

**Application Performance Management**

# **Einführung Performance-Analyse**

Michael Faes

# Persönliches

Michael Faes

[michael.faes@fhnw.ch](mailto:michael.faes@fhnw.ch)

Wohnort: Aesch BL



Dazwischen: Reisen, Fotografie,  
Beach Volley, Kino, Jungschar

Bachelor ETH Zürich

Praktikum Canoo AG

Master ETH

Doktorat ETH

- **Programmiersprachen**
- **Parallelismus, Concurrency**
- Software-Engineering-Tools
- Education-Tools

---

Dozent FHNW

Informatiklehrer Gymnasium

dieses Modul

“Make it Work, Make it Right, *Make it Fast.*”

— Kent Beck, Erfinder von Extreme Programming

In dieser Reihenfolge!

“Premature optimization is the root of all evil in programming.”

— Donald Knuth

Performance ist trotzdem oft wichtig! Die Kunst ist es, die grossen Gewinne zu identifizieren und einzuholen.

# Es geht um Performance

Performance **messen**

Performance-**Probleme**  
**identifizieren**

Performance **charakterisieren**

Performance **vorhersagen**

Performance **vergleichen**

Performance **«optimieren»**

Performance **visualisieren**

# Demo: Performance messen

Performance-Messungen mit *aussagekräftigen* Resultaten sind alles andere als trivial.

**Demo:** Ausführungszeit einer Methode in Java messen

## Probleme

- Variation
- Genauigkeit
- Warmup
- Interaktion mit anderem Code

```
public class Factorization {  
  
    private static final long NUMBER = 64273117L;  
  
    public static void main(String[] args) {  
        var start = System.currentTimeMillis();  
  
        var factors = factorize(NUMBER);  
        System.out.println(NUMBER + ": " + factors);  
  
        double time = (System.currentTimeMillis() - start);  
        System.out.printf("%.2f ms\n\n", time);  
    }  
  
    private static List<Long> factorize(long num) {  
        var factors = new ArrayList<Long>();  
        long factor = 2;  
        while (factor <= num) {  
            while (num % factor == 0) {  
                factors.add(factor);  
                num /= factor;  
                if (num == 1) {  
                    return factors;  
                }  
            }  
            factor++;  
        }  
        throw new AssertionError();  
    }  
}
```

# Wichtigste Lernziele

- Sie können *Techniken*, *Metriken* und *Workloads* zum Evaluieren der Performance eines Systems angemessen auswählen.
- Sie können *Performance-Messungen* korrekt durchführen.
- Sie kennen verschiedene Techniken zur *Performance-Optimierung* einer Applikation und können diese erfolgreich einsetzen.
- Sie können Daten aus Performance-Messungen durch *Visualisierung* und mittels *statistischer Methoden* auswerten.
- Sie kennen die Techniken *JIT-Kompilierung*, *Garbage Collection* und *Caching* und können deren Einfluss auf die Performance einschätzen.
- Sie kennen die Grundlagen von *Load Balancing*, *Clustering* und *Autoscaling* und können diese Techniken einsetzen, um eine Applikation skalierbar und hochverfügbar zu machen.

# Modulübersicht

## 3. Semesterplan

W	Datum	Inhalte
1	21.02.	Einführung Performance-Analyse & Parallelisierung
2	28.02.	Performance-Optimierung & Profiling
3	07.03.	Experiment-Design & Benchmarking
4	14.03.	Datenvisualisierung
5	21.03.	Garbage Collection
6	28.03.	Just-in-time-Kompilierung
7	04.04.	Caching
8	11.04.	<b>Assessment 1</b>
	18.04.	<i>Osterferien</i>
9	25.04.	Performance von Web-Apps & Load Testing
10	02.05.	Statistische Auswertung
	09.05.	<i>Projektwoche</i>
11	16.05.	Load Balancing
12	23.05.	Clustering
13	30.05.	Autoscaling
14	06.06.	<i>Pfingstmontag</i>
15	13.06.	<b>Assessment 2</b>

Wochen 1 – 4  
**Grundlagen**

Wochen 5 – 7  
**Spezialthemen**  
(Gastdozent: Dr. Majó)

Wochen 9 – 13  
**Web-Apps & Cloud**

# Leistungsbeurteilung

## 3. Semesterplan

W	Datum	Inhalte
1	21.02.	Einführung Performance-Analyse & Parallelisierung
2	28.02.	Performance-Optimierung & Profiling
3	07.03.	Experiment-Design & Benchmarking
4	14.03.	Datenvisualisierung
5	21.03.	Garbage Collection
6	28.03.	Just-in-time-Kompilierung
7	04.04.	Caching
8	11.04.	<b>Assessment 1</b>
	18.04.	<i>Osterferien</i>
9	25.04.	Performance von Web-Apps & Load Testing
10	02.05.	Statistische Auswertung
	09.05.	<i>Projektwoche</i>
11	16.05.	Load Balancing
12	23.05.	Clustering
13	30.05.	Autoscaling
14	06.06.	<i>Pfingstmontag</i>
15	13.06.	<b>Assessment 2</b>

## Leistungsbeurteilung 2 Assessments

60 min schriftlich

Hilfsmittel:

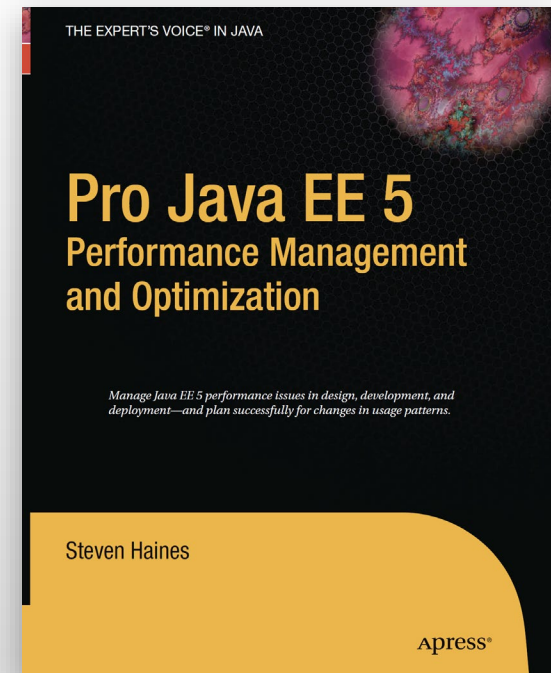
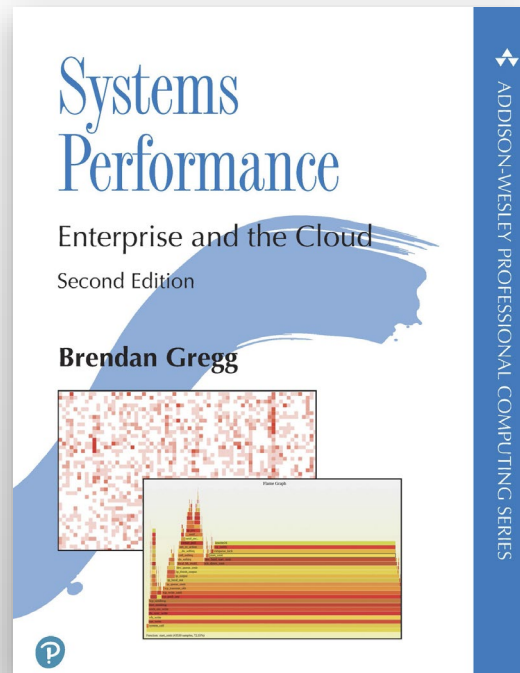
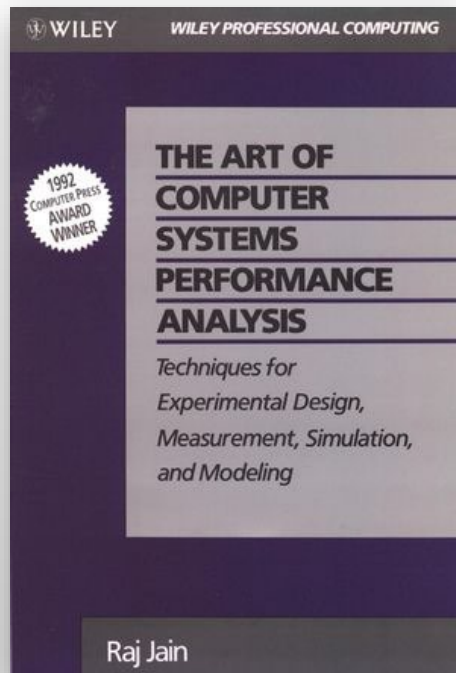
- je eine A4-Seite  
Zusammenfassung



# Lernmaterial

**Folien, Scripts & Übungen:** <https://github.com/apm-fhnw/apm-fs22>

**Bücher:** Kurs basiert auf verschiedenen Büchern. Keine Pflichtlektüre, aber für Interessierte vielleicht nützlich:



# Übersicht Woche 1

1. Modulübersicht & Administratives
2. Einführung Performance-Analyse
3. Evaluationstechniken & Performance-Metriken
4. Parallelisierung mit Java
5. Übung: *Doc Finder*

# **Einführung Performance-Analyse**

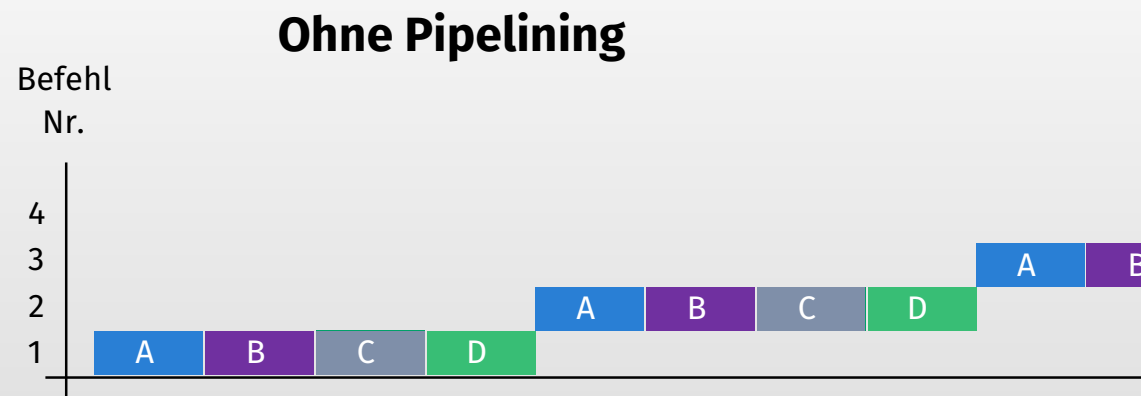
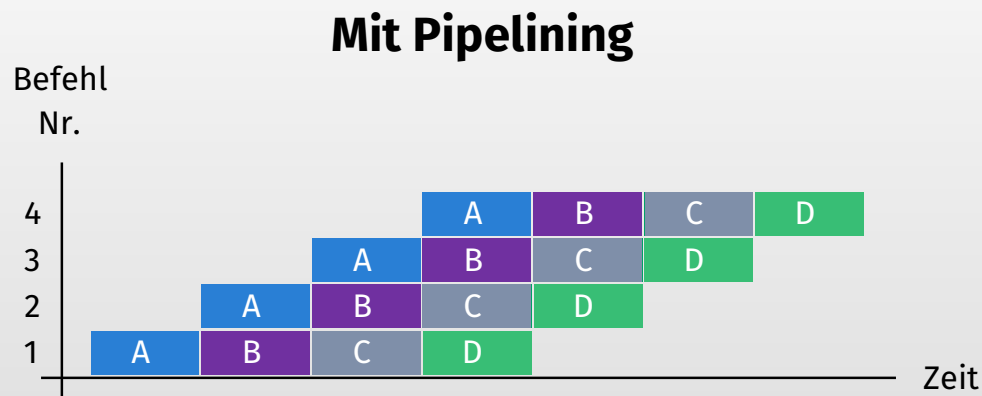
# Performance ist schwierig

Performance ist komplex. Siehe Demo...

**Grund:** Performance-Optimierungen sind nicht oft nicht «uniform».

- Oft wird für **typische Workloads** optimiert, da insgesamt effizienter

**Beispiel:** *Instruction Pipelining*: erhöht CPU-Leistung in **vielen** Fällen



## Weitere Beispiele

- *Branch Prediction*: Um Pipelining effizienter zu machen, wird Verhalten von Sprung-Instruktionen **vorhergesagt**.

Funktioniert in bis zu 98% der Fällen. Aber wenn nicht...

- *JIT-Kompilierung*: Nur Code, der **oft genug** ausgeführt wurde, wird kompiliert und optimiert. Wird für zahlreiche Sprachen gemacht:



- *Caching*: Sparen von mehrmaligen Berechnungen oder Abrufen durch Zwischenspeichern. Ebenfalls durch **Vorhersage**.

Performance hängt von unglaublich vielen Faktoren ab.

**Beispiel 1:** Paper [\*Producing wrong data without doing anything obviously wrong!\*](#)

- *Anzahl/Grösse der Umgebungsvariablen* führt zu unterschiedlichen Resultaten beim Vergleichen von Compiler-Optimierungen

**Beispiel 2:** *Shouting in the datacenter*



<https://youtu.be/tDacjrSCeq4>

Auswertung von Performance-Daten ebenfalls nicht trivial.

**Beispiel:** *Ratio Game*. Leistung eines Systems wurde in Transaktionen pro Sekunde (Durchsatz) gemessen:

System	Workload 1	Workload 2
A	20	10
B	10	20

Drei Möglichkeiten, die Systeme zu vergleichen:

Sys	Workload 1	Workload 2	∅
A	20	10	15
B	10	20	15

Durchschnittlicher Durchsatz

Sys	Workload 1	Workload 2	∅
A	200%	50%	125%
B	100%	100%	100%

Durchsatz relativ zu System B

Sys	Workload 1	Workload 2	∅
A	100%	100%	100%
B	50%	200%	125%

Durchsatz relativ zu System A

# Performance-Analyse ist...

## **... ein Handwerk**

Erfordert viel theoretisches und praktisches Wissen und systematische Vorgehensweisen

## **... eine Kunst**

Viele Experten haben eigene Methodologien und Herangehensweisen. Werden einige davon kennenlernen.

## **...eine Wissenschaft**

Wissenschaftler «entdecken» immer wieder Dinge über Performance, obwohl Systeme ja von Menschen gebaut sind

- Resultate werden an Konferenzen veröffentlicht, z. B. *SIGMETRICS*



# **Evaluationstechniken & Metriken**

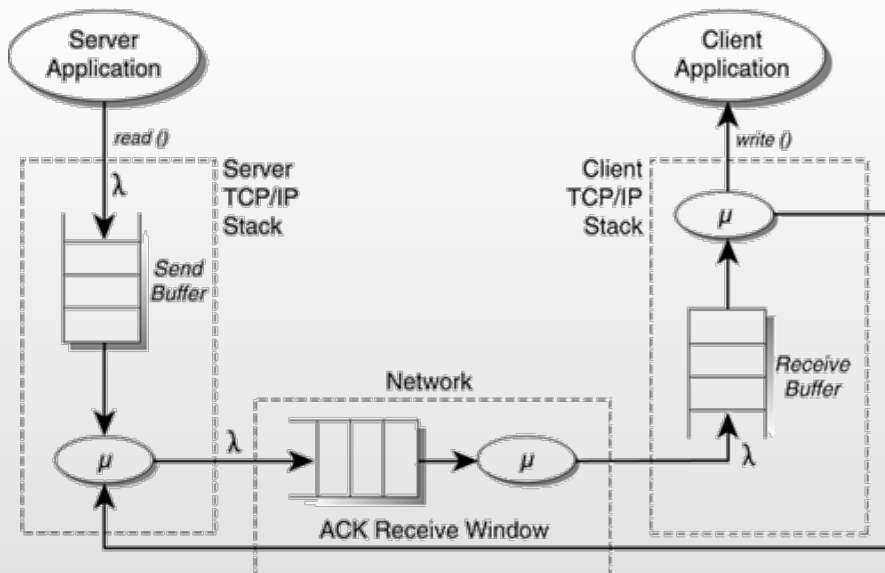
# Evaluationstechniken

Performance kann grundsätzlich auf drei Arten untersucht werden.

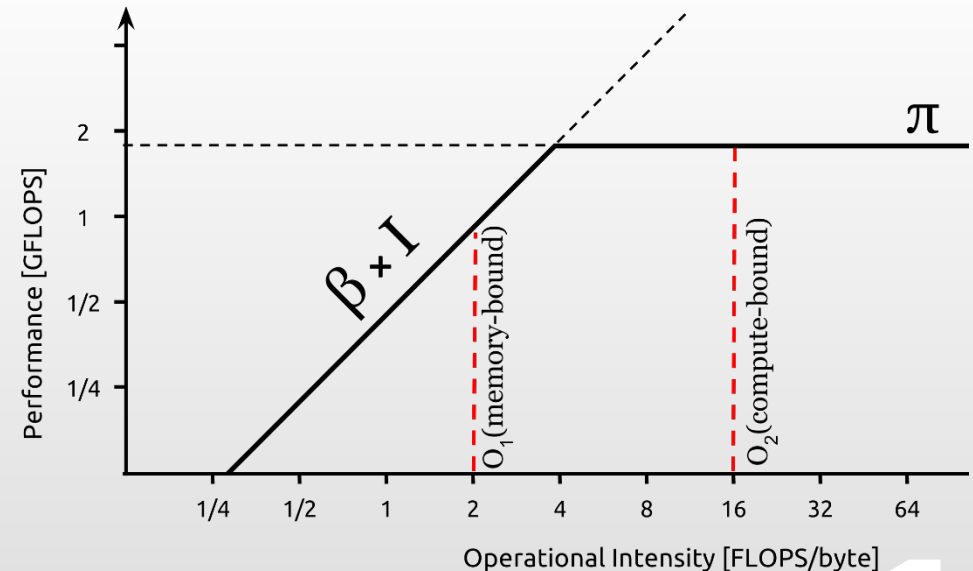
## 1. Analytische Modellierung

System wird vereinfacht, mathematisch beschrieben & untersucht.

**Beispiele:**



**Warteschlangentheorie**



**Roofline model**

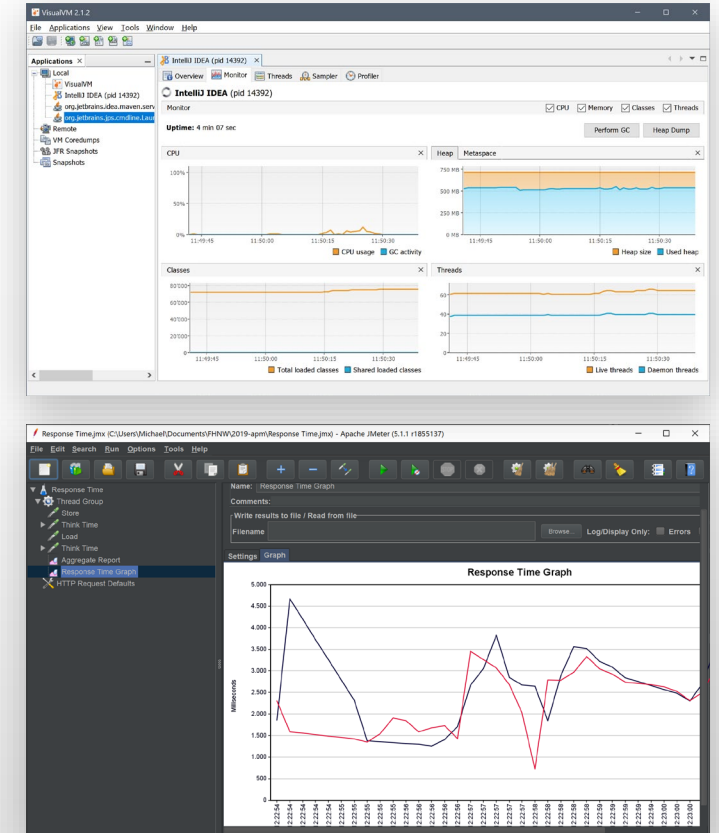
## 2. Simulation

Erlaubt mehr Details als analytische Modelle, aber liefert keine Zusammenhänge.

## 3. Messung

«The real thing»; setzt funktionierendes System/Prototyp voraus. Resultate sind immer *Realitäts-nah*, aber trotzdem *nicht unbedingt aussagekräftig*.

Fundamentales Problem: Man misst immer *in einer bestimmten Situation*.

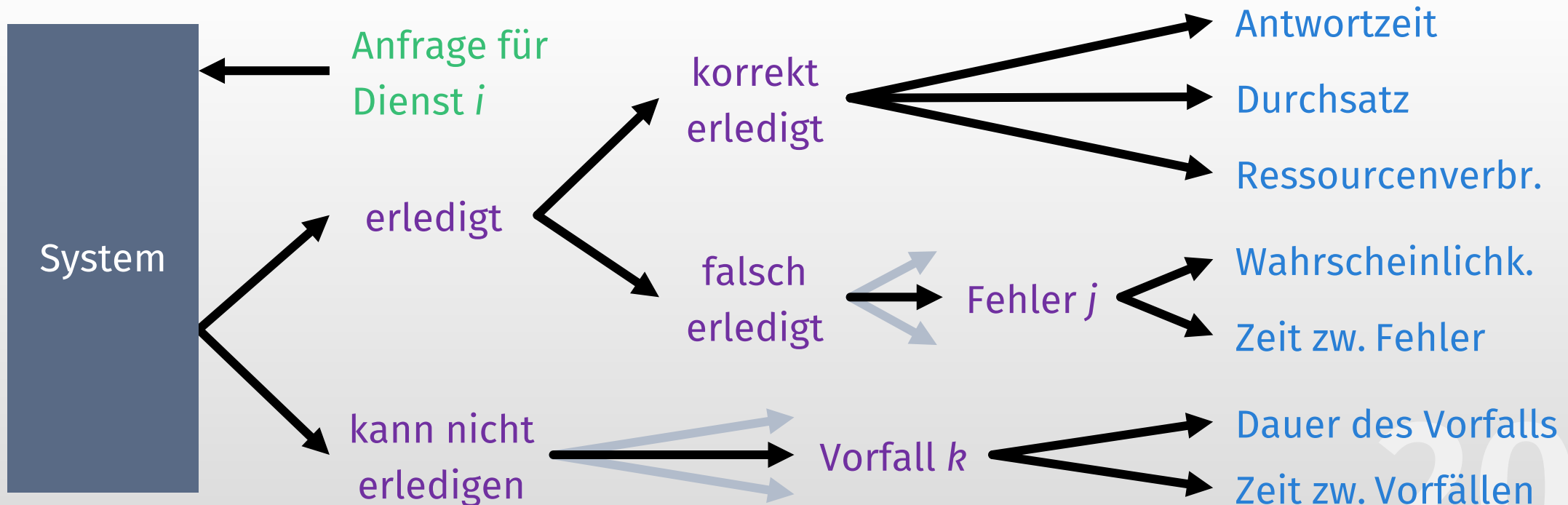


**In APM:** Vor allem Messung. Aber Modelle (oder Simulationen) sind nützlich, um Resultate einzuordnen/zu validieren.

# Performance-Metriken

Um Performance zu *quantifizieren* (messen, vergleichen, usw.) braucht es *Metriken*. Auswahl von Metriken hat grossen Einfluss auf Resultate.

**Grundlage:** Grobmodell eines *Systems*. System hat mehrere *Dienste*, welche *Anfragen* beantworten. Verschiedene *Ergebnisse* möglich:



Beispiele für Systeme und Dienste:

## **Web-Applikation**

Dienste könnten Features der App sein, z. B. Erstellen eines neuen Posts, Suchen nach Posts, usw.

## **REST-Service**

Dienste sind die einzelnen Endpunkte

## **Datenbanksystem**

Dienste könnten verschiedene Arten von Anfragen sein

## **CPU**

Mögliche Definitionen von Diensten:

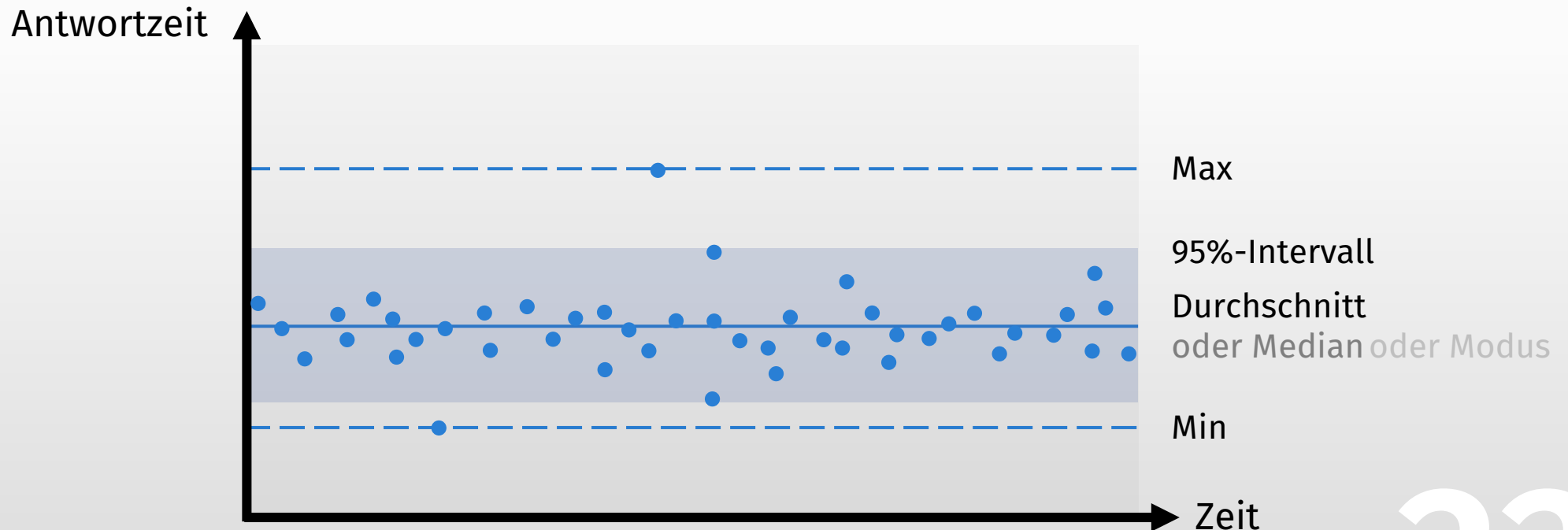
- Ausführen von einzelnen Instruktionen
- Ausführen von verschiedenen «*Kernels*»
- Ausführen einer kompletten Applikation

Für jeden Dienst  $i$ , für jeden Fehler  $j$  und für jeden Vorfall  $k$  gibt es mehrere Metriken!

**Grobe Unterteilung:** *Geschwindigkeit, Zuverlässigkeit, Verfügbarkeit*

- Entsprechen den drei Arten von Ergebnissen

**Zudem:** Metrik ist nicht *eine* Zahl, sondern *Verteilung*.

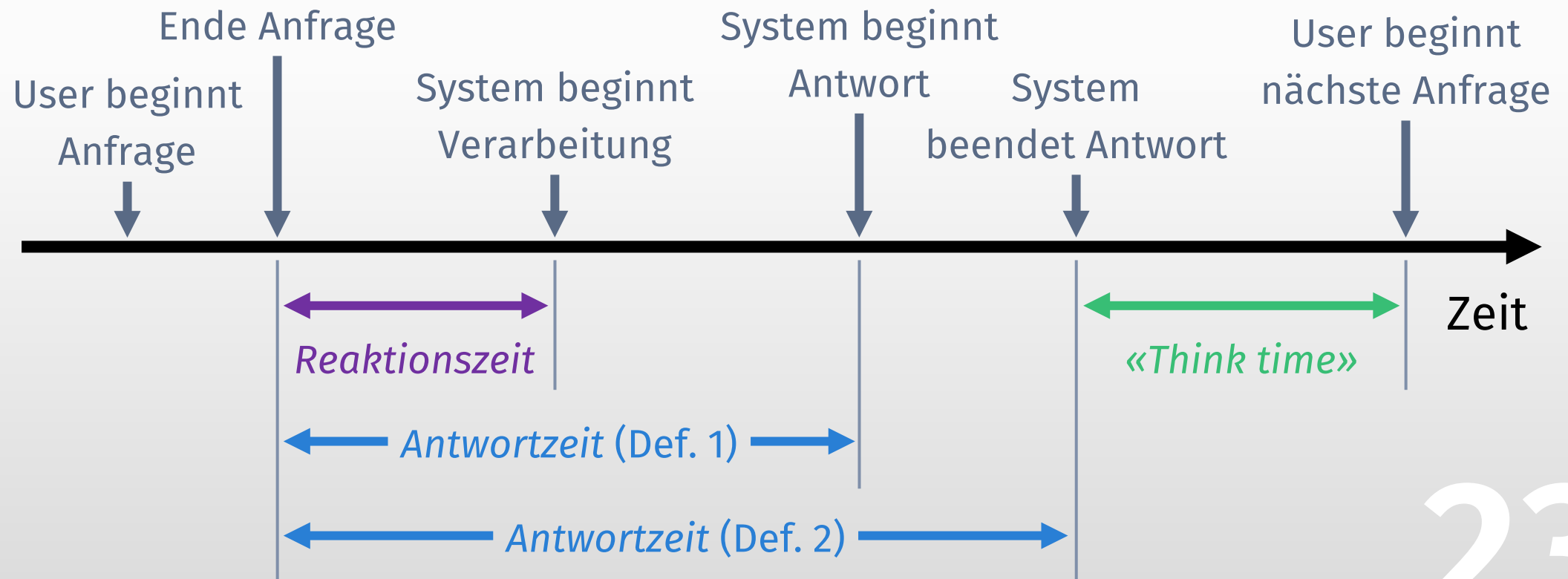


# Häufig verwendete Performance-Metriken

**Antwortzeit / Latenz** (response time / latency)

*Zeitintervall zwischen Anfrage und Antwort*

Vereinfachung! Anfrage und Antwort sind nicht ein einziger Zeitpunkt:



# Zeitmassstäbe in Computersystemen:

Event	Latency	Scaled
1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	3 ns	10 s
Level 3 cache access	10 ns	33 s
Main memory access (DRAM, from CPU)	100 ns	6 min
Solid-state disk I/O (flash memory)	10–100 µs	9–90 hours
Rotational disk I/O	1–10 ms	1–12 months
Internet: San Francisco to New York	40 ms	4 years
Internet: San Francisco to United Kingdom	81 ms	8 years
Lightweight hardware virtualization boot	100 ms	11 years
Internet: San Francisco to Australia	183 ms	19 years
OS virtualization system boot	< 1 s	105 years
TCP timer-based retransmit	1–3 s	105–317 years
SCSI command time-out	30 s	3 millennia
Hardware (HW) virtualization system boot	40 s	4 millennia
Physical system reboot	5 m	32 millennia



Quelle: Gregg 2020



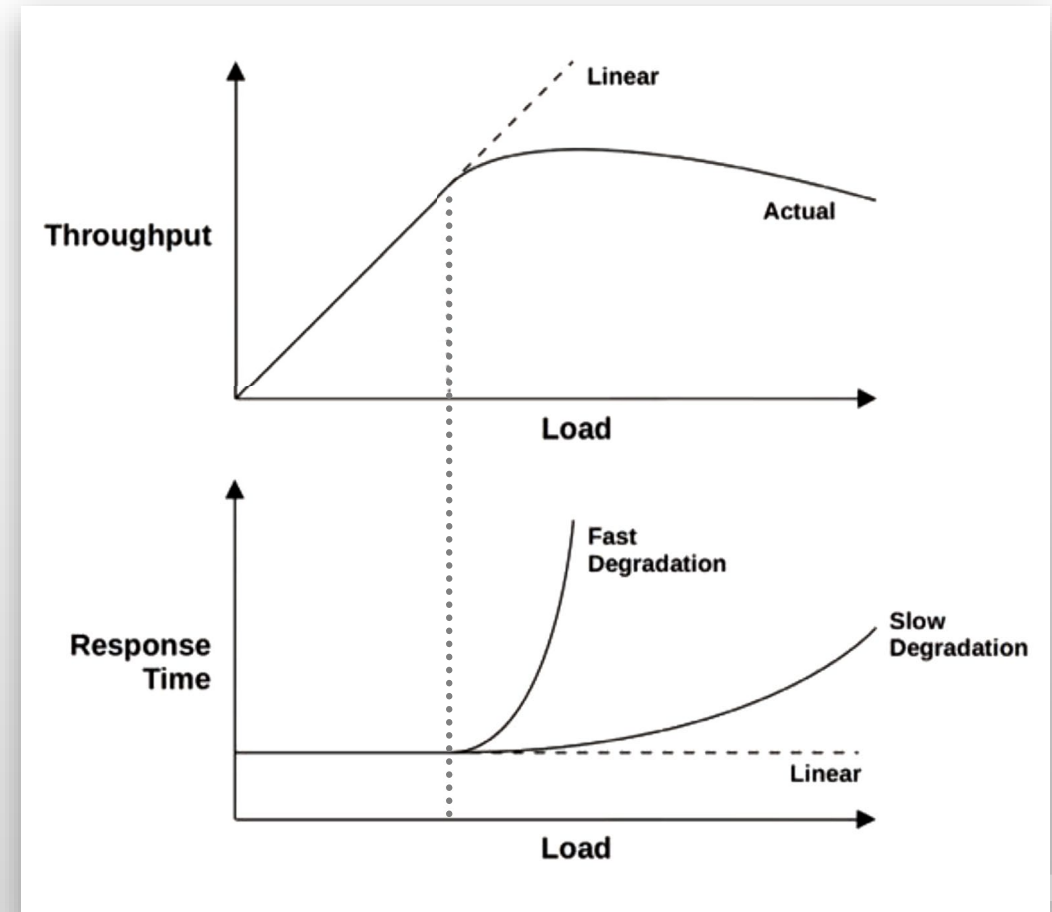
**Durchsatz** (throughput): *Erledigte Anfragen pro Zeiteinheit*

## Beispiele

- MIPS (*mio instructions per second*)
- GFLOPS
- Transaktionen pro Sekunde

Durchsatz wird normalerweise *grösser, je höher die Last*. Aber nur, solange System noch nicht voll ausgelastet ist!

Zusammenhang mit Antwortzeit:



Quelle: Gregg 2020

Maximaler Durchsatz unter idealen Bedingungen entspricht *nominaler Kapazität* eines Systems

- Wird oft erst erreicht, wenn Antwortzeit inakzeptabel gross ist

Interessanter ist oft *brauchbare Kapazität*: maximal möglicher Durchsatz *ohne vordefinierte Antwortzeit zu überschreiten*.

**Effizienz:** *Verhältnis zwischen brauchbarer und nominaler Kapazität*

### **Beispiel LAN**

- nominale Kapazität (*Bandbreite*): 1 Gbit/s
- brauchbare (erreichbare) Kapazität: 800 Mbit/s
- Effizienz: 80%

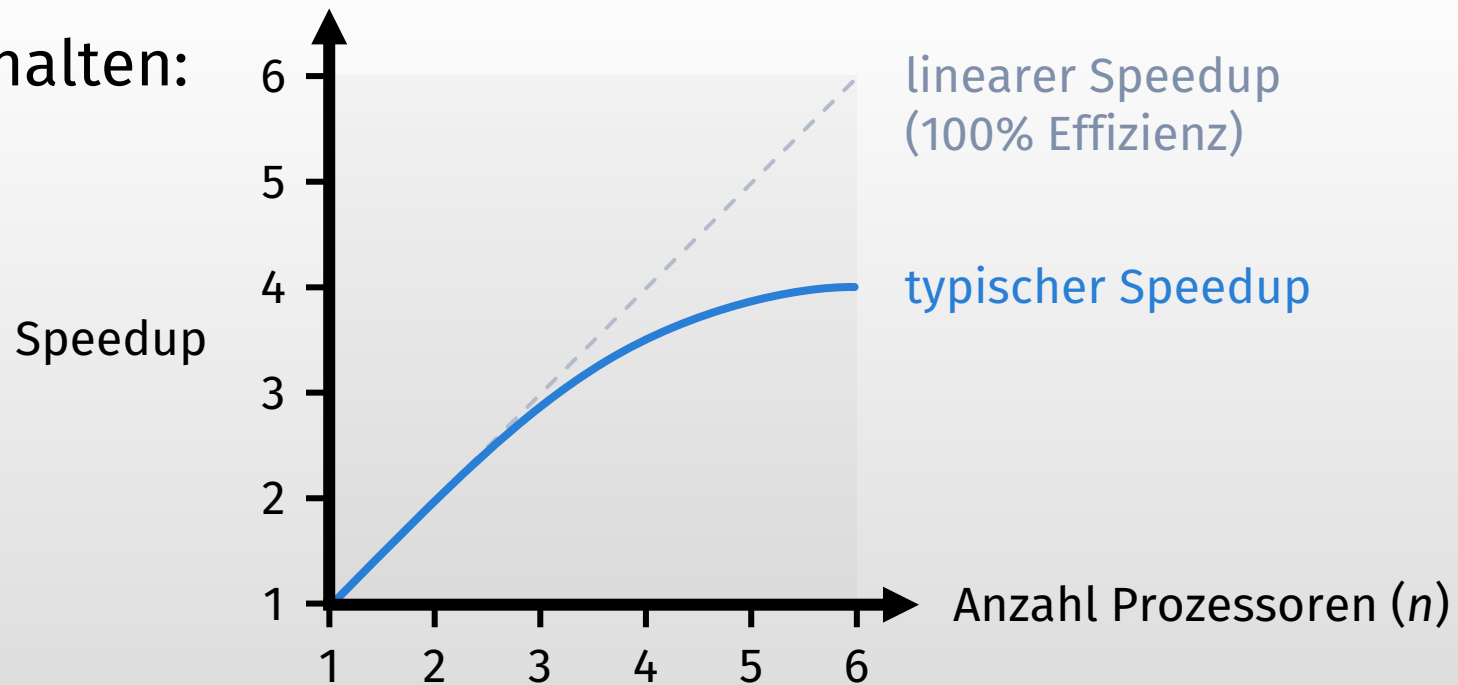
## Speedup

Spezifisch für parallele Verarbeitung: *Verhältnis von Durchsatz mit  $n$  Prozessoren zu Durchsatz mit 1 Prozessor.*

Weiteres Beispiel für Effizienz:

$$\text{Effizienz} = \frac{\text{Speedup}}{n}$$

Typisches Verhalten:



Zu folgenden Metriken mehr in späteren Wochen:

**Auslastung** (utilization)

*Verhältnis zwischen Zeit, in der Ressource verwendet wird, und Gesamtzeit (...)*

**Sättigung** (saturation)

*Ausmass von «Überbelastung», d. h. Arbeit, die wegen Vollauslastung nicht sofort erledigt werden kann*

**Zuverlässigkeit** (reliability)

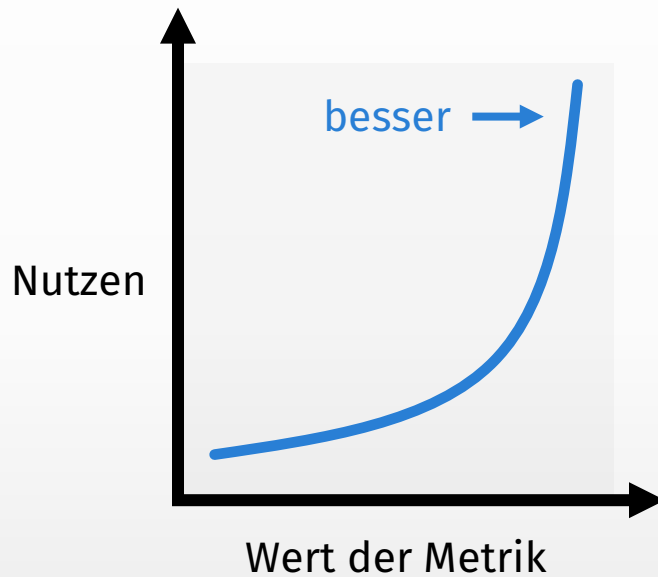
*Wahrscheinlichkeit von fehlerhaften Antworten, oder durchschnittliche Zeit zwischen solchen*

**Verfügbarkeit** (availability)

*Anteil der Zeit, während der das System Anfragen beantwortet, z. B. 99%*

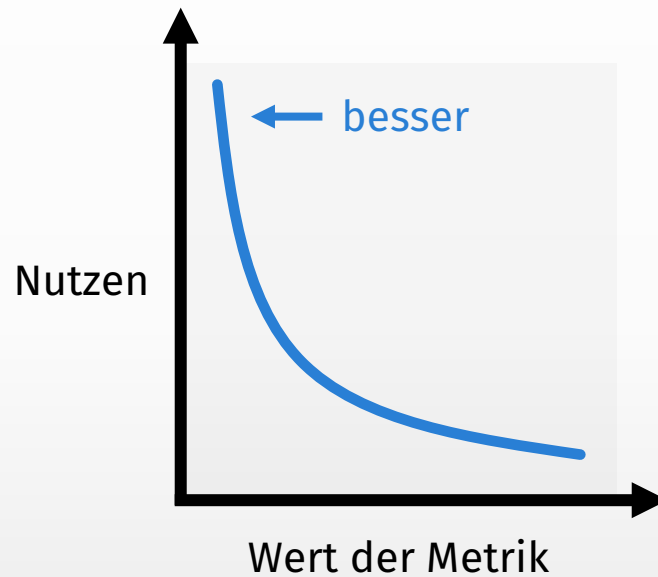
# Nutzenklassifikation von Metriken

Jede Metrik gehört zu einer von drei Klassen:



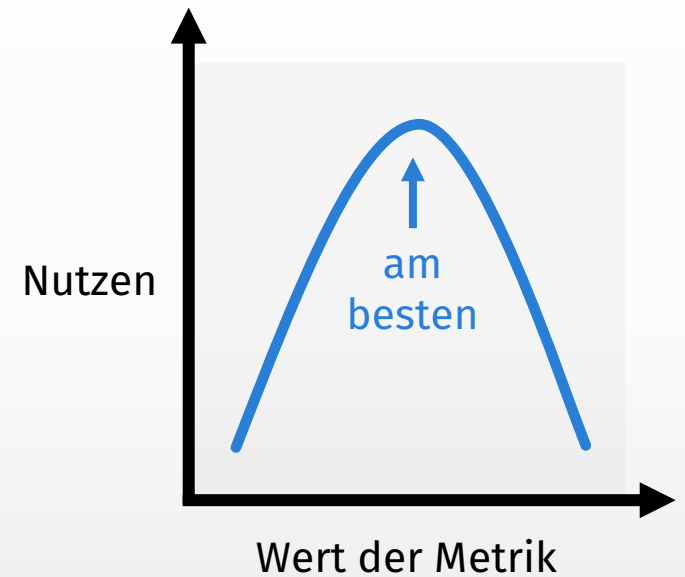
«höher ist besser»

Beispiel: Durchsatz



«tiefer ist besser»

Beispiel: Antwortzeit



«bestimmter

Wert ist am besten»

Beispiel: Auslastung

# Parallelisierung mit Java

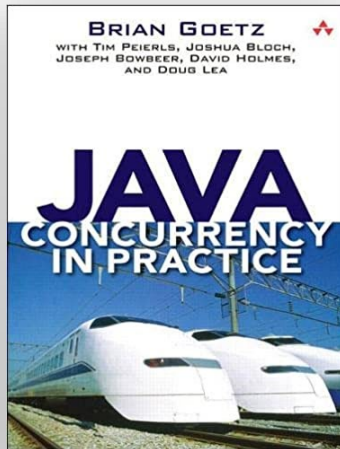
Ein Crashkurs

# Disclaimer



**Vorsicht:** Parallele Programme schreiben ist nicht etwas, das man in *einer Woche* lernt!

Falls Sie keine Erfahrung mit Concurrency und paralleler Programmierung haben, probieren Sie das nicht in «wichtigem» Code...



Weitere Ressourcen:

- Modul [Concurrent Programming](#)
- Buch [Java Concurrency in Practice](#) von Brian Goetz

# Die Thread-Klasse

Reihenfolge nicht  
immer gleich

```
public class HelloAndGoodbye {  
    public static void main(String[] args) {  
        Runnable hello = () -> {  
            for (int i = 1; i <= 1000; i++) {  
                System.out.println("Hello " + i);  
            }  
        };  
        Runnable goodbye = () -> {  
            for (int i = 1; i <= 1000; i++) {  
                System.out.println("Goodbye " + i);  
            }  
        };  
        var t1 = new Thread(hello);  
        var t2 = new Thread(goodbye);  
        t1.start();  
        t2.start();  
    }  
}
```

```
Goodbye 1  
Goodbye 2  
Hello 1  
Hello 2  
:  
Hello 32  
Hello 33  
Goodbye 3  
Goodbye 4  
:  
Goodbye 26  
Goodbye 27  
Hello 34  
Hello 35  
:  
Hello 68  
Hello 69  
Goodbye 28  
Goodbye 29  
Hello 70  
Hello 71  
Goodbye 30  
Goodbye 31
```



# Parallele Verarbeitung

Mehrere Threads können **gleichzeitig** Code ausführen, falls mehrere CPUs (bzw. CPU-Cores) vorhanden sind.

**Beispiel:** Zerlegen von Zahlen in Primfaktoren, z. B.  $60 = 2 \times 2 \times 3 \times 5$

```
var start = System.currentTimeMillis();  
  
for (var num : NUMBERS) {  
    var factors = factorize(num);  
    System.out.println(num + ": " + factors);  
}  
  
var time = (System.currentTimeMillis() - start) / 1000.0;  
System.out.printf("\n%.1f seconds\n", time);
```

eine Zahl nach  
der anderen...

```
...  
6198491414655: [3, 3, 5, 19, 61, 118847501]  
6689508176080: [2, 2, 2, 2, 5, 769, 108737129]
```

**10.6 seconds**

**Besser:** Aufgabe aufteilen. Jede Zahl kann *unabhängig von anderen* zerlegt werden. Mehrere Threads, jeder übernimmt ein paar Zahlen.

```
var threads = new ArrayList<Thread>();
var numsPerThread = NUMBERS.size() / NUM_THREADS;
for (var i = 0; i < NUMBERS.size(); i += numsPerThread) {
    var partition = NUMBERS.subList(i, i + numsPerThread);

    var t = new Thread(() -> {
        for (var num : partition) {
            var factors = factorize(num);
            System.out.println(num + ": " + factors);
        }
    });
    t.start();
    threads.add(t);
}

for (var t : threads) {
    t.join();
}
```

aufteilen...

und in mehreren  
Threads ausführen

andere  
Reihenfolge...

aber viel  
schneller!

```
...
2747561899166: [2, 7, 1667, 117729107]
8653241755480: [2, 2, 2, 5, 7, 7, 43, 102672541]
```

**1.5 seconds**

Muss das so kompliziert sein?

**Nein.**

```
NUMBERS.parallelStream().forEach(num -> {  
    var factors = factorize(num);  
    System.out.println(num + ": " + factors);  
});
```

```
...  
2717339902199: [13, 1873, 111599651]  
6428733687135: [3, 3, 5, 1451, 98456753]
```

**1.5 seconds**



`parallelStream()` teilt Daten in mehrere Unter-Streams auf, die unabhängig voneinander verarbeitet werden.

**Achtung:** Code in `forEach`-Lambda wird in verschiedenen Threads aufgerufen. Vorsicht bei Zugriff auf gemeinsame Variablen!

`parallelStream()` ist einfach, aber bietet keine Kontrolle über Performance.

**Alternative:** *Thread-Pool*, mittels `Executor`-Interface

```
var executor = Executors.newFixedThreadPool(NUM_THREADS);  
for (var num : NUMBERS) {  
    executor.submit(() -> {  
        var factors = factorize(num);  
        System.out.println(num + ": " + factors);  
    });  
}
```

Warten bis alle Tasks abgearbeitet wurden (einfachster Weg):

```
executor.shutdown();  
executor.awaitTermination(1, TimeUnit.HOURS);
```

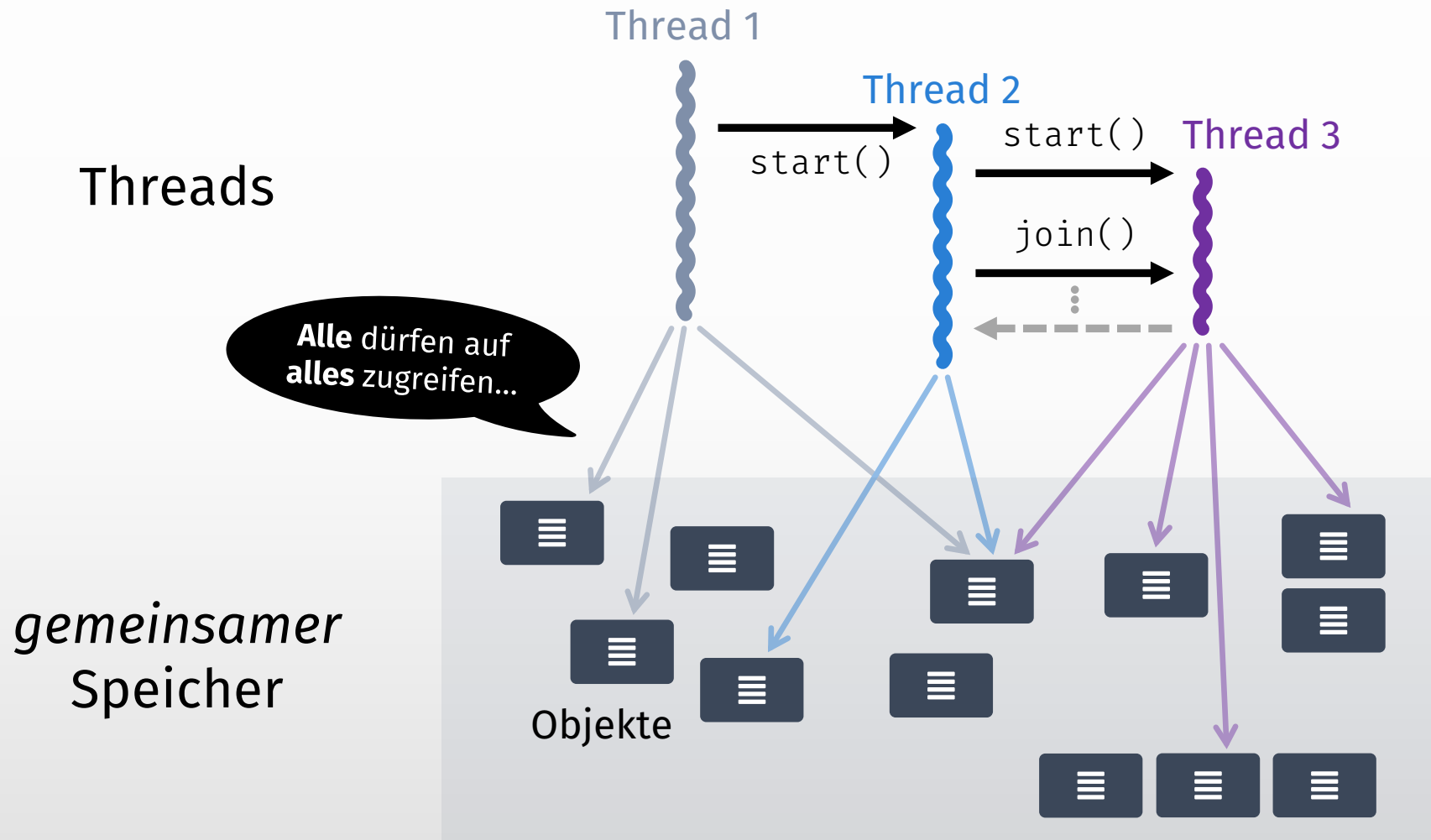
**Fragen?**



# Anhang

Probleme mit Multithreading

# Multithreading-Modell



# Problem 1: Sichtbarkeit

gemeinsamer Zustand:  
**boolean**-Attribut

```
private boolean done = false;

private void run() throws InterruptedException {
    var reader = new Thread(() -> {
        while (!done) {
            // nothing to do here...
        }
        System.out.println("reader is done");
    });
    reader.start();

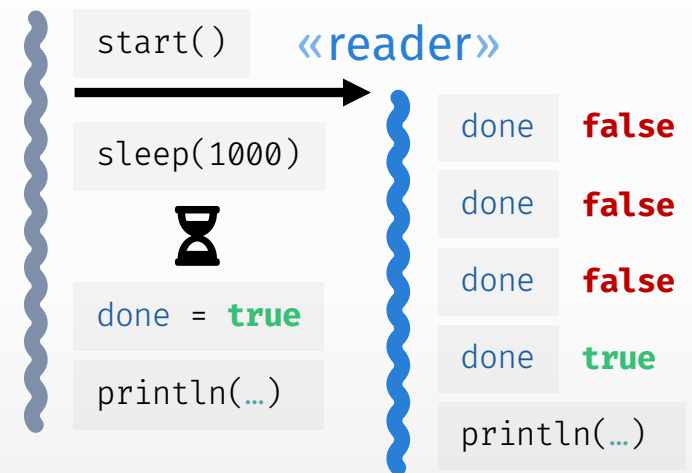
    // wait a little...
    Thread.sleep(1000);

    done = true;
    System.out.println("writer is done");
}
```

pausiert aktuellen  
Thread (ms)

Erwartetes Verhalten:

«writer»



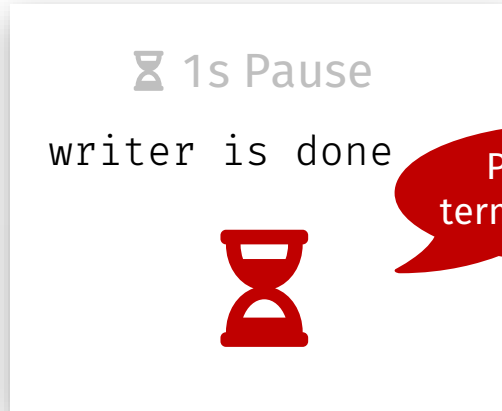
⌚ 1s Pause

writer is done  
reader is done

(oder umgekehrt)



## Beobachtetes Verhalten:



```
private boolean done = false;

private void run() throws ... {
    var reader = new Thread(() -> {
        while (!done) {}
        ...println("reader is done");
    });
    reader.start();

    Thread.sleep(1000);
    done = true;
    ...println("writer is done");
}
```

Wie ist so etwas möglich?

**Antwort:** *Java gibt grundsätzlich keine Garantie, dass Änderungen von einem Thread in anderen Threads jemals sichtbar werden.*



## Noch schlimmer: Kleine Änderung am Programm ändert das Verhalten.

```
private boolean done = false;

private void run() throws InterruptedException {
    var reader = new Thread(() -> {
        while (!done) {
            System.out.println("still waiting");
        }
        System.out.println("reader is done");
    });
    reader.start();

    // wait a little...
    Thread.sleep(1000);

    done = true;
    System.out.println("writer is done");
}
```

```
still waiting
still waiting
⋮
still waiting
still waiting
writer is done
reader is done
```

Jetzt terminiert  
das Programm  
(bei mir)

# Sichtbarkeit: Garantien

Java gibt einige *spezifische* Garantien über Sichtbarkeit:

1. Der Wert in einer **final**-Variable ist sichtbar, nachdem das Objekt erstellt wurde.
2. Der Wert in einer **static**-Variable ist sichtbar, nachdem der **static**-Block der Klasse ausgeführt wurde.
3. Änderung vor dem Freigeben eines *Locks* ist sichtbar für alle Threads, die das *Lock* danach erwerben.
4. Änderungen an einer **volatile**-Variable sind *immer sichtbar*.



Programm verhält sich **garantiert** korrekt, wenn wir gemeinsame Variable als **volatile** deklarieren, ...

```
private volatile boolean done = false;

private void run() throws InterruptedException {
    var reader = new Thread(() -> {
        while (!done) {}
        System.out.println("reader is done");
    });
    reader.start();

    Thread.sleep(1000);
    done = true;
    System.out.println("writer is done");
}
```

⌚ 1s Pause

writer is done  
reader is done



... das ist aber nicht unbedingt empfohlen (siehe später).

# Problem 2: Reihenfolge von Operationen

```
private volatile int value = 0;

private void run() {
    Runnable incrementer = () -> {
        for (int i = 0; i < 10000; i++) {
            value++;
        }
    };

    var t1 = new Thread(incrementer);
    var t2 = new Thread(incrementer);
    t1.start();
    t2.start();

    t1.join();
    t2.join();
    System.out.println(value);
}
```

Erwartetes Verhalten:

20000

Beobachtetes Verhalten:

14868

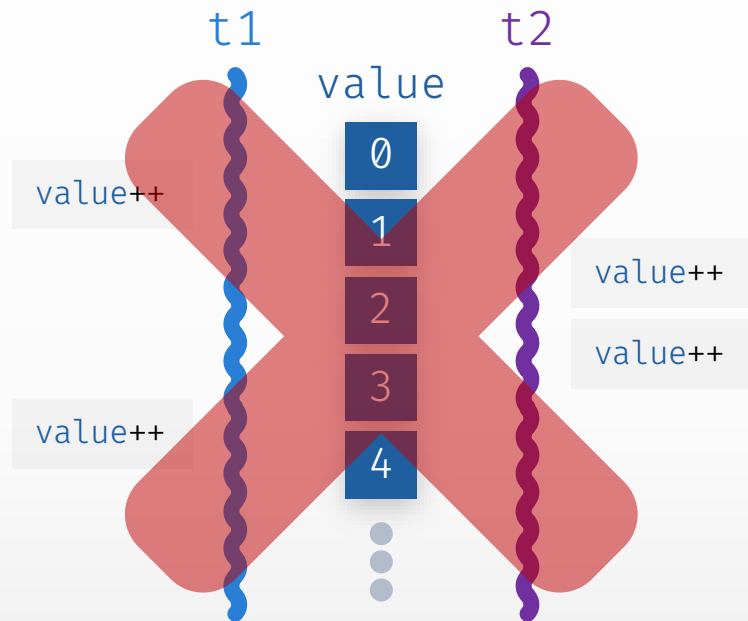
oder 16864

oder 17221

oder 12943



Wie ist jetzt das möglich? Haben doch extra **volatile** verwendet!



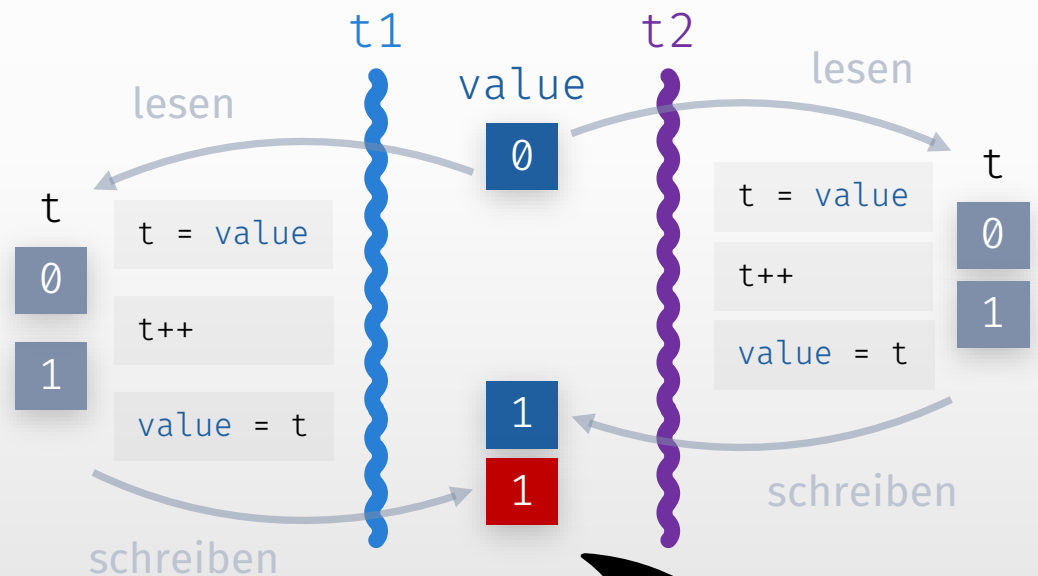
Sollte doch gehen,  
Reihenfolge ist egal!

## Problem:

`value++;` ist in Wirklichkeit:

temporäre  
Kopie

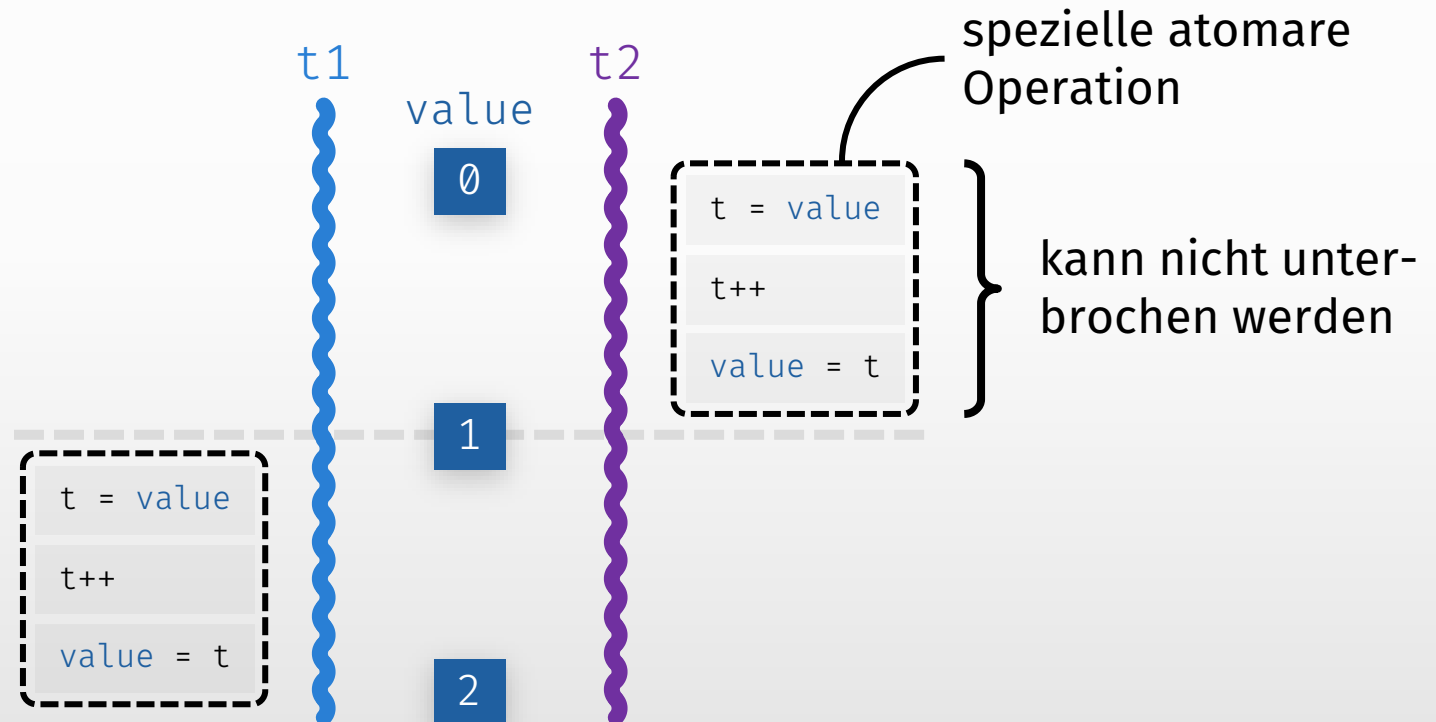
```
t = value;  
t++;  
value = t;
```



eine Änderung  
geht verloren

# Atomare Operationen

Um Problem zu lösen, müssen wir Überlappung von Operationen verhindern. Eine Möglichkeit: **atomare Operationen**.



## Beispiel: AtomicInteger-Klasse aus `java.util.concurrent`

!

Behälter für einen **int**-Wert

```
private final AtomicInteger value =  
    new AtomicInteger(0);  
  
private void run() throws InterruptedException {  
    Runnable incrementer = () -> {  
        for (int i = 0; i < 10000; i++) {  
            value.incrementAndGet();  
        }  
    };  
  
    var first = new Thread(incrementer);  
    var second = new Thread(incrementer);  
    first.start(); second.start();  
    first.join(); second.join();  
    System.out.println(value.get());  
}
```

atomare Operation

Beobachtetes Verhalten:

20000



immer, garantiert!



# 3. Problem: Deadlock

Alternative zu atomarer Operation (und **volatile**): **Lock**

```
private int value = 0;

private void run() throws InterruptedException {
    Runnable incrementer = () -> {
        for (int i = 0; i < 10000; i++) {
            synchronized (this) {
                value += 1;
            }
        }
    };

    var first = new Thread(incrementer);
    var second = new Thread(incrementer);
    first.start(); second.start();
    first.join(); second.join();
    System.out.println(value);
}
```

Beobachtetes Verhalten:

20000



**Vorteil von Locks:** beliebiger Codeblock wird «atomar» ausgeführt

- Immer nur ein Thread kann Code in **synchronized**-Block ausführen, andere warten

**Problem:** wenn *mehrere Locks* verwendet werden, kann es sein, dass zwei Threads jeweils auf das andere warten → **Deadlock**

```
synchronized (objektA) {  
    synchronized (objektB) {  
        ...  
    }  
}
```

**Thread 1**

```
synchronized (objektB) {  
    synchronized (objektA) {  
        ...  
    }  
}
```

**Thread 2**

Passiert *sehr selten*, aber wenn es passiert, dann wahrscheinlich zum ungünstigsten Zeitpunkt...