

Value Objects

Das nächste große Ding in Java?

Falk Sippach



10. Juli 2025



0



embarc.de

Abstract

Value Objects

Einer der großen Pluspunkte von Java war und ist das statische, starke Typsystem. Es hilft, viele Fehler bereits zur Compilezeit zu entdecken und macht die Entwicklung robuster sowie effizienter. Allerdings integrieren sich die vor etwa 30 Jahren aus Performancegründen eingeführten primitiven Datentypen **nicht** gut mit modernen Ansätze wie Generics, Stream API oder Pattern Matching. Value Objects versprechen Abhilfe und werden die Vorteile beider Welten kombinieren. Damit können wir in Zukunft unveränderbare Datentypen entwerfen, die sich wie primitive Datentypen verhalten. Das steigert nicht nur die Performance und senkt den Speicherverbrauch, es erhöht durch das Schreiben von ausdrucksstarken Typen auch die Les- und Wartbarkeit.

Schon seit etwa 10 Jahren wird im Rahmen vom Projekt Valhalla an dieser großen Änderung des Java Typsystems gearbeitet. Daran hängen einige komplexe Fragestellungen, wie z. B. der Umgang mit Default-Werten sowie null-Values, der Umbau der Wrapper-Typen (Integer, ...) und die Verwendung als generische Typisierung. Im Sommer 2024 hat Java Language Architekt Brian Goetz verkündet, dass nach der langen Zeit der Durchbruch in der Umsetzung erreicht ist. Darum wollen wir gemeinsam schauen, wie Value Classes, Null-Restricted und Nullable Types sowie erweitertes Primitive Boxing die Art und Weise verändern, wie wir in Zukunft programmieren werden.

Value Objects

1

1



embarc.de

Value Objects

2

Falk Sippach

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

Schwerpunkte

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



2



embarc.de

Value Objects

3

Teil der Java Community



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

Nächster Termin am 24.07.



monatliche Opensource
Code Camps

<https://cyberland.ijug.eu/>

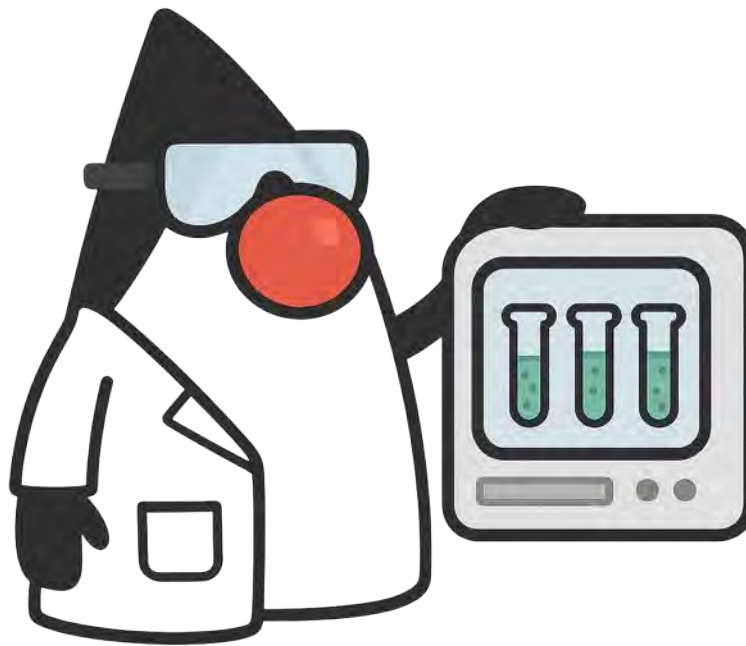
3



embarc.de

Value Objects

5



5



embarc.de

Value Objects

6

Pattern Matching &
kleine Syntax-
Verbesserungen



Amber



Panama

Vector API &
Foreign Function &
Memory API



Valhalla

Value Types



Loom

Virtual Threads



Leyden

+ Startup Time
+ Time to Peak
Performance
- Memory Footprint

6



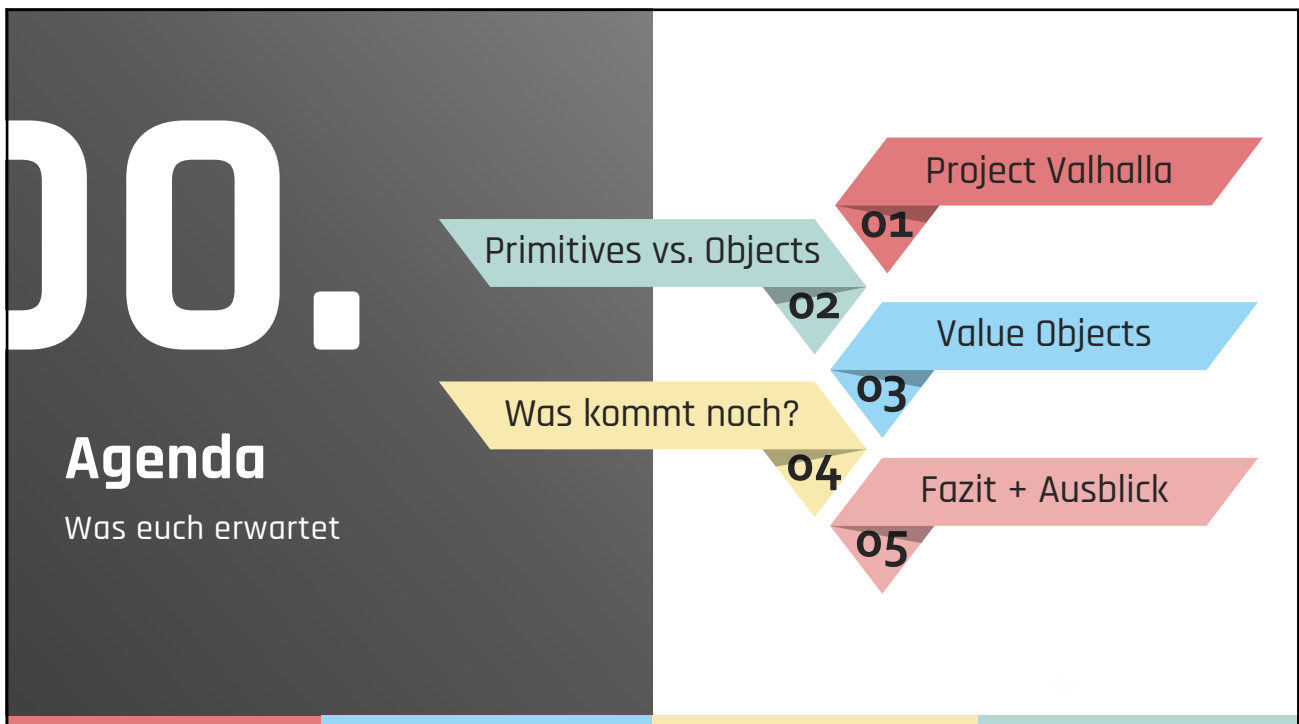
nordische Mythologie:

Valhalla =
**"Halle der
gefallenen
Krieger"**



Wortspiel:
Val = Value

Valhalla =
Halle der Werte
(Behauptung von ChatGPT)



01.

Project Valhalla

Java meets nordische Mythologie



10



Project Valhalla

Project Valhalla is augmenting the Java object model with value objects, combining the **abstractions of object-oriented programming** with the **performance characteristics of simple primitives**. Supplementary changes to Java's generics will carry these performance gains into generic APIs.
[\(https://openjdk.org/projects/valhalla/\)](https://openjdk.org/projects/valhalla/)

11



embarc.de

Das Motto



12



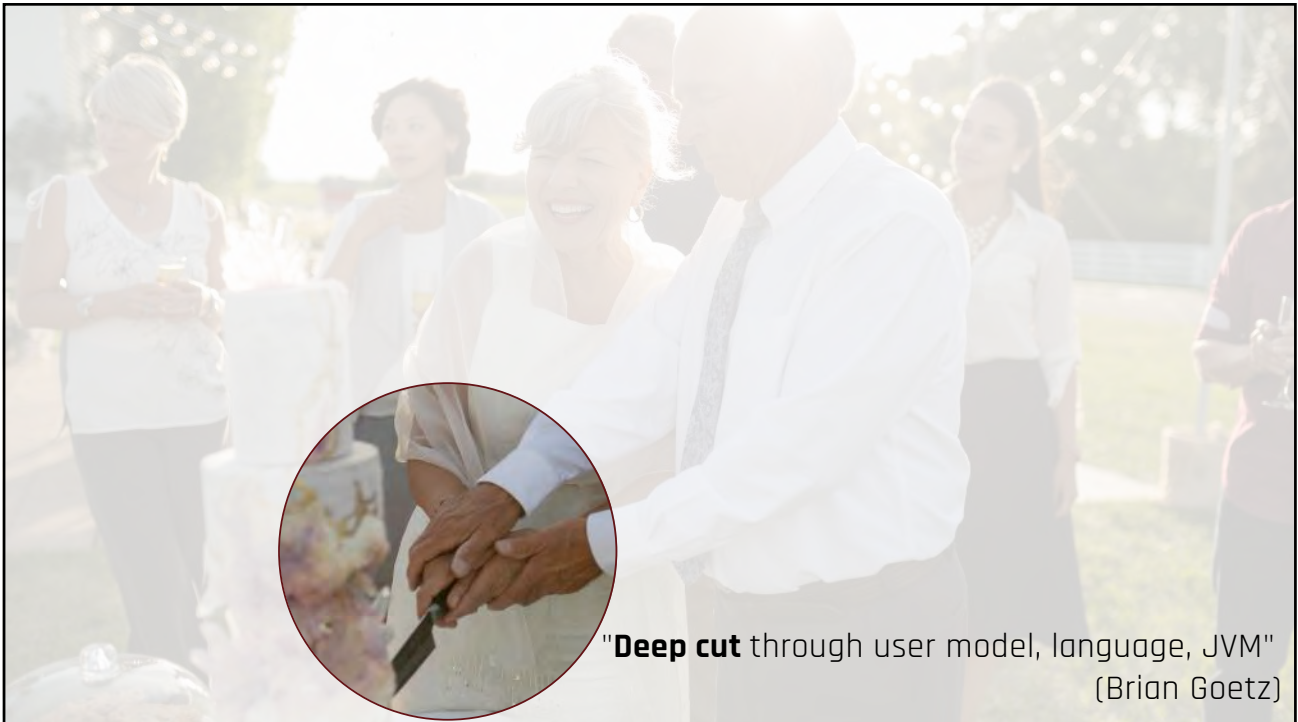
embarc.de

Java's **EPIC!!** Refactor

Value Objects

13

13



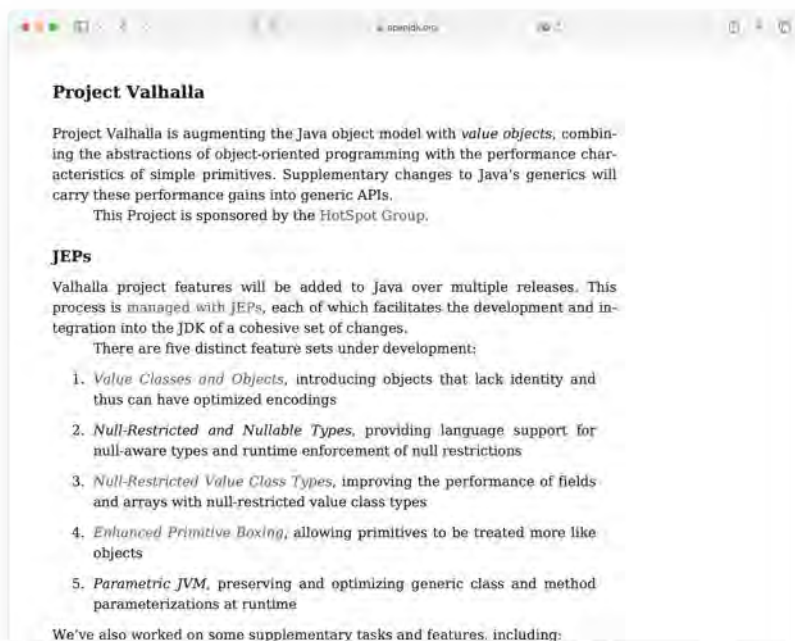
14



embarc.de

Value Objects

17



Project Valhalla

Project Valhalla is augmenting the Java object model with *value objects*, combining the abstractions of object-oriented programming with the performance characteristics of simple primitives. Supplementary changes to Java's generics will carry these performance gains into generic APIs.

This Project is sponsored by the HotSpot Group.

JEPs

Valhalla project features will be added to Java over multiple releases. This process is managed with JEPs, each of which facilitates the development and integration into the JDK of a cohesive set of changes.

There are five distinct feature sets under development:

1. *Value Classes and Objects*, introducing objects that lack identity and thus can have optimized encodings
2. *Null-Restricted and Nullable Types*, providing language support for null-aware types and runtime enforcement of null restrictions
3. *Null-Restricted Value Class Types*, improving the performance of fields and arrays with null-restricted value class types
4. *Enhanced Primitive Boxing*, allowing primitives to be treated more like objects
5. *Parametric JVM*, preserving and optimizing generic class and method parameterizations at runtime

We've also worked on some supplementary tasks and features, including:

<https://openjdk.org/projects/valhalla/>

17



Bereits ausgeliefert ...



JEP 181

Nest-Based
Access Control

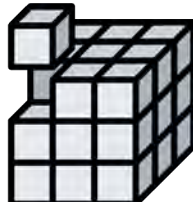
Java 11



JEP 309

Dynamic Class-
File Constants

Java 11



JEP 334

JVM Constants
API

Java 12



JEP 371

Hidden
Classes

Java 15



JEP 390

Warnings for
Value-Based
Classes

Java 16

19



JEP 181: Nest-Based Access Control



- Verbesserter Zugriff auf **private Member zwischen Klassen**, z. B. eine äußere und ihre inneren Klassen
- neues Konzept von "**Nestmates**" und erlaubt **direkten Zugriff** auf private Member innerhalb eines **Nests**
- Reduktion von Compiler-/Laufzeitkomplexität und Verbesserung der Performance.

Voraussetzung für Inline bzw. Value Klassen:

- effizientere Implementierung von **zusammengesetzten**, aber **nicht referenzierten Objekten**

20



embarc.de

Value Objects

21

JEP 309: Dynamic Class-File Constants



- neuer Bytecode-Mechanismus: **CONSTANT_Dynamic**
- Berechnung von Konstanten erst zur Laufzeit beim ersten Zugriff
- **effizientere Erzeugung** und **Caching** zur Laufzeit

Grundlage für Value Objects:

- Besseres Laufzeitverhalten durch **Lazy Initialization**
- **Flexiblerer Bytecode**

21



embarc.de

Value Objects

22

JEP 334: JVM Constants API



- **typensichere Programmierschnittstelle** für **Konstanten** im **Bytecode**
- **JVM-Konstanten** (wie Klassen, Methoden, Felder, primitive Werte) als **explizite Objekte im Java-Code** repräsentieren

Wichtig für Value Objects:

- **dynamisch strukturierte Daten** sicher **beschreiben**

22



embarc.de

JEP 371: Hidden Classes



- **Klassen**, die **zur Laufzeit erzeugt** werden, aber **nicht von regulärem Code referenziert** werden können
- ideal für **Frameworks, Bytecode-Generatoren und dynamische Proxies**, um Klassen zur internen Verwendung zu erzeugen

wichtige Infrastruktur für Inline/Value-Klassen:

- **temporäre**, effiziente **Klassenstrukturen** zu **erzeugen**
- z. B. für **dynamische Layouts** oder **runtime-spezifische Optimierungen**

value Objects

23

23



embarc.de

JEP 390: Warnings for Value-Based Classes



- **Klassen**, die sich **wie Werte** verhalten sollen (z. B. `java.lang.Integer`, `Optional`, `LocalDateTime`)
- sind **unveränderlich**, haben **keine Identität** und sollten **nicht synchronisiert** oder **über `==` verglichen** werden
- **Warnung**, wenn **verbotene Operationen** mit solchen Klassen durchgeführt werden
- **korrektes Nutzungsverhalten einfordern**, um zukünftig die Kompatibilität nicht zu brechen

value Objects

24

24

02.

Primitives Types vs. Objects

Altlasten des Java Memory
Models



25



Primitive Types

`int i = 12345;`

Deklaration einer Ganzzahl
(**int**) mit dem Wert **12345**.



Die **Variable i** wird im
Stack abgelegt. Es wird
kein Heap-Speicher
benötigt.

27



embarc.de

Value Objects

28

Welche Bedeutung hat *i*?

```
int i = 12345;
```



28



embarc.de

Value Objects

29

Ah, eine **Postleitzahl**!

```
int zipCode = 12345;
```

Aber nicht typischer,
Primitive Obsession!



29



```
record Address(..., int zipCode) {}
```

```
var a = new Address(..., "12345");
```

Immer noch
Primitive Obsession!



```
record Address(..., int no, int zipCode, ...) {}
```

```
var a = new Address(..., 12345, 2, ...);
```

Gefahr des **Vertauschens**.





embarc.de

Value Objects

32



Primitive Types



Object References

32



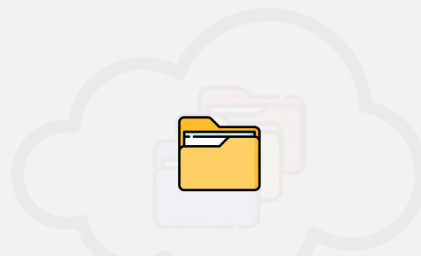
embarc.de

Value Objects

34



Stack



Heap

34



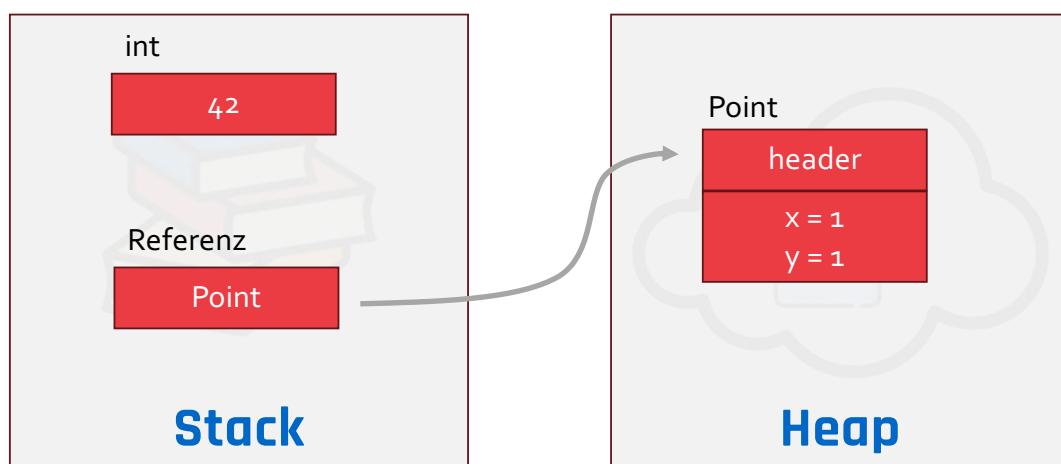
Die folgenden Zahlen hat GenAI ermittelt!



35

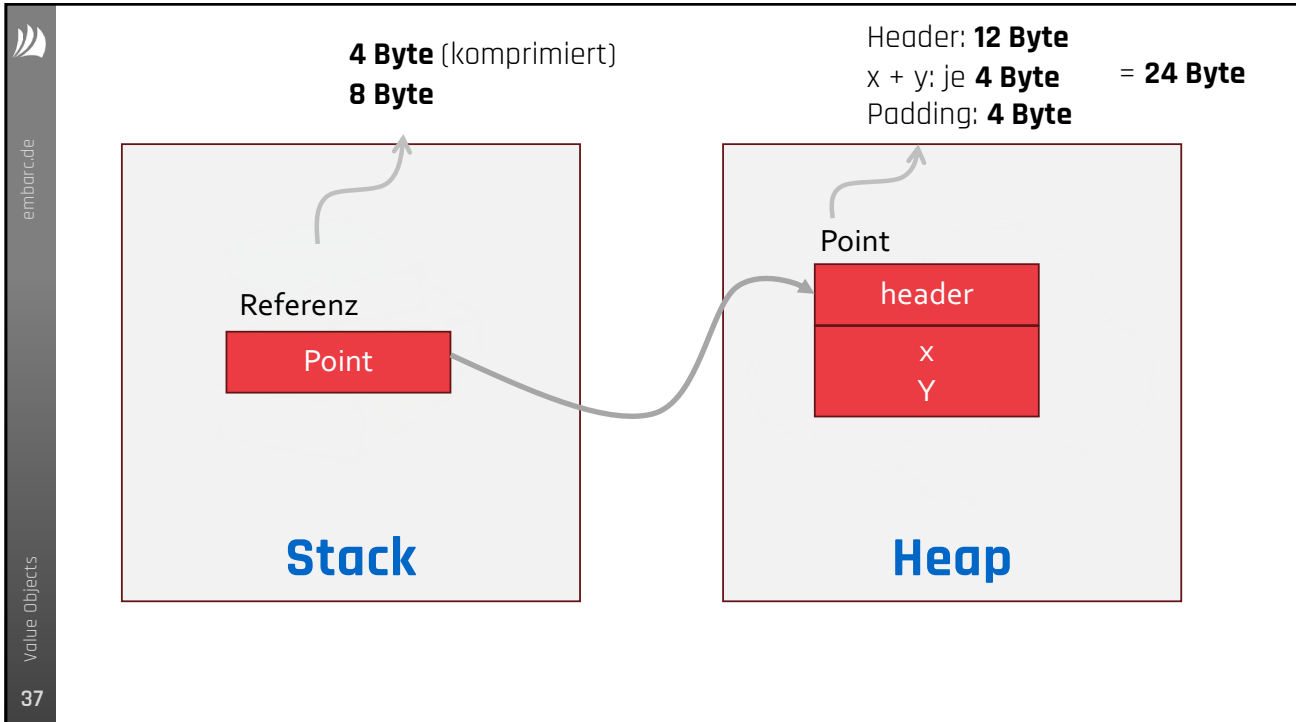


```
int i = 42; // 4 Byte
```



```
record Point(int x, int y); Point p = new Point(1, 1);
```

36



37

Was ist Padding?

- **Moderne Prozessoren arbeiten effizienter**, wenn **Daten an bestimmten Speicheradressen beginnen**, z. B. bei einem double auf einer 8-Byte-Grenze.
- Die JVM (z. B. HotSpot) richtet daher Objekte im Speicher so aus, dass sie auf **ein Vielfaches von 8 Byte passen**.
- Wenn die tatsächliche Größe eines Objekts nicht durch 8 teilbar ist, wird der **Speicher mit "Padding" aufgefüllt**.

38



Vergleich

Liste von Punkten

0	;	0
0	;	1
1	;	0
0	;	1
1	;	0
1	;	1

Mögliche Implementierung

effizient, aber schlecht wartbar

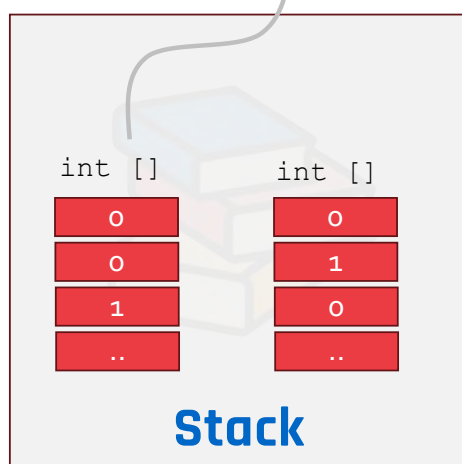
```
int[] xs; int[] ys;
```

Array von Point-Objekten

```
Point[] ps;
```



Header int[]: 12 Byte



1. Zwei int[]-Arrays (flache Struktur), je 10 Werte

Speicherverbrauch:

- Header von int[]: **12 Byte**
- Daten: 10 × 4 Byte = **40 Byte**
- Padding auf 8-Byte-Grenze: **4 Byte**

Pro Array: **56 Byte**

Gesamt für beide Arrays: **56 + 56 = 112 Byte**

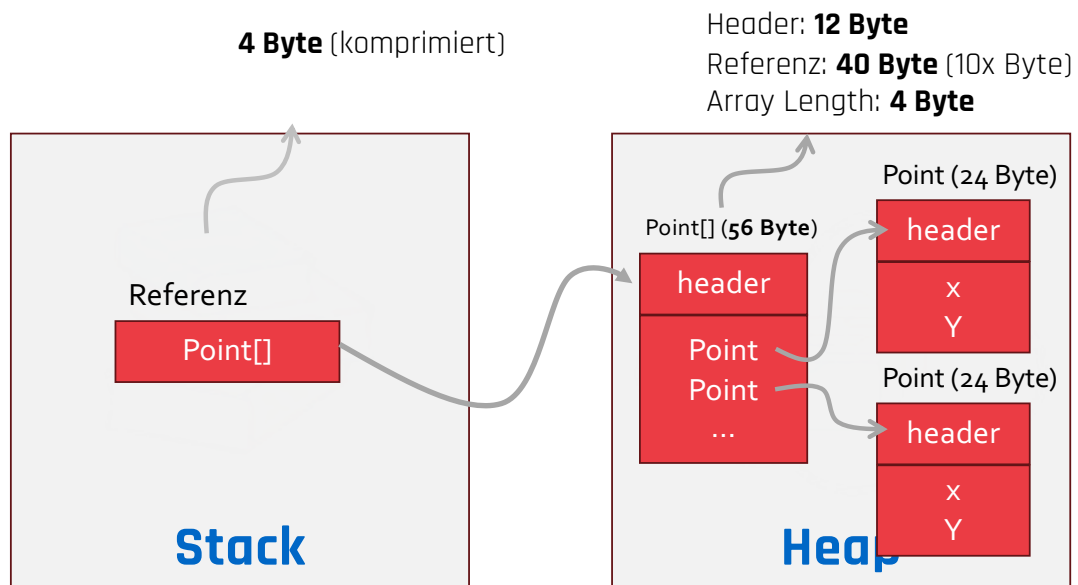
→ Sehr effizient, weil nur die reinen Daten gespeichert werden!



```
class Point {
    int x;
    int y;
}

Point[] points = new Point[10];

for (int i = 0; i < 10; i++) {
    points[i] = new Point();
}
```



```
Point[] points = new Point[10]; // > 300 Byte
```



Variante	Speicher (Byte)	Overhead durch Header & Referenzen?
Zwei int[]	112	Minimal
Point[] mit Objekten	332	Hoch (Referenzen, Header)

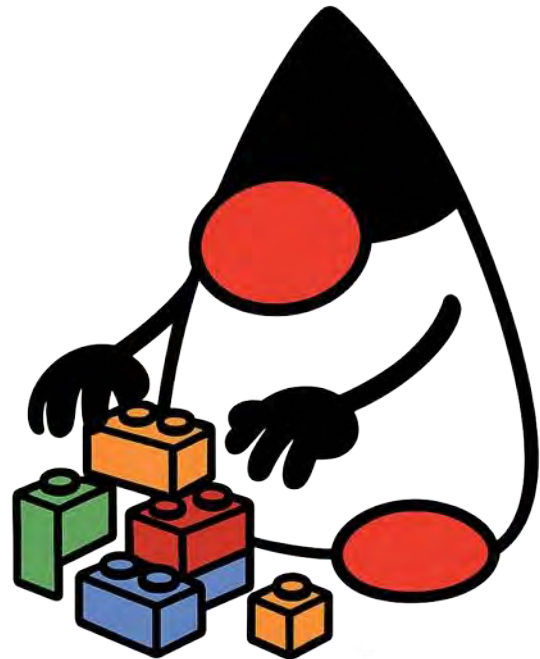


Primitive Types (int, double, boolean, ...) 	Object References (String, Arrays, Point, ...) 
✓ Direkt im Stack gespeichert (lokale Variablen)	✗ Liegen auf dem Heap, nur Referenz im Stack
✓ Keine Indirektion, schnelle Zugriffe	✗ Indirektion durch Pointer, langsamerer Zugriff
✓ Feste Größe, kein Header	✗ Enthalten Object-Header + Felder
✓ Speichernahe Verarbeitung (CPU-Cache-freundlich)	✗ Speicherfragmentierung, Garbage Collection nötig

03.

Value Objects

Wie sehen sie aus, was sind ihre Vorteile und Herausforderungen?



46



Vergleich

Liste von Punkten

```
[
  0 ; 0
  0 ; 1
  1 ; 0
  0 ; 1
  1 ; 0
  1 ; 1
]
```

Mögliche Implementierung

zwei Arrays von int-Werten

```
int[] xs; int[] ys;
```

Array von Point-Objekten

```
Point[] ps;
```

Array von Point Values

```
value record Point(int x, int y)
```

embarc.de

Value Objects

47

47

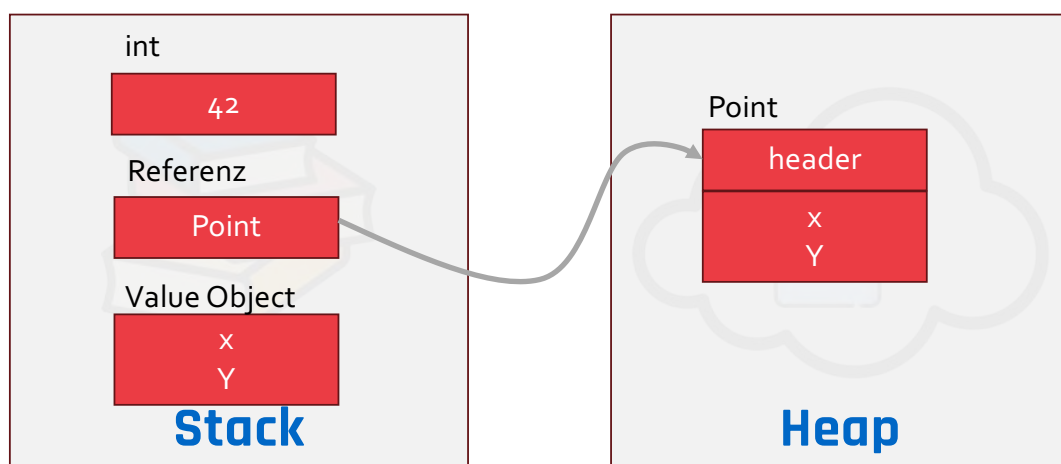


```
// value class Point { .. }

value record Point(int x, int y);

Point[] points = new Point[10];

for (int i = 0; i < 10; i++) {
    points[i] = new Point(i, i);
}
```





- Value Classes und Felder sind **final**, dürfen aber von Interfaces ableiten
- Value Classes haben **keine Identität**
- für **==** und **hashCode** werden die **Felder verwendet**
- **kein synchronized** erlaubt
- **JVM kann** (muss aber nicht) **Speicherlayout** und **Performance optimieren**

```
public value record Point(int x, int y) {
    public Point moveBy(int dx, int dy) {
        return new Point(x + dx, y + dy);
    }
}
```



Variante	Speicher (Byte)	Overhead durch Header & Referenzen?
Zwei int[]	112	Minimal
Point[] mit Objekten	332	Hoch (Referenzen, Header)
Point[] mit Value Type	96	Fast optimal
ArrayList<Point> mit Objekten	364	Noch mehr Overhead
ArrayList<Point> mit Value Type	128	Etwas mehr als Point[]

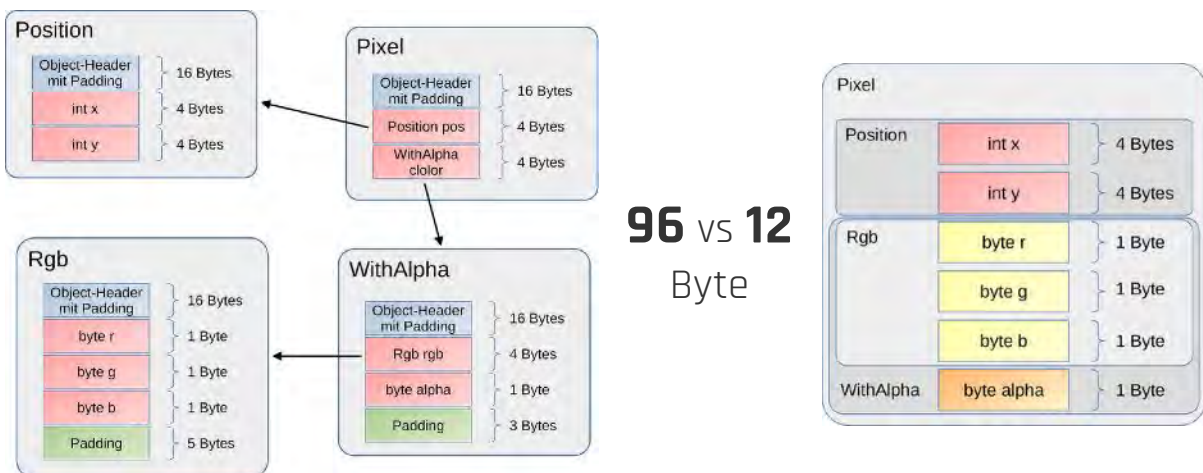


Was sparen wir mit Value Objects?

- deutlich **weniger Speicher** (Overhead von Objektenreferenzen entfällt)
- **schnellerer, direkter Zugriff**, da die Daten näher an der CPU liegen (**besserer Cache-Zugriff**)
- automatisches Abräumen im Stack, **keine Garbage Collection**



```
record Pixel(Position pos, ColorWithAlpha color)
record Position(int x, int y)
record ColorWithAlpha(Rgb rgb, byte alpha)
record Rgb(byte r, byte g, byte b)
```



Arno Haase: "Ist Java noch zeitgemäß? Eine Einordnung" (Java Magazin 6.2025)



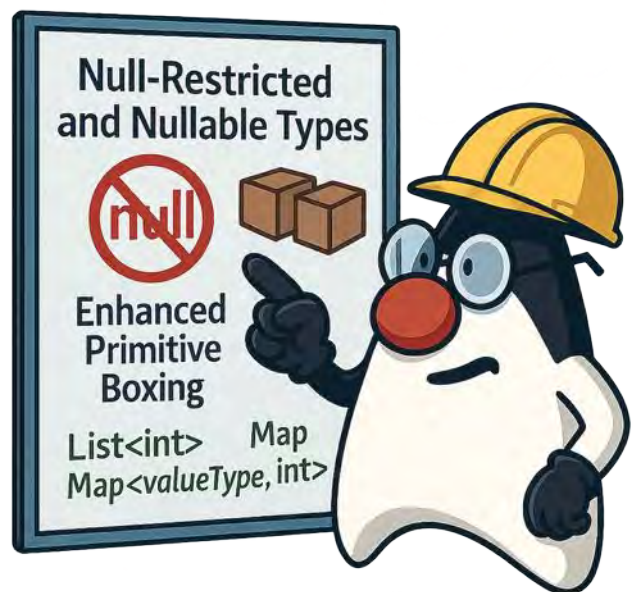
Vergleich Record vs. Value Record

	Record	Value Record
Speicherlayout	Referenz auf Objekt im Heap	Flach eingebettet (z. B. im Stack, Array)
Nullbarkeit	Kann null sein	Nicht null, wenn inline verwendet
Identität	Hat Objektidentität (==, Monitor)	Keine Objektidentität, kein Monitor
Synchronisierung	Möglich	Nicht erlaubt
getClass()	Gibt tatsächliche Klasse zurück	Nicht erlaubt bei inline usage
Semantik	Referenz-Semantik	Wert-Semantik (wie int, long)

04.

Was kommt noch?

Mit welchen Themen müssen wir uns noch beschäftigen?



und sicher mehr ...

JEP 402: Enhanced Primitive Boxing

JEP yy: Null-Restricted Value Types

JEP xx: Null-Restricted Types

JEP 401: Value Classes Java 25 26?

57

und sicher mehr ...

JEP 402: Enhanced Primitive Boxing

JEP yy: Null-Restricted Value Types

JEP xx: Null-Restricted Types

JEP 401: Value Classes

58



Dürfen Value Type Variablen null sein?

```
value class Point {
    int x, y;
}

Point p = null; // ✅ erlaubt
```

- weil Java **abwärts-kompatibel** bleiben muss
- keine neuen Konzepte wie „by-value“-Variablen
- **Value Types** als Objekte ohne Identität **in das bestehende System integriert**



Null-Restricted Types

- **Nullbarkeit** von Typen **explizit deklarieren**
- Ziele:
 - Typensicherheit verbessern, NullPointerExceptions vermeiden
 - Explizite Angabe von null-Fähigkeit im Code (ähnlich wie in Kotlin oder C#)
 - Konvertierungen mit Warnungen bei potenziell unsicherem Umgang mit null
 - Kompatibilität: Sicherstellung der Interoperabilität mit bestehendem Java-Code, der keine Aussagen über die Nullbarkeit seiner Typen trifft (ohne Quell- oder Binärinkompatibilitäten)



Null-Restricted Types

- diese Variable darf niemals null sein:

```
Point! p = new Point(1, 2); // 
Point! q = null;           //  Compile error
```

Wenn null erlaubt ist:

- Muss die JVM **immer Speicher für die Referenz** reservieren (oft 4 oder 8 Byte).
- Flattening und Inlining** ist **nicht möglich**, weil null ein Sonderfall ist.

Wenn null **nicht erlaubt** ist (null-restricted):

- Kann die JVM die Daten **direkt einbetten** (z. B. in ein Array).
- Performance und Speicherverbrauch** werden massiv verbessert.



Null-Restricted Types

Point! - null-restricted: null ist nicht erlaubt

Point? - nullable: null ist erlaubt

Point - keine Angabe: Verhalten wie bisher



63



embarc.de

Null-Restricted Value Class Types

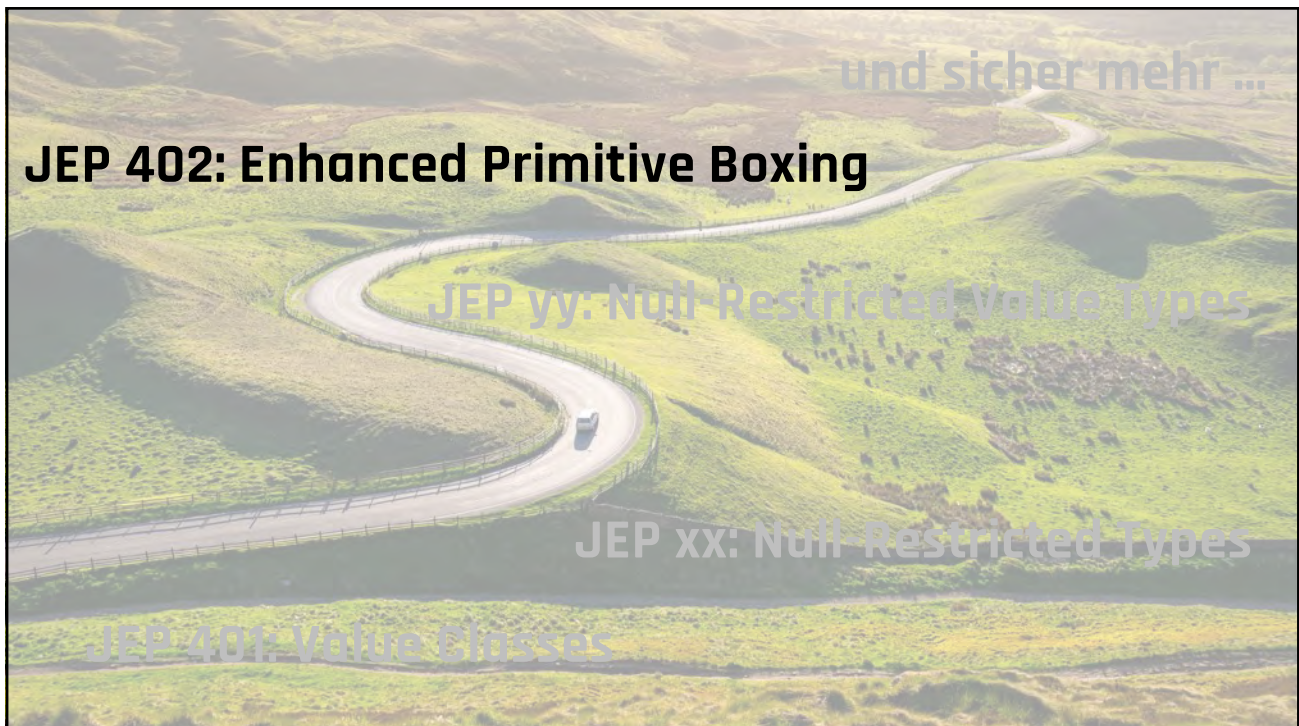
- **Nur für Value Classes:** definiert einen neuen Typ, der garantiert nicht null sein
- Dadurch kann die JVM:
 - Flattening im Speicher durchführen
 - Null-Checks vermeiden
 - Leistungsoptimierungen aktivieren

```
// garantiert niemals null
__ByValue Point p = new Point(1, 2);
```

Value Objects

64

64



65



embarc.de

Value Objects

66

Background: How We Got the Generics We Have
(Or, how I learned to stop worrying and love erasure)

Brian Goetz
June 2020

Before we can talk about where generics are going, we first have to talk about where they are, and how they got there. This document will focus primarily on how we arrived at the generics we have now, and why, as a means of setting the groundwork for how the generics we have now will influence the "better" generics we are trying to build.

In particular, we emphasize that erasure was in fact the sensible and pragmatic choice for adding generics to Java in 2004—and many of the forces that led us to choose translation by erasure may still be operating today.

Erasure

Ask any developer about Java generics, and you'll likely get an angry (though often uninformed) rant about *erasure*. Erasure is probably the most

<https://openjdk.org/projects/valhalla/design-notes/in-defense-of-erasure>

66



Entscheidung für Erasure bei der Einführung von Generics in Java eine **bewusste Abwägung** zwischen **Kompatibilität**, **Einfachheit** und den **technischen Möglichkeiten** der damaligen Zeit

Mit den Fortschritten in Project Valhalla werden nun Wege erkundet, die Vorteile von Generics weiter auszubauen und ihre **bisherigen Einschränkungen zu adressieren**.



JEP 402: Enhanced Primitive Boxing

Ziel: flexibleres Autoboxing für primitive Typen durch Unterstützung neuer boxbarer Typen als Value Classes.

Heute unterstützt Java **nur die acht klassischen Wrapper-Klassen** (Integer, Double, usw.) **beim Autoboxing**.

Neue Value Classes aus Project Valhalla können **nicht automatisch geboxed** werden. Dadurch ist die Verwendung z. B. in **generischen Collections eingeschränkt** (List<T>, Optional<T>, etc.).

```
@PrimitiveBox
value class UInt {
    int value;
```

05.

Fazit + Ausblick

Zusammenfassung,
Referenzen und weitere Infos



69



embarc.de

Was und wie?

Final & immutable

Equality by state (nicht Identität)

Keine Synchronisation

JVM-Optimierungen möglich (Flattening, Interning,
Scalar Replacement)

Beispiele: **Integer, LocalDate, Optional**

Value Objects

70

70



Warum?

- Sprechende Datentypen **selbst definieren** (DDD)
- **Lücke** zw. **primitiven** Datentypen und **Objekten** verringern
 - Value Based Classes (Integer, Optional, ...) in Value Classes umbauen
- Value Types auch als **generische Parameter** (List<int>)
- **bessere Performance, weniger Speicherverbrauch**



30 Jahre - die Java-Welt
und das Ökosystem sind
lebendig wie nie!



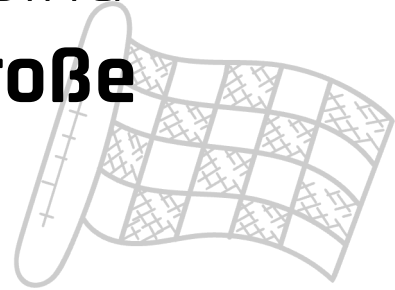


embarc.de

Value Objects

73

Value Objects sind
**das nächste große
Thema.**



73

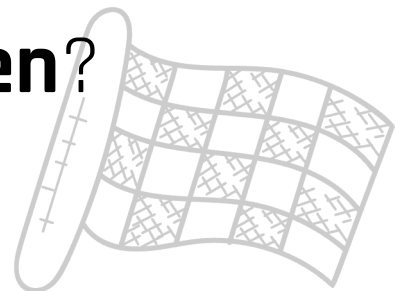


embarc.de

Value Objects

74

Valhalla, **when?**



74



Jetzt schon ausprobieren ... (1)



<https://javaalmanac.io/jdk/valhalla/>

75



Jetzt schon ausprobieren ... (2)

```
> git clone https://github.com/openjdk/valhalla/
> cd valhalla
> bash configure
> make images
```

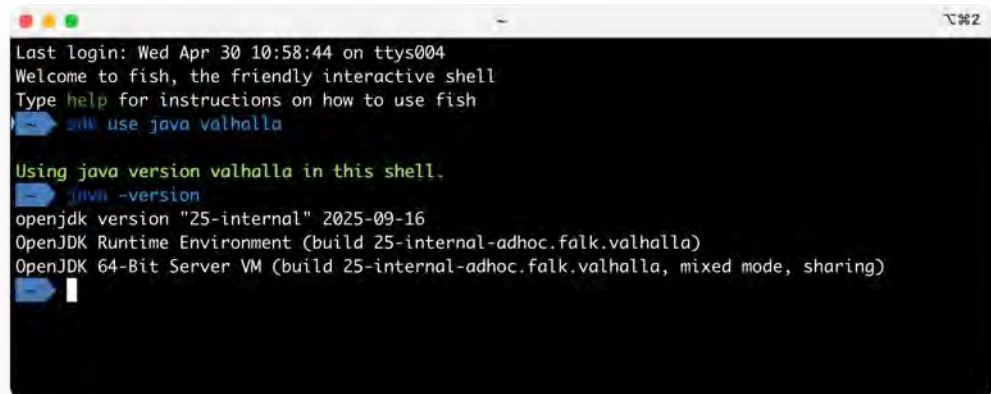


76



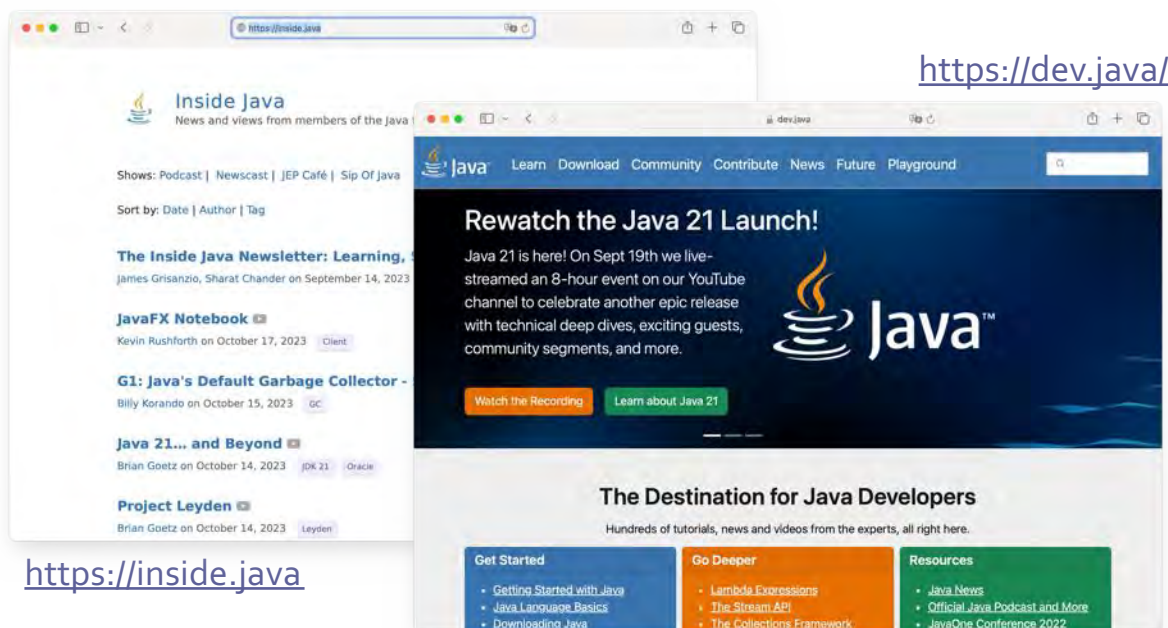
Mit sdkman umschalten ...

```
> sdk install java valhalla
/path/to/valhalla/build/macosx-x86_64-server-release/images/jdk
```



```
Last login: Wed Apr 30 10:58:44 on ttys004
Welcome to fish, the friendly interactive shell
Type help for instructions on how to use fish
> sdk use java valhalla

Using java version valhalla in this shell.
> java -version
openjdk version "25-internal" 2025-09-16
OpenJDK Runtime Environment (build 25-internal-adhoc.falk.valhalla)
OpenJDK 64-Bit Server VM (build 25-internal-adhoc.falk.valhalla, mixed mode, sharing)
```

<https://inside.java>

<https://dev.java/>

The Destination for Java Developers

Hundreds of tutorials, news and videos from the experts, all right here.

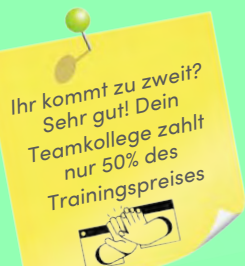
Get Started	Go Deeper	Resources
- Getting Started with Java - Java Language Basics - Downloading Java	- Lambda Expressions - The Stream API - The Collections Framework	- Java News - Official Java Podcast and More - JavaOne Conference 2022



Java Update Trainings

9 bis 11, 17, 21, 25

Sprecht mich an 

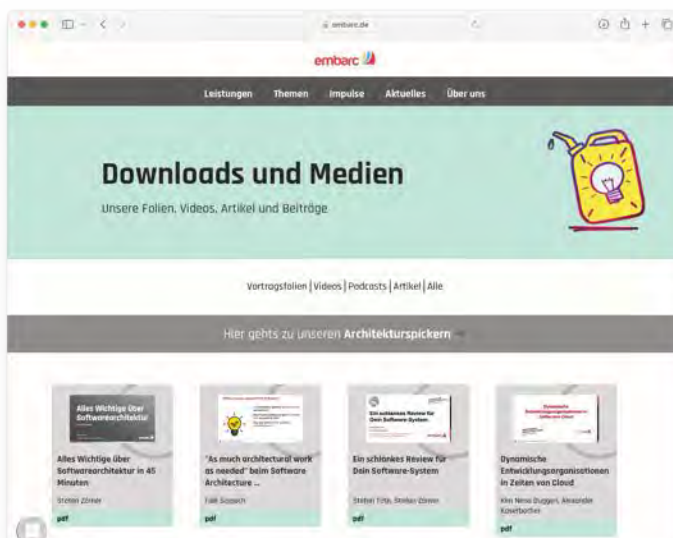


Unser Preismodell – Flexibel und transparent

Interesse an
iSAQB-
Weiterbil-
dungen?

79

Folien als PDF zum Download



➔ embarc.de/download/

80



Architektur-Spicker



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

Architektur-Spicker #15
Documentation-as-Code einsetzen



PDF, 4 Seiten
Kostenloser Download.

→ architektur-spicker.de/



Vielen Dank.

Ich freue mich auf Eure Fragen!

- ✉ falk.sippach@embarc.de
- in [linkedin.com/in/falk-sippach](https://www.linkedin.com/in/falk-sippach)
- X [@sippsack](https://twitter.com/sippsack)
- m [@sippsack@ijug.social](https://www.instagram.com/sippsack)





embarc.de

Falk Sippach

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

Schwerpunkte

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



Value Objects

84