

Value Classes

Java bereitet sich auf KI-Workloads vor

Falk Sippach

embarc · @sipp sack

BaselOne · 15.10.2026





Kanton Basel-Stadt



+++Ba
sel++
++Te+
+++ch

Basel+
basel.ch



optravis
company of msg

cyon



swiss made
software

ICT
Scouts
Wir fördern Zukunft

SD:>_
[swissdroids.ch]

#BaselOne26

baselone.ch

Abstract

Value Classes – Java bereitet sich auf KI-Workloads vor

Java verdankt seine Stärke seit jeher einem statischen, starken Typsystem. Doch die alte Trennung zwischen primitiven und Referenztypen ist bis heute eine Sollbruchstelle der Sprache: primitive Typen sind schnell und kompakt, integrieren sich aber nur schlecht mit Generics, Collections, Streams und modernen Sprachmitteln. Klassen sind ausdrucksstark, kosten aber oft unnötig Speicher, Indirektion und Performance.

Mit Value Classes will Project Valhalla diese Kluft schließen. Sie erlauben Datentypen ohne eigene Identität, die sich fachlich wie Klassen modellieren lassen, zur Laufzeit aber deutlich näher an primitiven Typen liegen. Das verspricht kompaktere Datenstrukturen, weniger Objekt-Overhead, bessere Datenlokalität und damit genau die Eigenschaften, die auch für numerische und KI-nahe Workloads immer wichtiger werden.

Denn wer mit Embeddings, Vektoren, Matrizen oder großen Mengen kleiner, unveränderlicher Datenstrukturen arbeitet, profitiert besonders von effizientem Speicherlayout und geringerer Indirektion. Gleichzeitig werden fachliche Typen wieder attraktiver: präzise, unveränderbar und trotzdem performant. Der Vortrag zeigt, wohin sich Java mit Value Classes, null-restricted / nullable types und einem vereinheitlichten Typmodell entwickelt und warum das gerade für die Zukunft von Java im Umfeld von AI, Data Processing und High-Performance wichtig wird.

Falk Sippach

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

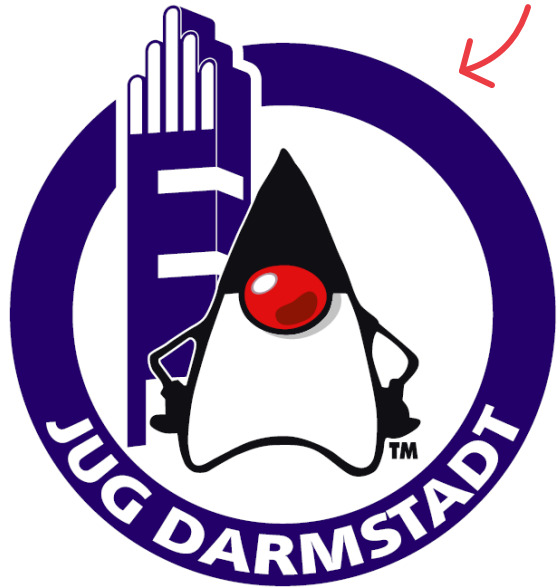
Schwerpunkte

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



Teil der Java Community

Nächster Termin
am **29.10.2026**



jug-da.de

monatliche Opensource
Code Camps



cyberland.ijug.eu



Valhalla, when?

Valhalla, now! (or at least very soon)



Brian Goetz · Java Language Architect

Project Valhalla: ein langer Anlauf

2014

Projektstart

Klassenabstraktion und primitiveähnliche Effizienz zusammenbringen.

2017

Minimal Value Types

Früher JVM-Prototyp: identitätslose Daten praktisch erproben.

2018-2019

L-World

Value-Typen enger in das bestehende Referenzmodell integrieren.

2025

JEP-401-Early-Access

Value Classes und Objects mit einem vollständigen EA-Build testen.

2026

Integration ins reguläre JDK

JEP 401 und 539 sind für JDK 28 integriert.

März 2027

JDK 28: Preview geplant

Erster Schritt – weitere Valhalla-Bausteine folgen separat.

Unser Weg durch Valhalla



1 • Hello Value Class

Syntax, Erzeugung und Klassenregeln



2 • Werte statt Identität

Veränderbarkeit, Identität und Gleichheit



3 • Optimierungen der JVM

Flattening, Scalarization und ihre Grenzen



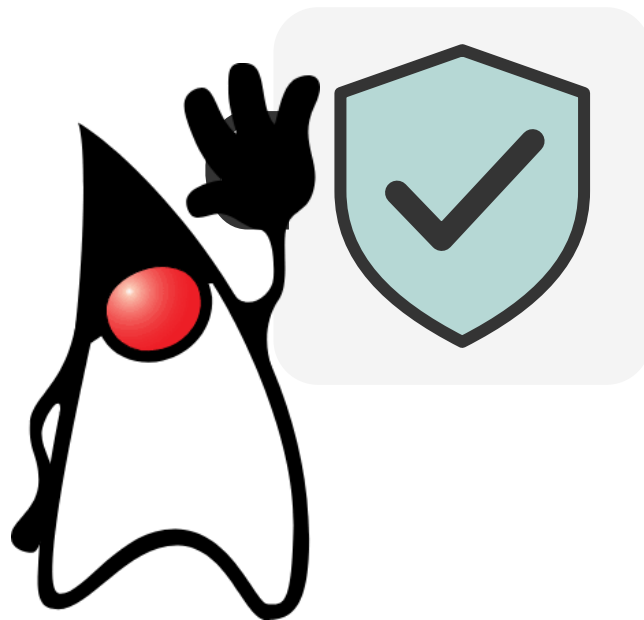
4 • Nutzen für KI und Fachmodelle

Numerische Daten und präzise Typen



5 • Status und Ausblick

JEPs, Einstieg und nächste Schritte



Unser Weg durch Valhalla

Unser Weg durch Valhalla



1 • Hello Value Class

Syntax, Erzeugung und Klassenregeln



2 • Werte statt Identität

Veränderbarkeit, Identität und Gleichheit



3 • Optimierungen der JVM

Flattening, Scalarization und ihre Grenzen



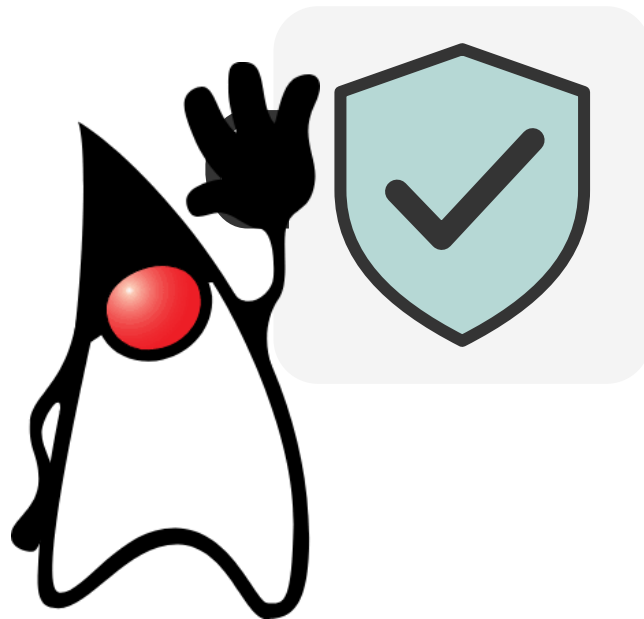
4 • Nutzen für KI und Fachmodelle

Numerische Daten und präzise Typen



5 • Status und Ausblick

JEPs, Einstieg und nächste Schritte



Kapitel 1 von 5

Hello Value Record

```
value record Rgb(byte r, byte g, byte b) {}

var red = new Rgb((byte) 255, (byte) 0, (byte) 0);
int channel = Byte.toUnsignedInt(red.r()); // 255
```



Konstruktor, Accessors und Record-Methoden bleiben vertraut.
Neu: **keine Objektidentität**, unveränderliche Feldbelegung.

Value Classes: Klassen bleiben Klassen

```
interface HasRed { int red(); }
value class Red implements HasRed {
    private byte r;
    Red(byte r) { this.r = r; }
    public int red() {
        int channel = Byte.toUnsignedInt(r);
        return channel;
    }
}

var color = new Red((byte) 255);
color = new Red((byte) 128);
```

Klassenmittel bleiben

Konstruktoren, Methoden, Kapselung, Interfaces.

Eigene Instanzfelder: final

Primitive und Referenztypen erlaubt.

Variable $\neq$ Value Object

Die Variable darf einen anderen Wert erhalten.

Konkrete Klasse: final

Oberklasse: Object oder abstract value class.

Unveränderlich - aber Shallow Immutable

```
value record NamedColor(Rgb rgb, List<String> names) {
    NamedColor {
        names = List.copyOf(names);
    }
}

var names = new ArrayList<>(List.of("Rot", "Rouge"));
var red = new NamedColor(
    new Rgb((byte) 255, (byte) 0, (byte) 0), names);

names.add("Rosso"); // red.names() bleibt unverändert
red.names().add("Rojo"); // UnsupportedOperationException
```

Finale Feldreferenz $\neq$ unveränderlicher Inhalt.

Hier schützt `List.copyOf` die Liste; `String` ist selbst immutable.

Erst initialisieren. Dann sichtbar werden.

```
value class Label {
    private final String name;

    Label(String name) {
        this.name = name;
        super();
        System.out.println(this);
    }

    public String toString() { return "Label[" + name + "]; }
}
```

Felder initialisieren, bevor die Instanz sichtbar wird.
JEP 513 + 539 liefern den Unterbau.

Unser Weg durch Valhalla



1 • Hello Value Class

Syntax, Erzeugung und Klassenregeln



2 • Werte statt Identität

Veränderbarkeit, Identität und Gleichheit



3 • Optimierungen der JVM

Flattening, Scalarization und ihre Grenzen



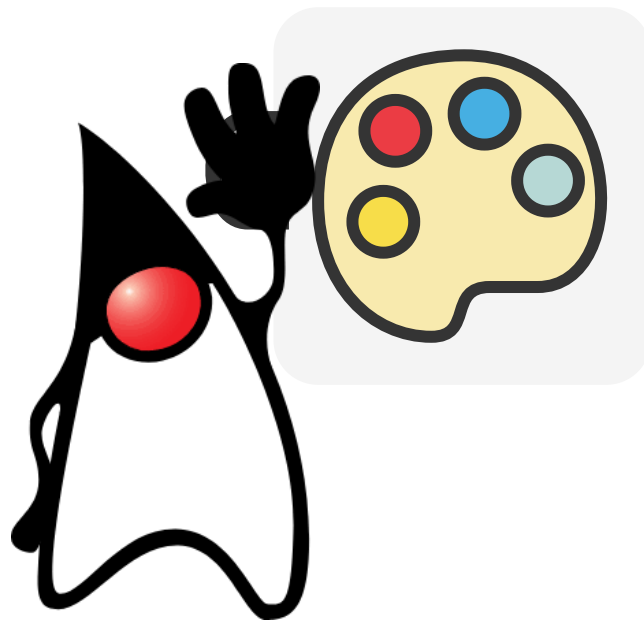
4 • Nutzen für KI und Fachmodelle

Numerische Daten und präzise Typen



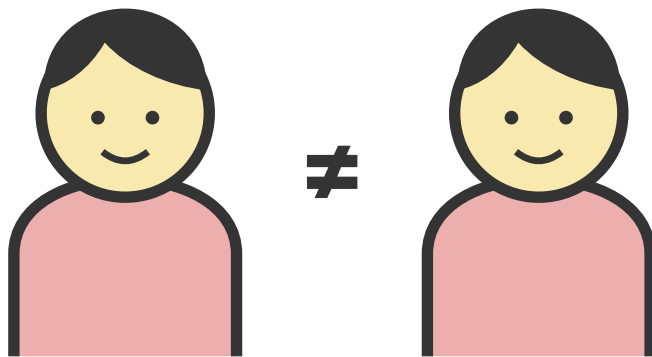
5 • Status und Ausblick

JEPs, Einstieg und nächste Schritte



Kapitel 2 von 5

Das Gleiche ist nicht dasselbe



Anna

Bea

Gleich aussehen ...

Zwillinge können sich gleichen.

... ist nicht dieselbe Person.

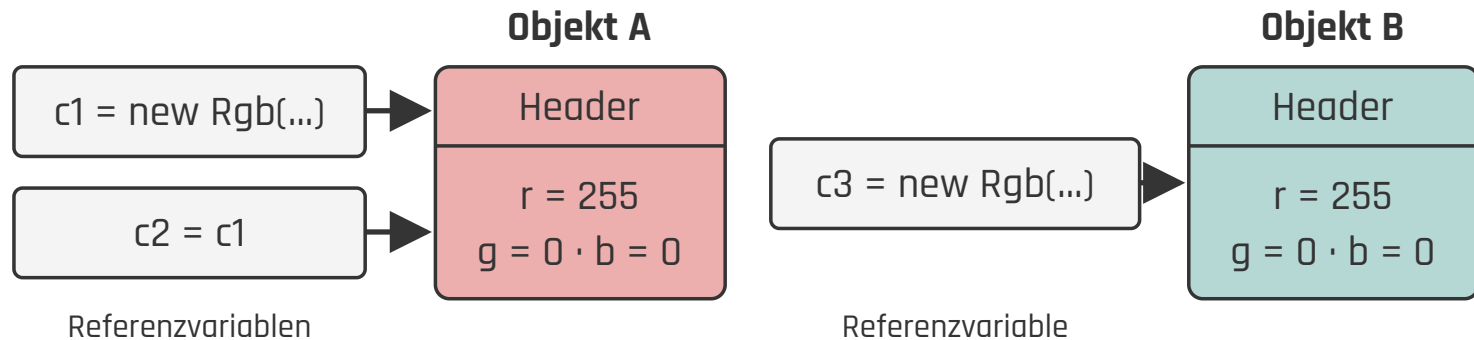
Anna bleibt Anna. Bea bleibt Bea.

Identität: Welches Objekt?

Gleichheit: Nach welchen Kriterien gleich?

Identität: dasselbe oder nur gleich?

`record Rgb(byte r, byte g, byte b) {}` – ein gewöhnlicher Record mit Identität



`c1 == c2`
true
dasselbe Objekt

`c1 == c3`
false
zwei Objekte

`c1.equals(c3)`
true
gleiche Kanalwerte

Gleiche Kanäle: zwei Vergleichsmodelle

Identity Record

```
record Rgb(byte r, byte g, byte b) {}
// Zwei separat erzeugte rote Farben:
a == b;           // false
a.equals(b);     // true
```

Value Record

```
value record Rgb(
    byte r, byte g, byte b) {}
// Dieselben Kanalwerte:
a == b;           // true
a.equals(b);     // true
```

Bei Identity Objects prüft `==` die Identität.

Bei Value Objects: gleiche Klasse und rekursiv ununterscheidbare Feldwerte.

NamedColor: equals ist nicht gleich ==

```
value record NamedColor(Rgb rgb, String name) {}
```

```
var a = new NamedColor(  
    new Rgb((byte) 255, (byte) 0, (byte) 0), new String("Rot"));
```

```
var b = new NamedColor(  
    new Rgb((byte) 255, (byte) 0, (byte) 0), new String("Rot"));
```

```
a.equals(b); // true: generiertes Record-equals  
a == b;      // false: unterschiedliche String-Identitäten
```

`equals` : Textinhalt · rekursives `==` : String-Identität

Die Grenze heißt Identität

```
var color = new Rgb((byte) 255, (byte) 0, (byte) 0);
```

```
synchronized (color) {}      // Compilerfehler
```

```
Object lock = color;
```

```
synchronized (lock) {}      // Fehler zur Laufzeit
```

```
new WeakReference<>(color); // ebenfalls nicht unterstützt
```

Unser Weg durch Valhalla



1 • Hello Value Class

Syntax, Erzeugung und Klassenregeln



2 • Werte statt Identität

Veränderbarkeit, Identität und Gleichheit



3 • Optimierungen der JVM

Flattening, Scalarization und ihre Grenzen



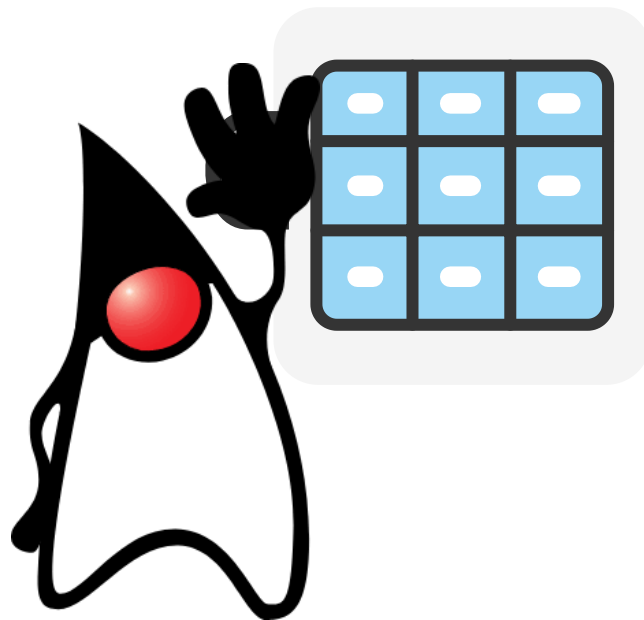
4 • Nutzen für KI und Fachmodelle

Numerische Daten und präzise Typen



5 • Status und Ausblick

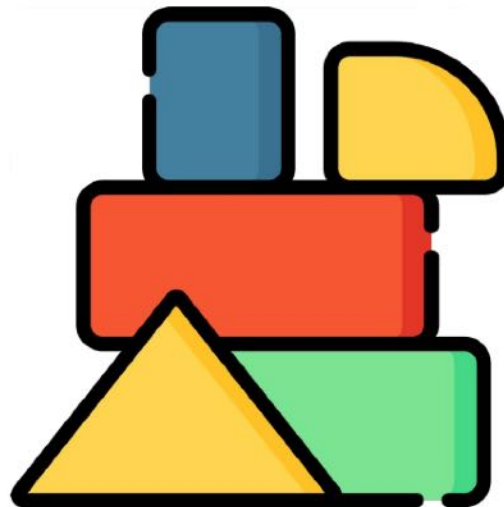
JEPs, Einstieg und nächste Schritte



Kapitel 3 von 5



Heap **Flattening**



Scalarization



Flattening maximiert **Dichte** in Datenstrukturen (Arrays, ...)

`Point[] -> x1,y1,x2,y2,...`



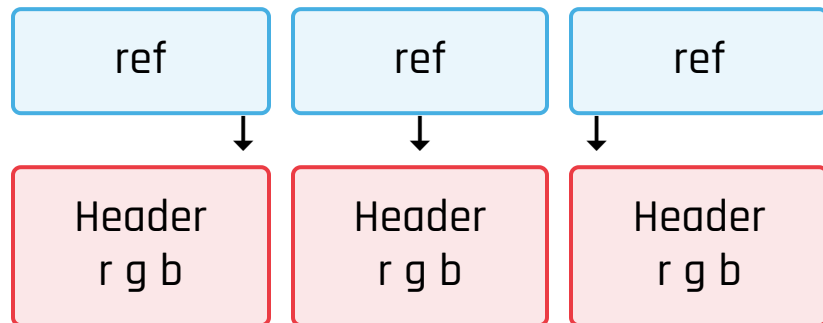
Inlining minimiert **Indirektion** innerhalb zusammengesetzter Objekte

```
Rectangle {
    Point topLeft;    -> x1,y1,x2,y2
    Point bottomRight;
}
```

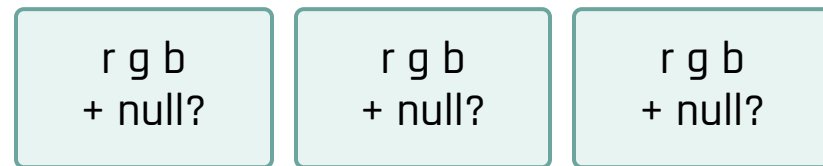
Flattening: Nutzdaten statt Umweg

```
Rgb[] pixels = new Rgb[1_000_000];
```

Referenzdarstellung



Mögliche flache Darstellung



Weniger separate Objekte,
weniger Indirektion, bessere Lokalität.

Vom Rgb zum Pixel: vier Typen, ein Modell

Identity Records

```
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) {}
```

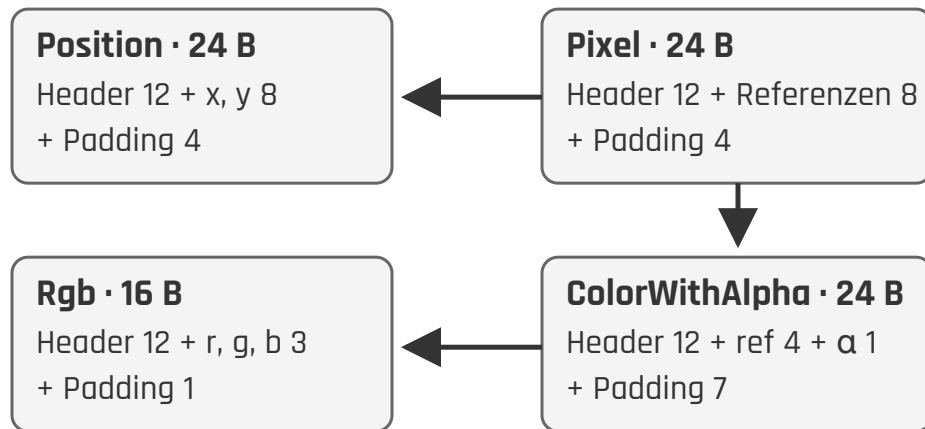
```
var pixel = new Pixel(new Position(10, 20),
                     new ColorWithAlpha(new Rgb((byte) 255, (byte) 0, (byte) 0), (byte) 255));
```

Value Records

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

Pixel: Objektgraph oder kompakte Werte?

Identity Objects: vier separate Objekte

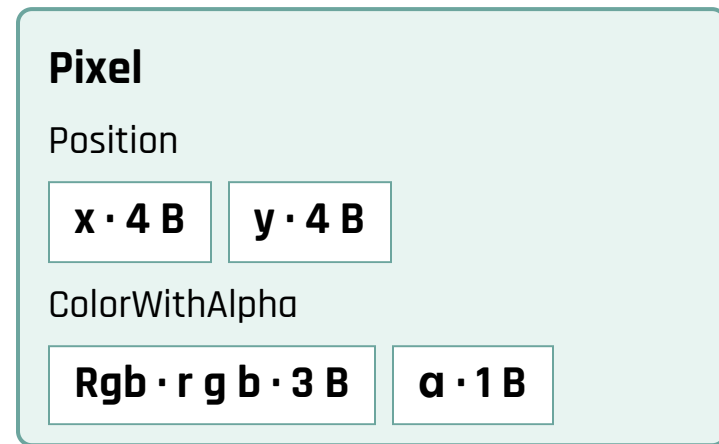


88 Bytes 4 Header · 3 interne Referenzen

Links: Header 12 B · Referenz 4 B · Ausrichtung 8 B · kein Sharing · ohne äußere Referenz.

Rechts: keine Layoutgarantie. Nullstatus, Padding und Atomarität begrenzen das Flattening.

Value Objects: idealisiert flach



12 Bytes reine Nutzdaten

Scalarization: nur die Feldwerte

Unser Java-Code

```

Rgb color = new Rgb(
    (byte) 255, (byte) 0, (byte) 0);

int red = Byte.toUnsignedInt(color.r());
    
```

Sinngemäß optimiert

```

byte r = (byte) 255;

int red = Byte.toUnsignedInt(r);
    
```

Für diese Rechnung reicht der Rotkanal.
Ein Rgb-Objekt muss dafür nicht entstehen.

Was kostet null?

```
Rgb color = null;
Rgb[] colors = new Rgb[10];

System.out.println(colors[0]); // null
```

Ein Encoding muss alle RGB-Werte **und zusätzlich null** darstellen.

Null-Marker, Padding, reservierte Bitmuster ...
Die konkrete Speicherung entscheidet die JVM.

Und was kostet Atomarität?

```
value record Vector2(double x, double y) {}

class Particle {
    Vector2 velocity;
}
```

Thread A schreibt

(1.0, 2.0)

Thread B darf keinen
zerrissenen Mischwert lesen.

Aktuelles Flattening ist durch atomare Encodings begrenzt.
Kein pauschales „alle Value-Arrays werden flach“.

Kompakter $\neq$ in jedem Fall schneller

Kleiner Datenbestand

Dekodieren kann zusätzliche Rechenarbeit bedeuten.

Großer Datenbestand

Weniger Pointer Chasing, bessere Cache-Nutzung.

JMH: Warm-up, Forks und Ergebnisse konsumieren.

JFR: Allokationen und GC beobachten.

Unser Weg durch Valhalla



1 • Hello Value Class

Syntax, Erzeugung und Klassenregeln



2 • Werte statt Identität

Veränderbarkeit, Identität und Gleichheit



3 • Optimierungen der JVM

Flattening, Scalarization und ihre Grenzen



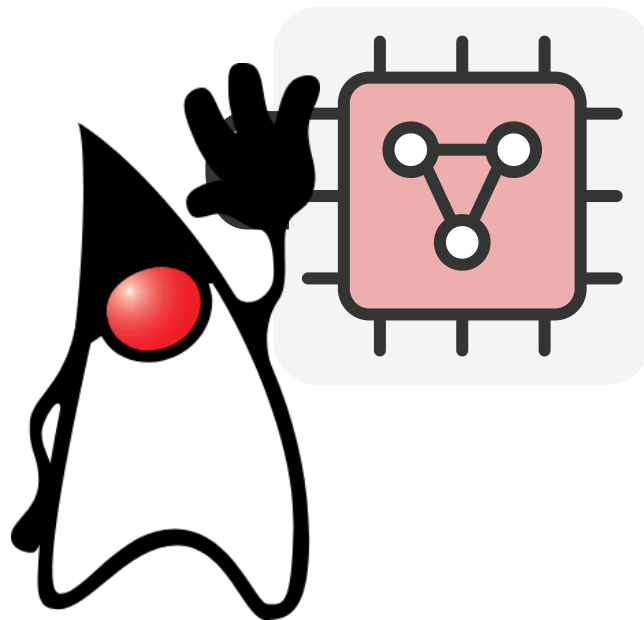
4 • Nutzen für KI und Fachmodelle

Numerische Daten und präzise Typen



5 • Status und Ausblick

JEPs, Einstieg und nächste Schritte



Kapitel 4 von 5

Java im KI-Zeitalter: Was fehlt noch?



Viele Daten, wenig Overhead

Kompakte Typen, bessere Lokalität

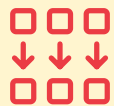
Valhalla → Value Classes



Native Bibliotheken anbinden

Funktionen und Speicher erreichen

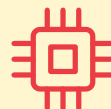
Panama → FFM API · jextract



Mehr rechnen pro CPU-Takt

Viele Zahlen gleichzeitig verarbeiten

Vector API → SIMD



GPUs aus Java nutzen

Java-Code analysieren und übertragen

Babylon → Code Reflection · HAT

Bausteine mit unterschiedlicher Reife: FFM final · Vector API Inkubator · Valhalla Preview-Ziel · Babylon/HAT experimentell

Warum passt das zu KI-Workloads?

```
value record BFloat16(short bits) {}
value record Vector2(float x, float y) {}
```

Fachliche Typen statt
unbeschrifteter Bitmuster

Viele kleine Werte:
Speicherverkehr wird relevant

Bessere Typabstraktion – kein automatisches SIMD oder GPU.

Fachliche Typen werden attraktiver

```
value record CustomerId(long value) {}
value record OrderId(long value) {}

void loadCustomer(CustomerId id) { /* ... */ }

loadCustomer(new CustomerId(42)); // passt
loadCustomer(new OrderId(42));    // Compilerfehler
```

Unser Weg durch Valhalla



1 • Hello Value Class

Syntax, Erzeugung und Klassenregeln



2 • Werte statt Identität

Veränderbarkeit, Identität und Gleichheit



3 • Optimierungen der JVM

Flattening, Scalarization und ihre Grenzen



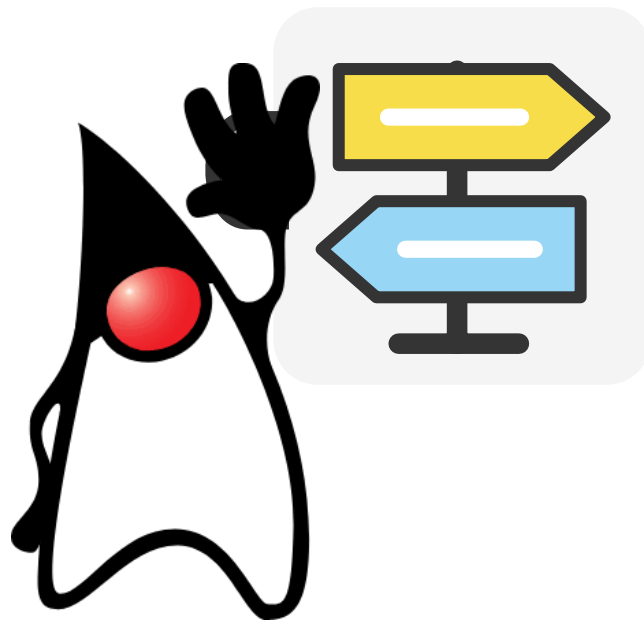
4 • Nutzen für KI und Fachmodelle

Numerische Daten und präzise Typen



5 • Status und Ausblick

JEPs, Einstieg und nächste Schritte



Kapitel 5 von 5



Was bringt welcher JEP?

Jetzt relevant

JEP	Rolle	Einordnung
401	Value Classes und Objects	Preview in JDK 28
539	Strict Field Initialization in der JVM	Preview-Unterbau

Bereits ausgelieferte Vorarbeiten

JEP	Infrastruktur / Migration	Java
181	Nest-Based Access Control	11
309	Dynamic Class-File Constants	11
334	JVM Constants API	12
371	Hidden Classes	15
390	Warnings for Value-Based Classes	16
513	Flexible Constructor Bodies	25

JDK 28 geplant für März 2027 · Preview ≠ finales Sprachfeature.
Vorarbeiten: gemeinsame Infrastruktur; JEP 390 bereitet die Migration vor.

Bekannte Klassen werden Value Classes

```
Integer a = 1996, b = 1996;
System.out.println(a == b); // true im beschriebenen Preview

var day = LocalDate.of(2026, 10, 15);
System.out.println(day == day.plusDays(1).minusDays(1)); // true
```

Im Preview unter anderem Wrapper, Optional und java.time-Typen.
String bleibt Identity Class. Preview-Aktivierung beeinflusst das Verhalten.

Was noch nicht dazugehört



**Spezialisierte
Generics**



**Weitere Null-/Default-
und Layoutmodelle**



**Primitive-Class-
Exploration**

Draft: Null-Restricted and Nullable Types

Draft: Null-Restricted Value Class Types

JEP 218: Generics over Primitive Types

Außerdem: JEP 402 · Enhanced Primitive Boxing · Valhalla-Roadmap · JEP-Index

Entwürfe und langfristige Ziele – nicht Lieferumfang von JEP 401.

Präzise Typen. Kompakte Daten.

Was wir modellieren wollen

Farben, Messwerte, Beträge,
Vektorkomponenten ...

Was die Hardware bewegt

Bytes, Cache Lines,
Referenzen und Nutzdaten

Muss gute Modellierung unnötigen Overhead kosten?

Drei Dinge zum Mitnehmen



Value Classes geben Identität auf

– nicht das Klassenmodell.



Flattening und Scalarization

sind unterschiedliche Freiheiten der JVM.



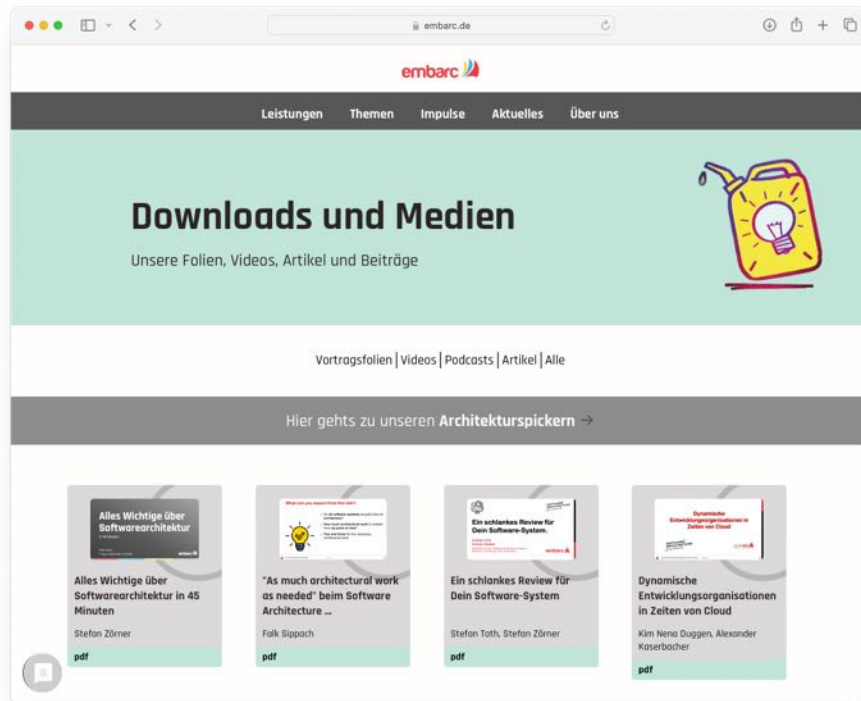
Bessere Datenmodelle können effizient werden.

Messen bleibt nötig.

Quellen und Vertiefung

- [JEP 401: Value Objects](#)
- [JEP 539: Strict Field Initialization](#)
- [JEP 513: Flexible Constructor Bodies](#)
- [OpenJDK: Value Objects – Performance und Grenzen](#)
- [Dan Smith: Try Out JEP 401](#)
- [Johan Sjölen: Value Classes Still Need Compiler Sympathy](#)
- [Cay Horstmann: Project Valhalla](#)
- [Nicolai Parlog: Valhalla, Now!](#)
- [José Paumard / Rémi Forax: Devoxx Belgium 2026](#)

Folien als PDF zum Download

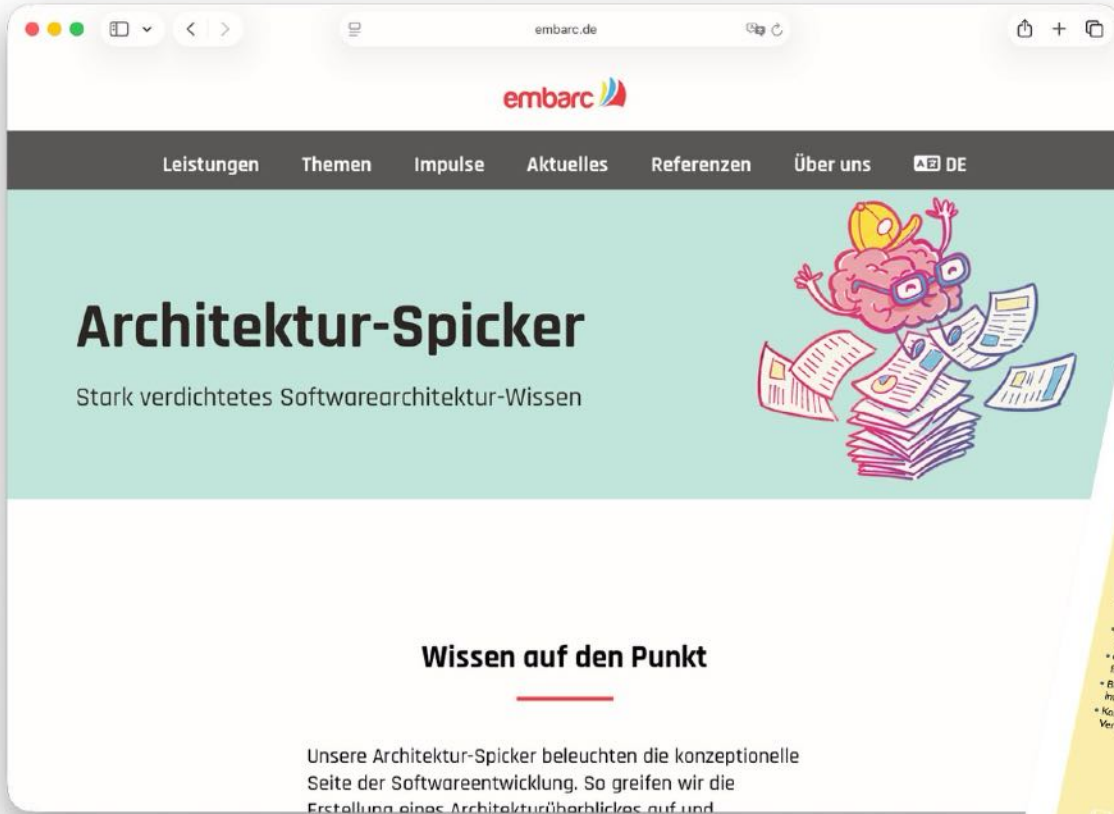


embarc.de/download/

GitHub-Repo zum Vortrag

github.com/sippsack/java-value-classes

- [sli.dev](#) Folien im Ordner [slides](#)
 - Live unter <https://sippsack.github.io/java-value-classes/>
- Demos im Ordner [demo/src](#)



Kostenloser Download:



[embarc.de/architektur-spicker](https://architektur-spicker.de)



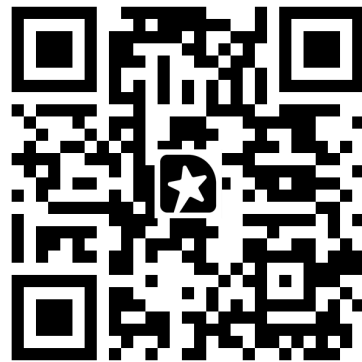
Vielen Dank.

Ich freue mich auf Eure Fragen!

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



Feedback ↓



Value Classes - Java bereitet sich auf KI-Workloads vor

