



---

# Best of Modern Java 21 – 25

## Meine Lieblingsfeatures

<https://github.com/Michaeli71/Best-Of-Modern-Java-21-25-My-Favorite-Features>



Michael Inden  
Head of Development, freiberuflicher Buchautor und Trainer

# Speaker Intro



[michael\\_inden@hotmail.com](mailto:michael_inden@hotmail.com)



- **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**
- **~3 ½ Jahre Head of Development bei Adcubum in Zürich**
- **Freiberuflicher Consultant, Trainer und Konferenz-Speaker**
- **Autor und Gutachter bei dpunkt.verlag, O'Reilly und APress**



<https://github.com/Michaeli71/Best-Of-Modern-Java-21-25-My-Favorite-Features>



# Agenda

# All Time Favorites + Meine Top 10 aus Java 21 LTS bis 25 LTS (chronologisch)



## All Time Favorites:

- Switch Expressions / Text Blocks / Records / Pattern Matching bei instanceof (Java 17)

## Meine Top 10 aus Java 21 LTS und Java 25 LTS:

1. Record Patterns (Java 21)
2. Pattern Matching bei switch (Java 21)
3. Virtual Threads (Java 21)
4. *Structured Concurrency (Preview) (Java 21 & 25)*
5. Unnamed Variables & Patterns (Java 22)
6. Launch Multi-File Source-Code Programs (Java 22)
7. Markdown Comments (Java 23)
8. Stream Gatherers (Java 24)
9. Flexible Constructor Bodies (Java 25)
10. Compacte Source Files and Instance Main Methods (Java 25)





---

# All Time Favorites

Switch Expressions

Text Blocks

Records

Pattern Matching bei instanceof

---

# Switch Expressions



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

Return-  
Value

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

- Elegantere Schreibweise beim case:

- Neben dem offensichtlichen Pfeil statt des Doppelpunkts
- auch mehrere Werte
- Kein break nötig, auch kein Fall-Through
- switch kann nun einen Wert zurückgeben, vermeidet künstliche Hilfsvariablen

# Switch Expressions

---



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

```
static String monthToString(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";
    };
}
```

# Text Blocks

---



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

# Enhancement Record

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



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

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

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

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

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

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

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

    // Zugriffsmethoden auf x und y
}
```

# Records 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 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 Pairs, Tuples usw.**
- **Records funktionieren prima mit primitiven Typen und auch mit Generics**
- **Implementierungen von Accessor-Methoden sowie equals() und hashCode() automatisch und vor allem kontraktkonform, ebenso toString()**

# 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

---



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



---

# Meine Top 10

# Java 21 LTS bis 25 LTS





---

# Platz 1: Record Patterns





- Basis für diesen JEP und seine Vorgänger ist das Pattern Matching für instanceof aus 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));
    }
}
```

- Zwar ist das oftmals schon praktisch, aber man muss noch teilweise umständlich auf die einzelnen Bestandteile zugreifen.



- Ziel ist es, Records deklarativ in ihre Bestandteile zerlegen und auf diese zugreifen zu können.

```
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));
}
```



- Record Patterns können auch verschachtelt werden, um eine deklarative, leistungsfähige und kombinierbare Form der Datennavigation und -verarbeitung zu ermöglichen.

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

```
enum Color { RED, GREEN, BLUE }
```

```
record ColoredPoint(Point point, Color color) {}
```

```
record Rectangle(ColoredPoint upperLeft, ColoredPoint lowerRight) {}
```

```
static void printColorOfUpperLeftPoint(Rectangle rect)
```

```
{
```

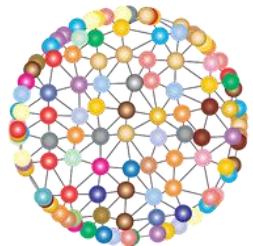
```
    if (rect instanceof Rectangle(ColoredPoint(point, color),  
                                ColoredPoint lowerRight))
```

```
{
```

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

```
}
```

```
}
```



**Wo können Record Patterns  
ihre Stärke ausspielen?**



- Nehmen wir einmal folgende Records als Datenmodell an:

```
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) {  
}
```



- In Legacy-Code findet man tief verschachtelte Abfragen wie diese:

```
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;
}
```



- Mit verschachtelten Record Patterns schreiben wir die obige Verschachtelung der ifs sicherer, eleganter und verständlicher wie folgt:

```
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 405/440: Record Patterns



```
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;
}
```

- **Die Prüfung mit instanceof schlägt automatisch fehl, falls eine der Record-Komponenten null ist, also hier Person oder City (destination).**
- **Lediglich die Attribute werden so nicht abgesichert und sind ggf. auf null zu prüfen.**
- **Wenn man sich jedoch den guten Stil angewöhnt, null als Wert von Parametern bei Aufrufen zu vermeiden, kann man sogar darauf verzichten.**



---

# Platz 2: Pattern Matching bei switch





- **Problemfeld:** Es können mehrere Patterns auf eine Eingabe matchen.

```
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 when 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 -> {}  
    }  
}
```

- Dasjenige, das «am allgemeingültigsten» passt, wird dominierendes Pattern genannt.
- Im Beispiel dominiert das kürzere Pattern `String s` das längere davor angegebene.

# JEP 441: Pattern Matching bei switch: Dominanzprüfung



- Problematisch wird es, wenn die Reihenfolge der Muster umgekehrt wird:

```
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 line 'case String str && str.length() > 5 -> System.out.println(str.strip());'. 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

# JEP 441: Pattern Matching bei switch mit 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");
    }
}
```



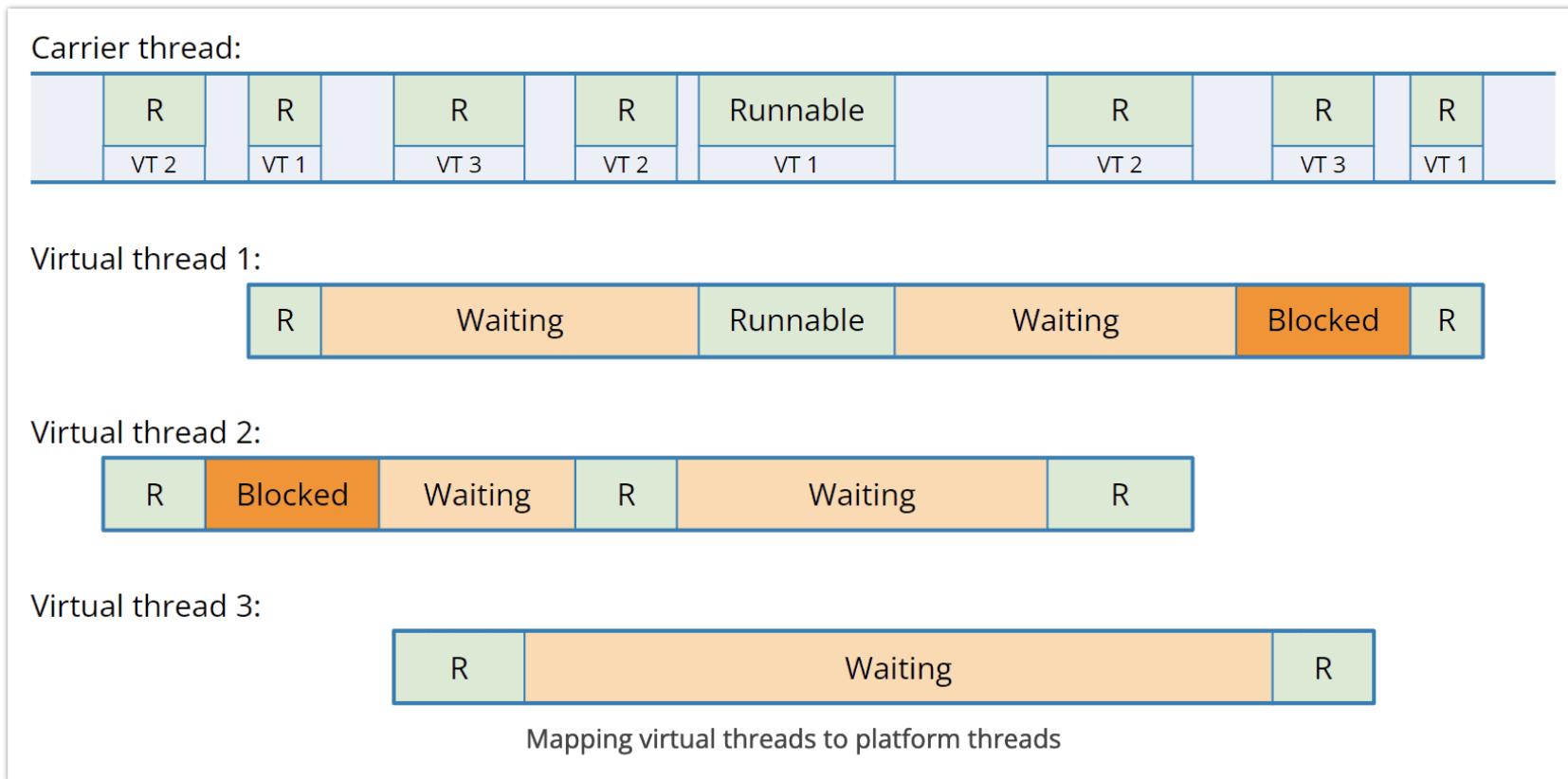
---

# Platz 3: Virtual Threads





- Dieser JEP führt das Konzept leichtgewichtiger virtueller Threads ein.
- Virtuelle Threads «fühlen» sich wie normale Threads an, sind auch vom Typ `java.lang.Thread`, werden aber nicht 1:1 auf Betriebssystem-Threads abgebildet.





- Was ist das Problem an blockierendem I/O? => miserable Serverauslastung

## Concurrency Issues

Why is it bad to block?

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



- Virtuelle Threads erlauben im Bereich der Serverprogrammierung wieder mit jeweils einem separaten Thread pro Request zu arbeiten. Hilfreich weil in der Regel viele Client-Requests blockierendes I/O wie das Abrufen von Ressourcen durchführen.

# JEP 444: Virtual Threads

---



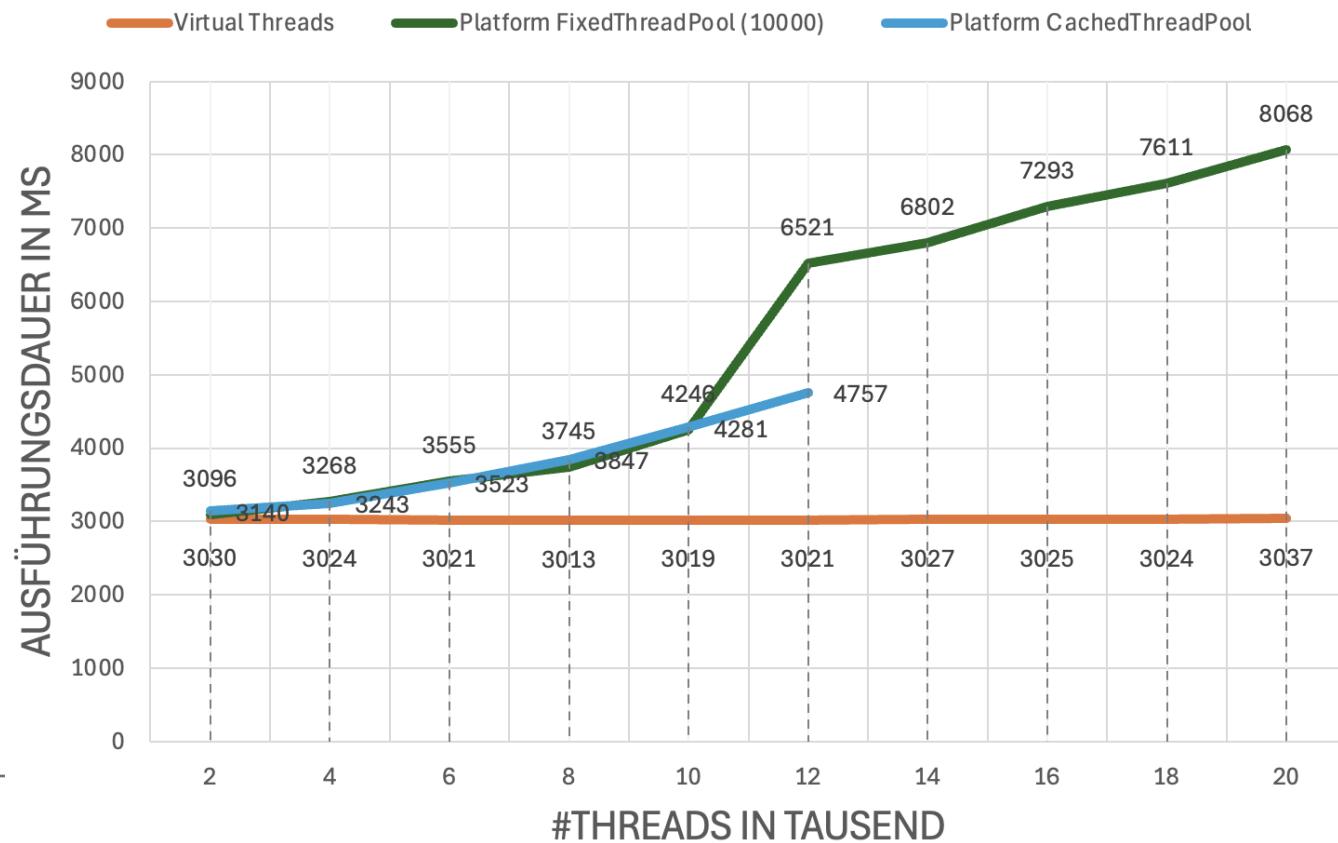
```
public static void main(String[] args)
{
    System.out.println("Start");

    try (var executor = Executors.newVirtualThreadPerTaskExecutor())
    {
        for (int i = 0; i < 10_000_000; i++)
        {
            final int pos = i;
            executor.submit(() -> {
                Thread.sleep(Duration.ofSeconds(5));
                return pos;
            });
        }
    }
    // executor.close() is called implicitly,
    // and waits until all tasks are completed
    System.out.println("End");
}
```



- Performance-Vergleich in Tausenderschritten für Plattform- und virtuelle Threads
- Die Threads warten jeweils einige Sekunden, um den Zugriff auf eine externe Schnittstelle zu simulieren. Am Ende wird die Gesamtzeit gemessen.

Vergleich Platform und Virtual Threads





---

# Platz 4: Structured Concurrency (Preview)





- Dieser JEP bringt die Structured Concurrency als Vereinfachung von Multithreading.
- Wenn eine Aufgabe aus mehreren Teilaufgaben besteht, die parallel verarbeitet werden können, ermöglicht Structured Concurrency, dies auf besonders lesbare und wartbare Weise umzusetzen.
- Betrachten wir das Ermitteln eines Users und dessen Bestellungen anhand einer User-ID:

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

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

- Beide Aktionen könnten parallel ablaufen.
- Wenn es in einer von beiden zu einer Exception kommt, wird die Ausführung abgebrochen.



```
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);  
}
```

- Weil die Teilaufgaben parallel ausgeführt werden, können sie unabhängig voneinander erfolgreich sein oder fehlschlagen. Die Fehlerbehandlung kann recht kompliziert werden.
- Oftmals möchte man beispielsweise nicht, dass das zweite get() aufgerufen wird, wenn bereits bei der Abarbeitung der Methode findUser() eine Exception aufgetreten ist.



- Umsetzung mit Structurd Concurrency in Form der Klasse StructuredTaskScope:

```
static Response handleJava25(Long userId) throws InterruptedException
{
    //var joiner = StructuredTaskScope.Joiner.awaitAllSuccessfulOrThrow();
    try (var scope = StructuredTaskScope.open())
    {
        var userSubtask = scope.fork(() -> findUser(userId));
        var orderSubtask = scope.fork(() -> fetchOrder(userId));

        scope.join();      // Join both forks

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

- Konkurrierende Teilaufgaben mit fork() abspalten und mit blockierendem Aufruf von join() Ergebnisse einsammeln
- join() wartet, bis alle Teilaufgaben abgearbeitet sind oder ein Fehler auftrat.



Die Klasse `StructuredTaskScope` besitzt zwei Spezialisierungen:

- `awaitAllSuccessfulOrThrow()` – fängt die erste Exception ab und beendet den `StructuredTaskScope`. Diese Strategie ist dafür gedacht, wenn die Ergebnisse aller Teilaufgaben benötigt werden ("`invoke all`"); wenn jedoch eine Teilaufgabe fehlschlägt, werden die Ergebnisse der anderen, noch nicht abgeschlossenen Teilaufgaben nicht mehr benötigt.
- `anySuccessfulResultOrThrow()` – ermittelt das erste eintreffende Ergebnis und beendet dann den `StructuredTaskScope`. Das hilft, wenn bereits das Ergebnis einer beliebigen Teilaufgabe ausreicht ("`invoke any`") und es nicht notwendig ist, auf die Ergebnisse anderer, nicht abgeschlossener Aufgaben zu warten.



- Umsetzung mit Structurd Concurrency in Form der Klasse StructuredTaskScope:

```
var joiner =
    StructuredTaskScope.Joiner.<NetworkConnection>anySuccessfulResultOrThrow();
try (var scope = StructuredTaskScope.open(joiner))
{
    var result1 = scope.fork(() -> tryToGetWifi());
    var result2 = scope.fork(() -> tryToGet5g());
    var result3 = scope.fork(() -> tryToGet4g());
    var result4 = scope.fork(() -> tryToGet3g());

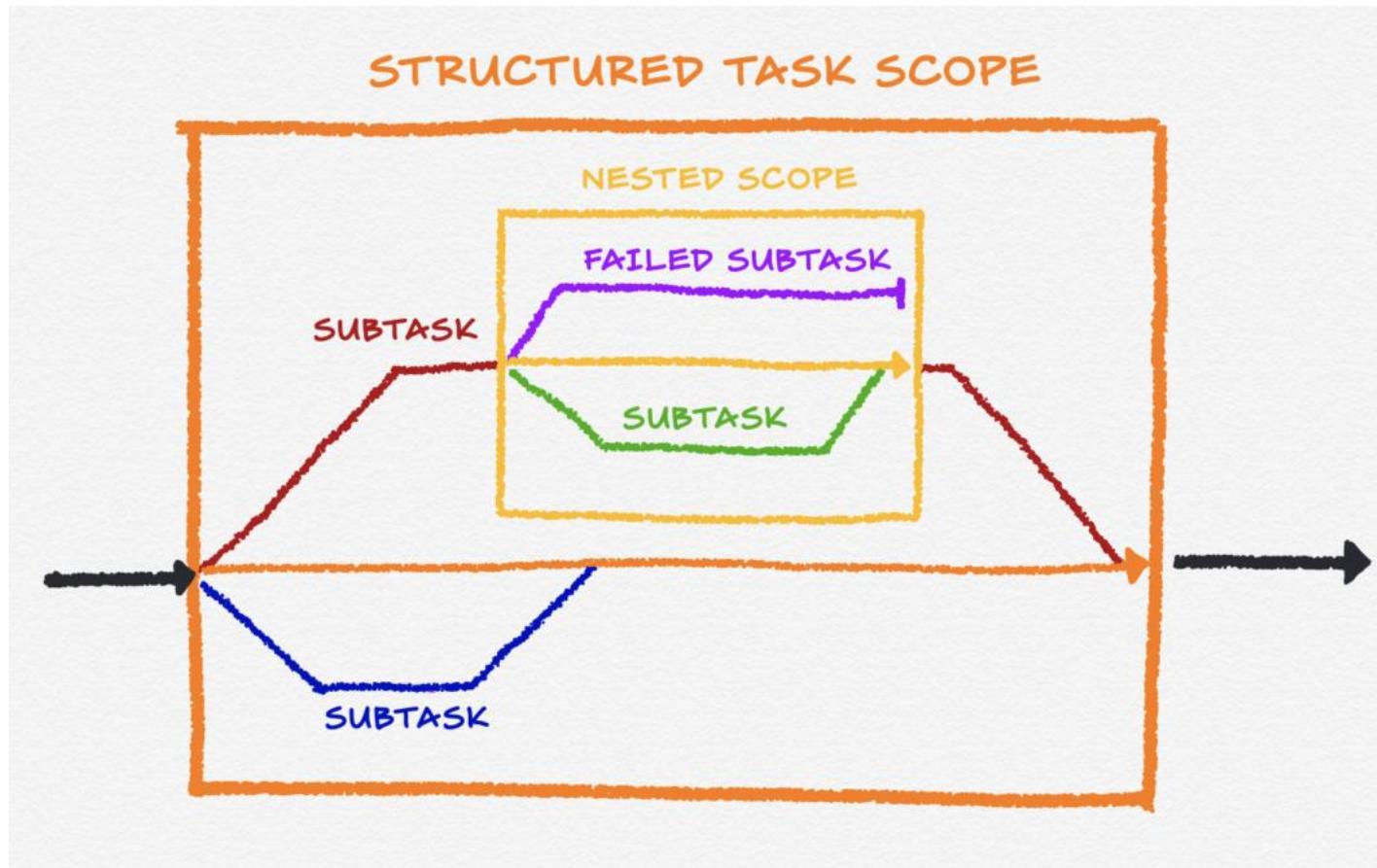
    NetworkConnection result = scope.join();

    System.out.println("Wifi " + result1.state() + "/5G " + result2.state() +
        "/4G " + result3.state() + "/3G " + result4.state());
    System.out.println("found connection: " + result);
}
```

- Konkurrierende Teilaufgaben mit fork() abspalten und mit blockierendem Aufruf von join() das zuerst vorliegende Ergebnis einsammeln
- join() wartet, bis eine Teilaufgabe erfolgreich ist
- state() liefert SUCCESS (erster), UNAVAILABLE (andere) oder FAILED (Exception)



- Structured Concurrency kann man auch verschachteln:



# JEP 505: Structured Concurrency



```
Proposal performCityTripProposals(String city,
                                  LocalDate startDate) throws InterruptedException
{
    try (var scope = StructuredTaskScope.open())
    {
        var hotelSubtask = scope.fork(() -> findHotels(city, startDate));
        var carRentalSubtask = scope.fork(() -> fetchCarRentals(city,
                                              startDate));

        scope.join();

        // Here, both forks have succeeded, so compose their results
        return new Proposal(hotelSubtask.get(), carRentalSubtask.get());
    }
}
```

Hier werden jeweils neue,  
verschachtelte StructuredTaskScope  
mit eigenem Joiner erstellt



- **Finde Vorschläge für ein Hotel**

```
private static String findHotels(String city, LocalDate startDate) throws InterruptedException {
    var anySuccessful = StructuredTaskScope.Joiner.<String>anySuccessfulResultOrThrow();
    try (var scope = StructuredTaskScope.open(anySuccessful)) {

        var proposal1 = scope.fork(() -> findProposalTrivago(city, startDate));
        var proposal2 = scope.fork(() -> findProposalBooking(city, startDate));
        var proposal3 = scope.fork(() -> findProposalCheck24(city, startDate));

        // Hier kommt der vielleicht Wunsch nach FirstNSuccessful auf,
        // falls man doch mehrere Vorschläge möchte
        return scope.join();
    }
}
```

Innerer StructuredTaskScope  
mit eigenem Joiner

- **Eigene Joiner lassen sich ziemlich einfach realisieren und nutzen:**

```
var twoSuccessful = new OwnJoinersExample.FirstNSuccessful<String>(2);
```



---

# Platz 5: Unnamed Variables and Patterns





- Was beobachtet man bei der Verwendung von Record Patterns?

```
Point point = new Point(3, 4);
var cp = new ColoredPoint(point, Color.GREEN);

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

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

- Nur ein paar Bestandteile/Attribute im Record Pattern sind wirklich von Interesse!

# JEP 456: Unnamed Variables & Patterns

---



- Und was ist mit ähnlichen Situationen in "normalem" 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  
}
```

- Einige Variablen sind im nachfolgenden Code unbenutzt



- JEP 456 finalisiert den Vorgänger JEP 443 und ermöglicht es, Variablen oder Teile innerhalb von Record Patterns durch ein `_` als unbenutzt und unbrauchbar zu markieren. Zur Erinnerung sei erwähnt, dass es die folgenden drei Varianten gibt:
  1. **Unnamed variable** – erlaubt die Verwendung von `_` für die Benennung oder Markierung von nicht verwendeten Variablen.
  2. **Unnamed pattern variable** – erlaubt das Weglassen des Bezeichners, der normalerweise dem Typ (oder var) in einem Record Pattern folgen würde.
  3. **Unnamed pattern** – erlaubt es, den Typ und den Namen einer Komponente eines Record Patterns vollständig wegzulassen (und durch ein `_` zu ersetzen).



- **Unnamed variable**

```
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;
```

- **Interessanterweise können auch mehrere unbenannte Variablen im selben Scope verwendet werden, was (neben einfachen Lambdas) vor allem für Record Patterns und in switch von Interesse ist.**

# JEP 456: Unnamed Variables & Patterns

---



- **Unnamed pattern variable**

```
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());
}
```

- **Gleiches gilt für case ColoredPoint(Point point, Color \_)**
-

# JEP 456: Unnamed Variables & Patterns



- **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);
}
```

- **Gleiches gilt für case.**

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



# JEP 456: Unnamed Variables & Patterns

---



```
static boolean checkFirstNameTravellingTimeAndZipCode(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;
}
```



---

# Platz 6: Launch Multi-File Source-Code Programs



# JEP 458: Launch Multi-File Source-Code Programs: Direct Compilation



- Erlaubt es, Java-Applikationen, die nur aus einer Datei bestehen, direkt in einem Rutsch zu kompilieren und auszuführen.
- erspart Arbeit und man muss nichts vom Bytecode und .class -Dateien wissen
- vor allem für die Ausführung von kleineren Java-Dateien als Skript und für den Einstieg in Java hilfreich

```
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

- Aber bis einschließlich Java 21 LTS nur für eine einzelne Java-Datei!

# 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!<<
```



---

# Platz 7: Markdown Comments



# JEP 467: Markdown Documentation Comments



- Man kann Sourcecode nun mit Markdown dokumentieren
- Diese wird auch in IntelliJ direkt auf dem kommentierten Programmelement (Klasse/Methode) angezeigt.

```
/// 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`.

 Java23Examples  



- Textpassagen sollen mitunter hervorgehoben werden

- *kursiv* (\*...\* or \_...\_) oder **fett** (\*\*...\*\*)
- Schriftart durch einen Backtick (`...) in Schreibmaschinenschrift ändern. Fett und/oder kursiv sind dabei ebenfalls möglich.
- Mehrzeilige Quellcode-Ausschnitte mit (```...```) im Kommentar.

```
/// **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;
}
```

Java23Examples



```
/// - 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	$\alpha$ (alpha)
b	$\beta$ (beta)
c	$\gamma$ (gamma) // $\Gamma$
...	...
z	$\omega$ (omega)



---

# Platz 8: Stream Gatherers





- Nehmen wir an, wir wollten alle Duplikate aus einem Stream herausfiltern und dazu ein Kriterium angeben:

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

- Mit einem Trick kann man das herkömmlich wie folgt lösen:

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



- **Gruppierung der Daten eines Streams in Abschnitte fixer Größe:**

```
var result = Stream.iterate(0, i -> i + 1).  
    windowFixed(4).      // Hypothetisch  
    limit(3).  
    toList();
```

// result ==> [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]

- Im Laufe der Jahre sind diverse Intermediate Operations wie etwa `distinctBy()` oder `windowFixed()` als Ergänzung für das Stream-API vorgeschlagen worden.
- Oftmals sind diese in spezifischen Kontexten sinnvoll, allerdings würden diese das Stream-API ziemlich aufblähen und den Einstieg in das (ohnehin schon umfangreiche) API (weiter) erschweren.



- Java 24 bringt analog zu `collect(Collector)` für Terminal Operations nun eine Methode `gather(Gatherer)` zur Bereitstellung einer benutzerdefinierten Intermediate Operation.
- Dazu dient das Interface `java.util.stream.Gatherer`, das anfangs jedoch ein wenig herausfordernd selbst zu implementieren wirken kann.
- Praktischerweise gibt es in der Utility-Klasse `java.util.stream.Gatherers` diverse vordefinierte Gatherer wie:
  - `windowFixed()`
  - `windowSliding()`
  - `fold()`
  - `scan()`

## JEP 485: Stream Gatherers – windowFixed()



- Um einen Stream in kleinere Bestandteile fixer Größe ohne Überlappung zu unterteilen, dient `windowFixed()`.

```
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);  
}
```

- Teilweise enthält der Datenbestand nicht genug Elemente. Das führt dazu, dass der letzte Teilbereich dann einfach weniger Elemente beinhaltet.

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

# JEP 485: Stream Gatherers – windowSliding()



- Um einen Stream in kleinere Bestandteile fixer Größe mit Überlappung zu unterteilen, dient `windowSliding()`.

```
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);  
}
```

- Teilweise enthält der Datenbestand nicht genug Elemente. Das führt dazu, dass der letzte Teilbereich dann einfach weniger Elemente beinhaltet (hier nicht gezeigt):

```
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 485: Stream Gatherers – fold()

---



- Um die Werte eines Streams miteinander zu verknüpfen, dient die Methode `fold()`. Ähnlich wie bei `reduce()` gibt man einen Startwert und eine Berechnungsvorschrift an:

```
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);  
}
```

- Um einen Wert auszulesen, dient wiederum der Aufruf von `findFirst()`, das liefert einen `Optional<T>`:

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



- Was passiert, wenn wir zur Kombination der Werte auch Aktionen ausführen wollen, die nicht für die Typen der Werte, hier int, definiert sind?
- Als Beispiel wird ein Zahlenwert in einen String gewandelt und dieser gemäß dem Zahlenwert mit repeat() wiederholt:

```
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);
}
```

- Als Ausgabe ergibt sich die Folgende:

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



- **Sollen die Elemente eines Streams zu neuen Kombinationen zusammengeführt werden, sodass jeweils immer ein Element dazukommt, dann ist dies die Aufgabe von scan().**
- **Die Methode arbeitet ähnlich wie fold(), das die Werte zu einem Ergebnis kombiniert. Bei scan() wird dagegen bei jeder Kombination der Werte ein neues Ergebnis produziert:**

```
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);
}
```

- **Die Ausgabe ist Folgende:**

```
repeat with scan(): [1, 122, 122333, 1223334444, 122333444455555, 122333444455555666666,
1223334444555556666667777777]
```

# JEP 485: Stream Gatherers – Praxisbeispiel: Präfixsummen



```
private static void cummulatedExpensesPrefixSum() {  
    var expenses = Stream.of(10, 15, 20, 30, 40, 150, 75,  
        10, 20, 30, 70, 50, 120, 100,  
        10, 50, 20, 25, 50, 150, 110).  
        gather(Gatherers.scan(() -> 0L, (result, number) -> result + number)).  
        toList();  
    IO.println("cummulatedExpenses: " + expenses);  
  
    // Abfragen O(1), anstatt immer wieder alle Werte für jede Abfrage erneut summieren  
    // zu müssen  
    var week1 = expenses.get(6);  
    var week2 = expenses.get(13) - expenses.get(6);  
    var week3 = expenses.get(20) - expenses.get(13);  
    IO.println("week1: " + week1);  
    IO.println("week2: " + week2);  
    IO.println("week3: " + week3);  
}
```

```
cummulatedExpenses: [10, 25, 45, 75, 115, 265, 340, 350, 370, 400, 470, 520, 640, 740,  
    750, 800, 820, 845, 895, 1045, 1155]
```

```
week1: 340  
week2: 400  
week3: 415
```



- **Aufgabenstellung: Gruppiere Ausgaben wochenweise und berechne darin jeweils die kummulierte Summe**
- **Dazu Kombination der Gatherer windowFixed() und scan():**

```
private static void cummulatedExpensesByWeek() {
    var expensesByWeek = Stream.of(10, 15, 20, 30, 40, 150, 75,
        10, 20, 30, 70, 50, 120, 100,
        10, 50, 20, 25, 50, 150, 110).
        gather(Gatherers.windowFixed(7)).map(window ->
            window.stream().gather(Gatherers.scan(() -> 0,
                (result, number) -> result + number)).toList().
        toList());
    IO.println("cummulatedExpenses by week: " + expensesByWeek);
}
```

- **Gatherer windowFixed() liefert eine Liste, die man wieder in Stream überführt**
- **Ausgabe:**

```
cummulatedExpenses by week: [[10, 25, 45, 75, 115, 265, 340],
    [10, 30, 60, 130, 180, 300, 400],
    [10, 60, 80, 105, 155, 305, 415]]
```



- **Herkömmlicher Ansatz für Parallelität:**

```
var result3 = IntStream.rangeClosed(0, 499).  
    boxed().  
    parallel().  
    map(LookupService::lookup).  
    toList();
```

- => Wartezeit von ca. 30 Sekunden, wenn `lookup()` ca. 1 Sekunde benötigt
- Vorteile durch Steuerung der Parallelität, Umstellung auf Virtual Threads reduziert Wartezeit auf ca. 2 Sekunden:

```
var result4 = IntStream.rangeClosed(0, 499).  
    boxed().  
    gather(Gatherers.mapConcurrent(250, LookupService::lookup)).  
    toList();
```

- Man mit der Anzahl der Parallelität experimentieren m 500 reduziert auf 1 Sekunde



---

# Platz 9: Flexible Constructor Bodies



# JEP 513: 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");
    }
}
```

- Wenn wir auf den Sourcecode schauen, so wirkt dieser nicht gerade elegant – die Prüfung erfolgt auch erst nach Konstruktion der Basisklasse ...
- Dadurch sind potenziell schon unnötige Aufrufe sowie Objektkonstruktionen erfolgt.
- Manchmal bietet es sich an, den oder die Konstruktorparameter zu validieren, bevor man diese (ansonsten ungeprüft) bei einem Aufruf des Basisklassenkonstruktors übergibt.

# JEP 513: Flexible Constructor Bodies



- **Herkömmliche Abhilfe: statische Hilfsmethode**

```
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;
    }
}
```



- Die Argumentprüfung wird deutlich besser lesbar und verständlich, wenn die Validierungslogik direkt im Konstruktor noch vor dem Aufruf von `super()` geschieht.
- Durch diese Neuerung lassen sich nun in einem Konstruktor dessen Argumente validieren, bevor dort der Konstruktor der Superklasse aufgerufen wird:

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

        super(value);
    }
}
```

# JEP 513: Flexible Constructor Bodies



- Manchmal bietet es sich an, Aktionen vor dem Aufruf von `this()` auszuführen, um mehrfache Aktionen zu vermeiden, hier `split()`:

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

# JEP 513: Flexible Constructor Bodies



- Mit der neuen Syntax können wir die Aktionen aus dem Aufruf von `this()` herausziehen und insbesondere das `split()` auch nur einmal aufrufen.
- Möchte man das für die Logik irrelevante Stripping eleganter und den Konstruktor leichter lesbar gestalten, so implementiert man noch eine zusätzliche Hilfsmethode `parseInt()`:

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



- Im Zusammenhang mit Vererbung kann es gelegentlich zu Überraschungen kommen, wenn in Konstruktoren Methoden aufgerufen werden, die in Unterklassen überschrieben werden.

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

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

        logValues();
    }

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

# JEP 513: Flexible Constructor Bodies – Neu 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**

Während der Verarbeitung des Basisklassenkonstruktors ist das **Attribut subClassInfo noch nicht zugewiesen**, da der Aufruf von `super()` VOR der Zuweisung an die Variable erfolgt. Dies führt zu der oben genannten, aber oftmals unerwarteten Ausgabe.

# JEP 513: Flexible Constructor Bodies – Neu seit 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

Während der Verarbeitung des Basisklassenkonstruktors ist das **Attribut subClassInfo jetzt bereits zugewiesen**, da der Aufruf von super() NACH der Zuweisung an die Variable erfolgt. Dies führt zu der oben gezeigten und **erwarteten Ausgabe**.



---

# Platz 10: Compact Source Files and Instance Main Methods





- Vielleicht ist es auch bei Ihnen schon eine Weile her, dass Sie Java gelernt haben.
- Wenn Sie Programmieranfängern Java beibringen wollen, wissen Sie, wie schwierig der Einstieg ist.
- Aus der Sicht von Anfängern besitzt Java eine wirklich steile Lernkurve.
- Es fängt schon mit dem einfachsten Hello-World an.

**package** preview;

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

- Mit Python reduziert auf das Wesentliche:

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

Sie als Trainer weisen auf die folgenden Fakten für Anfänger hin:

1. Vergessen Sie package, public , class, static, void, etc. die sind momentan noch unwichtig ...
2. Schauen Sie sich nur die Zeile mit System.out.println() an
3. Oh ja, System.out ist eine Instanz einer Klasse, aber auch das ist jetzt nicht wichtig.

Ziemlich viele verwirrende Wörter und Konzepte, die von der eigentlichen Aufgabe ablenken.

# JEP 495: Simple Source Files and Instance Main Methods

---



- **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!");  
}
```



## Weitere Möglichkeiten

```
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 <---
```



- Bereits mit Java 23 zwei maßgebliche Neuerungen hinzugefügt und in Java 25 LTS finalisiert:
  - **Interaktion mit der Konsole:** Implizit deklarierte Klassen importieren automatisch die drei statischen Methoden `print()`, `println()` und `readln()`, die in der Klasse `java.lang.IO` definiert sind und die textbasierte Interaktion mit der Konsole vereinfachen.
  - **Automatischer Modulimport von `java.base`:** Implizit deklarierte Klassen importieren automatisch alle öffentlichen Klassen und Schnittstellen der vom `java.base`-Modul exportierten Packages.
- Basierend auf beiden kann die `main()`-Methode seit Java 23 klarer und kürzer wie folgt geschrieben werden:

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



# Ausblick (Bisherige) JEPs in Java 26



## Targeted

- JEP 504: Remove the Applet API
- JEP 517: HTTP/3 for the HTTP Client API
- JEP 522: G1 GC: Improve Throughput by Reducing Synchronization

## Proposed to Target

- JEP 529: Vector API (Eleventh Incubator)

Java 26

## Candidate

- JEP 523: Make G1 the Default Garbage Collector in All Environments
- JEP 524: PEM Encodings of Cryptographic Objects (Second Preview)
- JEP 525: Structured Concurrency (Sixth Preview)
- JEP 526: Lazy Constants (Second Preview)
- JEP 527: Post-Quantum Hybrid Key Exchange for TLS 1.3
- JEP 528: Post-Mortem Crash Analysis with jcmands
- JEP 530: Primitive Types in Patterns, instanceof, and switch (Fourth Preview)



---

# Fazit



## Positives

---



- Stabile und zuverlässige 6-monatige Release-Zyklen und alle 2 Jahre LTS-Versionen
- Java wird einfacher und attraktiver
- Viele schöne Verbesserungen in Syntax und APIs wie switch, Records, Text Blocks
- Pattern Matching und Record Patterns final in Java 21
- Virtuelle Threads & Structured Concurrency u. v. m.
- JAVA 25 LTS ist seit Kurzem verfügbar ☺

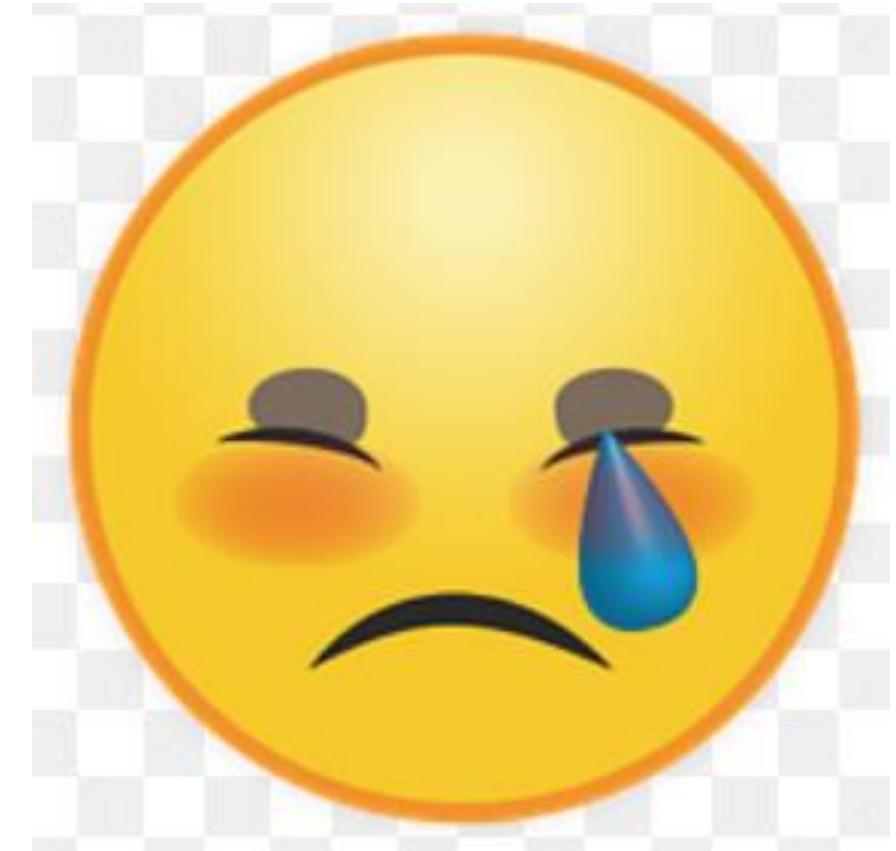


## Negatives

---



- **Releases waren zwar pünktlich, aber manchmal eher dünn bezüglich wichtiger Neuerungen**
- **Java 21/25 LTS enthalten einige unfertige Dinge ... meiner Meinung nach sollte ein LTS weniger Previews und möglichst keine Incubators enthalten**
- **Warum ist die Syntax von Pattern Matching bei instanceof und switch inkonsistent (when vs &&)?**





---

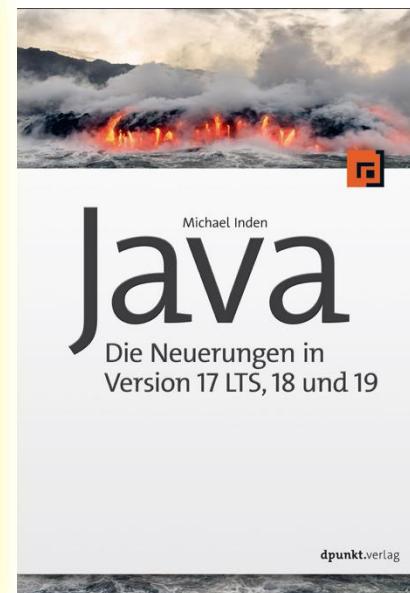
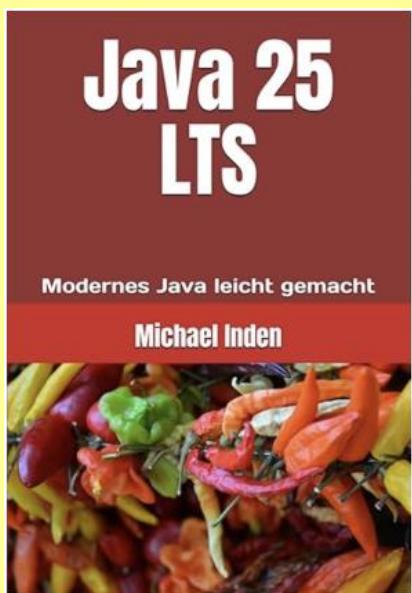
## Probier es aus :-)



<https://github.com/Michaeli71/Best-Of-Modern-Java-21-25-My-Favorite-Features>

---

Hilfe





# Questions?



# Thank You