Value Objects

Das nächste große Ding in Java?

Falk Sippach

JUG SAXONY DAY Radebeul, 26. September 2025

embarc

0

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.

1

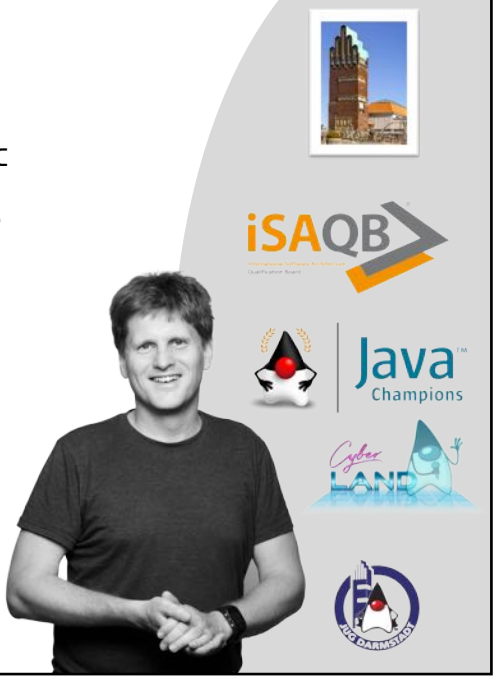
Falk Sippach

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

Schwerpunkte

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

embarc



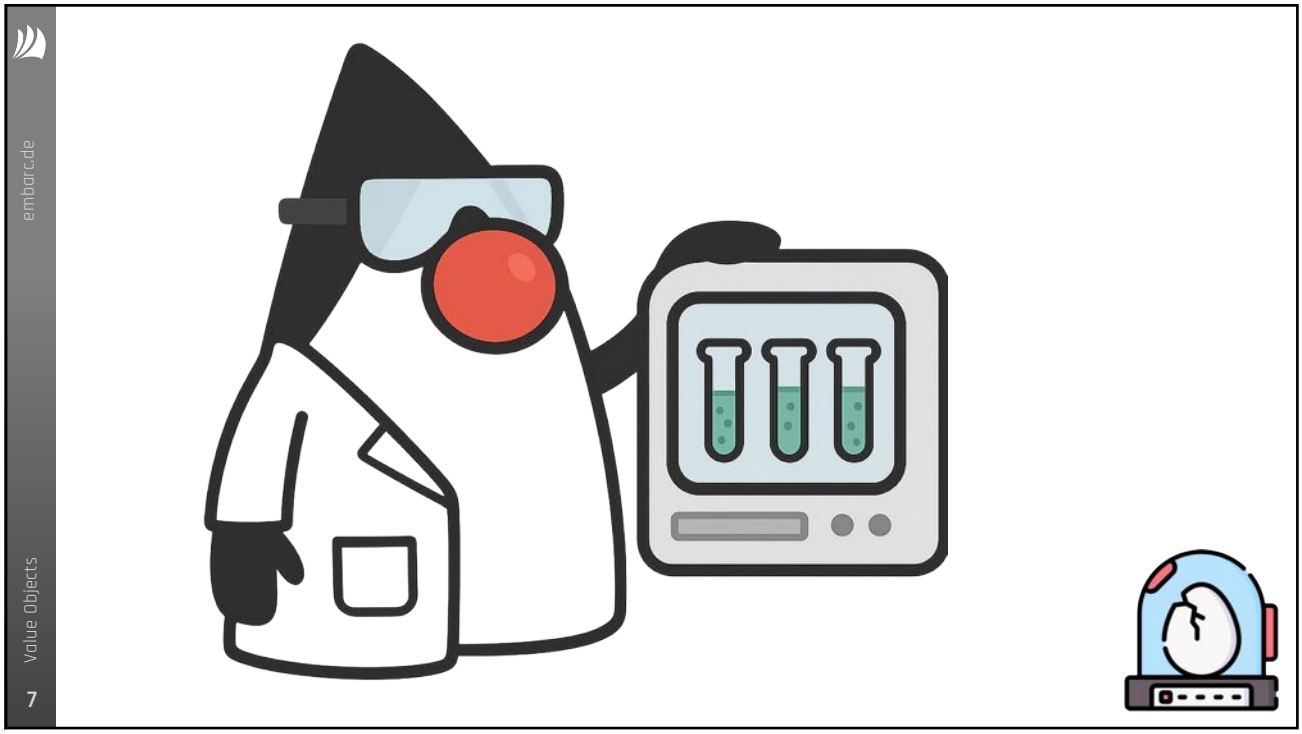
Teil der Java Community

Nächster Termin am 30.09..

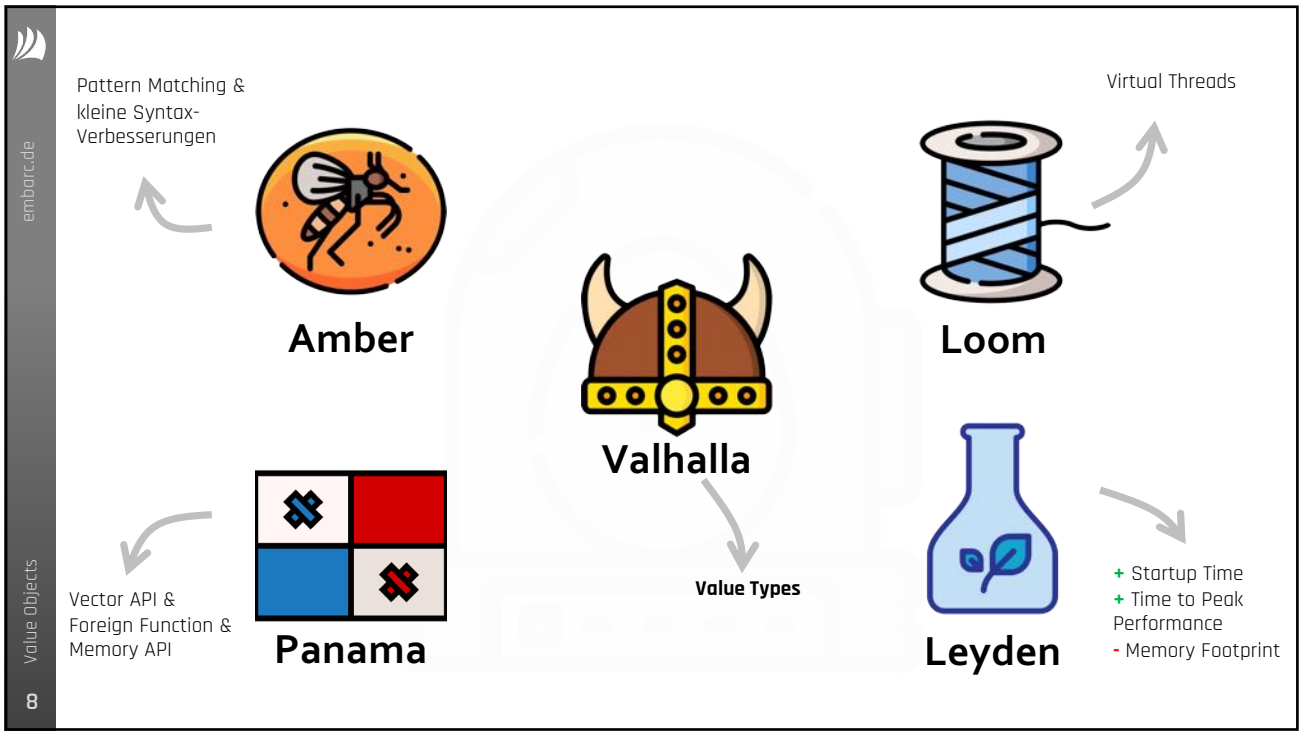
monatliche Opensource Code Camps

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

https://cyberland.ijug.eu/


embarc.de
Value Objects
7

7

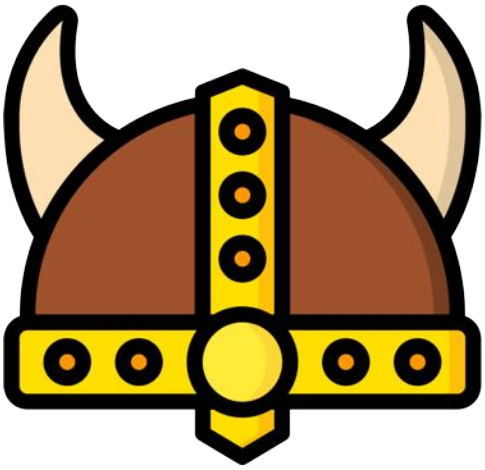

embarc.de
Value Objects
8
Pattern Matching &
kleine Syntax-
Verbesserungen
Amber
Virtual Threads
Loom
Valhalla
Value Types
Vector API &
Foreign Function &
Memory API
Panama
Leyden
+ Startup Time
+ Time to Peak
Performance
- Memory Footprint

8

embarc

nordische Mythologie:

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



Wortspiel:
Val = Value

Valhalla =
Halle der Werte
(Behauptung von ChatGPT)

9

00.

Agenda

Was euch erwartet

01 Project Valhalla

02 Primitives vs. Objects

03 Value Objects

04 Was kommt noch?

05 Fazit + Ausblick

11

01.

Project Valhalla

Java meets nordische Mythologie

12

embarc.de

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/)

13

Das Motto

>> **Codes** like a **class**, **works** like an **int**. <<

State of Valhalla: Part 1: The Road to Valhalla (2021)

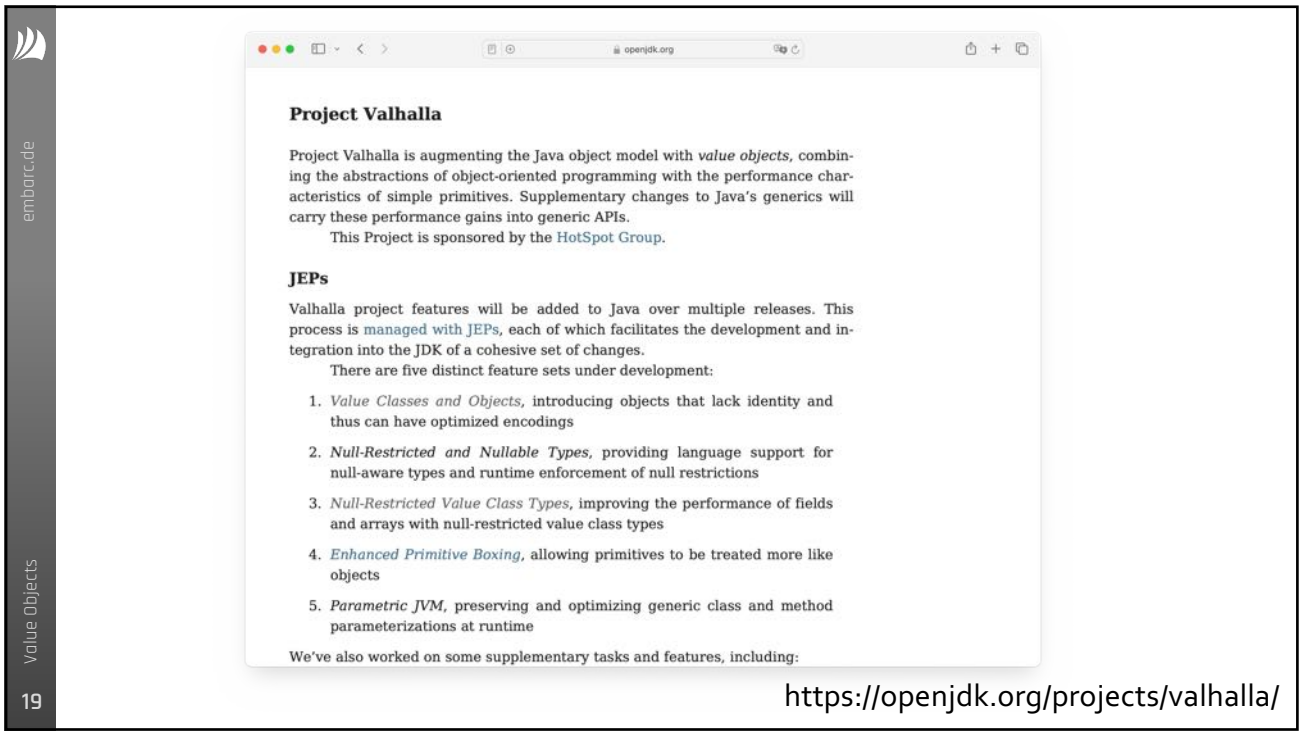
14

Java's EPIC!! Refactor

15


"**Deep cut** through user model, language, JVM"
(Brian Goetz)

16


embarc.de
Value Objects
19
openjdk.org
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/

19

Bereits ausgeliefert …

JEP 181

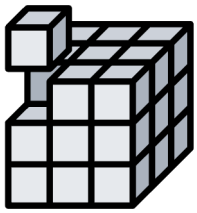
Nest-Based Access Control

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**

21

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**

Java **11**

22

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**

Java **11**

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**

Java **12**

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**

Java **15**

25

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 Nuzungsverhalten einfordern**, um zukünftig die Kompatibilität nicht zu brechen

Java **16**

26

02.

Primitives Types vs. Objects

Altlasten des Java Memory Models

27

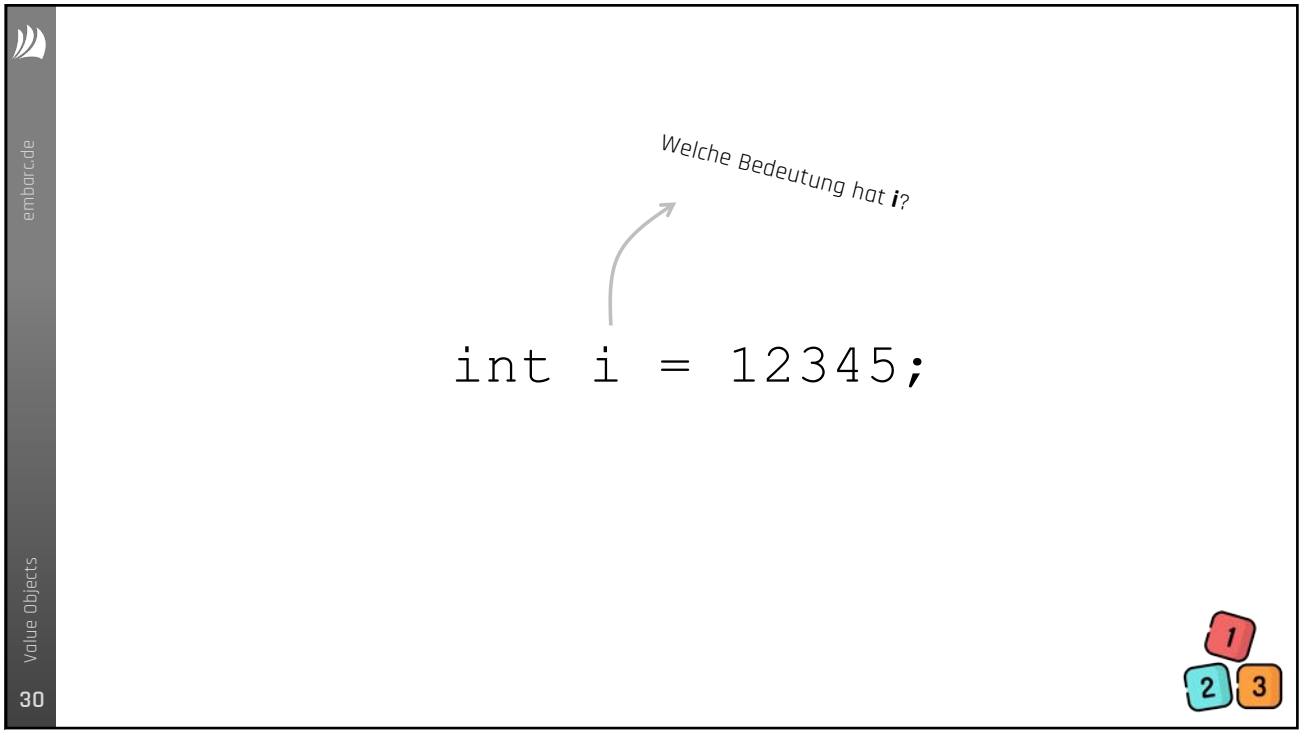
Primitive Types

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

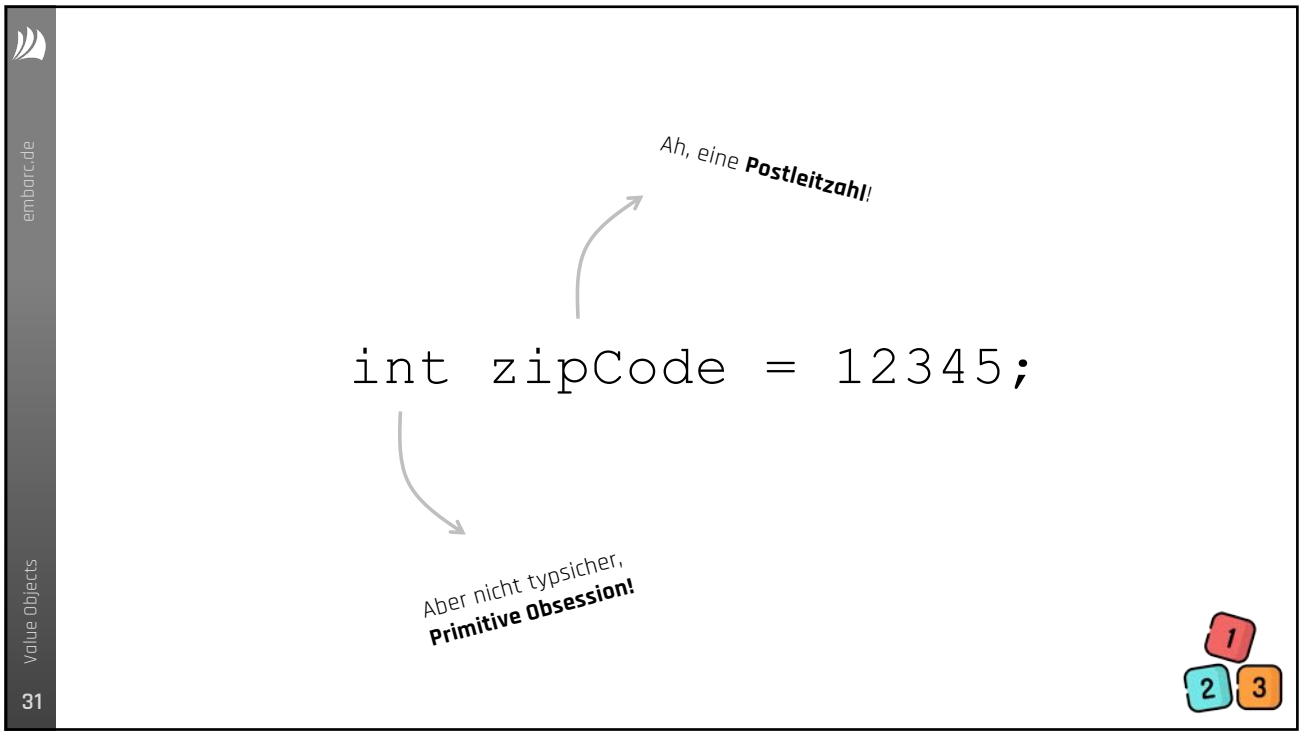
```
int i = 12345;
```

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

29


embarc.de
Value Objects
30
Welche Bedeutung hat **i**?
int i = 12345;

30


embarc.de
Value Objects
31
Ah, eine **Postleitzahl**!
int zipCode = 12345;
Aber nicht typsicher,
Primitive Obsession!

31

Immer noch
Primitive Obsession!

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

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

32


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

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

Gefahr des **Vertauschens**.

33

Primitive Types

Object References

34

Stack

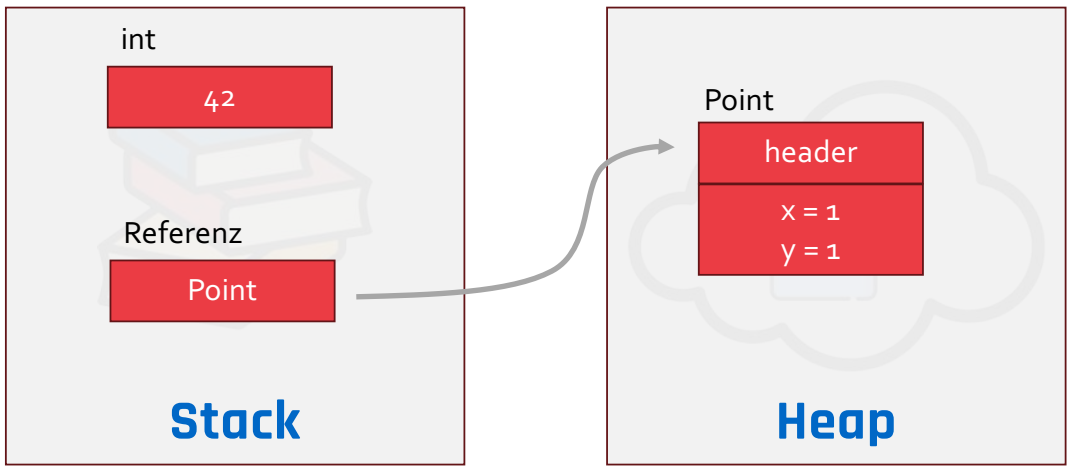
Heap

36

Die folgenden Zahlen hat GenAI ermittelt!

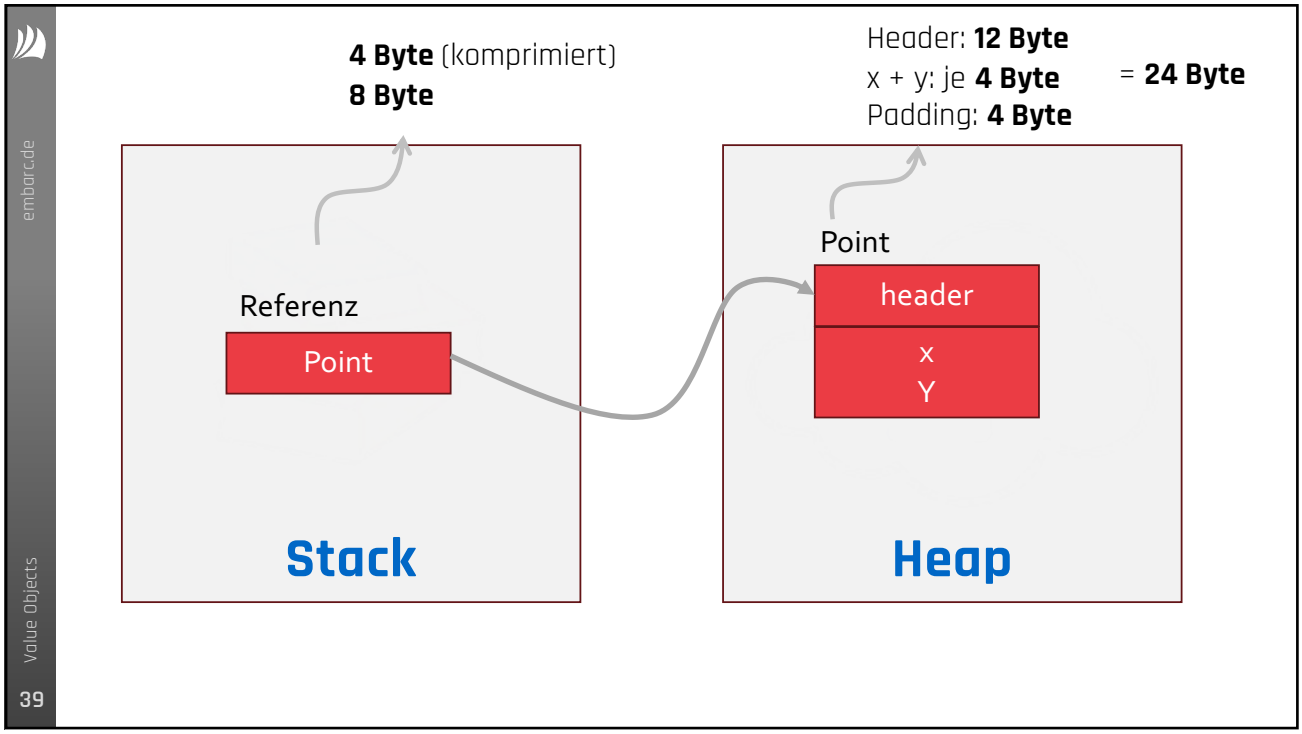
37


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




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


38



39

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**.

40

Vergleich

Liste von Punkten

$$\begin{bmatrix} 0 & ; & 0 \\ 0 & ; & 1 \\ 1 & ; & 0 \\ 0 & ; & 1 \\ 1 & ; & 0 \\ 1 & ; & 1 \end{bmatrix}$$

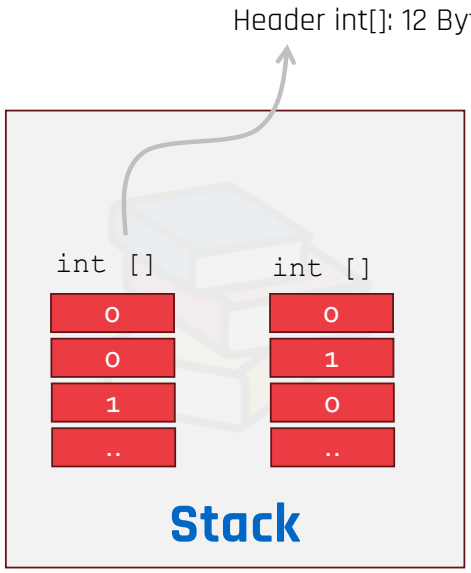
Mögliche Implementierung

effizient, aber schlecht wartbar

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

Array von Point-Objekten

```
Point[] ps;
```

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

Speicherverbrauch:

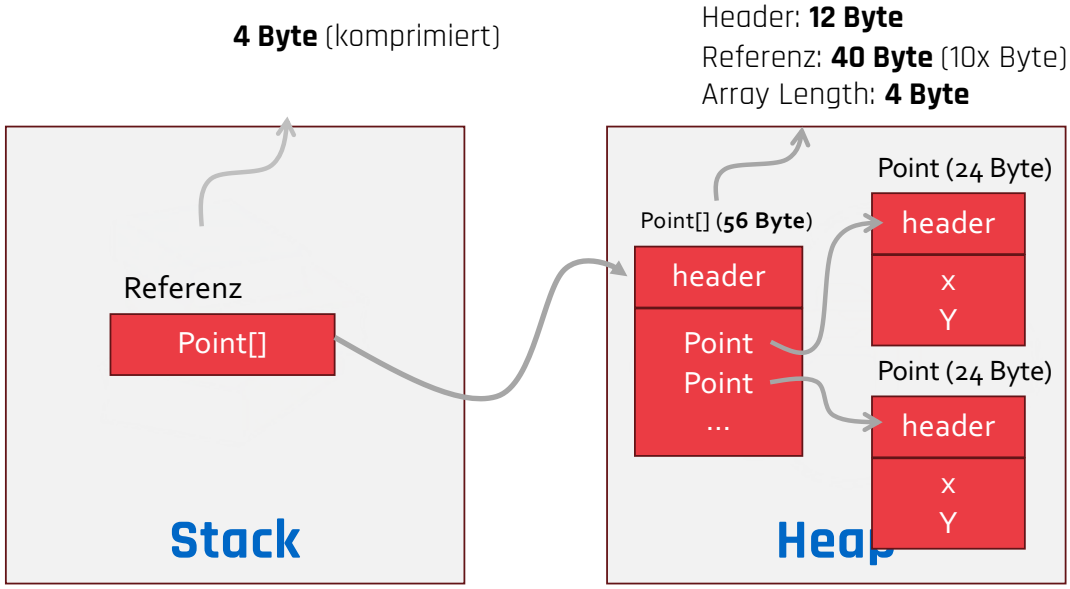
- Header von int[]: **12 Byte**
- Daten: 10 × 4 Byte = **40 Byte**
- Padding auf 8-Byte-Grenze: **4 Byte**

⏩ 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();
}
```


44




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


45

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

46

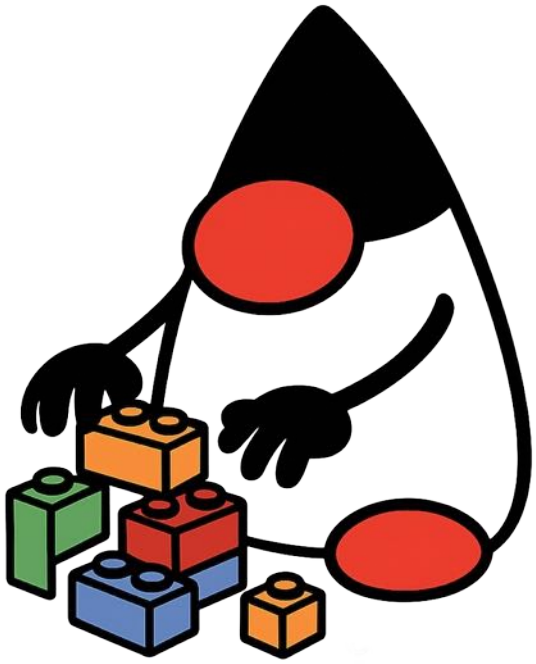
Primitive Types (int, double, boolean, ...)	**Object References** (String, Arrays, Point, ...)
✓ Direkt im Stack gespeichert (lokale Variablen)	✗ Liegen auf dem Heap, nur Referenz im Stack
✓ Keine Indirektion, schnelle Zugriffe	✗ Indirektion durch Pointer, langsamerer Zugriff
✓ Feste Größe, kein Header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

47

03.

Value Objects

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



48

Vergleich

Liste von Punkten

$$\begin{bmatrix} 0 & ; & 0 \\ 0 & ; & 1 \\ 1 & ; & 0 \\ 0 & ; & 1 \\ 1 & ; & 0 \\ 1 & ; & 1 \end{bmatrix}$$

Mögliche Implementierung

zwei Arrays von int-Werten

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

Array von Point-Objekten

```
Point[] ps;
```

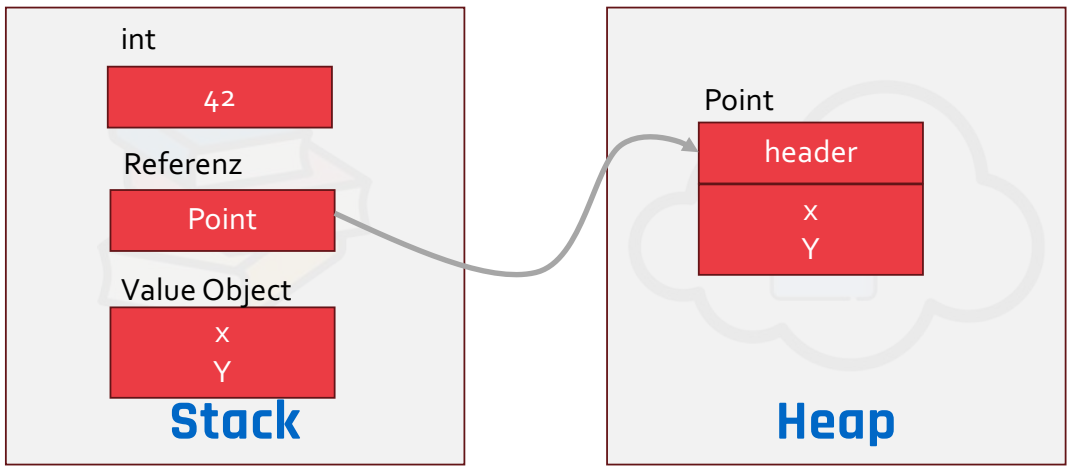
Array von Point Values

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

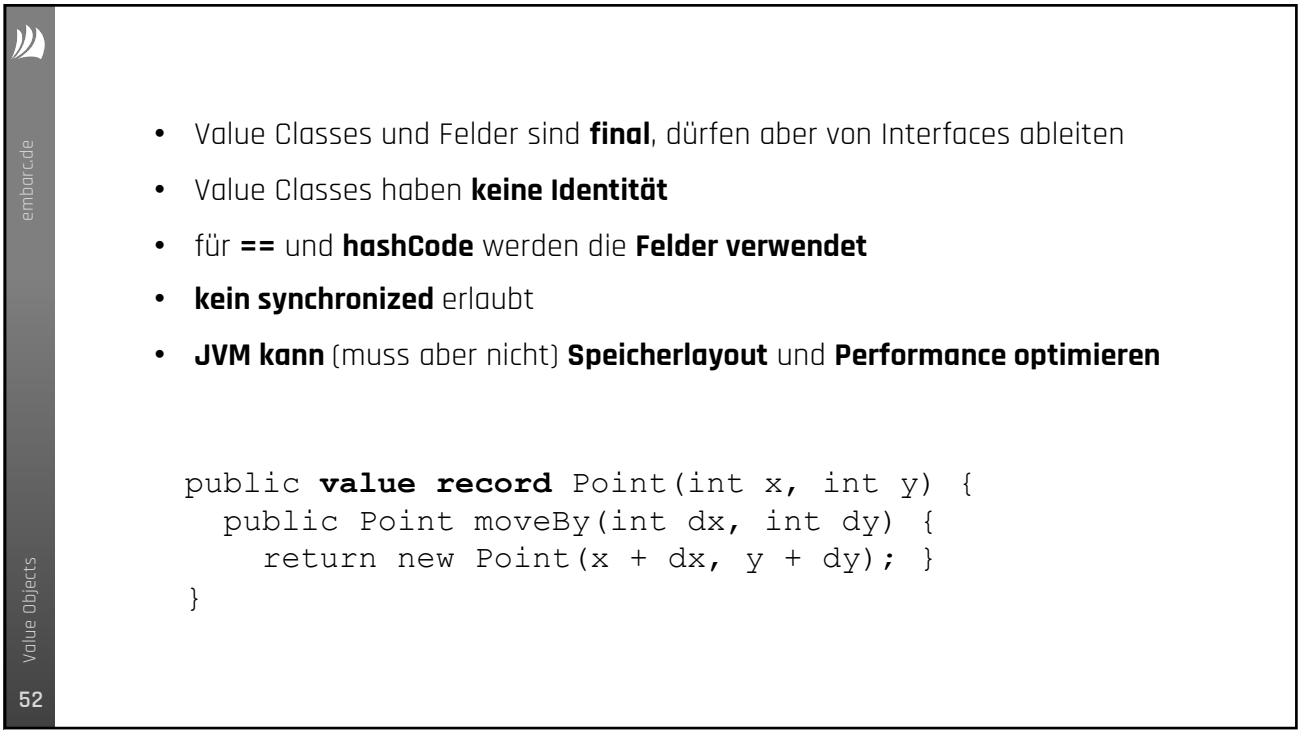

49

```
// 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);
}
```


50



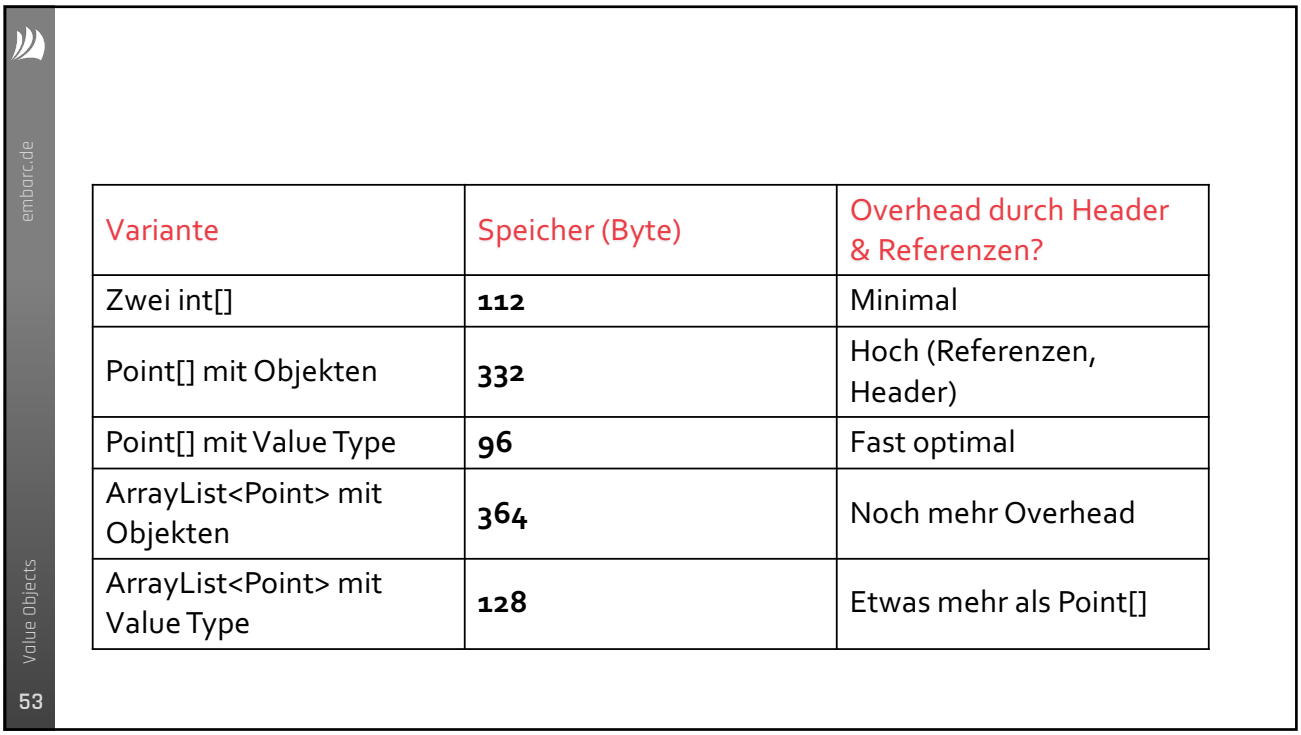
51



- 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); }
}
```

52



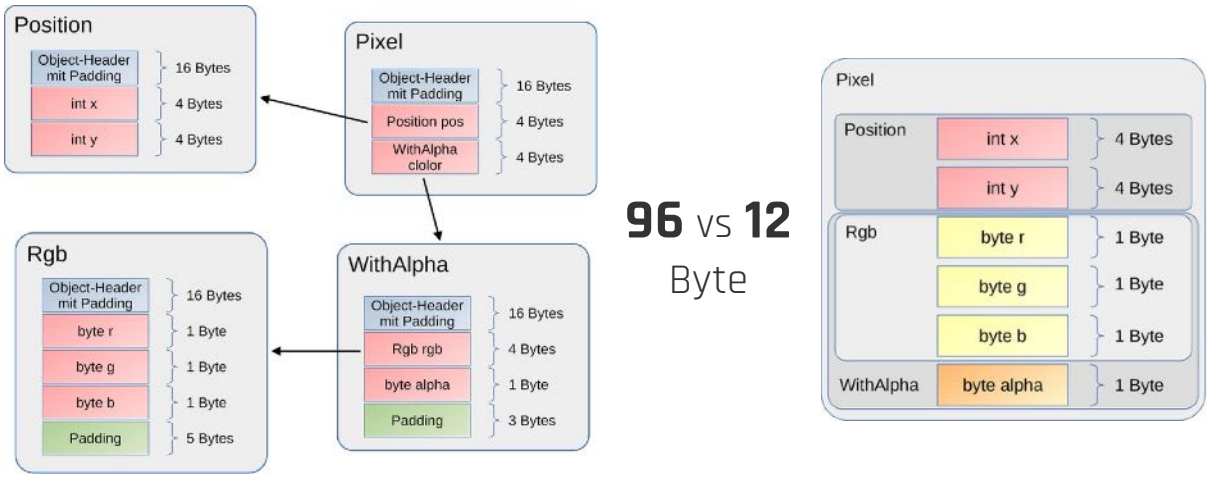
Variante	Speicher (Byte)	Overhead durch Header & Referenzen?
Zwei int[]	**112**	Minimal
Point[] mit Objekten	**332**	Hoch (Referenzen, Header)
Point[] mit Value Type	**96**	Fast optimal
ArrayList<Point> mit Objekten	**364**	Noch mehr Overhead
ArrayList<Point> mit Value Type	**128**	Etwas mehr als Point[]

53

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, byåte b)
```



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

Vergleich Record vs. Value Class/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)

56

Braucht es Records dann noch?

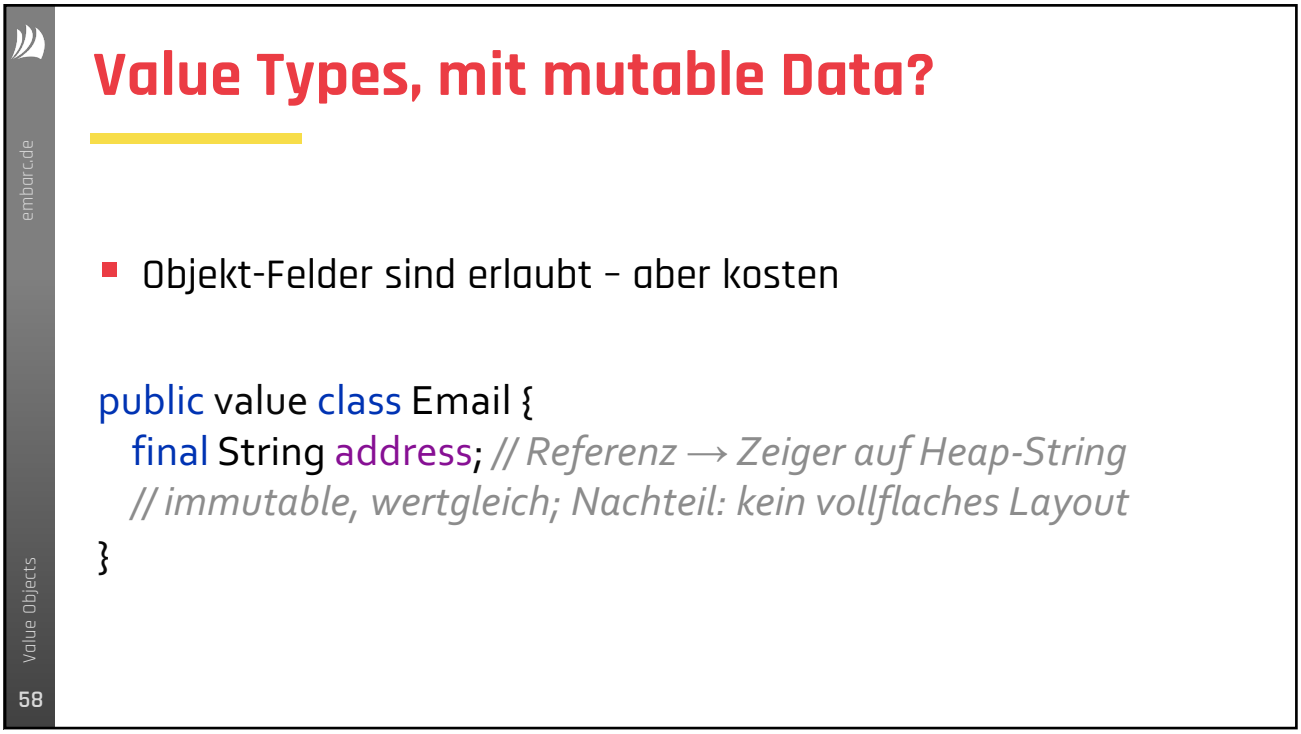
Oder können wir alle Records in Value Records umbauen?

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

```
record Node<T>(
    T value,
    List<Node<T>> children
) { }
```

➡ ?

57

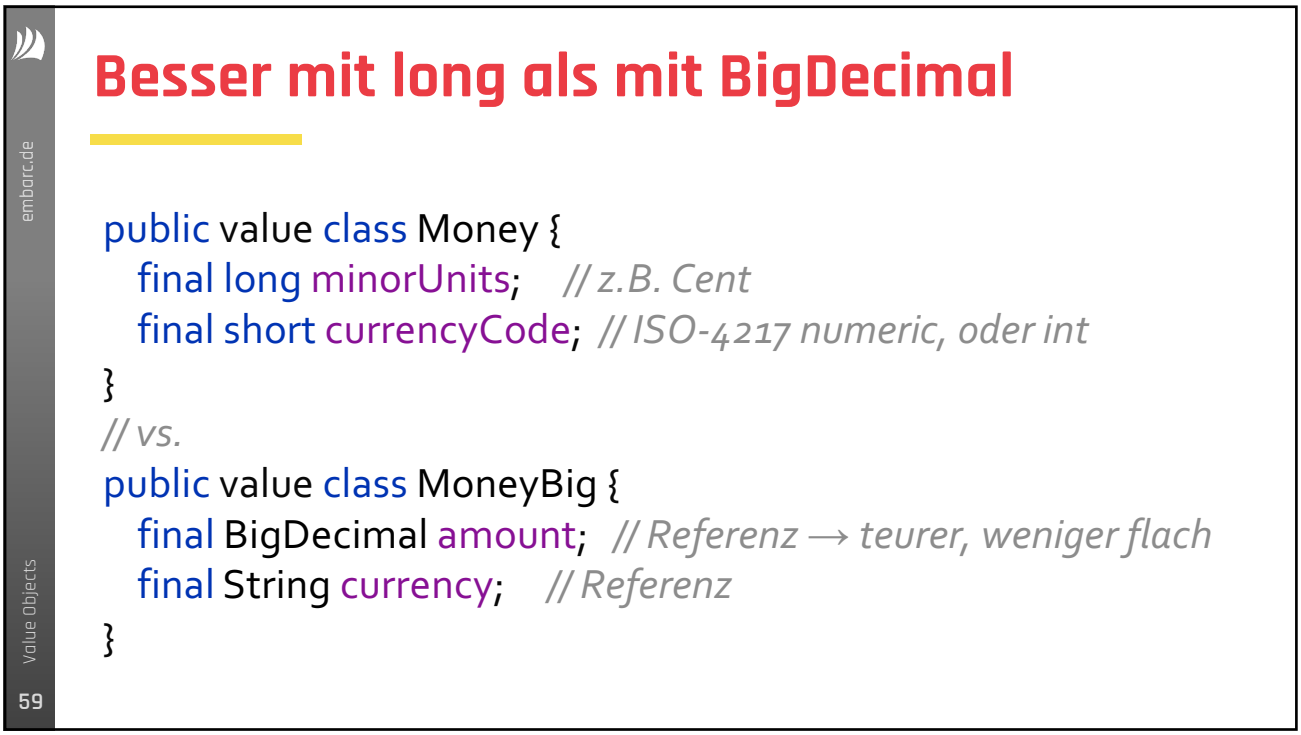


Value Types, mit mutable Data?

- Objekt-Felder sind erlaubt – aber kosten

```
public value class Email {
   final String address; // Referenz → Zeiger auf Heap-String
   // immutable, wertgleich; Nachteil: kein vollflaches Layout
}
```

58



Besser mit long als mit BigDecimal

```
public value class Money {
   final long minorUnits;    // z.B. Cent
   final short currencyCode; // ISO-4217 numeric, oder int
}
// vs.
public value class MoneyBig {
   final BigDecimal amount;  // Referenz → teurer, weniger flach
   final String currency;    // Referenz
}
```

59

Wann nicht als Value Types?

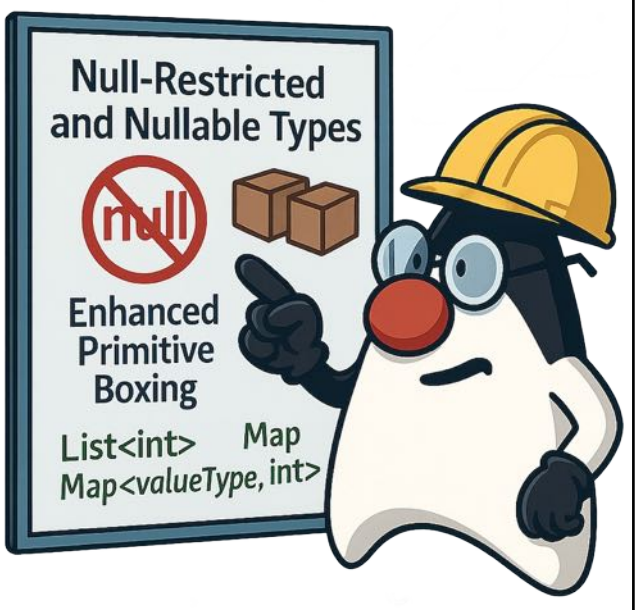
- Wenn du Identität / Mutabilität brauchst (Entity, Lebenszyklus, Caching, Lazy-Laden).
- Wenn die Struktur hauptsächlich aus großen/zahlreichen Referenzen besteht (geringer Layout-Nutzen).
- Wenn Zyklen/Grafen modelliert werden (Value Types sind für Werte, nicht für Beziehungsgraphen).

60

04.

Was kommt noch?

Mit welchen Themen müssen wir uns noch beschäftigen?



61


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?

63


und sicher mehr …
JEP 402: Enhanced Primitive Boxing
JEP yy: Null-Restricted Value Types
JEP xx: Null-Restricted Types
JEP 401: Value Classes

64

Dürfen Value Type Variablen null sein?

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

Point p = null;   // ✅ erlaubt
```


Warum dürfen jetzt noch null sein?

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

Point p = null;   // ✅ erlaubt
```

Codes like a class ...

Abwärts-kompatibilität

Einheitliches Objektmodell

Initialisierung mit Default-Werten?

Späterer Rückbau

Null-Restricted Types

- **(Nicht-) Nullbarkeit** von Typen **explizit deklarieren**

ZIELE

Typsicherheit (Compiler)

~~**NullPointer-Exceptions**~~

Sichtbarmachen im Code

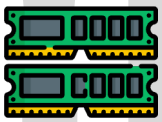
Warnungen bei Konvertierungen

Interoperabilität mit altem Code

69


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

NULL



Keine Referenz, kein Platz im Heap

Flattening + Inlining

Performance + Speicherverbrauch

70

```
Point[] -> x1,y1,x2,y2,…
```

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

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

Inlining minimiert **Indirektion** innerhalb zusammengesetzter Objekte

71

Null-Restricted Types

Point! – null ist nicht erlaubt

Point? – null ist erlaubt

Point – Verhalten wie bisher

73

74

embarc.de

Null-Restricted Value Class Types

Point ist **Value Type** und
points ist **null-restricted**

```
Point![] points
    = new Point![100];
```

Value Objects

76

76

Null-Restricted Value Class Types

Die 100 Point-Instanzen direkt/flach im Array speichern
Keine Referenzen nötig

```
Point![] points
= new Point[100];
```

Nullprüfungen vermeiden (zur Laufzeit)

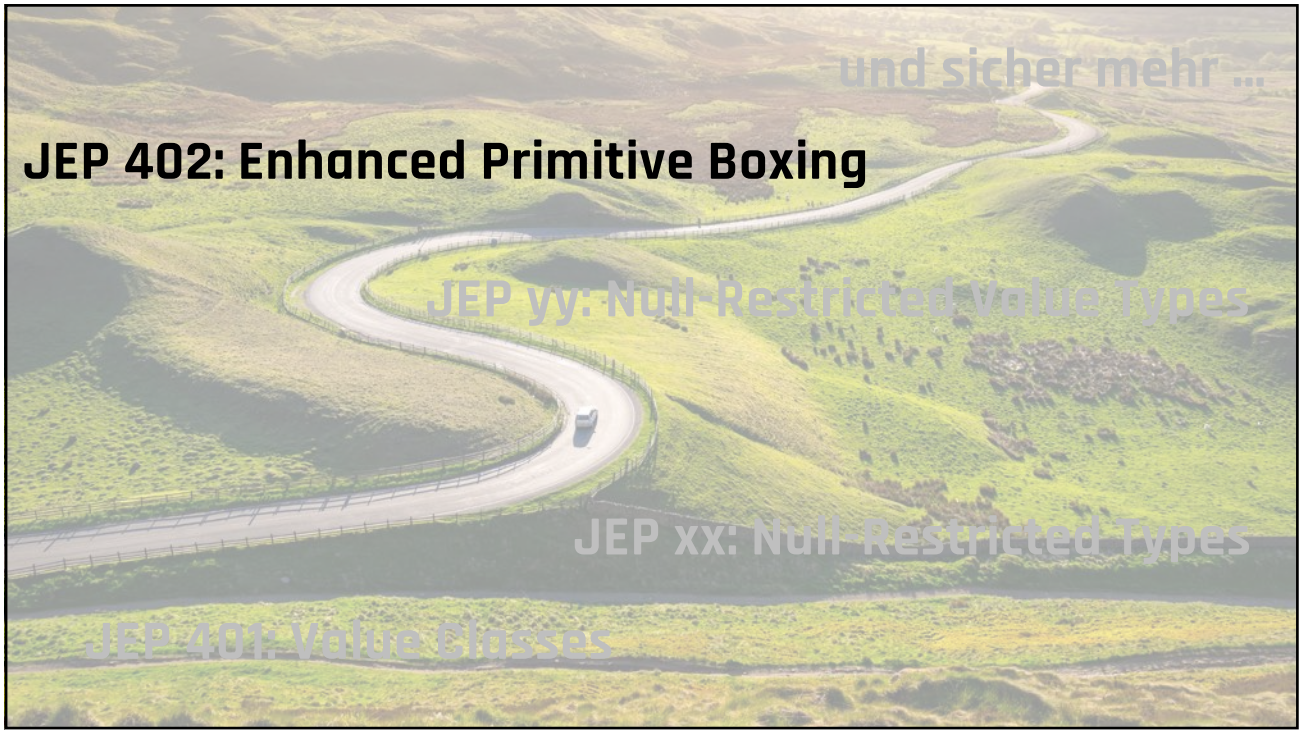
Massiv Speicher sparen
CPU-Cache und Performance verbessern

77

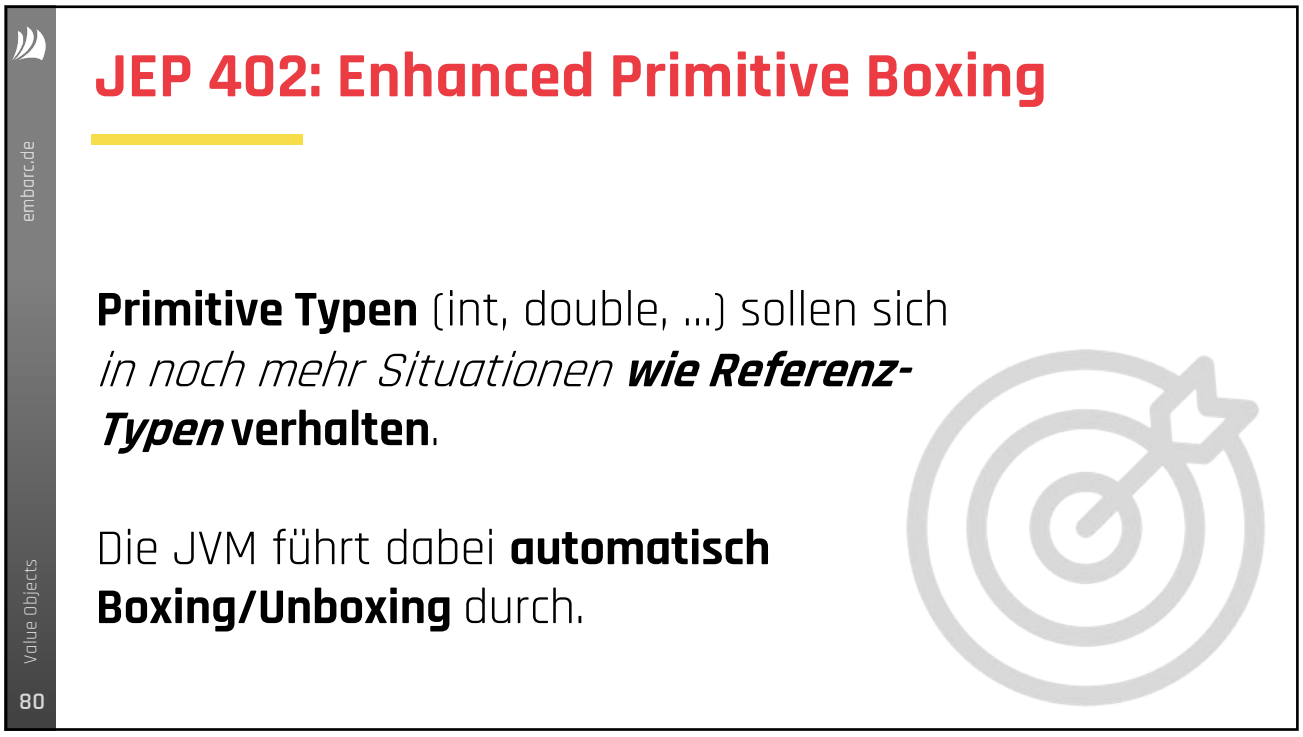
Null-safe Operator

```
var address = user?.address();
```

78


und sicher mehr ...
JEP 402: Enhanced Primitive Boxing
JEP yy: Null-Restricted Value Types
JEP xx: Null-Restricted Types
JEP 401: Value Classes

79


embarc.de
Value Objects
80
JEP 402: Enhanced Primitive Boxing
Primitive Typen (int, double, ...) sollen sich in noch mehr Situationen wie Referenz-Typen verhalten.
Die JVM führt dabei automatisch Boxing/Unboxing durch.

80

Wrapper API direkt auf Primitiven aufrufen

i wird vor dem Zugriff **zu Integer geboxt**

```
int i = 12;
int bits = i.SIZE;
```

```
double radius = 7;
double area   = radius.pow(2) * Math.PI;
```


81

Unboxed Rückgabetyp beim Überschreiben

```
interface Option {
  Object value();
}
interface IntOption extends Option {
  int value(); // erlaubt
}
```

int (Integer) statt **Object**

82

Primitive Typen als Typargumente in Generics

```
List<int> nums = List.of(1, 2, 3);

nums.add(4);

int first = nums.get(0);
```

Implizites
Boxing/Unboxing

83

Wechselseitige Konversion int[] <-> Integer![]

```
int[] raw = {1,2,3};

Object[] obj = raw;

obj[1] = 42;
```

int[] ⟶ **Integer!**[]
(boxed view)

84

Custom Autoboxing

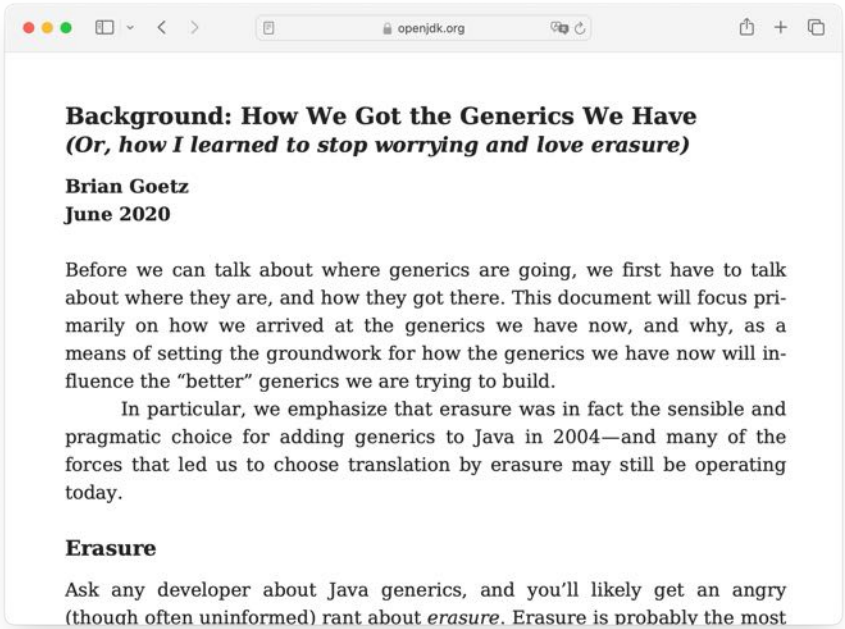
```
@PrimitiveBox
value class UInt {
    int value;
}
```

Unsigned Integer:

UInt uint = 5;

85

87



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

88

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**.

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

89

05.

Fazit + Ausblick

Zusammenfassung,
Referenzen und weitere Infos

90

Was und wie?

Final & **immutable**

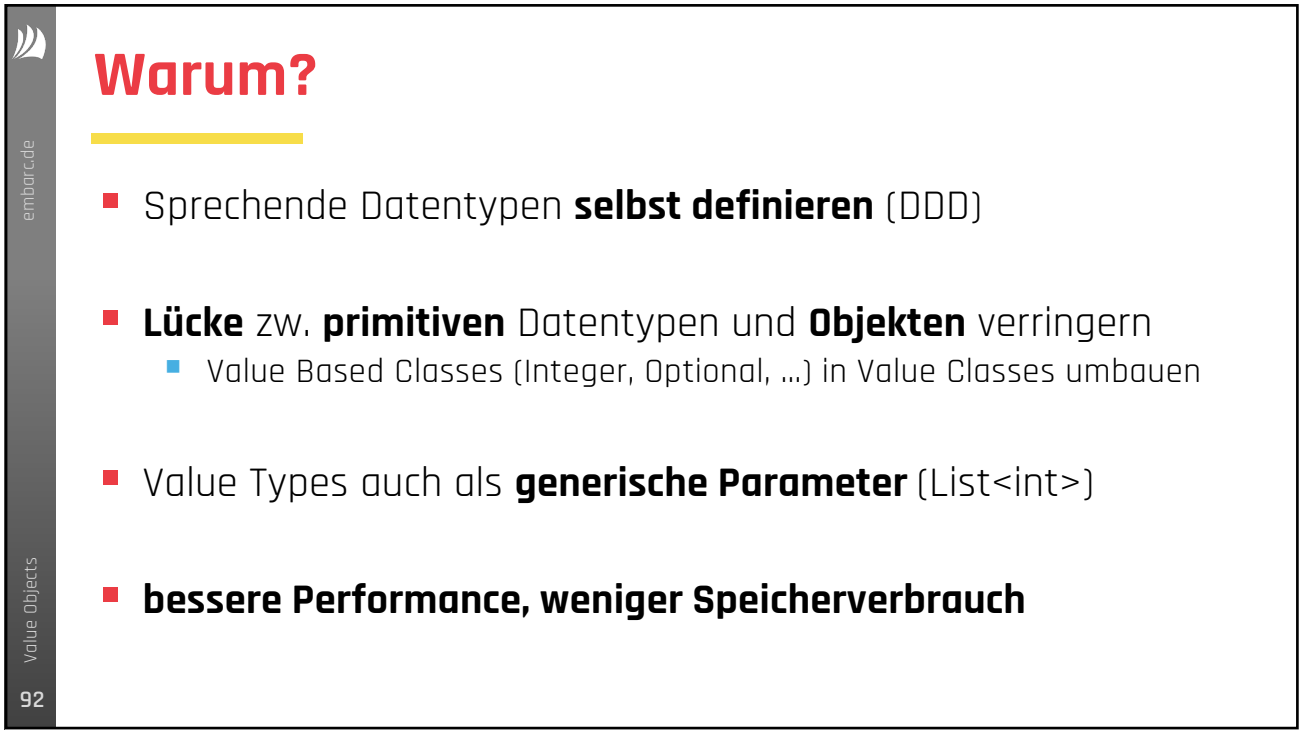
Equality by state (nicht Identität)

Keine Synchronisation

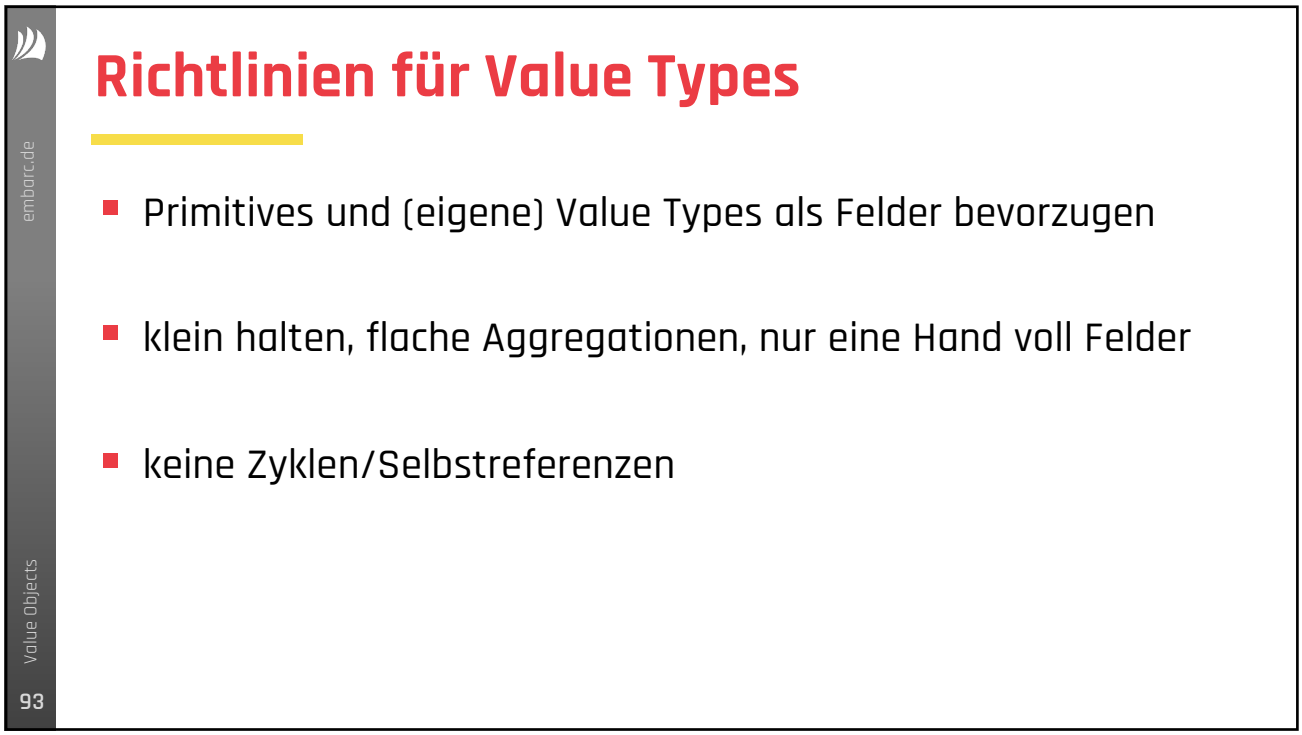
JVM-Optimierungen möglich (Flattening, Interning, Scalar Replacement)

Beispiele: **Integer**, **LocalDate**, **Optional**

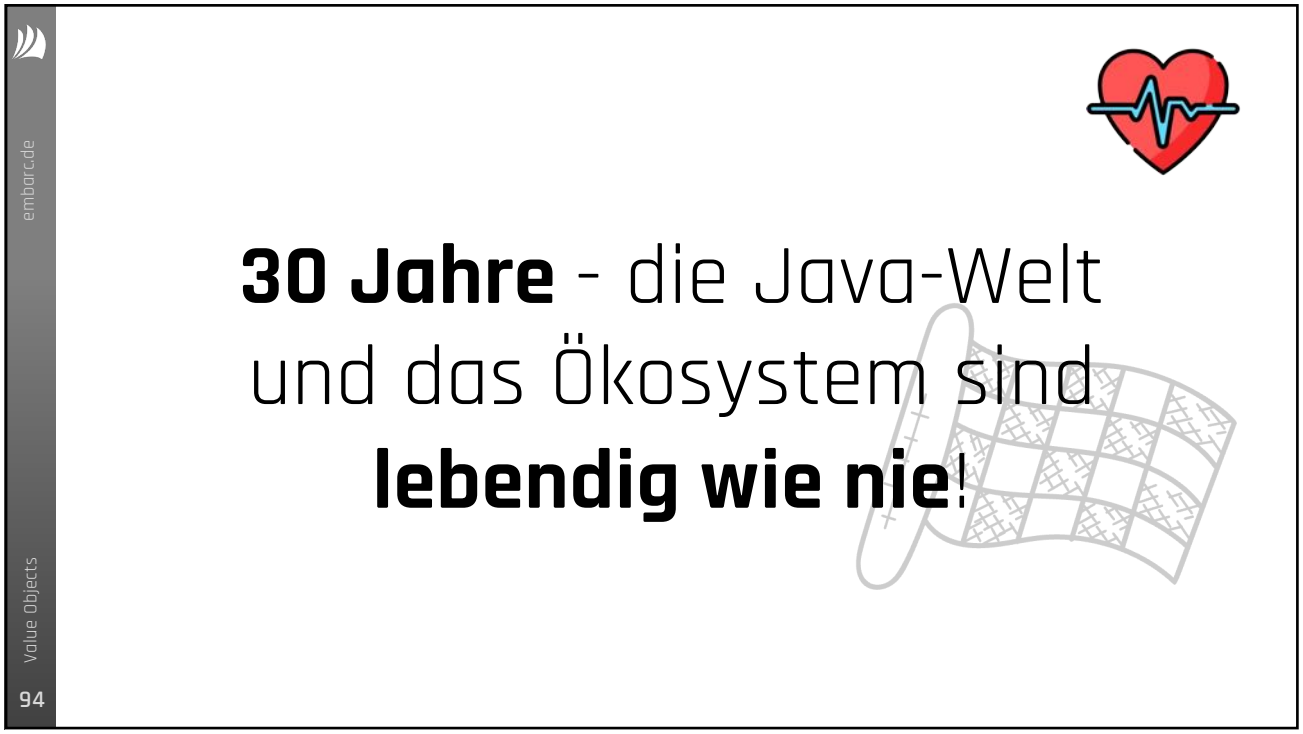
91


embarc.de
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
Value Objects
92

92


embarc.de
Richtlinien für Value Types
Primitives und (eigene) Value Types als Felder bevorzugen
klein halten, flache Aggregationen, nur eine Hand voll Felder
keine Zyklen/Selbstreferenzen
Value Objects
93

93


embarc.de

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

Value Objects
94

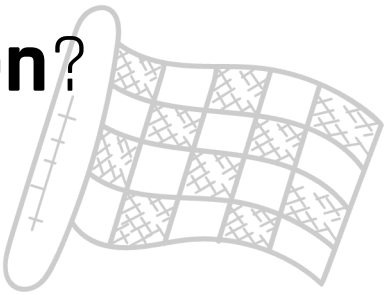
94


embarc.de
Value Objects sind
das nächste große
Thema.

Value Objects
95

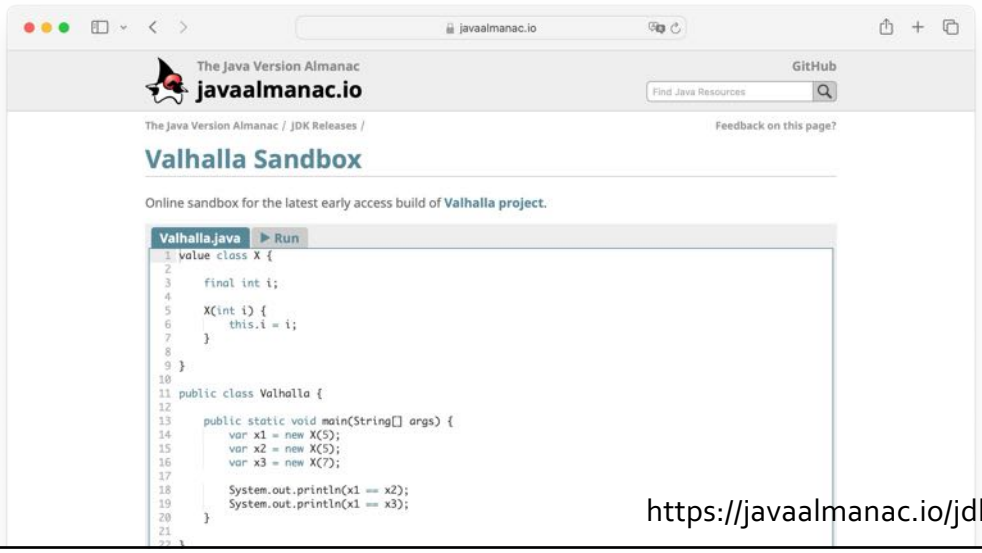
95

Valhalla, **when**?



96

Jetzt schon ausprobieren … (1)



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

97

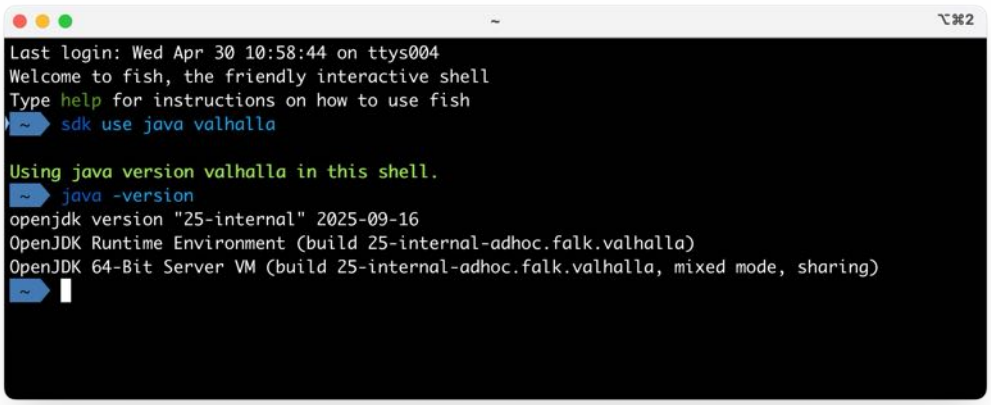
Jetzt schon ausprobieren … (2)

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

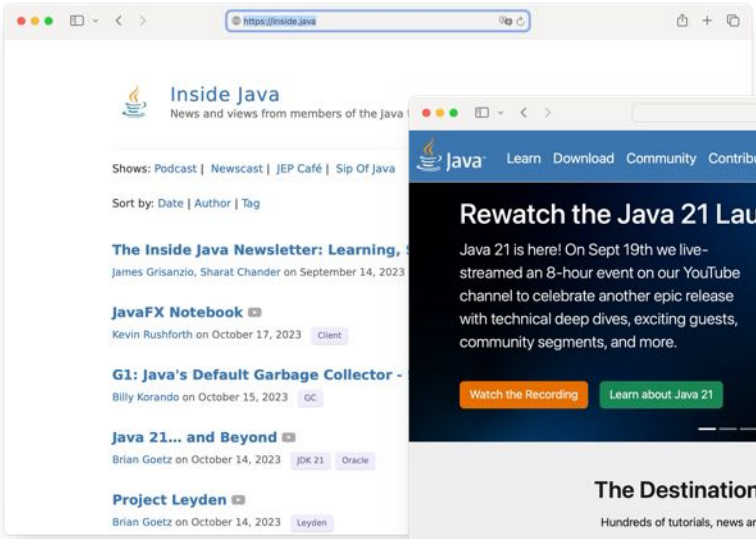
98

Mit sdkman umschalten …

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



99



https://inside.java

https://dev.java/

100

SOCREATORY
The Software Creators' Academy

EINE KOLLABORATION VON INNOQ embarc

Java Update Trainings

9 bis 11, 17, 21, 25

Sprecht mich an 💬

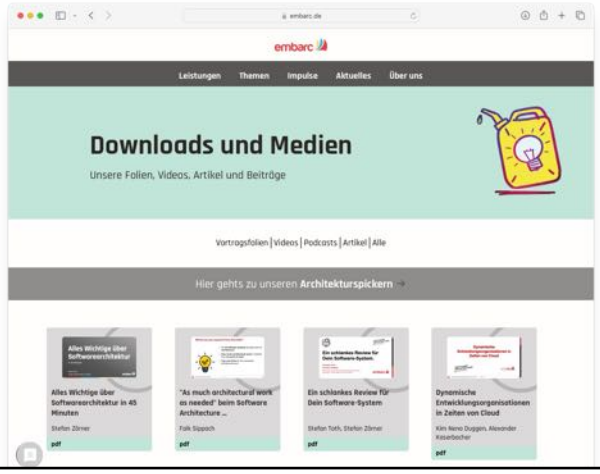
Ihr kommt zu zweit? Sehr gut! Dein Teamkollege zahlt nur 50% des Trainingspreises

Unser Preismodell - Flexibel und transparent

Interesse an iSAQB-Weiterbildungen?

101

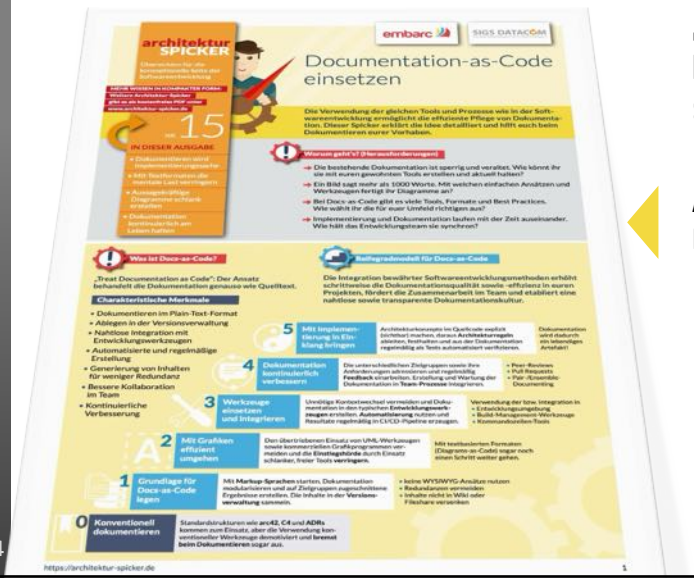
Folien und Blog-Post



➔embarc.de/java-value-objects

103

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/

104

Vielen Dank.

Ich freue mich auf Eure Fragen!

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

embarc

106

Falk Sippach

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

Schwerpunkte

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

embarc

107