
# Final heißt final:

## Wie Java 26 sich gegen Deep Reflection verteidigt

Sven Woltmann

Java Consultant, Speaker, Trainer

# Agenda

1. Deep Reflection – was ist das?
2. Live Coding: ein realistisches Angriffsszenario
3. Javas Plan: „Integrity by default"
4. JEP 500: „Prepare to Make Final Mean Final"
5. Was bedeutet das für mich?



# Wie hat das funktioniert?

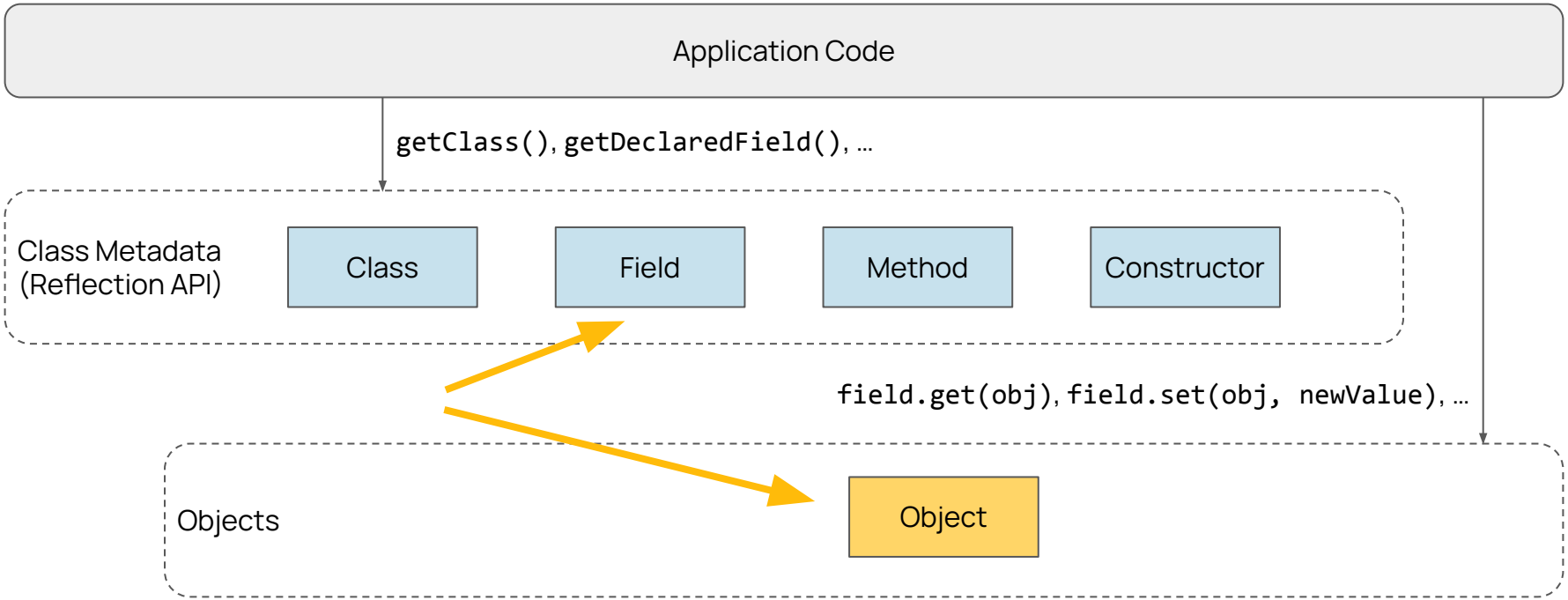
```
public class ApparentlyHarmlessThirdPartyLibrary {

  ...

  static {
    try {
      Field VALUE = Integer.class.getDeclaredField("value");
      VALUE.setAccessible(true);
      VALUE.set(2, 42);
    } catch (ReflectiveOperationException e) {
      throw new Error(e);
    }
  }
}
```

```
Integer a = 2;
IO.println("a = " + a);
```

```
a = 42
```



# Was ist Reflection?



# Was macht den „Deep“-Teil aus?

```
public class Box {

  private int privateField;

  private final int privateFinalField = 1;

  public int publicField;

  public final int publicFinalField = 2;

}
```



# Was macht den „Deep“-Teil aus?

```
Field PRIVATE_FIELD = Box.class.getDeclaredField("privateField");
Field PRIVATE_FINAL_FIELD = Box.class.getDeclaredField("privateFinalField");
Field PUBLIC_FIELD = Box.class.getDeclaredField("publicField");
Field PUBLIC_FINAL_FIELD = Box.class.getDeclaredField("publicFinalField");

PRIVATE_FIELD.setAccessible(true);
PRIVATE_FINAL_FIELD.setAccessible(true);
PUBLIC_FIELD.setAccessible(true);
PUBLIC_FINAL_FIELD.setAccessible(true);

Box box  = new Box();

PRIVATE_FIELD.set(box, 42);
IO.println(PRIVATE_FIELD.get(box));
PRIVATE_FINAL_FIELD.set(box, 42);
IO.println(PRIVATE_FINAL_FIELD.get(box));

PUBLIC_FIELD.set(box, 42);
IO.println(PUBLIC_FIELD.get(box));
PUBLIC_FINAL_FIELD.set(box, 42);
IO.println(PUBLIC_FINAL_FIELD.get(box));
```

# Was kann man damit alles machen?

private-Felder lesen und schreiben:

```
field.setAccessible(true);
field.get(obj);
field.set(obj, value);
```

final-Felder überschreiben

```
finalField.setAccessible(true);
finalField.set(obj, 200);
```

Private Methoden aufrufen

```
method.setAccessible(true);
method.invoke(obj, args);
```

Objekte über private Konstruktoren erzeugen

```
constructor.setAccessible(true);
constructor.newInstance(args);
```



# Was passiert, wenn jemand das gegen dich einsetzt?



# Ein realistisches Angriffsszenario



# Kein Exploit.
# Nur Deep Reflection.



## JDK 5, 2004

`final` = Immutability-Garantie

Basis des Java Memory Model

## Problem

Java Serialization muss `final`-Felder beim Deserialisieren befüllen

→ Reflection-API wird geöffnet

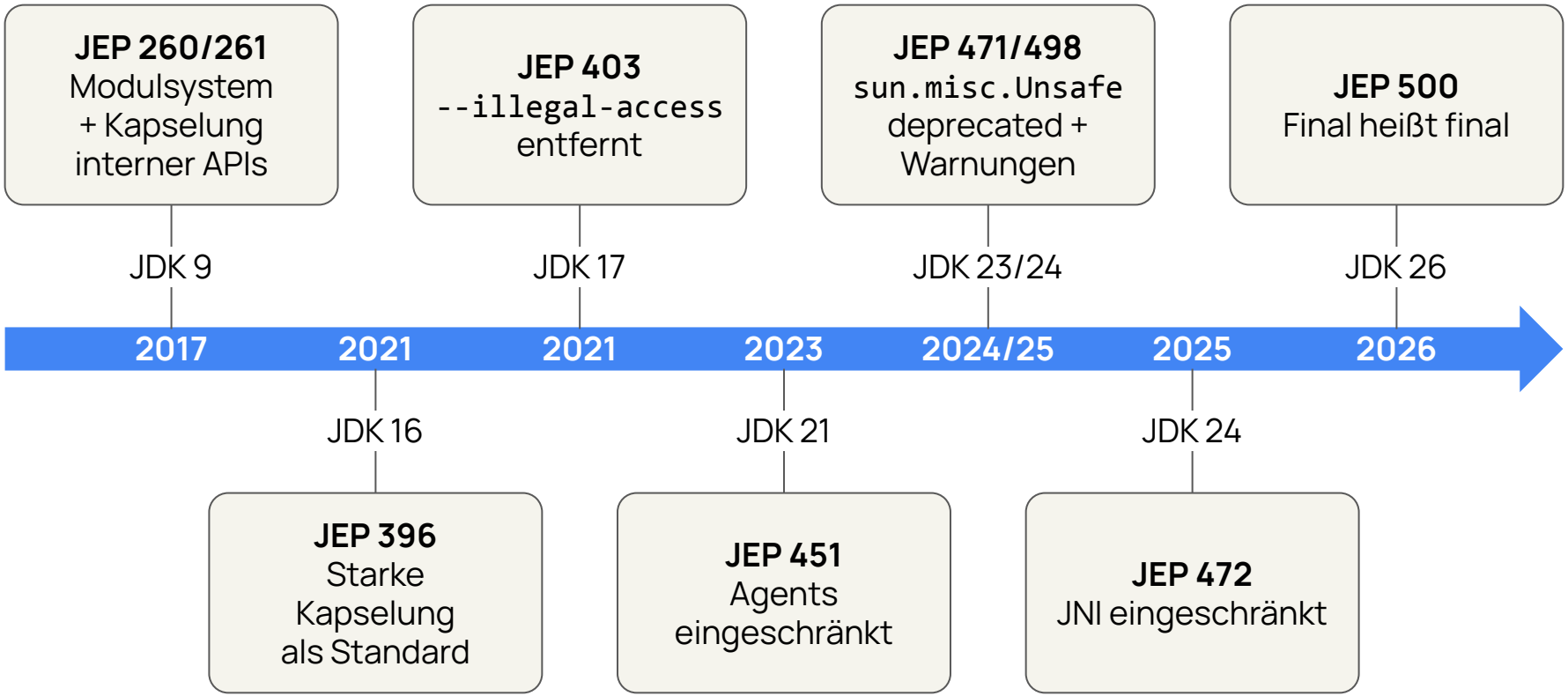
*“In retrospect, offering such unconstrained functionality was a poor choice.”*
– JEP 500

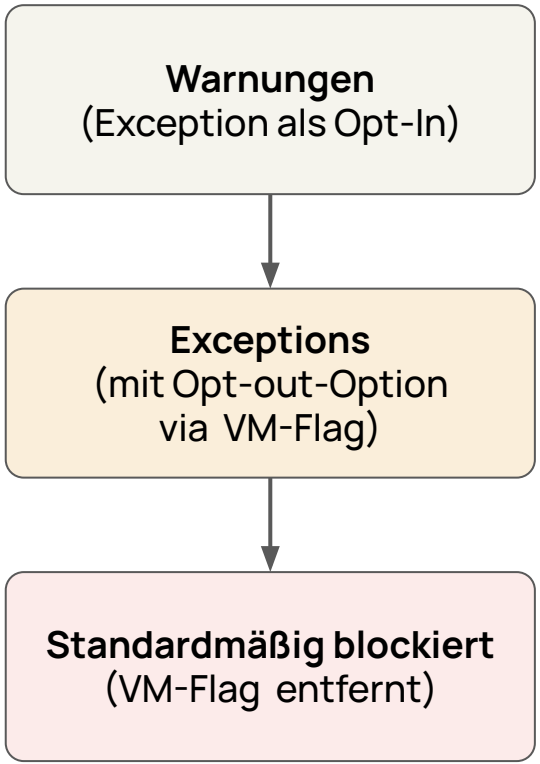


# Integrity by default

Javas langfristiges Programm für Laufzeitintegrität

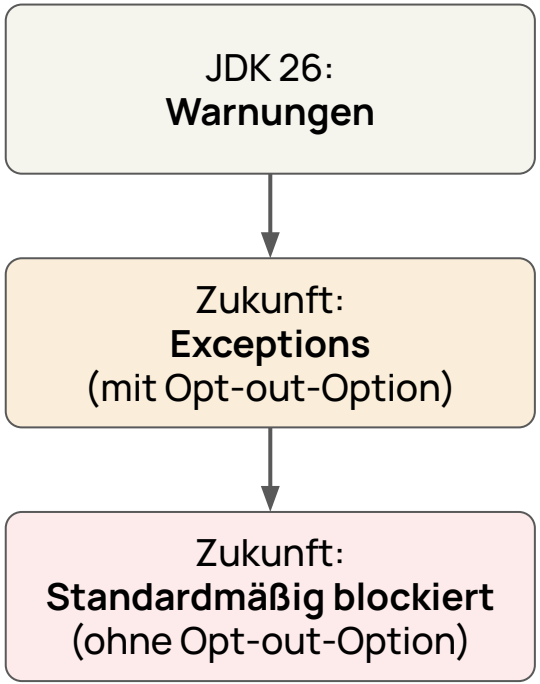






# JEP 500: Final heißt final

Deep Reflection auf final-Felder:



# JEP 500: Die neuen VM-Optionen

`--illegal-final-field-mutation`

| | |
|---|---|
| allow | Mutation erlaubt, keine Warnung |
| warn | Mutation erlaubt, einmalige Warnung |
| debug | Mutation erlaubt, Warnung + Stack Trace bei jedem Aufruf |
| deny | IllegalAccessException, Mutation verweigert |

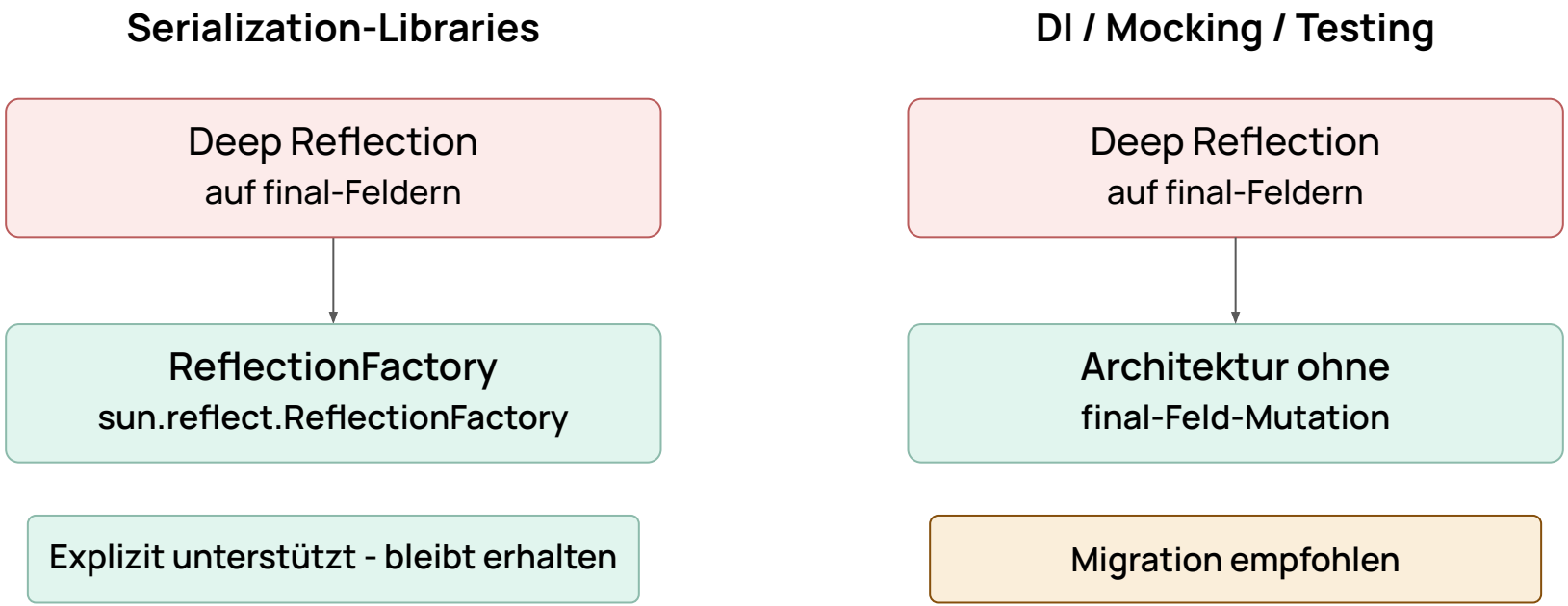
**Standard in Java 26**

**Standard in Zukunft**

`--enable-final-field-mutation`

| | |
|---|---|
| ALL-UNNAMED | Mutation erlaubt für allen Code auf dem Classpath |
| com.example.mymodule | Mutation erlaubt für ein bestimmtes Modul |

# JEP 500: Was bedeutet das für Libraries?



# Zurück zum Angriffsszenario

## (diesmal unter Java 26)



# Bin ich betroffen?

**Warnungen im Log beobachten**

`--illegal-final-field-mutation=warn` (Standard)

**Oder sofort auf deny schalten:**

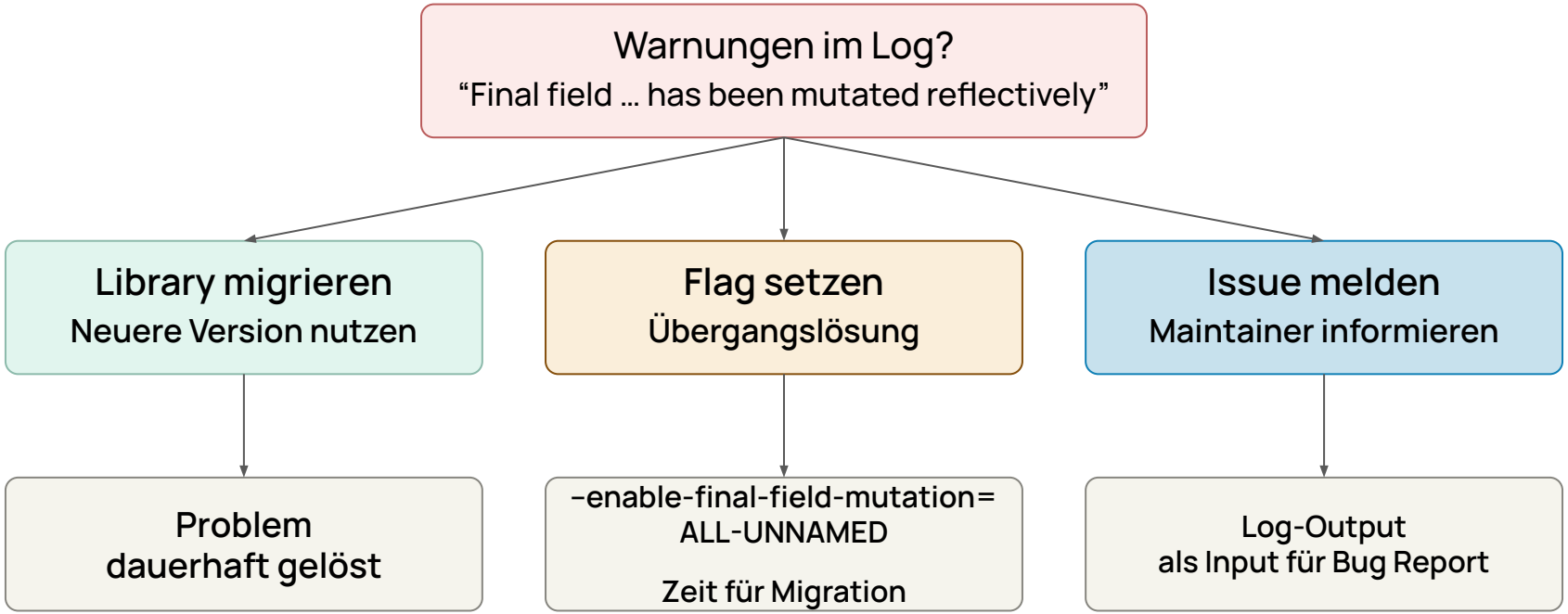
`--illegal-final-field-mutation=deny`

**Mit Java Flight Recorder analysieren:**

```
java -XX:StartFlightRecording:filename=recording.jfr ...
jfr print --events jdk.FinalFieldMutation recording.jfr
```



# Was tue ich, wenn ich betroffen bin?



# JEP 500: Prepare to Make Final Mean Final

| | |
|---|---|
| *Author* | Ron Pressler & Alex Buckley |
| *Owner* | Ron Pressler |
| *Type* | Feature |
| *Scope* | SE |
| *Status* | Closed / Delivered |
| *Release* | 26 |
| *Component* | core-libs |
| *Discussion* | jdk dash dev at openjdk dot org |
| *Reviewed by* | Alan Bateman, Brian Goetz |
| *Endorsed by* | Mark Reinhold |
| *Created* | 2025/02/06 10:25 |
| *Updated* | 2026/01/21 18:07 |
| *Issue* | 8349536 |

## Summary

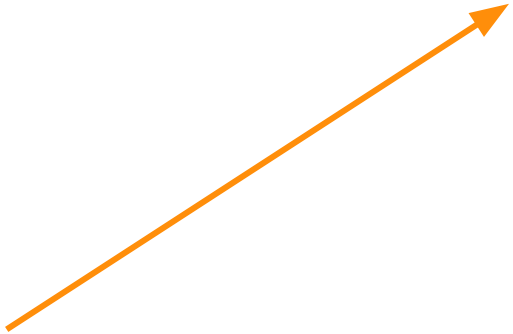
Issue warnings about uses of deep reflection to mutate final fields. These warnings aim to prepare developers for a future release that ensures integrity by default by restricting final field mutation, which will make Java programs safer and potentially faster. Application developers can avoid both current warnings and future restrictions by selectively enabling the ability to mutate final fields where essential.



# Das Problem – `final`-Feld als `null`

```java
class Base {
  Base() {
    IO.println(this);
  }
}
```

```java
class Child extends Base {
  private final String s;
  private final Integer i;

  Child(String s, Integer i) {
    this.s = s;
    this.i = i;
  }

  @Override
  public String toString() {
    return "s = %-8s i = %5d".formatted(s, i);
  }
}
```



```java
Child child = new Child("Berlin", 13088);
IO.println(child);
```



# Das Problem – `final`-Feld als `null`

```
class Base {
  Base() {
    IO.println(this);
  }
}
```

```
class Child extends Base {
  private final String s;
  private final int i;

  Child(String s, int i) {
    super();
    this.s = s;
    this.i = i;
  }

  @Override
  public String toString() {
    return "s = %-8s i = %5d".formatted(s, i);
  }
}
```

```
Child child = new Child("Berlin", 13088);
IO.println(child);
```



# Das Problem – `final`-Feld als `null`

```
class Base {
  Base() {
    IO.println(this);
  }
}
```

```
class Child extends Base {
  private final String s;
  private final int i;

  Child(String s, int i) {
    super();
    this.s = s;
    this.i = i;
  }

  @Override
  public String toString() {
    return "s = %-8s i = %5d".formatted(s, i);
  }
}
```

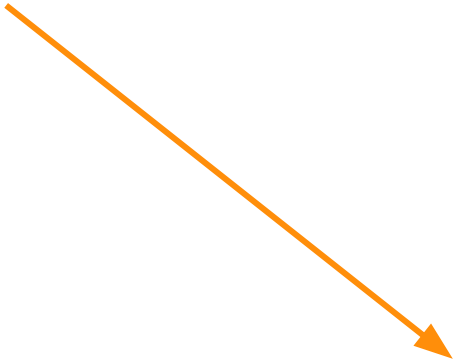
```
Child child = new Child("Berlin", 13088);
IO.println(child);
```



# Das Problem – `final`-Feld als `null`

```
class Base {
  Base() {
    IO.println(this);
  }
}
```

```
class Child extends Base {
  private final String s;
  private final int i;

  Child(String s, int i) {
    super();
    this.s = s;
    this.i = i;
  }

  @Override
  public String toString() {
    return "s = %-8s i = %5d".formatted(s, i);
  }
}
```



```
Child child = new Child("Berlin", 13088);
IO.println(child);
```



# Das Problem – `final`-Feld als `null`

```
class Base {
  Base() {
    IO.println(this);
  }
}
```

```
Child child = new Child("Berlin", 13088);
IO.println(child);
```

```
class Child extends Base {
  private final String s;
  private final int i;

  Child(String s, int i) {
    super();
    this.s = s;
    this.i = i;
  }

  @Override
  public String toString() {
    return "s = %-8s i = %5d".formatted(s, i);
  }
}
```



# Das Problem – `final`-Feld als `null`

```
class Base {
  Base() {
    IO.println(this);
  }
}
```

```
class Child extends Base {
  private final String s;
  private final int i;

  Child(String s, int i) {
    super();
    this.s = s;
    this.i = i;
  }

  @Override
  public String toString() {
    return "s = %-8s i = %5d".formatted(s, i);
  }
}
```

```
Child child = new Child("Berlin", 13088);
IO.println(child);
```



# Das Problem – `final`-Feld als `null`

```
class Base {
  Base() {
    IO.println(this);
  }
}
```

```
class Child extends Base {
  private final String s;
  private final int i;

  Child(String s, int i) {
    super();
    this.s = s;
    this.i = i;
  }

  @Override
  public String toString() {
    return "s = %-8s i = %5d".formatted(s, i);
  }
}
```

```
Child child = new Child("Berlin", 13088);
IO.println(child);
```

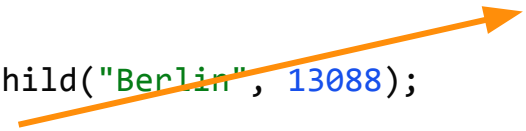


# Das Problem – `final`-Feld als `null`

```java
class Base {
  Base() {
    IO.println(this);
  }
}
```

```java
class Child extends Base {
  private final String s;
  private final int i;

  Child(String s, int i) {
    super();
    this.s = s;
    this.i = i;
  }

  @Override
  public String toString() {
    return "s = %-8s i = %5d".formatted(s, i);
  }
}
```

```java
Child child = new Child("Berlin", 13088);
IO.println(child);
```





# Das Problem – `final`-Feld als `null`

```
class Base {
  Base() {
    IO.println(this);
  }
}
```

```
Child child = new Child("Berlin", 13088);
IO.println(child);
```

```
class Child extends Base {
  private final String s;
  private final int i;

  Child(String s, int i) {
    super();
    this.s = s;
    this.i = i;
  }

  @Override
  public String toString() {
    return "s = %-8s i = %5d".formatted(s, i);
  }
}
```



# Das Problem – `final`-Feld als `null`

```
s = null     i =     0
s = Berlin   i = 13088
```



# JEP 513: Flexible Constructor Bodies

| | |
|---|---|
| *Author* | Archie Cobbs & Gavin Bierman |
| *Owner* | Gavin Bierman |
| *Type* | Feature |
| *Scope* | SE |
| *Status* | Closed / Delivered |
| *Release* | 25 |
| *Component* | specification / language |
| *Discussion* | amber dash dev at openjdk dot org |
| *Relates to* | JEP 492: Flexible Constructor Bodies (Third Preview) |
| *Reviewed by* | Alex Buckley, Brian Goetz |
| *Endorsed by* | Brian Goetz |
| *Created* | 2024/11/21 12:03 |
| *Updated* | 2025/12/15 23:40 |
| *Issue* | 8344702 |

## Summary

In the body of a constructor, allow statements to appear before an explicit constructor invocation, i.e., `super(...)` or `this(...)`. Such statements cannot reference the object under construction, but they can initialize its fields and perform other safe computations. This change allows many constructors to be expressed more naturally. It also allows fields to be initialized before they become visible to other code in the class, such as methods called from a superclass constructor, thereby improving safety.



## JEP draft: Null-Restricted Value Class Types (Preview)

| | |
|---|---|
| *Owner* | Dan Smith |
| *Type* | Feature |
| *Scope* | SE |
| *Status* | Draft |
| *Discussion* | valhalla dash dev at openjdk dot org |
| *Effort* | XL |
| *Duration* | XL |
| *Created* | 2023/09/22 23:57 |
| *Updated* | 2024/09/24 22:01 |
| *Issue* | 8316779 |

### Summary

Allow the type of a variable storing value objects to exclude `null`, enabling more compact storage and other optimizations at run time. This is a preview language and VM feature.

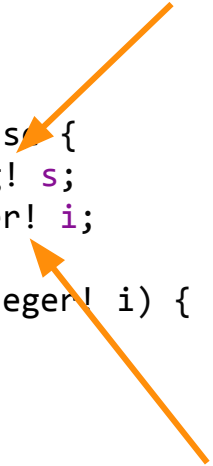


# Das Problem – `final`-Feld als `null`

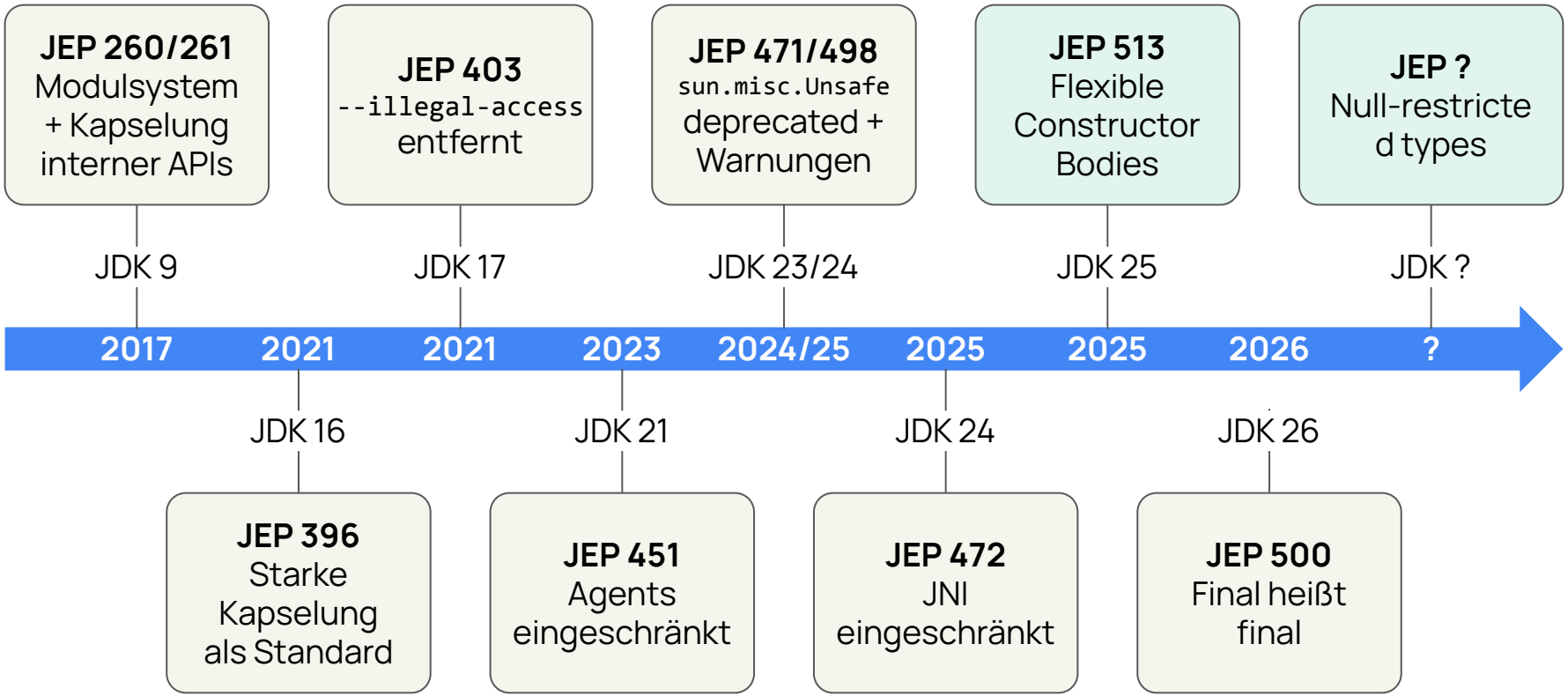
```
class Base {
  Base() {
    IO.println(this);
  }
}
```

```
Child child = new Child("Berlin", 13088);
IO.println(child);
```

```
class Child extends Base {
  private final String! s;
  private final Integer! i;

  Child(String! s, Integer! i) {
    this.s = s;
    this.i = i;
    super();
  }

  @Override
  public String toString() {
    return "s = %-8s i = %5d".formatted(s, i);
  }
}
```







## JEP draft: Integrity by Default

| | |
|---|---|
| *Authors* | Ron Pressler, Alex Buckley, & Mark Reinhold |
| *Owner* | Ron Pressler |
| *Type* | Informational |
| *Scope* | SE |
| *Status* | Draft |
| *Relates to* | JEP 261: Module System<br>JEP 260: Encapsulate Most Internal APIs<br>JEP 396: Strongly Encapsulate JDK Internals by Default<br>JEP 403: Strongly Encapsulate JDK Internals<br>JEP 451: Prepare to Disallow the Dynamic Loading of Agents<br>JEP 498: Warn upon Use of Memory-Access Methods in sun.misc.Unsafe<br>JEP 471: Deprecate the Memory-Access Methods in sun.misc.Unsafe for Removal<br>JEP 500: Prepare to Make Final Mean Final<br>JEP 472: Prepare to Restrict the Use of JNI |
| *Created* | 2023/04/13 16:06 |
| *Updated* | 2025/11/20 01:24 |
| *Issue* | 8305968 |

https://openjdk.org/jeps/8305968

### Summary

Developers expect that their code and data is protected against use that is unwanted or unwise. The Java Platform, however, contains unsafe APIs that can undermine this expectation, thereby damaging the correctness, maintainability, scalability, security, and performance of applications. Going forward, we will restrict the unsafe APIs so that, by default, libraries, frameworks, and tools cannot use them. Application authors will have the ability to override this default.

Links + Slides


**Sven Woltmann**

Java Consultant & Trainer

sven@happycoders.eu

https://www.happycoders.eu/

https://linkedin.com/in/sven-woltmann/

https://youtube.com/c/happycoders


LinkedIn