Best of Java 11 – 21

https://github.com/Michaeli71/JAX-London-Best-of-Java-11-21

Michael Inden

Head of Development, Independent SW consultant and author

Speaker Intro

- Michael Inden, Year of Birth 1971
- Diploma Computer Science, C.v.O. Uni Oldenburg
- ~8 ¼ Years **SSE** at Heidelberger Druckmaschinen AG in Kiel
- ~6 ¾ Years **TPL, SA** at IVU Traffic Technologies AG in Aachen
- ~4 ¼ Years **LSA / Trainer** at Zühlke Engineering AG in Zurich
- ~3 Years **TL / CTO** at Direct Mail Informatics / ASMIQ in Zurich
- Independent **Consultant, Conference Speaker and Trainer**
- **Since January 2022 Head of Development at Adcubum in Zurich**
- Author @ dpunkt.verlag and APress

E-Mail: michael_inden@hotmail.com

https://github.com/Michaeli71/JAX-London-Best-of-Java-11-21

Agenda

Workshop Contents

- **Preliminary notes / Build Tools & IDEs**
- **PART 1:** Syntax Enhancements & News and API-Changes up to Java 11
- **PART 2:** Additonal News and Changes up to Java 11

- **PART 3:** Syntax Enhancements in Java 12 to 17
- **PART 4:** News and API-Changes in Java 12 to 17
- **PART 5:** News in the JVM in Java 12 to 17

- **PART 6:** News in Java 18, 19 and 20
- **PART 7:** News in Java 21

- **Separate: Java Modularization**

https://github.com/Michaeli71/JAX-London-Best-of-Java-11-21

Workshop Contents

JDK	Release Date	Development Time	LTS	Workshop
Oracle JDK 9	9 / 2017	3,5 year	-	Part 1, 2
Oracle JDK 10	3 / 2018	6 months	-	Part 1, 2
Oracle JDK 11	**9 / 2018**	**6 months**	**yes, *commercial***	**Part 1, 2**
Oracle JDK 12	3 / 2019	6 months	-	Part 3, 4, 5
Oracle JDK 13	9 / 2019	6 months	-	Part 3, 4, 5
Oracle JDK 14	3 / 2020	6 months	-	Part 3, 4, 5
Oracle JDK 15	9 / 2020	6 months	-	Part 3, 4, 5
Oracle JDK 16	3 / 2021	6 months	-	Part 3, 4, 5
Oracle JDK 17	**9 / 2021**	**6 months**	**yes, *„free“, but licence***	**Part 3, 4, 5**
Oracle JDK 18	3 / 2022	6 month	-	Part 6
Oracle JDK 19	9 / 2022	6 month	-	Part 6
Oracle JDK 20	3 / 2023	6 month	-	Part 6
Oracle JDK 21	**9 / 2023**	**6 month**	**yes, *„free“, but licence***	**Part 7**

Long-Term Support Model

- **every two years Long Term Support (LTS) release**
  - get updates over a longer period of time
  - production versions
  - at the moment Java 8, 11, 17, 21
  - current LTS-Release is Java 21 (September 2023)
  - after Java 17 switching to a two years lts cadence => 2025 Java 25 LTS

- **other versions are "only" intermediate versions**
  - get updates only within 6 months
  - „Previews“ – features that are not finalized, integrated to get feedback
  - Ideal to get to know new features and to experiment (especially for private projects)
  - Incubators - even more rudimentary than Previews, are still at the stage where they may be completely abandoned again

Classification 6 month release cycle

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

Build-Tools and IDEs

Latest Java 21 installed (RC is sufficient)

- **Download latest version of Java 21**

```
$ java -version
openjdk version "21" 2023-09-19
OpenJDK Runtime Environment (build 21+35-2513)
OpenJDK 64-Bit Server VM (build 21+35-2513, mixed mode, sharing)
```

- **Activate Preview Features when working on console**

```
% java --enable-preview --source 21 src/main/java/HelloJava21.java
Hello Java 21
```

```
import javax.lang.model.SourceVersion;

public class HelloJava21 {
   public static void main(String[] args) {
      System.out.println("Hello Java " + SourceVersion.RELEASE_21.runtimeVersion());
   }
}
```

IDE & Tool Support for Java 21

- Eclipse: Version 2023-09 with additional plugin
- IntelliJ: Version 2023.2.2
- Maven: 3.9.4, Compiler Plugin: 3.11.0
- Gradle: 8.3
- Activation of Preview Features / Incubator necessary
  - In Dialogs
  - Im Build Scripts

IDE & Tool Support Java 21

- Eclipse 2023-09 with Plugin
- Activation of Preview Features / Incubator necessary

IDE & Tool Support

- **Activation of Preview Features / Incubator necessary**

IDE & Tool Support Java 21

- **Activation of Preview Features / Incubator necessary**

Gradle 8.3

```
sourceCompatibility=21
targetCompatibility=21
```

```
tasks.withType(JavaCompile).configureEach {
    options.compilerArgs += ["--enable-preview"]
}
```

```
tasks.withType(JavaCompile).configureEach {
    options.compilerArgs += ["--enable-preview",
                             "--add-modules", "jdk.incubator.vector"]
}
```

IDE & Tool Support

- **Activation of Preview Features / Incubator necessary**

Maven™ 3.9.4

```
<plugins>
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.11</version>
        <configuration>
            <source>21</source>
            <target>21</target>
            <!- Important for Preview Features -->
            <compilerArgs>--enable-preview</compilerArgs>
        </configuration>
    </plugin>
</plugins>
```

```
<plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.11.0</version>
    <configuration>
        <release>21</release>
        <compilerArgs>
            <arg>--enable-preview</arg>
            <arg>--add-modules</arg>
            <arg>jdk.incubator.vector</arg>
        </compilerArgs>
    </configuration>
</plugin>
```

IDE & Tool Support Java 21 for Vector API Incubator

Run Configurations

Create, manage, and run configurations

Run a Java application

Name: VectorApiExample

Main | (x)= Arguments | JRE | Dependencies | Source | Environment | Common | Prototype

Program arguments:

Variables...

VM arguments:

--add-modules jdk.incubator.vector

Variables...

- [x] Use the -XstartOnFirstThread argument when launching with SWT
- [x] Use the -XX:+ShowCodeDetailsInExceptionMessages argument when launching
- [ ] Use @argfile when launching

Working directory:

- (•) Default: ${workspace_loc:Java18Examples}
- ( ) Other:

Workspace... File System... Variables...

Show Command Line | Revert | Apply

IDE & Tool Support for Vector API Incubator

Preferences

Appearance & Behavior
- Appearance
- New UI
- Menus and Toolbars
- System Settings
- File Colors
- Scopes
- Notifications
- Quick Lists
- Path Variables
- Presentation Assistant

Keymap

Editor

Plugins

Version Control

Build, Execution, Deployment
- Build Tools
- Compiler
  - Excludes
  - Java Compiler
  - Annotation Processors
  - Validation
  - RMI Compiler
  - Groovy Compiler

Build, Execution, Deployment › Compiler › Java Compiler Reset

Use compiler: Javac

☑ Use '--release' option for cross-compilation (Java 9 and later)

Project bytecode version: Same as language leve

Per-module bytecode version:

Module	Target bytecode version
Java21Examples	21

Javac Options

☑ Use compiler from module target JDK when possible

☑ Generate debugging info

☑ Report use of deprecated features

☐ Generate no warnings

Additional command line parameters: ('/' recommended in paths for cross-platform configurations)

```
--enable-preview --add-modules jdk.incubator.vector
```

Override compiler parameters per-module:

Module	Compilation options
Java21Examples	--enable-preview --add-modules jdk.incubator.vector

Cancel Apply OK

IDE & Tool Support for Vector API Incubator

Run/Debug Configurations

Application
- VectorApiExample
- JEP430_StringTemplates_Advanced
- JEP443_UnnamedVarsAndPatterns
- SequencedCollectionExample

Edit configuration templates...

Name: VectorApiExample

Store as project file

Build and run

Modify options ⌥M

java 21 SDK of 'Java21E

--enable-preview --add-modules jdk.incubator.vector

api.VectorApiExample

Program arguments

Press ⌥ for field hints

Working directory: /Users/michaelinden/java8-workspace/Java21Examples

Environment variables:

Separate variables with semicolon: VAR=value; VAR1=value1

Open run/debug tool window when started

Code Coverage — Modify

Packages and classes to include in coverage data

Run | Apply | Cancel | OK

PART 1: Syntax Enhancements & News and API-Changes up to Java 11

Syntax Enhancements

@Deprecated-Annotation

- @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)

- **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.**

```
var justDeclaration;     // no value => no definition
var numbers = {0, 1, 2}; // missing type
```

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

- **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?

Local Variable Type Inference => var

- 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.**

Local Variable Type Inference Pitfall

- 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?

Local Variable Type Inference Pitfalls

- **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);
```

News and API-Changes up to Java 11

- Optional<T>
- Collection Factory Methods
- Strings

Optional<T>

Optional<T>

- **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**

Class Optional<T>

- **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>.**

why ifPresentOrElse(…) ?

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(…) ?

JDK 8

```
public class Example1 {

    public static void main(String[] args) {

        Optional<String> welcomeString = getWelcomeString();

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

    }

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

}
```

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

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");
    }


}
```

Optional<T> in JDK 9

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

why or(…) ?

JDK 8

```
public class PaymentProcessor {

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

        if (debitCardBalance.isPresent()) {

            balance = debitCardBalance.get();

        } else if (creditCardBalance.isPresent()) {

            balance = creditCardBalance.get();

        } else if (payPalBalance.isPresent()) {

            balance = payPalBalance.get();

        }

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

    }
```

or(…)

JDK 9

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

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

- `or(Supplier<Optional<T>> supplier)` 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.

Collection Factory Methods

Collection Factory Methods Intro

- **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 Literals

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

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

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

Collection Factory Methods

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

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

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

Collection Factory Methods

- **Behavior quite intuitive for lists ...**

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

MAX
MORITZ
MIKE
```

Collection Factory Methods

- **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)
```

Extensions in the Class `String`

Enhancements in `java.lang.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()`

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

- **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`.**

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 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));
}
```

=>

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

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

- **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-Java-11-21

PART 2: Additonal News and Changes up to Java 11

- Date API
- Files
- Multi-Threading with CompletableFuture
- HTTP/2
- Direct Compilation
- JShell

Date API Enhancements

class LocalDate

- `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);
}
```

Class LocalDate

- 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
```

Extensions in the Class `Files`

Utility class `java.nio.file.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
```

Utility class `java.nio.file.Files`

- **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
```

Multi-Threading with CompletableFuture

https://www.edureka.co/blog/what-is-mutithreading/

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

- 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>`

Introduction to `CompletableFuture<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());
```

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

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>`

- **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>`

```
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?

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

- 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>`

- **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>`

- **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>`

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>`

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.

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

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.

HTTP/2 API

HTTP/2 API Intro

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

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

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

printResponseInfo(response);


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

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

HTTP/2 API Intro (var shines here)

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

var request = HttpRequest.newBuilder(uri).GET().build();
var asString = HttpResponse.BodyHandlers.ofString();

var httpClient = HttpClient.newHttpClient();
var response = httpClient.send(request, asString);

printResponseInfo(response);


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

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

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

HTTP/2 API Async I

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

var request = HttpRequest.newBuilder(uri).GET().build();
var asString = HttpResponse.BodyHandlers.ofString();

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

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

HTTP/2 API Async I – waiting with premature termination

```
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?

HTTP/2 API Async II

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

var request = HttpRequest.newBuilder(uri).GET().build();
var asString = HttpResponse.BodyHandlers.ofString();
var 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);
```

Direct Compilation
Launch Single-File Source-Code Programs

Direct Compilation

- 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

```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);
    }
}
```

Direct Compilation – Shebang Execution

```
#!/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

michaeli — java ◂ jshell — 138×40

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

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

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

jshell> import java.time.*

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

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

Java command line `jshell`

- Code execution without class and method declaration
- Semicolon at end of line can (partially) be omitted
- (partly) no exception handling necessary
- for each command a variable for the return value is automatically created ($1, $2, ...)
- declaration of methods and classes possible
- adding of modules possible at startup
- exit jshell: /exit

DEMO

JShell - What we learned so far?

- 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`

Deprecations

Removed APIs and libraries

- Modularization of the JDK permits the recognition of dependencies

Modules in One Lesson by Mark Reinhold – https://www.youtube.com/watch?v=rFhhLXcOBsk

Removed APIs and libraries

- Modularization of the JDK enables the removal of individual modules

- Intersections with the Java EE specifications have been removed:

  - `java.activation` JAF
  - `java.corba` CORBA
  - `java.transaction` JTA
  - `java.xml.bind` JAXB
  - `java.xml.ws` JAX-WS, SAAJ
  - `java.xml.ws.annotation` Common Annotations
  - `java.se.ee` Aggregator-Modul for these moduls

Removed APIs and libraries

- tools related to removed modules got eliminated

- **`jdk.xml.ws`** (JAX-WS)
  - `wsgen`
  - `wsimport`
- **`jdk.xml.bind`** (JAXB)
  - `schemagen`
  - `xjc`
- **`java.corba`** (CORBA)
  - `idlj`, `orbd`, `servertool`, `tnamesrv`

Starting with Java 11 for JAXB: External dependencies necessary

JAXBExample_JDK8
JAXBExample_JDK11

```
dependencies
{
    implementation "javax.xml.bind:jaxb-api:2.3.0"
    implementation "com.sun.xml.bind:jaxb-core:2.3.0"
    implementation "com.sun.xml.bind:jaxb-impl:2.3.0"
    runtimeOnly "javax.activation:activation:1.1.1"
}
```

Exercises PART 2

https://github.com/Michaeli71/JAX-London-Best-of-Java-11-21

PART 3: Syntax Enhancements in Java 12 to 17

- Syntax Enhancements for `switch`
- `Text Blocks`
- `Records`
- Syntax Enhancements for `instanceof`
- Sealed Types

Syntax Enhancements

Switch Expressions

- 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 as a Remedy

- **Mapping of weekdays to their length… elegantly with modern Java:**

```
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 as a Remedy

- **Mapping of weekdays to their length… elegantly with modern Java:**

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;
};
```

- **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

Switch Expressions: Retrospect… Pitfalls

- **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 as a Remedy

- **Mapping months to their names... elegantly with modern Java:**

```
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 versions

- **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

With modern Java it 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
```

Text Blocks

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";
```

Text Blocks

- **NEW**

```
String javascriptCode = """
                        function hello()
                        {
                            print("Hello World");
                        }

                        hello();
                        """;


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!"
         }
         """;
```

Text Blocks (Alignment)

```
jshell> var multiLine = """
   ...>                  One
   ...>                   Two
   ...>                    Three"""
multiLine ==> "One\n Two\n  Three"


jshell> var multiLine = """
   ...>                 _  One
   ...>                  _  Two
   ...>                   _  Three
   ...>                _ _ _  Four
   ...>                """
multiLine ==> " _  One\n  _  Two\n   _  Three\n_ _ _  Four\n"
```

Text Blocks

```
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?

Enhancement Record

```
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 construtors and methods

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

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

```
jshell> var topLeft = new MyPoint("23, 11")
topLeft ==> MyPoint[x=23, y=11]

jshell> System.out.println(topLeft);
MyPoint[x=23, y=11]
```

Records for DTOs / Parameter Value Objects

```
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"));
}
```

Records modeling Tupels? – Excursion Pair<T>

- 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?

Records for modelling Pairs and Tupels

```
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?

Records with 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;
    }
}
```

Records with validity checks (short cut)

```
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);
        }
    }
}
```

(For now)
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);
    }
}
```

Records
Beyond the Basics

Storage in different sets

- **Let's define a simple record and two different data sets:**

```
record SimplePerson(String name, int age, String city) {}

Set<SimplePerson> speakers = new HashSet<>();
speakers.add(new SimplePerson("Michael", 51, "Zürich"));
speakers.add(new SimplePerson("Michael", 51, "Zürich"));
speakers.add(new SimplePerson("Anton", 42, "Aachen"));

System.out.print(speakers);

Set<SimplePerson> sortedSpeakers = new TreeSet<>();
sortedSpeakers.add(new SimplePerson("Michael", 51, "Zürich"));
sortedSpeakers.add(new SimplePerson("Michael", 51, "Zürich"));
sortedSpeakers.add(new SimplePerson("Anton", 42, "Aachen"));

System.out.print(sortedSpeakers);
```

That should be the result, right?

```
[SimplePerson[name=Michael, age=51, city=Zürich],
SimplePerson[name=Anton, age=42, city=Aachen]]
```

Storage in different sets

```
[SimplePerson[name=Michael, age=51, city=Zürich], SimplePerson[name=Anton, age=42, city=Aachen]]
Exception in thread "main" java.lang.ClassCastException: class
b_slides.RecordInterfaceExample$1SimplePerson cannot be cast to class java.lang.Comparable
(b_slides.RecordInterfaceExample$1SimplePerson is in unnamed module of loader 'app';
java.lang.Comparable is in module java.base of loader 'bootstrap')
        at java.base/java.util.TreeMap.compare(TreeMap.java:1569)
        at java.base/java.util.TreeMap.addEntryToEmptyMap(TreeMap.java:776)
        at java.base/java.util.TreeMap.put(TreeMap.java:785)
        at java.base/java.util.TreeMap.put(TreeMap.java:534)
        at java.base/java.util.TreeSet.add(TreeSet.java:255)
        at b_slides.RecordInterfaceExample.main(RecordInterfaceExample.java:32)
```

Records and Interfaces

- **Let's correct the definition of the simple record and implement an interface:**

```
record SimplePerson2(String name, int age, String city)
       implements Comparable<SimplePerson2>
{
    @Override
    public int compareTo(SimplePerson2 other)
    {
        return name.compareTo(other.name);
    }
}

Set<SimplePerson2> sortedSpeakers = new TreeSet<>();
sortedSpeakers.add(new SimplePerson2("Michael", 51, "Zürich"));
sortedSpeakers.add(new SimplePerson2("Michael", 51, "Zürich"));
sortedSpeakers.add(new SimplePerson2("Anton", 42, "Aachen"));
System.out.println(sortedSpeakers);
```

```
[SimplePerson2[name=Anton, age=42, city=Aachen],
SimplePerson2[name=Michael, age=51, city=Zürich]]
```

What happens if we have multiple records that differ in more than just name?

Records and Interfaces + Comparator

- **Let's correct the definition of the comparator:**

```
record SimplePerson3(String name, int age, String city)
       implements Comparable<SimplePerson3>
{
    static Comparator<SimplePerson3> byAllAttributes = Comparator.
                                       comparing(SimplePerson3::name).
                                       thenComparingInt(SimplePerson3::age).
                                       thenComparing(SimplePerson3::city);
    @Override
    public int compareTo(SimplePerson3 other)
    {
        return byAllAttributes.compare(this, other);
    }
}
```

Builder …

Builder like

```
record SimplePerson(String name, int age, String city)
{
    SimplePerson(String name)
    {
        this(name, 0, "");
    }

    SimplePerson(String name, int age)
    {
        this(name, age, "");
    }

    SimplePerson withAge(int newAge)
    {
        return new SimplePerson(name, newAge, city);
    }

    SimplePerson withCity(String newCity)
    {
        return new SimplePerson(name, age, newCity);
    }
}
```

Builder like

- **Problem area, many attributes**

```
record ComplexPerson(String firstname, String surname, LocalDate birthday,
                     int height, int weight, String addressStreet,
                     String addressNumber, String city)
{
    public static void main(String[] args)
    {
        ComplexPerson john = new ComplexPerson("John", "Smith", LocalDate.of(2010, 6,  21),
                                               170, 90, "Fuldastrasse", "16a", "Berlin");

        ComplexPerson mike = new ComplexPersonBuilder().withFirstName("Mike").
                                                        withBirthday(LocalDate.of(2021, 1,21)).
                                                        withSurName("Peters").build();
        System.out.println(mike);
    }
}
```

- **But: `ComplexPersonBuilder` would have to be implemented by yourself**

What would be another way to reduce complexity?

Builder like

- **Define records for individual components**

```
record Address(String addressStreet, String addressNumber, String city) {}

record BodyInfo(int height, int weight) {}

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


record ReducedComplexPerson(PersonDTO person, BodyInfo bodyInfo, Address address)
{
    public static void main(String[] args)
    {
        var john = new PersonDTO("John", "Smith", LocalDate.of(2010, 6,  21));
        var bodyInfo = new BodyInfo(170, 90);
        var address = new Address("Fuldastrasse", "16a", "Berlin");

        var rcp = new ReducedComplexPerson(john, bodyInfo, address);
    }
}
```

Date Range … Immutability

Record for Date range

```
public class RecordImmutabilityExample
{
    public static void main(String[] args)
    {
        record DateRange(Date start, Date end) {}

        DateRange range1 = new DateRange(new Date(71,1,7), new Date(71,2,27));
        System.out.println(range1);

        DateRange range2 = new DateRange(new Date(71,6,7), new Date(71,2,27));
        System.out.println(range2);
    }
}
```

```
DateRange[start=Sun Feb 07 00:00:00 CET 1971, end=Sat Mar 27 00:00:00 CET 1971]
DateRange[start=Wed Jul 07 00:00:00 CET 1971, end=Sat Mar 27 00:00:00 CET 1971]
```

- **Include validity check for invariant start < end**

Record for Date range

```
record DateRange(Date start, Date end)
{
    DateRange
    {
        if (!start.before(end))
            throw new IllegalArgumentException("start >= end");
    }
}

DateRange range1 = new DateRange(new Date(71,1,7), new Date(71,2,27));
System.out.println(range1);

DateRange range2 = new DateRange(new Date(71,6,7), new Date(71,2,27));
System.out.println(range2);
```

```
DateRange[start=Sun Feb 07 00:00:00 CET 1971, end=Sat Mar 27 00:00:00 CET 1971]
Exception in thread "main" java.lang.IllegalArgumentException: start >= end
at b_slides.RecordImmutabilityExample3$1DateRange.<init>(RecordImmutabilityExample3.java:21)
at b_slides.RecordImmutabilityExample3.main(RecordImmutabilityExample3.java:28)
```

Record for Date range

```
record DateRange(Date start, Date end)
{
    DateRange
    {
        if (!start.before(end))
            throw new IllegalArgumentException("start >= end");
    }
}

DateRange range1 = new DateRange(new Date(71,1,7), new Date(71,2,27));
System.out.println(range1);

range1.start.setTime(new Date(71,6,7).getTime());
System.out. println(range1);
```

```
DateRange[start=Sun Feb 07 00:00:00 CET 1971, end=Sat Mar 27 00:00:00 CET 1971]
DateRange[start=Wed Jul 07 00:00:00 CET 1971, end=Sat Mar 27 00:00:00 CET 1971]
```

- **Immutability only for immutable attributes => ATTENTION: reference semantics in Java**

Record for Date range

```
record DateRange(LocalDate start, LocalDate end)
{
    DateRange
    {
        if (!start.isBefore(end))
            throw new IllegalArgumentException("start >= end");
    }
}

DateRange range1 = new DateRange(LocalDate.of(1971,1,7), LocalDate.of(1971,2,27));
System.out.println(range1);

DateRange range2 = new DateRange(LocalDate.of(1971,6,7), LocalDate.of(1971,2,27));
System.out.println(range2);
```

```
DateRange[start=1971-01-07, end=1971-02-27]
Exception in thread "main" java.lang.IllegalArgumentException: start >= end
at b_slides.RecordImmutabilityExample4$1DateRange.<init>(RecordImmutabilityExample4.java:21)
at b_slides.RecordImmutabilityExample4.main(RecordImmutabilityExample4.java:28)
```

DEMO & Hands on

Pattern Matching instanceof

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?

Pattern Matching instanceof

- **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
}
```

Pattern Matching instanceof

```
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());
}
```

Pattern Matching instanceof

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

Sealed Types

Sealed Types

- **Control inheritance and specify which classes can extend a base class, i.e. which other classes or interfaces may form subtypes of it.**

Sealed Types – Control inheritance

- **Specify which other classes or interfaces may subtype from it.**

```
public class SealedTypesExamples
{
    sealed interface MathOp
        permits BaseOp, Add, Sub  // <= permitted subtypes
    {
        int calc(int x, int y);
    }

    // With non-sealed you can provide base classes within the inheritance hierarchy
    non-sealed class BaseOp implements MathOp // <= Do not seal base class
    {
        @Override
        public int calc(int x, int y)
        {
            return 0;
        }
    }
    ...
```

With sealed we can seal an inheritance hierarchy and allow only the explicitly specified types. These must be sealed, non-sealed or final.

Sealed Types

```
..
    // direct implementiation must be final
    final class Add implements MathOp
    {
        @Override
        public int calc(int x, int y)
        {
            return x + y;
        }
    }
    final class Sub implements MathOp
    {
        ...
    }

    // implementiation of base SHOULD be final
    final class Mult extends BaseOp
    {
    }
    final class Div extends BaseOp
    {
    }
}
```

A class that is marked as sealed must have subclasses. which in turn are listed after permits

A class marked as non-sealed can function as a base class and classes can be derived from it.

A class marked as final forms the end point of a derivation hierarchy as usual.

Things to know for Sealed Types

- specify which other classes or interfaces may subtype from it.
- Sealed types can expose behavior over interfaces in the development of libraries, but allow control over possible implementations.
- Sealed types restrict with regard to the extensibility of class hierarchies and should therefore be used with caution. Flexibility not foreseen during implementation can be subsequently disruptive.

Exercises PART 3

https://github.com/Michaeli71/JAX-London-Best-of-Java-11-21

PART 4:
What’s new in Java 12 to 17

- String APIs
- `Teeing()`-Kollektor
- `Stream.toList()`

Extensions in class `String`

Extension in `java.lang.String`

- **The String class has existed since JDK 1.0 and has experienced only a few API changes in the meantime.**

- **With Java 11 this changed: 6 new methods were introduced.**

- **In Java 12 the following two are added:**

  - `indent()` - Adjusts the indentation of a string.
  - `transform()` - Allows actions to transform a string

Extension in `java.lang.String: indent()`

- 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());
```

```
'    Test
'
9
```

- However, negative values are also allowed:

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

```
'6789
'
5
```

Extension in `java.lang.String: transform()`

- **Example: all characters of a string with `transform()`**
  - Convert to UPPERCASE
  - Then remove T
  - Finally split

```
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]
```

Does anyone see the potential benefit?

- **Analogous to `map()` in streams, for connecting transformations one after the other.**
- **Rather theoretical practical**

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

Teeing() Collector

Stream API Collectors in JDK 9

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.

Slide-in Stream API – Special Collector

Stream API Collectors in JDK 9

Two newly available collectors

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

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

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

Stream API Collectors in JDK 9

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

Example:

```
var programming = Stream.of("Java", "JavaScript", "Groovy", "JavaFX", "Spring", "Java");

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

With new collector:

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

As a result:

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

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

Stream API Collectors in JDK 9

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?

Stream API Collectors in JDK 9

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}
```

What is still missing?

Collectors

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)
```

Collectors

```
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(2, 3, 5, 7, 11, 13, 17);
        System.out.println(calcCountAndSum(primeNumbers));
}

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

Collectors

```
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(2, 3, 5, 7, 11, 13, 17);
        System.out.println(calcCountAndSum(primeNumbers));
}

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

```
LongPair[count=6, sum=21]
LongPair[count=7, sum=58]
```

Collectors

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);
```

Collectors

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()),
                                  filtering(endsWithM, toList()),
                                  (list1, list2) -> List.of(list1, list2));
System.out.println(result);
```

[Michael, Mike]

Collectors

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()),
                                  filtering(endsWithM, toList()),
                                  (list1, list2) -> List.of(list1, list2));
System.out.println(result);
```

[Michael, Mike]

[Tim, Tom]

Collectors

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()),
                                  filtering(endsWithM, toList()),
                                  (list1, list2) -> List.of(list1, list2));

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

[Michael, Mike]

[Tim, Tom]

[[Michael, Mike], [Tim, Tom]]

Java 16

Stream => List ... it was so cumbersome ...

```
List<String> namesMi = Stream.of("Tim", "Tom", "Mike", "Michael").
                              filter(str -> str.startsWith("Mi")).
                              collect(Collectors.toList());
```

FINALLY… toList()

```
List<String> namesMi = Stream.of("Tim", "Tom", "Mike", "Michael").
                              filter(str -> str.startsWith("Mi")).
                              toList();
```

Exercises PART 4

https://github.com/Michaeli71/JAX-London-Best-of-Java-11-21

PART 5: News in the JVM in Java 12 to 17

- Helpful `NullPointerExceptions`
- `JMH (Microbenchmarks)`
- `JPackage`
- ~~`JavaScript-Engine`~~ `(got removed)`

N P E

https://www.freepik.com/free-vector/realistic-bomb-illustration_3886825.htm

Helpful NullPointerExceptions

```
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)
```

Helpful NullPointerExceptions

```
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)
```

Helpful NullPointerExceptions

```
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)
```

Helpful NullPointerExceptions

```
public static void main(final String[] args)
{
    int width = getWindowManager().getWindow(5).size().width();
    System.out.println("Width: "  + width);
}
```

```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"jvm.NPE_Third_Example$Window.size()" because the return value of
"jvm.NPE_Third_Example$WindowManager.getWindow(int)" is null
at jvm.NPE_Third_Example.main(NPE_Third_Example.java:7)
```

Helpful NullPointerExceptions

```
public static WindowManager getWindowManager()
{
    return new WindowManager();
}


public static class WindowManager
{
    public Window getWindow(final int i)
    {
        return null;
    }
}


public static record Window(Size size) {}
public static record Size(int width, int height) {}
```

JMH

Benchmarking Intro

- **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.

Simpe Start Stop measurements (Please don't do it!!)

- 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 (Avoid if possible!)

- Repeated Start-/Stop measurements using more precise `System.nanoTime()`

```
// 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.

Repeated Start-/Stop Measurements with Warm-up (Gets complicated)

- **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;
```

JEP 230: Micobenchmark Suite / JMH

- 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

Microbenchmarks with JMH

- 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
```

Microbenchmarks with JMH

- 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.

Microbenchmarks with JMH

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

```
├── pom.xml
├── src
│   └── main
│       └── java
│           └── org
│               └── sample
│                   ├── MyBenchmark.java
│                   ├── FirstIntegerBenchmark.java
│                   └── SimpleMathBenchmark.java
└── target
    ├── benchmarks.jar
```

- **In 2 steps to a benchmark**

```
1) mvn clean package
2) java -jar target/benchmarks.jar
```

Own Microbenchmark with JMH

```
@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 testIndexOf(SearchState state) {
        return state.text.indexOf("M");
    }

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

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

Own Microbenchmark with JMH

```
@BenchmarkMode(Mode.AverageTime)
@Fork(2)
@State(Scope.Benchmark)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
public class SimpleStringJoinBenchmark
{
    private String from = "Michael";
    private String to = "Participants";
    private String subject = "Benchmarking with JMH";

    @Benchmark
    public String stringPlus(Blackhole blackhole)
    {
        String result = "From: " + from + "\nTo: " + to + "\nSubject: " + subject;
        blackhole.consume(result);
        return result;
    }
```

Inspired by http://alblue.bandlem.com/2016/04/jmh-stringbuffer-stringbuilder.html, but now using BlackHole and slightly modified

Own Microbenchmark with JMH

```
@Benchmark
public String stringPlusEqual(Blackhole blackhole)
{
    String result = "From: " + from;
    result += "\nTo: " + to;
    result += "\nSubject: " + subject;

    blackhole.consume(result);
    return result;
}

@Benchmark
public String builderAppendChained(Blackhole blackhole)
{
    String result = new StringBuilder().append("From: ").append(from).
                                        append("\nTo: ").append(to).
                                        append("\nSubject: ").append(subject).
                                        toString();
    blackhole.consume(result);
    return result;
}
```

ATTENTION – Own Microbenchmarks mit JMH

```java
@State(Scope.Benchmark)
public static class MyBenchmarkState {
    @Param({ "10000", "100000" })
    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;
}
```

ATTENTION – Own Microbenchmarks mit JMH

```java
@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 sb = new StringBuilder();
    for (int i = 0; i < state.value; i++) {
        sb.append("ABC");
    }
    String result = sb.toString();
    blackhole.consume(result);
    return result;
}
```

Microbenchmarks With JMH, adjust to support Java 17

- **POM adjustment for java 17:**

```
<!-- Java source/target to use for compilation. -->
        <javac.target>1.8</javac.target>
=>
<javac.target>17</javac.target>

<jmh.version>1.33</jmh.version>

<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 +=, Sting.concat(), StringBuilder.append()`

Sample 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

Sample results

Benchmark	Mode	Cnt	Score		Error	Units
SimpleStringJoinBenchmark.builderAppendChained	avgt	10	46.308	±	7.950	ns/op
SimpleStringJoinBenchmark.builderMultipleAppend	avgt	10	127.312	±	14.935	ns/op
SimpleStringJoinBenchmark.stringConcat	avgt	10	83.888	±	18.979	ns/op
SimpleStringJoinBenchmark.stringPlus	avgt	10	38.871	±	2.823	ns/op
SimpleStringJoinBenchmark.stringPlusEqual	avgt	10	39.346	±	3.015	ns/op

Sample results

Benchmark	Mode	Cnt	Score		Error	Units
SimpleStringJoinBenchmark.builderAppendChained	avgt	10	46.308	±	7.950	ns/op
SimpleStringJoinBenchmark.builderMultipleAppend	avgt	10	127.312	±	14.935	ns/op
SimpleStringJoinBenchmark.stringConcat	avgt	10	83.888	±	18.979	ns/op
SimpleStringJoinBenchmark.stringPlus	avgt	10	38.871	±	2.823	ns/op
SimpleStringJoinBenchmark.stringPlusEqual	avgt	10	39.346	±	3.015	ns/op

Benchmark	(value)	Mode	Cnt	Score		Error	Units
StringJoinBenchmark.concatUsingStringBuilder	10000	avgt	10	0.016	±	0.001	ms/op
StringJoinBenchmark.concatUsingStringBuilder	100000	avgt	10	0.172	±	0.009	ms/op
StringJoinBenchmark.stringConcatABC	10000	avgt	10	9.634	±	0.812	ms/op
StringJoinBenchmark.stringConcatABC	100000	avgt	10	662.953	±	4.029	ms/op
StringJoinBenchmark.stringPlusABC	10000	avgt	10	7.926	±	0.149	ms/op
StringJoinBenchmark.stringPlusABC	100000	avgt	10	662.343	±	2.912	ms/op

JPackage

- PackagingDemo
  - JRE System Library [JavaSE-16]
  - src/main/java
    - de.java17
      - ApplicationExample.java
        - ApplicationExample
          - main(String[]) : void
  - src
  - build.gradle
  - pom.xml

```java
public class ApplicationExample {

    public static void main(String[] args) {

        System.out.println("First JPackage");

        JOptionPane.showConfirmDialog(null, "Generated by jpackage", "DEMO",
                JOptionPane.OK_CANCEL_OPTION, JOptionPane.INFORMATION_MESSAGE, null)
    }
}
```

How to build?

```
mvn clean install
```

```
jpackage --input target/ --name JPackageDemoApp --main-jar PackagingDemoExamples-1.0.0-SNAPSHOT.jar --main-class de.java17.ApplicationExample --type dmg --java-options '--enable-preview'
```

What about 3rd party libraries?

```java
public class ApplicationExample {

    public static void main(String[] args) {
        System.out.println("First JPackage");

        Joiner joiner = Joiner.on(":");
        String result = joiner.join(List.of("Michael", "mag", "Python"));
        System.out.println(result);
        JOptionPane.showConfirmDialog(null, result);
    }
}
```

```
<!-- https://mvnrepository.com/artifact/com.google.guava/guava -->
<dependency>
<groupId>com.google.guava</groupId>
<artifactId>guava</artifactId>
<version>31.0.1-jre</version>
</dependency>
```

How to include the lib in the Application?

```
<plugin>
    <artifactId>maven-dependency-plugin</artifactId>
    <executions>
        <execution>
            <phase>process-sources</phase>
            <goals>
                <goal>copy-dependencies</goal>
            </goals>

            <configuration>
                <outputDirectory>target</outputDirectory>
                <includeScope>runtime</includeScope>
                <excludeScope>test</excludeScope>
            </configuration>
        </execution>
    </executions>
</plugin>
```

- target
  - classes
  - generated-sources
  - maven-archiver
  - maven-status
  - checker-qual-3.12.0.jar
  - error_prone_annotations-2.7.1.jar
  - failureaccess-1.0.1.jar
  - guava-31.0.1-jre.jar
  - j2objc-annotations-1.3.jar
  - jsr305-3.0.2.jar
  - listenablefuture-9999.0-empty-to-avoid-conflict-with-guav
  - PackagingDemoExamples-1.0.0-SNAPSHOT.jar

IntelliJ IDEA CE
iPhoto
JPackageDemoApp
Kalender
Karten

Select an Option
Michael:mag:Python
Cancel
No
Yes

DEMO & Hands on

~~Nashorn Java Script Engine~~

~~(deprected since Java 11 deprecated, removed with Java 15)~~

JShell-API

- 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());

    // unfortunately no (direct) access to value, only via varValue()
    jshell.variables().map(varSnippet -> "variable: '" + varSnippet.name() + "' / " +
                                         "type: " + varSnippet.typeName() + "' / " +
                                         "value: " + jshell.varValue(varSnippet))
                .forEach(System.out::println);
}
```

DEMO

Exercies PART 5

https://github.com/Michaeli71/JAX-London-Best-of-Java-11-21

Pattern Matching for `switch` (PREVIEW in Java 17)

Pattern Matching for `switch`

- Suppose we needed to write a generic method for formatting values.
- Provided we use Java 17, we can write this reasonably readably using pattern matching and `instanceof` as follows:*

```
static String formatterJdk16instanceof(Object obj) {
    String formatted = "unknown";
    if (obj instanceof Integer i) {
        formatted = String.format("int %d", i);
    } else if (obj instanceof Long l) {
        formatted = String.format("long %d", l);
    } else if (obj instanceof Double d) {
        formatted = String.format("double %f", d);
    } else if (obj instanceof String s) {
        formatted = String.format("String %s", s);
    }
    return formatted;
}
```

*the whole thing would be much worse without Java 16, since we would have to add a line of cast for each type

Pattern Matching bei `switch`

- **This syntax innovation is also supported for `switch` with Java 17 (initially in the form of preview features):**

```
static String formatterJdk17switch(Object obj) {
    String formatted = switch (obj)
    {
        case Integer i -> String.format("int %d", i);
        case Long l -> String.format("long %d", l);
        case Double d -> String.format("double %f", d);
        case String s -> String.format("String %s", s);
        default -> "unknown";
    };
    return formatted;
}
```

- **In this way, a `switch` can be used to check the type of an object and depending on a match fill the variable with an appropriate value.**

Pattern Matching for `switch`

- **Until Java 17 it was not possible to handle the value `null` in the cases of a `switch`, instead the following special handling was necessary:**

```
static void switchSpecialNullSupport(String str) {
    if (str == null) {
        System.out.println("special handling for null");
        return;
    }

    switch (str) {
        case "Java", "Python" -> System.out.println("cool language");
        default -> System.out.println("everything else");
    }
}
```

Pattern Matching for `switch`

- **Starting with Java 17 and preview features enabled, we can now specify `null`-values and unify the whole construct:**

```
static void switchSupportingNull(String str) {
    switch (str) {
        case null -> System.out.println("null is allowed in preview");
        case "Java", "Python" -> System.out.println("cool language");
        default -> System.out.println("everything else");
    }
}
```

- **To reproduce it, you need to enable preview features appropriately in JShell:**

```
jshell --enable-preview
|  Welcome to JShell -- Version 17.0.6
|  For an introduction type: /help intro
```

Pattern Matching for `switch`

- Analogous to `instanceof`, queries like the following are now also possible in the `cases`:

```
static void processData(Object obj) {
    switch (obj) {
        case String str && str.startsWith("V1") -> System.out.println("Processing V1");
        case String str && str.startsWith("V2") -> System.out.println("Processing V2");
        case Integer i && i > 10 && i < 100 -> System.out.println("Processing ints");
        default -> throw new IllegalArgumentException("invalid input");
    }
}
```

- The specification of additional conditions after the type check is a practical syntactical innovation to formulate queries compactly, like above the version distinction V1 and V2 or the value ranges of the integer

PART 6:
News in Java 18, 19 and 20

JEPs in Java 18, 19 und 20

Java 18 / 19 – What is included?

- JEP 400: UTF-8 by Default
- JEP 408: Simple Web Server
- **JEP 413: Code Snippets in Java API Documentation**
- JEP 416: Reimplement Core Reflection with Method Handles
- JEP 417: Vector API (Third Incubator)
- JEP 418: Internet-Address Resolution SPI
- JEP 419: Foreign Function & Memory API (Second Incubator)
- JEP 420: Pattern Matching for switch (Second Preview)
- JEP 421: Deprecate Finalization for Removal

18

- JEP 405: Record Patterns (Preview)
- JEP 422: Linux/RISC-V Port
- JEP 424: Foreign Function & Memory API (Preview)
- JEP 425: Virtual Threads (Preview)
- JEP 426: Vector API (Fourth Incubator)
- JEP 427: Pattern Matching for switch (Third Preview)
- JEP 428: Structured Concurrency (Incubator)

19

Java 19 / 20 – What is included?

- JEP 405: Record Patterns (Preview)
- JEP 422: Linux/RISC-V Port
- JEP 424: Foreign Function & Memory API (Preview)
- JEP 425: Virtual Threads (Preview)
- JEP 426: Vector API (Fourth Incubator)
- JEP 427: Pattern Matching for switch (Third Preview)
- JEP 428: Structured Concurrency (Incubator)

19

- JEP 429: Scoped Values (Incubator)
- JEP 432: Record Patterns (Second Preview)
- JEP 433: Pattern Matching for switch (Fourth Preview)
- JEP 434: Foreign Function & Memory API (Second Preview)
- JEP 436: Virtual Threads (Second Preview)
- JEP 437: Structured Concurrency (Second Incubator)
- JEP 438: Vector API (Fifth Incubator)

20

JEP 413: Code Snippets in Java API Documentation

JEP 413: Code Snippets in Java API Documentation

- **Until Java 18, there is no standard to include code fragments in the HTML documentation generated with Java Doc. In the context of this JEP, the inline tag @snippet is introduced. This allows code fragments to be included in a Java-Doc comment:**

```
/**
 * The following code shows how to use {@code Optional.isPresent}:
 * {@snippet :
 * if (optValue.isPresent())
 * {
 *     System.out.println("value: " + optValue.get());
 * }
 * }
 */
public static void method1(String[] args) {
    //
    //
    //
    //
}
```

void java18.misc.JavaDocExample.method1(String[] args)

The following code shows how to use `Optional.isPresent`:

```
if (optValue.isPresent())
{
    System.out.println("value: " + optValue.get());
}
```

JEP 413: Code Snippets in Java API Documentation

- **Highlighting**

```
/**
 * The following code shows how to use {@code Optional.isPresent} and
 * {@code Optional.get} in combination
 *
 * {@snippet :
 * if (optValue.isPresent()) // @highlight substring="isPresent"
 * {
 *     System.out.println("value: " + optValue.get()); // @highlight substring="get"
 * } }
 */
public static void newJavaDocExample(String[] args) {
    //
    //
    //
    //
}
```

void java18.misc.JavaDocExample.newJavaDocExample(String[] args)

The following code shows how to use `Optional.isPresent` and `Optional.get` in combination

```
if (optValue.isPresent())
{
    System.out.println("value: " + optValue.get());
}
```

Parameters:
args

JEP 405: Record Patterns (Preview in Java 19)

JEP 420: Record Patterns

- This JEP extends the pattern matching for `instanceof` from Java 16:

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

static void printCoordinateInfo(Object obj)
{
    if (obj instanceof Point point)
    {
        int x = point.x();
        int y = point.y();

        System.out.println("x: %d y: %d, sum: %d".formatted(x, y, x + y));
    }
}
```

- Although this is often already practical, you still have to access the individual components in some cases in a cumbersome way.
- The goal is to be able to decompose records into their components and access them.

JEP 420: Pattern Matching for `switch`

- **The goal is to be able to decompose records into their components and access them.**

```
static void printCoordinateInfo(Object obj)
{
    if (obj instanceof Point point)
    {
        int x = point.x();
        int y = point.y();
        System.out.println("x: %d y: %d, sum: %d".formatted(x, y, x + y));
    }
}
```

- **=>**

```
static void printCoordinateInfoNew(Object obj)
{
    if (obj instanceof Point(int x, int y))
    {
        System.out.println("x: %d y: %d, sum: %d".formatted(x, y, x + y));
    }
}
```

Jep405_RecordPatternsExample.java

JEP 420: Pattern Matching for `switch`

- **Record patterns can be nested to provide a declarative, powerful, and combinable form of data navigation and processing.**

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

enum Color { RED, GREEN, BLUE }

record ColoredPoint(Point p, Color c) {}
record Rectangle(ColoredPoint upperLeft, ColoredPoint lowerRight) {}

static void printColorOfUpperLeftPoint(Rectangle rect)
{
    if (rect instanceof Rectangle(ColoredPoint(Point p, Color c),
                                  ColoredPoint lr))
    {
        System.out.println(c);
    }
}
```

Jep405_InstanceofRecordMatchingAdvanced.java

DEMO & Hands on

```
Jep405_RecordPatternsExample.java
Jep405_InstanceofRecordMatchingAdvanced.java
```

JEP 405: Record Patterns

- **Record patterns make for elegance:**

```
record Person(String name, int age, Boolean hasDrivingLicense) { }

boolean isAllowedToDrive(Object obj) {
   if (obj instanceof Person person) {
       return person.age() >= 18 && person.hasDrivingLicense();
   }
   return false;
}

boolean isAllowedToDrive(Object obj) {
    if (obj instanceof Person(String name, int age, Boolean hasDrivingLicense)) {
        return age >= 18 && hasDrivingLicense;
    }
    return false;
}
```

- **However, please always have good OO design in mind!**

Jep405_DrivingLicense.java

How to make it even more elegant?

JEP 405: Record Patterns

- **Clean OO design would define the method in the record itself:**

```
record Person(String name, int age, Boolean hasDrivingLicense) {
    boolean isAllowedToDrive() {
        return age() >= 18 && hasDrivingLicense();
    }
}
```

In the previous example, accessing the attributes only serves to illustrate pattern matching.

Where can Record Patterns show their strength?

JEP 405: Record Patterns

- **Let's assume the following records as a data model:**

```
record Person(String firstname, String lastname, LocalDate birthday) {
}

record Phone(String areaCode, String number) {
}

record City(String name, String country, String languageCode) {
}

record FlightReservation(Person person,
                         Phone phoneNumber,
                         City from,
                         City destination) {
}
```

Jep405_FlighReservationExample.java

JEP 405: Record Patterns

- **Legacy code contains deeply nested queries like this:**

```
boolean checkAgeAndDestinationLanguageCorrectOld(Object obj)
{
    if (obj instanceof FlightReservation reservation)
    {
        if (reservation.person() != null)
        {
            Person person = reservation.person();

            if (reservation.destination() != null)
            {
                LocalDate birthday = person.birthday();
                String languageCode = destination.languageCode();

                if (birthday != null && languageCode != null)
                {
                    long years = ChronoUnit.YEARS.between(LocalDate.now(), birthday);
                    return years >= 18 && List.of("EN", "DE", "FR").contains(languageCode);
                }
            }
        }
    }
    return false;
}
```

Jep405_FlighReservationExample.java

JEP 405: Record Patterns

- **Using nested record patterns, we write the above nesting of ifs elegantly and much more understandably as follows:**

```
boolean checkAgeAndDestinationLanguageCorrectNew(Object obj)
{
    if (obj instanceof FlightReservation(
            Person(String firstname, String lastname, LocalDate birthday),
            Phone phoneNumber, City from,
            City(String name, String country, String languageCode)))
    {
        if (birthday != null && languageCode != null)
        {
            long years = ChronoUnit.YEARS.between(LocalDate.now(), birthday);
            return years >= 18 &&
                   List.of("EN", "DE", "FR").contains(languageCode);
        }
    }
    return false;
}
```

Jep405_FlighReservationExample.java

JEP 405: Record Patterns

```
boolean checkAgeAndDestinationLanguageCorrectNew(Object obj)
{
    if (obj instanceof FlightReservation(
            Person(String firstname, String lastname, LocalDate birthday),
            Phone phoneNumber, City from,
            City(String name, String country, String languageCode)))
    {
        if (birthday != null && languageCode != null)
        {
            long years = ChronoUnit.YEARS.between(LocalDate.now(), birthday);
            return years >= 18 &&
                   List.of("EN", "DE", "FR").contains(languageCode);
        }
    }
    return false;
}
```

- **Checking with `instanceof` automatically fails if one of the record components is `null`, i.e. here `Person` or `City` (`destination`).**
- **Only the attributes are not protected in this way and may need to be checked for null.**
- **However, if you get into the good habit of avoiding `null` as a value of parameters in calls, you can even do without it.**

JEP 405: Record Patterns Summary

In total, Java 19 provides the following three ways to perform pattern matching for records:

Pattern Matching - access via variables a

Record Pattern - decomposition into sing

Named Record Pattern - combination of 1

1) Pattern Matching –
   access via variables and methods
2) Record Pattern –
   decomposition into single component
3) Named Record Pattern –
   combination of 1) and 2)

```
record StringIntPair(String name, int value) {}

Object obj = new StringIntPair("Michael", 52);

// 1. Pattern Matching
if (obj instanceof StringIntPair pair) {
    System.out.println("object is a StringIntPair, name = " +
                       pair.name() + ", value = " + pair.value());
}

// 2. Record Pattern
if (obj instanceof StringIntPair(String name, int value)) {
    System.out.println("object is a StringIntPair, " +
                       "name = " + name + ", value = " + value);
}

// 3. Named Record Pattern
if (obj instanceof StringIntPair(String name, int value) pair) {
    System.out.println("object is a StringIntPair, name = " +
            pair.name() + ", value = " + pair.value() +
            "// name = " + name + ", value = " + value);
}
```

Jep405_RecordPatternsOverview.java

JEP 432: Record Patterns (Second Preview in Java 20)

JEP 432: Record Patterns

Java 20 introduces the following three enhancements to Record Patterns:

1. **Inference of type arguments and generic Record Patterns has been improved.**
2. **Record Patterns can now be used in for-each loops.**
3. **Named Record Patterns have been removed.**

JEP 432: Record Patterns – Inference & Generics

- **Queries in Java 19 require appropriate type specifications when generics are used in record patterns.**

```
void handleContainerOld(final Container<String> container)
{
    if (container instanceof Tuple<String>(var s1, var s2))
    {
       System.out.println("Tuple: " + s1 + ", " + s2);
    }
    else if (container instanceof Triple<String>(var s1, var s2, var s3))
    {
       System.out.println("Triple: " + s1 + ", " + s2 + ", " + s3);
    }
}

interface Container<T> {}
record Tuple<T>(T t1, T t2) implements Container<T> {}
record Triple<T>(T t1, T t2, T t3) implements Container<T> {}
```

Jep432_GenericRecordsPatternsExample.java

JEP 432: Record Patterns – Inference & Generics

- **Improved type inference allows clearer notation without type specification in <>:**

```
void handleContainerNew(Container<String> container)
{
    if (container instanceof Tuple(var s1, var s2))
    {
        System.out.println("Tuple: " + s1 + ", " + s2);
    }
    else if (container instanceof Triple(var s1, var s2, var s3))
    {
        System.out.println("Triple: " + s1 + ", " + s2 + ", " + s3);
    }
}
```

- **However, this looks a bit like the Raw Types of Collections.**

- **While the old syntax works without problems in Java 20, the new syntax produces the following error message under Java 19 «`raw deconstruction patterns are not allowed`»**

JEP 432: Record Patterns – Inference & Generics

- **Especially with nestings the old notation with type specification in <> was difficult to read:**

```
void nestedOld(Container<Tuple<String>> container)
{
    if (container instanceof Tuple<Tuple<String>>(Tuple(var s1, var s2),
                                                  Tuple(var s3, var s4)))
    {
        System.out.println("String " + String.join("/",
                                                   List.of(s1, s2, s3, s4)));
    }
    else
    {
        System.out.println(”Container " + container);
    }
}
```

Jep432_GenericRecordsPatternsExample.java

JEP 432: Record Patterns – Inference & Generics

- **Improved type inference allows clearer notation without type specification in <>:**

```
void nestedNew(Container<Tuple<String>> container)
{
    if (container instanceof Tuple(Tuple(var s1, var s2),
                                   Tuple(var s3, var s4)))
    {
        System.out.println("String " + String.join("/",
                                                   List.of(s1, s2, s3, s4)));
    }
    else
    {
       System.out.println(”Container " + container);
    }
}
```

- **Even better, in my understanding, would be the following notation:**

```
if (container instanceof Tuple<>(Tuple(var s1, var s2),
                                 Tuple(var s3, var s4)))
```

JEP 432: Record Patterns – for-each

- **Let us assume the following initial data:**

```
record City(String name, int inhabitants) {}

var cities = List.of(new City("Zürich", 400_000),
                     new City("Kiel", 265_000),
                     new City("Köln", 1_000_000),
                     new City("Berlin", 3_500_000));
```

- **Since Java 20, record patterns can be specified in a for-each loop. This makes it possible to access the attributes directly (just like with `instanceof` und `switch`):**

```
for (City(var name, var inhabitants) : cities)
{
    System.out.println(name + " has " + inhabitants + " inhabitants");
}
```

- **`null` values in the dataset throw a `MatchException`**

JEP 432: Record Patterns – Named Record Patterns

- **No more support for Named Record Patterns**

```
if (obj instanceof Person(var firstname, var lastname,
                          var dateOfBirth, var eyeColor) person)
```

- **Why? This notation leads to the "inconsistency" / ambiguity of being able to access variables in multiple ways, such as the first name as follows:**

- `person.firstname()` **– using the named variable and the accessor method.**
- `firstname` **– based on the deconstruction**

- **For more conciseness, this is no longer allowed with Java 20 and will result in a compile error. Only the following syntactically clearer variant is now supported:**

```
if (obj instanceof Person(var firstname, var lastname,
                          var dateOfBirth, var eyeColor))
```

JEP 420: Pattern Matching for `switch` (PREVIEW II)

JEP 420: Pattern Matching for `switch`

- **JEP 420 leads to two changes in the evaluation of the `cases` within `switch` during compilation:**
  - on the one hand the so-called **dominance check** changes,
  - on the other hand the **completeness analysis** was corrected.

- **However, the whole thing is again implemented in the form of a preview feature and to follow up you need to activate them appropriately, for example as follows in the JShell:**

```
michaelinden@MBP-von-Michael ~ % jshell --enable-preview
|  Welcome to JShell -- Version 18.0.1
|  For an introduction type: /help intro
```

JEP 420: Pattern Matching for `switch`: Dominance check

- **Problem area: Multiple patterns can match on one input.**

```
public static void main(String[] args) {
    multiMatch("Python");
    multiMatch(null);
}

static void multiMatch(Object obj) {
    switch (obj) {
        case null -> System.out.println("null");
        case String s && s.length() > 5 -> System.out.println(s.toUpperCase());
        case String s                   -> System.out.println(s.toLowerCase());
        case Integer i                  -> System.out.println(i * i);
        default -> {}
    }
}
```

- **The one that fits "most generally" is called the dominant pattern.**
- **In the example, the shorter pattern `String s` dominates the longer one specified before it.**

SwitchPreviewExample.java

JEP 420: Pattern Matching for `switch`: Dominance check

- **The whole thing becomes problematic when the order of the patterns is reversed:**

```
static void dominanceExample(Object obj)
{
    switch (obj)
    {
        case null -> System.out.println("null");
        case String str -> System.out.println(str.toLowerCase());
        case String str && str.length() > 5 -> System.out.println(str.strip());
        case Integer i -> System.o
        default -> { }
    }
}
```

Label is dominated by a preceding case label 'String str'

Move switch branch 'String str && str.length() > 5' before 'String st

© java.lang.String

- **The dominance check uncovers the problem and leads to a compile error since Java 18 and Java 17.0.6, because the second `case` is de facto unreachable code.**
- **With the first Java 17 versions this was not detected as error!**

JEP 420: Pattern Matching for `switch`: Dominance check

- **Problems with constants (was not detected with Java 17)**

```
static void dominaceExampleWithConstant(Object obj) {
    switch (obj.toString()) {
      case String s && s.length() > 5 -> System.out.println(s.toUpperCase());
      case "Sophie"                   -> System.out.println("My lovely daughter");
      defaul
    }
}
```

This case label is dominated by one of the preceding case label

Press 'F2' for focus

- **Correction:**

```
static void dominaceExampleWithConstant(Object obj) {
    switch (obj.toString()) {
      case "Sophie"                   -> System.out.println("My lovely daughter");
      case String s && s.length() > 5 -> System.out.println(s.toUpperCase());
      default -> System.out.println("FALLBACK");
    }
}
```

SwitchDominanceExample.java

JEP 420: Pattern Matching bei `switch`: Dominance check

- **Let's consider an example of querying various special cases of an integer:**
  - first some fixed values,
  - then the positive range of values, and
  - then the remaining remainder:

```
Integer value = 4711;

switch (value) {
    case -1, 0, 1 -> System.out.println("Handle special cases -1, 0 or 1");
    case Integer i && i > 1 ->
         System.out.println("Handle positive integer cases i > 1");
    case Integer i -> System.out.println("Handle all the remaining integers");
}
```

JEP 420: Pattern Matching for `switch`: Dominance check

```
Integer value = 4711;
switch (value) {
    case Integer i && i > 1 -> System.out.println("Handle positive integer cas
    case -1, 0, 1 -> System.out.println("Handle special cases -1, 0 or 1");
    case Integer
}
```

Label is dominated by a preceding case label 'Integer i && i > 1'

Move switch branch '1' before 'Integer i && i > 1' ⌥⇧↵ More actions... ⌥↵

```
switch (value) {
    case Integer i -> System.out.println("Handle all the remaining integers");
    case Integer i && i > 1 -> System.out.println("Handle positive integer cas
    case -1, 0, 1 -> System.out.println("Handle special cases -1, 0 or 1");
}
```

Label is dominated by a preceding case label 'Integer i'

Move switch branch '1' before 'Integer i' ⌥⇧↵ More actions... ⌥↵

SwitchDominanceExample.java

What is important to keep in mind for the dominance check?

JEP 420: Pattern Matching for switch – Special Cases I

```
static void testTriangleAndString(Object obj)
{
    switch (obj)
    {
        case Triangle t && t.calculateArea() > 7000 ->
            System.out.println("Large triangle");
        case String str && str.startsWith("INFO") && str.contains("SPECIAL") ->
            System.out.println("a very special info");
        case String str && str.startsWith("INFO") ->
            System.out.println("just an info");
        // not detected ...
        case String str && str.startsWith("INFO") && str.contains("SPECIAL") ->
            System.out.println("a very special info");
        default ->
           System.out.println("Something else: " + obj);
    }
}
```

DUPLICATES IN THE QUERY

Java 18: SwitchSpecialCasesExample.java / Java 19: SwitchWhen.java

JEP 420: Pattern Matching for switch – Special Cases II

```
static void testTriangleAndString(Object obj)
{
    switch (obj)
    {
        case Triangle t && t.calculateArea() > 7000 ->
            System.out.println("Large triangle");
        // case String str && str.startsWith("INFO") && str.contains("SPECIAL") ->
        //    System.out.println("a very special info");
        case String str && str.startsWith("INFO") ->
            System.out.println("just an info");
        // not detected ...
        case String str && str.startsWith("INFO") && str.contains("SPECIAL") ->
            System.out.println("a very special info");
        default ->
           System.out.println("Something else: " + obj);
    }
}
```

SPECIAL CASE IN THE QUERY=> DOMINANCE

JEP 420: Pattern Matching for `switch`: Completeness analysis

- bug fix in the area of completeness analysis = checking if all possible paths are covered by the `cases` in the `switch`

- In older Java version sometimes wrong error message "`switch statement does not cover all possible input values`"

```
static void performAction(BaseOp op) {
    switch (op) {
        case Add a -> System.out.println(a);
        case Sub s -> System.out.println(s);
        // default -> System.out.println("FALLBACK")
    }
}
```

```
static void performAction(BaseOp op) {
    switch (op) {
        case
        case
        // de
    }
}
```

'switch' statement does not cover all possible input values

Insert 'default' branch ⌥⇧⏎ More actions... ⌥⏎

```
@NotNull ↗
SwitchPreviewExample.BaseOp op
```

main

JEP 420: Pattern Matching for `switch`: Completeness analysis

- **Java 18 contains a bug fix in the completeness analysis area so that the following source code compiles without errors:**

```
static sealed abstract class BaseOp permits Add, Sub {
}

static final class Add extends BaseOp {
}

static final class Sub extends BaseOp {
}

static void performAction(BaseOp op) {
     switch (op) {
         case Add a -> System.out.println(a);
         case Sub s -> System.out.println(s);
     }
}
```

SwitchCompletenessExample.java

DEMO & Hands on

```
SwitchPreviewExample.java
SwitchSpecialCasesExample.java
SwitchDominanceExample.java
SwitchCompletenessExample.java
```

JEP 427: Pattern Matching for `switch` (Preview III)

JEP 427: Pattern Matching for `switch`

- **This JEP already has two predecessors that brought significant improvements to `switch`.**

- **This JEP is about a few refinements, namely the way to specify further queries, so-called guarded patterns, in `switch`.**

- **So far intuitively and as known from `if` with `&&`:**

```
case String str && str.startsWith("INFO") -> System.out.println("just an info");
```

- **Now the key word `when` has been newly introduced for this purpose.**

```
case String str when str.startsWith("INFO") -> System.out.println("just an info");
```

- **A bit special syntax:**

```
case String str when str.startsWith("INFO") && str.contains("SPECIAL") ->
        System.out.println("a very special info");
```

JEP 427: Pattern Matching for switch

```
interface Shape {}

record Rectangle() implements Shape {}
record Triangle()  implements Shape { int calculateArea() { return 7271; } }


static void testTriangleAndString(Object obj)
{
    switch (obj)
    {
        case Triangle t when t.calculateArea() > 7000 ->
                        System.out.println("Large triangle");
        case String str when str.startsWith("INFO") ->
                        System.out.println("just an info");
        default -> System.out.println("Something else: " + obj);
    }
}
```

JEP 427: Pattern Matching for switch with Record Patterns

```java
record Pos3D(int x, int y, int z) { }

enum RgbColor {RED, GREEN, BLUE}


static void recordPatternsAndMatching(Object obj) {
    switch (obj) {
        case RgbColor color when color == RgbColor.RED ->
             System.out.println("RED WARNING");

        case Pos3D pos when pos.z() == 0 ->
             System.out.println("Record: " + pos);

        case Pos3D(int x, int y, int z) when y > 0 ->
             System.out.println("Record decomposed: " + x + ", " + y + ", " + z);

        default -> System.out.println("Something else");
    }
}
```

SwitchWhen.java

DEMO & Hands on

SwitchWhen.java

JEP 433: Pattern Matching for `switch` (Fourth Preview in Java 20)

JEP 433: Pattern Matching for `switch`

JEP 433 aims to make switch more powerful and easier to read and maintain. The most important changes in Java 20 include the following two:

1. The ability to derive the type arguments for generic patterns and record patterns in `switch`.

2. A full, and exhaustive, description of all cases in a switch now throws a `MatchException`, rather than an `IncompatibleClassChangeError`, if no `case` from the `switch` applies at runtime.*

*This problem can only occur, however, if parts of the application are compiled independently and then later an enum or a class hierarchy is extended and the using class is not recompiled.

JEP 433: Pattern Matching for switch

- **With Java 19, you still had to specify the types in <> when pattern matching:**

```
record MyPair<T1, T2>(T1 first, T2 second) { }

static void recordInferenceJdk19(MyPair<String, Integer> pair)
{
    switch (pair) {
        case MyPair<String, Integer>(var text, var count)
            when text.contains("Michael") ->
                System.out.println(text + " is " + count + " years old");
        case MyPair<String, Integer>(var text, var count)
             when count > 5 && count < 10 ->
                System.out.println("repeated " + text.repeat(count));
        case MyPair<String, Integer>(var text, var count) ->
             System.out.println(text + count);
        default -> System.out.println("NOT HANDLED");
    }
}
```

Jep433_SwitchTypeInferenceExample.java

JEP 433: Pattern Matching for switch

- **Analogous to the change for instanceof, you can also dispense with the concrete type specifications for generic record patterns in switch with Java 20 (only as of JDK 20.0.1*)**

```
static void recordInferenceJdk20(MyPair<String, Integer> pair)
{
    switch (pair)
    {
       // var geht hier nicht, wenn man auf die typspezifischen
       // Methode zugreifen möchte,
       case MyPair(String text, var count) when text.contains("Michael") ->
             System.out.println(text + " is " + count + " years old");
       case MyPair(String text, Integer count) when count > 5 && count < 10 ->
             System.out.println("repeated " + text.repeat(count));
       case MyPair(var text, var count) -> System.out.println(text + count);
       default -> System.out.println("NOT HANDLED");
    }
}
```

Jep433_SwitchTypeInferenceExample.java

Exercises PART 6

https://github.com/Michaeli71/JAX-London-Best-of-Java-11-21

PART 7:
News in Java 21

Overview
JEPs in Java 21

Java 21 – What is included

- **JEP 430: String Templates (Preview)**
- **JEP 431: Sequenced Collections**
- JEP 439: Generational ZGC
- **JEP 440: Record Patterns**
- **JEP 441: Pattern Matching for switch**
- JEP 442: Foreign Function & Memory API (Third Preview)
- **JEP 443: Unnamed Patterns and Variables (Preview)**
- **JEP 444: Virtual Threads**
- **JEP 445: Unnamed Classes and Instance Main Methods (Preview)**
- JEP 446: Scoped Values (Preview)
- **JEP 448: Vector API (Sixth Incubator)**
- JEP 449: Deprecate the Windows 32-bit x86 Port for Removal
- JEP 451: Prepare to Disallow the Dynamic Loading of Agents
- JEP 452: Key Encapsulation Mechanism API
- **JEP 453: Structured Concurrency (Preview)**

Java 21

Total: 15 **Normal: 8** **Preview: 6** **Incubator: 1**

https://javaalmanac.io/

Feedback on this page?

The Java Version Almanac

Systematic collection of information about the history and the future of Java.

Details	Status	Documentation	Download	Compare API to										
Java 22	DEV	API | Notes	JDK | JRE	21	20	19	18	17	16	15	14	13	12	...
Java 21	DEV	API | Notes	JDK | JRE	20	19	18	17	16	15	14	13	12	11	...
Java 20	REL	API | Lang | VM | Notes	JDK | JRE	19	18	17	16	15	14	13	12	11	10	...
Java 19	EOL	API | Lang | VM | Notes	JDK | JRE	18	17	16	15	14	13	12	11	10	9	...
Java 18	EOL	API | Lang | VM | Notes	JDK | JRE	17	16	15	14	13	12	11	10	9	8	...
Java 17	LTS	API | Lang | VM | Notes	JDK | JRE	16	15	14	13	12	11	10	9	8	7	...
Java 16	EOL	API | Lang | VM | Notes	JDK | JRE	15	14	13	12	11	10	9	8	7	6	...
Java 15	EOL	API | Lang | VM | Notes	JDK | JRE	14	13	12	11	10	9	8	7	6	5	...
Java 14	EOL	API | Lang | VM | Notes	JDK | JRE	13	12	11	10	9	8	7	6	5	1.4	...
Java 13	EOL	API | Lang | VM | Notes	JDK | JRE	12	11	10	9	8	7	6	5	1.4	1.3	...
Java 12	EOL	API | Lang | VM | Notes	JDK | JRE	11	10	9	8	7	6	5	1.4	1.3	1.2	...
Java 11	LTS	API | Lang | VM | Notes	JDK | JRE	10	9	8	7	6	5	1.4	1.3	1.2	1.1	
Java 10	EOL	API | Lang | VM | Notes	JDK | JRE	9	8	7	6	5	1.4	1.3	1.2	1.1		
Java 9	EOL	API | Lang | VM | Notes	JDK | JRE	8	7	6	5	1.4	1.3	1.2	1.1			
Java 8	LTS	API | Lang | VM | Notes	JDK | JRE	7	6	5	1.4	1.3	1.2	1.1				
Java 7	EOL	API | Lang | VM | Notes	JDK | JRE	6	5	1.4	1.3	1.2	1.1					
Java 6	EOL	API | Lang | VM | Notes	JDK | JRE	5	1.4	1.3	1.2	1.1						
Java 5	EOL	API | Lang | VM | Notes		1.4	1.3	1.2	1.1							
Java 1.4	EOL	API		1.3	1.2	1.1								
Java 1.3	EOL	API		1.2	1.1									
Java 1.2	EOL	API | Lang		1.1										
Java 1.1	EOL	API												
Java 1.0	EOL	API | Lang | VM												
Pre 1.0	EOL													

Info – https://javaalmanac.io/

Java 21

Status	In Development
Release Date	2023-09-15
EOL Date	2028-09
Class File Version	65.0
API Changes	Compare to 20 - 19 - 18 - 17 - 16 - 15 - 14 - 13 - 12 - 11 - 10 - 9 - 8 - 7 - 6 - 5 - 1.4 - 1.3 - 1.2 - 1.1
Documentation	Release Notes, JavaDoc
SCM	git

Data Source

This will be the next LTS Release after Java 17.

New Features

JVM

- Generational ZGC (JEP 439)
- Deprecate the Windows 32-bit x86 Port for Removal (JEP 449)
- Prepare to Disallow the Dynamic Loading of Agents (JEP 451)

Language

- String Templates 1. Preview (JEP 430, Java Almanac)
- Record Patterns (JEP 440, Java Almanac)
- Pattern Matching for switch (JEP 441, Java Almanac)
- Unnamed Patterns and Variables 1. Preview (JEP 443)
- Unnamed Classes and Instance Main Methods 1. Preview (JEP 445, Java Almanac)

API

- Sequenced Collections (JEP 431)
- Foreign Function & Memory API 3. Preview (JEP 442)
- Virtual Threads (JEP 444, Java Almanac)
- Scoped Values 1. Preview (JEP 446)
- Vector API 6. Incubator (JEP 448)
- Key Encapsulation Mechanism API (JEP 452)
- Structured Concurrency 1. Preview (JEP 453)

Sandbox – https://javaalmanac.io/

Sandbox

Instantly compile and run Java 21 snippets without a local Java installation.

Java21.java ▶ Run 21+35-2513

```java
import java.lang.reflect.ClassFileFormatVersion;

public class Java21 {

    public static void main(String[] args) {
        var v = ClassFileFormatVersion.latest();
        System.out.printf("Hello Java bytecode version %s!", v.major());
    }

}
```

No Support for Preview Features!

Java21.java ▶ Run 21+35-2513

```
Hello Java bytecode version 65!
```

Sandbox – https://javaalmanac.io/

```java
public class Java21 {

    public static void main(String[] args) {
        multiMatch("Python");
        multiMatch(null);
        multiMatch(7);

        record Person(String name, int age) {}
        multiMatch(new Person("Michael", 52));
    }

    static void multiMatch(Object obj) {
        switch (obj) {
            case null -> System.out.println("null");
            case String str when str.length() > 5 -> System.out.println(str.toUpperCase());
            case String str -> System.out.println(s.toLowerCase());
            case Integer i -> System.out.println(i * i);
            default -> throw new IllegalArgumentException("Unsupported type " + obj.getClass());
        }
    }
}
```

Sandbox – https://javaalmanac.io/

Java21.java ▶ Run 21+35-2513

```java
public class Java21 {

    public static void main(String[] args) {
        multiMatch("Python");
        multiMatch(null);
        multiMatch(7);

        record Person(String name, int age) {}
        multiMatch(new Person("Michael", 52));
    }

    static void multiMatch(Object obj) {
        switch (obj) {
            case null -> System.out.println("null");
            case String str when str.length() > 5 -> System.out.println(str.toUpperCase());
            case String str  -> System.out.println(str.toLowerCase());
            case Integer i -> System.out.println(i * i);
            default -> throw new IllegalArgumentException("Unsupported type " + obj.getClass());
        }
    }
}
```

Java21.java ▶ Run 21+35-2513

```
PYTHON
null
49
Exception in thread "main" java.lang.IllegalArgumentException: Unsupported type class Java21$1Person
        at Java21.multiMatch(Java21.java:18)
        at Java21.main(Java21.java:9)
```

Java 21 – What is coming?

Regular (final) Features

- JEP 431: Sequenced Collections
- *JEP 440: Record Patterns**
- *JEP 441: Pattern Matching for switch***
- JEP 444: Virtual Threads

*only minimal changes and no support in for loop

**minimal changes, no parenthesization around record patterns (parenthesized patterns, like `if (obj instanceof (String s))`, new support for qualified enum accesses

Preview

- JEP 430: String Templates (Preview)
- JEP 443: Unnamed Patterns and Variables (Preview)
- JEP 445: Unnamed Classes and Instance Main Methods (Preview)
- JEP 453: Structured Concurrency (Preview)

Incubator

- JEP 448: Vector API (Sixth Incubator)

Regular (final) features in Java 21

JEP 431: Sequenced Collections

https://openjdk.org/jeps/431

Sequenced Collections

- **Java's Collection API is one of the oldest and well-designed APIs in JDK.**
- **Three major types: list, set, and map**
- **What is missing is something like an ordered sequence of elements**
- **What we observe that some collections have an encounter order, meaning it is defined in which order the elements are traversed**
  - From front to back, index based for lists
  - HashSet has no encounter order
  - TreeSet defines it indirectly by Comparable or passed Comparator
  - LinkedHashSet keeps the insertion order

Sequenced Collections – Motivation

- **In the past there are several ways to access the first or last element**

	First element	Last element
List	`list.get(0)`	`list.get(list.size() - 1)`
Deque	`deque.getFirst()`	`deque.getLast()`
SortedSet	`sortedSet.first()`	`sortedSet.last()`
LinkedHashSet	`linkedHashSet.iterator().next()`	`// missing`

- **Hard to remember and error prone**

- **To solve this now «Sequenced Collections» represent a collection whose elements have a defined encounter order.**

https://openjdk.org/jeps/431

Sequenced Collections – Retrofitted into existing type hierarchy

https://cr.openjdk.org/~smarks/collections/SequencedCollectionDiagram20220216.png

Sequenced Collections – Motivation

- **«Sequenced Collections» represent a collection whose elements have a defined encounter order.**

```
interface SequencedCollection<E> extends Collection<E> {
    // new method
    SequencedCollection<E> reversed();
    // methods promoted from Deque
    void addFirst(E);
    void addLast(E);
    E getFirst();
    E getLast();
    E removeFirst();
    E removeLast();
}
```

`SequencedCollection` defines the modifiying methods:

- `addFirst(E)` – inserts an element at the beginning
- `addLast(E)` – appends an element to the end
- `removeFirst()` – removes the first element and returns it
- `removeLast()` – removes the last element and returns it

For immutable collections, all four methods throw an `UnsupportedOperationException`.

- **Addtionally these «Sequenced Collections» provide methods for adding, modifying or deleting elements at the beginning or end of the collection**
- **Furthermore they allow to process the elements in the reversed oder.**

Sequenced Collections – The Magic Behind … Default Methods

```java
public interface SequencedCollection<E> extends Collection<E>
{
    SequencedCollection<E> reversed();

    default void addFirst(E e) {
        throw new UnsupportedOperationException();
    }

    default void addLast(E e) {
        throw new UnsupportedOperationException();
    }

    default E getFirst() {
        return this.iterator().next();
    }

    default E getLast() {
        return this.reversed().iterator().next();
    }

    …

    …

    default E removeFirst() {
        var it = this.iterator();
        E e = it.next();
        it.remove();
        return e;
    }

    default E removeLast() {
        var it = this.reversed().iterator();
        E e = it.next();
        it.remove();
        return e;
    }
}
```

Sequenced Collections – Sets and Maps

```
interface SequencedSet<E> extends Set<E>, SequencedCollection<E> {
    SequencedSet<E> reversed();    // covariant override
}


interface SequencedMap<K,V> extends Map<K,V> {
    // new methods
    SequencedMap<K,V> reversed();
    SequencedSet<K> sequencedKeySet();
    SequencedCollection<V> sequencedValues();
    SequencedSet<Entry<K,V>> sequencedEntrySet();
    V putFirst(K, V);
    V putLast(K, V);
    // methods promoted from NavigableMap
    Entry<K, V> firstEntry();
    Entry<K, V> lastEntry();
    Entry<K, V> pollFirstEntry();
    Entry<K, V> pollLastEntry();
}
```

`SequencedMap` API does not fit that well. It uses `NavigableMap` as a base, so instead of `getFirstEntry()` it offers `firstEntry()`, and instead of `removeLastEntry()` it defines `pollLastEntry()`. As mentioned these names are not according to `SequencedCollection`. But trying to do this would have caused `NavigableMap` to get four new methods that do the same thing as four other methods it already has.

Sequenced Collection – In Action

```java
public static void sequenceCollectionExample() {
    System.out.println("Processing letterSequence with list");
    SequencedCollection<String> letterSequence = List.of("A", "B", "C", "D", "E");
    System.out.println(letterSequence.getFirst() + " / " +
                       letterSequence.getLast());

    System.out.println("Processing letterSequence in reverse order");
    SequencedCollection<String> reversed = letterSequence.reversed();
    reversed.forEach(System.out::print);
    System.out.println();
    System.out.println("reverse order stream skip 3");
    reversed.stream().skip(3).forEach(System.out::print);
    System.out.println();
    System.out.println(reversed.getFirst() +
                       " / " +
                       reversed.getLast());
    System.out.println();
}
```

```
Processing letterSequence with list
A / E
Processing letterSequence in
reverse order
EDCBA
reverse order stream skip 3
BA
E / A
```

Sequenced Collection – In Action

```java
public static void sequenceSetExample() {
    // Plain Sets do not have encounter order ... run multiple time to see variation
    System.out.println("Processing set of letters A-D");
    Set.of("A", "B", "C", "D").forEach(System.out::print);
    System.out.println();
    System.out.println("Processing set of letters A-I");
    Set.of("A", "B", "C", "D", "E", "F", "G", "H", "I").forEach(System.out::print);
    System.out.println();

    // TreeSet has order
    System.out.println("Processing letterSequence with tree set");
    SequencedSet<String> sortedLetters = new TreeSet<>((Set.of("C", "B", "A", "D")));
    System.out.println(sortedLetters.getFirst() + " / " + sortedLetters.getLast());
    sortedLetters.reversed().forEach(System.out::print);
    System.out.println();
}
```

```
Processing set of letters A-D
DCBA
Processing set of letters A-I
IHGFEDCBA
Processing letterSequence with tree set
A / D
DCBA
```

JEP 444: Virtual Threads

https://openjdk.org/jeps/444

JEP 444: Virtual Threads

- **This JEP introduces the concept of lightweight virtual threads.**
- **Virtual threads "feel" like normal threads, but are not mapped 1:1 to operating system threads.**

Mapping virtual threads to platform threads

https://www.happycoders.eu/java/virtual-threads/

JEP 444: Virtual Threads

- This JEP introduces the concept of lightweight virtual threads that do not map directly to operating system threads.
- Better yet, existing code that uses the existing thread API can be converted to virtual threads with minimal changes.
- With virtual threads it is possible to work in the area of server programming again with a separate thread per request and to support an asynchronous programming style in a more lightweight way.
- Via factory methods like `newVirtualThreadPerTaskExecutor()` one can choose whether virtual threads or platform threads (e.g. with `Executors.newCachedThreadPool()`) should be used.

JEP 444: Virtual Threads

```java
public static void main(String[] args)
{
    System.out.println("Start");

    try (var executor = Executors.newVirtualThreadPerTaskExecutor())
    {
        for (int  i = 0; i < 10_000; i++)
        {
            final int pos = i;
            executor.submit(() -> {
                Thread.sleep(Duration.ofSeconds(1));
                return pos;
            });
        }
    }
    // executor.close() is called implicitly, and waits
    System.out.println("End");
}
```

VirtualThreadsExample.java

JEP 444: Virtual Threads

- **Performance comparison in thousandths for platform and virtual threads.**
- **The threads wait one second each to simulate access to an external interface. At the end, the total time is measured.**

https://entwickler.de/java/javaneunzehn-features-virtualthreads

DEMO & Hands on

```
PlatformThreads.java
VirtualThreads.java

$ java src/main/java/api/VirtualThreadsExample.java
```

Preview Features in Java 21

JEP 430: String Templates (Preview)

https://openjdk.org/jeps/430

String Concatenation

- **To convert strings that contain texts and variable components, there are different variants in Java, starting from simple concatenation with + up to formatted conversion.**

```
String result = "Calculation: " + x + " plus " + y + " equals " + (x + y);
System.out.println(result);

String resultSB = new StringBuilder().
                  append("Calculation: ").append(x).append(" plus ").
                  append(y).append(" equals ").append(x + y).toString();
System.out.println(resultSB);

System.out.println(String.format("Calculation: %d plus %d equals %d", x, y,
                                                                    x + y));
System.out.println("Calculation: %d plus %d equals %d".formatted(x, y, x + y));

var messageFormat = new MessageFormat(" Calculation: {0} plus {1} equals {2}");
System.out.println(messageFormat.format(new Object[] { x, y, x + y }));
```

- **All have their special strengths and especially weaknesses**

String Interpolation

- **String Interpolation or formatted strings exist in many programming languages as an alternative to string concatenation as text containing special placeholders:**

```
• Python     f“Calculation: {x} + {y} = {x + y}"
• Kotlin     "Calculation: $x + $y = ${x + y}"
• Swift:     "Calculation: \(x) + \(y) = \(x + y)"
• C#         $"Calculation: {x} + {y}= {x + y}"
```

- **String templates complement the previous variants by an elegant possibility to specify expressions which are evaluated at runtime and integrated into the string appropriately.**

```
System.out.println(STR."Calculation: \{x} plus \{y} equals \{x + y}");
```

- **STR is a so-called string processor which works in combination with placeholders given as `\{varName}`**

String Templates

- **Example**

```
String firstName = "Michael";
String lastName = "Inden";
String firstLastName = STR."\{firstName} \{lastName}";
String lastFirstName = STR."\{lastName}, \{firstName}";
System.out.println(firstLastName);
System.out.println(lastFirstName);
```

```
Michael Inden
Inden, Michael
```

- **Example 2**

```
Path filePath = Path.of("example.txt");
String infoOld = "The file " + filePath + " " +
                (filePath.toFile().exists() ? "does" : "does not" + " exist");

String infoNew = STR."The file \{filePath} " +
                 STR."\{filePath.toFile().exists() ? "does " : "does not "}" +
                 "exist";
```

```
java --enable-preview -cp target/Java21Examples-1.0.jar syntax.JEP430_StringTemplates
```

String Templates and Text Blocks

- **Example**

```
int statusCode = 201;
var msg = "CREATED";

String json = STR. """
         {
             "statusCode": \{statusCode},
             "msg": "\{msg}"
         }""";
System.out.println(json);
```

```
{
    "statusCode": 201,
    "msg": "CREATED"
}
```

```
java --enable-preview -cp target/Java21Examples-1.0.jar syntax.JEP430_StringTemplates
```

String Templates and Text Blocks

```
String title = "My First Web Page";
String text = "My Hobbies:";
var hobbies = List.of("Cycling", "Hiking",
"Shopping");
String html = STR."""
    <html>
        <head><title>\{title}</title>
        </head>
        <body>
            <p>\{text}</p>
            <ul>
                <li>\{hobbies.get(0)}</li>
                <li>\{hobbies.get(1)}</li>
                <li>\{hobbies.get(2)}</li>
            </ul>
        </body>
    </html>""";
System.out.println(html);
```

```
<html>
    <head><title>My First Web Page</title>
    </head>
    <body>
        <p>My Hobbies:</p>
        <ul>
            <li>Cycling</li>
            <li>Hiking</li>
            <li>Shopping</li>
        </ul>
    </body>
</html>
```

String Templates – Calculations

```
int x = 10, y = 20;
String calculation = STR."\{x} + \{y} = \{x + y}";
System.out.println(calculation);
```

- => 10 + 20 = 30

```
int index = 0;
String modifiedIndex = STR."\{index++}, \{index++}, \{index++}, \{index++}";
System.out.println(modifiedIndex);
```

- => 0, 1, 2, 3

```
String currentTime = STR."Current time: \{
        DateTimeFormatter.ofPattern("HH:mm").format(LocalTime.now())}";
System.out.println(currentTime);
```

- => Current time: 14:42

```
java --enable-preview -cp target/Java21Examples-1.0.jar syntax.JEP430_StringTemplates
```

String Templates – not always the best choice …

```
var sophiesBirthday = LocalDateTime.parse("2020-11-23T09:02");

var infoSophie = STR."Sophie was born on \{
        DateTimeFormatter.ofPattern("dd.MM.YYYY").format(sophiesBirthday)} at \{
        DateTimeFormatter.ofPattern("HH:mm").format(sophiesBirthday)}";
System.out.println(infoSophie);

System.out.println("Sophie was born on %s at %s".formatted(
          DateTimeFormatter.ofPattern("dd.MM.YYYY").format(sophiesBirthday),
          DateTimeFormatter.ofPattern("HH:mm").format(sophiesBirthday)));
```

- **=>**

```
Sophie was born on 23.11.2020 at 09:02
Sophie was born on 23.11.2020 at 09:02
```

String Templates – Alternative String Processors

```java
private static void alternativeStringProcessors() {
    int x = 47;
    int y = 11;
    String calculation1 = FMT."%6d\{x} + %6d\{y} = %6d\{x + y}";
    System.out.println("fmt calculation 1: " +calculation1);

    float base = 3.0f;
    float addon = 0.1415f;

    String calculation2 = FMT."%2.4f\{base} + %2.4f\{addon} = %2.4f\{base + addon}";
    System.out.println("fmt calculation 2 " + calculation2);

    String calculation3 = FMT."Math.PI * 1.000 = %4.6f\{Math.PI * 1000}";
    System.out.println("fmt calculation 3 " + calculation3);
}
```

```
fmt calculation 1:     47 +     11 =     58
fmt calculation 2: 3.0000 + 0.1415 = 3.1415
fmt calculation 3: Math.PI * 1.000 = 3141.592654
```

```
java --enable-preview -cp target/Java21Examples-1.0.jar syntax.JEP430_StringTemplates
```

String Templates – Own String Processors

```
String name = "Michael";
int age = 52;
System.out.println(STR."Hello \{name}. You are \{age} years old.");

var myProc = new MyProcessor();
System.out.println(myProc."Hello \{name}. You are \{age} years old.");
```

```
Hello Michael. You are 52 years old.

-- process() --
info fragments:[Hello , . You are ,  years old.]
info values: [Michael, 52]
info interpolate: Hello Michael. You are 52 years old.

Hello >>Michael<<. You are >>52<< years old.
```

```
java --enable-preview -cp target/Java21Examples-1.0.jar syntax.JEP430_StringTemplates_Advanced
```

String Templates – Own String Processors

```
static class MyProcessor implements StringTemplate.Processor<String,
                                    IllegalArgumentException> {
    @Override
    public String process(StringTemplate stringTemplate)
                  throws IllegalArgumentException {
        System.out.println("\n-- process() --");
        System.out.println("info fragments:" + stringTemplate.fragments());
        System.out.println("info values: " + stringTemplate.values());
        System.out.println("info interpolate: " + stringTemplate.interpolate());
        System.out.println();

        var adjustedValues = stringTemplate.values().stream().
                                          map(str -> ">>" + str + "<<").
                                          toList();

        return StringTemplate.interpolate(stringTemplate.fragments(),
                                         adjustedValues);
    }
}
```

```
java --enable-preview -cp target/Java21Examples-1.0.jar syntax.JEP430_StringTemplates_Advanced
```

JEP 443: Unnamed Patterns and Variables (Preview)

https://openjdk.org/jeps/443

JEP 443: Unnamed Patterns and Variables (Preview)

- Let's do a short recap for all that are not following the latest and greatest Java trends
- Pattern Matching and Record Patterns have evolved massively in the last Java versions

```
Object obj = new Point(23, 11);

// Pattern Matching
if (obj instanceof Point point)
{
    int x = point.x();
    int y = point.y();
    System.out.println("x: %d y: %d, sum: %d".formatted(x, y, x + y));
}

// Record Pattern
if (obj instanceof Point(int x, int y))
{
    System.out.println("x: %d y: %d, sum: %d".formatted(x, y, x + y));
}
```

```
record Point(int x, int y) { }
enum Color { RED, GREEN, BLUE }
record ColoredPoint(Point p, Color c) { }
```

~~java --enable-preview -cp target/Java21Examples-1.0.jar syntax.JEP443_UnnamedVarsAndPatterns~~
java --enable-preview --source 21 src/main/java/syntax/JEP443_UnnamedVarsAndPatterns.java

JEP 443: Unnamed Patterns and Variables (Preview)

- **What do you observe using record patterns?**

```
Point p3_4 = new Point(3, 4);
var green_p3_4 = new ColoredPoint(p3_4, Color.GREEN);

if (green_p3_4 instanceof ColoredPoint(Point point, Color color))
{
    System.out.println("x = " + point.x());
}

if (green_p3_4 instanceof ColoredPoint(Point(int x, int y),
                                       Color color))
{
    System.out.println("x = " + x);
}
```

- **Only a few parts are really of interest**

~~java --enable-preview -cp target/Java21Examples-1.0.jar syntax.JEP443_UnnamedVarsAndPatterns~~
java --enable-preview --source 21 src/main/java/syntax/JEP443_UnnamedVarsAndPatterns.java

JEP 443: Unnamed Patterns and Variables (Preview)

- **And what about similar situations in «normal» Java code:**

```
BiFunction<String, String, String> doubleFirst =
                                    (String str1, String str2) -> str1.repeat(2);


try
{
    Files.writeString(Path.of("UnnamedVars.txt"), "Underscore");
}
catch (IOException ex)
{
    // just some logging
}
```

- **Some variables are unused in the code that follows**

~~java --enable-preview -cp target/Java21Examples-1.0.jar syntax.JEP443_UnnamedVarsAndPatterns~~
java --enable-preview --source 21 src/main/java/syntax/JEP443_UnnamedVarsAndPatterns.java

JEP 443: Unnamed Patterns and Variables (Preview)

- **This JEP addresses all these cases by allowing different elements in an expression or a variable to be replaced by a single _. The goal is to mark a pattern component or variable as unusable and let the compiler prohibit that the variable is used because it is not meant to.**

```
try
{
    Files.writeString(Path.of("UnnamedVars.txt"), "Underscore");
}
catch (IOException _)
{
    // java: Ab Release 21 ist nur das Unterstrichschlüsselwort "_" zulässig, um
    // unbenannte Muster, lokale Variablen, Ausnahmeparameter oder
    // Lambda-Parameter zu deklarieren
    //_.printStackTrace();
}
```

- **It is particularly worth mentioning that a variable marked by _ can neither be read nor written, as also shown by the error message implied in the comment. *(Hint Python)**

```
~~java --enable-preview -cp target/Java21Examples-1.0.jar syntax.JEP443_UnnamedVarsAndPatterns~~
java --enable-preview --source 21 src/main/java/syntax/JEP443_UnnamedVarsAndPatterns.java
```

JEP 443: Unnamed Patterns and Variables (Preview)

- **The following three variations exist:**

  1. **unnamed variable** – allows to use _ for naming or marking unused variables

  2. **unnamed pattern variable** – allows the identifier that would normally follow the type (or var) in a record pattern to be omitted

  3. **unnamed pattern** – allows to omit the type and name of a record pattern component completely (and replace with single _)

```
~~java --enable-preview -cp target/Java21Examples-1.0.jar syntax.JEP443_UnnamedVarsAndPatterns~~
java --enable-preview --source 21 src/main/java/syntax/JEP443_UnnamedVarsAndPatterns.java
```

JEP 443: Unnamed Patterns and Variables (Preview)

- **Unnamed variable I**

```
BiFunction<String, String, String> doubleFirst =
                                    (String str1, String _) -> str1.repeat(2);

interface IntTriFunction
{
    int apply(int x, int y, int z);
}

IntTriFunction addFirstTwo = (int x, int y, int _) -> x + y;

IntTriFunction doubleSecond = (int _, int y, int _) -> y * 2;
```

- **Interestingly, multiple unnamed variables can also be used in the same scope, which (besides simple lambdas) is of interest especially for record patterns and in switch.**

~~java --enable-preview -cp target/Java21Examples-1.0.jar syntax.JEP443_UnnamedVarsAndPatterns~~
java --enable-preview --source 21 src/main/java/syntax/JEP443_UnnamedVarsAndPatterns.java

JEP 443: Unnamed Patterns and Variables (Preview)

- **Unnamed variable II**

```
try {
    Files.writeString(Path.of("UnnamedVars.txt"), "Underscore");
}
catch (IOException _)
{
    // just some logging
}

String userInput = "E605";
try {
    processInput(Integer.parseInt(userInput));
}
catch (NumberFormatException _)
{
    System.out.println("Expected number, but was: '" + userInput + "'");
}
```

~~java --enable-preview -cp target/Java21Examples-1.0.jar syntax.JEP443_UnnamedVarsAndPatterns~~
java --enable-preview --source 21 src/main/java/syntax/JEP443_UnnamedVarsAndPatterns.java

JEP 443: Unnamed Patterns and Variables (Preview)

- **Unnamed variable III – but a bit crazy? why? Let's rethink …**

```
int LIMIT = 1000;
int total = 0;
List<Order> orders = List.of(new Order("iPhone"),
                             new Order("Pizza"), new Order("Water"));

for (var _ : orders) {
    if (total < LIMIT) {
        total++;
    }
}
System.out.println("total" + total);
```

~~java --enable-preview -cp target/Java21Examples-1.0.jar syntax.JEP443_UnnamedVarsAndPatterns~~
java --enable-preview --source 21 src/main/java/syntax/JEP443_UnnamedVarsAndPatterns.java

JEP 443: Unnamed Patterns and Variables (Preview)

- **Unnamed pattern variable I**

```
if (obj instanceof Point(int x, int y))
{
    System.out.println("x: " + x);
}
```

- **=>**

```
if (obj instanceof Point(int x, int _))
{
    System.out.println("x: " + x);
}
```

- **Same applies for `case Point(int x, int _)`**

~~java --enable-preview -cp target/Java21Examples-1.0.jar syntax.JEP443_UnnamedVarsAndPatterns~~
java --enable-preview --source 21 src/main/java/syntax/JEP443_UnnamedVarsAndPatterns.java

JEP 443: Unnamed Patterns and Variables (Preview)

- **Unnamed pattern variable II**

```
if (green_p3_4 instanceof ColoredPoint(Point point, Color color))
{
    System.out.println("x = " + point.x());
}
```

- **=>**

```
if (green_p3_4 instanceof ColoredPoint(Point point, Color _))
{
    System.out.println("x = " + point.x());
}
```

- **Same applies for `case ColoredPoint(Point point, Color _)`**

~~java --enable-preview -cp target/Java21Examples-1.0.jar syntax.JEP443_UnnamedVarsAndPatterns~~
java --enable-preview --source 21 src/main/java/syntax/JEP443_UnnamedVarsAndPatterns.java

JEP 443: Unnamed Patterns and Variables (Preview)

- **Unnamed pattern variable III**

```
if (green_p3_4 instanceof ColoredPoint(Point(int x, int y), Color color))
{
    System.out.println("x = " + x);
}
```

- **=>**

```
if (cp instanceof ColoredPoint(Point(int x, var _), Color _))
{
    System.out.println(x);
}
```

- **Same applies for `case`.**

~~java --enable-preview -cp target/Java21Examples-1.0.jar syntax.JEP443_UnnamedVarsAndPatterns~~
java --enable-preview --source 21 src/main/java/syntax/JEP443_UnnamedVarsAndPatterns.java

JEP 443: Unnamed Patterns and Variables (Preview)

- **Unnamed pattern**

```
if (green_p3_4 instanceof ColoredPoint(Point(int x, int y), Color color))
{
    System.out.println("x = " + x);
}
```

- **=>**

```
if (cp instanceof ColoredPoint(Point(int x, _), _))
{
    System.out.println("x = " + x);
}
```

```
instanceof _
instanceof _(int x, int y)
```

- **Same applies for `case`.**

~~java --enable-preview -cp target/Java21Examples-1.0.jar syntax.JEP443_UnnamedVarsAndPatterns~~
java --enable-preview --source 21 src/main/java/syntax/JEP443_UnnamedVarsAndPatterns.java

JEP 445: Unnamed Classes and Instance Main Methods (Preview)

https://openjdk.org/jeps/445

JEP 445: Unnamed classes and instance `main()` method

- Maybe it's been a while since you learned Java, too.
- If you want to teach Java to novice programmers, you realize **how difficult it is to get started**.
- From the beginner's perspective Java has a **really steep learning curve**.
- It already starts with the simplest Hello-World.

```java
package preview;

public class OldStyleHelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

- Python – reduced to the essentials:

```python
print("Hello, World!")
```

You as trainer mention the following facts for beginners:

1) Forget about `package`, `public` , `class`, `static`, `void`, etc. they are not important right now …
2) Just look at the one line with the `System.out.println()`
3) Oh yes, `System.out` is an instance of a class, but even that is not important now.

Quite a lot of confusing and distracting words and concepts apart from the actual task.

JEP 445: Unnamed classes and instance `main()` method

- This JEP tries to **make it easier to get started** with Java and to **make it as comfortable as possible for smaller experiments**, especially in combination with Direct Compilation.

- **The goal is to develop Java in the direction of simpler initial learning.**
- This was not the case in the past and one always had to explain various concepts like packages, classes, arrays, visibility, which are actually only important and usefull in the context of larger programs, but confuse beginners at first.

- With increasing knowledge and a growing experience, more advanced concepts such as classes, visibility control and static components can then be introduced step by step.

- **It was important to the language architects at Oracle that no other Java dialect emerges or that it requires a separate toolchain.**

Simplification I: Instance `main()`

- Now it is allowed to define the `main()` method not **`static`** and not **`public`** as well as **without parameters**, which already results in less boilerplate code and improves the comprehensibility:

```
package preview;

class InstanceMainMethod {
    void main() {
        System.out.println("Hello, World!");
    }
}
```

- Even the package keyword and definition can be omitted:

```
class InstanceMainMethod {
    void main() {
        System.out.println("Hello, World!");
    }
}
```

Simplification I: Instance `main()`

- As before, these classes can be compiled and started like a normal Java program, with Direct Compilation even analogous to Python's script-based execution:

```
$ java --enable-preview --source 21 src/main/java/prevuew/InstanceMainMethod.java
Hello, World!
```

- Conveniently, this means you don't have to introduce visibility modifiers or static elements to write a small Java program. Moreover, it is not necessary to go into detail about passing a parameter as well as its array type.

- A good first step, but it goes even further and gets both better and shorter

Simplification II: Unnamed class

- If a class defines only simple methods, as is the case here, the new feature of the unnamed classes allows to omit even the `class` definition. Now we have almost as short a program as with the one-liner in Python:

```
void main() {
    System.out.println("Hello, World!");
}
```

- The resulting Unnamed Class (obviously) has no class definition but can specify attributes and methods. Moreover, it is automatically assigned in an Unnamed Package.

- Run with (if filename is `SimplerHelloWorld.java`)

```
$ java --enable-preview --source 21 SimplerHelloWorld.java
Hello, World!
```

JEP 445: Unnamed classes and instance main() method

- **PAST**

```
public class InstanceMainMethodOld {
    public static void main(final String[] args) {
        System.out.println("Hello, World!");
    }
}
```

- **PRESENT**

```
class InstanceMainMethod {
    void main() {
        System.out.println("Hello, World!");
    }
}
```

- **PRESENT OPTIMIZED**

```
void main() {
    System.out.println("Hello World!");
}
```

Further possibilities

```
String greeting = "Hello again!!";

String enhancer(String input, int times)
{
   return " ---> " + input.repeat(times) + " <---";
}

void main()
{
    System.out.println("Hello World!");
    System.out.println(greeting);
    System.out.println(enhancer("Michael", 2));
}
```

```
$ java --enable-preview --source 21\
src/main/java/preview/UnnamedClassesMoreFeatures.java
Hello, World!
Hello again!
 ---> MichaelMichael <---
```

Conclusion: Java on the way to script-based execution

- These new features make it much easier to get started with Java.
- Furthermore, if you want to create smaller tools, which can be executed via Direct Compilation without compiling first, then the effort can be reduced to a minimum.
- So not only **beginners benefit**, but also **old hands**, but only when it comes to small programs.

- In fact, one can go further and possibly identify the `System.out.println()` as potential for simplification, more general its about console output as well as input.
- This is at least being thought about at Oracle. Thereby something can be learned from Python with the built-in functions `print()` and `input()`.

JEP 445: Unnamed classes and instance `main()` method

- **Execution order** -- The functionality implemented with this JEP simplifies a lot. To find out how the starting point this procedure is used:
  1. `static void main(String[] args)`
  2. `static void main()` without parameters
  3. `void main(String[] args)`
  4. `void main()` without parameters

- **Multiple Mains – guess which one gets executed ☺**

```
public class MultipleMains {
    protected static void main() {
        System.out.println("protected static void main()");
    }

    public void main(String[] args) {
        System.out.println("public void main(String[] args)");
    }
}
```

```
$ java --enable-preview --source 21 src/main/java/preview/MultipleMains.java
```

JEP 453: Structured Concurrency

JEP 428: Structured Concurrency

- This JEP brings Structured Concurrency as a simplification of multithreading.

- Here, different tasks that are executed in multiple threads are considered as a single unit. This improves reliability, reduces the risk for errors and simplifies their handling.

- Let's consider determining a user and their orders based on a user ID:

```
static Response handleSynchronously(Long userId) throws InterruptedException
{
    var user = findUser(userId);
    var orders = fetchOrders(userId);

    return new Response(user, orders
}
```

- Both actions could run in parallel.

JEP 428: Structured Concurrency

- **Traditional implementation with `ExecutorService`:**

```
static Response handleOldStyle(Long userId) throws ExecutionException,
                                                   InterruptedException
{
    var executorService = Executors.newCachedThreadPool();

    var userFuture = executorService.submit(() -> findUser(userId));
    var ordersFuture = executorService.submit(() -> fetchOrders(userId));

    var user = userFuture.get();        // Join findUser
    var orders = ordersFuture.get();    // Join fetchOrders

    return new Response(user, orders);
}
```

- **We pass the two subtasks to the executor and wait for the partial results. This happy path is implemented quickly.**

JEP 453: Structured Concurrency

```
static Response handleOldStyle(Long userId) throws ExecutionException,
                                                   InterruptedException
{
    var executorService = Executors.newCachedThreadPool();

    var userFuture = executorService.submit(() -> findUser(userId));
    var ordersFuture = executorService.submit(() -> fetchOrders(userId));

    var user = userFuture.get();        // Join findUser
    var orders = ordersFuture.get();    // Join fetchOrders

    return new Response(user, orders):
}
```

- **Because the subtasks are executed in parallel, they can succeed or fail independently. Then the handling can become quite complicated.**
- **Often, for example, one does not want the second `get()` to be called if an exception has already occurred during the processing of the `findUser()` method.**

JEP 453: Structured Concurrency

- **Because the subtasks are executed in parallel, they can succeed or fail independently. In such cases, the handling can become quite complicated.**

- **General question: How do we handle exceptions?**

  - Specifically, if an exception occurs in one subtask, how can we cancel the others?
  - How can we stop the processing of the subtasks altogether, for example, when we no longer need the results?

- **We can achieve this with a few tricks, but the source code then becomes complex, contains various queries and becomes difficult to understand and maintain overall.**

JEP 428: Structured Concurrency

- **Implementation of Structurd Concurrency with class `StructuredTaskScope`:**

```
static Response handle(Long userId) throws ExecutionException,
                                           InterruptedException
{
    try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
        var userSubtask  = scope.fork(() -> findUser(userId));
        var ordersSubtask = scope.fork(() -> fetchOrder(userId));

        scope.join();           // Join both forks
        scope.throwIfFailed(); // ... and propagate errors

        // Here, both forks have succeeded, so compose their results
        return new Response(userSubtask.get(), ordersSubtask.get());
    }
}
```

- **With structured concurrency, one splits off competing subtasks with `fork()`.**
- **The results are collected with a blocking call to `join()`, which waits until all subtasks are processed or an error occurred.**

JEP 453: Structured Concurrency

The `StructuredTaskScope` class has two specializations:

- `ShutdownOnSuccess` – determines the first incoming result and then terminates the StructuredTaskScope. This helps when the result of any subtask is sufficient already ("invoke any") and it is not necessary to wait for the results of other uncompleted tasks.

- `ShutdownOnFailure` – catches the first exception and terminates the StructuredTaskScope. This class is intended when results of all subtasks are needed ("invoke all"); if one subtask fails, the results of the other uncompleted subtasks are no longer needed.

JEP 453: Structured Concurrency

Advantages of using the `StructuredTaskScope` class:

- Task and Subtasks form a self-contained unit.
- Neither `ExecutorService` nor threads from a thread pool are used. Each subtask is executed in a new virtual thread.
- Error in one of the subtasks => all other subtasks are aborted.
- If the calling thread is aborted, the subtasks are also aborted.
- Call hierarchy (calling thread and subtasks) well visible in the thread dump.
- Using `ExecutorService`, you can see only thread names like "pool-X-thread-Y" in the thread dump. The assignment of pool thread to calling thread for subtasks is hardly possible.

DEMO & Hands on

`StructuredConcurrency.java`

Incubator Features in Java 21

JEP 448: Vector API
(Sixth Incubator in Java 21)

https://openjdk.org/jeps/448

https://www.sciencedirect.com/topics/computer-science/single-instruction-multiple-data

JEP 417: Vector API

- This JEP has nothing to do with the class `java.util.Vector`! Rather, it is about a platform-independent support of so-called vector calculations.

- Modern processors are able, for example, to perform an addition or multiplication not only for two values, but for a large number of values. This is also known as Single Instruction Multiple Data (SIMD). The following graphic visualizes the basic principle:

https://medium.com/@Styp/auto-vectorization-how-to-get-beaten-by-compiler-optimization-java-jit-vector-api-92c72b97fba3

JEP 417: Vector API

- The Vector API aims to improve the performance of vector calculations.
- A vector calculation consists of a sequence of operations with vectors. You can think of a vector as an array of primitive values.
- If you wanted to combine two vectors or arrays with a mathematical operation, you would use a loop over all values.

```
int[] a = { 1, 2, 3, 4, 5, 6, 7, 8 };
int[] b = { 1, 2, 3, 4, 5, 6, 7, 8 };

var c = new int[a.length];
for (int i = 0; i < a.length; i++)
{
     c[i] = a[i] + b[i];
}
```

JEP 417: Vector API

- **With the Vector API it is possible to exploit the special features and optimizations in modern processors. Therefore one provides certain help for the computations.**

```
int[] a = { 1, 2, 3, 4, 5, 6, 7, 8 };
int[] b = { 1, 2, 3, 4, 5, 6, 7, 8 };

var c = new int[a.length];
var vectorA = IntVector.fromArray(IntVector.SPECIES_256, a, 0);
var vectorB = IntVector.fromArray(IntVector.SPECIES_256, b, 0);
var vectorC = vectorA.add(vectorB);
// var vectorC = vectorA.mul(vectorB);
vectorC.intoArray(c, 0);
```

- **The controlling factor here is the size of the vector, which we set to 256 bits by the argument `IntVector.SPECIES_256`.**

- **After that the appropriate action here can be `add()` or `mul()` or others.**

JEP 417: Vector API

- **Let's consider the example from JEP 417 and the scalar calculation as a starting point:**

```
void scalarComputation(float[] a, float[] b, float[] c)
{
    for (int i = 0; i < a.length; i++)
    {
        c[i] = (a[i] * a[i] + b[i] * b[i]) * -1.0f;
    }
}
```

- **The algorithm is absolutely comprehensible and easy to follow.**

JEP 417: Vector API

```
static final VectorSpecies<Float> SPECIES = FloatVector.SPECIES_PREFERRED;

void vectorComputation(float[] a, float[] b, float[] c)
{
    int i = 0;
    int upperBound = SPECIES.loopBound(a.length);
    for (; i < upperBound; i += SPECIES.length())
    {
        var va = FloatVector.fromArray(SPECIES, a, i);
        var vb = FloatVector.fromArray(SPECIES, b, i);
        var vc = va.mul(va)
                   .add(vb.mul(vb))
                   .neg();
        vc.intoArray(c, i);
    }

    for (; i < a.length; i++)
    {
        c[i] = (a[i] * a[i] + b[i] * b[i]) * -1.0f;
    }
}
```

- If the initial data does not fit perfectly and is larger, the actions must be divided to fit.

https://medium.com/@Styp/java-18-vector-api-do-we-get-free-speed -up-c4510eda50d2

https://openjdk.org/jeps/417

JEP 417: Vector API Benchmarking

Method	Peak Speed-Up
No Unroll, No Vector	1.00x
With Unroll, No Vector	1.22x
No Unroll, With Vector	1.01x
With Unroll, With Vector	2.15x
Java Default	2.06x
Vector Implementation	2.08x

https://medium.com/@Styp/auto-vectorization-how-to-get-beaten-by-compiler-optimization-java-jit-vector-api-92c72b97fba3

JEP 448: Vector API (in Console and JShell)

```
$ java --enable-preview --source 21 --add-modules jdk.incubator.vector \
       src/main/java/api/VectorApiExample.java

WARNING: Using incubator modules: jdk.incubator.vector
[2, 4, 6, 8, 10, 12, 14, 16]
```

```
$ jshell --enable-preview --add-modules jdk.incubator.vector
|  Willkommen bei JShell - Version 21
|  Geben Sie für eine Einführung Folgendes ein: /help intro

jshell> import jdk.incubator.vector.FloatVector;

jshell> import jdk.incubator.vector.IntVector;

jshell> import jdk.incubator.vector.VectorSpecies;
```

Exercises PART 7

https://github.com/Michaeli71/JAX-London-Best-of-Java-11-21

Conclusion

Positive things

- Reliable 6-month release cadence and LTS versi
- Java becomes easier and more attractive
- Many nice improvements in syntax and APIs like switch, records, text blocks, …
- Pattern Matching and record patterns
- JPackage
- HTTP 2 (Java 11)
- Virtual Threads & Structured Concurrency

On the negative side

- Next releases were on time, but sometimes bringing just a few new features (execpt Java 14), even only preview features.
- Java 21 LTS contains lots of unfinished things …
  in my opinion LTS should contain only few preview and incubators, ideally none
- We have to wait 2 more years to have the nice unnamed classes and vars accessible for stable use
- Why is the syntax of pattern matching inconsistent for `instanceof` and `switch`?

Help

HELP
IS ON THE
WAY

Michael Inden
Java
Die Neuerungen in
Version 17 LTS, 18 und 19
dpunkt.verlag

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

Java Challenges
100+ Proven Tasks that Will Prepare
You for Anything
—
Michael Inden
Apress

Questions?

Thank You