

> Überraschungen und Grundlagen bei der nebenläufigen Programmierung in Java



JUG Karlsruhe
16. Oktober 2013

Referent: Christian Kumpe



- ▶ **Was zeigt dieser Vortrag?**

Ein „einfaches“ Beispiel mit allerlei Tücken.

- ▶ **Was erklärt dieser Vortrag?**

Welche Komponenten mit welchen *grundlegenden* Regeln für das Laufzeitverhalten von nebenläufigen Programmen in Java verantwortlich sind.

- ▶ **Was erklärt dieser Vortrag *nicht*?**

Der Vortrag ist nicht als Einführung oder Anleitung zur nebenläufigen Programmierung in Java gedacht.

Christian Kumpe
Senior Softwareentwickler

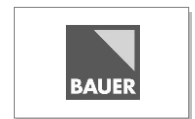
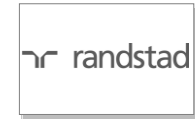
- ▶ Informatikstudium am KIT (Universität Karlsruhe)
- ▶ Freelancer im Bereich Web und Java
- ▶ Seit Mai 2011 bei Netpioneer
- ▶ Über 10 Jahre Java-Erfahrung



christian.kumpe@netpioneer.de



- ▶ Gründung 1996
- ▶ IT-Dienstleister für hochwertige Softwarelösungen im Onlinesektor
- ▶ Über 100 Mitarbeiter
davon um die 80 in Karlsruhe
- ▶ 30 aktive Bestandskunden
- ▶ Hauptsitz in Karlsruhe
- ▶ Standort in Berlin



```
public class SimpleExample {
    static class Looper extends Thread {
        boolean finish = false;

        @Override
        public void run() {
            while (!finish) {
                // do something
            }
        }
    }

    public static void main(String... args) throws Exception {
        Looper looper = new Looper();
        looper.start();

        Thread.sleep(1000); // wait 1s
        looper.finish = true;

        System.out.println("Wait for Looper to terminate...");
        looper.join();
        System.out.println("Done.");
    }
}
```

DEMO

- ▶ **Das ursprüngliche Programm ist fehlerhaft!**

Änderungen an `finish` kommen im Looper Thread nicht an.

- ▶ **`volatile` korrigiert den Fehler!**

Das Programm terminiert wie gewünscht.

- ▶ **`System.out.println()` ; behebt das Problem auch!**

War das hier nur Zufall?

- ▶ **Auch der Debugger hat Einfluss auf das Verhalten des Programms!**

Wie kann das sein?



- ▶ **Der JIT Compiler entfernt die Überprüfung von `finish`.**

Dadurch wird die `while`-Schleife zur Endlosschleife.

- ▶ **Ein `volatile` verbietet das.**

Weil der JIT Compiler jetzt damit rechnen muss, dass die Variable von einem anderen Thread verändert wird.

- ▶ **`System.out.println()` synchronisiert zwischen Threads.**

Da beide Threads `System.out` als Lock verwenden, wird die Änderung der Variable im Looper Thread sichtbar.

- ▶ **Das Setzen des Breakpoints sorgt für eine Deoptimierung.**

Damit findet wieder eine Überprüfung der Variable statt.

**Darf der
das?**

► **Kompilierung**

Java Source Code wird vom Java Compiler in Bytecode gewandelt:

```
$ javac SimpleExample.java
```

► **Ausführung**

Bytecode wird von der *Java Virtual Machine (JVM)* ausgeführt:

```
$ java SimpleExample  
Wait for Looper to terminate..  
Done.
```

Dabei wird der *Bytecode* interpretiert oder vom *Just in Time (JIT) Compiler* in *Maschinencode* umgewandelt und ausgeführt.

► **Laufzeitverhalten**

Das plattformunabhängige Laufzeitverhalten wird dabei in der *Java Language Specification (JLS)* bzw. der *Java Virtual Machine Specification (JVMS)* festgelegt.



- ▶ **Ist Teil der Java Language Specification.**

Liefert die Grundlagen für das beobachtete Verhalten und wird im Folgenden genauer betrachtet.

- ▶ **Wofür braucht man ein Memory Model?**

Wenn einer Variable *v* in Thread A ein Wert zu gewiesen wird...

```
public int v;  
...  
v = 1;
```

...unter welchen Umständen sieht Thread B diesen Wert?

```
if (v == 1) {  
    System.out.println("Der Wert ist 1");  
}
```

- ▶ **Reihenfolge der Aktionen innerhalb eines Threads.**

Innerhalb eines Threads gilt die *as-if-serial* Semantik.

- ▶ **Reihenfolge der Aktionen verschiedener Threads.**

Für die Aktionen zwischen verschiedenen Threads gelten die *happens-before* Regeln.

- ▶ **Was wird damit gesteuert?**

Der Java *Compiler* und die *JVM* mit ihrem *JIT Compiler* müssen sich so verhalten und die CPU, die Caches und den Speicher so steuern, dass das geforderte Laufzeitverhalten auf der jeweiligen Plattform resultiert.

▶ **Programm order rule**

Each action in a thread *happens-before* every action in that thread that comes later in the program order.

▶ **Monitor lock rule**

An unlock on a monitor lock *happens-before* every subsequent lock on that same monitor lock.

▶ **Volatile variable rule**

A write to a **volatile** field *happens-before* every subsequent read of that field.

▶ **Thread start rule**

A call to `Thread.start` on a thread *happens-before* every action in the started thread.

(Übernommen aus „Java Concurrency In Practice“ von Brian Goetz u.a.)

▶ **Thread termination rule**

Any action in a thread *happens-before* any other thread detects that thread has terminated, either by successfully return from `Thread.join` or by `Thread.isAlive` returning **false**.

▶ **Interruption rule**

A thread calling `Thread.interrupt` on another thread *happens-before* the interrupted thread detects the interrupt (either by having `InterruptedException` thrown or invoking `Thread.isInterrupted` or `Thread.interrupted`).

▶ **Finalizer rule**

The end of a constructor for an object *happens-before* the start of the finalizer for that object.

▶ **Transitivity**

If A *happens-before* B, and B *happens-before* C, then A *happens-before* C.

(Übernommen aus „Java Concurrency In Practice“ von Brian Goetz u.a.)

► Können wir die Annahmen überprüfen?

Wie können wir überprüfen welche Optimierungen der JIT Compiler zur Laufzeit vornimmt?

► HotSpot bietet Schalter für weitere Debugging-Informationen.

Mit -XX:+PrintCompilation können wir sehen, dass der JIT Compiler die fragliche Methode optimiert hat:

```
$ java -XX:+PrintCompilation SimpleExample
      79      1 %          SimpleExample$Looper::run @ 0 (8 bytes)
Wait for Looper to terminate...
```

► Und wenn wir's genau sehen wollen...

Können wir uns mit -XX:+PrintAssembly den generierten Assembler Code ansehen!

Dazu wird das HotSpot Disassembler Plugin benötigt:

<https://kenai.com/projects/base-hsdis/>

DEMO

- ▶ **Die Optimierungen der JVM können manchmal überraschend sein.**

Sollten aber immer konform zur Java Language Specification sein!

- ▶ **Ein grundlegendes Wissen darüber kann bei der Fehlersuche hilfreich sein.**

Zumindest sollte man sich bewusst sein, was alles passieren kann.

- ▶ **Die HotSpot JVM mit ihrem JIT Compiler ist eine spannende Software!**

Und es lohnt sich, ihr ab und an unter die Haube zu schauen!

► Literatur

„Java Concurrency In Practice“ von Brian Goetz, Tim Peierls, Joshua Bloch und weiteren, erschienen im Addison Wesley Verlag. ISBN 0-321-34960-1.

„Java Performance“ von Charlie Hunt und Binu John, erschienen im Addison Wesley Verlag. ISBN 0-137-14252-8.

► Links

HotSpot Internals: <https://wikis.oracle.com/display/HotSpotInternals/Home>

OpenJDK JVM Internals: <http://www.progdoc.de/papers/Jax2012/jax2012.html>

WILLST DU MIT MIR

(ARBEITEN) GEHEN?

Bitte ankreuzen

☐ JA

☐ NEIN

☐ VIELLEICHT



netpioneer

ENDE

Herzlichen Dank!

Ich hoffe Ihr seid um eine
Erfahrung reicher geworden 😊

Habt Ihr noch Fragen?

► **Pizza und Getränke in der Cafeteria von Netpioneer im 7. OG**

Bier gibt's im Kühlschrank, weitere Getränke unterm Tisch.

► **Fragen zu Netpioneer?**

Fragt die Leute mit den Namensschildern, die wissen Bescheid!

► **In der Tiefgarage geparkt?**

Die Münzen, um die Tiefgarage wieder verlassen zu können, gibt's bei uns.



Unsere Kickertische stehen bereit!

Im 7. OG neben der Cafeteria.