

Final heißt final:

Warum $2 + 2$ bald nicht mehr 5
ergeben darf

Sven Woltmann

Java-Trainer, Speaker & Autor



Sven Woltmann

Java-Trainer, Speaker & Autor

„Mein Ziel: komplexe Themen so vermitteln, dass sie verständlich, praxisnah und direkt anwendbar sind.“

Was ist da passiert?

```
public class ApparentlyHarmlessThirdPartyLibrary {
```

```
...
```

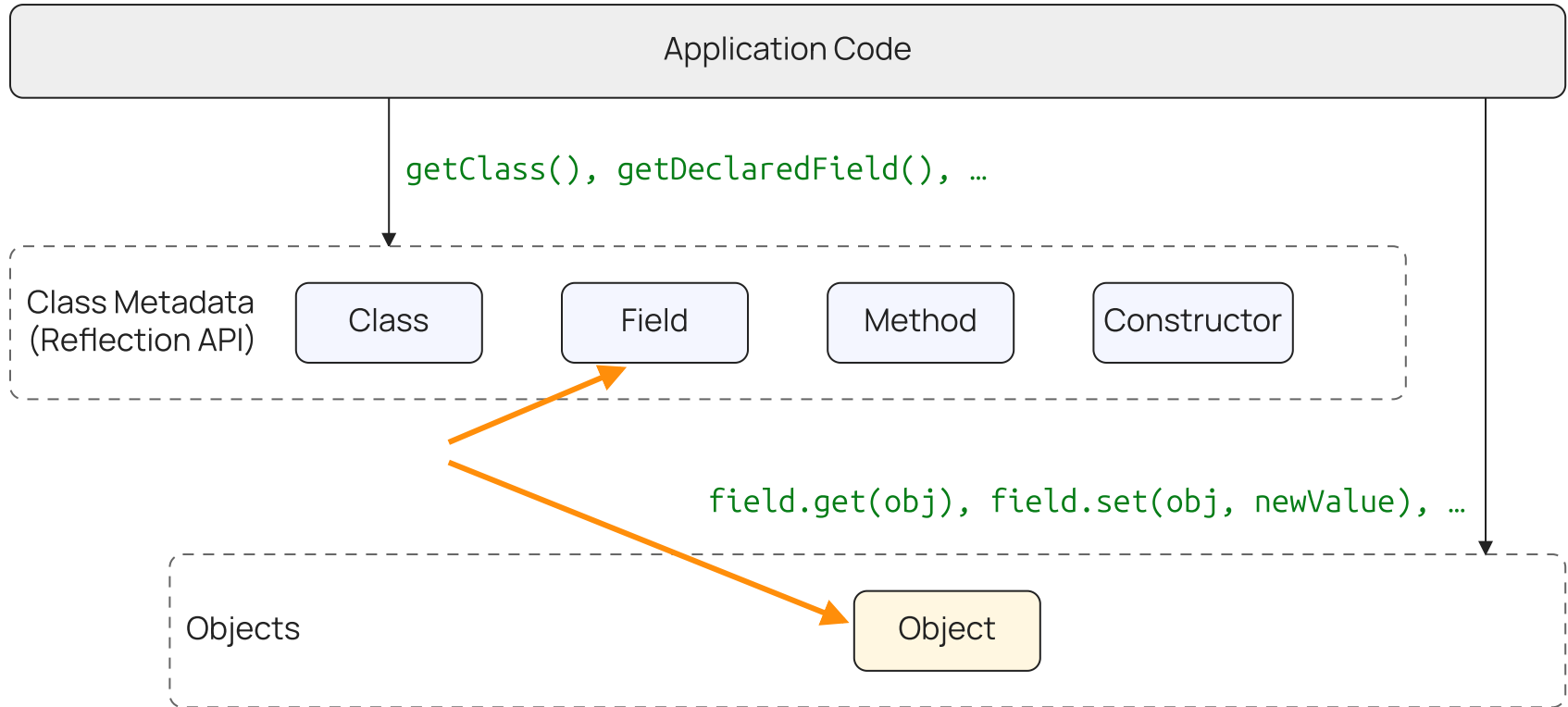
```
static {  
    try {  
        Field VALUE = Integer.class.getDeclaredField("value");  
        VALUE.setAccessible(true);  
        VALUE.set(4, 5);  
    } catch (ReflectiveOperationException e) {  
        throw new Error(e);  
    }  
}
```

```
Integer sum = 2 + 2;  
System.out.println("2 + 2 = " + sum);
```



2 + 2 = 5

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;  
}
```

Demo-Code: github.com/SvenWoltmann/final-means-final

Was geht und was geht nicht?

```
final int value = 1;
```

```
VALUE.setAccessible(true);
```

```
VALUE.set(obj, 5);
```

```
// gelingt:
```

```
// Wert wird überschrieben
```

```
static final int value = 1;
```

```
VALUE.setAccessible(true);
```

```
VALUE.set(null, 5);
```

```
// IllegalAccessException:
```

```
// Can not set static final field
```

Dasselbe Programm, drei JVMs

```
$ java8 TwoPlusTwo  
2 + 2 = 5
```

```
$ java11 TwoPlusTwo  
WARNING: An illegal reflective access operation has occurred  
WARNING: Illegal reflective access by ApparentlyHarmlessThirdPartyLibrary  
         to field java.lang.Integer.value  
WARNING: All illegal access operations will be denied in a future release  
2 + 2 = 5
```

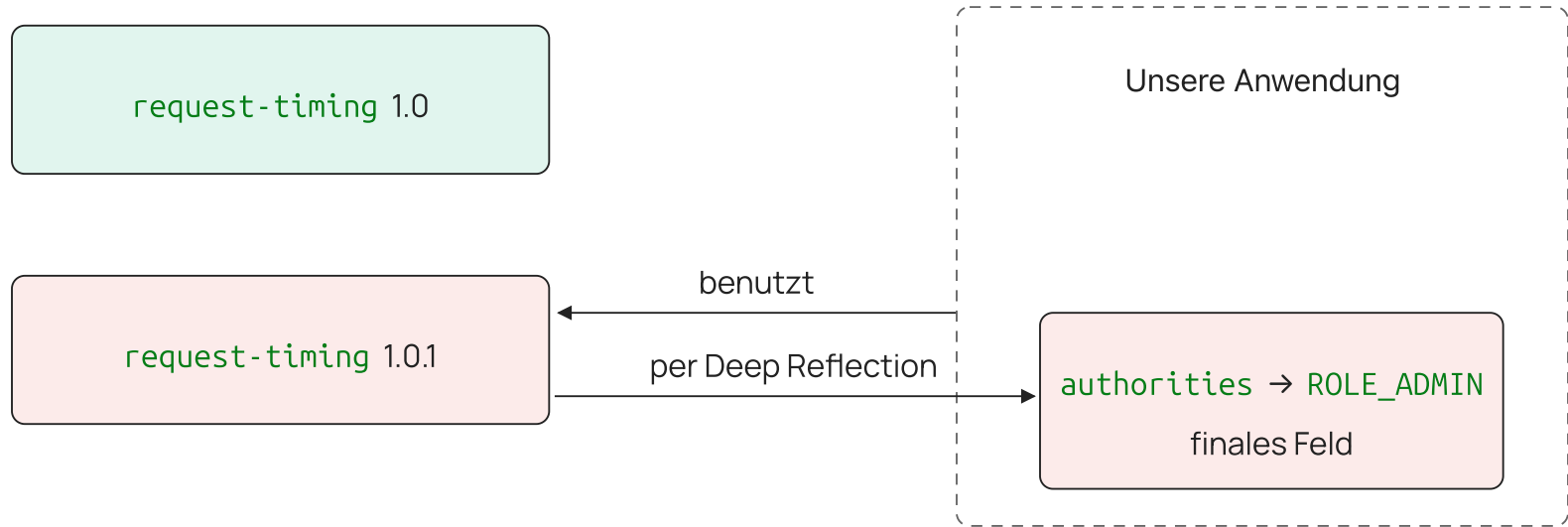
```
$ java17 TwoPlusTwo  
Exception in thread "main" java.lang.reflect.InaccessibleObjectException:  
  Unable to make field private final int java.lang.Integer.value accessible:  
  module java.base does not "opens java.lang" to unnamed module
```

Nur bei
Aufrufen über
Modulgrenzen
hinweg

Was passiert, wenn jemand das gegen dich einsetzt?

Ein realistisches Angriffsszenario

Was das Patch-Update mitbrachte



**Kein Exploit.
Nur Deep Reflection.**

Supply-Chain-Angriffe sind real

npm
EventStream

2018

Bitcoin-Wallets ausgeräumt

npm
{UAParser.js}

2021

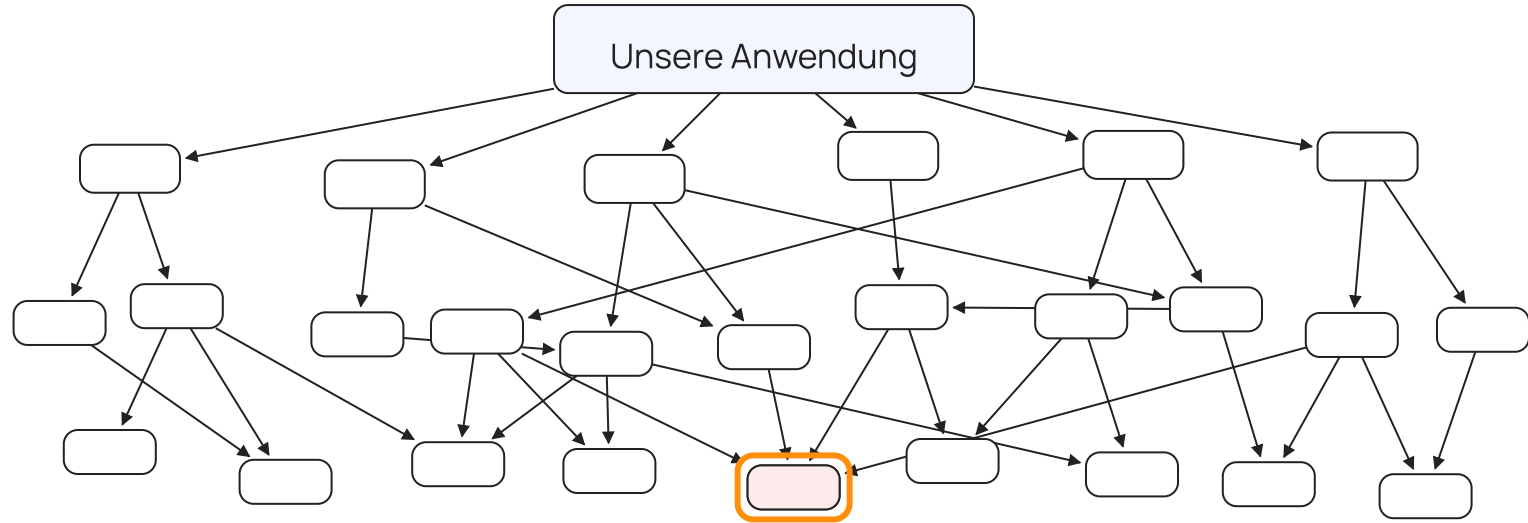
Krypto-Miner eingeschleust



2024

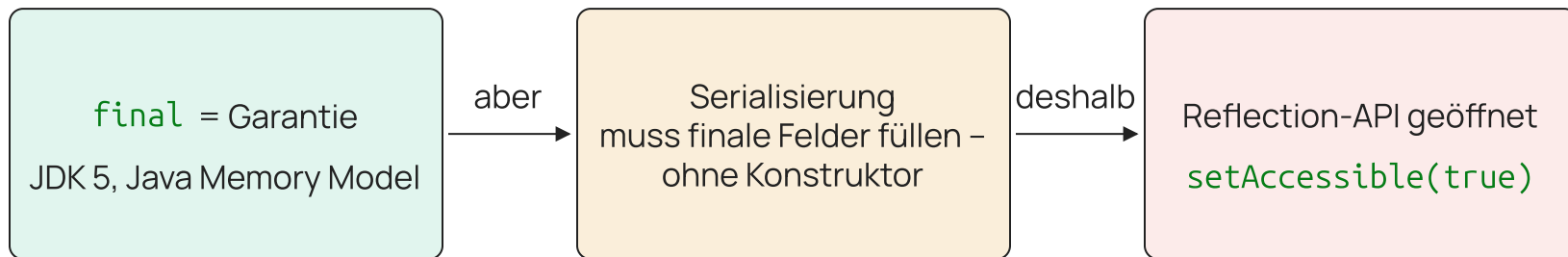
Hintertür Richtung SSH

Dein Classpath – dreistellig viele JARs



Keine Lücke in deiner Anwendung nötig –
nur ein Weg in deinen Classpath.

Warum gibt es die Hintertür?

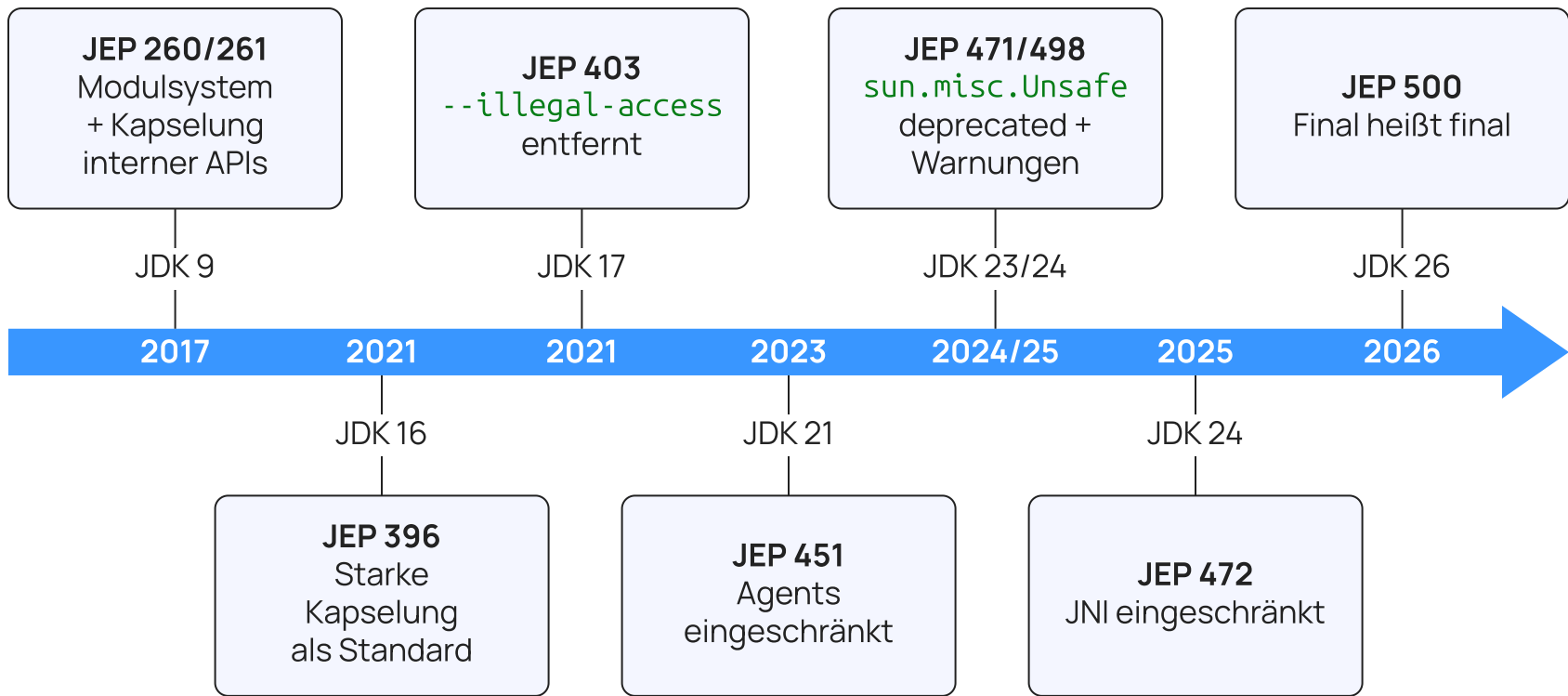


"In retrospect, offering such unconstrained functionality was a poor choice."

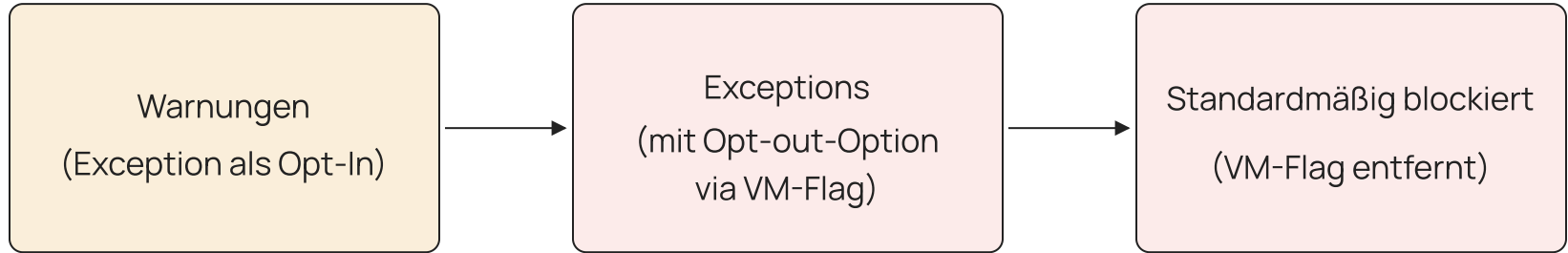
– JEP 500

Integrity by default

Javas langfristiges Programm für Laufzeitintegrität



Immer dasselbe Muster



JEP 500: Wann `Field::set` noch schreibt

`Field::set` schreibt ein finales Feld nur, wenn alle drei erfüllt sind:

- ✓ Paket zum Modul des Aufrufers geöffnet
seit Java 9/16/17 – Kapselung – wohin: `--add-opens`
 - ✓ Final-Field-Mutation für das Modul erlaubt
neu – Java 26, JEP 500 – wer: `--enable-final-field-mutation`
 - ✓ `setAccessible(true)` ist erfolgreich
wie seit JDK 5
- } alle drei
nötig

JEP 500: Die neuen Kommandozeilen-Optionen

`--enable-final-field-mutation`

ALL-UNNAMED

→ Mutation erlaubt für allen Code auf dem Classpath

com.example.mymodule

→ Mutation erlaubt für ein bestimmtes Modul

`--illegal-final-field-mutation`

allow

→ Mutation erlaubt, keine Warnung

warn

→ Mutation erlaubt, einmalige Warnung

debug

→ Mutation erlaubt, Warnung +
Stack Trace bei jedem Aufruf

deny

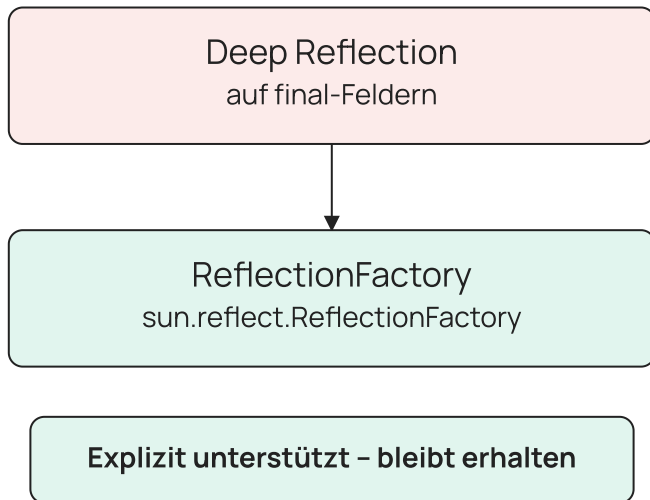
→ `IllegalAccessException`, Mutation verweigert

Standard in Java 26

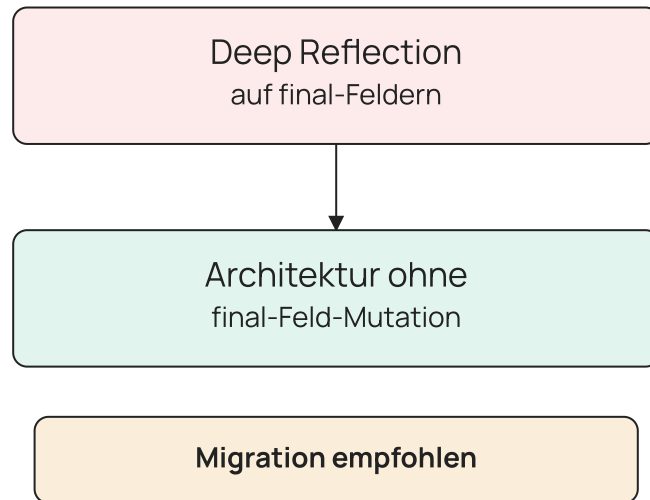
Standard in Zukunft

JEP 500: Was bedeutet das für Libraries?

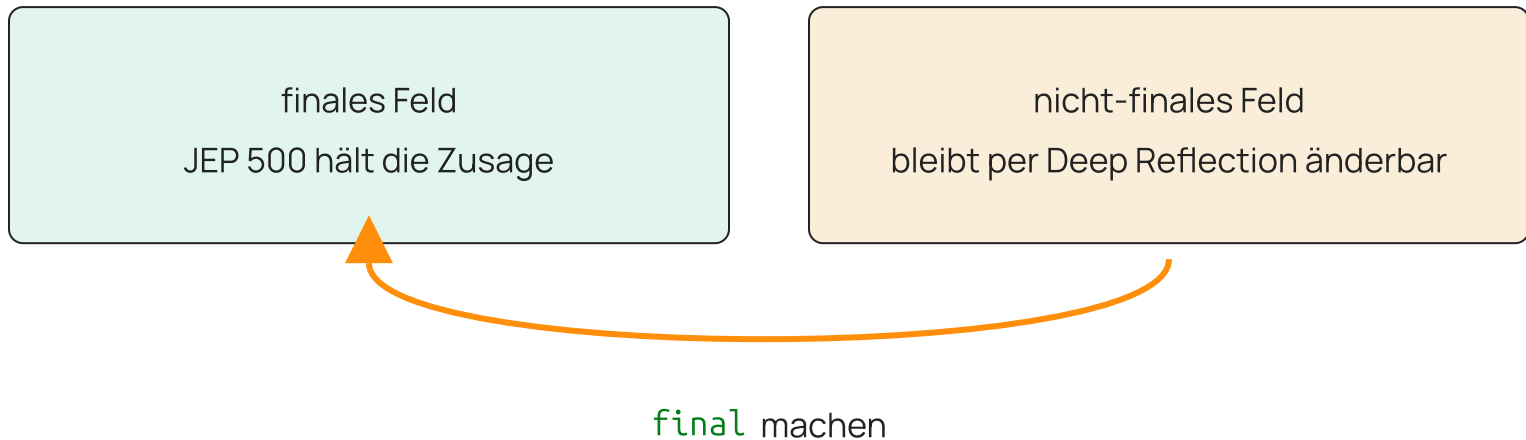
Serialization-Libraries



DI / Mocking / Testing



Was JEP 500 schützt – und was nicht



Constant Folding: schon zur Compile-Zeit

```
static final int RATE = 2 + 2;
```

```
int apply(int amount) {  
    return amount * RATE;  
}
```

```
int apply(int):
```

```
    iload_1    // amount
```

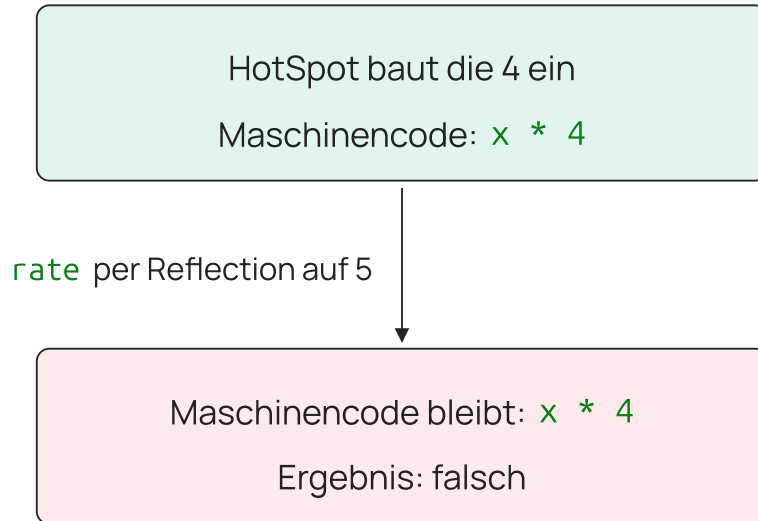
```
    → iconst_4 // die Konstante 4
```

```
    imul
```

```
    ireturn
```

Constant Folding: und zur Laufzeit

```
final int rate = 4;  
int total(int x) {  
    return x * rate;  
}
```



HotSpot vertraut nur wenigen finalen Feldern

Zurück zum Angriff – jetzt unter Java 27

```
WARNING: Final field authorities in class  
    org.springframework.security.authentication.AbstractAuthenticationToken  
    has been mutated reflectively by class  
    io.metricstack.requesttiming.RequestTimingFilter in unnamed module  
    (file:../io/metricstack/request-timing/1.0.1/request-timing-1.0.1.jar)  
WARNING: Use --enable-final-field-mutation=ALL-UNNAMED to avoid a warning  
WARNING: Mutating final fields will be blocked in a future release  
    unless final field mutation is enabled
```

Final heißt jetzt final

Spring-App mit `--illegal-final-field-mutation=deny`:

→ `Field::set` wirft eine `IllegalAccessException`:

```
java.lang.IllegalAccessException: class io.metricstack.requesttiming.RequestTimingFilter cannot set
final field org.springframework.security.authentication.AbstractAuthenticationToken.authorities,
unnamed module is not allowed to mutate final fields
    at java.base/java.lang.reflect.Field.preSetFinal(Field.java:1516)
    at java.base/java.lang.reflect.Field.setFinal(Field.java:1459)
    at java.base/java.lang.reflect.Field.set(Field.java:921)
    at io.metricstack.requesttiming.RequestTimingFilter.collectRequestContext(RequestTimingFilter:68)
    at io.metricstack.requesttiming.RequestTimingFilter.doFilter(RequestTimingFilter:40)
    ...
```

→ `alice` hat keine Admin-Rechte mehr.

→ Da die Library die Exception abfängt, läuft die Anwendung weiter – nur der Angriff nicht.

Von Java 8 bis in die Zukunft

JVM	Optionen	2 + 2	Spring-App
Java 8	–	5	Angriff möglich
Java 11	–	5 + Reflection-Warnung	Angriff möglich
Java 17	–	<code>InaccessibleObjectException</code>	Angriff möglich
Java 17	<code>--add-opens</code>	5	
Java 26	<code>--add-opens</code>	5 + Final-Field-Warnung	Angriff möglich, mit Warnung
Java 26	<code>--add-opens +</code> <code>--enable-final-field-mutation</code>	5	
Zukünftige Version	<code>--add-opens</code> , ohne <code>--enable-final-field-mutation</code>	<code>IllegalAccessException</code>	Angriff blockiert

Zwei unabhängige Bedingungen:

`--add-opens` regelt, *wohin* du reflektierst,

`--enable-final-field-mutation` regelt, *wer* schreiben darf.

Bin ich betroffen?

Auf Java 26/27 steht die Antwort schon im Log:

Standard `warn`: Warnung beim ersten Mal je Modul

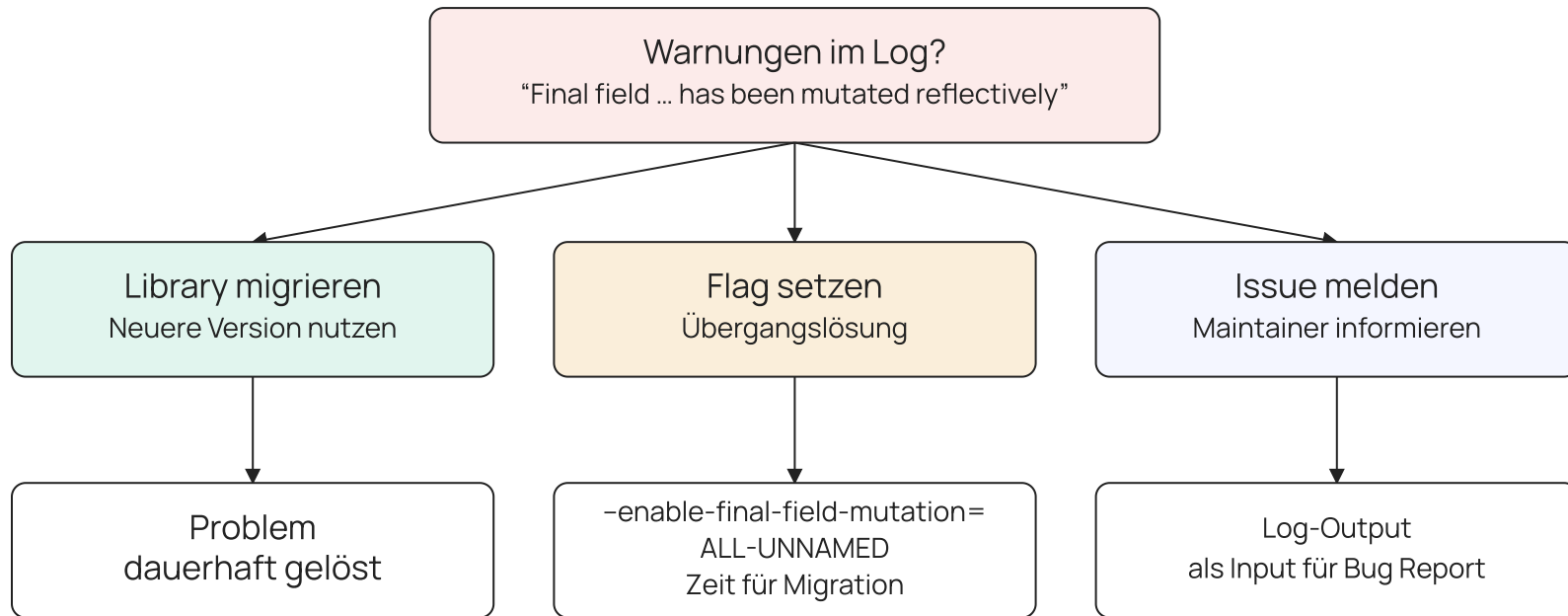
Jede Mutation mit Stack Trace:

```
java --illegal-final-field-mutation=debug ...
```

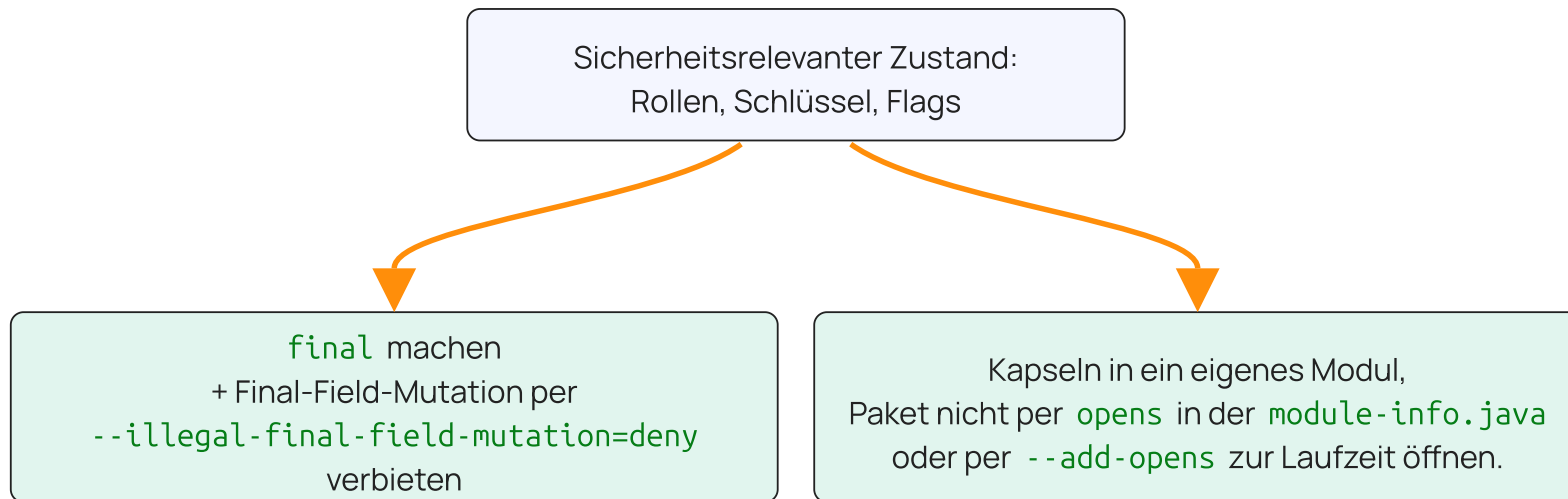
Maschinell auswerten – Java Flight Recorder:

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

Was tue ich, wenn ich betroffen bin?



Was ihr selbst tun könnt



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);  
    }  
}
```

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

JEP 513: Flexible Constructor Bodies

```
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) {  
        this.s = s;  
        this.i = i;  
        super();  
    }
```



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

```
s = Berlin    i = 13088  
s = Berlin    i = 13088
```

JEP 539: Strict Field Initialization

JEP 513 gibt euch das Werkzeug – JEP 539 erzwingt den Einsatz (zumindest für Value Classes).

Ein *strikt initialisiertes* Feld muss vor `super()` gesetzt sein – und ein finales danach unveränderlich bleiben.

→ Das finale Feld ist nur noch in *einem* Zustand sichtbar – von der JVM erzwungen.

Aber noch mit Grenzen:

- Preview in Java 28 (`--enable-preview`)

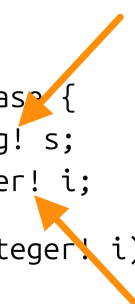
- `javac` markiert Felder als „strictly initialized“ bisher nur für *Value Classes* (JEP 401).

Und für normale Klassen (Identity Classes)?

Aktuell nicht geplant – es würde große Teile des Java-Ökosystems unkompilierbar machen. Die Technik ließe es zu; mit einer langen Übergangsphase wäre es denkbar.

Entwurf: Null-Restricted and Nullable Types

```
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);  
    }  
}
```



Ein Schritt von vielen.

JEP 500: Prepare to Make Final Mean Final (Java 26)

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

JEP Draft „Integrity by Default“ ist noch ein Entwurf – und noch nicht zu Ende.

Links +
Slides



Sven Woltmann

Java-Trainer, Speaker & Autor



sven@happycoders.eu



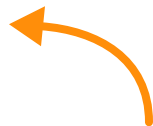
<https://www.happycoders.eu/>



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



<https://youtube.com/c/happycoders>



LinkedIn