Java 26 + 27

Die aktuellen Neuerungen

Falk Sippach

22.09.2026 JAVA FORUM NORD

embarc

0

Teil der Java Community

Nächster Termin am 01.10.

https://www.jug-da.de/

monatliche Opensource Code Camps

https://cyberland.ijug.eu/

3

Agenda

Was euch erwartet

01 Java – still a thing?

Java in 2026 02

03 Die neuen Features

Fazit + Ausblick 04

4

01. Halbjährlicher Release-Prozess

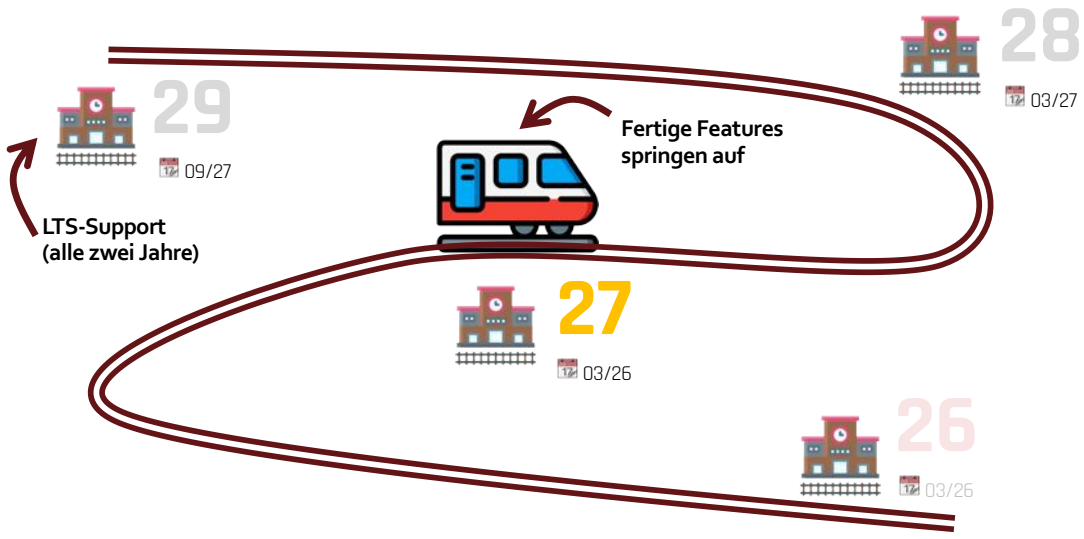
Seit über 5 Jahren wird halbjährlich eine neue Version des OpenJDK herausgebracht. Wie läuft der Prozess?

5

Warum ist **Java** auch nach über **30 Jahren** immer **noch so populär**?

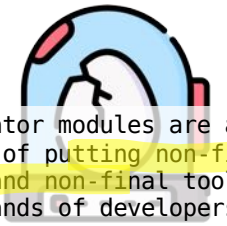
6

Halbjährlicher Release-Train



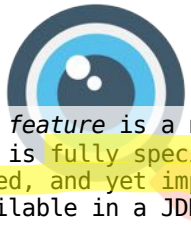
7

Inkubator und Preview



Incubator modules are a means of putting non-final APIs and non-final tools in the hands of developers, while the APIs/tools progress towards either finalization or removal in a future release.

https://openjdk.org/jeps/11



A *preview feature* is a new feature [..] that is fully specified, fully implemented, and yet impermanent. It is available in a JDK feature release to provoke developer feedback based on real world use; this may lead to it becoming permanent in a future Java SE Platform.

https://openjdk.org/jeps/12

9

02.

Java in 2026

Was ist in diesem Jahr passiert?

10

Wo kommen wir her?

2014	2018	2021	2023	2025
8	11	17	21	25
Lambdas Streams java.time	HTTP Client JFR neue String-APIs	Sealed Classes Records Type Patterns	Virtual Threads Record/Switch Patterns Sequenced Collections	Compact Source Files Flexible Constructors Module Imports Class-File API

Java 17, 21 und 25 sind die direkte Vorgeschichte für die Features in 2026.

11

Zoom: Java 17, 21 und 25

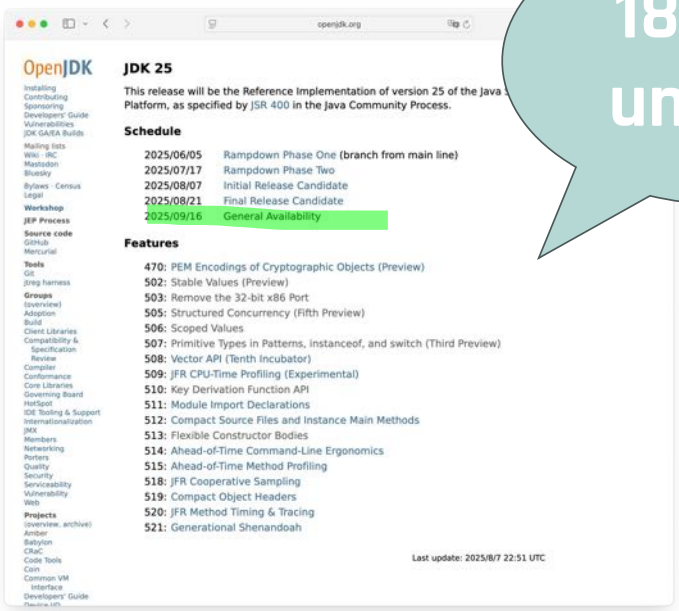
Diese drei LTS-Releases erklären fast alles, was 2026 in Java 26 und 27 weiter ausgebaut wird.

Java 17	Java 21	Java 25
Sealed Classes	Virtual Threads	Scoped Values
Records	Record Patterns	Flexible Constructor Bodies
Type Patterns	Pattern Matching for switch	Class-File API
	Sequenced Collections	Compact Object Headers
		Security APIs

Java 26 + 27: **Primitive Type Patterns, Structured Concurrency, Lazy Constants, PEM Encodings + kryptographische Funktionen, Vector API**

12

Java 25



OpenJDK

JDK 25

This release will be the Reference Implementation of version 25 of the Java
Platform, as specified by JSR 400 in the Java Community Process.

Schedule

2025/06/05	Rampdown Phase One (branch from main line)
2025/07/17	Rampdown Phase Two
2025/08/07	Initial Release Candidate
2025/08/21	Final Release Candidate
2025/09/16	General Availability

Features

- 470: PEM Encodings of Cryptographic Objects (Preview)
- 502: Stable Values (Preview)
- 503: Remove the 32-bit x86 Port
- 505: Structured Concurrency (Fifth Preview)
- 506: Scoped Values
- 507: Primitive Types in Patterns, instanceof, and switch (Third Preview)
- 508: Vector API (Tenth Incubator)
- 509: JFR CPU-Time Profiling (Experimental)
- 510: Key Derivation Function API
- 511: Module Import Declarations
- 512: Compact Source Files and Instance Main Methods
- 513: Flexible Constructor Bodies
- 514: Ahead-of-Time Command-Line Ergonomics
- 515: Ahead-of-Time Method Profiling
- 518: JFR Cooperative Sampling
- 519: Compact Object Headers
- 520: JFR Method Timing & Tracing
- 521: Generational Shenandoah

Last update: 2025/8/7 22:51 UTC

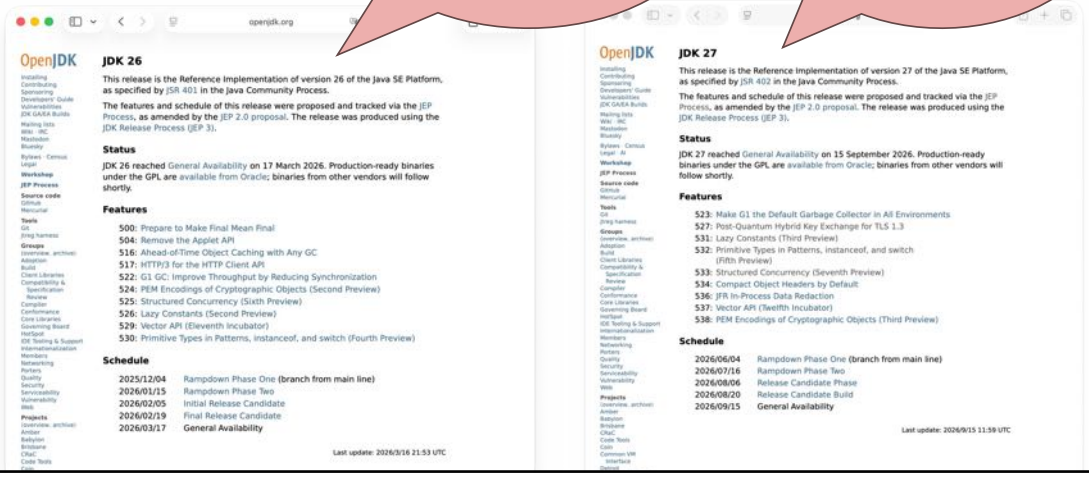
18 JEPs und LTS

13

Java 26 + 27

😞 nur noch 10 JEPs

😩 nur noch 9 JEPs



OpenJDK

JDK 26

This release is the Reference Implementation of version 26 of the Java SE Platform, as specified by JSR 401 in the Java Community Process.

The features and schedule of this release were proposed and tracked via the JEP Process, as amended by the JEP 2.0 proposal. The release was produced using the JDK Release Process (JEP 3).

Status

JDK 26 reached General Availability on 17 March 2026. Production-ready binaries under the GPL are available from Oracle; binaries from other vendors will follow shortly.

Features

- 500: Prepare to Make Final Mean Final
- 504: Remove the Applet API
- 516: Ahead-of-Time Object Caching with Any GC
- 517: HTTP/3 for the HTTP Client API
- 522: G1 GC: Improve Throughput by Reducing Synchronization
- 524: PEM Encodings of Cryptographic Objects (Second Preview)
- 525: Structured Concurrency (Sixth Preview)
- 526: Lazy Constants (Second Preview)
- 529: Vector API (Eleventh Incubator)
- 530: Primitive Types in Patterns, instanceof, and switch (Fourth Preview)

Schedule

2025/12/04	Rampdown Phase One (branch from main line)
2026/01/15	Rampdown Phase Two
2026/02/05	Initial Release Candidate
2026/02/19	Final Release Candidate
2026/03/17	General Availability

Last update: 2026/3/16 21:53 UTC

OpenJDK

JDK 27

This release is the Reference Implementation of version 27 of the Java SE Platform, as specified by JSR 402 in the Java Community Process.

The features and schedule of this release were proposed and tracked via the JEP Process, as amended by the JEP 2.0 proposal. The release was produced using the JDK Release Process (JEP 3).

Status

JDK 27 reached General Availability on 15 September 2026. Production-ready binaries under the GPL are available from Oracle; binaries from other vendors will follow shortly.

Features

- 523: Make G1 the Default Garbage Collector in All Environments
- 527: Post-Quantum Hybrid Key Exchange for TLS 1.3
- 531: Lazy Constants (Third Preview)
- 532: Primitive Types in Patterns, instanceof, and switch (Fifth Preview)
- 533: Structured Concurrency (Seventh Preview)
- 534: Compact Object Headers by Default
- 536: JFR In-Process Data Redaction
- 537: Vector API (Twelfth Incubator)
- 538: PEM Encodings of Cryptographic Objects (Third Preview)

Schedule

2026/06/04	Rampdown Phase One (branch from main line)
2026/07/16	Rampdown Phase Two
2026/08/06	Release Candidate Phase
2026/08/20	Release Candidate Build
2026/09/15	General Availability

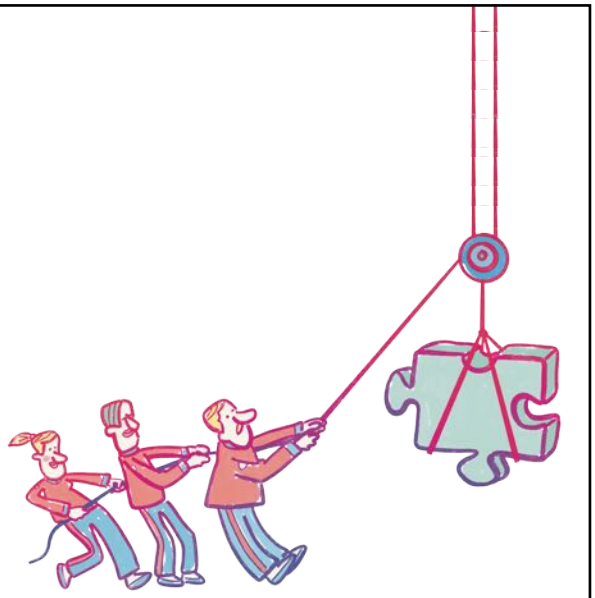
Last update: 2026/9/15 11:59 UTC

14

03.

Die neuen Features

Was ist überraschend, was ist besonders relevant?



15

Neue Features

Garbage Collectoren

Interna

API-Änderungen (JDK)

16

Sicherer: Security & Integrity by Default

17

final soll wirklich final bedeuten

- final aktuell noch per (Deep) Reflection (*setAccessible(true)*) veränderbar
- JVM warnt seit Java 26
- Mutation nur per Opt-in

18

PEM Encodings of Cryptographic Objects

- vereinfachter Umgang mit kryptografischen Schlüsseln und Zertifikaten
- Die neue API übernimmt zentrale Aufgaben wie das Einlesen von PEM-Daten, das Entfernen der Header und Footer sowie das Base64-Decoding.

```
-----BEGIN CERTIFICATE-----
...
-----END CERTIFICATE-----
```

new
26
09/26


```
KeyPairGenerator ed25519 =
KeyPairGenerator.getInstance("Ed25519");
KeyPair edKeyPair = ed25519.generateKeyPair();
PEMEncoder encoder = PEMEncoder.of();

byte[] edPrivateKeyBlock =
encoder.encode(edKeyPair.getPrivate());
IO.println(new String(edPrivateKeyBlock,
StandardCharsets.US_ASCII));
```

```
-----BEGIN PRIVATE KEY-----
MC4CAQAwBQYDK2VwBCIEIF7X4IKaGvf8lNqZQIUqNMJuId89BcX6FNNenebK+qkA
-----END PRIVATE KEY-----
```

Post-Quantum TLS

- Problem: **Harvest now, decrypt later**
- TLS 1.3 + ML-KEM
- Klassisch und quantenresistent kombiniert
- Für Anwendungen weitgehend transparent

22

JFR Data Redaction

- Secrets automatisch erkennen
- Vor Speicherung bereits schwärzen
- Secure by Default

```
token=[REDACTED]
//statt
token=secret123
```


23

Effizienter: Runtime & bessere Defaults

24

JEP 450 - Compact Object Headers

Speicherverbrauch minimieren

Verkleinerung des Objekt-Headers
von 128 auf 64 Bit

geringerer Heap-Bedarf, weniger GC-Arbeit und
Verbesserungen bei CPU-Zeit und Datenlokalität

new
24
03/25

25

JEP 450 - Compact Object Headers

- Final in Java 25
- **Default in Java 26**
- deaktivierbar:

```
java -XX:-UseCompactObjectHeaders ...
```

new
27
03/25

26

JEP 483: Ahead-of-Time Class Loading & Linking

- reduziert Initialisierungsarbeit und verkürzt die Warmup-Phase
- häufig genutzte Klassen werden bereits zur Build-Zeit vorbereitet und nicht mehr erst zur Laufzeit geladen
- Anstatt diese Strukturen bei jedem Start neu aufzubauen, können sie vorbereitet und beim Start direkt wiederverwendet werden

new
24
03/25

27

JEP 516: Ahead-of-Time Object Caching with Any GC

- Ahead-of-Time Object Caching nun unabhängig vom verwendeten Garbage Collector
- Kombination und Optimierung mit unterschiedlichen GC-Konfigurationen

new
26
03/26

28

Neue Generation von Low-Latency GCs

Garbage Collectoren: ZGC und Shennadoah

moderne Architekturen: Multi-Core + TB RAM

Ziel: kurze GC-Pausen im ms-Bereich

29

G1: Improve Throughput by Reducing Synchronization

- G1 seit vielen Jahren Standard-GC
  - relativ stabile Pausenzeiten bei gutem Durchsatz
  - aber interne Skalierungsgrenzen bei Systemen mit vielen CPU-Kernen
- JEP 522 reduziert gezielt diese Synchronisationspunkte
  - weniger Lock-Contention innerhalb des Garbage Collectors
  - Parallelisierung kann besser ausgenutzt

Default in Java 27 (auch für kleine JVMs)

new
26
03/26

30

Praktischer: neue APIs

31

HTTP/3 Unterstützung

```
var client = HttpClient.newBuilder()
   .version(HttpClient.Version.HTTP_3)
   .build();
```


32

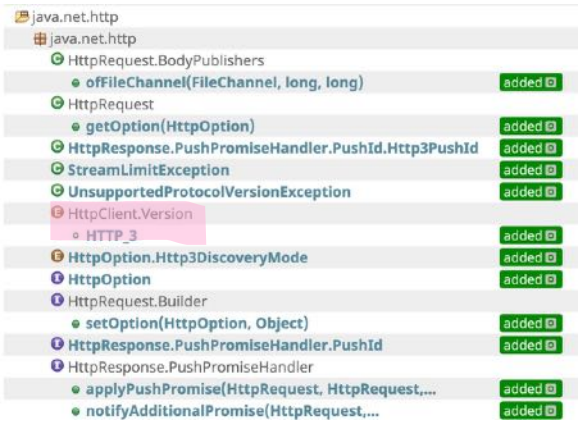
HTTP/3

- basiert nicht mehr auf TCP, sondern auf QUIC, einem UDP-basierten Transportprotokoll
- Vorteile:
  - geringere Latenz beim Verbindungsaufbau
  - bessere Performance bei Paketverlust
  - kein Head-of-Line-Blocking zwischen parallelen Streams
  - stabilere Verbindungen bei Netzwechsel, etwa bei mobilen Clients

33

HTTP-Client mit HTTP/3 Unterstützung

- *java.net.http.HttpClient*
  mittlerweile Standardwerkzeug
  für HTTP-Kommunikation

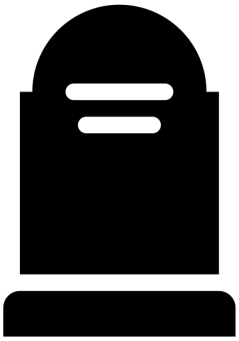


Beispiel

```
HttpClient client = HttpClient.newBuilder()
        .version(HttpClient.Version.HTTP_3)
        .build();

HttpRequest request = HttpRequest.newBuilder()
        .uri(URI.create("https://www.embarc.de"))
        .GET()
        .build();

HttpResponse<String> response = client.send(request,
HttpResponse.BodyHandlers.ofString());

System.out.println(response.headers());
```

RIP: Remove Applet API (JEP 504)

- zentrales Versprechen in den 90ern:
  Java-Code sollte direkt im Browser laufen



26

03/26

36

Ausdrucksstärkere Sprache

37

Pattern Matching

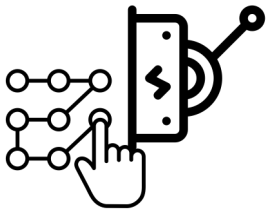
*„Pattern matching is a mechanism for **checking a value against a pattern**. A successful match can also **deconstruct a value into its constituent parts**.*

*It is a more **powerful version of the switch statement in Java** and it can likewise be used in place of a series of if/else statements.“*

https://docs.scala-lang.org/tour/pattern-matching.html

38

Switch Expression



Pattern Matching for switch

Record Patterns (Deconstruction)

Type Patterns (Pattern Matching for instanceof)

Records

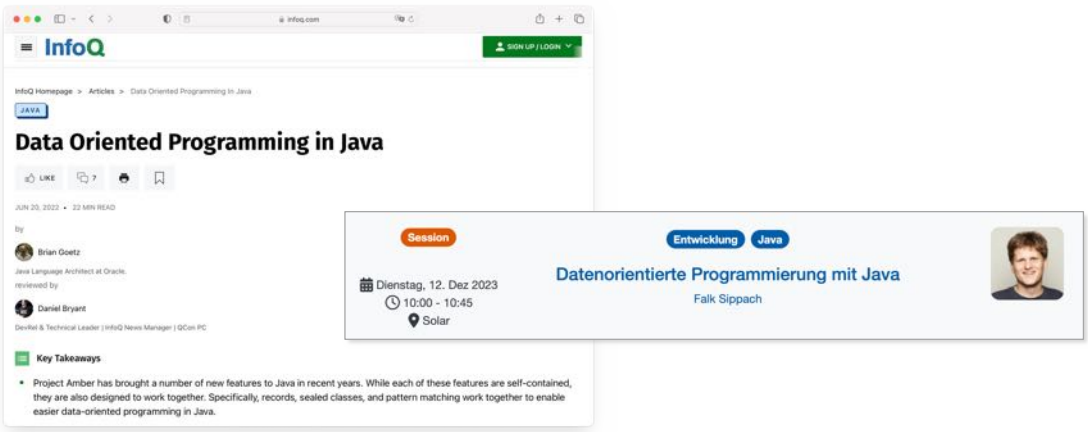
Unnamed Patterns and Variables

Primitive Type Patterns

Sealed Classes

39

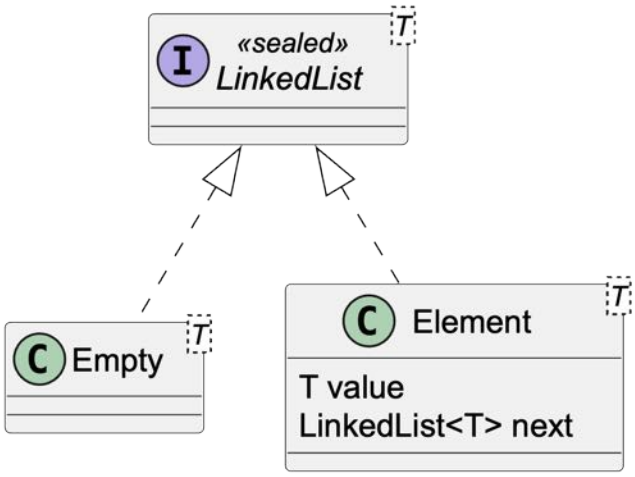
Neues Programmierparadigma?



https://www.infoq.com/articles/data-oriented-programming-java/

40

Algebraische Datentypen

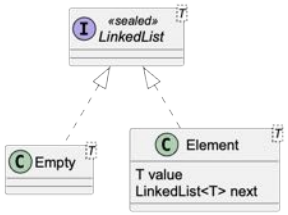


42

Sealed Class/Interface

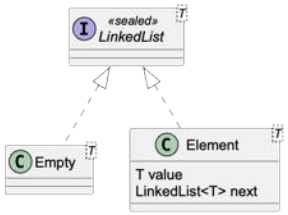
```
public sealed interface LinkedList<T>
  permits LinkedList.Element, LinkedList.Empty {

  record Element<T>(T value, LinkedList<T> next)
     implements LinkedList<T> {}

  final class Empty<T> implements LinkedList<T> {}

  static <T> LinkedList<T> of(T... values) {
    ..
  }
}
```

Record



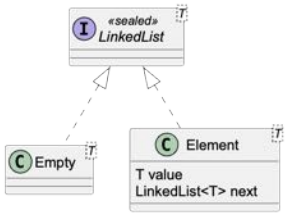
43


```
static <T> LinkedList<T> of(T... values) {
  if (values.length == 0) return new LinkedList.Empty<>();

  LinkedList<T> current = new LinkedList.Empty<>();

  for (int i = values.length - 1; i >= 0; i--) {
    current = new LinkedList.Element<>(values[i], current);
  }
  return current;
}
```



44

```
LinkedList<Integer> list = of(1, 2, 3);
System.out.println(contains(5, list)); // false
System.out.println(contains(1, list)); // true
```



45

switch-Expression

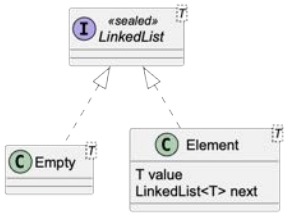
```
static <T> boolean contains(T value, LinkedList<T> list) {
  return switch (list) {
    case Empty empty -> false;
    case Element<T>(T v, var tail)
     when Objects.equals(v, value) -> true;
    case Element<T>(T v, var tail) -> contains(value, tail);
  };
}
```

Type Pattern
(Pattern Matching for instanceof)

Record Pattern

when-Clause (Guarded Pattern)

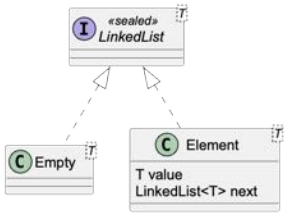
Kein Default notwendig (Exhaustiveness-Prüfung durch Compiler)!



46

```
static <T> boolean contains(T value, LinkedList<T> list) {
  return switch (list) {
    case Empty _ -> false;
    case Element<T>(T v, _)
      when Objects.equals(v, value) -> true;
    case Element<T>(_, var tail) -> contains(value, tail);
  };
}
```

Unnamed Pattern



47

Primitive Type Patterns (1)

Primitive Patterns räumen drei Altlasten auf

- **Asymmetrie**
- unnötiges **Boxing**/**Handarbeit**
- und **Lücken** bei Record-Deconstruction und switch.

48

Primitive Type Patterns (2)

```
static String level(int flights) {
   return switch (flights) {
      case 0 -> "New";
      case 1, 2 -> "Bronze";
      case int i when i >= 100 -> "Gold";     // Guard
      case int i -> "Silver (" + i + ')';
   };
}
```

49

Primitive Type Patterns (3)

```
int value = -128;
if (value instanceof byte b && b > 0) {
   System.out.println("b has a positive byte value: " + b);
} else {
   System.out.println("negative or not of type byte: " + value);
}
```

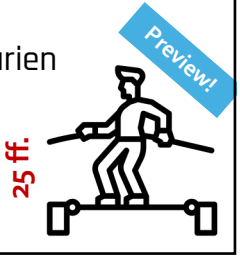
50

Primitive Type Patterns (4)

```
record PointWithDoubles(double x, double y) {}
record ColoredPoint(PointWithDoubles p, String color) {}

void describe(ColoredPoint cp) {
  switch (cp) {
    case ColoredPoint(PointWithDoubles(int x, int y), var color) ->
        System.out.printf("Pixel color %s at (%d, %d)%n", color, x, y);
    case ColoredPoint(PointWithDoubles(double x, double y), var color) ->
        System.out.printf("Precise color %s at (%.2f, %.2f)%n", color, x, y);
  }
}
```


Lazy Constants (war: Stable Values)

- "Immutability on demand" bzw. "Deferred Immutability"
  - API um fehleranfälliges Double Lock Idiom zu umgehen
- Ziele
  - schnellerer Start
  - Entkopplung von Erzeugung und Initialisierung
  - garantierte At-Most-One-Initialisierung in parallelen Szenarien
  - Konstanten-Optimierung durch JIT



```
static final Map<String, Pattern> PATTERNS = Map.of(
      "email", Pattern.compile("..."),
      "phone", Pattern.compile("..."),
      "zip", Pattern.compile("...")
);
```

Initialisierung beim Start der Anwendung
wird potentiell **nie verwendet**
verlängert Start-Up

55

Double Lock Idiom

```
Logger logger() {
  if (logger == null) {
    synchronized (this.getClass()) {
      if (logger == null) {
        logger = Logger.getLogger(OrderController.class.getName());
      }
    }
  }
  return logger;
}
```

aufwändig
schlecht les-/wartbar
fehleranfällig

56

Beispiel Lazy Constants

```
private final LazyConstant<Logger> lazyLogger =
  LazyConstant.of(() ->
    Logger.getLogger(OrderController.class.getName()));

void startOrder() {
  lazyLogger.get().info("order started");
}
```


57

Strukturierter: Concurrency neu gedacht

58

Die Geschichte von Threads in Java

1.1
1997
Green Threads

1.3
1998
Native Threads

5
2004
Thread Pools/**Executor**

7
2011
Fork-Join-Pools
Parallel Streams

8
2014 - 2016
Completable Future,
RxJava, Reactive
Streams, Reactor

9
2017 - 2018
Flow
Kotlin Coroutines

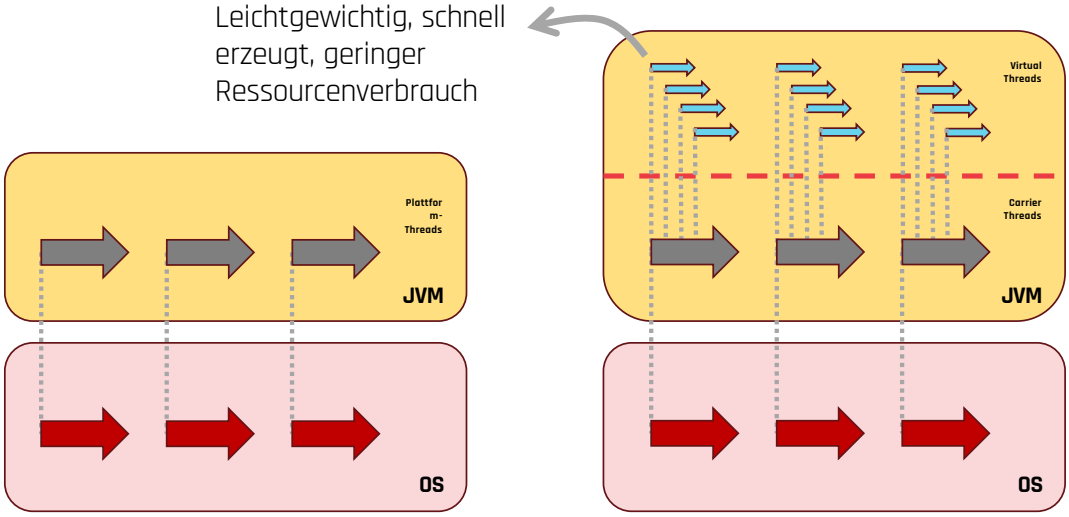
19
2022
Virtual Thread
(Preview)

21
2023
Virtual Threads Final

59

Plattform- vs. Virtual Threads

Leichtgewichtig, schnell
erzeugt, geringer
Ressourcenverbrauch



61

Millionen von Threads …

```
Thread.Builder builder = Thread.ofVirtual();
// Thread.Builder threadBuilder = Thread.ofPlatform();

builder.start(() -> {
    // im Thread auszuführender Code
    System.out.println(
        Thread.currentThread().isVirtual());
});
```

Virtual Threads

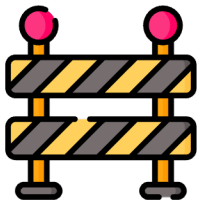
62

Aber es geht gar nicht um Millionen von Threads

klassisches Thread-**API**

Vereinfachter Umgang mit vielen, **blockierenden** Tasks **(IO)**

nicht schneller oder effizienter (laufen auch nur auf Platform Threads)



nicht für sehr rechenintensive (**lange blockierende**) Tasks

63

Für wen ist es gut?

viele/mehr parallele Requests

aber **externe Ressourcen** bleiben **Bottleneck**

nicht für die Verarbeitung großer Datenmengen bzw. **langlaufende Berechnungen**

"naive" Programmierung von Threads (**einfacheres API, leichtgewichtig, kein Pooling**)

Virtual Threads unter der Haube

Wir alle, die Webanwendungen bauen und davon **indirekt profitieren**.

Nebenläufige Programmierung

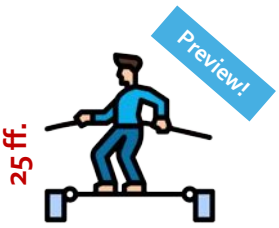
Implementierung von **sehr vielen** und/oder **blockierenden Aufgaben**

65

Weiteres im Umfeld von Virtual Threads

Structured Concurrency

Scoped Values

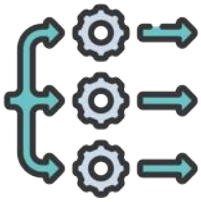


Lazy Constants (Stable Values)

Funktioniert auch mit **Platform Threads!**

66

Structured Concurrency



Einfaches **Aufsplitten** eine Haupt-Tasks in Sub-Tasks

Behält Verbindung: **Main- und Sub-Task**

Verringert Fehler entstanden durch **Annullierung und Abschaltungen**

Passt sehr gut zu **Virtual Threads**

Vor Structured Concurrency

```
private static final ExecutorService executor =
    Executors.newCachedThreadPool();

Future<String> task1 = executor.submit(() -> { … }
Future<String> task2 = executor.submit(() -> { … }
Future<String> task3 = executor.submit(() -> { … }

System.out.println(task1.get());
System.out.println(task2.get());
System.out.println(task3.get());
```

Fragestellungen (Szenarien):
- 1. Thread gewinnt, restliche abbrechen
- alle abbrechen, wenn ein Fehler passiert
- alle müssen erfolgreich sein
- …

Structured Concurrency

alternative Strategien über:
Joiner (auch **eigene** Implementierung)

```
record Response(String user, int order) {}

Response handle() throws InterruptedException {
  try (var scope = StructuredTaskScope.<Object, Void>open()) {
    var user  = scope.fork(this::findUser);   // liefert String
    var order = scope.fork(this::fetchOrder); // liefert Integer
    scope.join(); // wartet auf alle
    return new Response((String) user.get(), (Integer) order.get());
  }
}

// Platzhalter:
String findUser()   { return "alice"; }
int    fetchOrder() { return 42; }
```

wartet auf alle, wirft ggf. **Exception** und **beendet** die **anderen Tasks**

19 - 27

basiert auf Virtual Threads

69

Structured Concurrency – Strategien

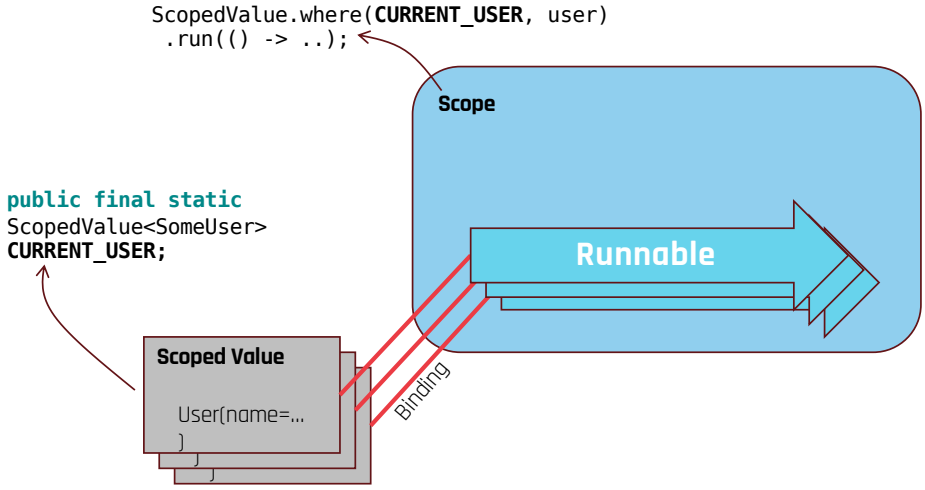
```
// Erstes erfolgreiches Ergebnis gewinnt
StructuredTaskScope.open(Joiner.<T>anySuccessfulResultOrThrow(),
        cfg -> cfg.withTimeout(java.time.Duration.ofSeconds(2)))

// Alle oder keiner
StructuredTaskScope.open(Joiner.<T>allSuccessfulOrThrow())

// Erfolgreiche und fehlgeschlagene Tasks unterscheiden
StructuredTaskScope.open(Joiner.<T>awaitAll())

// Früher Abbruch per Prädikat
StructuredTaskScope.open(
        Joiner.<String>allUntil(st ->
                st.state() == Subtask.State.SUCCESS
                && st.get().contains("OK")))
```

19 - 27

basiert auf Virtual Threads

70

Scoped Values – Alternative zu ThreadLocal



Aktivierung Preview + Inkubator Features

```
$ javac --enable-preview -source 26 --add-modules
    jdk.incubator.concurrent StructuredConcurrency.java

$ java --enable-preview --add-modules
    jdk.incubator.concurrent StructuredConcurrency
```

Die Java-Welt und
das Ökosystem sind
lebendig wie nie!

77

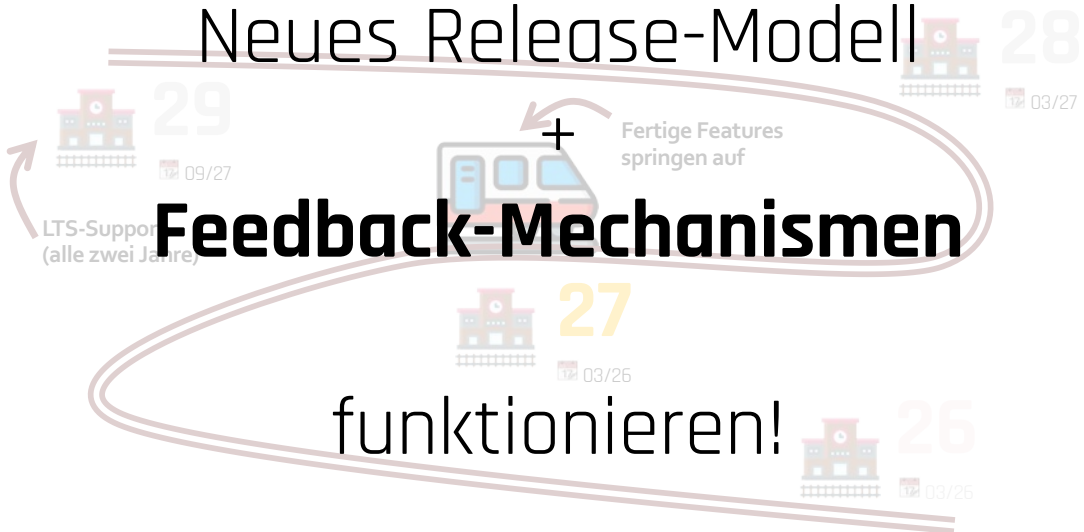
Halbjährlicher Release-Train

Neues Release-Modell

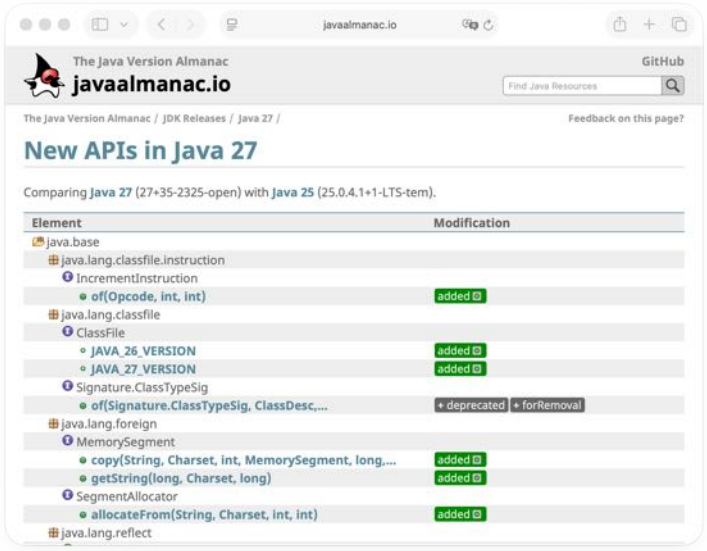
+

Feedback-Mechanismen

funktionieren!



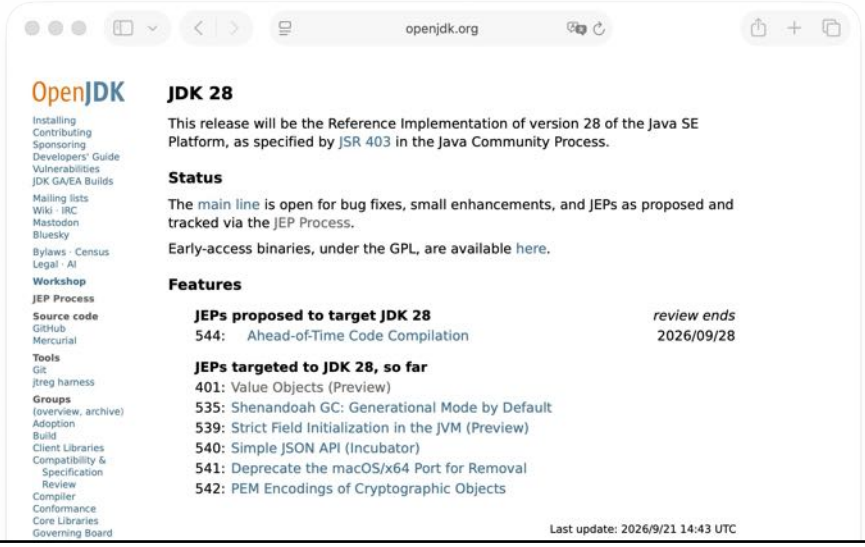
78



https://javaalmanac.io/jdk/27/apidiff/25/

79

Java 28



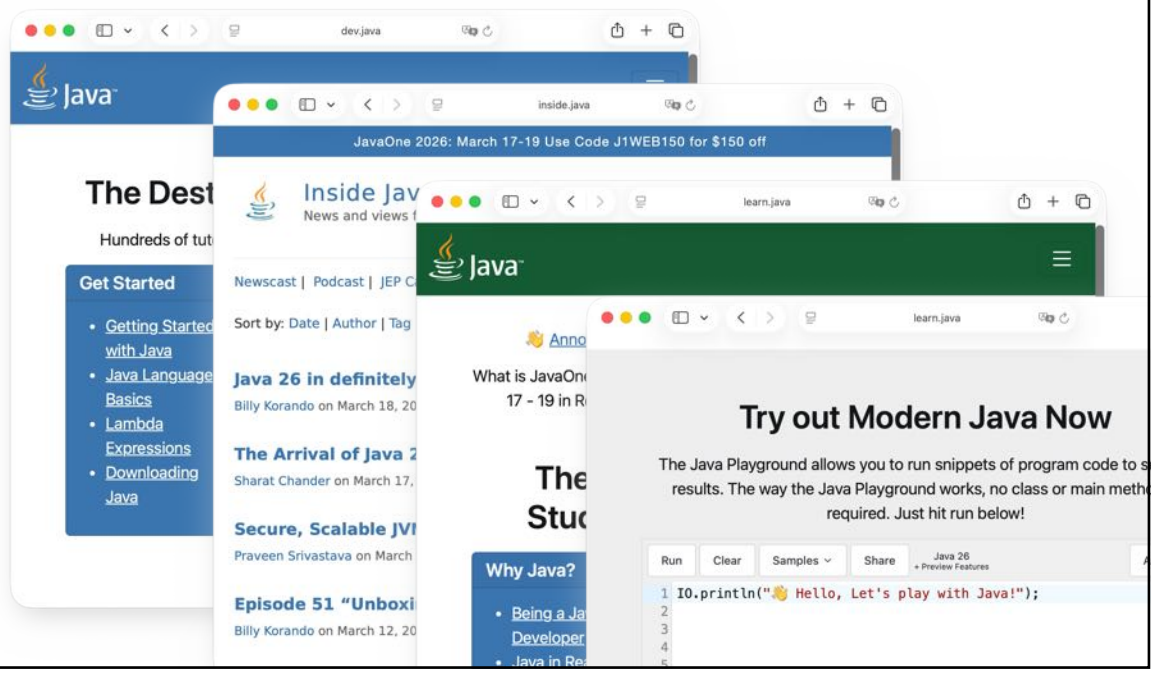
pre
28
03/27

80

Java 28

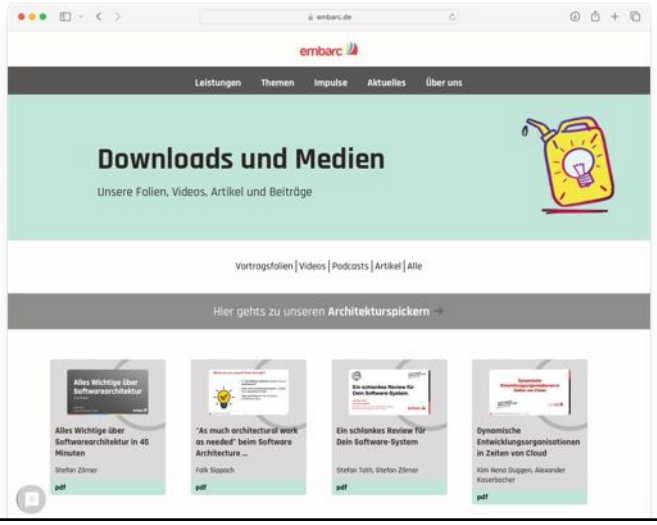
- JEP 401: Value Classes & Objects
  - JEP 539: Strict Field Initialization in the JVM
- JEP 540: Simple JSON API

81



84

Folien als PDF zum Download



➔ embarc.de/download/

86

87

Architektur-Spicker (1 -16)

„Mit unseren Architektur-Spickern beleuchten wir die konzeptionelle Seite der Softwareentwicklung."

Architektur-Spicker #1
Der Architekturüberblick

PDF, 4 Seiten
Kostenloser Download.

➔ embarc.de/architektur-spicker/

88

Vielen Dank.

Ich freue mich auf Eure Fragen!

falk.sippach@embarc.de
linkedin.com/in/falk-sippach
@sippsack
@sippsack@ijug.social

90

Falk Sippach

- Softwarearchitekt, Berater, Trainer bei embarc
- früher bei Orientation in Objects (OIO), Trivadis

Schwerpunkte

- Architekturberatung und -bewertung
- Cloud- und Java-Technologien

embarc

iSAQB

Java Champions