



Cool New Java Features – from Java 17 to 23

<https://github.com/Michaeli71/JAX-London-Best-of-Java-17-23>



Michael Inden

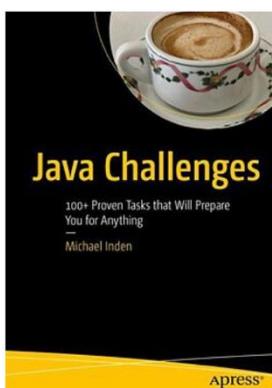
Head of Development, Independent SW consultant and author

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





Agenda

Workshop Contents



- Preliminary notes / Build Tools & IDEs
- **PART 1:** Syntax Enhancements up to Java 17 LTS
- **PART 2:** News and API-/JVM-Changes up to Java 17 LTS
- **PART 3:** News in Java 18 to 21 LTS
- **PART 4:** News in Java 22 and 23
- **Separate:** Java Modularization

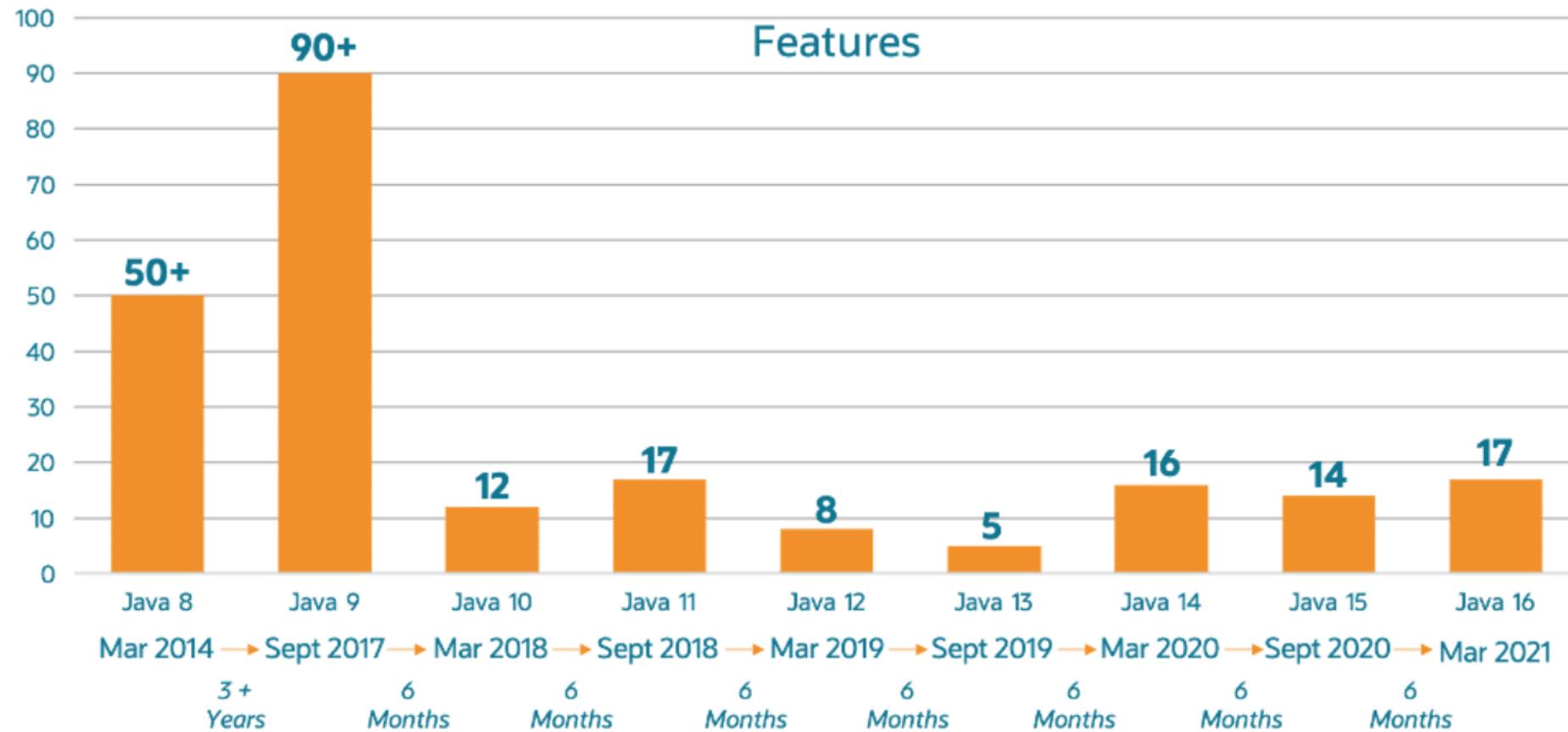


Long-Term Support Model



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





Build-Tools and IDEs



Latest Java 21 LTS installed



- **Download latest version of Java 21 LTS**

```
$ java --version  
java 21.0.4 2024-07-16 LTS  
Java(TM) SE Runtime Environment (build 21.0.4+8-LTS-274)  
Java HotSpot(TM) 64-Bit Server VM (build 21.0.4+8-LTS-274, 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 LTS



- Eclipse: Version 2023-12 (not all previews supported)
- IntelliJ: Version 2023.2.2
- Maven: 3.9.6, Compiler Plugin: 3.11.0
- Gradle: 8.5
- Activation of Preview Features / Incubator necessary
 - In Dialogs
 - In Build Scripts



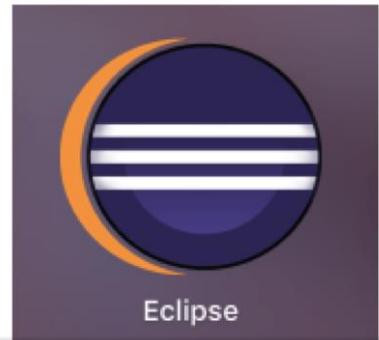
Maven™

 **Gradle**

IDE & Tool Support Java 21 LTS



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



Eclipse Marketplace

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

Search Recent Popular Favorites Installed Research at the top

Find:

Java 21 Support for Eclipse 2023-09 (4.1.0)
This marketplace solution provides Java 21 support... [more info](#)
by [Eclipse Foundation](#), EPL 2.0

0 installs: 209 (209 last month)

Eclipse Marketplace

type filter text

- > Resource Builders Coverage Java Build Path
- > Java Code Style
- > **Java Compiler**
- Javadoc Location
- > Java Editor Project Facets Project Natures Project References Refactoring History Run/Debug Settings
- > Task Repository Task Tags Validation WikiText

Java Compiler

Enable project specific settings [Configure Workspace Settings...](#)

JDK Compliance

Use compliance from execution environment 'JavaSE-20' on the [Java Build Path](#)

Compiler compliance level:

Use '--release' option

Use default compliance settings

Enable preview features for Java 21

Preview features with severity level:

Generated .class files compatibility:

Source compatibility:

Disallow identifiers called 'assert':

Disallow identifiers called 'enum':

IDE & Tool Support



- Activation of Preview Features / Incubator necessary



The screenshot shows the IntelliJ IDEA Project Structure dialog. The left sidebar has 'Project Settings' expanded, with 'Project' selected. The main area is titled 'Project' and contains the following fields:

- Name: Java21Examples (highlighted with a red arrow)
- SDK: 21 java version "21" (highlighted with a red arrow)
- Language level: 21 (Preview) - String templates, unnamed classes and instance main methods etc.
- Compiler output: (with a note below: Used for module subdirectories, Production and Test directories for the corresponding sources.)

IDE & Tool Support Java 21 LTS



- Activation of Preview Features / Incubator necessary

```
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"]  
}
```





- Activation of Preview Features / Incubator necessary

```
<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>
```





PART 1: Syntax Enhancements up to Java 17 LTS

- var
- Switch Expressions
- Text Blocks
- Records
- Syntaxerweiterung bei instanceof
- Sealed Types



var



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.

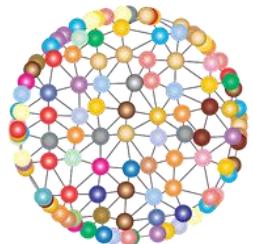
```
// var => Set<Map.Entry<String, Long>>
var entries = personAgeMapping.entrySet();
```

```
// 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

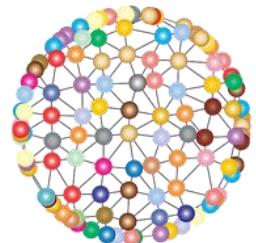


- Sometimes you may be tempted to just change the type with var:

```
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);
```



Switch Expressions



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



- Assume 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;
    };
    throw new IllegalArgumentException("Unexpected color: " + color);
    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



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>  
""";
```

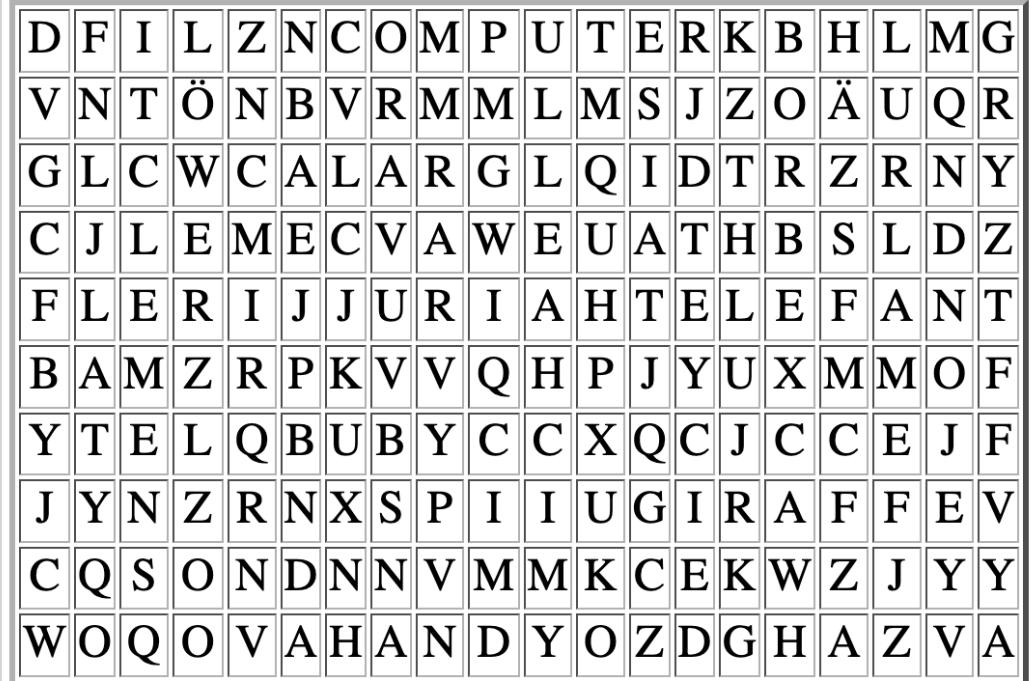
Text Blocks



```
public String exportAsHtml()
{
    String result = """
        <html>
            <head>
                <style>
                    td {
                        font-size: 18pt;
                    }
                </style>
            </head>
            <body>
"""
    result += createTable();
    result += createWordList();

    result += """
        </body>
    </html>
"""
}

return result;
```



- LÖWE
- COMPUTER
- BÄR
- GIRAFFE
- HANDY
- CLEMENS
- ELEFANT
- MICHAEL
- TIM

Text Blocks



- NEU

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

```
String multiLineStringWithPlaceHolders = """
    SELECT %s
    FROM %
    WHERE %
""".formatted("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 = """"
```

```
...>
```

```
...>
```

```
...>
```

```
multiLine ==> "One\n Two\n Three"
```

- No line break at the end
- Spaces before first character NOT taken into account
- Upfront spaces get removed

```
jshell> var multiLine = """"
```

```
...>
```

```
...>
```

```
...>
```

```
...>
```

```
...>
```

```
multiLine ==> "_ One\n _ Two\n _ Three\n ___ Four\n"
```

- Line break at the end
- spaces before first character taken into account
- Upfront spaces get removed up to first character

- **All trailing spaces get removed automatically**

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, concise 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) {}
```



What you get

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

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

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

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

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

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

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

    // Zugriffsmethoden auf x und y
}
```

Records additional constructors and methods



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

jshell> System.out.println(topLeft.shortForm());
[23, 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 Tuples



```
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



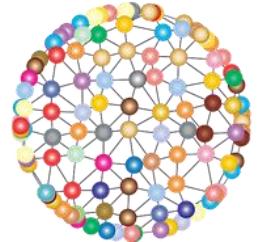
```
record ClosedInterval(int lower, int upper)
{
    public ClosedInterval(int lower, int upper)
    {
        if (lower > upper)
        {
            String errorMsg = String.format("invalid: %d (lower) >= %d (upper)",
                lower, upper);
            throw new IllegalArgumentException(errorMsg);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

        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:

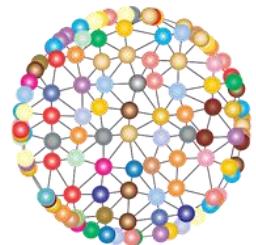
```
record SimplePerson(String name, int age, String city) {}
```

```
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\$1 SimplePerson 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



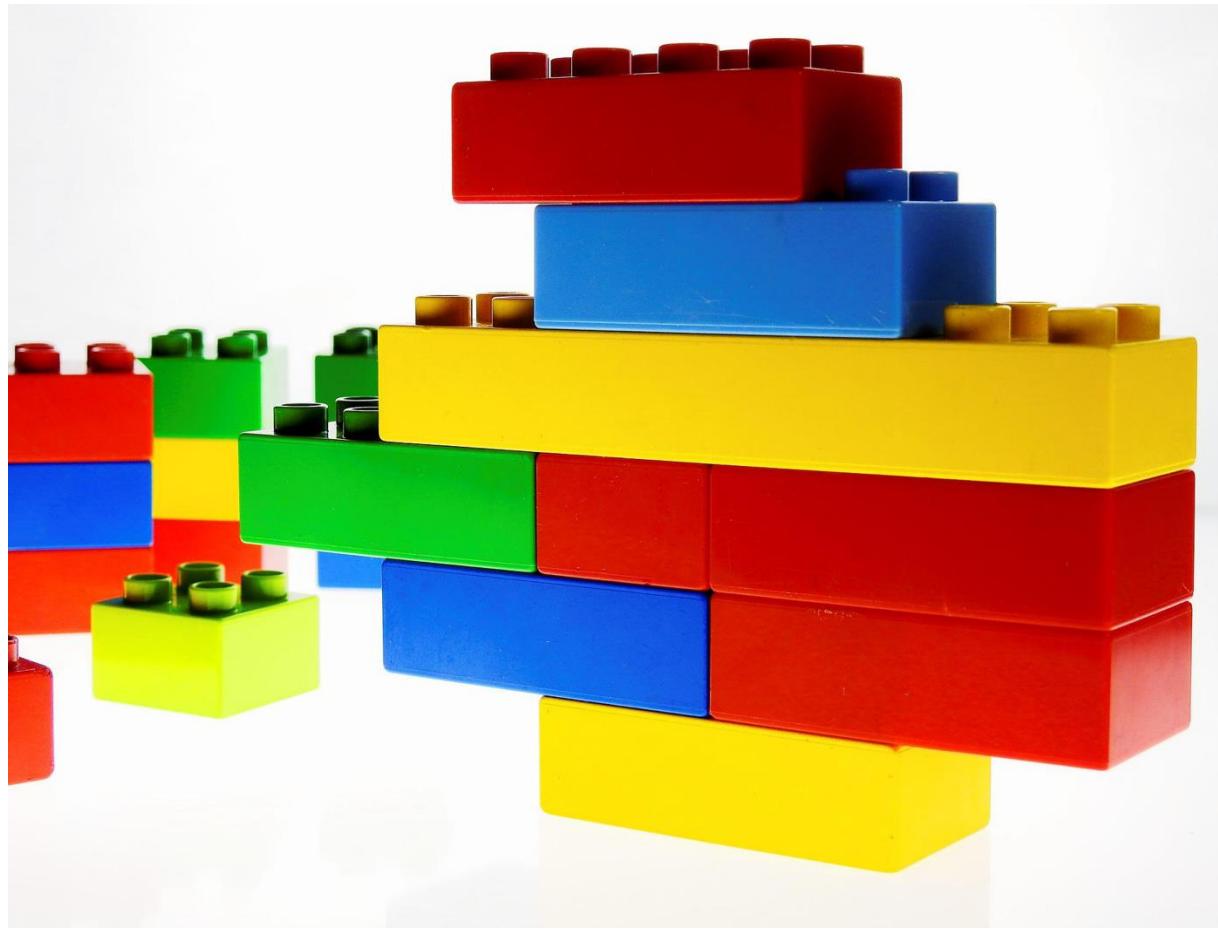
- Let's correct the definition of the comparator:

```
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);
    }
}
```

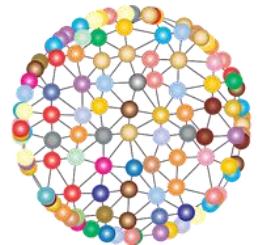
Builder like



- 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

A large, stylized text "12.12" is displayed in a bold, white font with a thick red outline. The text is centered on a solid black rectangular background.

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)
```

Records



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());
}
```



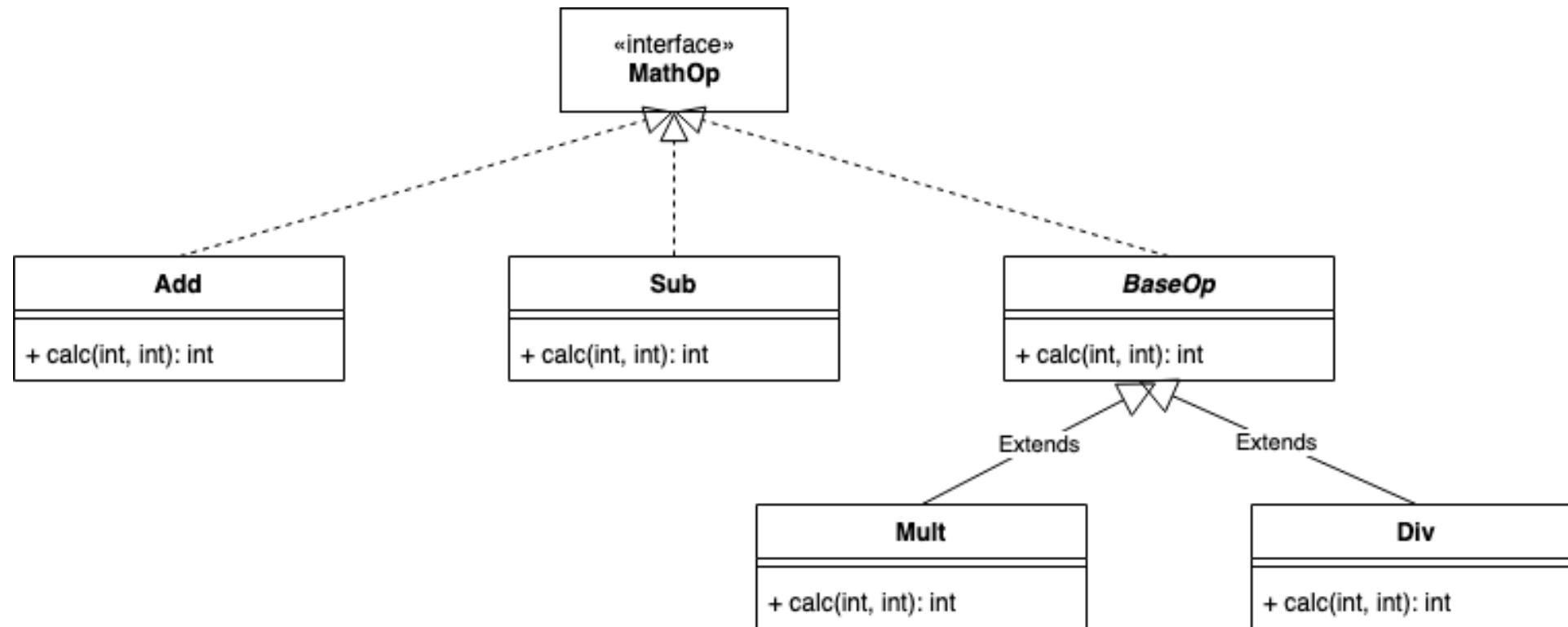
Sealed Types



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.

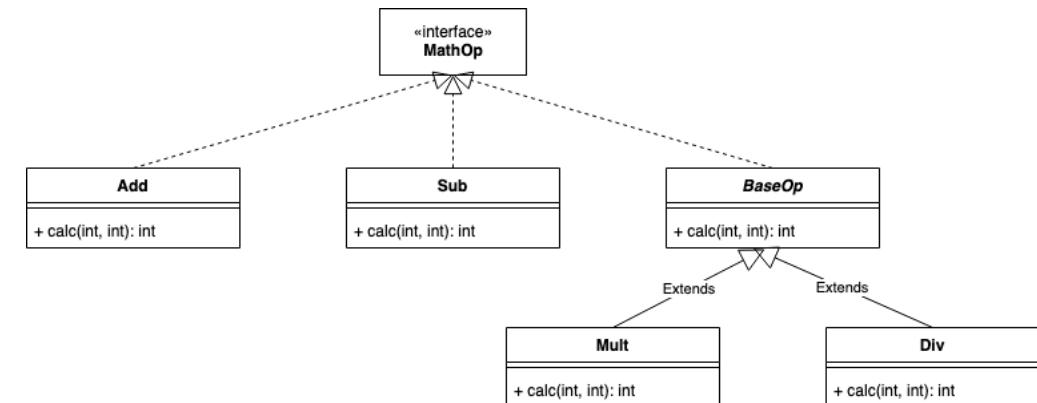
```
public class SealedTypesExamples
{
    sealed interface MathOp
        permits BaseOp, Add, Sub // <= permitted subtypes
    {
        int calc(int x, int y);
    }
}
```

// 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



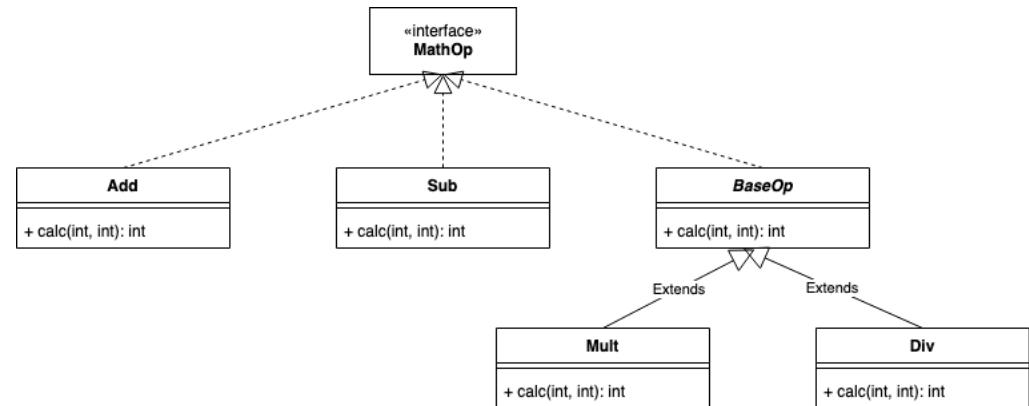
```
..  
// direct implementation must be final  
final class Add implements MathOp  
{  
    @Override  
    public int calc(int x, int y)  
    {  
        return x + y;  
    }  
}  
final class Sub implements MathOp  
{  
    ...  
}  
// implementation of base SHOULD be final  
final class Mult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

Same applies for Interfaces



Things to know for Sealed Types

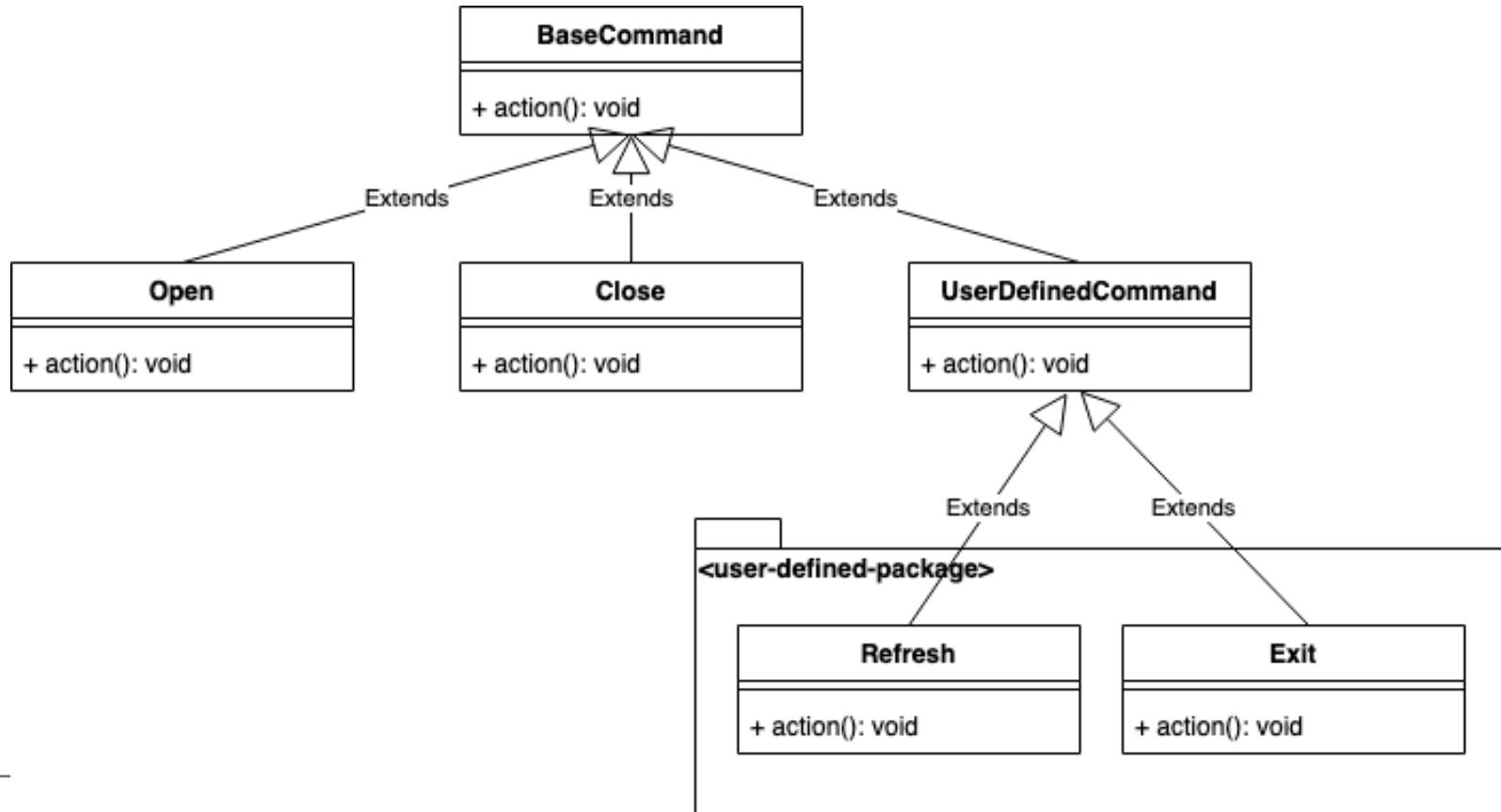


- specify which other classes or interfaces may subtype from it.
- Sealed types can expose behavior over interfaces in the development of libraries, but allow control over possible implementations.
- Sealed types restrict with regard to the extensibility of class hierarchies and should therefore be used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

Things to know about Sealed Types



- Sealed types can help when developing libraries: Expose behavior via interfaces, but retain control over possible implementations
- Extension Points





Exercises PART 1

<https://github.com/Michaeli71/JAX-London-Best-of-Java-17-23>



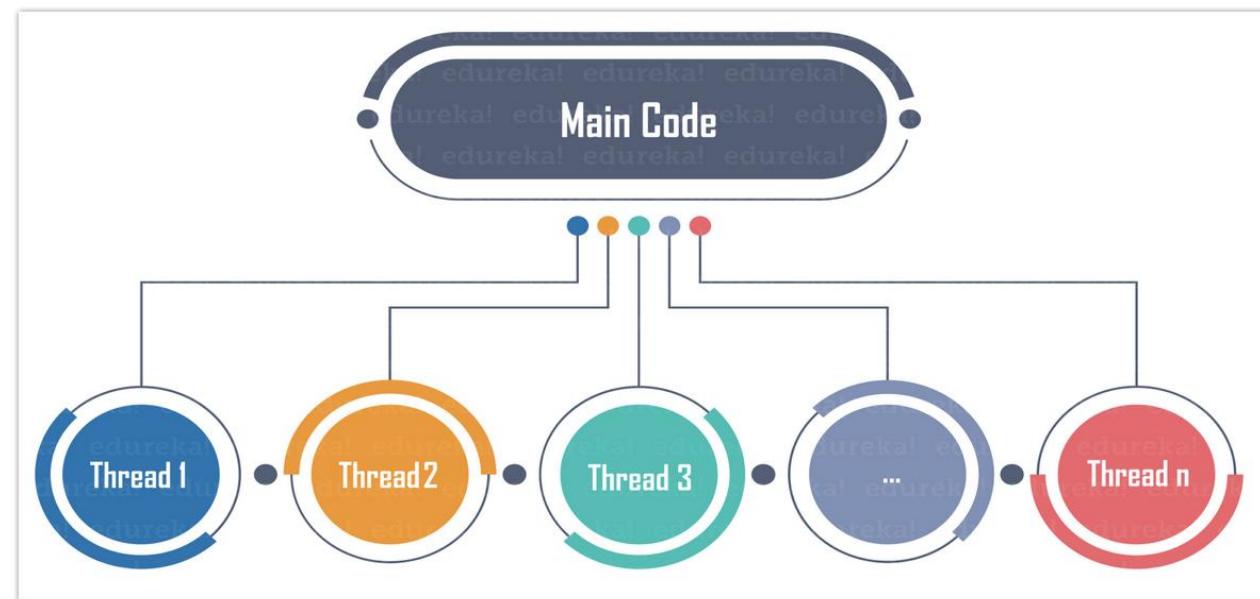


PART 2: News and API-/JVM-Changes up to Java 17 LTS

- Multi-Threading and CompletableFuture
- HTTP/2
- Stream.toList()
- Direct Compilation
- Helpful NullPointerExceptions
- JPackage



Multi-Threading with CompletableFuture



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());
```

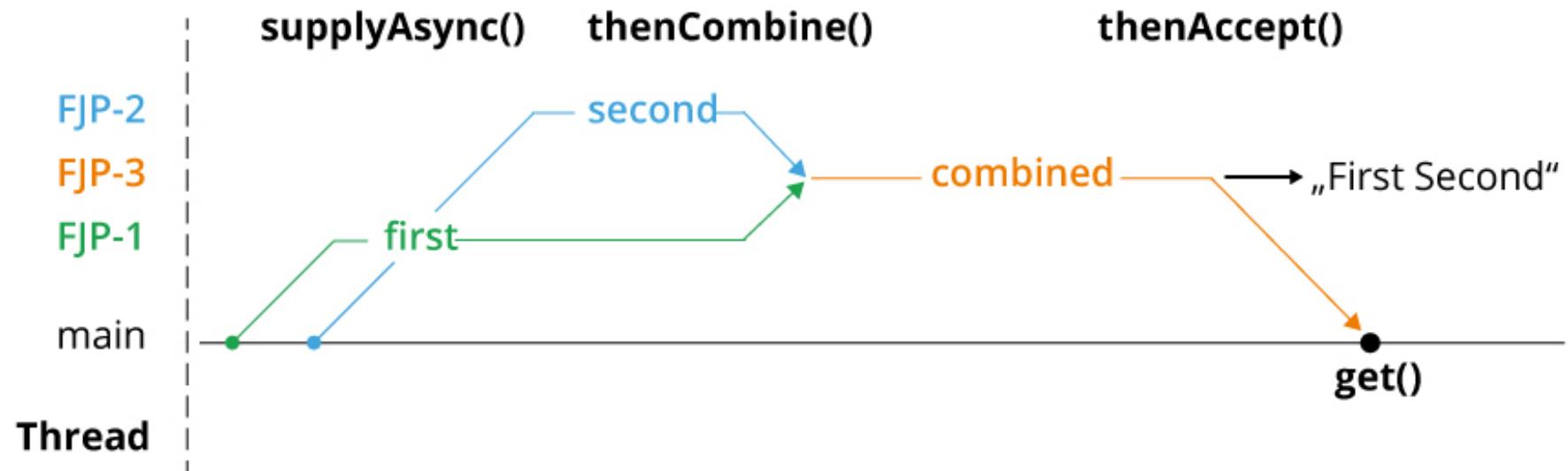
Introduction to CompletableFuture<T>



```
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>

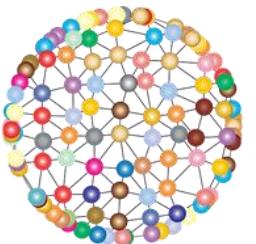


Example: The following actions are to take place:

- 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>

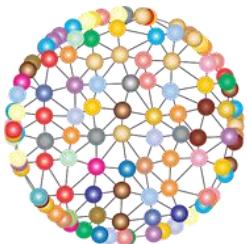


```
public void execute()
{
    final List<String> data = retrieveData();
    final Long value1 = processData1(data);
    final Long value2 = processData2(data);
    final Long value3 = processData3(data);
    final String result = calcResult(value1, value2, value3);
    System.out.println("result: "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

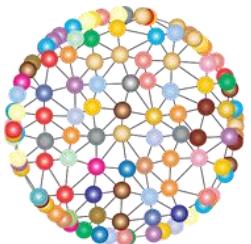
Multi-Threading and the Class CompletableFuture<T>



```
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()**

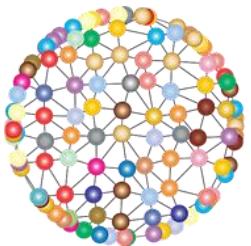
- Provide a fallback value if the data cannot be determined:

```
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);

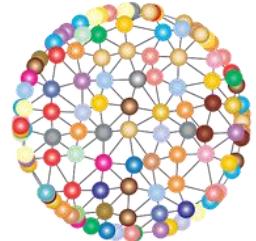
printResponseInfo(response);

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");
}
```

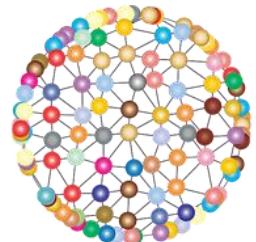
HTTP/2 API Async I – busy waiting with premature termination



```
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);
```



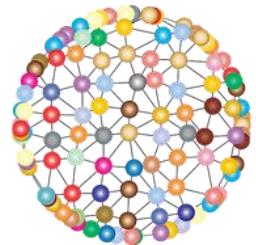
Java 16



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



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



**How often did you called
toArray() and how often
Collectors.toList()?**

1 : 1000 ?

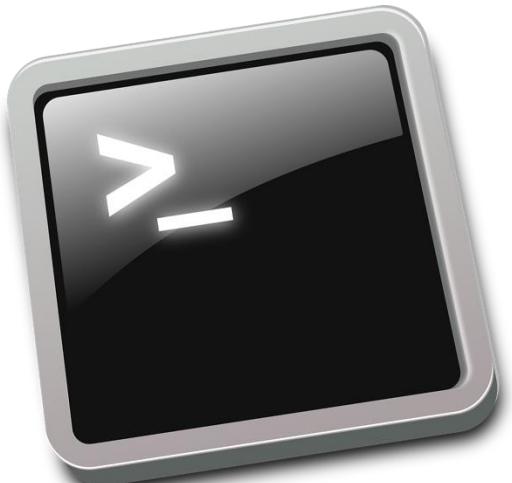
FINALLY... `toList()`



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



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

```
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)**

Direct Compilation – Shebang Execution with external dependency



```
#!/usr/bin/java --source 11 -cp ./guava-32.1.3-jre.jar
```

```
import java.nio.file.*;
import com.google.common.base.Joiner;

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

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



DEMO



N P E

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(); }
}
```

Don't do this in production!

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) {}
```



JPackage





▼ PackagingDemo

► JRE System Library [JavaSE-16]

▼ src/main/java

 ▼ de.java17

 ▼ ApplicationExample.java

 ▼ ApplicationExample

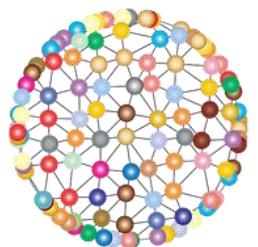
 main(String[]) : void

► src

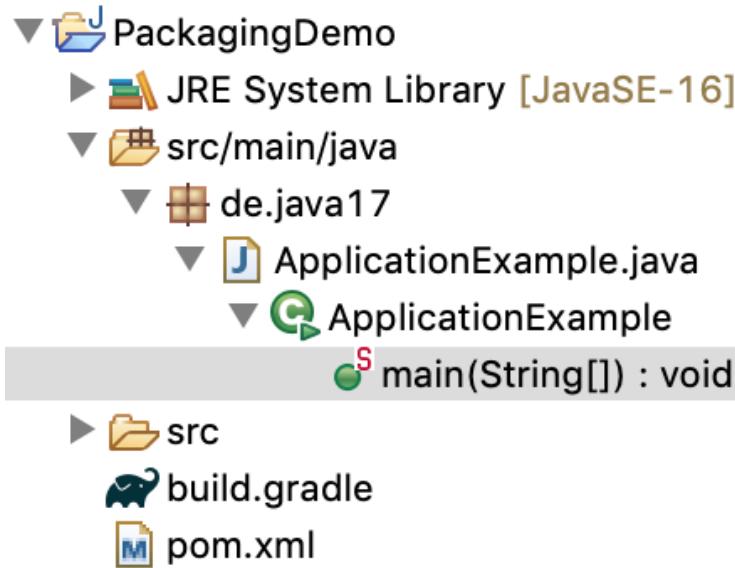
 build.gradle

 pom.xml

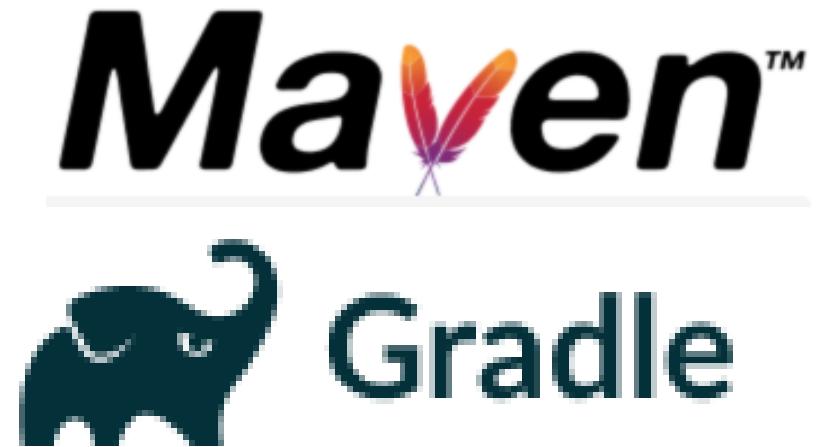
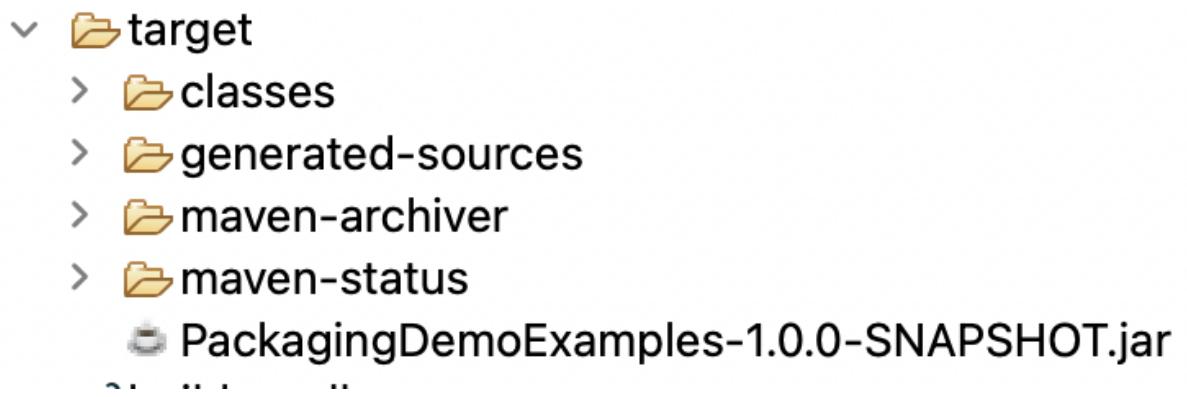
```
public class ApplicationExample {  
    public static void main(String[] args) {  
        System.out.println("First JPackage");  
  
        JOptionPane.showConfirmDialog(null, "Generated by jpackage", "DEMO",  
            JOptionPane.OK_CANCEL_OPTION, JOptionPane.INFORMATION_MESSAGE, null)  
    }  
}
```



How to build?

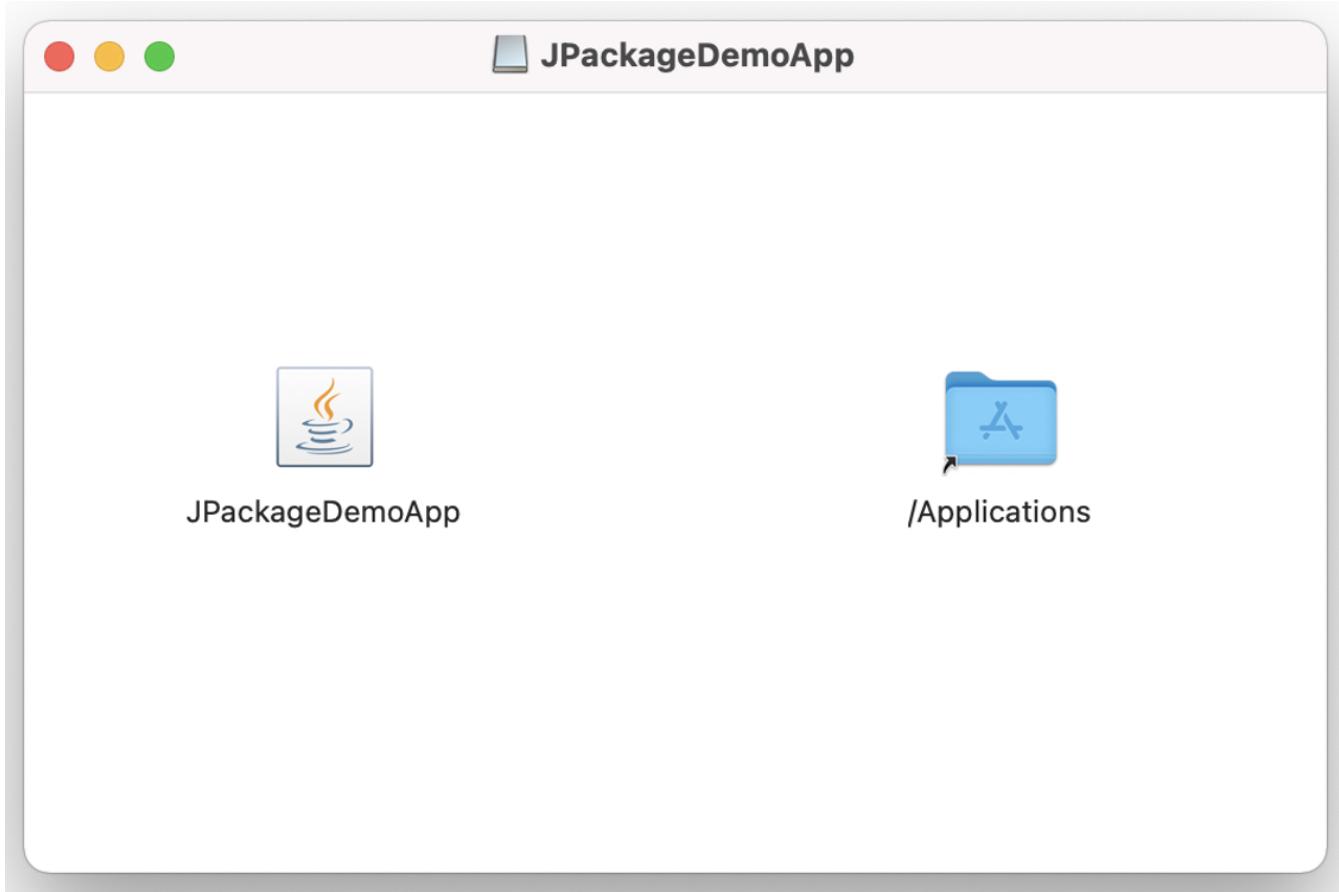


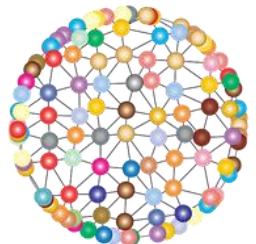
mvn clean install





```
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?



```
public class ApplicationExample {

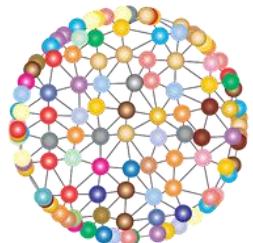
    public static void main(String[] args) {
        System.out.println("First JPackage");

        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>33.1.0-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>compile</includeScope>
        <excludeScope>test</excludeScope>
      </configuration>
    </execution>
  </executions>
</plugin>
```

JPackage



		Heute, 11:49	--	Ordner
✓	target			
	checker-qual-3.42.0.jar	16.12.23, 00:05	231 KB	Java-JAR-Datei
>	classes	Heute, 11:49	--	Ordner
	error_prone_annotations-2.26.1.jar	12.03.24, 19:48	19 KB	Java-JAR-Datei
	failureaccess-1.0.2.jar	17.10.23, 15:16	5 KB	Java-JAR-Datei
>	generated-sources	Heute, 11:49	--	Ordner
	guava-33.1.0-jre.jar	13.03.24, 20:03	3.1 MB	Java-JAR-Datei
	j2objc-annotations-3.0.0.jar	09.03.24, 18:12	12 KB	Java-JAR-Datei
	jsr305-3.0.2.jar	28.07.21, 20:47	20 KB	Java-JAR-Datei
	listenablefuture-99...flict-with-guava.jar	28.07.21, 20:47	2 KB	Java-JAR-Datei
>	maven-archiver	Heute, 11:49	--	Ordner
>	maven-status	Heute, 11:49	--	Ordner
	PackagingDemoEx...1.0.0-SNAPSHOT.jar	Heute, 11:49	3 KB	Java-JAR-Datei

JPackage



```
jpackage --input target/ --name JPackageDemoAppWithGuava  
--main-jar PackagingDemoExamples-1.0.0-SNAPSHOT.jar  
--main-class de.java17.ApplicationExample --type dmg
```



JPackage



Icon	Name	Date	Size	Type
IntelliJ IDEA icon	IntelliJ IDEA Ultimate 2024.1	06.04.24, 22:45	3.67 GB	Programm
Java icon	JPackageDemoApp	15.01.24, 13:49	156.8 MB	Programm
Java icon	JPackageDemoAppWithGuava	Heute, 11:53	160.3 MB	Programm





DEMO & Hands on



Exercises PART 2

<https://github.com/Michaeli71/JAX-London-Best-of-Java-17-23>





PART 3: News in Java 18 to 21 LTS



JEPs in Java 18, 19, 20 and 21 LTS

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

- **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)

18

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)

- 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)

19

20

Java 21 LTS – 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
LTS

Total: 15

Normal: 8

Preview: 6

Incubator: 1

The Java Version Almanac

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

Details	Status	Documentation	Download	Compare API to	21	20	19	18	17	16	15	14	13	12	...
Java 22	DEV	API Notes	JDK JRE	21	20	19	18	17	16	15	14	13	12	11	...
Java 21	LTS	API Lang VM Notes	JDK JRE	20	19	18	17	16	15	14	13	12	11	10	...
Java 20	EOL	API Lang VM Notes	JDK JRE	19	18	17	16	15	14	13	12	11	10	9	...
Java 19	EOL	API Lang VM Notes	JDK JRE	18	17	16	15	14	13	12	11	10	9	8	...
Java 18	EOL	API Lang VM Notes	JDK JRE	17	16	15	14	13	12	11	10	9	8	7	...
Java 17	LTS	API Lang VM Notes	JDK JRE	16	15	14	13	12	11	10	9	8	7	6	...
Java 16	EOL	API Lang VM Notes	JDK JRE	15	14	13	12	11	10	9	8	7	6	5	...
Java 15	EOL	API Lang VM Notes	JDK JRE	14	13	12	11	10	9	8	7	6	5	4	...
Java 14	EOL	API Lang VM Notes	JDK JRE	13	12	11	10	9	8	7	6	5	4	1.4	...
Java 13	EOL	API Lang VM Notes	JDK JRE	12	11	10	9	8	7	6	5	4	3	1.3	...
Java 12	EOL	API Lang VM Notes	JDK JRE	11	10	9	8	7	6	5	4	3	2	1.2	...
Java 11	LTS	API Lang VM Notes	JDK JRE	10	9	8	7	6	5	4	3	2	1.3	1.2	1.1
Java 10	EOL	API Lang VM Notes	JDK JRE	9	8	7	6	5	4	3	2	1.3	1.2	1.1	...
Java 9	EOL	API Lang VM Notes	JDK JRE	8	7	6	5	4	3	2	1.3	1.2	1.1
Java 8	LTS	API Lang VM Notes	JDK JRE	7	6	5	4	3	2	1.3	1.2	1.1
Java 7	EOL	API Lang VM Notes	JDK JRE	6	5	4	3	2	1.3	1.2	1.1
Java 6	EOL	API Lang VM Notes	JDK JRE	5	4	3	2	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

```
1 import java.lang.reflect.ClassFileVersion;
2
3 public class Java21 {
4
5     public static void main(String[] args) {
6         var v = ClassFileVersion.latest();
7         System.out.printf("Hello Java bytecode version %s!", v.major());
8     }
9
10 }
```

No Support
for Preview
Features!

Java21.java ▶ Run

21+35-2513

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

Sandbox – <https://javaalmanac.io/>



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

Sandbox – <https://javaalmanac.io/>



Java21.java ► Run 21+35-2513

```
1 public class Java21 {
2
3     public static void main(String[] args) {
4         multiMatch("Python");
5         multiMatch(null);
6         multiMatch(7);
7
8         record Person(String name, int age) {}
9         multiMatch(new Person("Michael", 52));
10    }
11
12    static void multiMatch(Object obj) {
13        switch (obj) {
14            case null -> System.out.println("null");
15            case String str when str.length() > 5 -> System.out.println(str.toUpperCase());
16            case String str -> System.out.println(str.toLowerCase());
17            case Integer i -> System.out.println(i * i);
18            default -> throw new IllegalArgumentException("Unsupported type " + obj.getClass());
19        }
20    }
21 }
```

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)
```



Regular (final) features in Java 21

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



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<E>`, `Set<E>` and `Map<K, V>`
- 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
 - `List<E>`: from front to back, index based for lists
 - `HashSet<E>` has **no** encounter order
 - `TreeSet<E>` defines it indirectly by `Comparable<T>` or passed `Comparator<T>`
 - `LinkedHashSet<E>` 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.

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();  
}
```

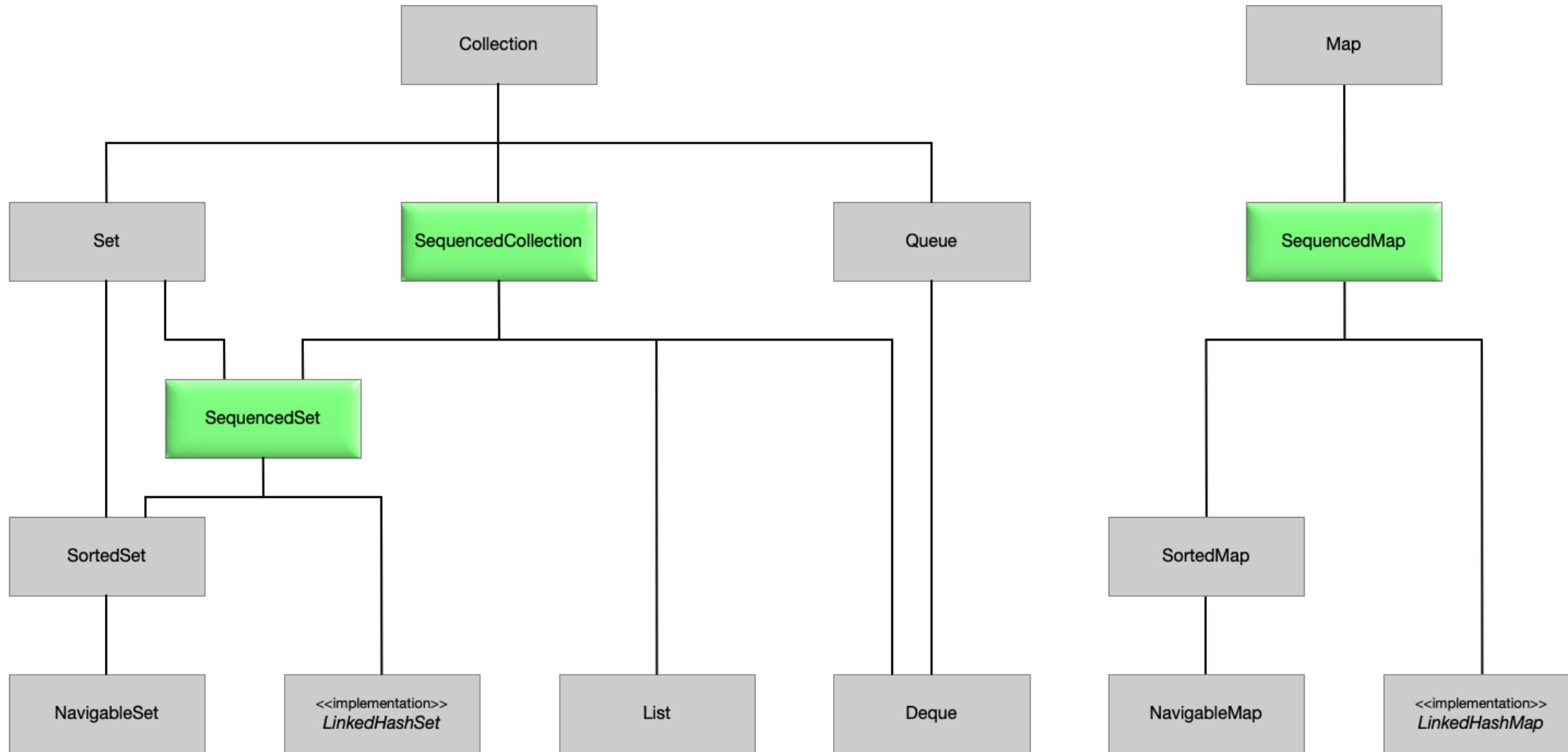
SequencedCollection defines the modifying methods:

- 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.

- Additionally 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 order.

Sequenced Collections – Retrofitted into existing type hierarchy



Sequenced Collections – The Magic Behind ... Default Methods



```
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



```
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



```
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 440: Record Patterns

<https://openjdk.org/jeps/440>





- The basis for this JEP and its predecessors JEP 405 and JEP 432 is 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 440: 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));
    }
}
```

JEP 440: 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);  
    }  
}
```

JEP 440: Record Patterns



- **Record patterns make for elegance:**

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

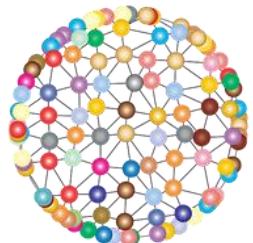
```
boolean isAllowedToDrive(Object obj) {
    if (obj instanceof Person person) {
        return person.age() >= 18 && person.hasDrivingLicense();
    }
    return false;
}
```

```
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!**



**How to make it even
more elegant?**



JEP 440: Record Patterns



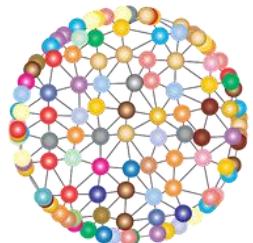
- Clean OO design would define the method in the record itself:

```
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 440: 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 origin,  
                        City destination) {  
}
```

JEP 440: 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();
            LocalDate birthday = person.birthday();

            if (reservation.destination() != null) {
                City destination = reservation.destination();
                String languageCode = destination.languageCode();

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

JEP 440: 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 origin,
        City(String name, String country, String languageCode)))
    {
        if (birthday != null && languageCode != null)
        {
            long years = ChronoUnit.YEARS.between(birthday, LocalDate.now());
            return years >= 18 &&
                List.of("EN", "DE", "FR").contains(languageCode);
        }
    }
    return false;
}
```

JEP 440: 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.**



DEMO & Hands on

Jep405_FlighReservationExample.java



JEP 441: Pattern Matching and `switch`

<https://openjdk.org/jeps/441>





- JEP 441 and its predecessors JEP 420, JEP 427 and JEP 433 lead to changes in the evaluation of the case within switch during compilation
 - On the one hand, the so-called **dominance check** has changed,
 - on the other hand, the **completeness analysis** has been corrected.
- in the syntax when specifying conditions with when instead of &&
- the support of record patterns
- a few special features
 - no more support for parentheses around record patterns:
`(if (obj instanceof (String s)))`
 - Support for qualified enum accesses

JEP 441: 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.

JEP 441: 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.out.println(i);
        default -> { }
    }
}
```

A screenshot of an IDE showing Java code. A tooltip is displayed over the third case statement. The tooltip contains the following text:
Label is dominated by a preceding case label 'String str'
Move switch branch 'String str && str.length() > 5' before 'String str'
© 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 441: Pattern Matching bei switch: Dominance check



- Problems with constants (not detected with Java 17)

```
// Fehler im Eclipse-Compiler bei Dominance-Check, unreachable wird nicht erkannt
no usages

static void dominanceExampleWithConstant(Object obj) {
    switch (obj.toString()) {
        case String str when str.length() > 5 -> System.out.println(str.strip());
        case "Sophie" -> System.out.println("My lovely daughter");
        default -> Switch label '"Sophie"' is unreachable ...
    }
}
```

A screenshot of the Eclipse IDE showing a Java code editor. The code is a switch statement with two cases: one for strings longer than 5 characters and one for the constant "Sophie". A tooltip appears over the "default" branch, stating "Switch label '"Sophie"' is unreachable". Below the editor is a context menu with options: "Remove switch branch '"Sophie"'", "More actions...", and a copy/paste icon.

- Correction so that the most specialized pattern is at the top:

```
switch (obj.toString()) {
    case "Sophie" -> System.out.println("My lovely daughter");
    case String str when str.length() > 5 -> System.out.println(str.strip());
```



- 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;
```

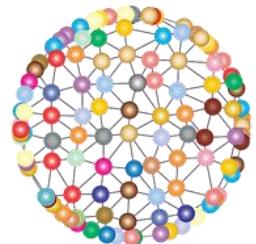
```
switch (value) {
    case -1, 0, 1 -> System.out.println("Handle special cases -1, 0 or 1");
    case Integer i && i >= 1 ->
        System.out.println("Handle positive integer cases i > 1");
    case Integer i -> System.out.println("Handle all the remaining integers");
}
```

JEP 441: Pattern Matching for switch: Dominance check



```
Integer value = calcValue();
switch (value) {
    case Integer i when i >= 1 -> System.out.println("Handle positive integer cases i > 1");
    case -1, 0, 1 -> System.out.println("Handle special cases -1, 0 or 1");
    case Integer i: Switch label '1' is unreachable
}
    : ining integers"
}
    Remove switch label '1' ↕ More actions... ↕
```

```
Integer value = calcValue();
switch (value) {
    case Integer i -> System.out.println("Handle all the remaining integers");
    case Integer i when i >= 1 -> System.out.println("Handle positive integer cases i > 1");
    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... ↕
```



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

JEP 441: 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

JEP 441: 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 441: 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")  
    }  
}
```



JEP 441: 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);  
    }  
}
```



DEMO & Hands on

SwitchMultiPatternExample.java
SwitchSpecialCasesExample.java
SwitchDominanceExample.java
SwitchCompletenessExample.java

JEP 441: Pattern Matching for switch with Record Patterns



```
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");
    }
}
```

JEP 441: 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>(String text, var count)  
            when text.contains("Michael") ->  
                System.out.println(text + " is " + count + " years old");  
        case MyPair<String, Integer>(String text, Integer 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");  
    }  
}
```

JEP 441: 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 is not possible here if you want to access the type-specific
        // methods.
        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");
    }
}
```

JEP 441: Pattern Matching for switch: Typ inference with Patterns



- With Java 21 LTS, the type reference has also been improved and you can now use var in the type specification:

```
static void recordInferenceJdk21(MyPair<String, Integer> pair)
{
    switch (pair)
    {
        case MyPair(var text, var count) when text.contains("Michael") ->
            System.out.println(text + " is " + count + " years old");
        case MyPair(var text, var 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");
    }
}
```



DEMO & Hands on

SwitchRecordPatternsExample.java
SwitchTypeInferenceExample.java



JEP 444: Virtual Threads

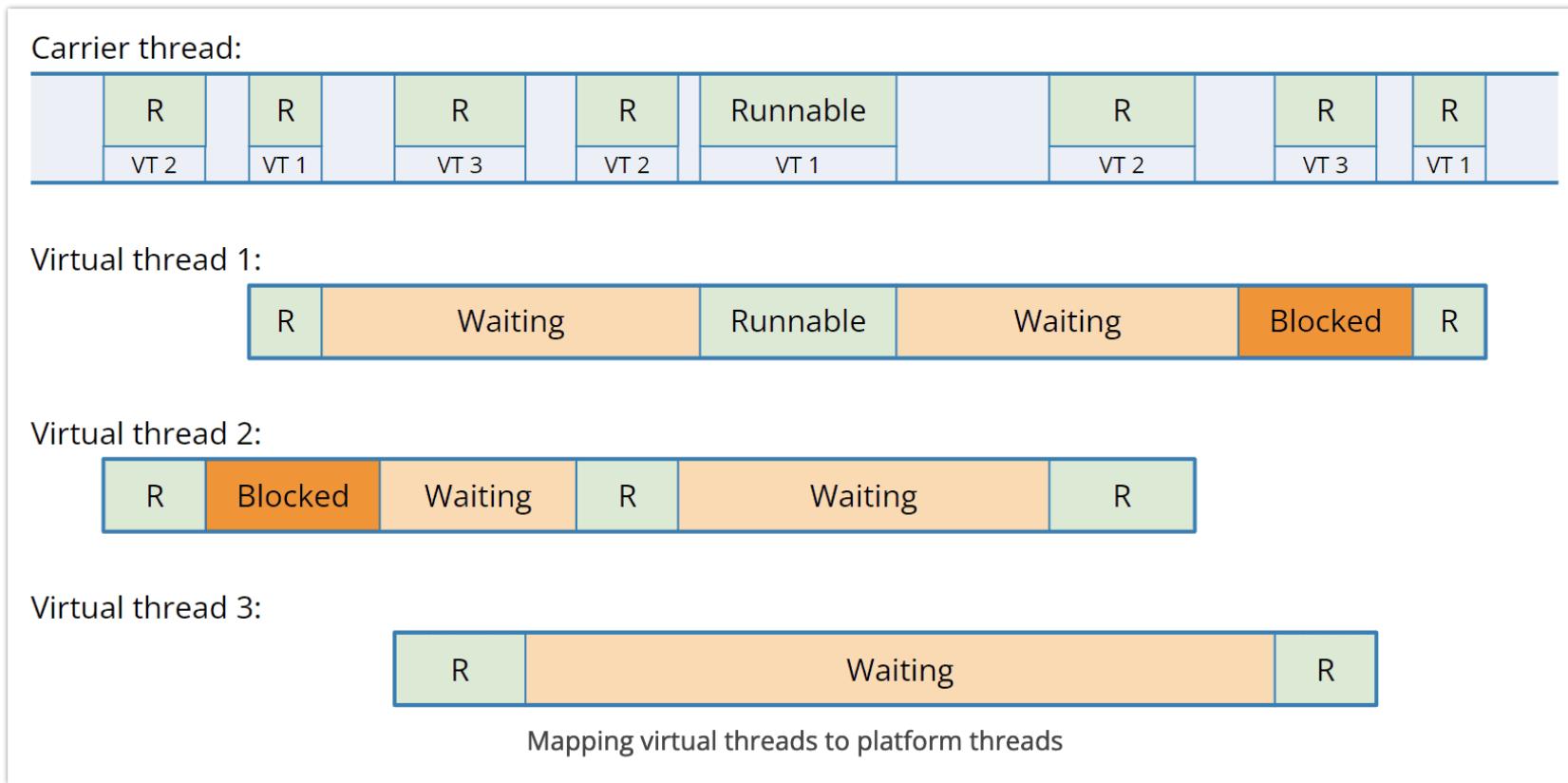
<https://openjdk.org/jeps/444>



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.



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



```
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");
}
```



- **Virtual threads permit to work with a separate thread per request in the area of server programming. This is helpful because many client requests usually perform blocking I/O such as retrieving resources.**
- **What is the problem with blocking I/O? => miserable server utilization**

Concurrency Issues

Why is it bad to block?

```
Json request      = buildContractRequest(id);
String contractJson = contractServer.getContract(request);
Contract contract = Json.unmarshal(contractJson);
```





- Performance comparison in steps of a thousand for platform and virtual threads
- The threads each wait a few seconds to simulate access to an external interface. The total time is measured at the end.

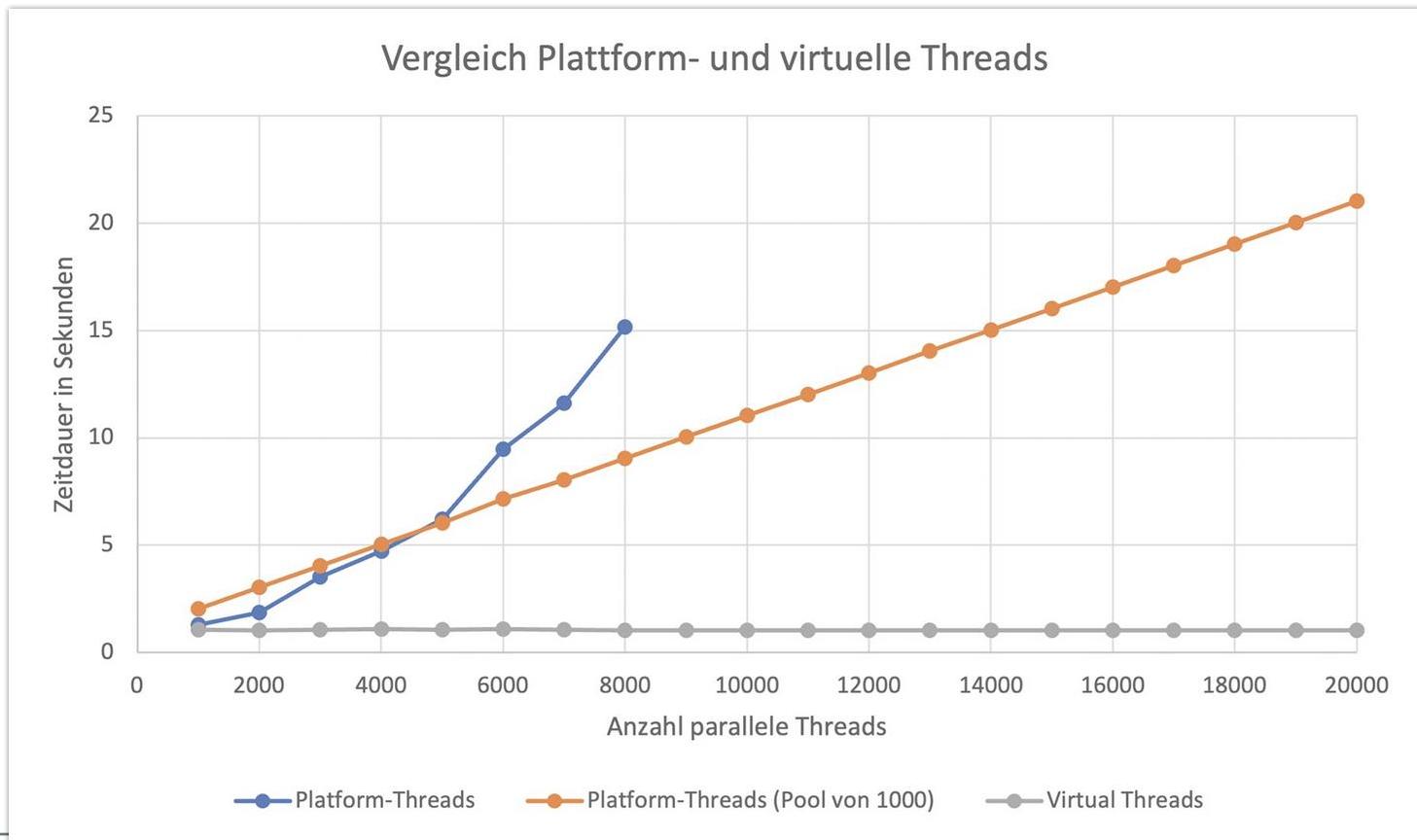
```
private static void submit1000Threads(ExecutorService executor) {  
    for (int i = 0; i < 1_000; i++) {  
        final int pos = i;  
        executor.submit(() -> {  
            Thread.sleep(Duration.ofSeconds(5));  
            return pos;  
        });  
    }  
}
```

- Several runs for 10k, 20k, ..., 50k

```
for (int i = 0; i < 10 * factor; i++) {  
    submit1000Threads(executor);  
}
```



- 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.





DEMO & Hands on

PlatformThreads.java
VirtualThreads.java



Exercises PART 3a

<https://github.com/Michaeli71/JAX-London-Best-of-Java-17-23>





Preview Features in Java 21

- 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)



String Templates
discontinued in
Java 23 after
2nd Preview in
Java 22



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 of turning old into new**

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

- =>

```
System.out.println(STR."color: \{color} ==> \{numOfChars}");
```

- **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

String Templates



- **Example 2: Caution when splitting templates!!!**

```
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";
```



- **Does that really make sense here? Or would it not be better to do the evaluation separately in both cases?**

```
String existInfo = filePath.toFile().exists() ? "does" : "does not";
String infoOldV2 = "The file " + filePath + " " + existInfo + " exist";
String infoNewV2 = STR."The file \{filePath} \{existInfo} exist";
```

String Templates and Text Blocks



- **Example**

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

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

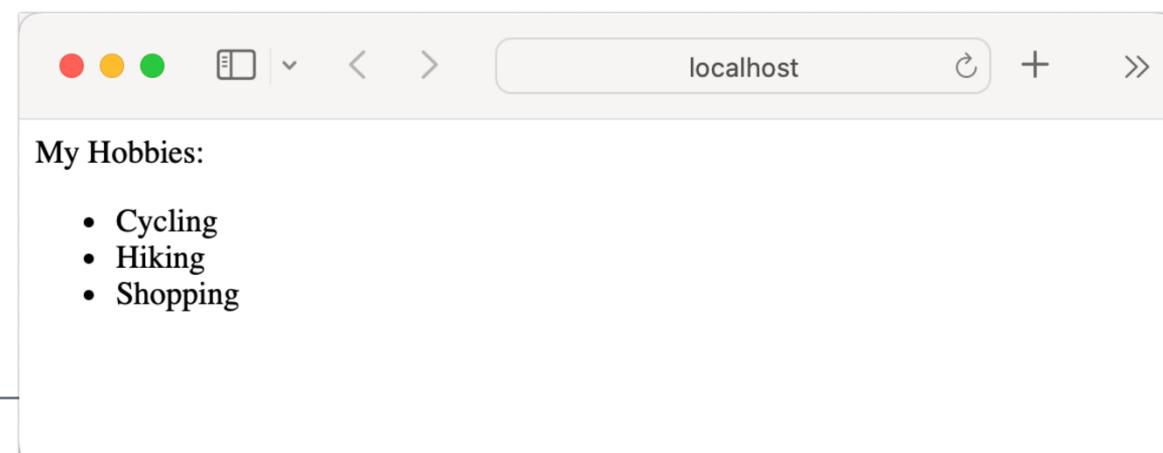
```
{  
    "statusCode": 201,  
    "msg": "CREATED"  
}
```

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



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);

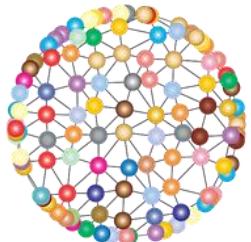
System.out.println("Sophie was born on %s at %s".formatted(
    DateTimeFormatter.ofPattern("dd.MM.YYYY").format(sophiesBirthday),
    DateTimeFormatter.ofPattern("HH:mm").format(sophiesBirthday))));
```

- =>

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



**What else can you do with
String Templates?**



String Templates – Alternative String Processors



```
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
```

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.
```

String Templates – Own String Processors





DEMO & Hands on

JEP430_StringTemplates.java
JEP430_StringTemplates_FMT.java
JEP430_StringTemplates_Advanced.java
JEP430_StringTemplatesInPractise.java



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) {}
```

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

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

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)

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 `_`)



- **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.

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 + "'");
}
```

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);
```

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 _)**

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 _)**

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.**

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.**



Unnamed Variables & Patterns



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

var cp = new ColoredPoint(new Point(1234, 567), Color.BLUE);

if (cp instanceof ColoredPoint(Point(int x,
        var |), // UNNAMED PATTERN VARIABLE
        |)) UNNAMED PATTERN
{
    System.out.println("x: " + x);
}
```

Unnamed Variables & Patterns



```
static boolean checkFirstNameAndCountryCodeAgainImproved_UNNAMED_2(Object obj)
{
    if (obj instanceof Journey(
        Person(var firstname, _, _),
        TravellInfo(_, var maxTravellingTime), _,
        City(var zipCode, _))) {

        if (firstname != null && maxTravellingTime != null && zipCode != null) {

            return firstname.length() > 2 && maxTravellingTime.toHours() < 7 &&
                zipCode >= 8000 && zipCode < 8100;
        }
    }
    return false;
}
```



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.

package preview;

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

- Python – reduced to the essentials:

```
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 useful 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/prevue/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.
- 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)");  
    }  
}
```



JEP 453: Structured Concurrency (Preview)

<https://openjdk.org/jeps/453>





- 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 453: 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.



- 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.



- **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.**



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.
- `Shutdown onFailure` – 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.



- Implementation with structured concurrency in the form of the class **StructuredTaskScope**:

```
try (var scope =
      new StructuredTaskScope.ShutdownOnSuccess<DeliveryService>())
{
    var result1 = scope.fork(() -> tryToGetPostService());
    var result2 = scope.fork(() -> tryToGetFallbackService());
    var result3 = scope.fork(() -> tryToGetCheapService());
    var result4 = scope.fork(() -> tryToGetFastDeliveryService());
    scope.join(); // NO throwIfFailed()

    System.out.println(STR."PS \{result1.state()\}/FS \{result2.state()\}" +
                      STR."/CS \{result3.state()\}/FDS \{result4.state()\}");
    System.out.println("found delivery service: " + scope.result());
}
```

- Split competing subtasks using **fork()** and collect the first available result using a blocking call to **join()**
- join()** waits until one subtask is successful
- state()** returns **SUCCESS** (first), **UNAVAILABLE** (others) or **FAILED** (exception)



Advantages of using the `StructuredTaskScope` class:

- **Task and subtasks form a self-contained unit**
- **No ExecutorService and threads from a thread pool are used.
Each subtask is executed in a new virtual thread.**
- **ShutdownOnSuccess: waits for all**
 - Errors in one of the subtasks => all other subtasks are aborted.
- **Shutdown onFailure: waits for one**
 - Error in one of the subtasks => status FAILED
- **If the calling thread is canceled, the subtasks are also canceled.**
- **Call hierarchy (calling thread and subtasks) is a bit easier to see in the thread dump.**

JEP 453: Structured Concurrency



Exception in thread "main" java.util.concurrent.ExecutionException: java.lang.IllegalStateException: JUST TO PROVIDE EX
at java.base/java.util.concurrent.FutureTask.report(FutureTask.java:122)
at java.base/java.util.concurrent.FutureTask.get(FutureTask.java:191)
at preview.api.StructuredConcurrency.**handleOldStyle**(StructuredConcurrency.java:48)
at preview.api.StructuredConcurrency.main(StructuredConcurrency.java:26)

Caused by: java.lang.IllegalStateException: JUST TO PROVIDE EX

at preview.api.StructuredConcurrency.**fetchOrders**(StructuredConcurrency.java:73)
at preview.api.StructuredConcurrency.lambda\$handleOldStyle\$1(StructuredConcurrency.java:43)
at java.base/java.util.concurrent.FutureTask.run(FutureTask.java:317)
at java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1144)
at java.base/java.util.concurrent.ThreadPoolExecutor\$Worker.run(ThreadPoolExecutor.java:642)
at java.base/java.lang.Thread.run(Thread.java:1583)

Exception in thread "main" java.util.concurrent.ExecutionException: java.lang.IllegalStateException: JUST TO PROVIDE EX
at java.base/java.util.concurrent.StructuredTaskScope\$ShutdownOnFailure.throwIfFailed(StructuredTaskScope.java:1318)
at java.base/java.util.concurrent.StructuredTaskScope\$ShutdownOnFailure.**throwIfFailed**(StructuredTaskScope.java:1295)
at preview.api.StructuredConcurrency.**handle**(StructuredConcurrency.java:57)
at preview.api.StructuredConcurrency.main(StructuredConcurrency.java:27)

Caused by: java.lang.IllegalStateException: JUST TO PROVIDE EX

at preview.api.StructuredConcurrency.**fetchOrders**(StructuredConcurrency.java:73)
at preview.api.StructuredConcurrency.lambda\$handle\$3(StructuredConcurrency.java:54)
at java.base/java.util.concurrent.StructuredTaskScope\$SubtaskImpl.run(StructuredTaskScope.java:889)
at java.base/java.lang.VirtualThread.run(VirtualThread.java:311)



DEMO & Hands on

StructuredConcurrency.java



Exercises PART 3b

<https://github.com/Michaeli71/JAX-London-Best-of-Java-17-23>





PART 4: News in Java 22 and 23

Java 22 – What is included?



- JEP 423: Region Pinning for G1
- [JEP 447: Statements before super\(...\) \(Preview\)](#)
- [JEP 454: Foreign Function & Memory API](#)
- [JEP 456: Unnamed Variables & Patterns](#)
- JEP 457: Class-File API (Preview)
- [JEP 458: Launch Multi-File Source-Code Programs](#)
- [JEP 459: String Templates \(Second Preview\)](#)
- JEP 460: Vector API (Seventh Incubator)
- [JEP 461: Stream Gatherers \(Preview\)](#)
- JEP 462: Structured Concurrency (Second Preview)*
- [JEP 463: Implicitly Declared Classes and Instance Main Methods \(Second Preview\)](#)
- [JEP 464: Scoped Values \(Second Preview\)](#)

Java 22

Total: 12

Normal: 4

Preview: 7

Incubator: 1



Final Features in Java 22

- JEP 454: Foreign Function & Memory API
- JEP 456: Unnamed Variables & Patterns
- JEP 458: Launch Multi-File Source-Code Programs



JEP 454: Foreign Function & Memory API

<https://openjdk.org/jeps/454>



Foreign Function & Memory API



- JEP 454 on the Foreign Function & Memory API provides an API for accessing functionalities and memory areas outside the JVM.
- Sometimes, it is necessary to access functionalities written in C or C++, for example, from Java programs. It may also be required to address memory areas outside the JVM.
- Although the JNI (Java Native Interface), which supports the calling of native code, has been available for such tasks since Java's early days, it also has various shortcomings. There can also be problems with garbage collection if you reserve memory in C with malloc and then want to return it to the Java call.

Foreign Function & Memory API



```
public static void main(final String[] args) throws Throwable
{
    // 1. SymbolLookup for common lib
    var linker = Linker.nativeLinker();
    final SymbolLookup stdlib = linker.defaultLookup();

    // 2. MethodHandle for "strlen" from C Standard Library
    var strlenMethodHandle = linker.downcallHandle(stdlib.find("strlen").orElseThrow(),
        FunctionDescriptor.of(JAVA_LONG, ADDRESS));

    // 3. convert Java in C String and provide in C memory
    final Arena auto = Arena.ofAuto();
    var strMemorySegment = auto.allocateFrom("direct c call of strlen");

    // 4. Call of "foreign function"
    final long len = (long) strlenMethodHandle.invoke(strMemorySegment);

    System.out.println("len = " + len);
    // 5. Memory will be cleaned up automatically due to "Arena.ofAuto()"
}
```



JEP 456: Unnamed Variables & Patterns

<https://openjdk.org/jeps/456>



JEP 456: Unnamed Patterns and Variables



- This JEP finalizes its predecessor JEP 443 and makes it possible to mark variables or parts within record patterns as unused and unusable by using a `_`.
- As a reminder, the following three variants are supported:
 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 `_`)

Unnamed Variables & Patterns



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

var cp = new ColoredPoint(new Point(1234, 567), Color.BLUE);

if (cp instanceof ColoredPoint(Point(int x,
        var |), // UNNAMED PATTERN VARIABLE
        |)) UNNAMED PATTERN
{
    System.out.println("x: " + x);
}
```



JEP 458: Launch Multi-File Source-Code Programs

<https://openjdk.org/jeps/458>



JEP 458: Launch Multi-File Source-Code Programs



- Direct compilation has existed since Java 11 LTS and allows individual Java files to be executed directly without explicit prior compilation.
- This feature is quite useful for small experiments and to create simple command line tools.
- One limitation up to and including Java 21 LTS was that direct compilation could only be used to execute a single Java file.
- The larger programs become, the more you want to split functionality into different classes in their own Java files. This is not yet supported for direct compilation.
- This JEP addresses this very issue and implements an improvement in the Java program start. This allows the transition from small programs to larger programs to be done incrementally without having to introduce a build tool like Maven or Gradle.

JEP 458: Launch Multi-File Source-Code Programs



```
package jep458_Launch_MultiFile_SourceCode_Programs;

public class MainApp
{
    public static void main(final String[] args) {
        var result = Helper.performCalculation();
        System.out.println(result);
    }
}
```

```
package jep458_Launch_MultiFile_SourceCode_Programs;

class Helper
{
    public static String performCalculation() {
        return "Heavy, long running calculation!";
    }
}
```

```
$ java MainApp.java
Heavy, long running calculation!
```

JEP 458: Launch Multi-File Source-Code Programs



```
package jep458_Launch_MultiFile_SourceCode_Programs;

public class MainAppV2
{
    public static void main(final String[] args) {
        var result = StringHelper.mark(Helper.performCalculation());
        System.out.println(result);
    }
}
```

```
package jep458_Launch_MultiFile_SourceCode_Programs;

class StringHelper
{
    public static String mark(String input) {
        return ">>" + input + "<<";
    }
}
```

```
$ java MainAppV2.java
>>Heavy, long running calculation!<<
```

MainAppV2.java



DEMO & Hands on



Preview Features in Java 22

- JEP 447: Statements before super(...) (Preview)
- JEP 461: Stream Gatherers (Preview)
- JEP 463: Implicitly Declared Classes and Instance Main Methods (Second Preview)
- JEP 464: Scoped Values (Second Preview)

All are
part of
Java 23 ...
so let's
move on



PART 4: News in Java 22 and 23

IDE & Tool Support for Java 23



- **Eclipse: Version 2024-09 (with Plugin)**
- **IntelliJ: Version 2024.2.2**
- **Maven: 3.9.9, Compiler Plugin: 3.13.0**
- **Gradle: 8.10**
- **Activation of preview features / Incubator necessary**
 - In dialogs
 - In build scripts



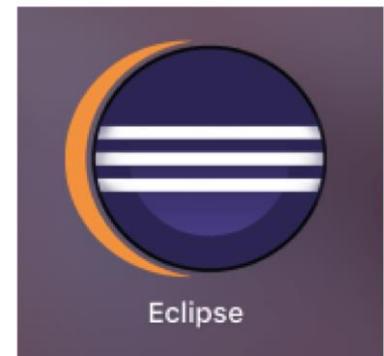
Maven™

 **Gradle**

IDE & Tool Support Java 23



- Eclipse 2024-09 with Plugin
- Activation of preview features necessary



Eclipse Marketplace

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

Search Recent Popular Favorites Installed Research at the Eclipse

Find: Java 23

All Markets

Java 23 Support for Eclipse 2024-09 (4.33)

This marketplace solution provides Java 23 support for Eclipse 2024-09 (4.3). It is the latest Eclipse release, which is... [more info](#)

by Eclipse Foundation, EPL 2.0

java

0 installs: 70 (71 last month)

Eclipse Marketplace

Properties for Java23Examples

Java Compiler

Enable project specific settings

JDK Compliance

Use compliance from execution environment 'JavaSE-23' on the [Java Build Path](#)

Compiler compliance level: 23

Use '--release' option

Use default compliance settings

Enable preview features for Java 23

Preview features with severity level: Info

Generated .class files compatibility: 23

Source compatibility: 23

Classfile Generation

IDE & Tool Support



- Activation of preview features / Incubator necessary

Project Structure

Project
Default settings for all modules. Configure these parameters for each module on the module page as needed.

Name: Java23Examples

SDK: 23 Oracle OpenJDK 23 - aarch64 

Language level: 23 (Preview) - Primitive types in patterns, implicitly declared classes, etc. 

Compiler output: 

Used for module subdirectories, Production and Test directories for the corresponding sources.

?

Cancel Apply OK

The screenshot shows the 'Project Structure' dialog in IntelliJ IDEA. The 'Project' tab is selected. The 'Name' field contains 'Java23Examples'. The 'SDK' dropdown is set to '23 Oracle OpenJDK 23 - aarch64'. The 'Language level' dropdown is set to '23 (Preview) - Primitive types in patterns, implicitly declared classes, etc.'. Red arrows point to the 'Edit' buttons next to the 'SDK' and 'Language level' dropdowns. At the bottom, there are 'Cancel', 'Apply', and 'OK' buttons.





- Activation of Preview Features / Incubator necessary

sourceCompatibility=23
targetCompatibility=23



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

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





- Activation of Preview Features / Incubator necessary

```
java {  
    toolchain {  
        languageVersion = JavaLanguageVersion.of(23)  
    }  
}
```

```
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

```
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.13.0</version>
    <configuration>
      <source>23</source>
      <target>23</target>
      <!-- Important for Preview Features -->
      <compilerArgs>--enable-preview</compilerArgs>
    </configuration>
  </plugin>
</plugins>
```

```
<plugin>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.13.0</version>
  <configuration>
    <release>23</release>
    <compilerArgs>
      <arg>--enable-preview</arg>
      <arg>--add-modules</arg>
      <arg>jdk.incubator.vector</arg>
    </compilerArgs>
  </configuration>
</plugin>
```

Maven™

3.9.9

IDE & Tool Support Java 23 for Vector API Incubator



Run Configurations

Create, manage, and run configurations

Run a Java application

Name: FirstVectorExample

Main Arguments JRE Dependencies Source Environment Common Prototype

Program arguments:

VM arguments:

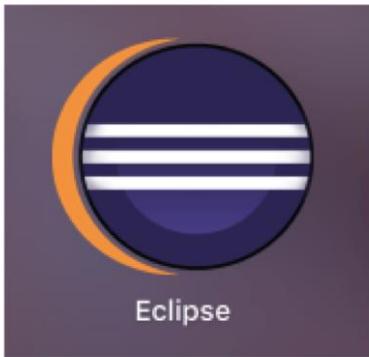
```
--enable-preview --add-modules jdk.incubator.vector
```

Variables... Variables...

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

Working directory:

Show Command Line Revert Apply



IDE & Tool Support for Vector API Incubator



Java

Editor

- Inspections
- File and Code Templates
- Live Templates
- Copyright**
- Formatting**
- Java
- JavaScript
- GUI Designer
- Intentions
- Natural Languages**
- Grammar and Style

Plugins

Build, Execution, Deployment

- Compiler**
- Java Compiler
- Annotation Processors
- Kotlin Compiler

- Debugger**
- Data Views
- Java
- Java Type Renderers
- JavaScript
- Stepping
- Unit Tests

Settings

Build, Execution, Deployment > Compiler > Java Compiler

Revert changes ← →

Use compiler: Javac

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

Project bytecode version: Same as language level

Per-module bytecode version:

Module	Target bytecode version
Java23Examples	23

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:

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

'/' recommended in paths for cross-platform configurations

Override compiler parameters per-module:

+ -

Cancel Apply OK



IDE & Tool Support for Vector API Incubator



Run/Debug Configurations

Name: FirstVectorExample Store as project file

Run on: Local machine Manage targets...

Run configurations may be executed locally or on a target: for example in a Docker Container or on a remote host using SSH.

Build and run Modify options ↴

java 23 SDK of 'Java23' --enable-preview --add-modules jdk.incubator.vector

jep469_Vector_Api.FirstVectorExample

Program arguments

VM options. CLI arguments to the 'Java' command. Example: -ea -Xmx2048m. ↴V

Working directory: \VA-BEST-OF/2024/W_JAX_it_tage_Java_23_WORKSHOP_2024/Java23Exam

Environment variables: Environment variables or .env files

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

- + jep469_Vector_Api.*

Edit configuration templates...

?

Run Cancel Apply OK





Overview: JEPs in Java 23

Java 23 – What's included?



- **JEP 455:** [Primitive Types in Patterns, instanceof, and switch \(Preview\)](#)
- JEP 466: Class-File API (Second Preview)
- **JEP 467:** [Markdown Documentation Comments](#)
- JEP 471: Deprecate the Memory-Access Methods in sun.misc.Unsafe for Removal
- **JEP 473:** [Stream Gatherers \(Second Preview\)](#)
- JEP 474: ZGC: Generational Mode by Default
- **JEP 476:** [Module Import Declarations \(Preview\)](#)
- **JEP 477:** [Implicitly Declared Classes and Instance Main Methods \(Third Preview\)](#)
- JEP 480: Structured Concurrency (Third Preview)*
- **JEP 481:** [Scoped Values \(Third Preview\) + neu: aus Tex](#)
- **JEP 482:** [Flexible Constructor Bodies \(Second Preview\)](#)
- **JEP 469:** [Vector API \(Eighth Incubator\)](#)

Java 23

Total: 12

Normal: 3

Preview: 8

Incubator: 1

*no changes, just more time for feedback, already covered in Java 21 LTS part



Final Features in Java 23

- JEP 467: Markdown Documentation Comments



JEP 467: Markdown Documentation Comments

<https://openjdk.org/jeps/467>



JEP 467: Markdown Documentation Comments



- At the time Java became popular in the late 1990s, the choice of HTML elements as the JavaDoc comment specification was absolutely logical.
- In recent years, Markdown has become increasingly popular for documentation.
- Markdown comments are identified by three slashes (///):

```
/// Returns the greater of two `int` values. That is, the
/// result is the argument closer to the value of
/// [Integer#MAX_VALUE]. If the arguments have the same value,
/// the result is that same value.
///
/// @param a an argument.
/// @param b another argument.
/// @return the larger of `a` and `b`.
public static int max(int a, int b) {
    return (a >= b) ? a : b;
}
```

JEP 467: Markdown Documentation Comments



- This JEP allows you to create comments with Markdown instead of HTML elements.
- In IntelliJ via the menu Tools > Generate JavaDoc....
- Generates JavaDoc in the generated-doc folder

✓	Java23Examples ~/Desktop/Vorträge/A_JAVA-BE
>	.gradle
>	.idea
>	.settings
>	build
✓	generated-doc
>	index-files
>	jep454_Foreign_Function
>	jep466_Class_File_Api
>	jep469_Vector_Api
>	jep473_Stream_Gatherers
>	legal
>	resource-files
>	script-files
>	syntax
	allclasses-index.html
	allpackages-index.html
	DateAndTimeAPI.html
	element-list
	FirstGathererExample.html
	help-doc.html
	index.html
	InitialExample.html

JEP 467: Markdown Documentation Comments



- It is also displayed in IntelliJ directly on the commented program element (class / method)

```
/// Returns the greater of two `int` values. That is, the
/// result is the argument closer to the value of
/// [Integer#MAX_VALUE]. If the arguments have the same
/// value, the result is that same value.
///
/// @param a an argument.
/// @param b another argument.
/// @return the larger of `a` and `b`.
public static int max(int a, int b) {
    return (a >= b) ? a : b;
}
```

c syntax.MarkDownComment

```
@Contract(pure = true) ➔ ➔
public static int max(
    int a,
    int b
)
```

Returns the greater of two `int` values. That is, the result is the argument closer to the value of `Integer#MAX_VALUE`. If the arguments have the same value, the result is that same value.

Params: `a` – an argument.
`b` – another argument.

Returns: the larger of `a` and `b`.



- Text passages should be emphasized from time to time

- italics (*...* or) or bold (**...**)
- Change font by backtick ('...') to typewriter font. Bold and/or italics are also possible.
- Integrate multi-line source code snippets with ("...") into a comment.

```
/// **FETT** \
/// *kursiv* \
/// _kursiv_ \
/// _**FETT und KURSIV**_ \
/// `code-font` \
/// _**`code-font FETT und KURSIV`**_ \
///
/// Mehrzeiliger Sourcecode:
/// ``
/// public static int max(int a, int b) {
///     return (a >= b) ? a : b;
/// }
/// ``
```

syntax

```
public class MarkDownComment
```

FETT

kursiv

kursiv

FETT und KURSIV

code-font

code-font FETT und KURSIV \

Mehrzeiliger Sourcecode:

```
public static int max(int a, int b) {
    return (a >= b) ? a : b;
}
```



- **References to any program elements (modules, classes, methods, etc.) by [<ref>]**
- **Types defined in `java.lang` can be written without the package, other types must be specified fully qualified.**

`/// [java.base/] - verweist auf ein Modul \`

`/// [java.util] - verweist auf ein Package`

`///`

`/// Hinweis: _**`java.lang`**_ kann man im Verweis weglassen: \`

`/// [java.lang.String] - verweist auf eine Klasse \`

`/// [String] - verweist auf eine Klasse \`

`/// [Integer] - verweist auf eine Klasse`

`///`

`/// [String#chars()] - verweist auf eine Methode \`

`/// [Integer#valueOf(String, int)] - verweist auf eine Methode`

`///`

`/// [String#CASE_INSENSITIVE_ORDER] - verweist auf ein Attribut \`

`/// [SPECIAL_ORDER][String#CASE_INSENSITIVE_ORDER] - Verweis mit Beschriftung`

JEP 467: Markdown Documentation Comments – References



```
/// [java.base/] - verweist auf ein Modul \
/// [java.util] - verweist auf ein Package
///
/// Hinweis: _**`java.lang`**_ kann man im Verweis weglassen: \
/// [java.lang.String] - verweist auf eine Klasse \
/// [String] - verweist auf eine Klasse \
/// [Integer] - verweist auf eine Klasse
///
/// [String#chars()] - verweist auf eine Methode \
/// [Integer#valueOf(String, int)] - verweist auf eine Methode
///
/// [String#CASE_INSENSITIVE_ORDER] - verweist auf en Attribut
/// [SPECIAL_ORDER][String#CASE_INSENSITIVE_ORDER]
```

java.base[↗] - verweist auf ein Modul
java.util[↗] - verweist auf ein Package

Hinweis: **java.lang** kann man im Verweis weglassen:
String[↗] - verweist auf eine Klasse
String[↗] - verweist auf eine Klasse
Integer[↗] - verweist auf eine Klasse

String.chars()[↗] - verweist auf eine Methode
Integer.valueOf(String, int)[↗] - verweist auf eine Methode

String.CASE_INSENSITIVE_ORDER[↗] - verweist auf ein Attribut
SPECIAL_ORDER[↗] - Verweis mit Beschriftung



```
/// - Punkt A  
/// * Punkt B  
/// - Punkt C  
///  
/// 1. Eintrag 1  
/// 1. Eintrag 2 -- **wird automatisch nummeriert, also 2.**  
/// 1. Eintrag 3  
/// 2. Eintrag 4 -- **wird automatisch auf 4. geändert**
```

- Punkt A
- Punkt B
- Punkt C
- 1. Eintrag 1
- 2. Eintrag 2 -- **wird automatisch nummeriert, als**
- 3. Eintrag 3
- 4. Eintrag 4 -- **wird automatisch auf 4. geändert**

```
/// | Latein | Griechisch |  
/// |-----|-----|  
/// | a | &alpha; (alpha) |  
/// | b | &beta; (beta) |  
/// | c | &gamma; (gamma) // &Gamma; |  
/// | ... | ... |  
/// | z | &omega; (omega) |
```

Latein Griechisch

a	α (alpha)
b	β (beta)
c	γ (gamma) // Γ
...	...
z	ω (omega)



Preview Features in Java 23

- JEP 455: Primitive Types in Patterns, instanceof, and switch (Preview)
- JEP 473: Stream Gatherers (Second Preview)
- JEP 476: Module Import Declarations (Preview)
- JEP 477: Implicitly Declared Classes and Instance Main Methods (Third Preview)
- JEP 481: Scoped Values (Third Preview)
- JEP 482: Flexible Constructor Bodies (Second Preview)



JEP 455: Primitive Types in Patterns, instanceof, and switch (Preview)

<https://openjdk.org/jeps/455>



JEP 455: Primitive Types in Patterns, instanceof, and switch (Preview)



- With instanceof and switch, type and value checks can be performed. Conveniently, modern Java also allows record patterns to be specified there.

```
// Pattern Matching with instanceof
if (obj instanceof String str && str.length() > 7) {
    System.out.println("string " + str + " is longer than 7 characters");
} else if (obj instanceof Double d && d > 100) {
    System.out.println("double " + d + " is larger than 100");
} else {
    System.out.println("arbitrary obj: " + obj);
}
```

```
// Pattern Matching with switch
switch (obj) {
    case String str when str.length() > 7 ->
        System.out.println("string " + str + " is longer than 7 characters");
    case Double d when d > 100 ->
        System.out.println("double " + d + " is larger than 100");
    default -> System.out.println("arbitrary obj: " + obj);
}
```

- The aim of this JEP was to improve pattern matching and make it more universally valid by also supporting so-called primitive type patterns to allow primitive types.

JEP 455: Primitive Types in Patterns, instanceof, and switch (Preview)



- Example

```
String msg = switch (logLevel.severity())
{
    case 0 -> "info";
    case 1 -> "warning";
    case 2 -> "error";
    default -> "unknown severity: " + logLevel.severity();
};
```

Duplicate call,
otherwise no access
to value in default



```
String msg = switch (logLevel.severity()) {
    case 0 -> "info";
    case 1 -> "warning";
    case 2 -> "error";
    // JEP 455
    case int severity -> "unknown severity: " + severity;
};
```

JEP 455: Primitive Types in Patterns, instanceof, and switch (Preview)



- **Example with Guards**

```
void evaluate(final int score)
{
    var result = switch (score)
    {
        case int value when value > 100 -> "DISQUALIFIED due to cheating";
        case int value when value >= 80 -> "excellent";
        case int value when value >= 50 -> "okaish";
        case int value when value >= 0 && value <= 50 -> "below expectations";
        default -> throw new IllegalStateException("Invalid score: " + score);
    };
    System.out.println("Your score: " + result);
}
```

JEP 455: Primitive Types in Patterns, instanceof, and switch (Preview)



- Corner Case with floating point numbers

```
float val = 1.0f;  
switch (val) {  
    case 1.0f -> System.out.println("1.0");  
    case 0.99999999f -> System.out.println("0.9999..");  
    default -> System.out.println("something else");  
}
```

```
float val = 1.0f;  
switch (val) {  
    case 1.0f -> System.out.println("1.0");  
    case 0.99999999f -> System.out.println("0.9999..");  
    default -> Sys  
}
```

Duplicate label '1'

Remove switch branch '0.99999999f' ↕

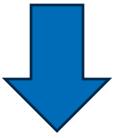
More actions... ↕

JEP 455: Primitive Types in Patterns, instanceof, and switch (Preview)



- Value Range Checks simplified

```
void checkByteAndPrintOld(int value)
{
    if (value >= -128 && value <= 127)
    {
        byte byteValue = (byte)value;
        System.out.println("byte: " + byteValue);
    }
}
```



```
void checkByteAndPrint(int value)
{
    if (value instanceof byte byteValue)
    {
        System.out.println("byte: " + byteValue);
    }
}
```



JEP 473: Stream Gatherers (Second Preview)

<https://openjdk.org/jeps/473>





- The streams introduced in Java 8 were already quite powerful right from the start.
- Various extensions were added to terminal operations in the following Java versions. Terminal operations are used to complete a stream's calculations and convert the stream into a collection or a result value, for example.
- This JEP implements an extension of the stream API to support user-defined intermediate operations. Until now, various predefined intermediate operations have existed, but there has been no extension option.
- Such an extension is desirable for performing tasks that previously could not be achieved easily or only with tricks and were rather cumbersome.

JEP 473: Stream Gatherers



- Let's assume we want to filter out all duplicates from a stream and specify a criterion for this:

```
var result = Stream.of("Tim", "Tom", "Jim", "Mike").  
    distinctBy(String::length). // Hypothetical  
    toList();
```

- You can solve this conventionally with a trick as follows:

```
var result = Stream.of("Tim", "Tom", "Jim", "Mike").  
    map(DistinctByLength::new).  
    distinct().  
    map(DistinctByLength::str).  
    toList();
```

JEP 473: Stream Gatherers



- The trick is to use a record that encapsulates a string and implements equals() and hashCode() specifically aligned to the string length:

```
var result = Stream.of("Tim", "Tom", "Jim", "Mike").  
    map(DistinctByLength::new).  
    distinct().  
    map(DistinctByLength::str).  
    toList();
```

```
record DistinctByLength(String str) {  
  
    @Override public boolean equals(Object obj) {  
        return obj instanceof DistinctByLength(String other)  
            && str.length() == other.length();  
    }  
}
```

```
    @Override public int hashCode() {  
        return str == null ? 0 : Integer.hashCode(str.length());  
    }  
}
```



- Another example is grouping a stream's data into sections of a fixed size. For example, four numbers are to be combined/grouped into one unit, and only the first three groups are to be included in the result.

```
var result = Stream.iterate(0, i -> i + 1).  
    windowFixed(4).      // Hypothetical  
    limit(3).  
    toList();  
  
// result ==> [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]
```

- Over the years, various intermediate operations such as `distinctBy()` or `windowFixed()` have been proposed as additions to the Stream API.
- These are often useful in specific contexts, but they would make the Stream API rather bloated and (further) complicate access to the (already extensive) API.



- Java 22 and 23 now offer a method `gather(Gatherer)` for providing a user-defined intermediate operation, analogous to `collect(Collector)` for terminal operations.
- The interface `java.util.stream.Gatherer` is used for this purpose, but it is a little challenging to implement yourself.
- Conveniently, there are various predefined Gatherers in the utility class `java.util.stream.Gatherers` such as:
 - `windowFixed()`
 - `windowSliding()`
 - `fold()`
 - `scan()`

JEP 473: Stream Gatherers – windowFixed()



- To divide a stream into smaller components of fixed size without overlapping, `windowFixed()` is used.

```
private static void windowFixed() {  
    var result = Stream.iterate(0, i -> i + 1).  
        gather(Gatherers.windowFixed(4)).  
        limit(3).  
        toList();  
    System.out.println("windowFixed(4): " + result);  
  
    var result2 = Stream.of(0, 1, 2, 3, 4, 5, 6).  
        gather(Gatherers.windowFixed(3)).  
        toList();  
    System.out.println("windowFixed(3): " + result2);  
}
```

- In some cases, the dataset does not contain enough elements. This means that the last subrange simply contains fewer elements.

```
windowFixed(4): [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]  
windowFixed(3): [[0, 1, 2], [3, 4, 5], [6]]
```

JEP 473: Stream Gatherers – windowSliding()



- To divide a stream into smaller components of fixed size with overlapping, `windowSliding()` is used:

```
private static void windowSliding() {  
    var result = Stream.iterate(0, i -> i + 1).  
        gather(Gatherers.windowSliding(4)).  
        limit(3).  
        toList();  
    System.out.println("windowSliding(4): " + result);  
  
    var result2 = Stream.of(0, 1, 2, 3, 4, 5, 6).  
        gather(Gatherers.windowSliding(3)).  
        toList();  
    System.out.println("windowSliding(3): " + result2);  
}
```

- In some cases, the dataset does not contain enough elements. This means that the last sub-range simply contains fewer elements (not shown here):

```
windowSliding(4): [[0, 1, 2, 3], [1, 2, 3, 4], [2, 3, 4, 5]]  
windowSliding(3): [[0, 1, 2], [1, 2, 3], [2, 3, 4], [3, 4, 5], [4, 5, 6]]
```

JEP 473: Stream Gatherers – fold()



- The `fold()` method is used to combine the values of a stream. Similar to `reduce()`, a start value and a calculation rule are specified:

```
private static void foldSum()
{
    var crossSum = Stream.of(1, 2, 3, 4, 5, 6, 7).
        gather(Gatherers.fold(() -> 0L,
            (result, number) -> result + number)).
        findFirst();
    System.out.println("sum with fold(): " + crossSum);
}
```

- To access the value, a call to `findFirst()` is used, which returns an `Optional<T>`:
- `sum with fold(): Optional[28]`
- `gather()` returns a stream as a result (here one with one-element). Calling `toList()` instead of `findFirst()` you would get a one-element list:

`sum with fold(): [28]`

JEP 473: Stream Gatherers – fold()



- The **fold()** method is used to combine the values of a stream. Similar to `reduce()`, a start value and a calculation rule are specified:

```
private static void foldMult()
{
    var crossMult = Stream.of(10, 20, 30, 40, 50).
        gather(Gatherers.fold(() -> 1L,
                           (result, number) -> result * number)).
        findFirst();
    System.out.println("mult with fold(): " + crossMult);
}
```

- To access the value, the call to `findFirst()` is used again, which returns an `Optional<T>`:

```
mult with fold(): Optional[12000000]
```

JEP 473: Stream Gatherers – fold()



- What happens if we also want to execute actions for the value combination and these actions are not defined for the types of the values, in this case int?
- As an example, a numerical value is converted into a string and this is repeated according to the numerical value with repeat():

```
private static void foldRepeat()
{
    var repeatedNumbers = Stream.of(1, 2, 3, 4, 5, 6, 7).
        gather(Gatherers.fold(() -> """",
            (result, number) -> result + ("'" +
                number).repeat(number))).  

        toList();
    System.out.println("repeat with fold(): " + repeatedNumbers);
}
```

- The output is as follows:

```
repeat with fold(): [12233344445555666667777777]
```

JEP 473: Stream Gatherers – scan()



- If the elements of a stream are to be merged into new combinations so that one element is added at a time, then this is the task of `scan()`.
- The method works in a similar way to `fold()`, which combines the values to produce a result. With `scan()`, however, a new result is produced for each combination of values:

```
private static void scanRepeat()
{
    var repeatedNumbers = Stream.of(1, 2, 3, 4, 5, 6, 7).
        gather(Gatherers.scan(() -> "",  

                           (result, number) -> result + (" " +  

                           number).repeat(number))).  

        toList();
    System.out.println("repeat with scan(): " + repeatedNumbers);
}
```

- The output is as follows:

```
repeat with scan(): [1, 122, 122333, 1223334444, 122333444455555, 122333444455555666666,  

12233344445555566666777777]
```



DEMO & Hands on



JEP 476: Module Import Declarations (Preview)

<https://openjdk.org/jeps/476>



JEP 476: Module Import Declarations (Preview)



- With JEP 476, it is possible to use the types exported by a module with one import
- More precisely: all public types from the packages exported by a module
- Recap:
 - Since the early days of Java, a conscious decision has been made to ensure that all types from the `java.lang` package are automatically available in every class and do not have to be imported explicitly.
 - The reason for this is convenience and ease of use.
 - This mechanism allows you to use the common types `String`, `Integer`, or `Exception` directly in every program.

JEP 476: Module Import Declarations (Preview)



- At the latest when programs become bigger, types from other packages are frequently required.
- Such as the container classes `ArrayList<E>`, `HashSet<E>` and `HashMap<K, V>`, or the associated interfaces `List<E>`, `Set<E>` and `Map<K, V>`.
- This requires separate imports such as these:

```
import java.util.ArrayList;  
import java.util.HashSet;  
import java.util.List;  
import java.util.Set;
```

- There are controversial opinions and debates in the Java community as to whether a general import like this is better or worse:

```
import java.util.*;
```

JEP 476: Module Import Declarations (Preview)



- With Java 23, it is now easier to reuse things by importing entire modules at once.

```
import module <module-name>;
```

- This makes it possible to avoid very repetitive and similar import instructions.
- This provides all public types, i.e., classes, interfaces, and enums, from all packages exported by the module without further import.
- As indicated direct imports can result in quite a few lines. For example, if you wanted to import all types analogous to those from the module import

```
import module java.base;
```

would require over 50 individual imports, including many types that are integrated with `import java.io.*` and `import java.util.*`.

JEP 476: Module Import Declarations (Preview)



- This provides all public types, i.e., classes, interfaces, and enums, from all packages exported by the module without further import.
- Conveniently, this also works transitively. If you write
`import module java.sql;`
- the types from the `java.xml` module are also available. This is practical to keep the module imports clear.
- It is also easier to get started and use classes in the context of direct compilation and implicitly declared classes.

JEP 476: Module Import Declarations (Preview)



```
import java.util.List;
import java.util.stream.Stream;

public class InitialModuleImportExample {
    public static void main(final String[] args) {
        List<String> result = Stream.of("AB", "BC", "CD", "DE").
            filter(str -> str.contains("C")).
            toList();
        System.out.println(result);
    }
}

import module java.base;

public class InitialModuleImportExample2 {
    public static void main(final String[] args) {
        List<String> result = Stream.of("AB", "BC", "CD", "DE").
            filter(str -> str.contains("C")).
            toList();
        System.out.println(result);
    }
}
```

JEP 476: Module Import Declarations (Preview)



```
import javax.swing.*;
import java.io.IOException;
import java.util.List;
import java.util.stream.Stream;

public class ModuleImportExample3 {
    public static void main(String[] args) {
        List<String> result = Stream.of("AB", "BC", "CD", "DE").
            filter(str -> str.contains("C")).
            toList();
        System.out.println(result);

        IO.print("New Style IO");

        JOptionPane.showMessageDialog(null, "Info");
    }
}
```

JEP 476: Module Import Declarations (Preview)



```
import module java.base; // import java.util.* , java.util.stream.* , ...  
import module java.desktop; // import javax.swing.* ; usw.
```

```
public class ModuleImportExample3 {  
    public static void main(String[] args) {  
        List<String> result = Stream.of( ...values: "AB", "BC", "CD", "DE") .  
            fi | Reference to 'List' is ambiguous, both 'java.util.List' and 'java.awt.List' match :  
            t |  
        System.out.println("New Style IO");  
        JOptionPane.showMessageDialog(null, "Info");  
    }  
}
```

- However, we now stumble across the error that the List type is no longer unique. As the error message clearly shows, the List type is defined in the java.awt package and in java.util.
- Reason: Because a module import provides all types from all packages exported by the module, it is possible that types with the same name from different packages exist.

JEP 476: Module Import Declarations (Preview)



```
import module java.base; // import java.util.* , java.util.stream.* , ...  
import module java.desktop; // import javax.swing.* ; usw.
```

```
import java.util.List;
```

```
public class ModuleImportExample3 {  
    public static void main(String[] args) {  
        List<String> result = Stream.of("AB", "BC", "CD", "DE").  
            filter(str -> str.contains("C")).  
            toList();  
        System.out.println(result);  
  
        IO.print("New Style IO");  
  
        JOptionPane.showMessageDialog(null, "Info");  
    }  
}
```

JEP 476: Module Import Declarations (Preview)



- For the Date class it is also quite likely that you will encounter problems with a module import as follows, because Date exists in the java.util and java.sql packages:

```
jshell> import module java.base;
```

```
jshell> import module java.sql;
```

```
jshell> new Date(4711)
```

```
| Fehler:
```

```
| Referenz zu Date ist mehrdeutig
```

```
| Sowohl Klasse java.sql.Date in java.sql als auch Klasse java.util.Date in java.util stimmen überein
```

```
| new Date(4711)
```

```
| ^__^
```

- And import solves the problem:

```
jshell> import java.util.Date;
```

```
jshell> var today = new Date()  
today ==> Sun Sep 08 17:03:32 CEST 2024
```



JEP 477: Implicitly Declared Classes and Instance Main Methods (Third Preview)

<https://openjdk.org/jeps/477>



JEP 477: Implicitly Declared Classes and Instance Main Methods



- Java 21 LTS introduced the Unnamed Classes and Instance Main Methods as a preview feature with JEP 445. This allows to write “Hello World” as «short» as possible:

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

- The resulting implicit classes must not have a package specification.

```
package jep477.is.not.supported;

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

JEP 477: Implicitly Declared Classes and Instance Main Methods



- The selection of the appropriate `main()` method got simplified:
 - If there is a static method with a parameter, this is used: «Parameter First»
 - Java 21 LTS offered a complicated four-stage process:

```
public class InstanceMainMethodExample
{
    public void main()
    {
        System.out.println("InstanceMainMethodExample");
    }
}
```

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

- =>

Static main

JEP 477: Implicitly Declared Classes and Instance Main Methods



- Selection of the appropriate `main()` method based on parameter:

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

    public static void main()
    {
        System.out.println("Static main");
    }
}
```

- =>

InstanceMainMethodExample

JEP 477: Implicitly Declared Classes and Instance Main Methods



- Two `main()` methods with same signature are not allowed:

```
public class InstanceMainMethodExample3 {  
    public void main() { System.out.println("InstanceMainMethodExample"); }  
  
    public static void main() {  
        System.out.println("Main Method Example");  
    }  
}
```

'main()' is already defined in 'jep477_Implicitly_Declared_Classes.InstanceMainMethodExample3'

↳ jep477_Implicitly_Declared_Classes.InstanceMainMethodExample3

public static void main()

Java23Examples

⋮

JEP 477: Implicitly Declared Classes and Instance Main Methods



- Selection of the appropriate `main()` method simplified:

```
public class InstanceMainMethodExample
{
    public void main()
    {
        System.out.println("InstanceMainMethodExample");
    }

    /*
    public static void main(final String[] args)
    {
        System.out.println("Static main");
    }
    */
}
```

- =>

InstanceMainMethodExample



DEMO & Hands on

JEP 477: Implicitly Declared Classes and Instance Main Methods



- JEP 477 from Java 23 contains the innovations from Java 22 already described. In addition, two significant innovations have been added in Java 23:
 - **Interaction with the console:** Implicitly declared classes automatically import three static methods `print()`, `println()` and `readln()` defined in the `java.io.IO` class that simplify textual interaction with the console.
 - **Automatic module import from `java.base`:** Implicitly declared classes automatically import all public classes and interfaces of the packages exported by the `java.base` module.
- Based on both, the `main()` method in Java 23 can be written more clearly and briefly as follows:

```
void main()
{
    println("Shortest and Python-like 'Hello World!'");
}
```



- **Interaction with console tend to be cumbersome and complicated, especially input**
- **The `java.io.IO` class acts as a workaround providing three new methods:**

```
public static void println(Object obj);
public static void print(Object obj);
public static String readln(String prompt);
```

- **These help to read in name and print greeting as follows:**

```
void main()
{
    var name = readln("Please enter your name: ");
    println("Hello " + name);
}
```

- **For implicitly declared classes it's like doing this:**

```
import static java.io.IO.*;
```

JEP 477: Automatic module import of java.base



- With increasing experience, you tend to want to implement more complex things, namely those that go beyond simple gimmicks and interactions with the console.
- Let's imagine, for example, that the inputs are to be managed in a list or saved in a file for later use.
- The JDK offers nice functionalities for this, but they usually require some imports
- Not easy for beginners to remember packages and hierarchies
- Oracle has decided to make it easier to get started with Java by automatically importing module `java.base` for each implicitly declared class.
- This means important and popular APIs in common packages such as `java.io`, `java.math` and `java.util` can be used directly.

JEP 477: Automatic module import of java.base



- Ease of development of smaller programs:

```
void main()
{
    String name = readIn("Please enter your name:");

    var authors = List.of("Tim", "Tom", "Mike", "Michael");
    for (var author_name : authors)
    {
        if (name.equalsIgnoreCase(author_name))
        {
            println(name + " you are registered as author!");
        }
    }
}
```

- Works as if these imports are stated:

```
import java.util.List;
import static java.io.IO.*;
```



- A program that is implemented as an implicitly declared class allows you to focus more intensely on the task at hand.
- Everything irrelevant can (initially) be omitted. Nevertheless, all components are interpreted in the same way as in a regular class.
- Transition is absolutely easy:
 - Just add a class declaration around main()
 - Add a Package statement
 - Add the imports

```
package jep477_Implicitly_Declared_Classes;  
  
import java.util.List;  
  
import static java.io.IO.*;  
  
public class GrowingClass  
{  
    void main()  
    {  
        ... // same as before  
    }  
}
```



JEP 481: Scoped Values (Third Preview)

<https://openjdk.org/jeps/481>





- **Scoped values are a modern alternative to ThreadLocal variables that are intended to store information scope/thread-specific.**
- **In combination with virtual threads, scoped values are often a better alternative because they have the following advantages over ThreadLocal variables:**
 - They are only valid for a specific scope and a specific execution and therefore time period.
 - They are immutable during execution, but can be rebound afterwards.
 - They automatically pass on their assignment to all child threads, such as those created by a StructuredTaskScope.
 - In the case of ThreadLocal, its value is copied to the child threads in the form of an InheritableThreadLocal.
- **The last point is important because there can potentially be millions of virtual threads and a copy of data can then be unfavorable in terms of storage technology.**

JEP 481: Scoped Values – Providing Context Values



- **Scoped Values are modelled as static variables.**
- **When executing the `performLogin()` method, the user data is extracted from the request passed as a parameter and then stored in the scoped value using `where()`.**

```
public class LoginUtil
{
    public static final ScopedValue<User> LOGGED_IN_USER = ScopedValue.newInstance();

    // ...

    public void performLogin(final Request request)
    {
        User loggedInUser = authenticateUser(request);
        ScopedValue.where(LOGGED_IN_USER,
            loggedInUser).run(() -> service.performAction());
    }

    // ...
}
```

- **Processing is started in the scope using the `run()` method and an action in the form of a Runnable. In this case, the `performAction()` method of a service is called, but does not receive any parameters:**

JEP 481: Scoped Values – Usage / Access



- **Accessing the values calling a get() method on a static variable:**

```
public class XyzService
{
    public void performAction()
    {
        var loggedInUser = LoginUtil.LOGGED_IN_USER.get();

        System.out.println("performing action with: " + loggedInUser);
        System.out.println("collected data: " + retrieveDataFor(loggedInUser));
    }

    private String retrieveDataFor(User loggedInUser)
    {
        return "SOME DATA";
    }
}
```

• =>

```
performing action with: User[name=FALLBACK, pwd=[C@15aeb7ab]
collected data: SOME DATA
```

JEP 481: Scoped Values – Special Cases



- Checking for value and providing fallback or throwing exception:

```
if (LoginUtil.LOGGED_IN_USER.isBound())
{
    var loggedInUser = LoginUtil.LOGGED_IN_USER.get()
    System.out.println("performing action with: " + loggedInUser);
}
else
{
    // perform fallback actions
}
```

```
var loggedInUser = LoginUtil.LOGGED_IN_USER.orElse(new User("FALLBACK",
    "PWD".toCharArray()));
```

```
var loggedInUser = LOGGED_IN_USER.orElseThrow(() ->
    new IllegalStateException("invalid user"));
```



- Let's assume `performCalculation()` may throw a `IOException` and thus a checked exception. This has to be handled in an strange way like this:

```
try
{
    var result = ScopedValue.callWhere(ScopedValuesExample.LOGGED_IN_USER,
                                       user, Java23Examples::performCalculation);
    System.out.println("Calculated result: " + result);
}
catch (Exception e)
{
    if (e instanceof IOException ioe)
        handleIOException(ioe);

    throw new RuntimeException(e);
}
```

- An attempt to write `catch (IOException ioe)` results in a compilation error: Unhandled exception: `java.lang.Exception`.



- As an innovation in Java 23, there are improvements in exception handling

```
try
{
    var result = ScopedValue.callWhere(ScopedValuesExample.LOGGED_IN_USER,
                                       user, Java23Examples::performCalculation);
    System.out.println("Calculated result: " + result);
}
catch (IOException ioe)
{
    handleIOException(ioe);
}
```

- And even better for unchecked exceptions:

```
var result = ScopedValue.callWhere(ScopedValuesExample.LOGGED_IN_USER,
                                       user,
                                       Java23Examples::performCalculationUnchecked);
System.out.println("Calculated result: " + result);
```



DEMO & Hands on

- LoginUtil & XyzService
 - ScopedValuesExample
 - Java23Examples
-



JEP 482: Flexible Constructor Bodies (Second Preview)

<https://openjdk.org/jeps/482>



JEP 482: Flexible Constructor Bodies



- This JEP aims to give developers more freedom when implementing constructors.
- In particular, certain actions will be possible before super() (and even this()) is called, for example, to check constructor arguments.*
- As before, it should be guaranteed that constructors are executed from top to bottom in the inheritance hierarchy during class instantiation. This means that the instructions in a subclass constructor cannot affect the instantiation of the base class.
- In addition, the whole thing should not require any changes to the JVM.
- In Java 22, this was introduced as JEP 447: Statements before super(...) (Preview) and is now continued with JEP 482 under a different name.

*Until now, this has only been possible using tricks such as static helper methods or additional helper constructors.

JEP 482: Flexible Constructor Bodies



- Sometimes it makes sense to validate the constructor parameter(s) before passing them (otherwise unchecked) when calling the base class constructor.

```
class BaseInteger
{
    private final long value;

    BaseInteger(final long value)
    {
        this.value = value;
    }

    public long getValue()
    {
        return value;
    }

    public static void main(final String[] args)
    {
        new BaseInteger(4711);
    }
}
```

JEP 482: Flexible Constructor Bodies



```
public class PositiveBigIntegerOld1 extends BigInteger
{
    public PositiveBigIntegerOld1(long value)
    {
        super(value);          // Potentially unnecessary work

        if (value <= 0)
            throw new IllegalArgumentException("non-positive value");
    }
}
```

- If we look at the source code, it doesn't look very elegant. Furthermore, the check only takes place after the base class has been constructed ...
- Potentially unnecessary calls and object constructions have already taken place
- Especially older legacy code is often ingloriously characterized by the fact that (too) many actions already take place in the constructor.

JEP 482: Flexible Constructor Bodies



- **Conventional workaround: static helper method**

```
public class PositiveBigIntegerOld2 extends BigInteger
{
    public PositiveBigIntegerOld2(final long value)
    {
        super(verifyPositive(value));
    }

    private static long verifyPositive(final long value)
    {
        if (value <= 0)
            throw new IllegalArgumentException("non-positive value");

        return value;
    }
}
```



- The argument check is much easier to read and understand if the validation logic takes place directly in the constructor before `super()` is called.
- JEP 447 and JEP 482 allow the arguments of a constructor to be validated before the constructor of the super class is called:

```
public class PositiveBigIntegerNew extends BigInteger
{
    public PositiveBigIntegerNew(final long value) {
        if (value <= 0)
            throw new IllegalArgumentException("non-positive value");

        super(value);
    }
}
```

JEP 482: Flexible Constructor Bodies



- Sometimes it makes sense to execute actions before calling `this()` to avoid multiple actions, like calls to `split()`, in the following:

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

record MyPoint3dOld(int x, int y, int zy)
{
    public MyPoint3dOld(final String values)
    {
        this(Integer.parseInt(values.split(",")[0].strip()),
              Integer.parseInt(values.split(",")[1].strip()),
              Integer.parseInt(values.split(",")[2].strip()));
    }
}
```



- With the new syntax, we can extract the actions from the call to `this()` and, in particular, call the `split()` only once.
- An additional helper method `parseInt()` may be introduced if you want to make the stripping more elegant and the constructor easier to read:

```
record MyPoint3d(int x, int y, int z)
{
    public MyPoint3d(final String values)
    {
        var separatedValues = values.split(",");
        int x = parseInt(separatedValues[0]);
        int y = parseInt(separatedValues[1]);
        int z = parseInt(separatedValues[2]);

        this(x, y, z);
    }

    private static int parseInt(final String strValue) {
        return Integer.parseInt(strValue.strip());
    }
}
```



- In the context of inheritance, surprises can occasionally occur when methods are called in constructors that are overwritten in subclasses.

```
public class BaseClass
{
    private final int baseValue;

    public BaseClass(int baseValue)
    {
        this.baseValue = baseValue;

        logValues();
    }

    protected void logValues()
    {
        System.out.println("baseValue: " + baseValue);
    }
}
```

JEP 482: Flexible Constructor Bodies – News in Java 23



```
public class SubClass extends BaseClass
{
    private final String subClassInfo;

    public SubClass(int baseValue, String subClassInfo)
    {
        super(baseValue);
        this.subClassInfo = subClassInfo;
    }

    protected void logValues()
    {
        super.logValues();
        System.out.println("subClassInfo: " + subClassInfo);
    }

    public static void main(final String[] args)
    {
        new SubClass(42, "SURPRISE");
    }
}
```

baseValue: 42
subClassInfo: **null**

During the processing of the base class constructor, the attribute subClassInfo is still unassigned, as the call to super() takes place BEFORE the assignment to the variable. This results in the above but unexpected output.

JEP 482: Flexible Constructor Bodies – News in Java 23



```
public class NewSubClass extends BaseClass {  
  
    private final String subClassInfo;  
  
    public NewSubClass(int baseValue, String subClassInfo)  
    {  
        this.subClassInfo = subClassInfo;  
        super(baseValue);  
    }  
  
    protected void logValues()  
    {  
        super.logValues();  
        System.out.println("subClassInfo: " + subClassInfo);  
    }  
  
    public static void main(final String[] args)  
    {  
        new NewSubClass(42, "AS_EXPECTED");  
    }  
}
```

baseValue: 42
subClassInfo: AS_EXPECTED

During the processing of the base class constructor, the attribute subClassInfo is now already unassigned, as the call to super() takes place AFTER the assignment to the variable. This results in the above and expected output.



DEMO & Hands on

- SubClass
 - NewSubClass
-



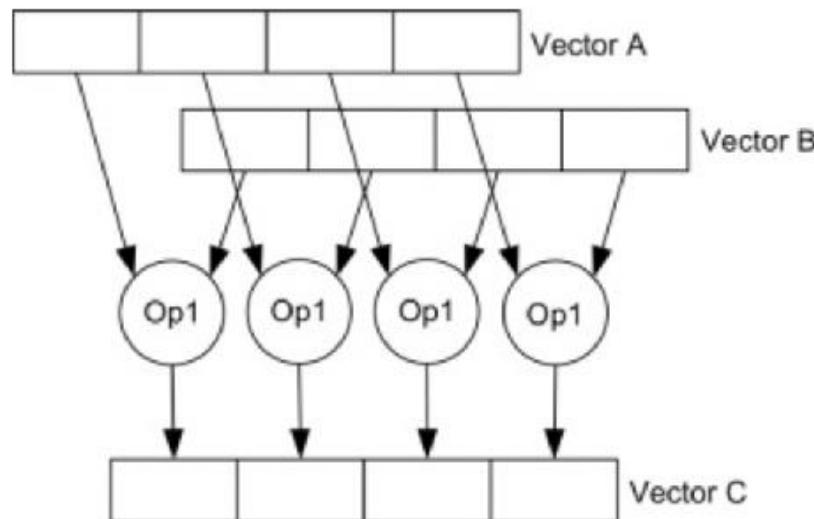
Incubator Features in Java 23

- JEP 469: Vector API (Eighth Incubator)
-



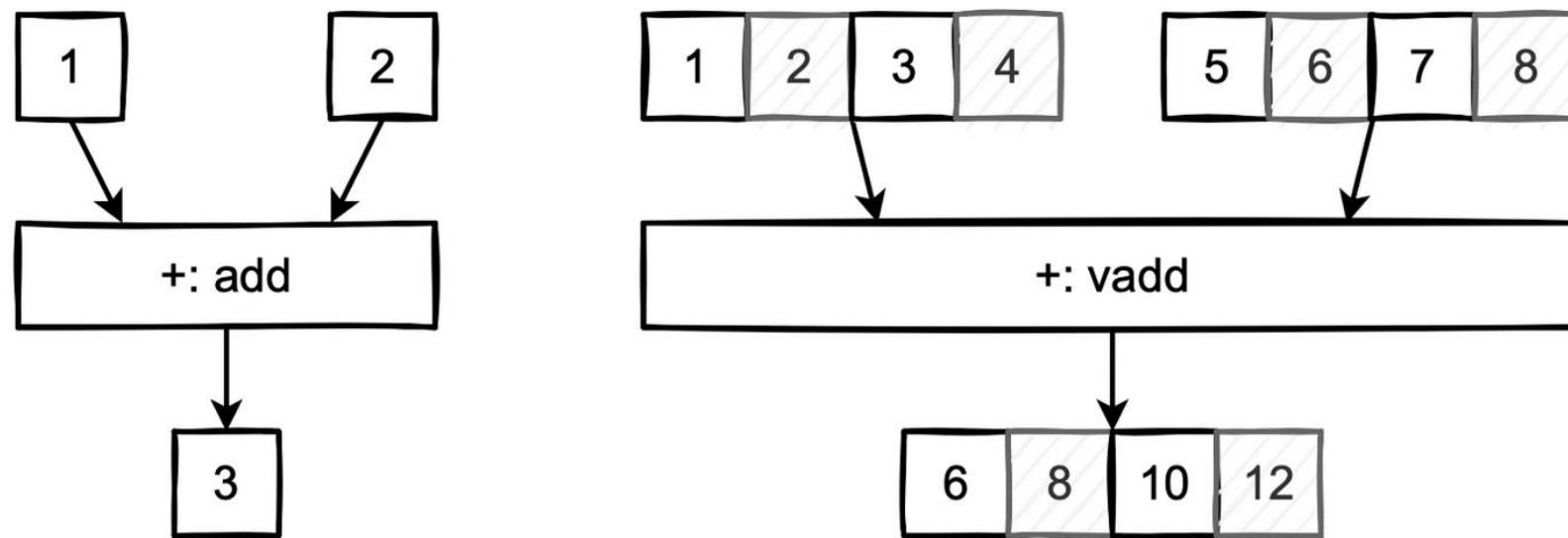
JEP 469: Vector API (Eighth Incubator in Java 23)

<https://openjdk.org/jeps/469>





- 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:





- 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 conventionally 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];
}
```



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

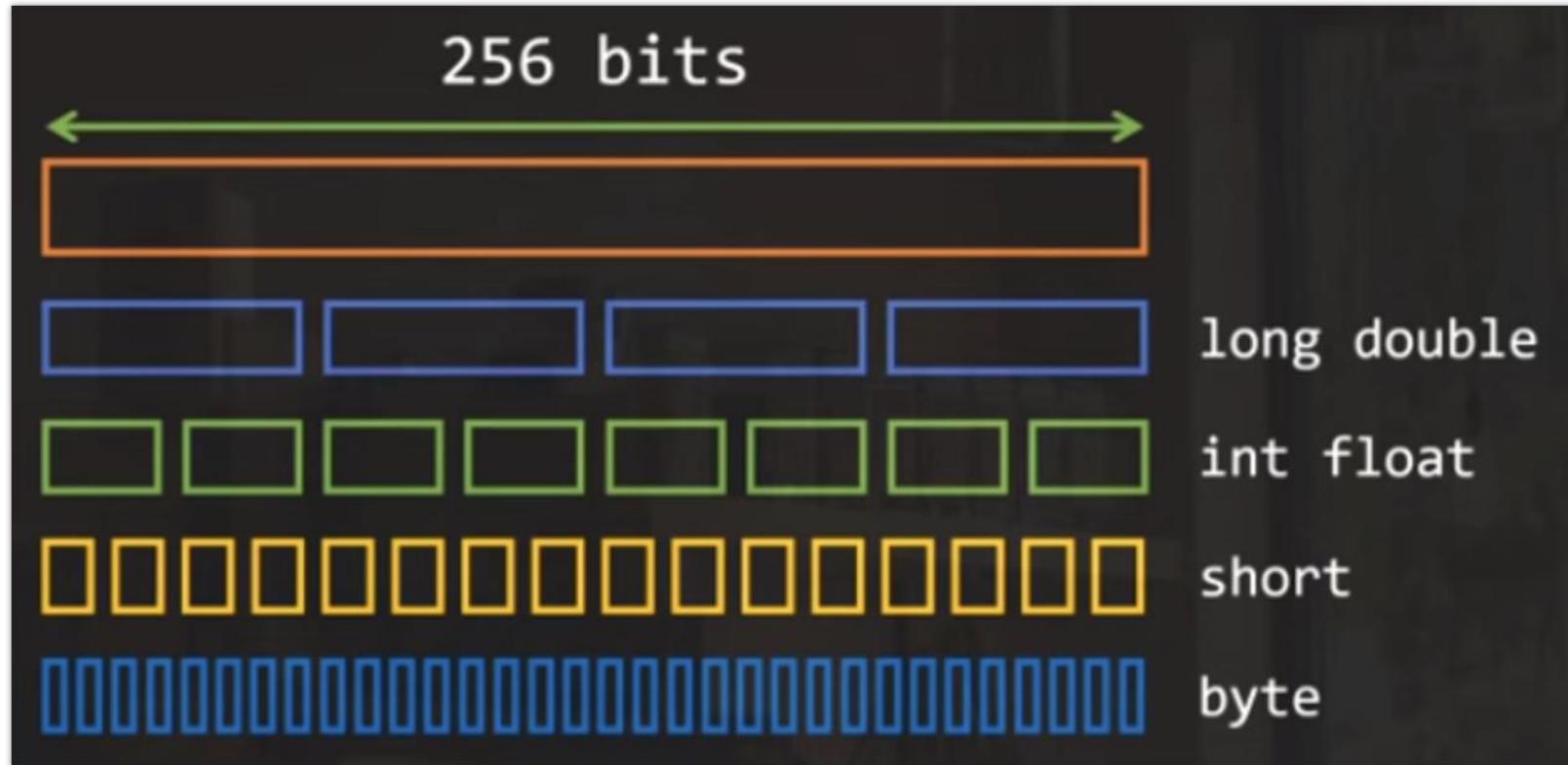
```
int[] a = { 1, 2, 3, 4, 5, 6, 7, 8 };
int[] b = { 1, 2, 3, 4, 5, 6, 7, 8 };
```

```
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`.
- Thereafter the appropriate action is specified by a method like `add()` or `mul()` or others.



- Effect of vector size:



- Prefer to use: `IntVector.SPECIES_PREFERRED`



- Let's consider the example from JEP 469 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 469: 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;
    }
}
```

WARNING: Using incubator modules: jdk.incubator.vector
result using scalar calculation: [-2.0, -8.0, -18.0, -32.0, -50.0, -72.0, -98.0, -128.0, -162.0, -200.0]
result using Vector API: [-2.0, -8.0, -18.0, -32.0, -50.0, -72.0, -98.0, -128.0, -162.0, -200.0]

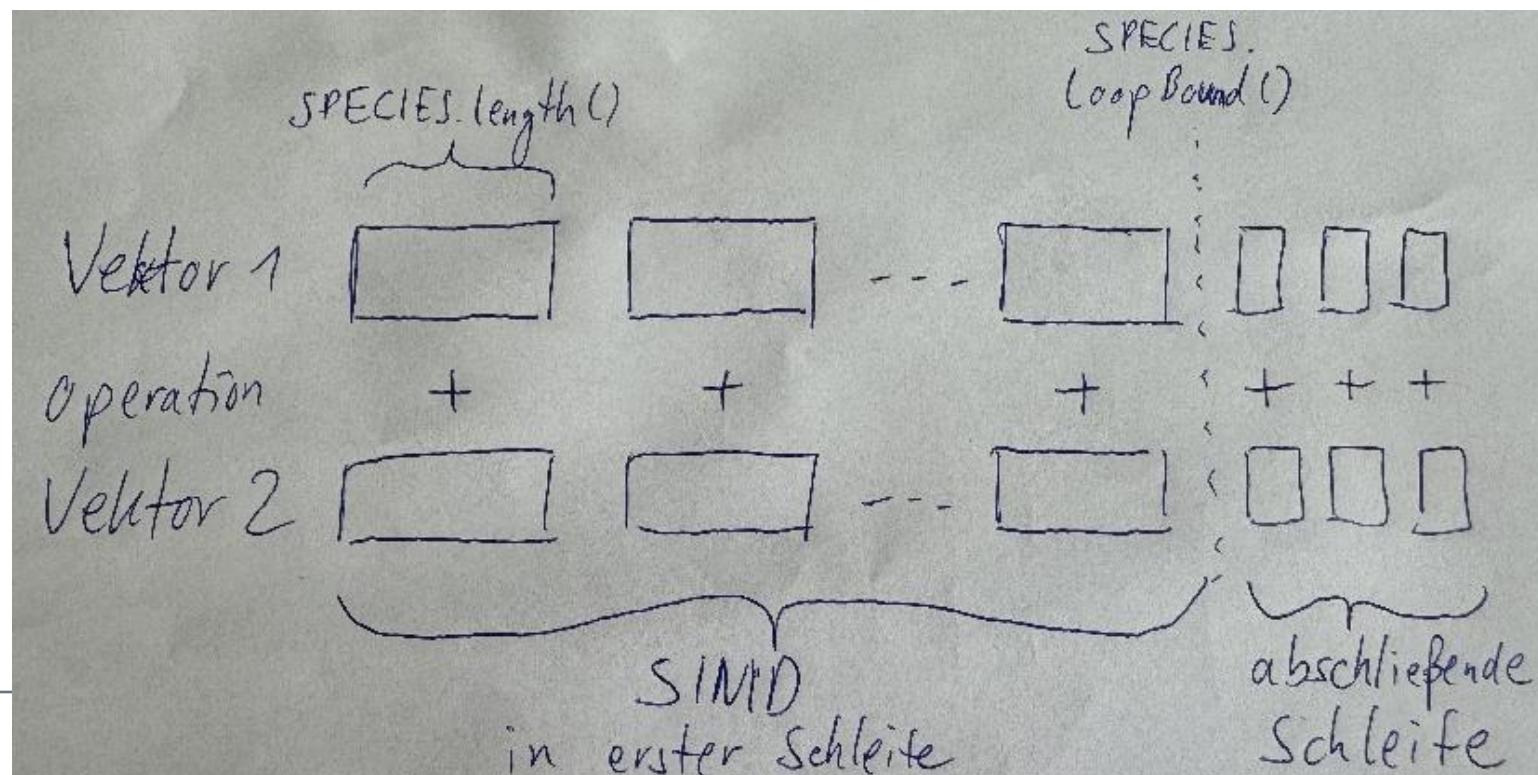
JEP 469: Vector API



```
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.





- Let's consider commenting out the last loop:

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

- =>

WARNING: Using incubator modules: jdk.incubator.vector

result using scalar calculation: [-2.0, -8.0, -18.0, -32.0, -50.0, -72.0, -98.0, -128.0, -162.0, -200.0]

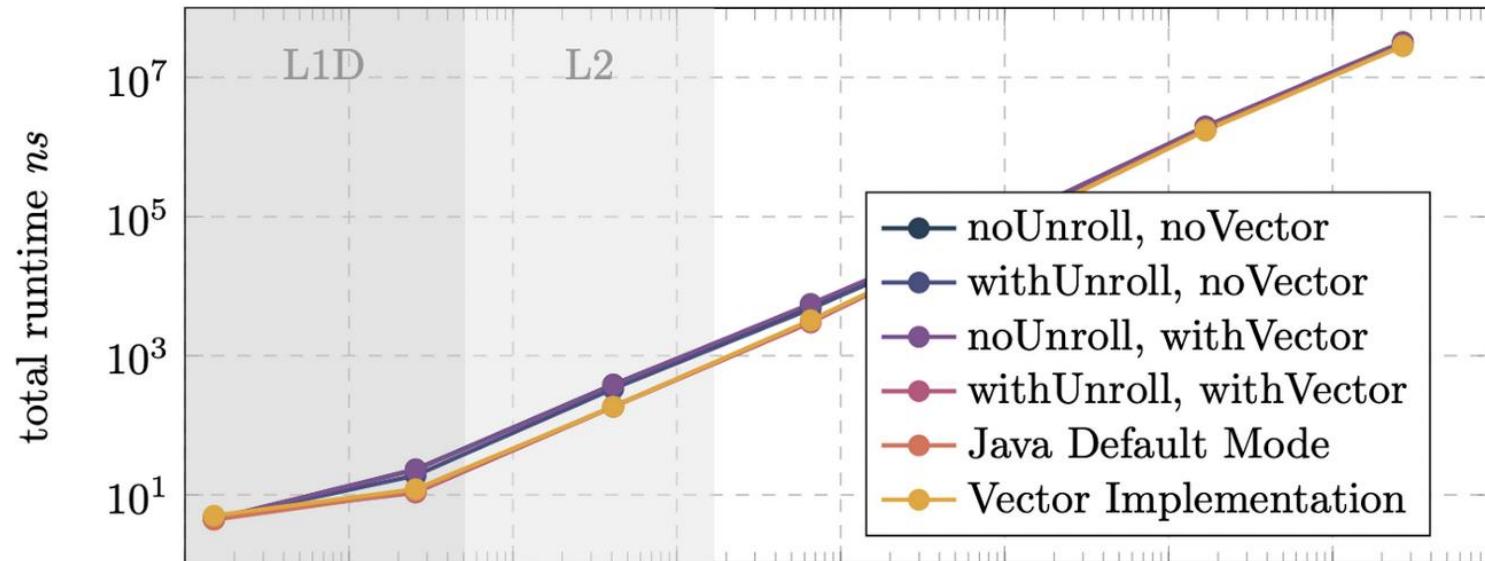
result using Vector API: [-2.0, -8.0, -18.0, -32.0, -50.0, -72.0, -98.0, -128.0, 0.0, 0.0]

- Some values would then simply not be calculated and therefore have the value 0.0.
- The influence of the final loop on the performance is negligible, because with very large vectors only a minimal fraction of the values are calculated with it. However, this is absolutely necessary in order to ensure a correct calculation for any vector length.

JEP 469: Vector API Benchmarking



Benchmark: $c[n] = a[n] + b[n]$



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

JEP 469: Vector API (in Console and JShell)



```
$ java --enable-preview --source 23 --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 23  
| 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;
```



DEMO & Hands on

- FirstVectorExample
 - SecondVectorExample
 - SimplePerformanceComparison
-



Exercises PART 4

<https://github.com/Michaeli71/JAX-London-Best-of-Java-17-23>





Conclusion

Positive things



- Reliable 6-month release cadence and LTS versions will be released every 2 years
- Java becomes easier and more attractive
- Many nice improvements in syntax and APIs like switch, records, text blocks, ...
- Pattern Matching and record patterns
- JPackage and HTTP 2
- Virtual Threads & Structured Concurrency



On the negative side



- Next releases were on time, but sometimes bringing just a few new features (except 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



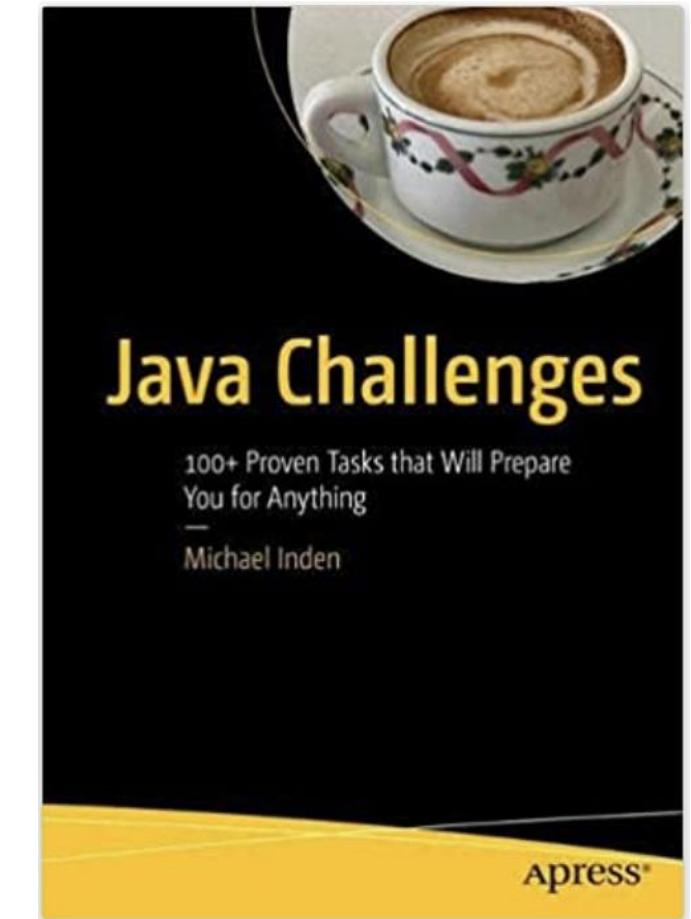
Michael Inden
Java
Die Neuerungen in
Version 17 LTS, 18 und 19

dpunkt.verlag



Michael Inden
**Der Weg zum
Java-Profi**

Konzepte und Techniken für die
professionelle Java-Entwicklung



Apress®



Questions?



Thank You