ORACLE

Value Classes and Objects in Java

Dan Smith

Senior Developer

Java Platform Group

March 20, 2026

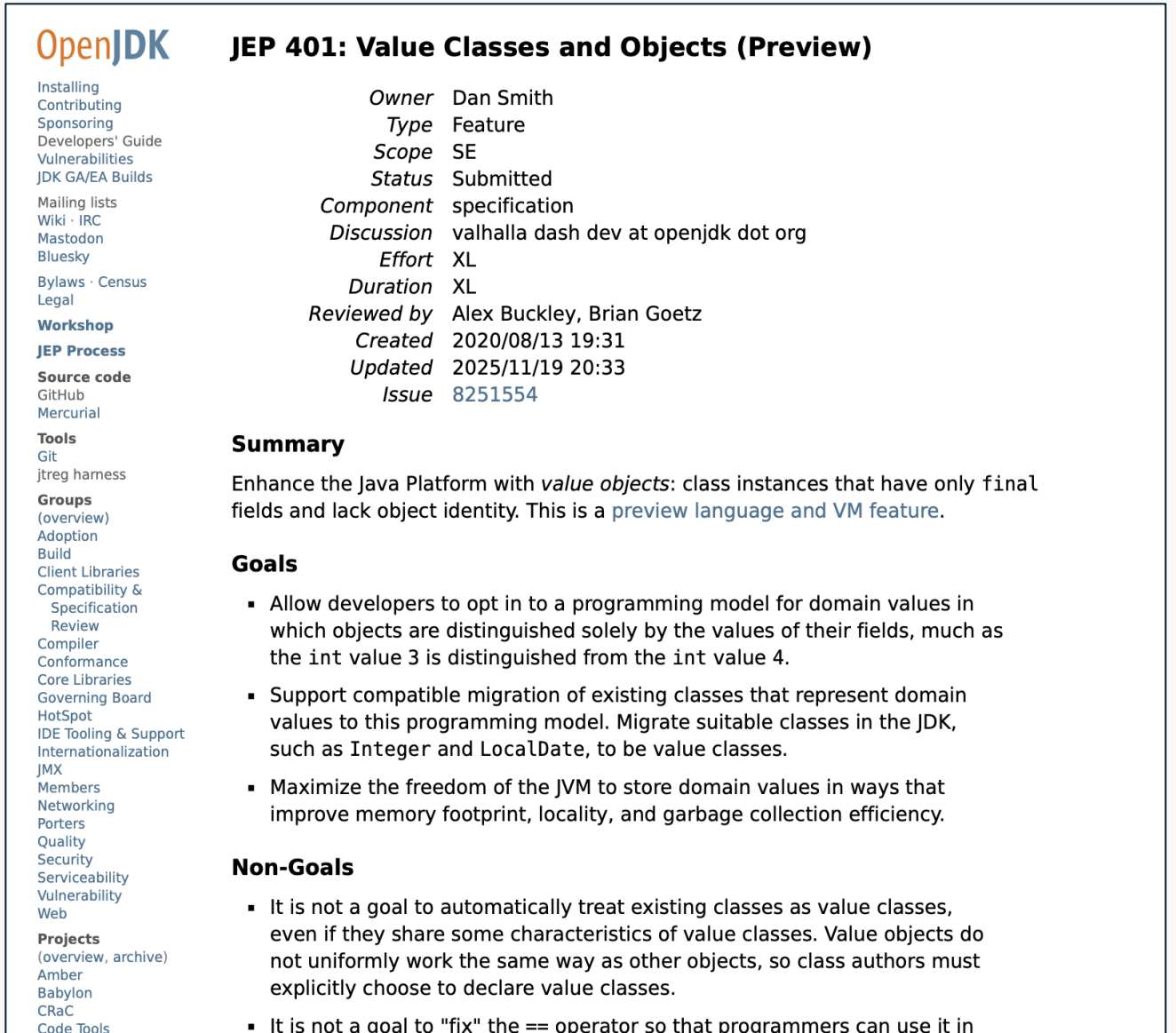
JEP 401: Value Classes and Objects (Preview)

Read the JEP: openjdk.org/jeps/401

Project background & early access:

openjdk.org/projects/valhalla/

JEP is currently Submitted, hoping to be Candidate in a couple of months and Targeted soon after



OpenJDK

Installing
Contributing
Sponsoring
Developers' Guide
Vulnerabilities
JDK GA/EA Builds

Mailing lists
Wiki · IRC
Mastodon
Bluesky

Bylaws · Census
Legal

Workshop

JEP Process

Source code
GitHub
Mercurial

Tools
Git
jtreg harness

Groups
(overview)
Adoption
Build
Client Libraries
Compatibility & Specification Review
Compiler
Conformance
Core Libraries
Governing Board
HotSpot
IDE Tooling & Support
Internationalization
JMX
Members
Networking
Porters
Quality
Security
Serviceability
Vulnerability
Web

Projects
(overview, archive)
Amber
Babylon
CRaC
Code Tools

JEP 401: Value Classes and Objects (Preview)

Owner	Dan Smith
Type	Feature
Scope	SE
Status	Submitted
Component	specification
Discussion	valhalla dash dev at openjdk dot org
Effort	XL
Duration	XL
Reviewed by	Alex Buckley, Brian Goetz
Created	2020/08/13 19:31
Updated	2025/11/19 20:33
Issue	8251554

Summary

Enhance the Java Platform with *value objects*: class instances that have only `final` fields and lack object identity. This is a preview language and VM feature.

Goals

- Allow developers to opt in to a programming model for domain values in which objects are distinguished solely by the values of their fields, much as the `int` value 3 is distinguished from the `int` value 4.
- Support compatible migration of existing classes that represent domain values to this programming model. Migrate suitable classes in the JDK, such as `Integer` and `LocalDate`, to be value classes.
- Maximize the freedom of the JVM to store domain values in ways that improve memory footprint, locality, and garbage collection efficiency.

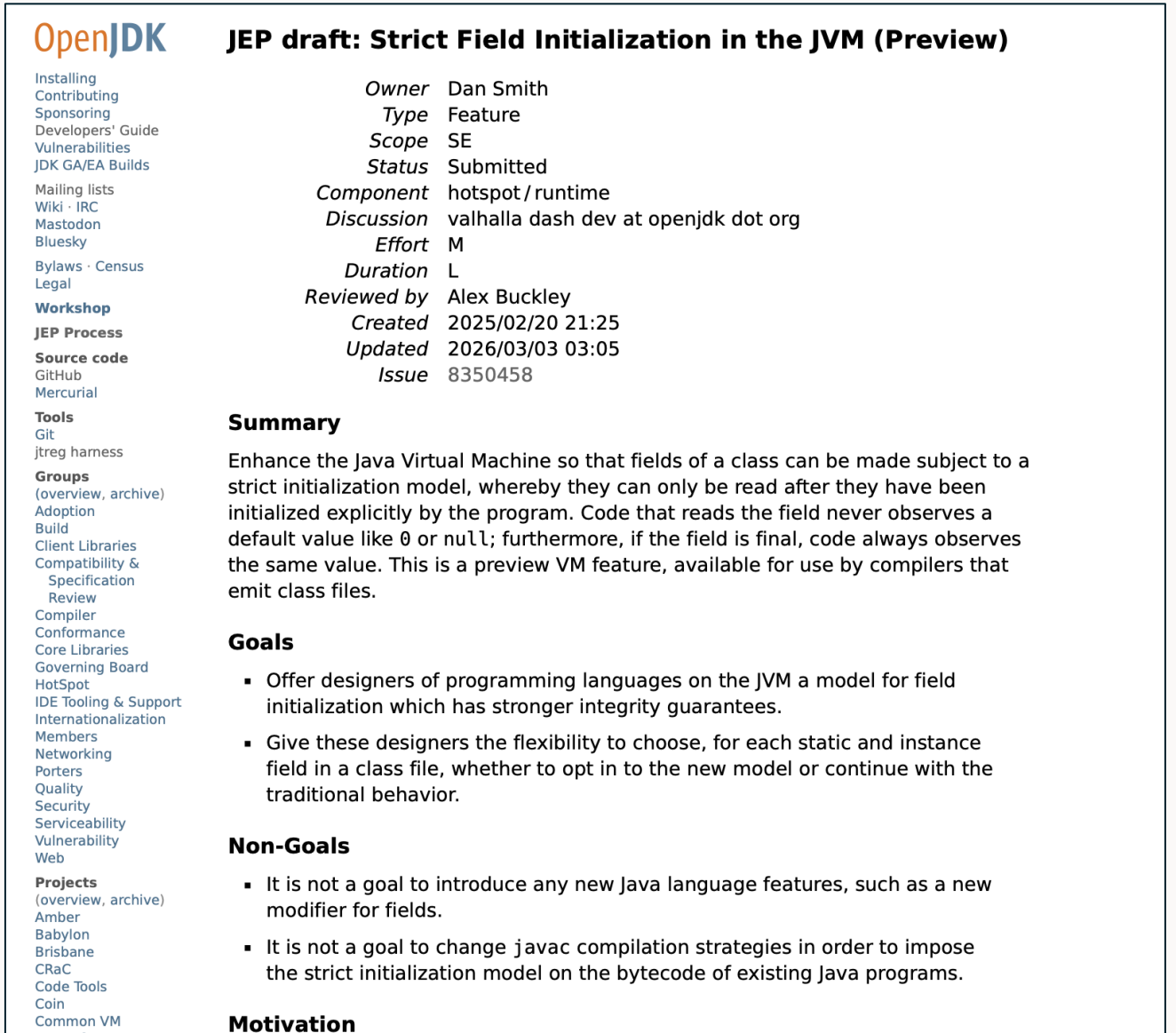
Non-Goals

- It is not a goal to automatically treat existing classes as value classes, even if they share some characteristics of value classes. Value objects do not uniformly work the same way as other objects, so class authors must explicitly choose to declare value classes.
- It is not a goal to "fix" the == operator so that programmers can use it in

JEP: Strict Field Initialization in the JVM (Preview)

Additional supplementary JEP for JVM field initialization: openjdk.org/jeps/8350458

Same timleine: currently Submitted, hoping to be Candidate in a couple of months and Targeted soon after



OpenJDK

Installing
Contributing
Sponsoring
Developers' Guide
Vulnerabilities
JDK GA/EA Builds

Mailing lists
Wiki · IRC
Mastodon
Bluesky

Bylaws · Census
Legal

Workshop

JEP Process

Source code
GitHub
Mercurial

Tools
Git
jtreg harness

Groups
(overview, archive)
Adoption
Build
Client Libraries
Compatibility & Specification Review
Compiler
Conformance
Core Libraries
Governing Board
HotSpot
IDE Tooling & Support
Internationalization
Members
Networking
Porters
Quality
Security
Serviceability
Vulnerability
Web

Projects
(overview, archive)
Amber
Babylon
Brisbane
CRaC
Code Tools
Coin
Common VM

JEP draft: Strict Field Initialization in the JVM (Preview)

Owner	Dan Smith
Type	Feature
Scope	SE
Status	Submitted
Component	hotspot / runtime
Discussion	valhalla dash dev at openjdk dot org
Effort	M
Duration	L
Reviewed by	Alex Buckley
Created	2025/02/20 21:25
Updated	2026/03/03 03:05
Issue	8350458

Summary

Enhance the Java Virtual Machine so that fields of a class can be made subject to a strict initialization model, whereby they can only be read after they have been initialized explicitly by the program. Code that reads the field never observes a default value like `0` or `null`; furthermore, if the field is final, code always observes the same value. This is a preview VM feature, available for use by compilers that emit class files.

Goals

- Offer designers of programming languages on the JVM a model for field initialization which has stronger integrity guarantees.
- Give these designers the flexibility to choose, for each static and instance field in a class file, whether to opt in to the new model or continue with the traditional behavior.

Non-Goals

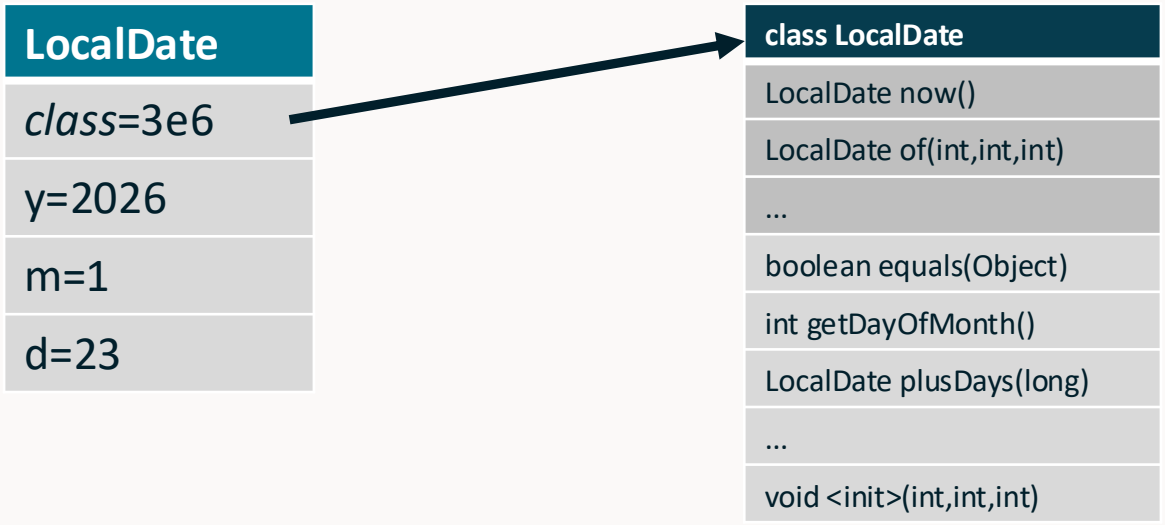
- It is not a goal to introduce any new Java language features, such as a new modifier for fields.
- It is not a goal to change `javac` compilation strategies in order to impose the strict initialization model on the bytecode of existing Java programs.

Motivation

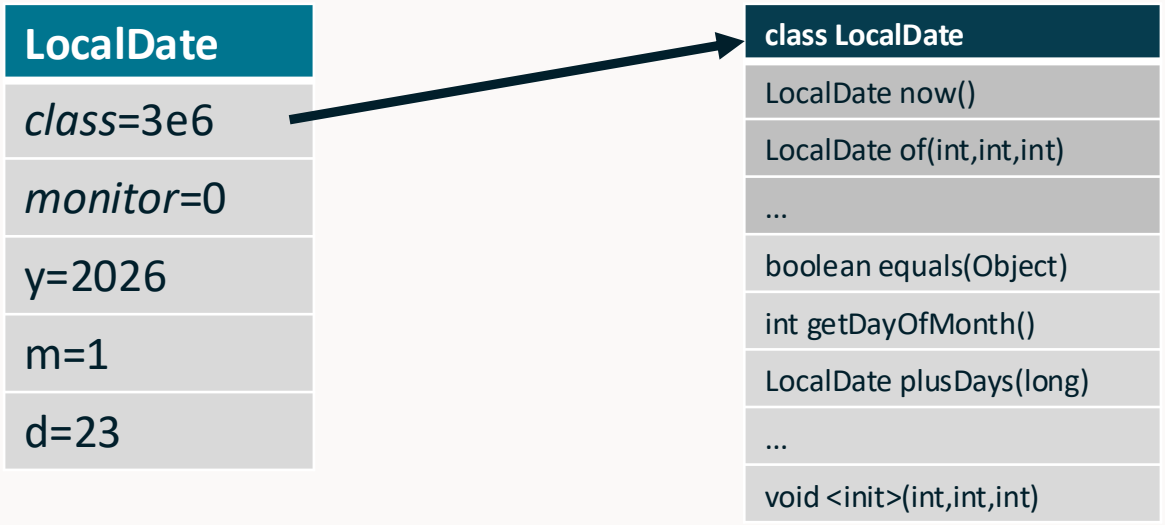
Objects in Java

LocalDate
y=2026
m=1
d=23

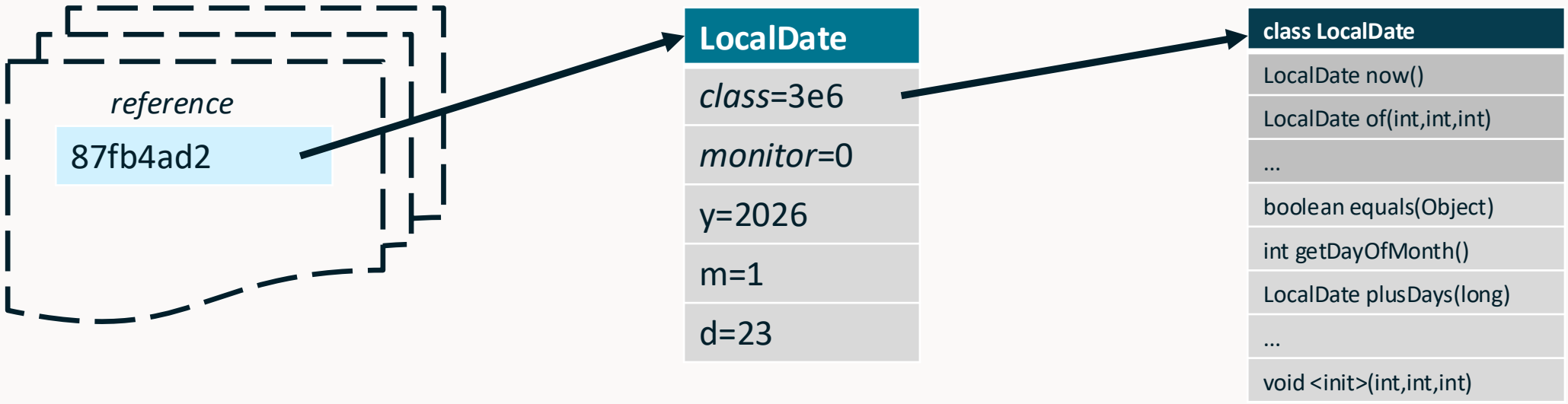
Objects in Java



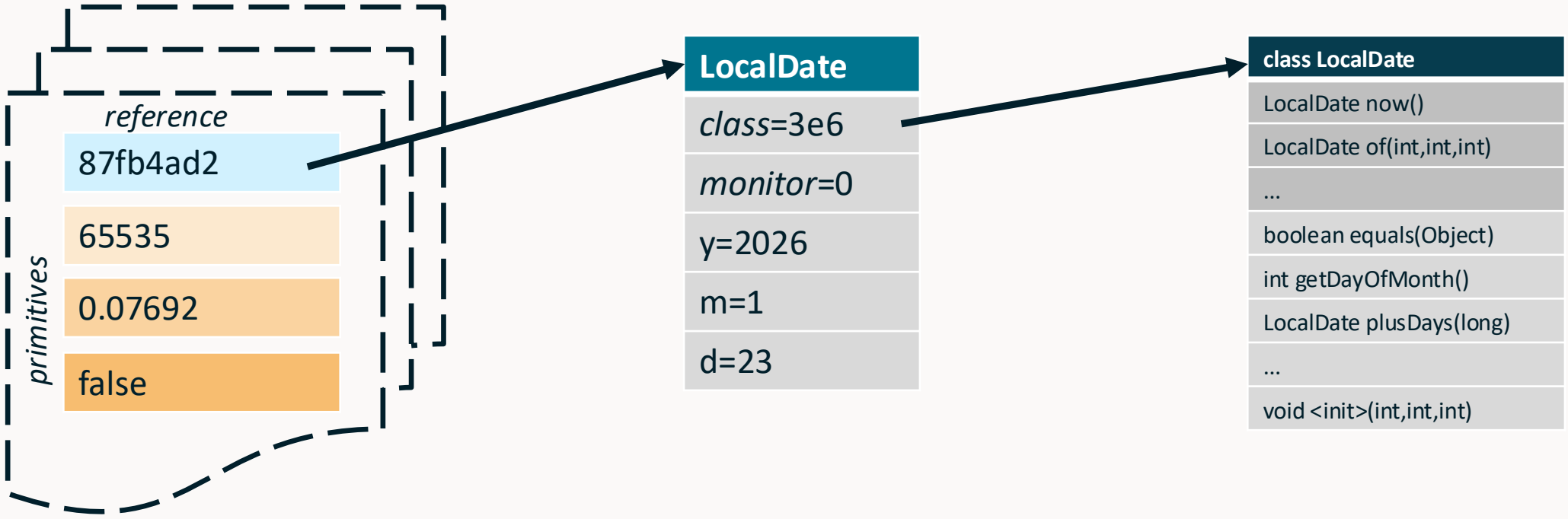
Objects in Java



Objects in Java



Objects in Java



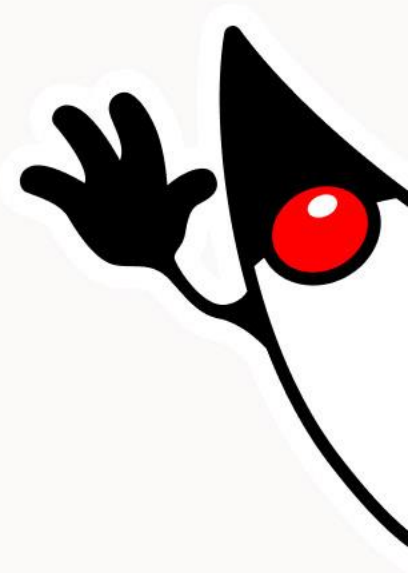
Primitives vs. objects

Advantages of programming with primitives:

- Familiar built-in types
- Dedicated operators
- Small run-time overhead
- Available directly, no memory indirections

Advantages of programming with objects:

- User-defined types, strongly enforced
- Aggregating multiple values
- Custom operations (methods)
- Managed instantiation (constructors)
- Access control
- Subtyping & polymorphism



Idea: Can programs that work with objects behave more like programs that work with primitives?

Object identity & simple domain values

Because objects have *identity*, every object:

- Has a unique memory location
- Supports state mutation
- Has a synchronization monitor
- Can be distinguished from every other object with ==

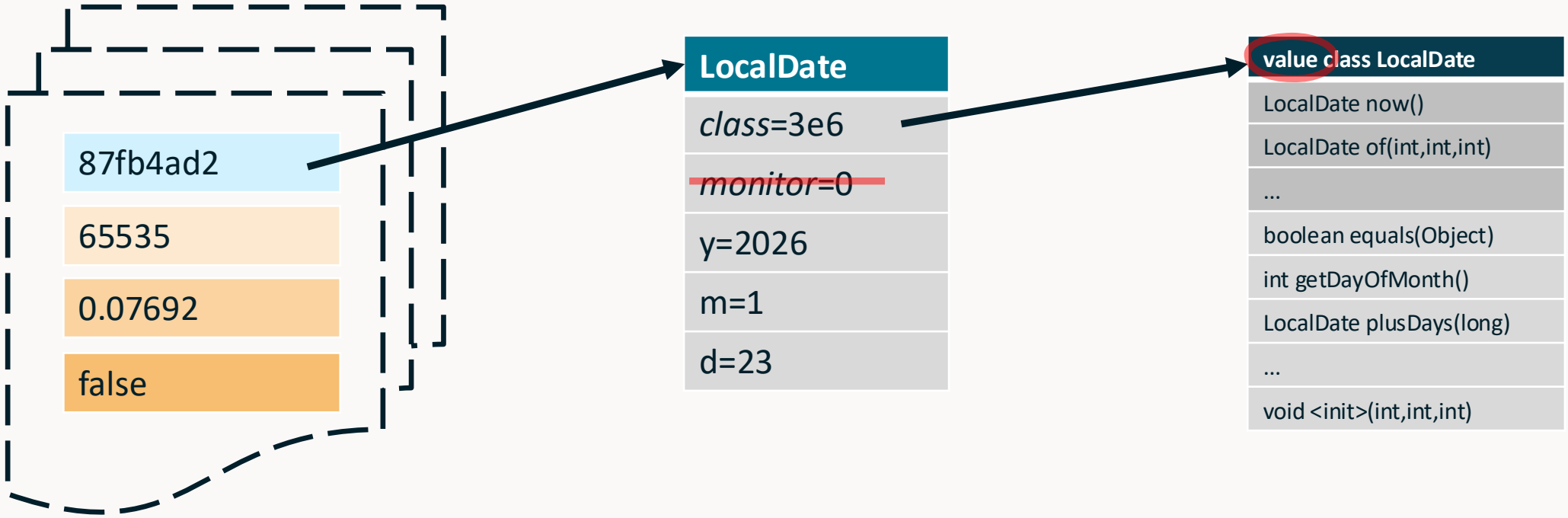
But does every object *need* these features?

Let's call an object a "simple domain value" if it:

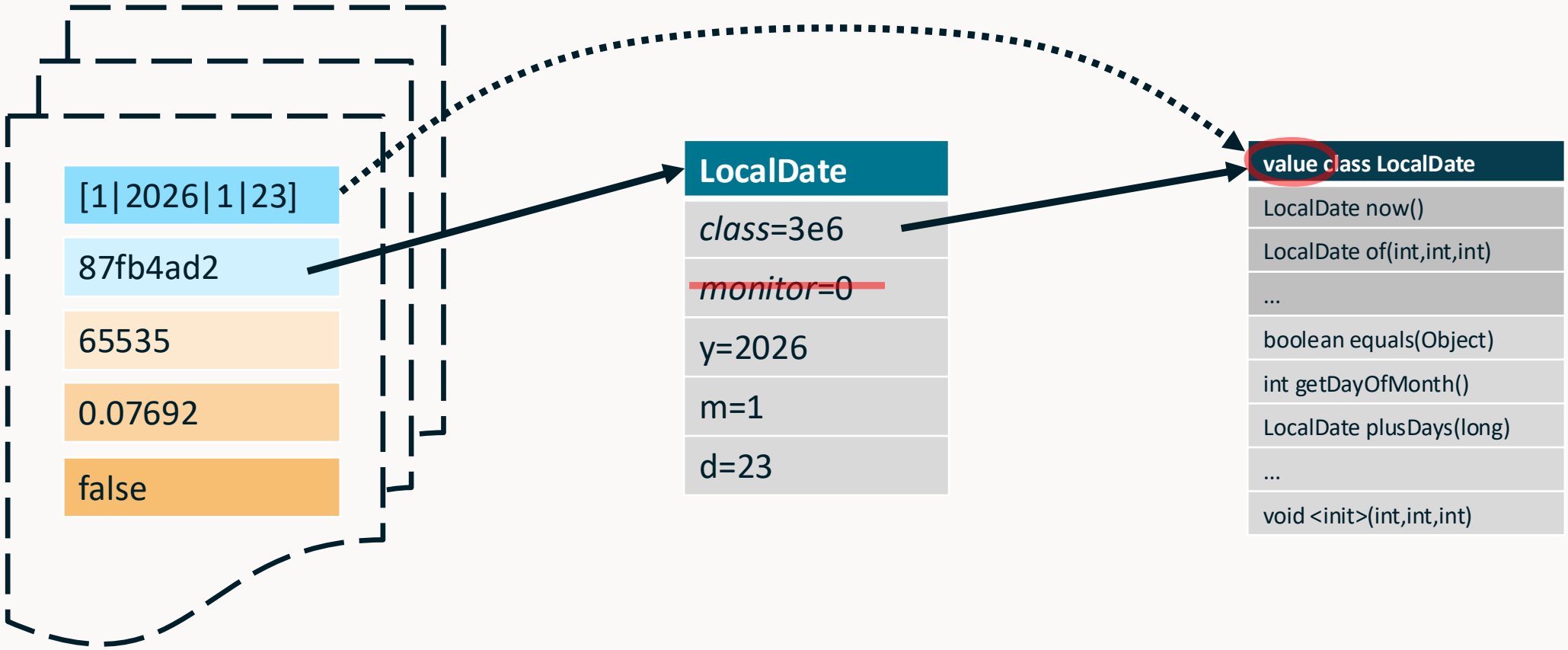
- Does not need to be unique
- Has immutable state
- Does not need to support synchronization
- Is *interchangeable* with other objects that have the same state

Classes that define simple domain values can be declared ***value classes***, meaning their instances don't need identity. Their instances are ***value objects***.

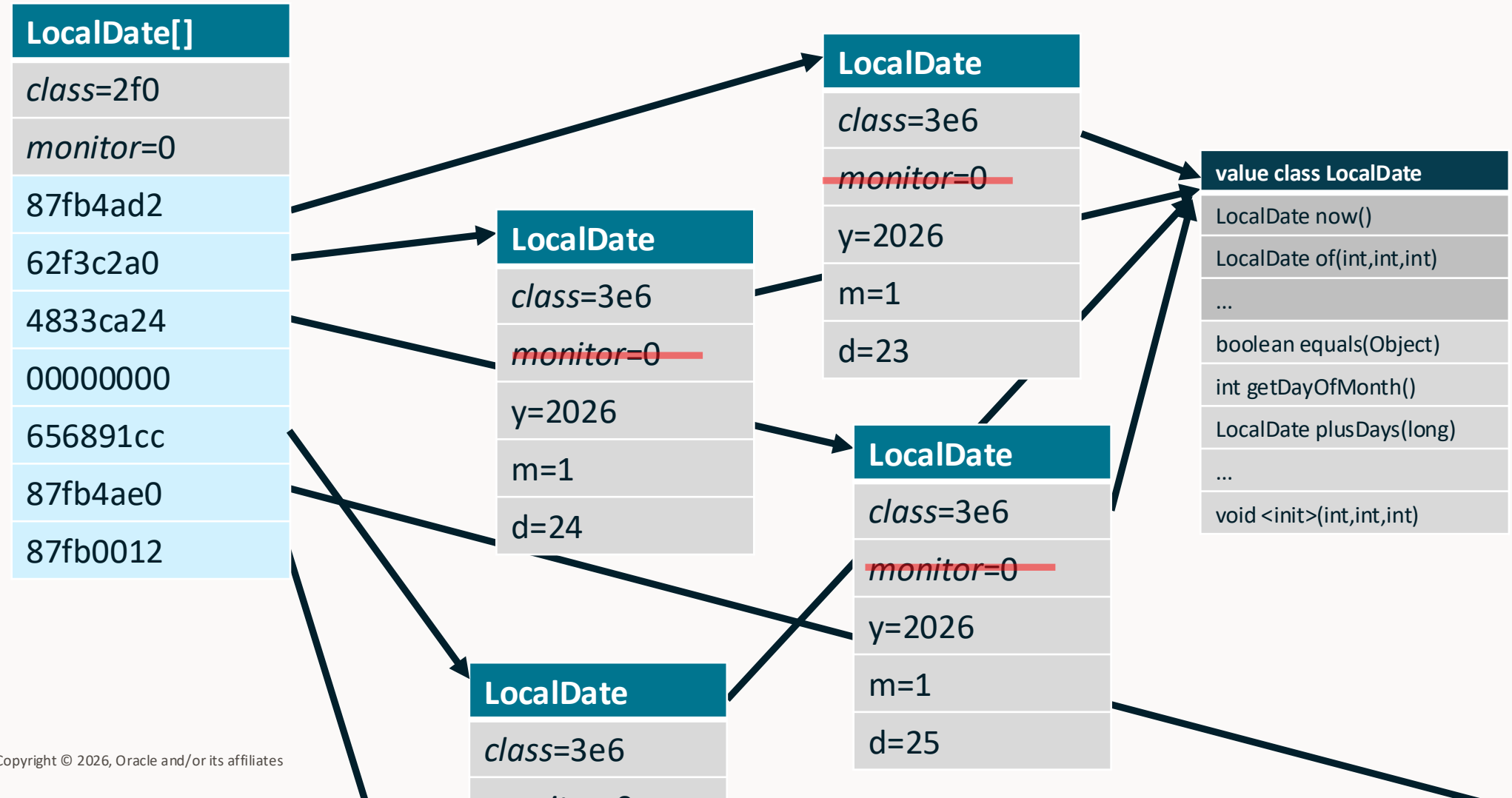
LocalDate as a value object



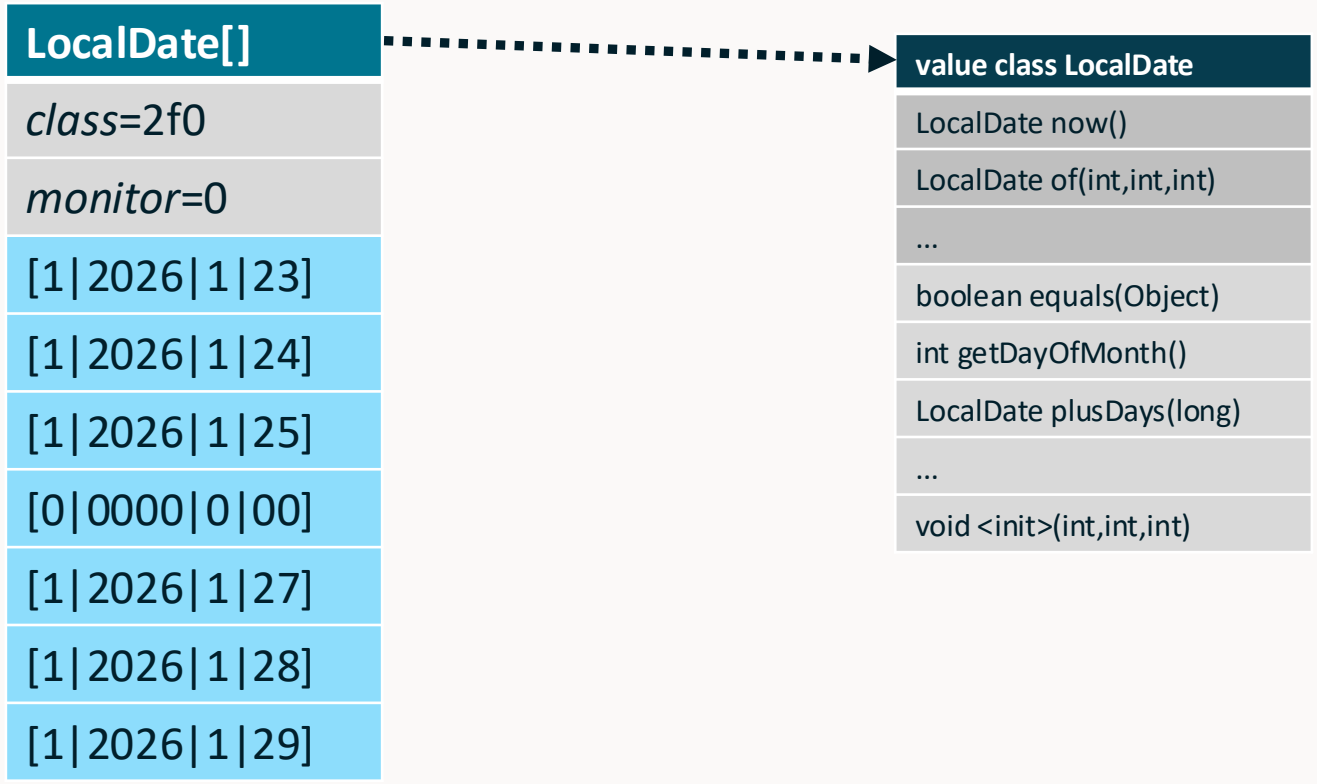
LocalDate as a value object



Arrays of pointer-encoded references



Arrays of flattened references



Hands on with value objects

1) Download from jdk.java.net/valhalla

jdk.java.net

GA Releases
JDK 26
JavaFX 26
JMC 9.1.2

Early-Access Releases
JDK 27
JavaFX 27
JavaFX Direct3D 12
Jextract
Leyden
Loom
Valhalla

Reference Implementations
Java SE 26
Java SE 25
Java SE 24
Java SE 23
Java SE 22
Java SE 21
Java SE 20
Java SE 19
Java SE 18
Java SE 17
Java SE 16
Java SE 15
Java SE 14
Java SE 13
Java SE 12

Project Valhalla Early-Access Builds

This is an early-access build from Project Valhalla implementing JEP 401: Value Classes and Objects (Preview). The goal of this build is to establish a milestone and solicit feedback as we work toward integrating this functionality into the JDK. Binaries are provided as a convenience so that users do not need to build from the source code.

Please review the feature documentation to learn more, and send your feedback via e-mail to valhalla-dev@openjdk.org.

Build 27-jep401ea3+1-1 (2026/3/11)

These early-access builds are provided under the GNU General Public License, version 2, with the Classpath Exception. The implementation is based on an incomplete version of JDK 27.

Linux/AArch64	tar.gz	246766083 bytes	(sha256)
Linux/x64	tar.gz	248889276	(sha256)
macOS/AArch64	tar.gz	241595720	(sha256)
macOS/x64	tar.gz	243791251	(sha256)
Windows/x64	zip	248360190	(sha256)

If you have difficulty downloading any of these files please contact download-help@openjdk.org.

Hands on with value objects

2) Experiment in JShell

```
dan@Matias ~
[% -> cd Downloads/jdk-26.jdk/Contents/Home/bin                                    ]

dan@Matias ~/Downloads/jdk-26.jdk/Contents/Home/bin
[% -> ./java --version                                                             ]
openjdk 26-jep401ea2 2026-03-17
OpenJDK Runtime Environment (build 26-jep401ea2+1-1)
OpenJDK 64-Bit Server VM (build 26-jep401ea2+1-1, mixed mode, sharing)

dan@Matias ~/Downloads/jdk-26.jdk/Contents/Home/bin
[% -> ./jshell --enable-preview                                                    ]
|  Welcome to JShell -- Version 26-jep401ea2
|  For an introduction type: /help intro

[jshell> Object[] objs = { 23, "ab", LocalDate.now(), List.of(1, 2, 3)}           ]
objs ==> Object[4] { 23, "ab", 2025-12-03, [1, 2, 3] }

[jshell> Arrays.stream(objs).forEach(o -> System.out.printf("%s has identity: %s%n", o, Objects.hasIdentity(o)))  ]
23 has identity: false
ab has identity: true
2025-12-03 has identity: false
[1, 2, 3] has identity: true

[jshell> LocalDate.now().plusDays(365)                                             ]
$3 ==> 2026-12-03

[jshell> $3.minusDays(365) == objs[2]                                              ]
$4 ==> true

jshell>
```

Hands on with value objects

3) Compare performance of identity objects and value objects

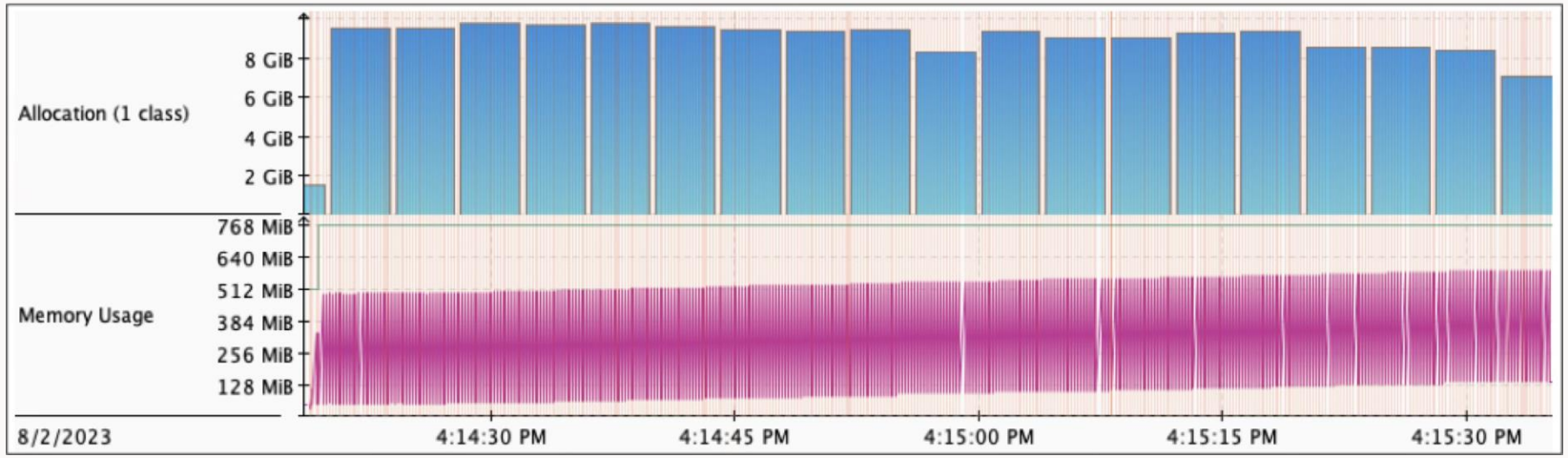
```
void main(String... args) {
   int size = 50_000_000; int attempts = 10;
   if (args.length > 0)
     size = Integer.parseInt(args[0]);
   if (args.length > 1)
     attempts = Integer.parseInt(args[1]);
   LocalDate[] arr = makeArray(size);
   for (int i = 1; i <= attempts; i++) {
      double t = time(() -> sumYears(arr));
      IO.println("Attempt " + i + ": " + t);
   }
}


long sumYears(LocalDate[] dates) {
   long result = 0;
   for (var d : dates) result += d.getYear();
   return result;
}
```

```
LocalDate[] makeArray(int size) {
   HashSet<LocalDate> set = new HashSet<>();
   for (int i = 0; i < size; i++)
      set.add(LocalDate.ofEpochDay(i));
   return set.toArray(new LocalDate[0]);
}


double time(Runnable r) {
   var start = Instant.now();
   r.run();
   var end = Instant.now();
   return Duration.between(start,end)
         .toNanos() / 1_000_000.0;
}
```

Hand on with value objects

4) Observe behavior of your own applications

Memory allocation activity doing image processing with each pixel represented by an identity object:



Memory allocation activity doing image processing with each pixel represented by a value object:

Summary of Specification Changes

Java language

- `value` class modifier
  - Field restrictions
  - Inheritance restrictions
  - Initialization restrictions
- Early construction updates
  - Readable fields during early construction
  - Record initialization restrictions
- `synchronized` restrictions
- New == behavior

Java Virtual Machine

- `ACC_STRICT_INIT` field modifier
- `StackMapTable` enhanced with `early_larval_frame` to track field initialization
- `ACC_IDENTITY` class modifier
- `LoadableDescriptors` attribute
- New `acmp` & `monitorenter` behavior

Java Class Library

- Migrated value classes in preview mode (primitive wrappers, `Optional`, `java.time`)
- Methods `Objects.hasIdentity` & `Objects.requireIdentity` (`throws IdentityException`)
- Reflection, etc., support for new modifiers
- `Object` method clarifications
- APIs that don't support value objects: serialization*, deep reflection, garbage collection

Authoring Value Classes

- One new syntax feature: the value keyword!
- Instances of value classes are value objects & do not have identity
- All fields are implicitly final & cannot be mutated
- Methods look just like the methods of any other class
- Equality: by default, the equals and hashCode methods are based on field values; they can be overridden as usual
- Subclassing: value classes can extend *abstract value classes* or Object; they can implement any interface; concrete value classes cannot be extended
- Construction: value class constructors must assign to fields before delegating to the superclass (super()); initialization cannot depend on this

```
value class EURCurrency {
    private int cs;  // implicitly final
    private EURCurrency(int cs) { this.cs = cs; }

    public EURCurrency(int euros, int cents) {
        this(euros * 100 +
             (euros < 0 ? -cents : cents));
    }

    public int euros() { return cs/100; }
    public int cents() { return Math.abs(cs%100); }
    public String toString() {
        return "€%d,%d".formatted(euros(), cents());
    }
}
```

Value Classes & Equality

- Usually, if you want to tell if two objects are “equal”, you should use one of:
  - x.equals(y)
  - Objects.equals(x, y)
- The class author defines equals; if they don’t override the default equals implementation, it compares value objects’ field values with ==; this is *often*, but not *always* what they want
- In this example, two LazySubstring instances may represent the same sequence of characters, but derive them from different field values
  - new LazySubstring(“ringing”, 1, 4)
  - new LazySubstring(“ringing”, 4, 7)
- If different value objects have different field values, they are not == even if they are equals

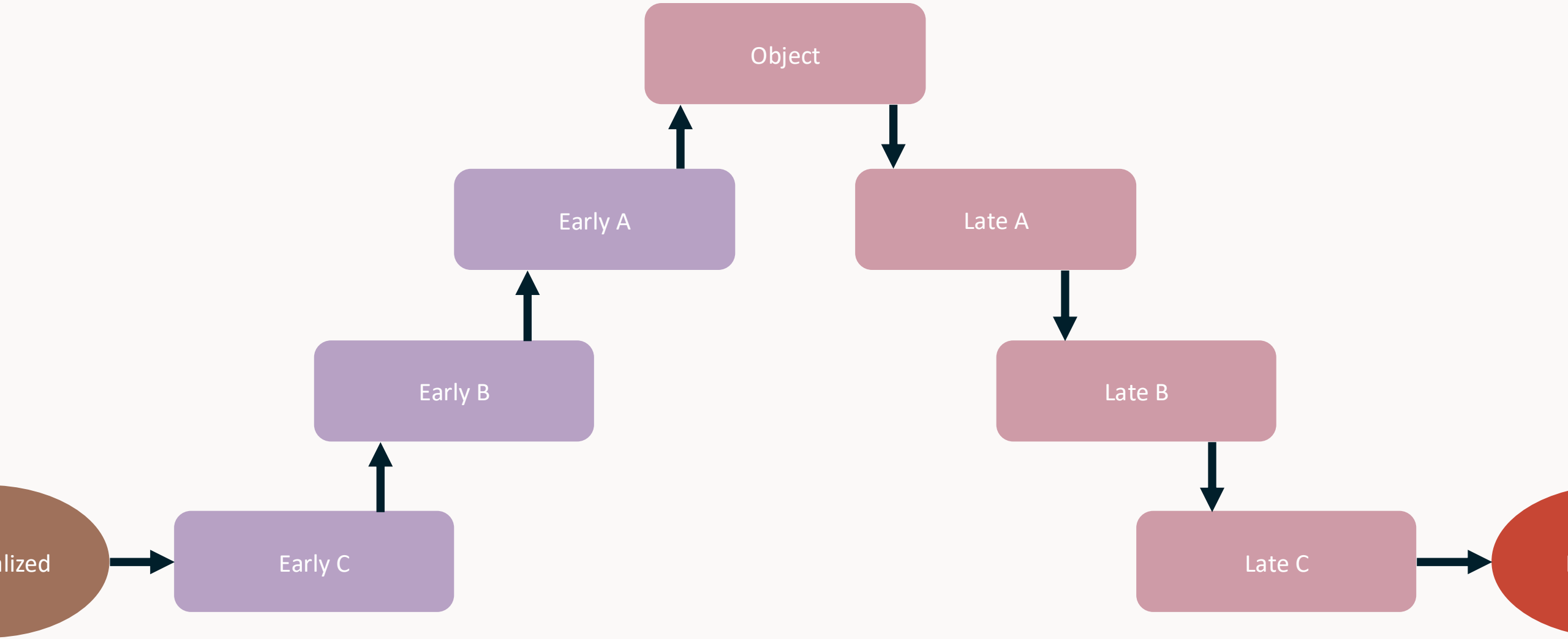
```
value class LazySubstring {
    private String str;
    private int start, end;
    public LazySubstring(String s, int i, int j) {
        str = s; start = i; end = j;
    }

    public String toString() {
        return str.substring(start, end);
    }
    public boolean equals(Object o) {
        return o instanceof LazySubstring &&
            toString().equals(o.toString());
    }
    public int hashCode() {
        return Objects.hash(LazySubstring.class, toString());
    }
}
```

Value Classes & Subclassing

- If a class is an abstract value class, it supports both identity and value subclasses
- A regular abstract class only supports identity subclasses
- If your abstract class doesn't need identity, make it a value class!
- Abstract value classes can have fields and constructor logic

```
abstract value class Number implements Serializable {
    public abstract int intValue();
    public abstract long longValue();
    public byte byteValue() {
        return (byte) intValue();
    }
    …
}

value class Integer extends Number
        implements Comparable<Integer> {
    private int value;
    public int intValue() { return value; }
    public long longValue() { return value; }
    int compareTo(Integer that) {
        return that – this;
    }
    …
}
```

Construction timing (see also JEP 513: Flexible Constructor Bodies)



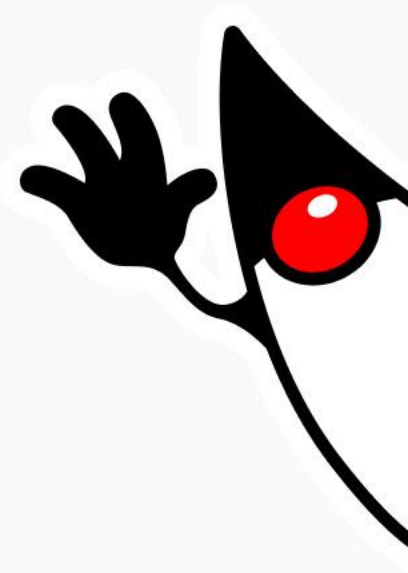
Value Classes & Construction

- Fields of value classes *must* be initialized during early construction
- For convenience, the default timing of constructors changes in value classes
  - Identity class: super(), initializers, constructor
  - Value class: initializers, constructor, super()
- In early construction, fields can be read and written, but methods can’t be invoked, and this can’t otherwise be referenced

```
value class EURCurrency {
   private int cs;  // implicitly final

   private EURCurrency(int cs) {
      this.cs = cs;
      IO.println(this); // error!
      // implicit: super();
   }

   public EURCurrency(int euros, int cents) {
      this(euros * 100 +
           (euros < 0 ? -cents : cents));
   }

   ...
}
```

What's Next?

- Preparing for preview release (stabilization, TOIs, code reviews, ...)
- Future language work: null checking & atomicity control, array enhancements, unifying primitives and classes, generic reification
- Future VM work: performance tuning, developer tooling, new layouts & techniques, generic specialization
- Future library work: marshalling, identify migration candidates, internal performance improvements, new APIs



Project: openjdk.org/projects/valhalla

JLS & JVMS: cr.openjdk.org/~dlsmith/jep401/latest

EA Download & Javadoc: jdk.java.net/valhalla


ORACLE