

Valhalla Nullness Emotion

Rémi Forax

University Gustave Eiffel, Paris, France

IntelliJ Conf - June 2025



CALVIN & HOBBS © BIL WATTERSON

Don't believe what I'm saying !

Valhalla

- OpenJDK project
- Lead by Brian Goetz and John Rose
- Started on July, 2014 !



The primitive world ...



... the class world



Why Valhalla ?

No cost abstraction

- Code like a class, works like an int

CPUs have changed

- Flat representation vs pointer chasing

Primitives are a nuisance

- Arrays, generics, functional interface & stream

Class vs Primitive

have identity
(thus mutability)

have fields,
methods & interfaces

instances are nullable

have encapsulation

Classes

no identity

are flattened on stack
/ on heap

may have no integrity
(long on 32bits CPUs)

have a default value

Primitives

Class vs Value class vs Primitive

have identity
(thus mutability)

have fields,
methods & interfaces

instances are nullable

have encapsulation

Classes

no identity

have fields,
methods & interfaces
are flattened on stack

may be flattened on heap
/ may be nullable

may have no integrity

may have a default value
/ no encapsulation

Value Classes

no identity

are flattened on stack
/ on heap

may have no integrity
(long on 32bits CPUs)

have a default value

Primitives

Stage 1 – Value keyword

JEP 401 : Value Classes and Object

<https://openjdk.org/jeps/401>

DEMO !

<https://github.com/forax/intellij-conf-2025>

Benchmarks

Identity class

Benchmark	Mode	Cnt	Score	Error	Units
computeMandelbrot	avgt	5	474.451	± 5.279	ms/op
computeMandelbrotPrimitive	avgt	5	258.706	± 1.109	ms/op

Value class

Benchmark	Mode	Cnt	Score	Error	Units
computeMandelbrot	avgt	5	258.697	± 1.038	ms/op
computeMandelbrotPrimitive	avgt	5	259.169	± 1.162	ms/op

A value class

Identity-dependent methods have a new semantics

==

- bitwise comparison of each fields

System.identityHashCode()

- compute the value by hashing each fields

synchronized(), new WeakRef<>()

- throw new IdentityException

Inheritance & Subtyping

A value class is final

- No subclasses

A value class can implements any interfaces

A value class can inherits abstract
“value enabled” classes

- j.l.Object, j.l.Number, j.l.Record

Strict field initialization

Fields must be initialized **before** the call to `super()`

```
value class MyInteger {  
    private final int value;  
  
    public MyInteger() {  
        this.value = value;  
        super();  
    }  
}
```

Value enabled abstract class
have the same requirement !

Retrofit *value based* classes

java.lang: wrapper types are value types

Byte, Integer, Long, Double, etc

java.util

Optional, List|Set|Map.of()/copyOf()

java.time

Month, Year, LocalDateTime, etc

Java 25 : JEP draft Warning for Identity-Sensitive Libraries

<https://bugs.openjdk.org/browse/JDK-8340476>

Current state of the implementation

have fields,
methods & interfaces

(done)

have no identity

(done)

are flattened on stack

(done)

may have no integrity

(done)

may be flattened on heap

(in progress)

may be nullable

(in progress)

may have default value

(not yet !)

Value Classes

Current state of the implementation

have fields,
methods & interfaces

(done)

have no identity

(done)

are flattened on stack

(done)

may have no integrity

(done)

may be flattened on heap

(in progress)

may be nullable

(in progress)

may have default value

(not yet !)

Value Classes

Stage 2 – Nullness emotion

JEP draft : Null-Restricted and Nullable type
<https://openjdk.org/jeps/8303099>

Flattening on heap ??

Storing an instance of a value class

```
value record Complex(double im, double re) { }
```

field

```
class ComplexBox {  
    Complex complex; // can the VM avoid to use a pointer ?  
}
```

array

```
new Complex[1_024] // can the VM avoid to use pointers ?
```

Flattening on heap (if small)

Storing an instance of a small value class

```
value record MyInteger(int value) { }
```

field

```
class MyIntegerBox {  
    MyInteger complex; // can use 64 bits (32bits + a null byte)  
}
```

array

```
new MyInteger[1_024] // can use 64 bits here too
```

Current hardware limit is 64 bits

=> otherwise tearing (several loads/store)

Flattening on heap (if 64bits)

Add nullness emotion “!”

```
value record MyDouble(double value) { }
```

field

```
class MyDoubleBox {  
    MyDouble! complex; // use 64bits  
    ...  
}
```

array

```
new MyDouble![1_024] // use 64bits
```

Required to be initialized

Not-null field / array element must be never **null**

field : must be initialized **before** the call to the super constructor

```
class ComplexBox {  
    Complex! complex;           // use 64bits  
    ComplexBox(Complex complex) {  
        this.complex = complex;  
        super();  
    }  
}
```

array : we need a new syntax

```
new Complex![1_024] (/* TODO */)
```

DEMO !

<https://github.com/forax/intellij-conf-2025>

Benchmarks

Summing an array of 1_000_000

Benchmark	Mode	Cnt	Score	Error	Units
primitives	avgt	5	291,008	± 0,830	us/op
identities	avgt	5	332,557	± 1,756	us/op
nullRestrictedValues	avgt	5	291,261	± 0,752	us/op
nullRestrictedValueBoxes	avgt	5	291,209	± 1,852	us/op

Null state analysis

3 ½ states

- not-null, maybe-null, unspecified (maybe-null with no warning) + parametric

Null state analysis

- Enhance existing definite assignment analysis
- not-null if
 - The expression is known to be not null
 - after a null check (explicit or implicit)
- Otherwise maybe-null / unspecified

So it's like in Kotlin ?

Kotlin semantics uses erasure !

The only possible causes of an NPE in Kotlin are:

- An explicit call to `throw NullPointerException()` ↗.
- Usage of the not-null assertion operator `!!`.
- Data inconsistency during initialization, such as when:
 - An uninitialized `this` available in a constructor is used somewhere else (a "leaking this" ↗).
 - A superclass constructor calling an open member whose implementation in the derived class uses an uninitialized state.

From : <https://kotlinlang.org/docs/null-safety.html#nullable-types-and-non-nullable-types>

Not-null by default **is not** an option

Can not use erasure like Kotlin

- The VM has to be sure

Backward compatibility

- Previous code should still compile

```
class ComplexBox {  
    Complex complex; // can not be a ! by default  
}
```

- Nullness analysis produces warnings
And overriding rules are relaxed

How to make it great !

Two ideas ??

- Two states : users can not denote “unspecified”
 - if you opt-in (use ‘!’ in the file) to the nullable analysis, everything not not-null is nullable
- Inference of local variable nullness
 - Only method declarations and fields require nullness emotion
 - Also great for migration

Stage 3 – Retrofit Primitives

JEP 402 : Enhanced primitive boxing

<https://openjdk.org/jeps/402>

Value class vs Primitive

no identity

have fields,
methods & interfaces

are flattened on stack

may be **flattened on heap**
/ may be nullable

may have no integrity

may have a default value
/ no encapsulation

Value Classes

no identity

are flattened on stack
/ on heap

may have no integrity
(long on 32bits CPUs)

have a default value

Primitives

Value class ~~vs Primitive~~

no identity

have fields,
methods & interfaces

are flattened on stack

may be **flattened on heap**
/ may be nullable

may have no integrity

may have a default value
/ no encapsulation

Value Classes

no identity

are flattened on stack
/ on heap

may have no integrity
(long on 32bits CPUs)

have a default value

Primitives

Primitive as value class

Primitive type should behave like value class

One model to rule them all

But we can not fully have `int == Integer!`

because of the bytecode format

Fix inconsistencies

Method calls on primitive

```
int i = 3; i.toString()
```

```
3.toString() // but parser not to be fixed
```

Primitive as type argument

```
new ArrayList<int>() <=> new ArrayList<Integer!>()
```

Array reinterpretation

```
int[] <=> Integer![]
```

Overriding with primitive

```
int m() can override Integer m()
```

Value classes as primitives

Some value classes are more like primitive types (Float16, Complex)

- They have a default value
- They allow tearing of components

Value with implicit constructor

Having a default value (all fields are zeroes) bypass encapsulation

- Users should opt-in to an implicit constructor

```
value class MyInteger {  
  private final int value;  
  implicit MyInteger();    // strawman syntax  
  MyInteger(int value) {  
    this.value = value;  
    super();  
  }  
}
```

Value class with a default value

Non-null field / array can be left uninitialized

```
class MyIntegerBox {  
    private MyInteger! integer; // equivalent to MyInteger(0)  
  
    public Box() {  
        // can be empty  
    }  
  
    public void foo(MyInteger! integer) { // argument can not be null  
        var array = new MyInteger![4]; // initialized with zeroes  
        ...  
    }  
}
```

Value Class with no Integrity

Give up integrity

- Implements the interface `LooselyConsistentValue`

Example with Complex

```
value record Complex(double re, double im)
    implements LooselyConsistentValue {
    ...
}
```

Valhalla Rocket Model

Valhalla: The rocket model

- 1) Value keyword
- 2) Nullness emotion
- 3) Retrofit primitives
Operator overloading ?
- 4) Specialized generics

