



---

# **Best of Java 15 mit Ausblick auf Java 16 und die Zukunft**

**Michael Inden**

**Freiberuflicher Consultant, Buchautor und Trainer**

# Speaker Intro



- **Michael Inden, Jahrgang 1971**
- **Diplom-Informatiker, C.v.O. Uni Oldenburg**
- **~8 ¼ Jahre SSE bei Heidelberger Druckmaschinen AG in Kiel**
- **~6 ¾ Jahre TPL, SA bei IVU Traffic Technologies AG in Aachen**
- **~4 ¼ Jahre LSA, Trainer bei Zühlke Engineering AG in Zürich**
- **~3 Jahre TL / CTO bei Direct Mail Informatics / ASMIQ in Zürich**
- **Derzeit Freiberuflicher Consultant, Autor und Trainer**
- **Regelmässiger Speaker / Trainer auf Konferenzen wie (W-)JAX, JAX London, ch.open, Oracle Code One**

E-Mail: [michael.inden@hotmail.ch](mailto:michael.inden@hotmail.ch)

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





# Agenda

# Workshop Contents

---



- **PART 1: PRESENT:** Wichtige Neuerungen aus Java 15
- **PART 2: FUTURE:** Ausblick auf Java 16 und die Zukunft



---

# PART 1: PRESENT

## Wichtige Neuheiten aus Java 15

- Syntaxerweiterungen bei switch (FINAL)
- Hilfreiche NullPointerExceptions (FINAL)
- Syntaxerweiterung Text Blocks (FINAL)
  
- Syntaxerweiterung Records (PREVIEW)
- Syntaxerweiterung bei instanceof (PREVIEW)

# Java 15 JDK Enhancement Proposals (JEP) (September 2020):

---



[\*\*Java 14: Switch Expressions\*\*](#)

[\*\*Java 14: Helpful NullPointerExceptions\*\*](#)

[\*\*JEP 339: Edwards-Curve Digital Signature Algorithm \(EdDSA\)\*\*](#)

[\*\*JEP 360: Sealed Classes \(Preview\)\*\*](#)

[\*\*JEP 371: Hidden Classes\*\*](#)

[\*\*JEP 372: Remove the Nashorn JavaScript Engine\*\*](#)

[\*\*JEP 374: Disable and Deprecate Biased Locking\*\*](#)

[\*\*JEP 375: Pattern Matching for instanceof \(Second Preview\)\*\*](#)

[\*\*JEP 377: ZGC: A Scalable Low-Latency Garbage Collector\*\*](#)

[\*\*JEP 378: Text Blocks\*\*](#)

[\*\*JEP 379: Shenandoah: A Low-Pause-Time Garbage Collector\*\*](#)

[\*\*JEP 381: Remove the Solaris and SPARC Ports\*\*](#)

[\*\*JEP 383: Foreign-Memory Access API \(Second Incubator\)\*\*](#)

[\*\*JEP 384: Records \(Second Preview\)\*\*](#)

[\*\*JEP 385: Deprecate RMI Activation for Removal\*\*](#)

---



---

# Switch Expressions



# Switch Expressions

---



- **switch-case-Konstrukt noch bei den uralten Wurzeln aus der Anfangszeit von Java**
  - **Kompromisse im Sprachdesign, die C++-Entwicklern den Umstieg erleichtern sollten.**
  - **Relikte wie das break und beim Fehlen des solchen das Fall-Through**
  - **Flüchtigkeitsfehler kamen immer wieder vor**
  - **Zudem war man beim case recht eingeschränkt bei der Angabe der Werte.**
  - **Switch Expressions als neue Syntax bringt Abhilfe**
-

# Switch Expressions: Blick zurück



- Abbildung von Wochentagen auf deren Länge ...

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

# Switch Expressions: Java 15 als Abhilfe



- Abbildung von Wochentagen auf deren Länge ... elegant mit Java 15:

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

# Switch Expressions: Java 15 als Abhilfe



- Abbildung von Wochentagen auf deren Länge ... elegant mit Java 15:

Return-  
Value

```
DayOfWeek day = DayOfWeek.FRIDAY;  
int num0fLetters = switch (day)  
{  
    case MONDAY, FRIDAY, SUNDAY -> 6;  
    case TUESDAY -> 7;  
    case THURSDAY, SATURDAY -> 8;  
    case WEDNESDAY -> 9;  
};
```

# Switch: Blick zurück ... Fallstricke der alten Syntax



- Abbildung von Monaten auf deren Namen ...

```
// ACHTUNG: Mitunter ganz übler Fehler: default mitten zwischen den cases
String monthString = "";
switch (month)
{
    case JANUARY:
        monthString = "January";
        break;
    default:
        monthString = "N/A"; // hier auch noch Fall Through
    case FEBRUARY:
        monthString = "February";
        break;
    case MARCH:
        monthString = "March";
        break;
    case JULY:
        monthString = "July";
        break;
}
System.out.println("OLD: " + month + " = " + monthString); // February (falls nicht Jan, Mar, Jul
```

# Switch Expressions: Java 15 als Abhilfe



- Abbildung von Monaten auf deren Namen ... elegant mit Java 15:

```
static String monthToName(final Month month)
{
    return switch (month)
    {
        case JANUARY -> "January";
        default -> "N/A"; // hier KEIN Fall Through
        case FEBRUARY -> "February";
        case MARCH -> "March";
        case JULY -> "JULY";
    };
}
```

# Switch Expressions mit yield für Rückgabe

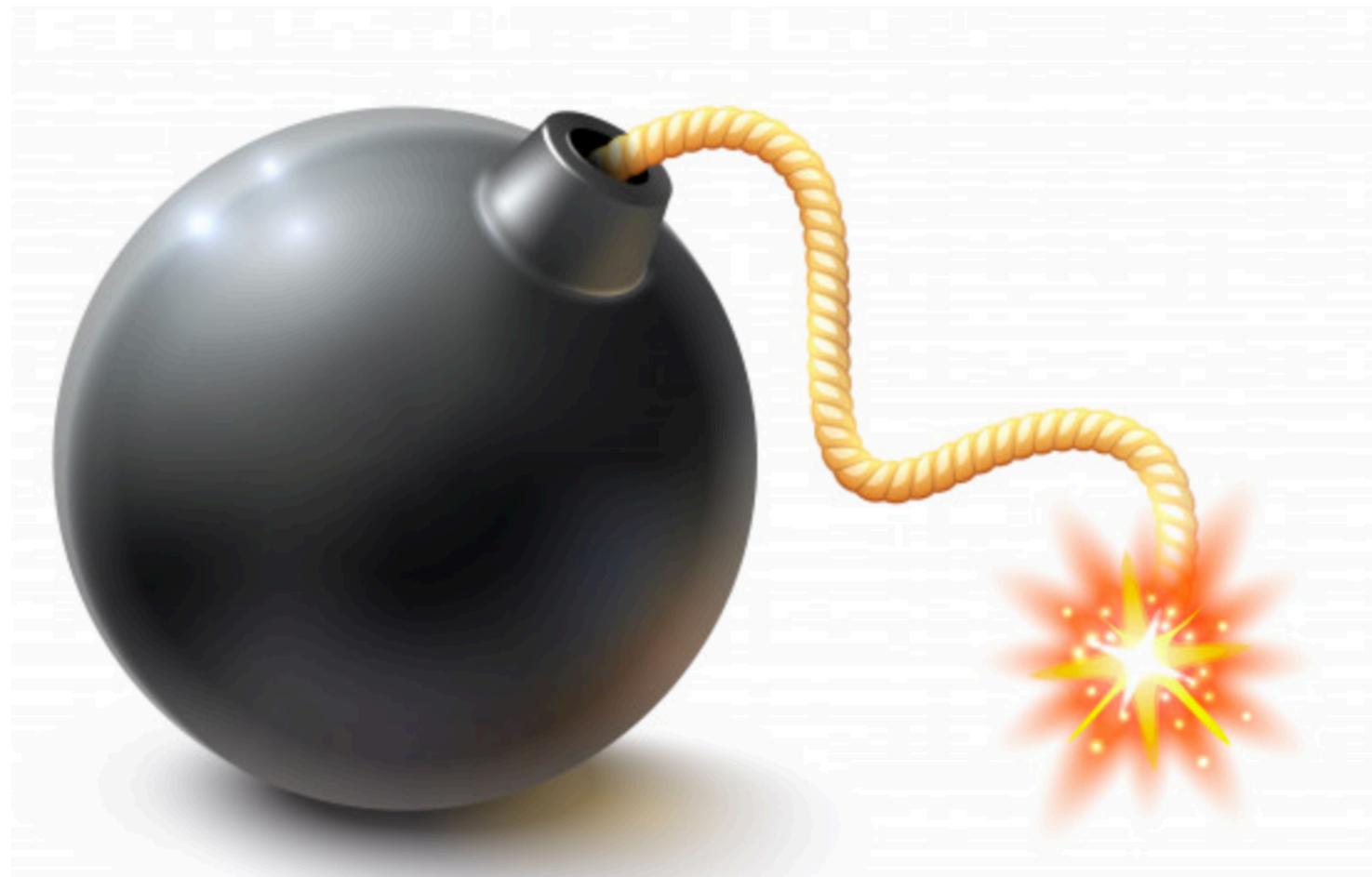


- Besonderheit, wenn weitere Aktionen statt reiner Rückgabe gewünscht:

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

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

SUNDAY is FUN DAY  
6



N P E

# Hilfreiche 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\(NPE\\_Example.java:8\)](#)

# Hilfreiche 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)

# Hilfreiche NullPointerExceptions



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

java.lang.NullPointerException: Cannot invoke
"String.lastIndexOf(String)" because "stringArray[2]" is null
at java14.NPE_Second_Example.main(NPE_Second_Example.java:10)
java.lang.NullPointerException: Cannot invoke
"java.lang.Integer.intValue()" because "value" is null
at java14.NPE_Second_Example.main(NPE_Second_Example.java:20)
```



# Text Blocks



## Text Blocks

---



- langersehnte Erweiterung, nämlich **mehrzeilige Strings** ohne mühselige Verknüpfungen definieren zu können und auf fehlerträchtiges Escaping zu verzichten.
- Erleichtert unter anderem den Umgang mit SQL-Befehlen, regulären Ausdrücken oder der Definition von JavaScript in Java-Sourcecode.
- **ALT**

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

## Text Blocks

---



- NEU

```
String javaScriptCode = """  
    void print(0bject o)  
    {  
        System.out.println(0bjects.toString(o));  
    }  
""";
```

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

# Text Blocks

---



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

## Traditional String Literals

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

```
String multiLineHtml = """  
    <html>  
        <body>  
            <p>Hello World.</p>  
        </body>  
    </html>  
""";
```

## Text Blocks

---



- NEU

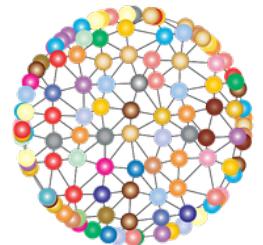
```
String jsonObj = """"
{
    name: "Mike",
    birthday: "1971-02-07",
    comment: "Text blocks are nice!"
}
"""";
```



---

# Records





**Wäre es nicht cool,  
Datencontainer-Objekte  
auf einfache Weise  
definieren zu können?**

## Erweiterung Record

---



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

- vereinfachte Form von Klassen für einfache Datencontainer
- Sehr kurze, kompakte Schreibweise
- API ergibt sich implizit aus den als Konstruktorparametern definierten Attributen

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

- Implementierungen von Accessor-Methoden sowie equals() und hashCode() automatisch und vor allem kontraktkonform

# Erweiterung Record

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

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

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

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

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

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

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

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

    // Zugriffsmethoden auf x und y
}
```

## Records für Data Transfer / Parameter Value Objects (DTOs/PVOs)



```
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 für komplexere Rückgabewerte und Parameter



```
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 für Tupel? – Ausflug Pair<T>

---



- Was ist an diesem self made Pair falsch?

```
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 + "]";
    }
}
```

## Records für Pairs und Tupel

---



```
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) {};
```

- **Extrem wenig Schreibaufwand**
  - **Sehr praktisch für Pair, Tuples usw.**
  - **Records funktionieren prima mit primitiven Typen und auch mit Generics**
-



---

Ist ja cool ... ABER:  
Wie kann ich denn  
Gültigkeitsprüfungen  
integrieren?



# Records mit Gültigkeitsprüfung

---



```
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 mit Gültigkeitsprüfung (Kurzschreibweise)

---



```
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);
        }
    }
}
```



---

# Pattern Matching bei instanceof



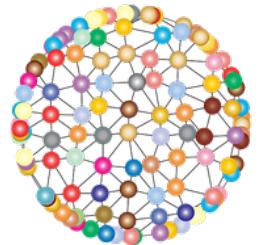
# Pattern Matching bei instanceof

---



- ALT

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



**Immer diese Casts...  
Geht es nicht einfacher?**

# Pattern Matching bei instanceof

---



- **ALT**

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

- **NEU**

```
if (obj instanceof Person person)
{
    // Hier kann man auf die Variable person direkt zugreifen
}
```

## Pattern Matching bei instanceof

---



```
Object obj2 = "Hallo Java 14";  
  
if (obj2 instanceof String str)  
{  
    // Hier kann man str nutzen  
    System.out.println("Länge: " + str.length());  
}  
else  
{  
    // Hier kein Zugriff auf str  
    System.out.println(obj.getClass());  
}
```

## Pattern Matching bei instanceof

---



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



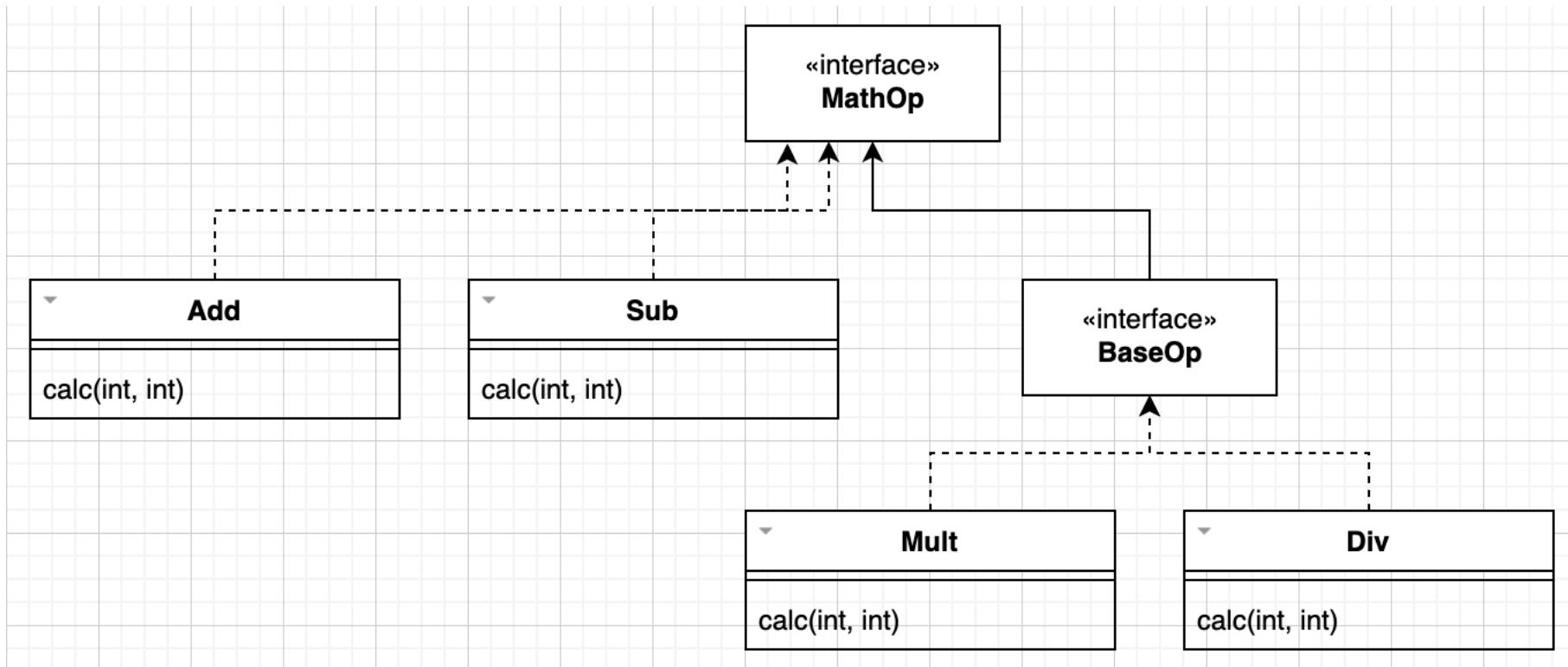
# Sealed Types



# Sealed Types



- **Vererbung steuern** und spezifizieren, welche Klassen eine Basisklasse erweitern können, also welche anderen Klassen oder Interfaces davon Subtypen bilden dürfen.



# Sealed Types – Vererbung steuern



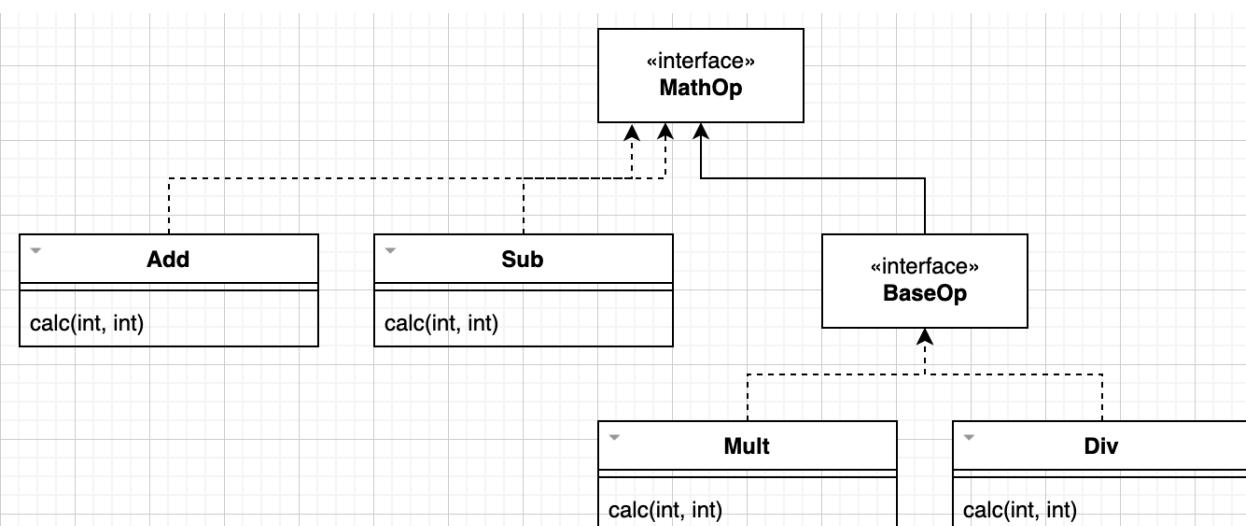
- Spezifizieren, welche anderen Klassen oder Interfaces davon Subtypen bilden dürfen.

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

// Mit non-sealed kann man innerhalb der Vererbungshierarchie Basisklassen bereitstellen

```
non-sealed class BaseOp implements MathOp // <= Basisklasse nicht versiegeln
{
    @Override
    public int calc(int x, int y)
    {
        return 0;
    }
}
...
```

Mit sealed können wir eine Vererbungshierarchie versiegeln und nur die explizit angegebenen Typen erlauben. Diese müssen sealed, non-sealed oder final sein.

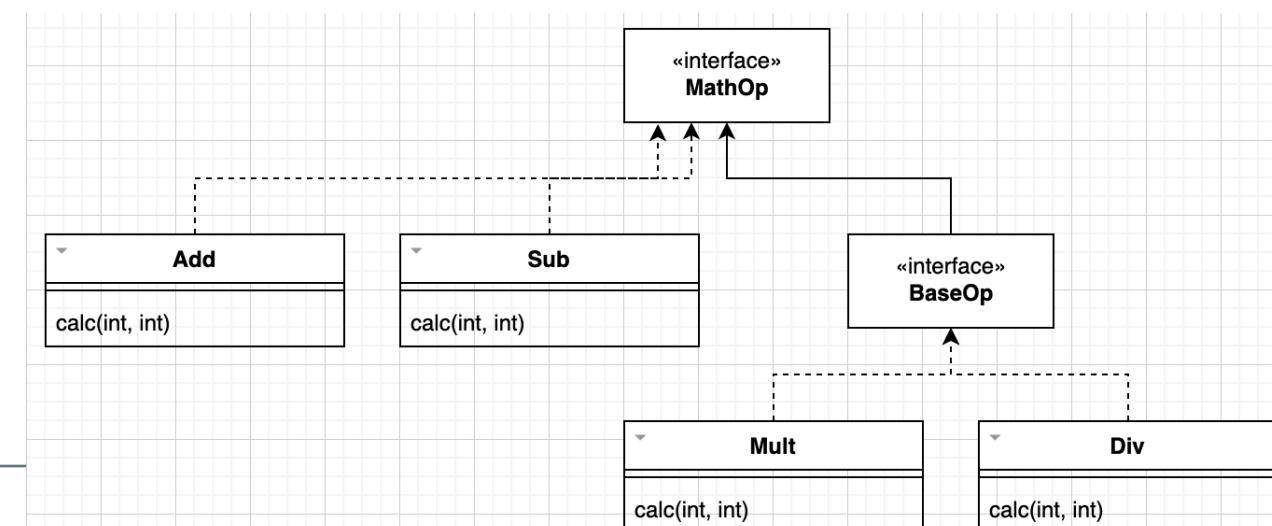


# Sealed Types



```
..  
// direkte Implementierung muss final sein  
final class Add implements MathOp  
{  
    @Override  
    public int calc(int x, int y)  
    {  
        return x + y;  
    }  
}  
  
final class Sub implements MathOp  
{  
    ...  
}  
// Ableitung aus Basisklasse muss final sein  
final class Mult extends BaseOp  
{  
}  
  
final class Div extends BaseOp  
{  
}
```

- Eine als sealed markierte Klasse muss Subklassen besitzen, die wiederum hinter permits aufgeführt werden
- Eine als non-sealed markierte Klasse kann als Basisklasse fungieren und von dieser können Klassen abgeleitet werden.
- Eine als final markierte Klasse bildet – wie gewohnt – den Endpunkt einer Ableitungshierarchie.





---

# Nashorn Java Script Engine

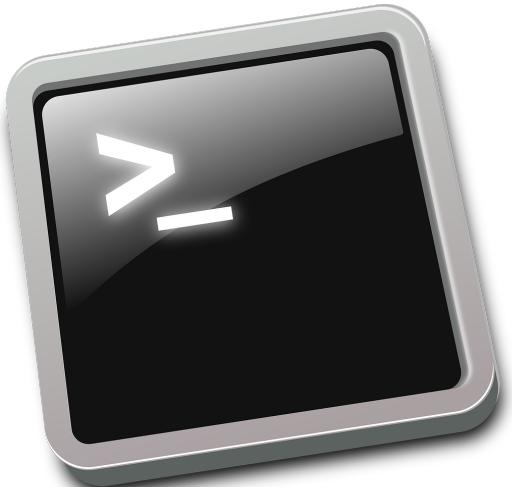
(seit Java 11 deprecated, mit Java 15 entfernt)





---

# JShell als Abhilfe?!



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

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

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

jshell> import java.time.*

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

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



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

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

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

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

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

# JShell-API



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

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

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



---

# **Part 2: FUTURE**

## **Ausblick auf Java 16 und die Zukunft**

## Zunächst ein Blick zurück

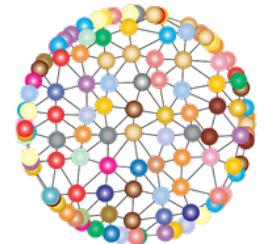


- **Java ist dieses Jahr 25 geworden**
- Lange Historie von 1995 bis 2020: Vom Newcomer zum etablierten Allzwecktool
- Viele interessante und wichtige Entwicklungen, wie Generics, Lambdas, Streams u.v.m.

## Very Long Term History

To see the bigger picture, please find below the positions of the top 10 programming languages of many years back. Please note that these are *average* positions for a period of 12 months.

Programming Language	2020	2015	2010	2005	2000	1995	1990	1985
C	1	2	2	1	1	1	1	1
Java	2	1	1	2	3	29	-	-
Python	3	5	6	7	22	13	-	-
C++	4	3	3	3	2	2	2	8



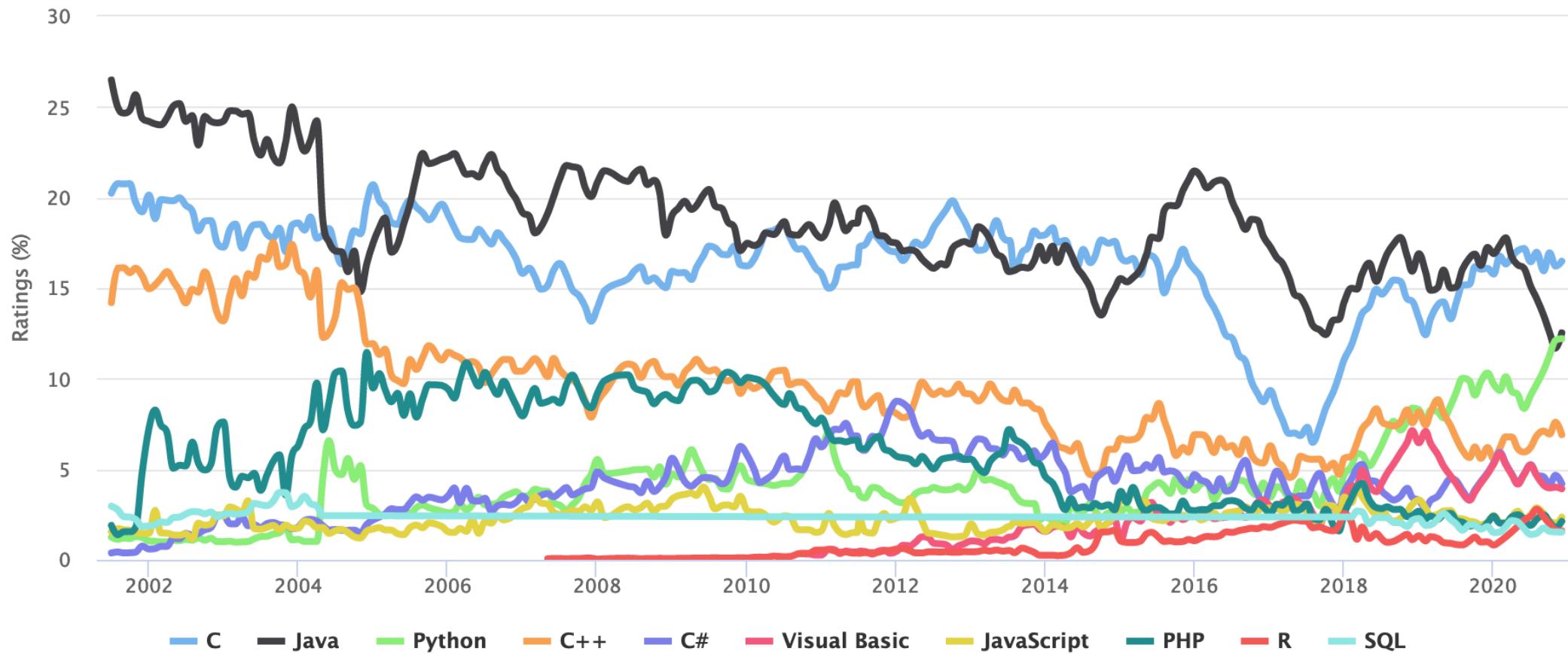
**Aber nach nun 25 Jahren:  
Wo geht die Reise hin und  
wie sieht es aktuell mit der  
Konkurrenz aus?**

# Popularitätstrends der Top 10 Programmiersprachen



TIOBE Programming Community Index

Source: [www.tiobe.com](http://www.tiobe.com)



## Zunächst ein Blick zurück



- Javas lange Erfolgsgeschichte und mehrere Male Platz 1, aber es kamen auch Rückschläge
- Anfangs gab es **Performance-Probleme**, mittlerweile ist die JVM oftmals nahe an C++ / C
- Dann wurde Java für den **Desktop** durch **Web-Applikationen** nahezu unbedeutend
- Andere Programmiersprachen haben modernere Konzepte, sind eleganter und kompakter
- **Letzte Bastion: Server-Applikationen mit Spring und Jakarta EE als stabilen Säulen**

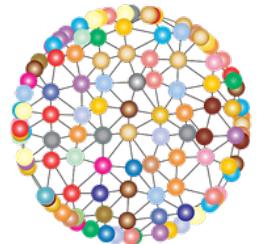
Dec 2020	Dec 2019	Change	Programming Language	Ratings	Change
1	2	▲	C	16.48%	+0.40%
2	1	▼	Java	12.53%	-4.72%
3	3		Python	12.21%	+1.90%

# Was kann man aus der Vergangenheit lernen?

---



- **Python, Kotlin, TypeScript, JavaScript usw. werden immer populärer**
- Java muss sich weiterentwickeln, C immer noch stark und Python wächst als mächtige Konkurrenz
- **Meiner Einschätzung nach Python hat deutlich mehr Potenzial als C!**
- Gute IDEs, leichte Erlernbarkeit, schnelle, motivierende Erfolge auch für Einsteiger
- Moderne Sprachkonzepte: List Comprehensions, Tupel, ein REPL (Read Eval Print Loop)
- Insgesamt fühlt sich Java doch noch etwas schwerfälliger an, neue Gegenbewegungen:
  - Switch Expressions
  - Records
  - Text Blocks



# Was ist für die zukünftigen Java-Versionen geplant?

# Java Enhancement Proposals (JEPs) für Java 16

---



- [JEP 338: Vector API \(Incubator\)](#)
  - [JEP 347: Enable C++14 Language Features](#)
  - [JEP 357: Migrate from Mercurial to Git](#)
  - [Jep 369: Migrate to GitHub](#)
  - [JEP376: ZGC: Concurrent Thread-Stack Processing](#)
  - [JEP 380: Unix-Domain Socket Channels](#)
  - [JEP 386: Alpine Linux Port](#)
  - **[JEP 387: Elastic Metaspace](#)**
  - [JEP 388: Windows/AArch64 Port](#)
  - [Jep 389: Foreign Linker API \(Incubator\)](#)
  - [JEP 390: Warnings for Value-Based Classes](#)
  - **[JEP 392: Packaging Tool](#)**
  - [JEP 393: Foreign-Memory Access API \(Third Incubator\)](#)
  - **[JEP 394: Pattern Matching for instanceof](#)**
  - **[JEP 395: Records](#)**
  - [JEP 396: Strongly Encapsulate JDK Internals by Default](#)
  - **[JEP 397: Sealed Classes \(Second Preview\)](#)**
-

# Wichtige Zukunftsprojekte im Bereich Java

---



- **Project Amber**
  - Syntax- und Sprachverbesserungen
  - Anpassungen an moderne Erwartungshaltung
- **Project Valhalla**
  - Memory-Anpassung an moderne Hardware
  - Value Types, Generics für primitive Typen
- **Project Loom**
  - einfach handhabbare, leichtgewichtige Concurrency mit gleichzeitig Support von hohem Durchsatz
  - Verbesserte Performance und Skalierbarkeit
- **Project Panama**
  - Bessere Interoperabilität mit nativem Code

# Schwierige Entscheidungen für Oracle

---



- **Möglichkeit 1:** Java 17 mit der Gefahr des halbfertigen «**Rohrkrepierers**»
  - zwar einige sinnvolle Syntax-Neuerungen sowie
  - verschiedene andere Verbesserungen unter anderem auch bei der Performance
  - allerdings sind noch diverse wünschenswerte Dinge in der Pipeline.
- **Möglichkeit 2:** Abkehr vom bisherigen Release-Zyklus und **LTS-Verschiebung auf Java 19**
  - Java 17 wird doch kein LTS
  - man entwickelt noch beispielsweise bis zum September 2022 und Java 19
  - ermöglicht die Fertigstellung der Funktionalitäten aus den Projekten Amber, Loom, Panama und Valhalla



- **Möglichkeit 1:** Java 17 mit der Gefahr des halbfertigen «**Rohrkrepierers**»
- **Möglichkeit 2:** Abkehr vom bisherigen Release-Zyklus und **LTS-Verschiebung auf Java 19**
- **Möglichkeit 1:**
  - **vielleicht Unzufriedenheit und**
  - **wäre unschön, auf wichtige Funktionalitäten 3 Jahre bis zum nächsten LTS zu warten**
- **Möglichkeit 2:**
  - **zwar ein weiteres Jahr Warten, aber**
  - **das LTS könnte dann ein richtig grosser Wurf werden, der die Community begeistert**
  - **und Java fit für die Zukunft macht**



# Fazit

# Positives in den letzten Java-Versionen

---



- die Sprache wird moderner
- eine paar Dinge aus Project COIN (Syntax)
- Switch / Records / Text Blocks
- diverse praktische Erweiterungen in den APIs
- HTTP/2-Support
- Modularisierung mit Project JIGSAW  
aber leider (immer noch) kein wirklich gutes Tooling

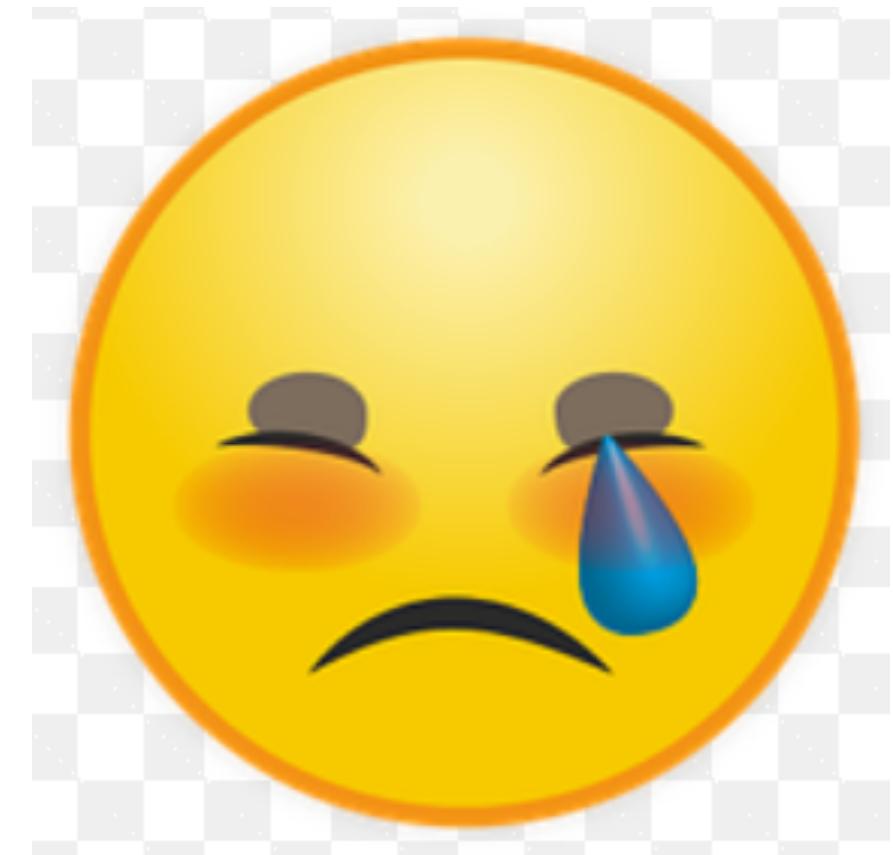


# Negatives in den letzten Java-Versionen

---



- **Mehrmalige Verschiebung von Java 9**  
**Sept. 2016 => März 2017 => Juli 2017 => Sept. 2017**
- **Folge-Release waren zwar immer pünktlich, aber (außer Java 14) ziemlich dünn bezüglich substanzialer Neuerungen**
- **Leider oft wieder weniger als geplant**
  - keine Versionierung bei JIGSAW
  - kein JSON-Support
  - statt Collection-Literalen nur Collection Factory Methods
  - Statt ZIP nur TEE (ing)-Kollektor





---

# Questions?



# Thank You