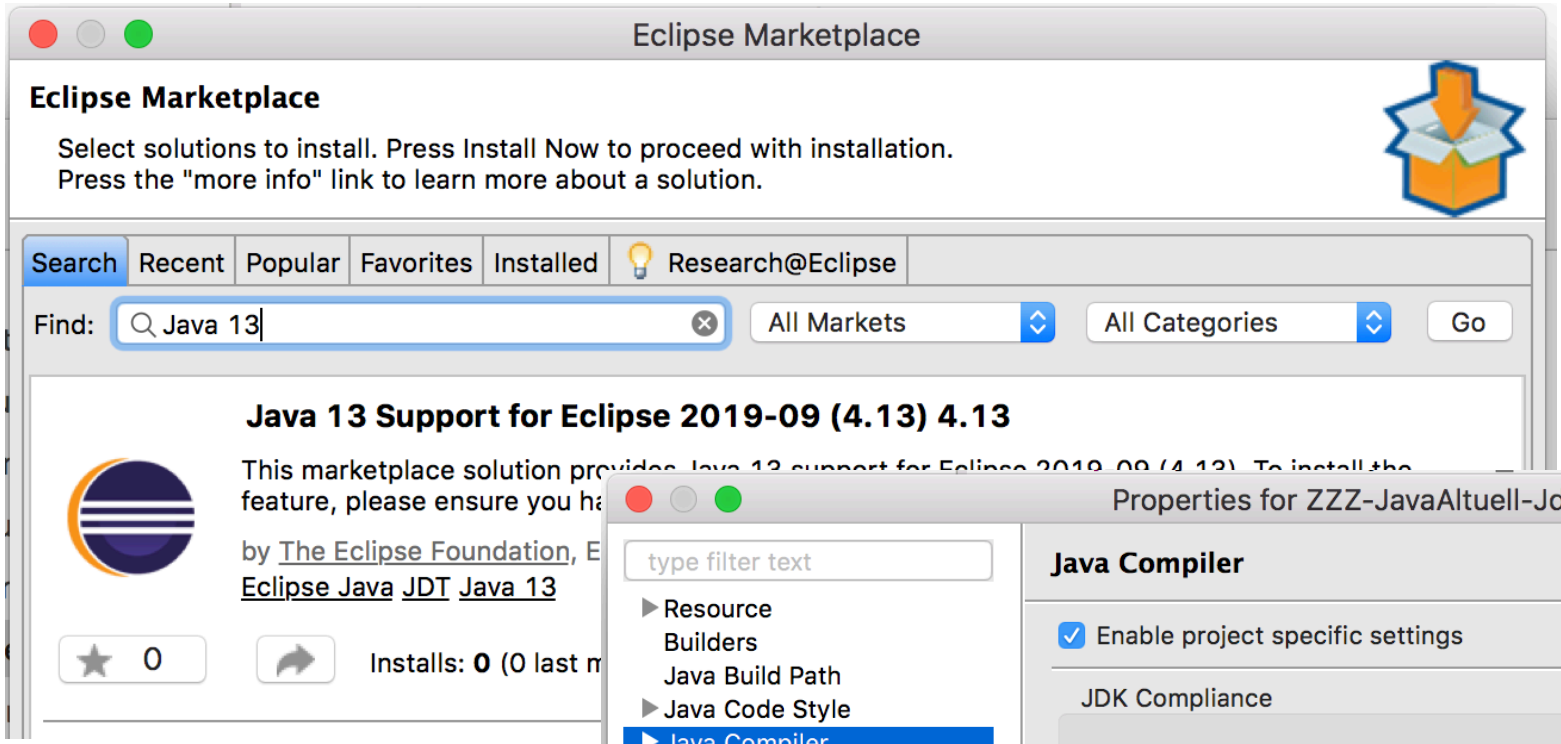


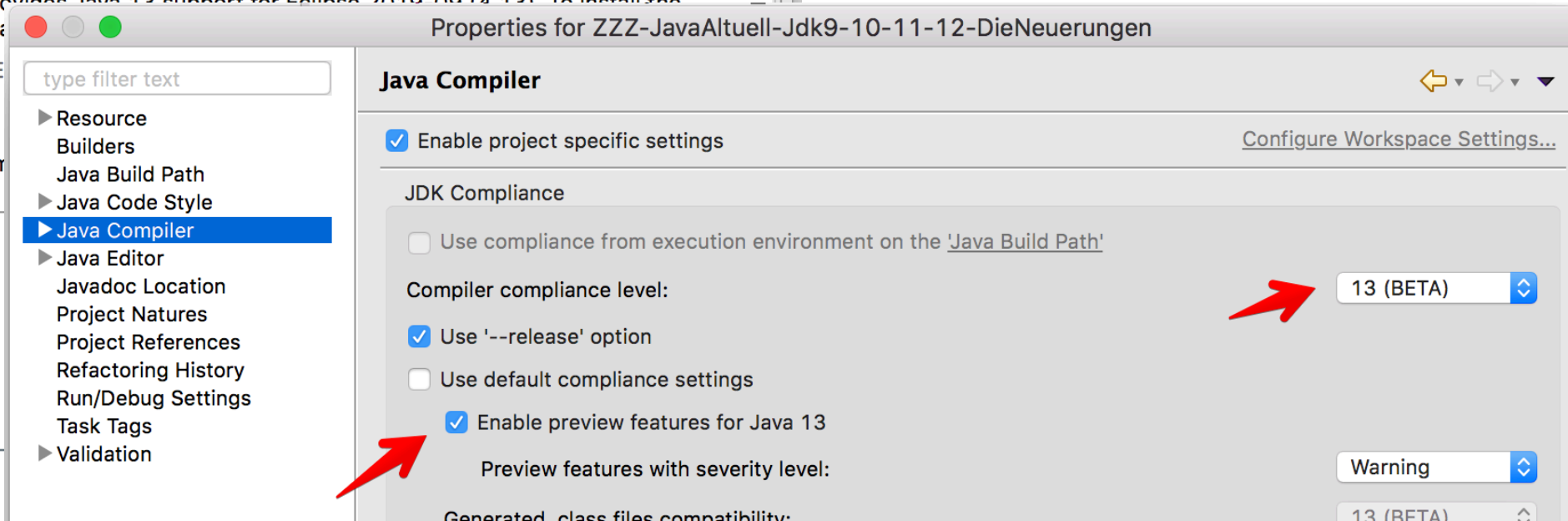
Maven™

Gradle

IDE & Tool Support

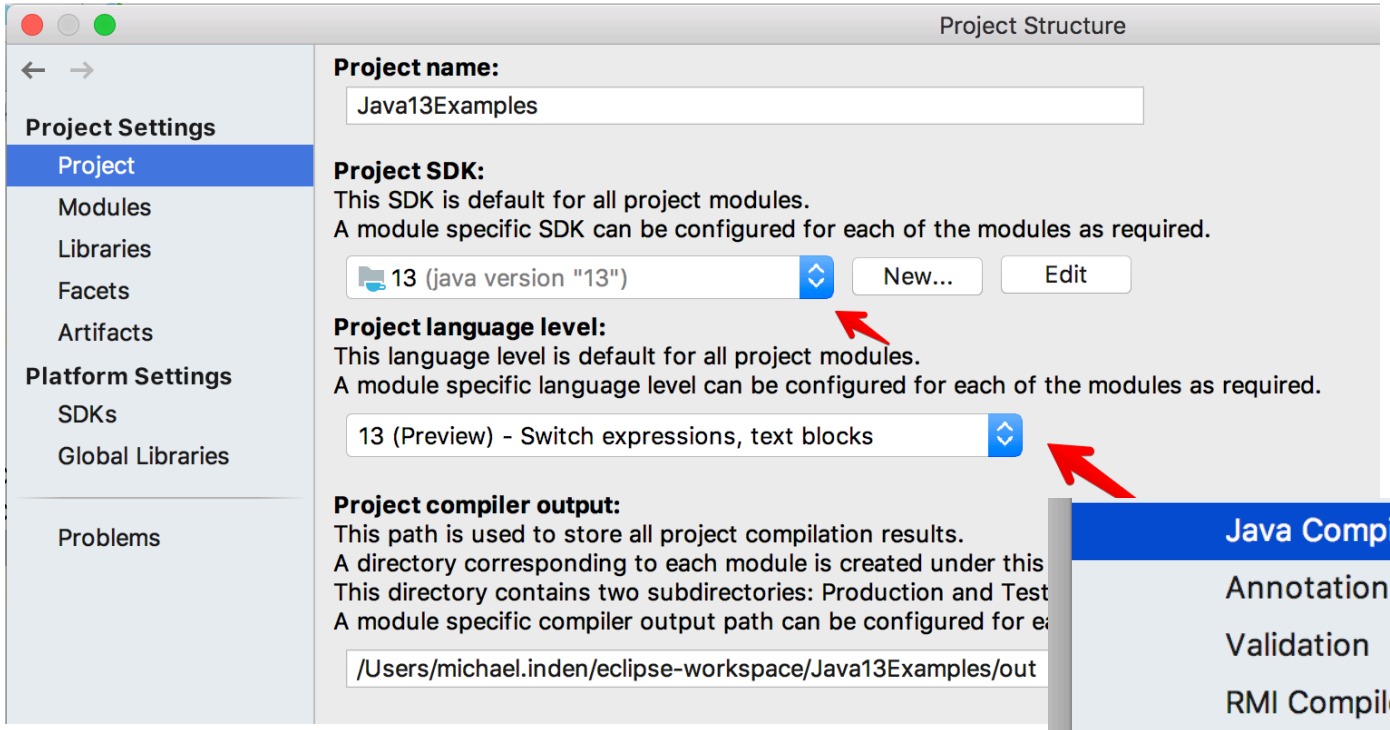
- **Installation of Plugin & Activation of Preview-Features needed**

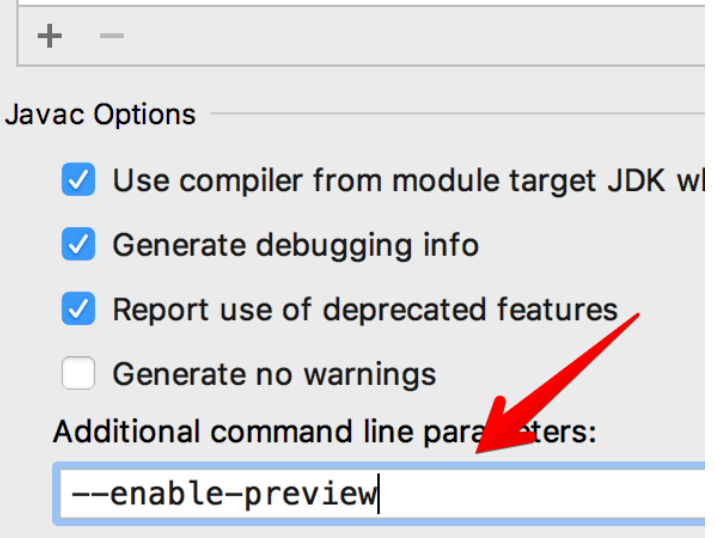




IDE & Tool Support

- **Activation of Preview-Features is needed**





IDE & Tool Support

ASMIQ

- **Activation of Preview-Features is needed**


```
sourceCompatibility=13
targetCompatibility=13

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

- **Additionally Gradle has to be started with Java 12 or lower => gradle.properties:**

```
// JAVA_HOME points to Java 12 or below in the console
// For compilation use Java 13
org.gradle.java.home=/Library/Java/JavaVirtualMachines/jdk-13.jdk/Contents/Home
```

IDE & Tool Support

ASMIQ

- **Activation of Preview-Features is needed**

Maven™

```
<plugins>
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.1</version>
        <configuration>
            <source>13</source>
            <target>13</target>
            <!– Important for Java 12/13 Syntax –->
            <compilerArgs>--enable-preview</compilerArgs>
        </configuration>
    </plugin>
</plugins>
```

Syntax Enhancements

Switch Expressions

ASMIQ

- 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

Switch Expressions: Retrospect

- **Mapping of weekdays to their length...**

```
DayOfWeek day = DayOfWeek.FRIDAY;
int numOfLetters = -1;

switch (day)
{
    case MONDAY:
    case FRIDAY:
    case SUNDAY:
        numOfLetters = 6;
        break;
    case TUESDAY:
        numOfLetters = 7;
        break;
    case THURSDAY:
    case SATURDAY:
        numOfLetters = 8;
        break;
    case WEDNESDAY:
        numOfLetters = 9;
        break;
}
```

Switch Expressions: Java 12 / 13 as a Remedy

ASMIQ

- **Mapping of weekdays to their length… elegantly with Java 12 / 13:**

```
DayOfWeek day = DayOfWeek.FRIDAY;
int numLetters = -1;

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

Switch Expressions: Java 12 / 13 as a Remedy

- **Mapping of weekdays to their length… elegantly with Java 12 / 13:**

Return-Value

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

Switch Expressions: Retrospect… Pitfalls

ASMIQ

- **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
```

Switch Expressions: Java 12 / 13 as a Remedy

ASMIQ

- **Mapping months to their names... elegantly with Java 12 / 13:**

```
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";
    };
}
```

Switch Expressions: Pitfalls with break in older Java

ASMIQ

- **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 numOfChars;

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

Switch Expressions: `yield` with return value

ASMIQ

With Java 12 / 13 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);
}
```

Text Blocks

Text Blocks

ASMIQ

- **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";
```

Text Blocks

- **NEW**

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

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

Text Blocks

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

Traditional String Literals

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

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

Text Blocks

- **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"});
```

Text Blocks

- **NEW**

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

Exercises 8 – 11

Questions?


HELP
IS ON THE
WAY


Michael Inden
Java
Die Neuerungen in Version 9 bis 12
Modularisierung, Syntax- und API-Erweiterungen
dpunkt.verlag


4.
Auflage
Michael Inden
Der Weg zum Java-Profi
Konzepte und Techniken für die professionelle Java-Entwicklung
Aktuell zu Java 9
dpunkt.verlag

Thank You