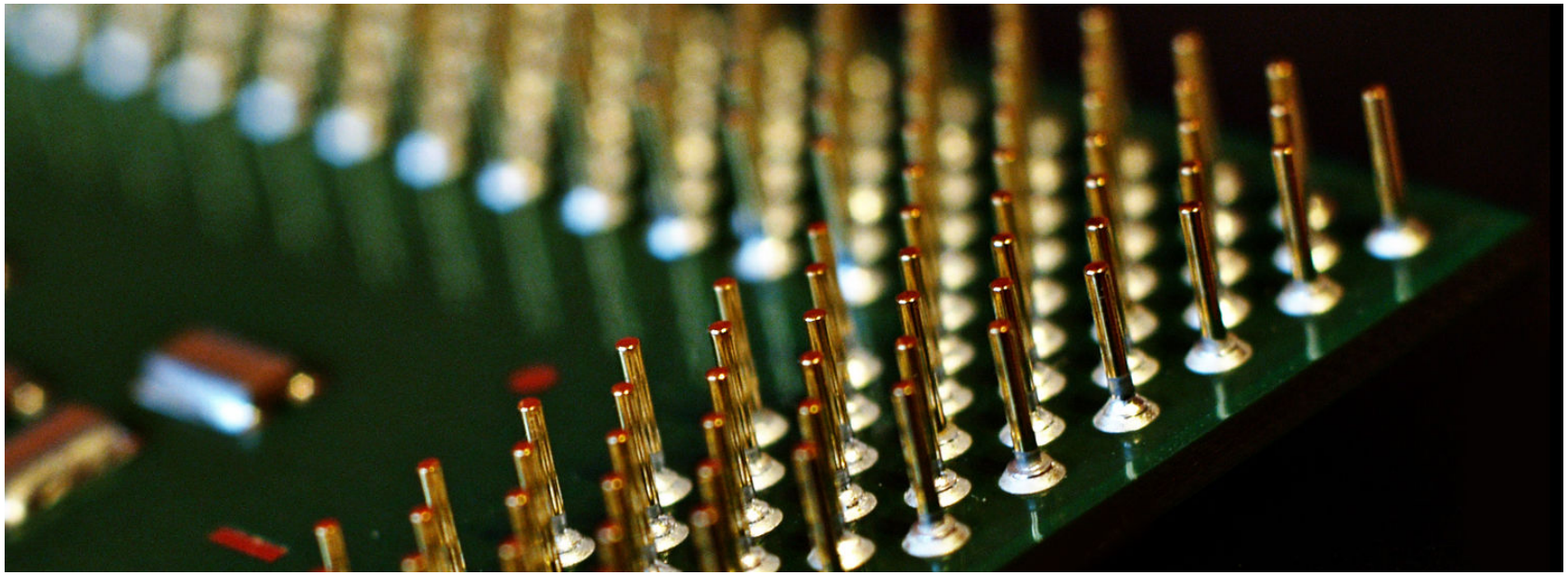
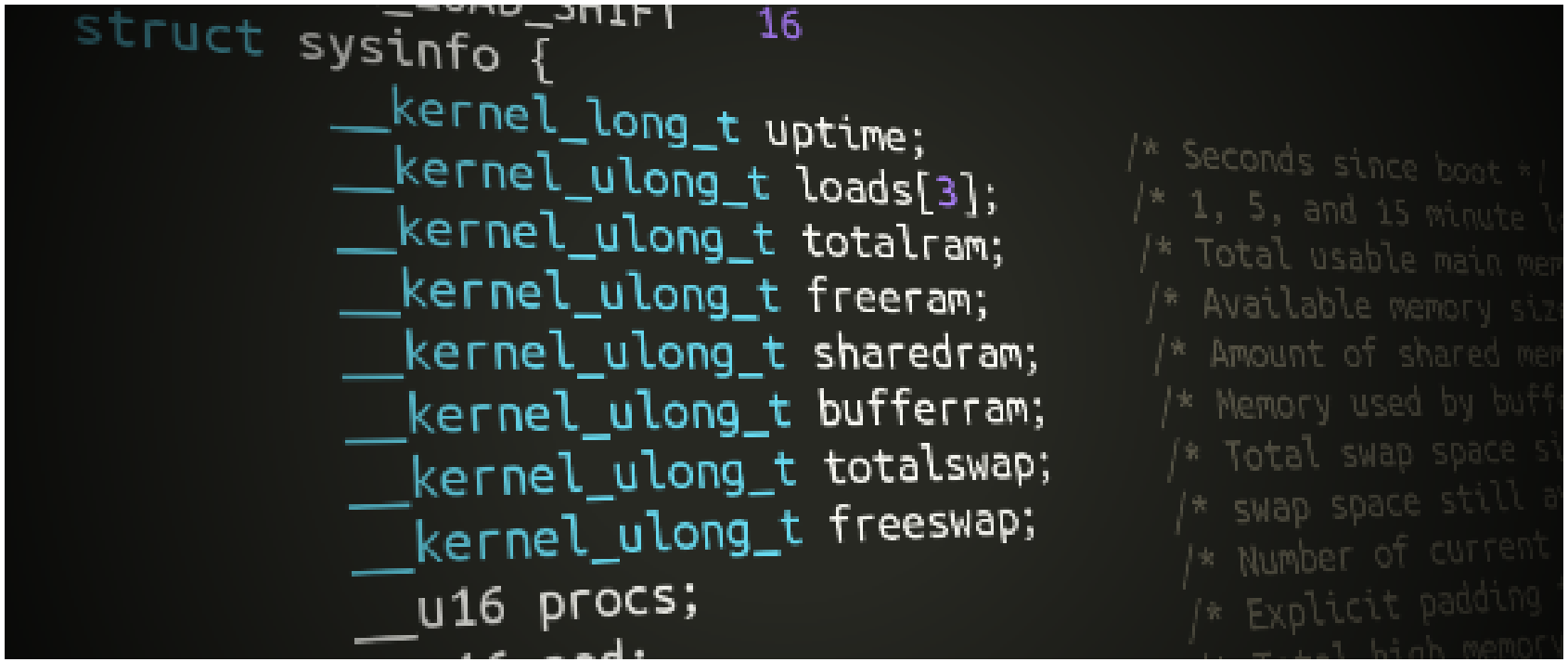


Parallelles Programmieren

Dr. Stefan Schlott



Formalitäten

About me...

Mail: stefan@ploing.de

...bitte Mails mit aussagekräftigem Betreff

Beruflich: BeOne Stuttgart GmbH

Steckenpferde: Moderne Sprachen (Scala, Kotlin, Rust, ...), Parallelisierung und Skalierbarkeit, Security, Privacy

Big Picture: Vorlesungsinhalt

Warum ist Parallelisierung nötig?

Wie sorge ich für parallele Ausführung?

Was gibt es da an Problemen?

Alternativen zu den Schmerzen der Low-Level-Parallelisierung

Termine

- 30.09. - Einführung, Prozesse, IPC
- 07.10. - Threads, Sperrkonzepte
- 14.10. - Parallelisierungs-Ansätze, Threadpools (Raum 136)
- 21.10. - Futures, Reactive Streams, Aktoren
- 28.10. - Aktoren, weitere Konzepte
- 18.11. Klausur (Raum 135)

Übungsaufgaben

Übungen während der Vorlesung

Übungen in Java (bzw auf der JVM). Benötigtes Arbeitsgerät:

- OpenJDK 21
- Geeignete IDE (Empfehlung: IntellIJ IDEA)

Dringend empfohlen:

Aufgaben aus der Vorlesung fertigprogrammieren

Besprechung zu Beginn der folgenden Stunde

(ggfs. Präsentieren der Lösung)

Tweet an Sie

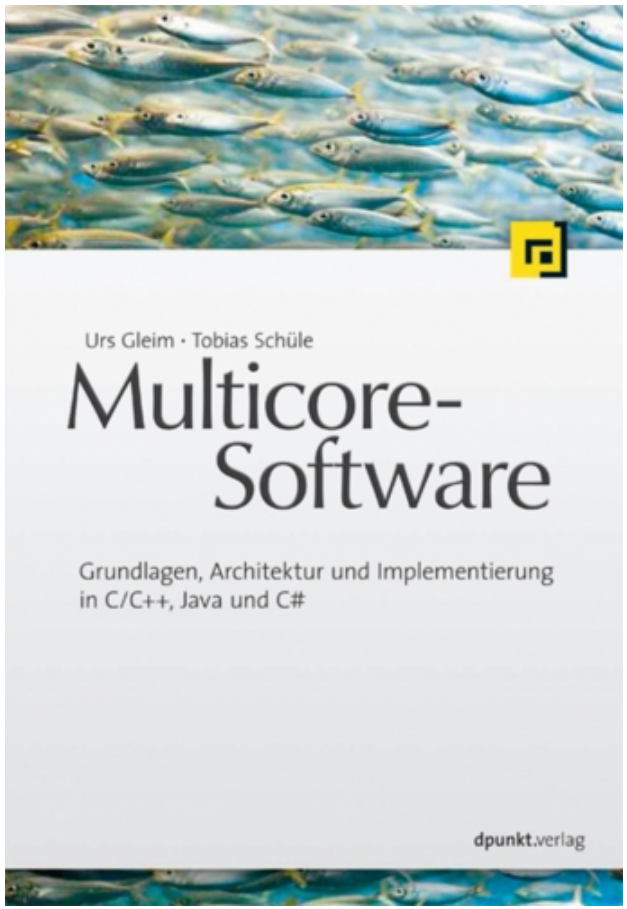
Jede Woche bis Sonntag, 18 Uhr: Mail an stefan@ploing.de

Betreff: ParProg-Spickzettel

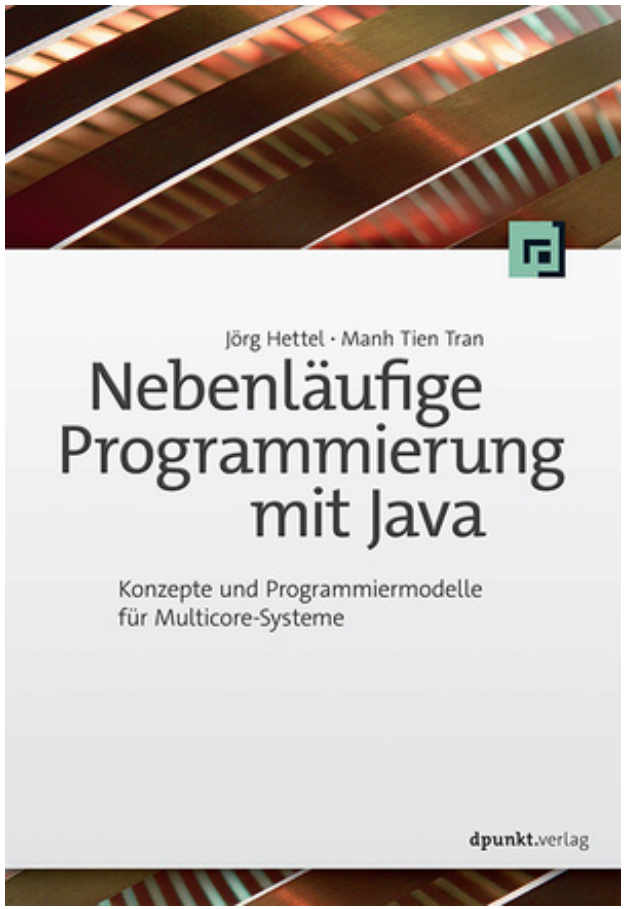
Mailtext: Matrikelnummer + ":" + bis zu 160 Zeichen Text (in einer Zeile)

...erhalten Sie gesammelt als „persönlicher Spickzettel“ für die Klausur

Literatur

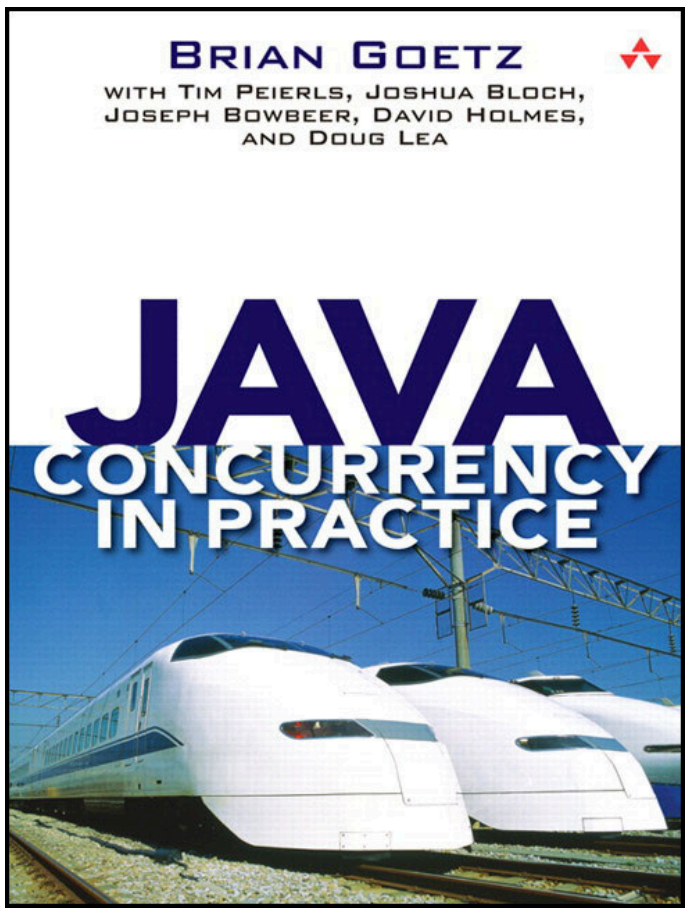


U. Gleim: Multicore-Software

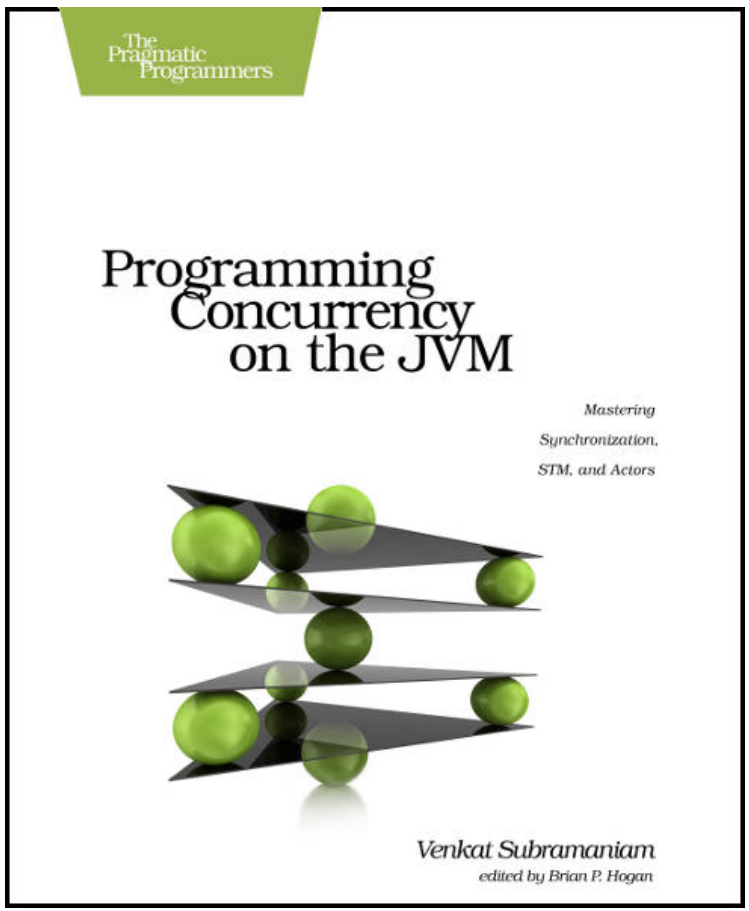


J. Hettel: Nebenläufige Programmierung mit Java

Literatur

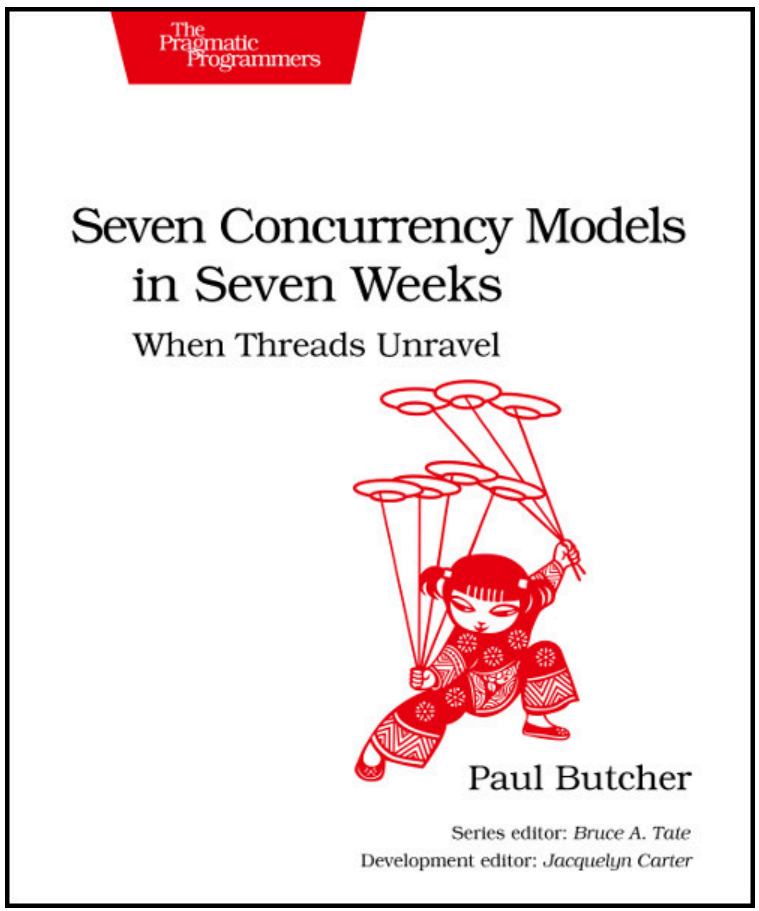


B. Goetz: Java concurrency in practice

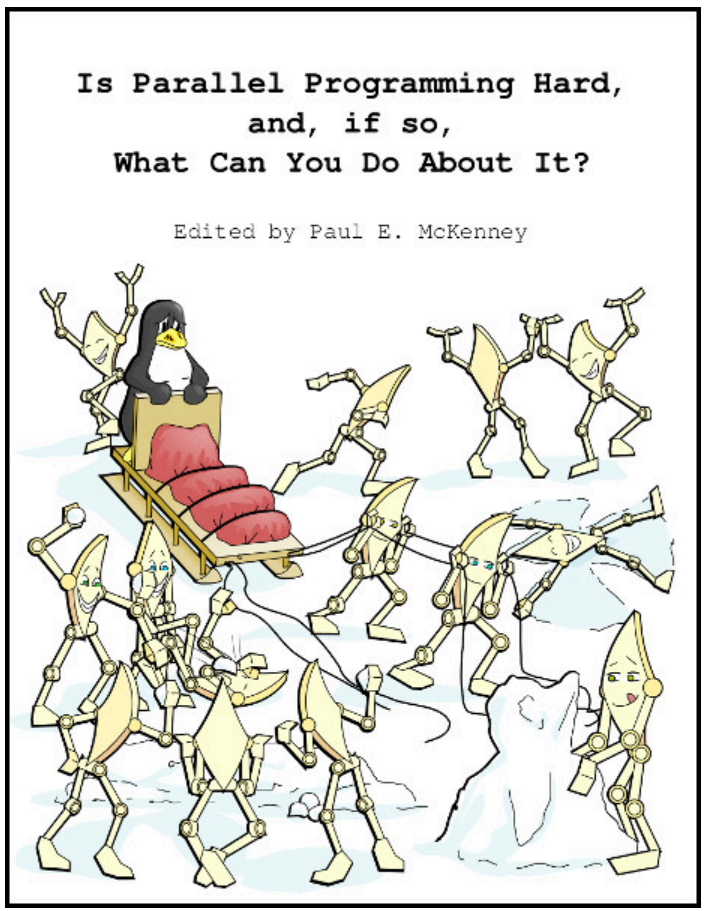


V. Subramaniam: Programming Concurrency on the JVM

Literatur



P. Butcher: 7 concurrency models in 7 weeks



P. McKenney: Is Parallel Programming Hard

Survival Guide

∄ Script

Folien

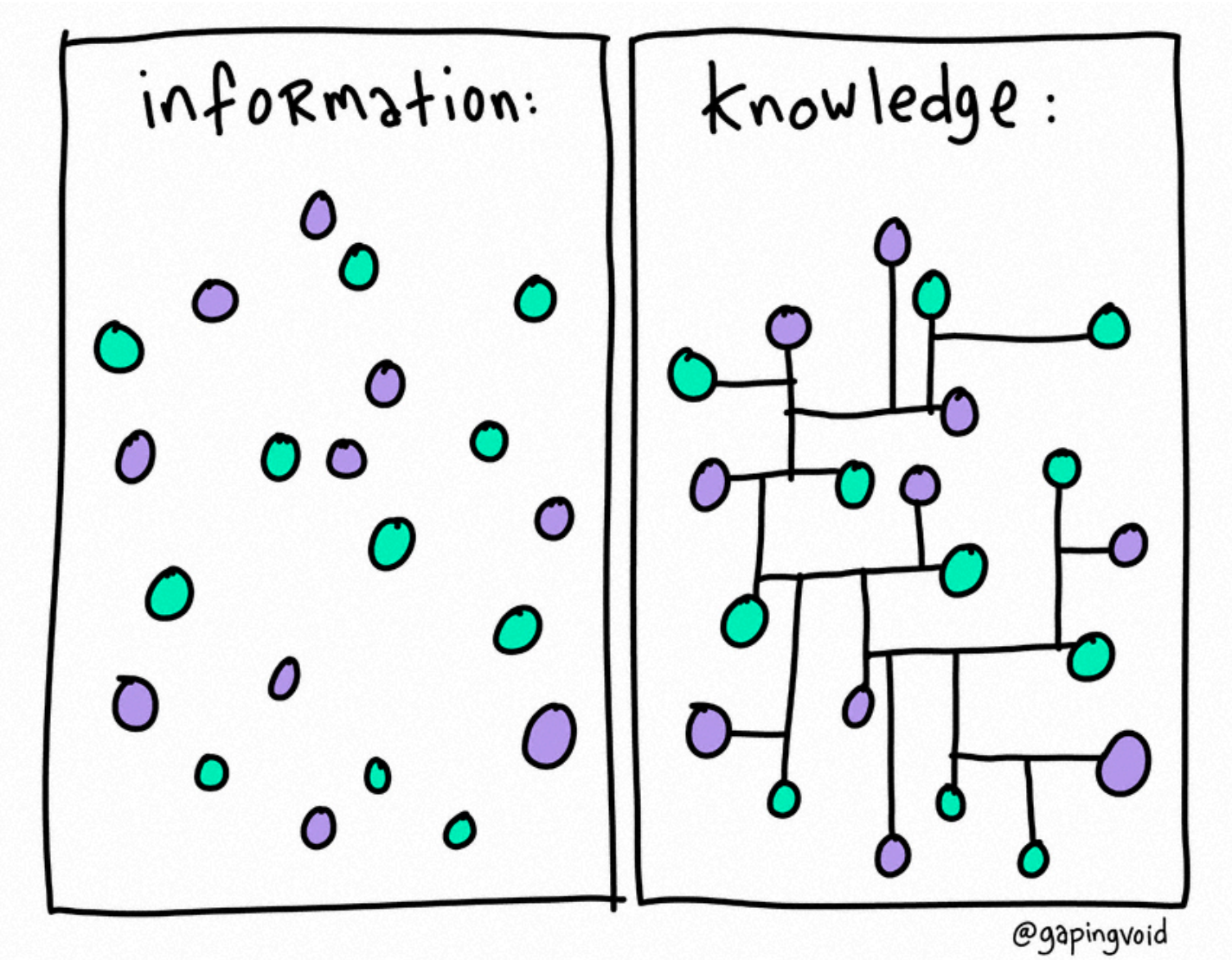
≠

Script

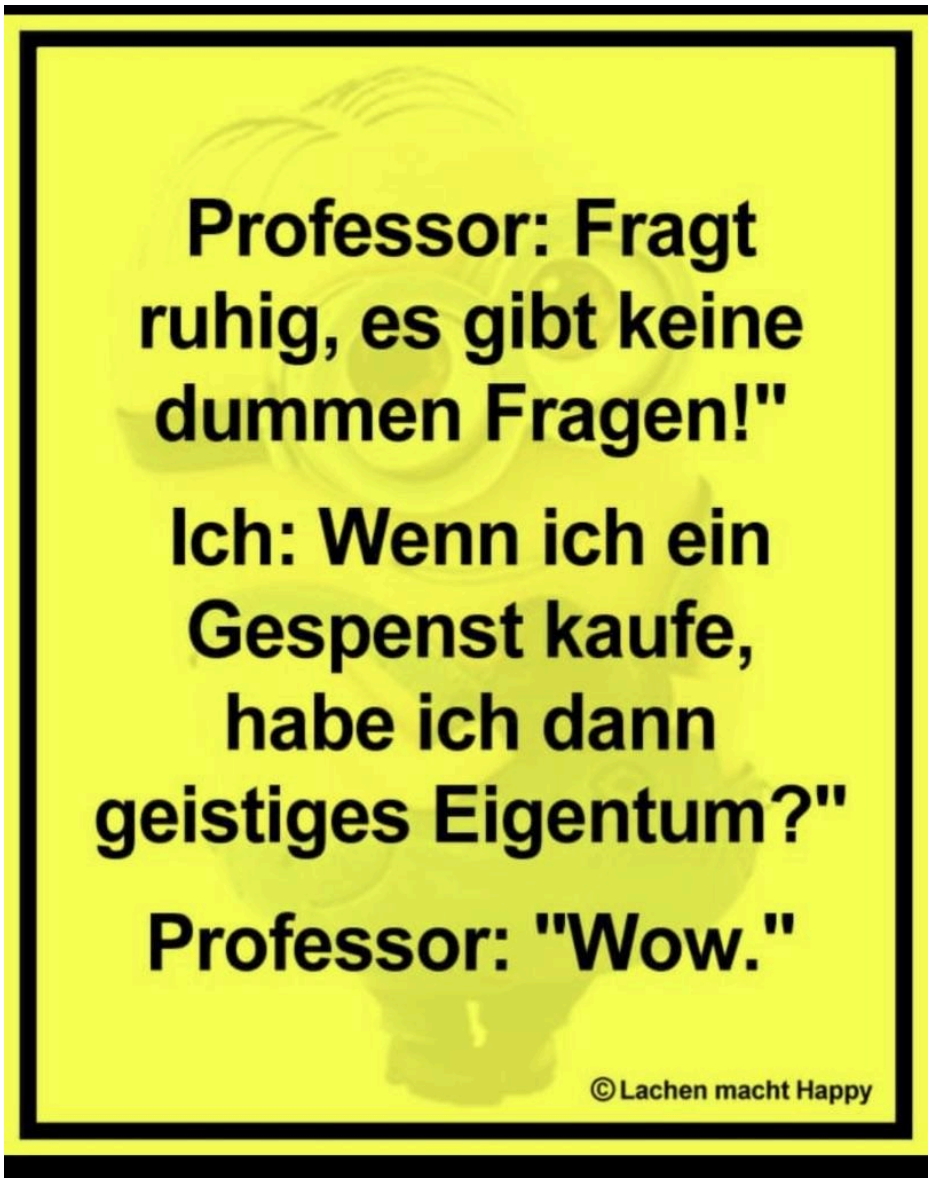
Aktive Mitarbeit!

Notizen!

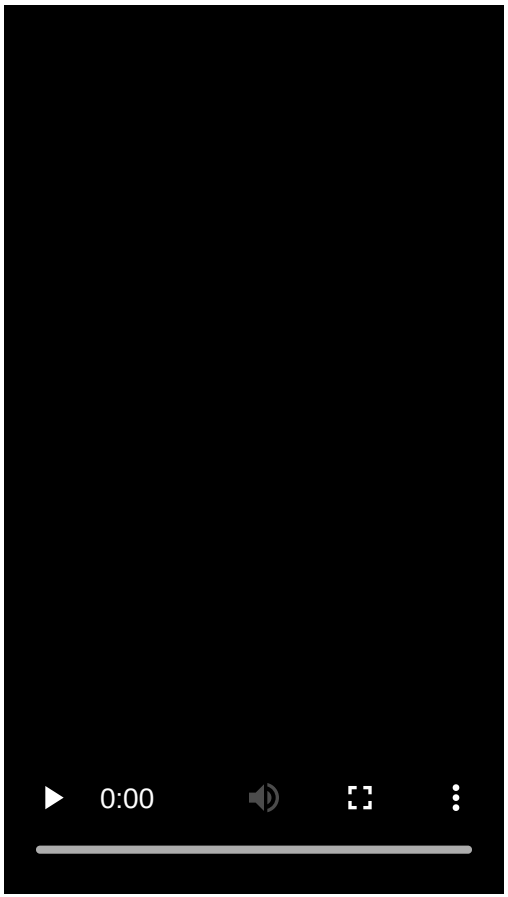
Sinn der Übungen



Fragen!


Professor: Fragt ruhig, es gibt keine dummen Fragen!"
Ich: Wenn ich ein Gespenst kaufe, habe ich dann geistiges Eigentum?"
Professor: "Wow."
© Lachen macht Happy

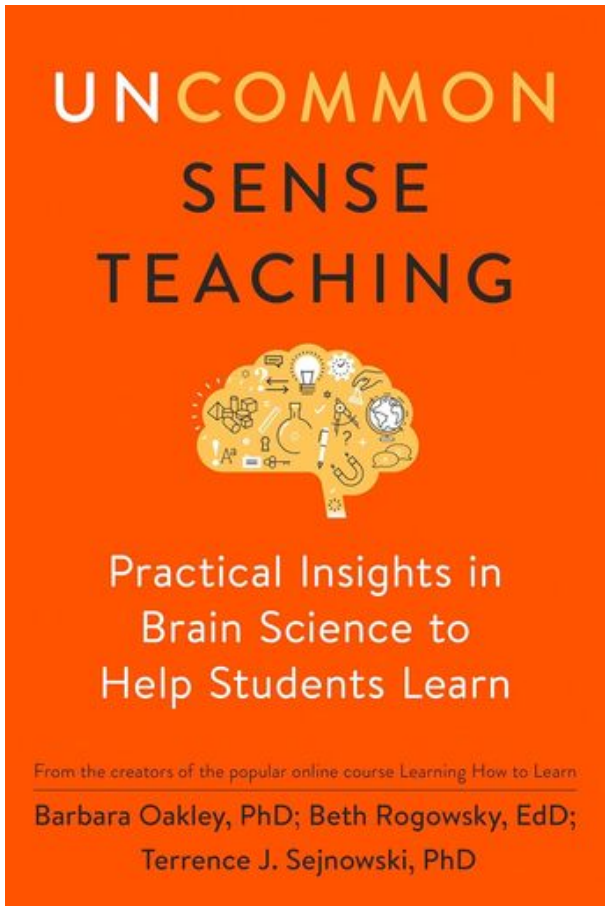
Übungen und Klausur...



Quelle: @AwardsDarwin

Richtig lernen

Make it stick



Uncommon sense teaching

Richtig lernen (billiger ;-))

Weinstein et al. Cognitive Research: Principles and Implications (2018) 3:2
DOI 10.1186/s41235-017-0087-y

Cognitive Research: Principles and Implications

TUTORIAL REVIEW Open Access

Teaching the science of learning

Yana Weinstein[1*], Christopher R. Madan[2,3] and Megan A. Sumeracki[4]

Abstract

The science of learning has made a considerable contribution to our understanding of effective teaching and learning strategies. However, few instructors outside of the field are privy to this research. In this tutorial review, we focus on six specific cognitive strategies that have received robust support from decades of research: spaced practice, interleaving, retrieval practice, elaboration, concrete examples, and dual coding. We describe the basic research behind each strategy and relevant applied research, present examples of existing and suggested implementation, and make recommendations for further research that would broaden the reach of these strategies.

Keywords: Education, Learning, Memory, Teaching

Significance

Education does not currently adhere to the medical model of evidence-based practice (Roediger, 2013). However, over the past few decades, our field has made significant advances in applying cognitive processes to education. From this work, specific recommendations can be made for students to maximize their learning efficiency (Dunlosky, Rawson, Marsh, Nathan, & Willingham, 2013; Roediger, Finn, & Weinstein, 2012). In particular, a review published 10 years ago identified a limited number of study techniques that have received solid evidence from multiple replications testing their effectiveness in and out of the classroom (Pashler et al., 2007). A recent textbook analysis (Pomerance, Greenberg, & Walsh, 2016) took the six key learning strategies from this report by Pashler and colleagues, and found that very few teacher-training textbooks cover any of these six principles – and none cover them all, suggesting that these strategies are not systematically making their way into the classroom. This is the case in spite of multiple recent academic (e.g., Dunlosky et al., 2013) and general audience (e.g., Dunlosky, 2013) publications about these strategies. In this tutorial review, we present the basic science behind each of these six key principles, along with more recent research on their effectiveness in live classrooms, and suggest ideas for pedagogical implementation. The target audience of this review is (a) educators who might be interested in integrating the strategies into their teaching practice, (b) science of learning researchers who are looking for open questions to help determine future research priorities, and (c) researchers in other subfields who are interested in the ways that principles from cognitive psychology have been applied to education.

While the typical teacher may not be exposed to this research during teacher training, a small cohort of teachers intensely interested in cognitive psychology has recently emerged. These teachers are mainly based in the UK, and, anecdotally (e.g., Dennis (2016), personal communication), appear to have taken an interest in the science of learning after reading *Make it Stick* (Brown, Roediger, & McDaniel, 2014; see Clark (2016) for an enthusiastic review of this book on a teacher's blog, and "Learning Scientists" (2016c) for a collection). In addition, a grassroots teacher movement has led to the creation of "researchED" – a series of conferences on evidence-based education (researchED, 2013). The teachers who form part of this network frequently discuss cognitive psychology techniques and their applications to education on social media (mainly Twitter; e.g., Fordham, 2016; Penfound, 2016) and on their blogs, such as Evidence Into Practice (https://evidenceintopractice.wordpress.com/), My Learning Journey (http://reflectionsofmyteaching.blogspot.com/), and The Effortful Educator (https://theeffortfuleducator.com/). In general, the teachers who write about these issues pay careful attention to the relevant literature, often citing some of the work described in this review.

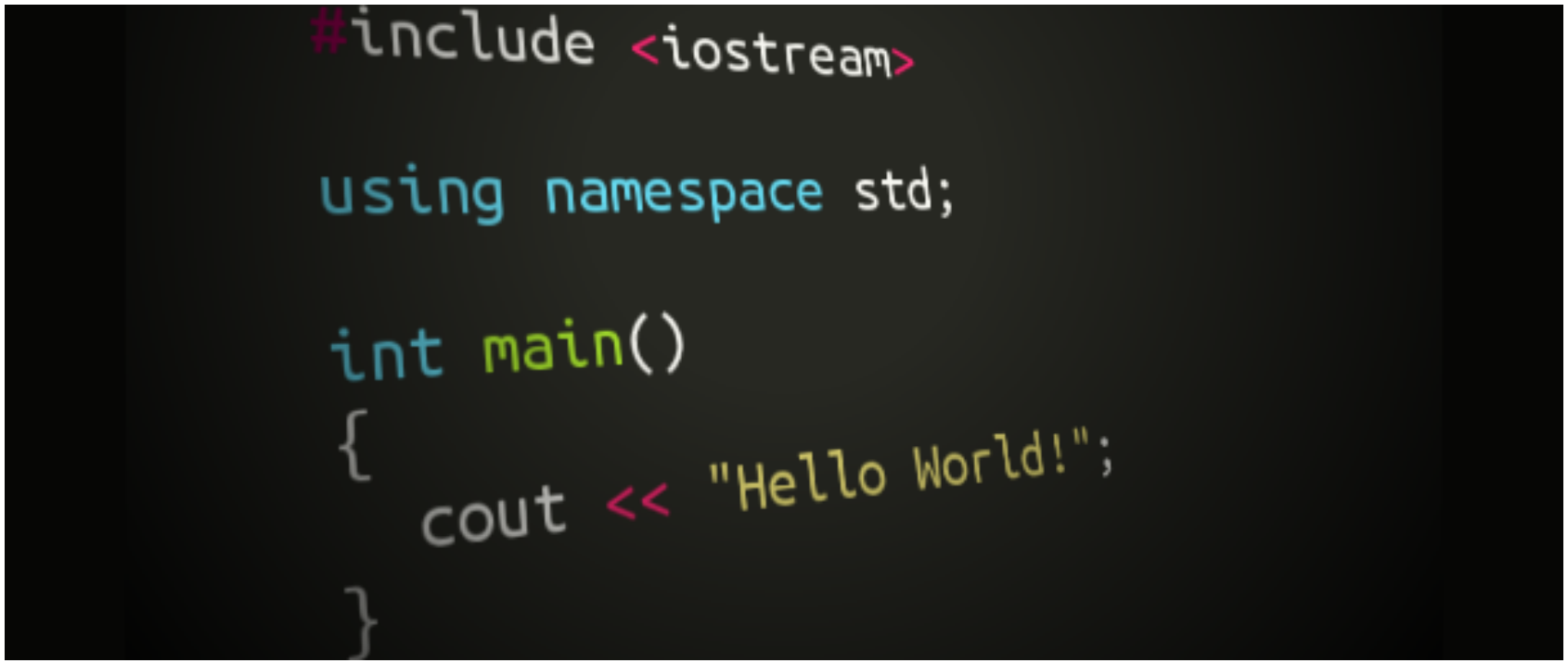
* Correspondence: Yana_Weinstein@uml.edu
[1]Department of Psychology, University of Massachusetts Lowell, Lowell, MA, USA
Full list of author information is available at the end of the article

Teaching the Science of Learning

Paper mit den in den Büchern besprochenen Lerntechniken

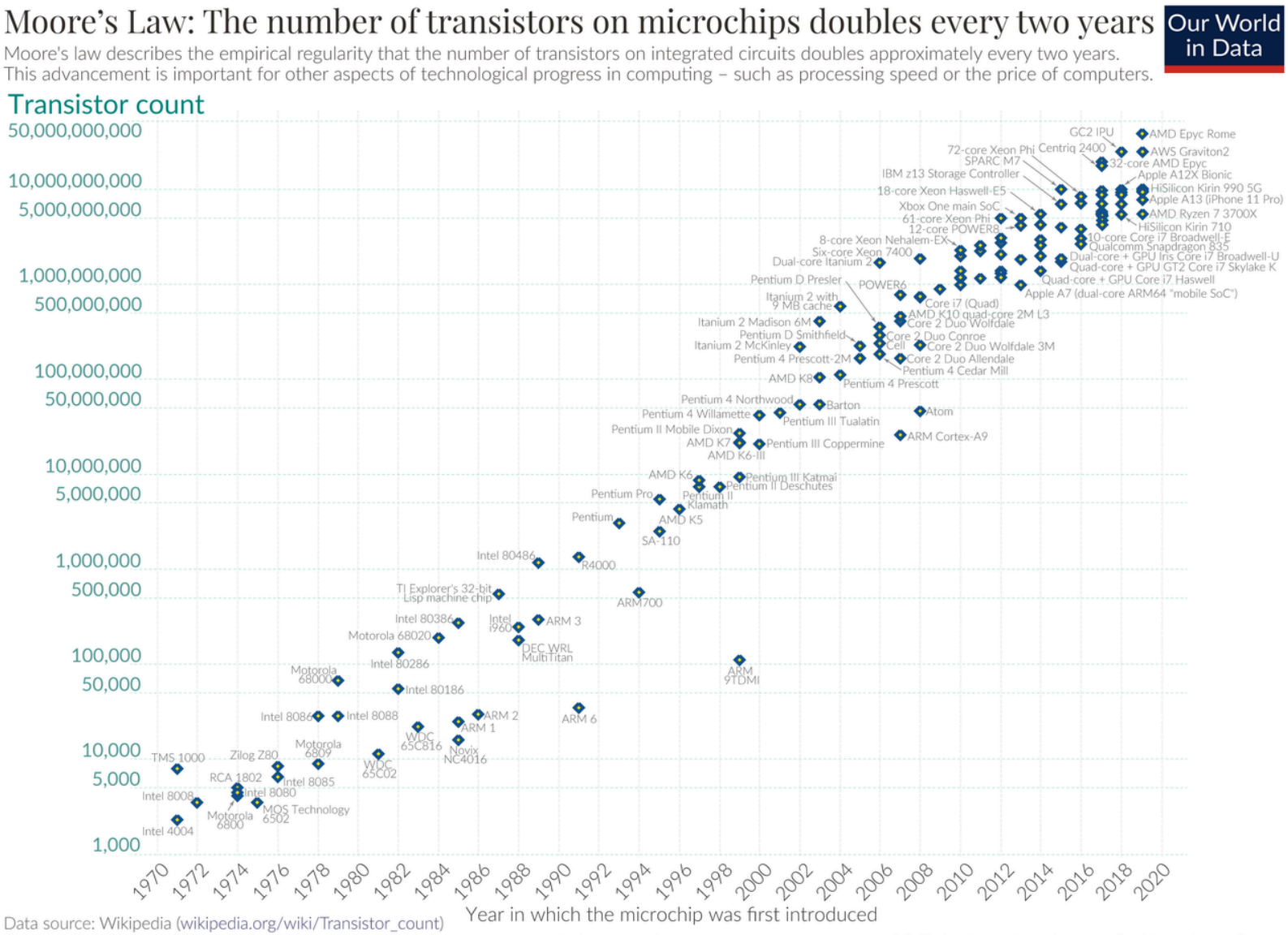
Richtig lernen: TL;DR

- Wiederholt durchlesen, Unterstreichen: Ineffektiv
- „Effortful learning": Arbeiten an der Grenze der eigenen Möglichkeit (anstrengend, aber nicht zu schwer)
- Jeden Textabschnitt reflektieren: „Was habe ich gelernt", schriftlich zusammenfassen
- Flashcards, selbst/gegenseitig abfragen
- Zeitlich gestaffelt wiederholen (aus der „Grenze des Vergessens" hervorholen)
- „Interleaving": Reihenfolge bei Wiederholung ändern, Verknüpfungen zu anderen Fakten/Ideen suchen

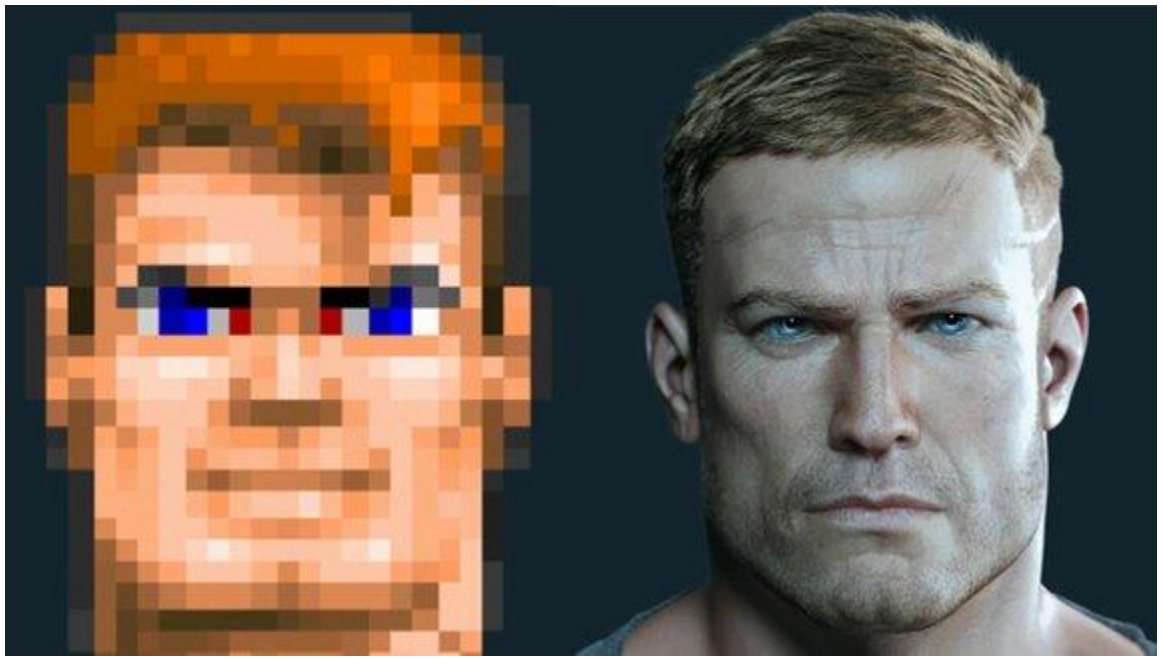

#include <iostream>
using namespace std;
int main()
{
cout << "Hello World!";
}

Einführung

Moore's Law


Moore's Law: The number of transistors on microchips doubles every two years
Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years.
This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.
Our World in Data
Transistor count
50,000,000,000
10,000,000,000
5,000,000,000
1,000,000,000
500,000,000
100,000,000
50,000,000
10,000,000
5,000,000
1,000,000
500,000
100,000
50,000
10,000
5,000
1,000
GC2 IPU
AMD Epyc Rome
72-core Xeon Phi
Centriq 2400
AWS Graviton2
SPARC M7
32-core AMD Epyc
IBM z13 Storage Controller
Apple A12X Bionic
18-core Xeon Haswell-E5
HiSilicon Kirin 990 5G
Xbox One main SoC
Apple A13 (iPhone 11 Pro)
61-core Xeon Phi
AMD Ryzen 7 3700X
12-core POWER8
HiSilicon Kirin 710
8-core Xeon Nehalem-EX
10-core Core i7 Broadwell-E
Six-core Xeon 7400
Qualcomm Snapdragon 835
Dual-core Itanium 2
Dual-core + GPU Iris Core i7 Broadwell-U
Quad-core + GPU GT2 Core i7 Skylake K
Pentium D Presler
POWER6
Quad-core + GPU Core i7 Haswell
Itanium 2 with 9 MB cache
Apple A7 (dual-core ARM64 "mobile SoC")
Core i7 (Quad)
AMD K10 quad-core 2M L3
Itanium 2 Madison 6M
Core 2 Duo Wolfdale
Pentium D Smithfield
Core 2 Duo Conroe
Itanium 2 McKinley
Cell
Core 2 Duo Wolfdale 3M
Pentium 4 Prescott-2M
Core 2 Duo Allendale
Pentium 4 Cedar Mill
AMD K8
Pentium 4 Prescott
Pentium 4 Northwood
Barton
Atom
Pentium 4 Willamette
Pentium III Tualatin
Pentium II Mobile Dixon
ARM Cortex-A9
AMD K7
Pentium III Coppermine
AMD K6-III
AMD K6
Pentium III Katmai
Pentium II Deschutes
Pentium Pro
Pentium II Klamath
Pentium
AMD K5
SA-110
Intel 80486
R4000
TI Explorer's 32-bit Lisp machine chip
ARM700
Intel 80386
Intel i960
ARM 3
Motorola 68020
DEC WRL MultiTitan
Intel 80286
ARM 9TDMI
Motorola 68000
Intel 80186
Intel 8086
Intel 8088
ARM 2
ARM 1
ARM 6
WDC 65C816
Motorola 6809
Novix NC4016
TMS 1000
Zilog Z80
WDC 65C02
RCA 1802
Intel 8085
Intel 8008
Intel 8080
MOS Technology 6502
Motorola 6800
Intel 4004
1970
1972
1974
1976
1978
1980
1982
1984
1986
1988
1990
1992
1994
1996
1998
2000
2002
2004
2006
2008
2010
2012
2014
2016
2018
2020
Year in which the microchip was first introduced
Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)
OurWorldinData.org – Research and data to make progress against the world's largest problems.
Licensed under CC-BY by the authors Hannah Ritchie and Max Roser.

Moore's Law



Quelle: Jeff Atwood

Wolfenstein: 1992 vs. 2014

Moore's Law

Quelle: vox.com

Lara Croft: 1996 - 2014

Moore's Law

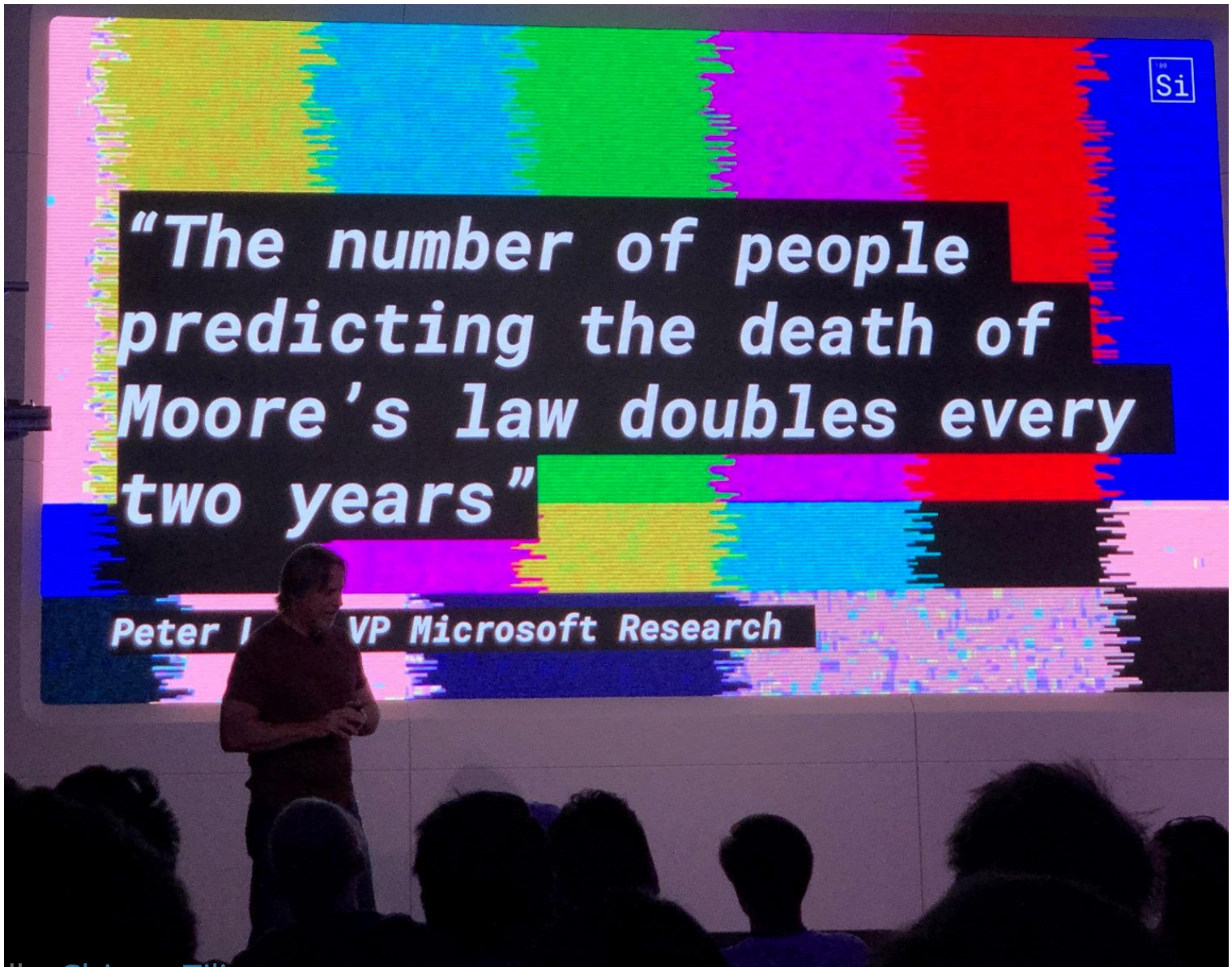
> *The number of transistors on integrated circuits doubles approximately every two years.*

...sagt nichts aus über die Anzahl der Kerne

...sagt nichts aus über die Rechenleistung einer CPU

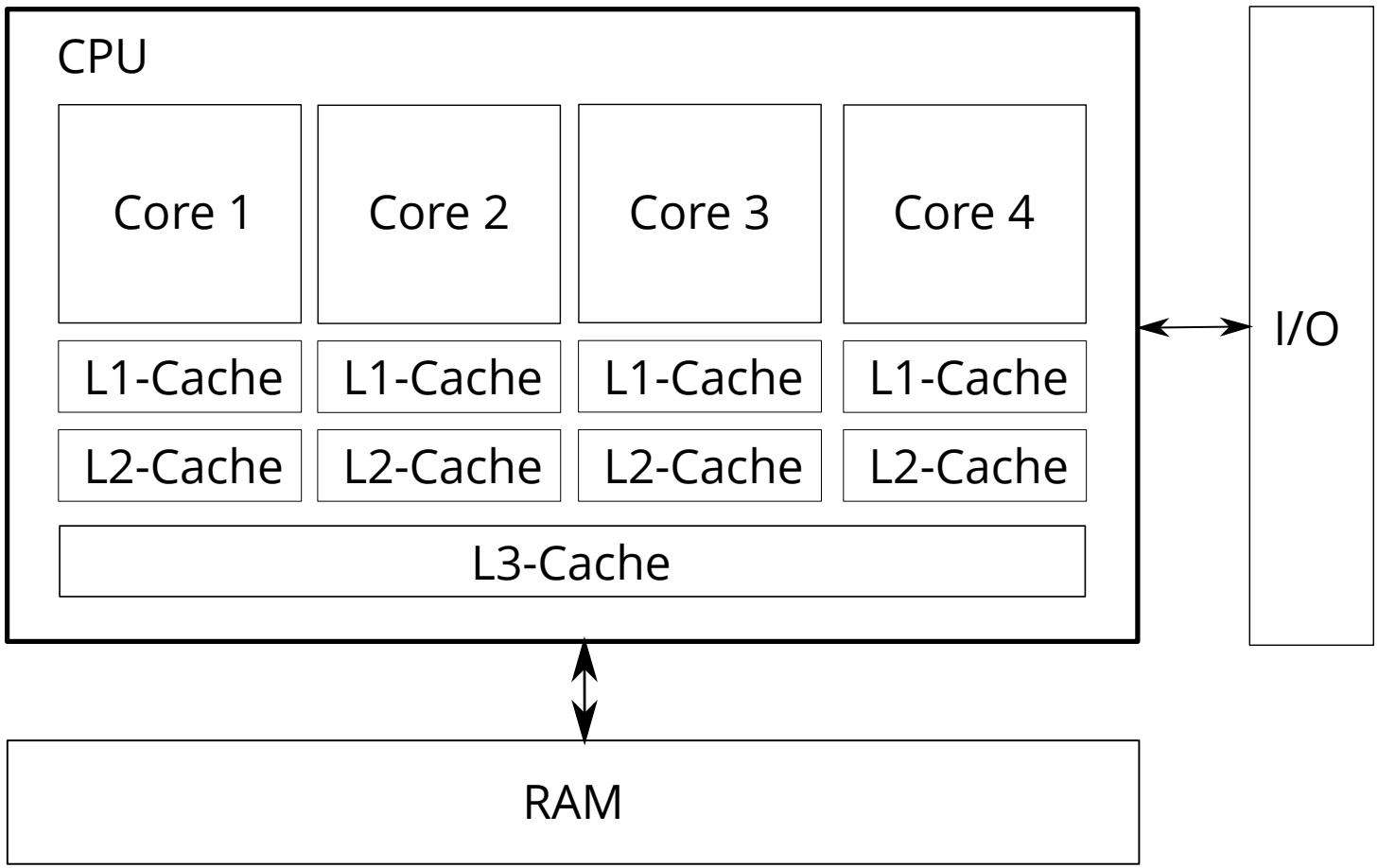
Leistungssteigerung durch Erhöhen der Taktfrequenz:
Hitzeproblem (4 GHz „Frequency Wall“)

Mehr Leistung durch mehrere Kerne und/oder mehrere CPUs



Quelle: Shivon Zilis

Typische Multicore-Architektur



Gegenüber Single-Cores: Gemeinsame Ressourcen!

„The free lunch is over!“

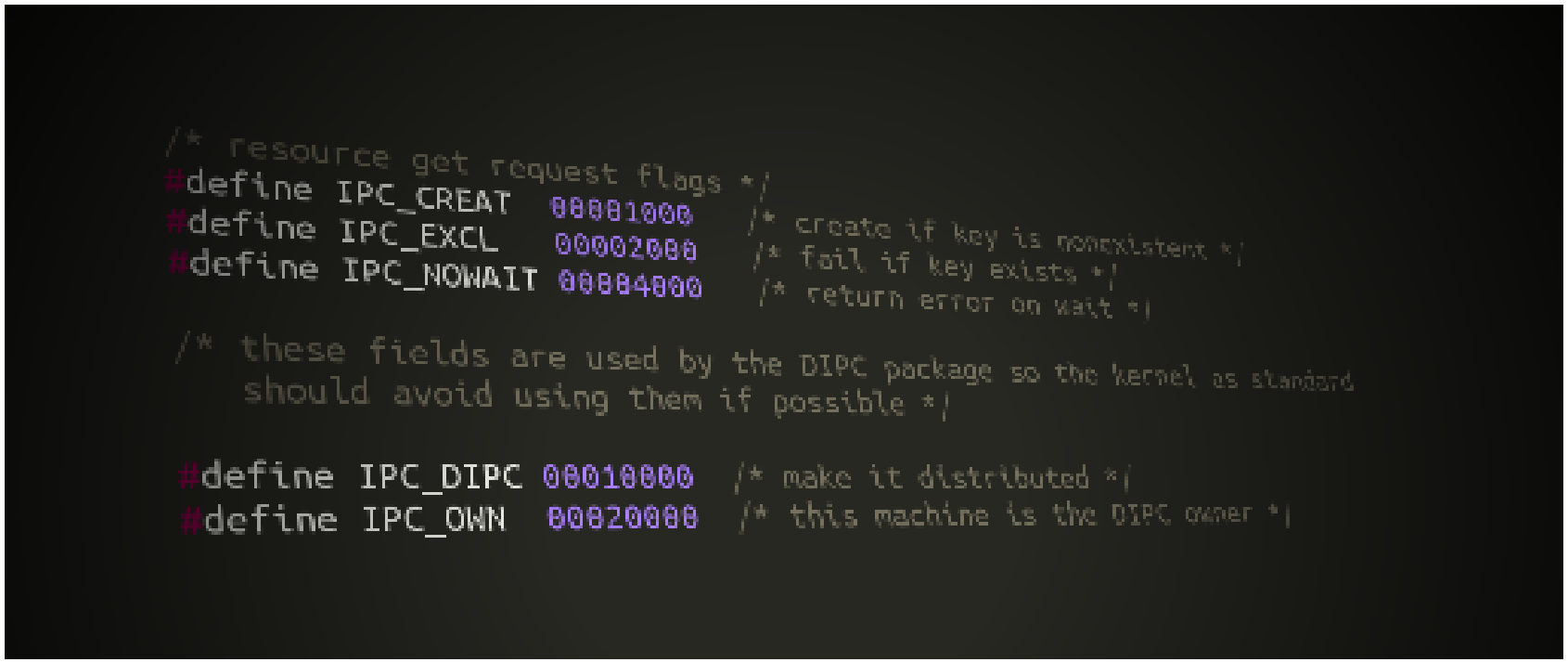
— Herb Sutter (2005)

Unser Ziel: Ressourcen maximal ausnutzen

Nebenläufigkeit vs. Parallelität

Parallelität (Parallelism): Gleichzeitiger Ablauf von Programmen/-teilen, welche keine gemeinsamen Ressourcen (Speicherbereiche, Objekte) teilen

Nebenläufigkeit (Concurrency): Gleichzeitiger Abläufe, welche während der Ausführung auf gemeinsame Ressourcen zugreifen



```
/* resource get request flags */
#define IPC_CREAT  00001000   /* create if key is nonexistent */
#define IPC_EXCL   00002000   /* fail if key exists */
#define IPC_NOWAIT 00004000   /* return error on wait */

/* these fields are used by the DIPC package so the kernel as standard
   should avoid using them if possible */

#define IPC_DIPC 00010000  /* make it distributed */
#define IPC_OWN  00020000  /* this machine is the DIPC owner */
```

Prozesse und IPC

Was gehört zu einem Prozeß?

- Programm
- Heap
- Stack
- Prozessorregister, Stackpointer, Program Counter
- Offene/verwendete Ressourcen

Eigenschaften

Einheit der Nebenläufigkeit (Multitasking)

Vom OS-Scheduler zur Ausführung gebracht (präemptives Multitasking)

Separater Adressraum: Fehlerhafte Programmierung bleibt in Prozeß isoliert

Rechteverwaltung: Läuft mit bestimmten Privilegien, kann u.U. diese selbst weiter einschränken

Fork zur Aufgabenverteilung

Aufgaben werden in separatem Kindprozeß (fork) behandelt, welcher bei Bedarf angelegt wird

Sinnvoll bei länger laufenden Aufgaben

Beim Unix-Prozess-Modell: Einfache Übergabe von Startparametern (gesamter Zustand wird geerbt)

Ergebnis muss über geeigneten Kommunikationskanal übertragen werden

Prefork-Modell

Anlegen von Prozessen vergleichsweise „teuer“. Deshalb: Prozesse erzeugen, bevor die eigentliche Anfrage eintrifft

Eigentliche Aufgabe muss über geeigneten Kommunikationskanal übertragen werden (ebenso ggfs. das Ergebnis)

Typisch für Prefork: Prozess wird nach Bearbeitung einer Anfrage nicht beendet, sondern wartet auf die nächste Anfrage

Prefork-Modell

Unter Umständen macht es Sinn, die Zahl der Kindprozesse in gewissen Grenzen variabel zu halten

- Beispiel Apache (MPM Prefork): Konfiguration minimaler und maximaler Anzahl an Prozessen
- Sind alle Prozesse beschäftigt und die maximale Anzahl noch nicht erreicht: Neuer Worker-Prozess
- Ist ein Worker-Prozess eine gewisse Zeit idle und die minimale Anzahl noch nicht unterschritten: Beenden

Fork-/Prefork-Modell

Nachteile:

- Anlegen von Prozessen ist relativ schwergewichtig (häufiges Anlegen und Wegwerfen sorgt für unnötig Systemlast)
- Austausch größerer Datenmengen aufwendig (häufiger Kontextwechsel, ist „teuer“)

Vorteile:

- Im Falle eines Programmierfehlers stirbt nur ein einzelner Prozeß, nicht das gesamte Programm. Kann bei der Implementierung ausgenutzt werden (möglichst einfacher Vaterprozess, der nur Arbeit weiterverteilt)
- Kindprozesse können unterschiedliche Rechte haben (Beispiel Apache: Pro virtuellem Host separate User-ID)

Pipes

- Von der Kommandozeile bekannt: cat foo.txt | sort
- Das Filehandle von stdout von cat wird mit stdin von sort verbunden
- Kommunikation über Puffer:
  - Lesen aus leerem Puffer: Blockiert
  - Schreiben in vollen Puffer: Blockiert
- Programmatisch anlegen mit pipe(). Ergibt unidirektionale Verbindung, liefert zwei Filehandles (ein lesendes und ein schreibendes Ende)
- Wird die lesende Seite geschlossen, erzeugt ein Schreibversuch den Fehler „broken pipe“

Named Pipes

- Funktionieren analog zu (anonymen) pipes
- Haben allerdings einen symbolischen Namen im Dateisystem
- Lesen bzw. Schreiben in Named Pipe: Einfach die „Datei“ öffnen
- Einfache Methode, um einem laufenden Programm Steuerbefehle zukommen zu lassen
- Erzeugen an der Kommandozeile: mkfifo (identischer Name in C)

Shared Memory

Es besteht die Möglichkeit, gemeinsam genutzten Speicherbereich zu allokieren

Kann entweder privat sein (Weitergabe über fork) oder eine well-known ID (und Zugriffsrechte wie Dateien) besitzen

Zugriff unsynchronisiert! Prozesse wissen von gegenseitigen Lese- und Schreibzugriffen nichts

Je nach OS: Shared-Memory-Blöcke überleben Prozessende!

Race Conditions

Race Condition: Mehrere Prozesse verwenden dieselbe Ressource gleichzeitig

Bei File Handles: Strom-Semantik! Race-Condition, wenn dieselbe Operation (lesen/schreiben) gleichzeitig verwendet wird.

- Schreiben: Ein einzelner Leser bekommt eine zufällige Mischung aus beiden Schreiboperationen.
  Schreiboperationen sind per se nicht atomar
- Lesen: Mehrere Leser greifen auf denselben Puffer zu und lesen zufällige Anteile daraus

Race Conditions

Bildlich gesprochen… ;-)

Programmieraufgabe: Race Conditions

Erstelle ein Programm in Java, welches eine „zeitintensive“ Rechenaufgabe auf eine konfigurierbare Anzahl von „Prozessen“ verteilt.

- Kommunikation: Simulierte Pipes, String-Schnittstelle (simuliert stdin und stdout echter Prozesse)
- Jeder Kind„prozess“ soll mittels sleep 1 Sekunde warten und anschließend das Ergebnis „42“ zurückliefern
- Der Vaterprozess soll die Ergebnisse sammeln, addieren und das Ergebnis ausgeben

Projekt-Template: racecond.zip

Hinweise Import in IntellIJ

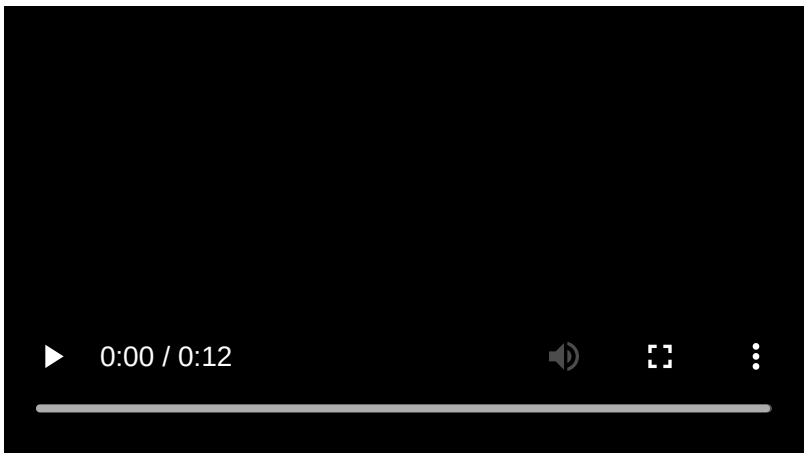
Jede zip-Datei erzeugt ihren „eignenen“ Unterordner - also „hier entpacken“ wählen

Import in IntellIJ: Den Ordner wählen, in dem auch die build.gradle-Datei liegt (Wurzelverzeichnis des Projekts)

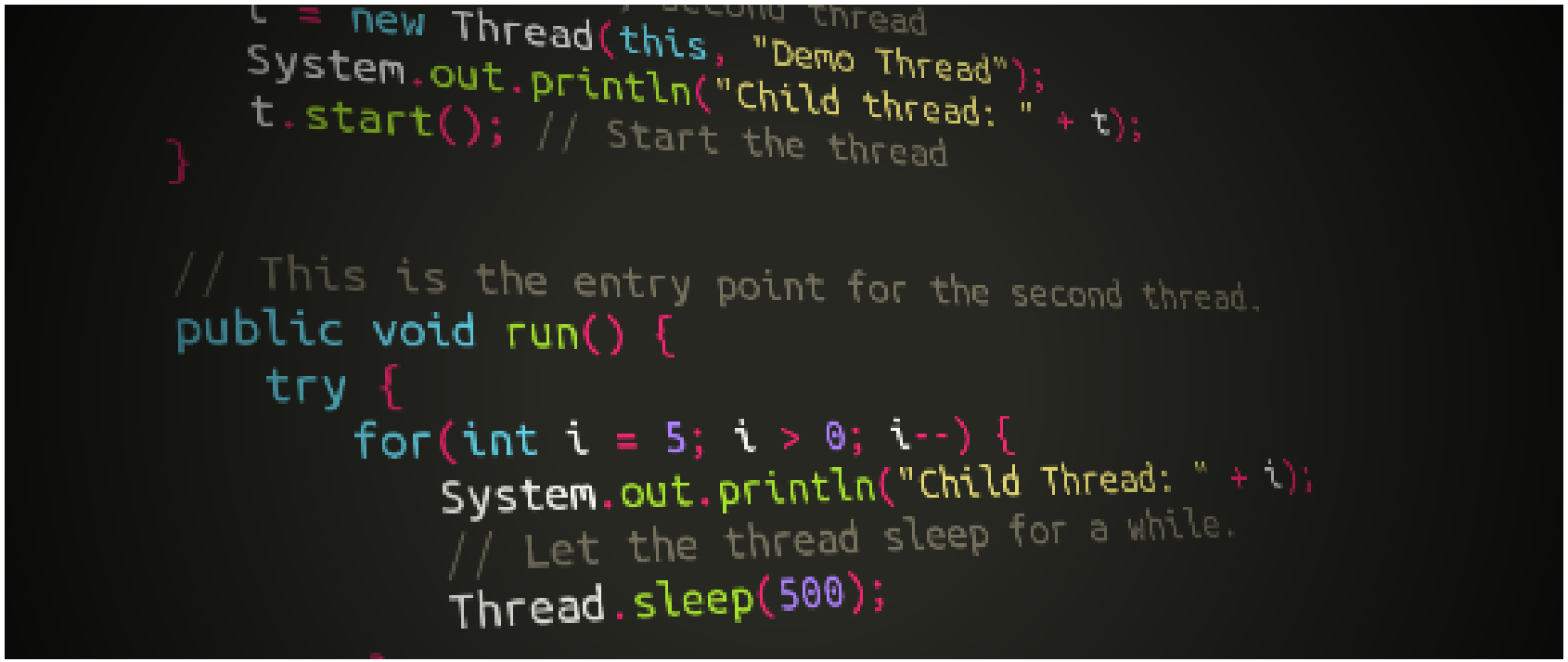
Fehler beim Download von Abhängikeiten: „PKIX path building failed: sun.security.provider.certpath.SunCertPathBuilderException: unable to find valid certification path to requested target“: Ein Firmenproxy besitzt ein selbstsigniertes Zertifikat - dieses muss in den JDK Truststore importiert werden

- https-Seite im Browser öffnen, Root-Zertifikat speichern
- bin\keytool -import -alias companyca -file rootcert.cer -keystore lib\security\cacerts -storepass changeit

First attempt at async programming



Quelle: @jonathansampson



Multithreading

Threads vs. Prozesse

Prozess: Schwergewichtig, gegeneinander abgeschottet

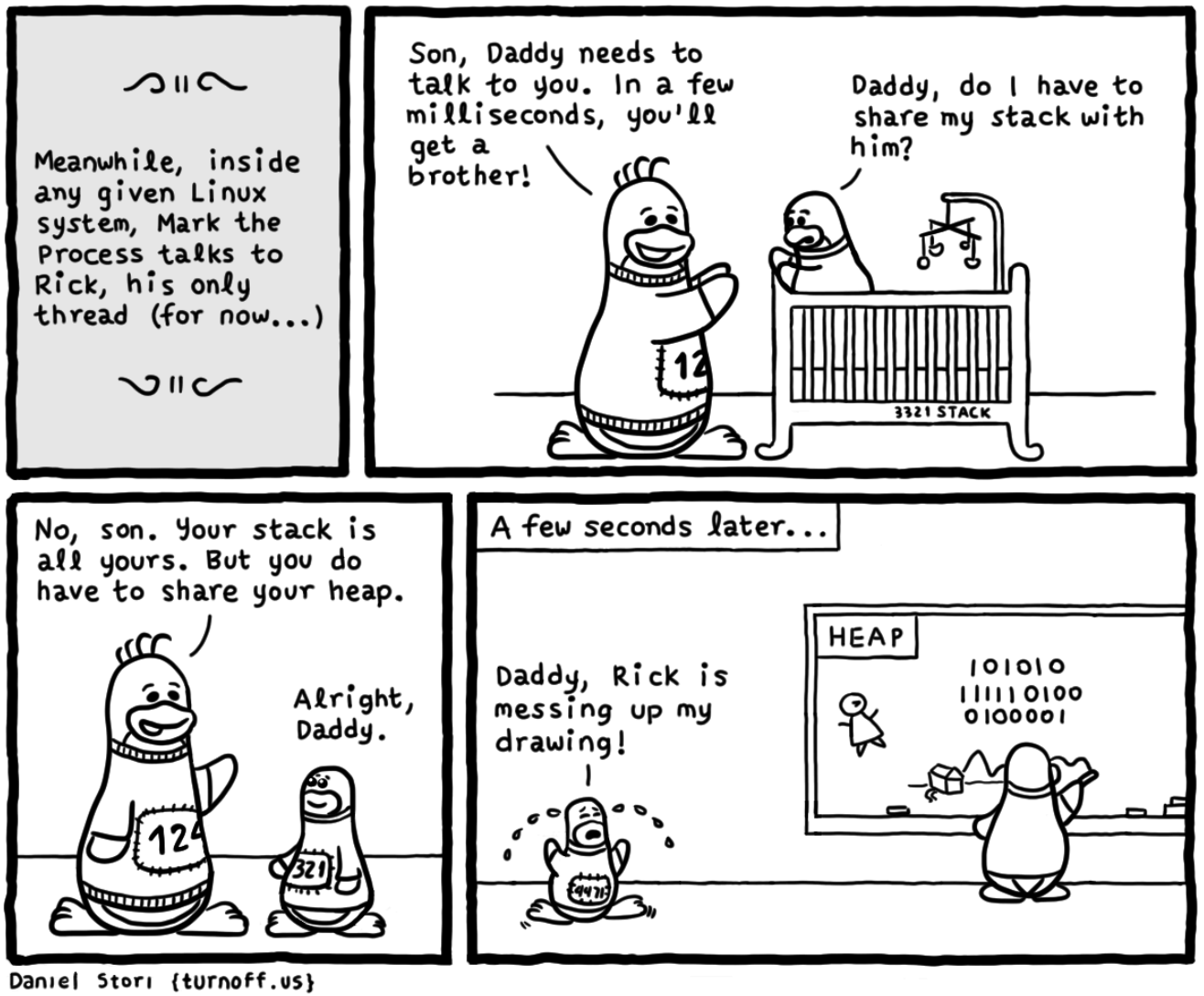
Thread (auch manchmal „Lightweight Process“ genannt): Leichtgewichtig, mehrere existieren innerhalb desselben Prozesskontexts

Teilen sich gemeinsamen Speicher (gemeinsamer Code, gemeinsamer Heap), besitzen eigenen Stack

Je nach Sprache/System: Zusätzlicher Thread-lokaler Speicher

Threadwechsel: Austausch von CPU-Registern, Stackpointer und Program Counter

Ein Vorgeschmack...



Quelle: turnoff.us

Threads in Java

Entweder: Thread-Klasse ableiten und `run()` überschreiben

Oder: Eigene Klasse, welche `Runnable` implementiert und mit Hilfe eines Threads ausgeführt wird

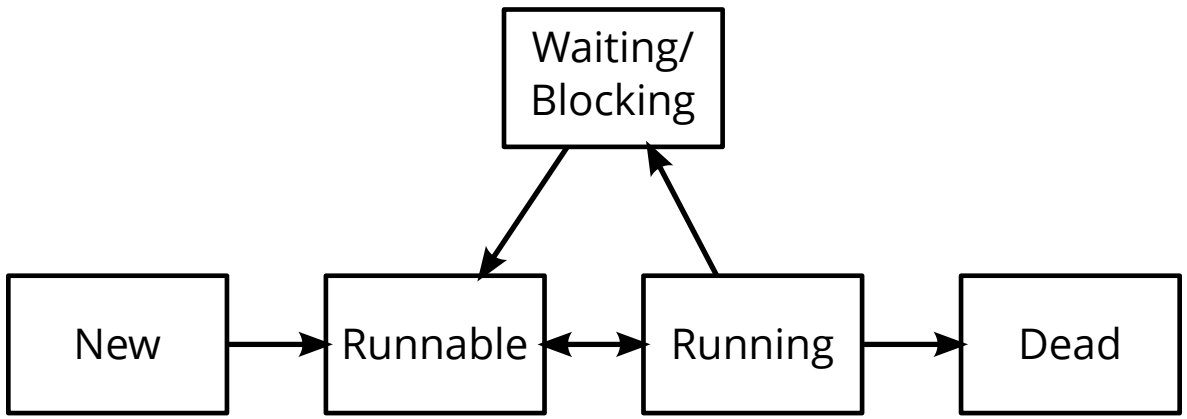
Thread wird nach dem Erzeugen mittels `start()` gestartet

Thread wird beendet, wenn die run-Methode verlassen wird

Prozess wird beendet, wenn der letzte Thread beendet wird oder nur noch Daemon Threads laufen

Daemon Thread: `setDaemon(true)` vor Threadstart

Java Thread-Zustände



New → Runnable: `start()` (Eine Thread-Instanz kann nur 1x gestartet werden)

Runnable/Running: Scheduler, `yield()`

Running → Dead: Verlassen der run-Methode

Waiting/Blocking: `sleep()`, blocking I/O, Synchronisation, ...

Green Threads vs. Native Threads

Native Threads: Threads werden von der virtuellen Maschine auf echte Betriebssystem-Threads abgebildet

Green Threads: VM kümmert sich um die Abbildung auf OS-Prozesse/-Threads und übernimmt das Scheduling

- Aufwendigere Implementierung für die VM; blockierende I/O problematisch (Verwendung asynchroner I/O-Routinen oder separate Prozesse/Threads für I/O)
- Potentiell „billiger“ und schneller zu erzeugen (benötigt keine OS-Ressourcen)
- Unabhängig vom der Betriebssystem-Unterstützung

Bis Java 1.2 noch Standard; dann entfernt (evtl. in speziellen Embedded JVMs?).

Auf einen Thread warten

Warten auf Ende eines Threads: `t.join()`

Blockiert, bis Thread beendet ist

Thread (von außen) anhalten

Thread terminiert, wenn `run` verlassen wird

Signalisierung von außen („Wunsch auf Beenden“):
`t.interrupt()`

Methoden wie `sleep` oder `wait` „wachen auf“ und werfen eine `InterruptedException`

Die Methode `t.isInterrupted()` innerhalb des Threads liefert `true`

Das Programm muß darauf geeignet reagieren

Beispiel: sleep

```
Thread t = new Thread(new Runnable() {
  @Override public void run() {
    while (true) {
      System.out.println("Thread running…");
      try {
        Thread.sleep(250);
      } catch (InterruptedException e) {
        System.out.println("Thread interrupted, exiting.");
        break;  // Verantwortung des Programmierers: run verlassen
      }
    }
  }
});

t.start();
Thread.sleep(1000);
t.interrupt();
t.join();
```

Beispiel: Aufwendige Berechnung

```
Thread t = new Thread(new Runnable() {
  @Override public void run() {
    // Verantwortung des Programmierers: Prüfen auf Abbruch-Signal
    while (!Thread.currentThread().isInterrupted()) {
        System.out.println("Thread running…");
        long sum = 0;
        for (long i=0; i<10000000000L; i++) {
            sum += i;
        }
    }
    System.out.println("Thread interrupted, exiting.");
  }
});

t.start();
Thread.sleep(1000);
t.interrupt();
t.join();
```

Was ist mit Thread.stop()?

tl;dr: *Nicht verwenden!*

Unterbricht *sofort* den Thread, indem innerhalb des Threads eine `ThreadDeath`-Exception geworfen wird

finally-Blöcke werden ausgeführt, synchronized-Klammern korrekt beendet

Unterbrechung möglicherweise in kritischen Abschnitten, ggfs. inkonsistente Daten. Probleme beim Versuch der Konsistenz-Recovery durch Fangen von `ThreadDeath`:

- Unklar, wo die Unterbrechung auftrat, daher keine Information, welche Daten korrupt sind
- Die Behandlung der Exception kann erneut durch `ThreadDeath` unterbrochen werden

Probleme bei gleichzeitigem Speicherzugriff

Zusätzlich zu den Race Conditions bei Prozessen!

Dirty reads: Lesen, während eine Schreiboperation noch nicht vollständig abgeschlossen ist

Sichtbarkeitsprobleme aufgrund „weicher Bedingungen“ für das Speichermodell

- CPU-Caches: U.U. keine Cache-Synchronisation aus Performance-Gründen
- Keine per-se konsistente Sicht, um JIT-Optimierungen zu erlauben bzw. unnötige Synchronisation zu vermeiden

Java (und viele andere Sprachen) geben keinerlei Garantie, welche Threads wie lange ausgeführt bzw. gescheduled werden

Beispiel: Naive Bank

```
public static void transfer(Account fromAccount, Account toAccount, long amount) {
  fromAccount.amount -= amount;
  toAccount.amount += amount;
  try {
    Thread.sleep(1);
  } catch (InterruptedException e) { }
}

class MoneyTransfer implements Runnable {
  Account fromAccount, toAccount;
  long amount;
  int count;

  MoneyTransfer(Account fromAccount, Account toAccount, long amount, int count) {
    this.fromAccount = fromAccount;
    this.toAccount = toAccount;
    this.amount = amount;
    this.count = count;
  }

  @Override
  public void run() {
    for (int i=0; i<count; i++) {
      transfer(fromAccount, toAccount, amount);
    }
  }
}
```

Beispiel: Naive Bank

```
public void run() throws InterruptedException {
  Account a = new Account(10000);
  Account b = new Account(10000);

  System.out.println("Start balance: A=" + a + ", B=" + b);

  Thread t1 = new Thread(new MoneyTransfer(a,b,10,1000));
  Thread t2 = new Thread(new MoneyTransfer(b,a,10,1000));
  t1.start();
  t2.start();
  t1.join();
  t2.join();

  System.out.println("End balance: A=" + a + ", B=" + b);
}
```

Ergebnisse:

- End balance: A=Account(amount=9980), B=Account(amount=9950)
- End balance: A=Account(amount=10040), B=Account(amount=10020)
- ...

Nicht alles ist eitel Sonnenschein…

Entweder hat jeder seins…
…oder der Programmierer muss sich kümmern

(und durch Sperren o.ä. für Ordnung sorgen)

Fragen bei nebenläufigen Operationen

Synchronisation und Sichtbarkeit: Wann ist ein Datum, das von einem Thread geschrieben wurde, für die anderen sichtbar?

Thread-Safety: Kann eine Funktion bzw. eine Klasse ohne weitere Vorkehrungen von mehreren Threads parallel verwendet werden?

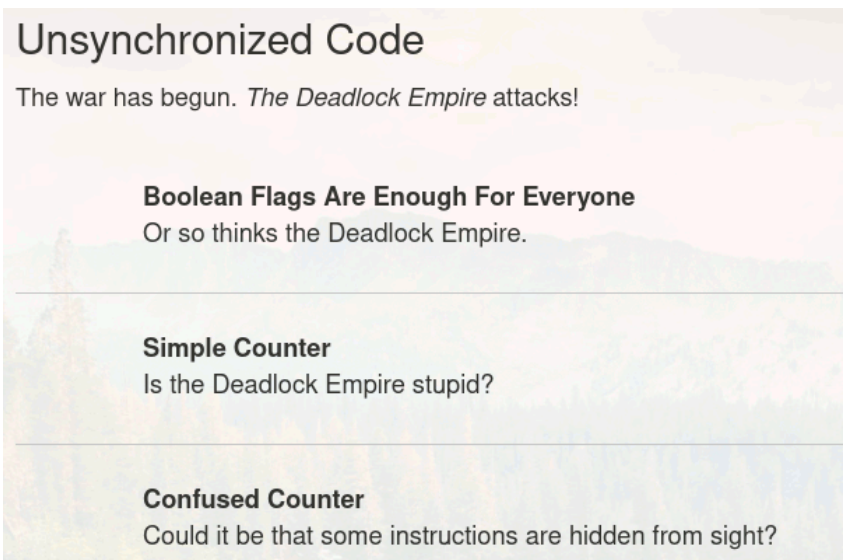
Atomare Operationen: Bei welchen Operationen sind die Effekte entweder komplett oder garnicht sichtbar? Anders formuliert: Bei welchen Operationen sind nie inkonsistente Zwischenstände sichtbar?

Terminierung: Wann gilt ein Prozess als beendet?

Übung: Deadlock Empire pt. 1

https://deadlockempire.github.io/

Abschnitt „Unsynchronized Code“



Synchronisierung: Critical Section

Nur ein Thread darf gleichzeitig einen kritischen Abschnitt betreten. Identifizierung erfolgt über eine Monitorvariable (Synchronisationsobjekt):

```
MyClass foo = new MyClass();
synchronized(foo) {
  // ...
}
```

Zusätzlich unterstützt Java das Schlüsselwort `synchronized` in Methodensignaturen:

```
public synchronized int getAnswer() { /* After long calculation */ return 42; }
```

Dies ist äquivalent zu

```
public int getAnswer() {
  synchronized(this) { /* After long calculation */ return 42; }
}
```

Synchronisierung: Locks

Sehr ähnlich: Erster Thread, der ein Lock anfragt, erhält dieses. Alle anderen Threads blockieren, bis das Lock freigegeben wird.

```
Lock lock = new ReentrantLock();
lock.lock();
try {
  // ...
} finally {
  lock.unlock();
}
```

Variante: `tryLock()`: Versuch (ohne Blockierung), die Sperre anzufordern

Reentrant: Derselbe Thread kann die Anforderungs-Barriere erneut passieren (z.B. bei rekursiven Aufrufen)

Wichtig: Freigabe garantieren! (siehe nächste Folie)

try... finally

Im Gegensatz zu `synchronized`: Es gibt *keinen* Automatismus für die Freigabe von angeforderten Locks beim Verlassen des Scopes

Unchecked Exceptions können theoretisch *überall* auftreten

Daher: Essentiell notwendig, bei Paaren wie `lock`/`unlock`...

- Den Code, der die Sperre hält, in einem `try`-Abschnitt zu halten
- Die Freigabe im anschließenden `finally`-Abschnitt durchzuführen
- Der `try`-Abschnitt muß unmittelbar nach dem (erfolgreichen) Anfordern der Sperre beginnen

Read-Write-Locks

Unterscheidet Leser und Schreiber.

```
ReadWriteLock lock = new ReentrantReadWriteLock();

Lock rl = lock.readLock();
rl.lock();
try {
  // Beliebig viele Leser, aber kein Schreiber können sich hier aufhalten
} finally {
  rl.unlock();
}

Lock wl = lock.writeLock();
wl.lock();
try {
  // Nur ein einziger Schreiber (und keine Leser) können sich hier aufhalten
} finally {
  wl.unlock();
}
```

Fairness (optionaler Konstruktorparameter): Wartende Threads werden (in etwa) „first-come, first-served“ behandelt

Deadlocks

Deadlock: Wechselseitiges Anfordern von Locks

```
public static void transfer(Account fromAccount, Account toAccount, long amount) {
  synchronized (fromAccount) {
    synchronized (toAccount) {
      fromAccount.amount -= amount;
      toAccount.amount += amount;
      try {
        Thread.sleep(1);
      } catch (InterruptedException e) { }
    }
  }
}
```

Erkennung: „Wartet-auf“-Graph

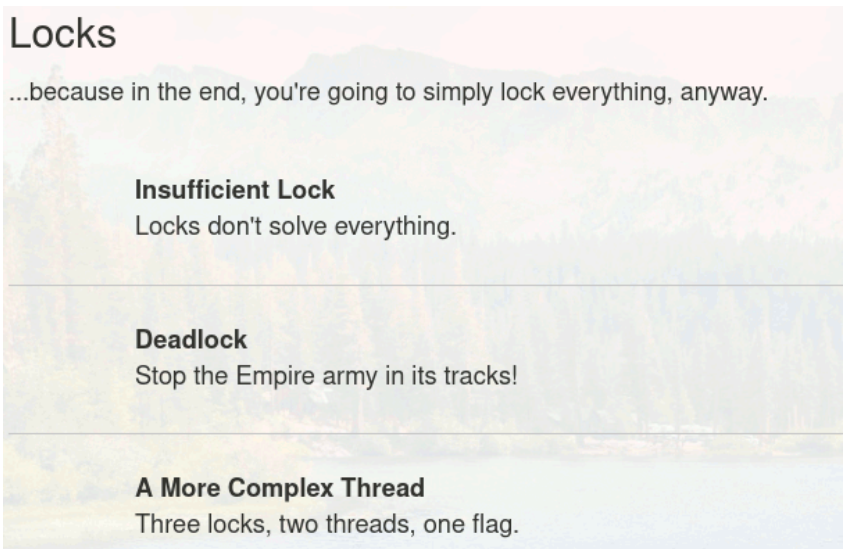
Kann nur lokale Locks erkennen! Probleme bei Warten auf Netzwerkkommunikation o.ä.

Vermeidungsstrategien: Lock-Reihenfolge

Übung: Deadlock Empire pt. 2

https://deadlockempire.github.io/

Abschnitt „Locks“



Lock-Reihenfolge

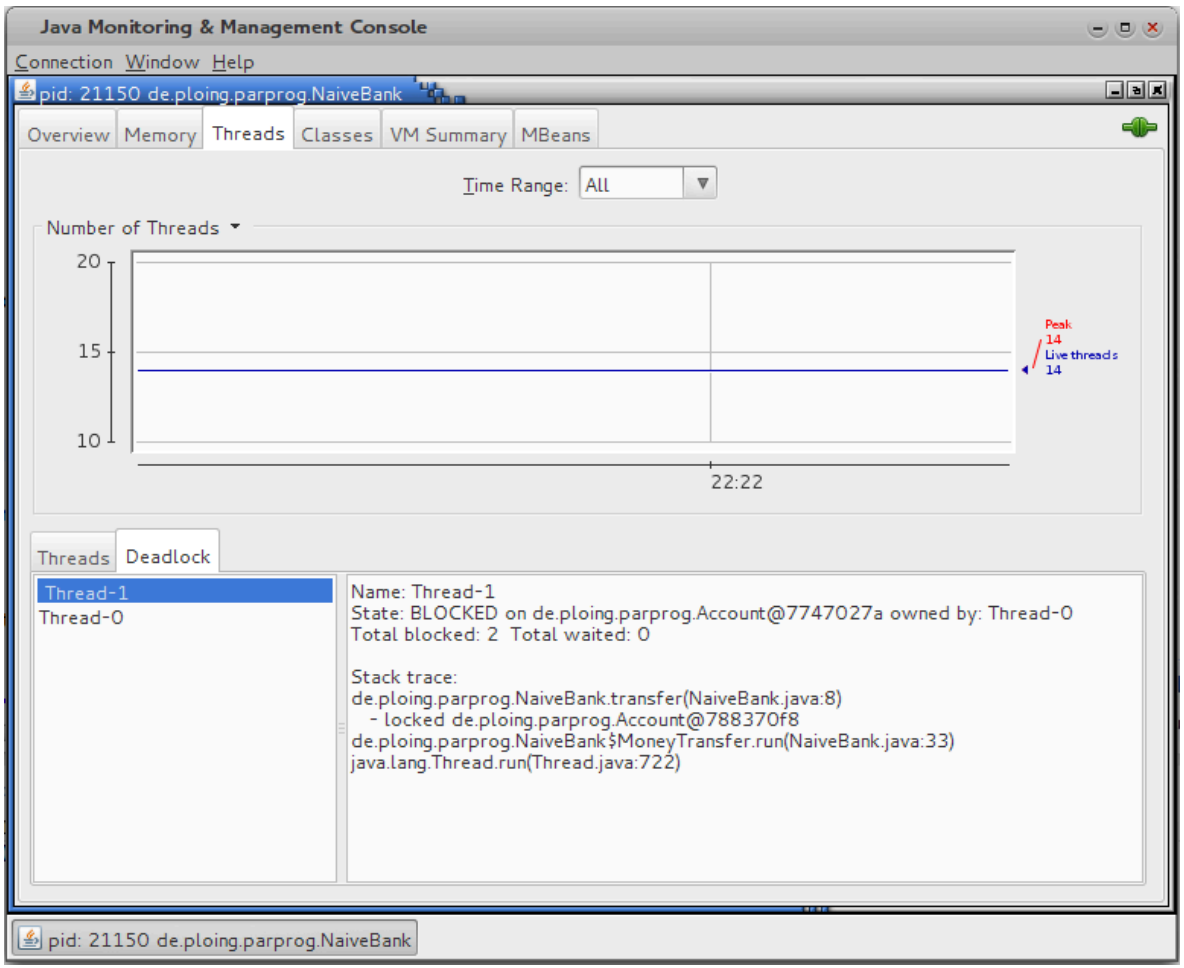
Idee: Vermeiden von Verschränkungen durch Definieren einer Reihenfolge

Typische Strategie:

- Zwischen Verschiedenen Klassen: Bestimmte Reihenfolge (z.B. immer erst alle Konten, dann alle Kundendaten)
- Innerhalb einer Klasse: Nach bestimmtem Sortierkriterium (z.B. Kundendaten nach Kundennummer)

Deadlock-Debugging

Anzeige mit JConsole



Alternativ (Kommandozeile): jps, jstack

Starvation, Livelocks

Starvation: Wenn Ressourcen (unnötig) lange gehalten werden (und der Wartemechanismus keine Fairness garantiert), kann es vorkommen, daß ein einzelner Thread „ewig“ wartet und „verhungert“.

Starvation kann ein Symptom dafür sein, dass die Sperre aufgrund eines Programmierfehlers nicht freigegeben wurde.

Livelock: Wenn Threads gegenseitig auf Aktivitäten reagieren, kann es zu einer Endlos-Feedback-Schleife kommen.

Programmieraufgabe: Smart Bank

Erstelle ein Programm in Java auf Basis des Templates smartbank.zip, welches threadsichere Operationen auf Bankkonten erlaubt. Die Bank soll folgende Funktionen implementieren:

```
public synchronized Account createAccount();
public long getBalance(Account account);
public void deposit(Account account, long amount);
public void withdraw(Account account, long amount);
public void transfer(Account fromAccount, Account toAccount, long amount);
```

Die Implementierung von `Account` ist freigestellt. Der Kontostand muß immer im Wertebereich [0 .. 100.000] sein.

Bei einer Operation auf Konten sollen unbeteiligte Accounts nicht unnötig blockiert werden (kein globales Lock o.ä.).

Semaphore

Ähnlich wie Lock, nur daß eine bestimmte Anzahl an Threads gleichzeitig in einen Abschnitt dürfen

```
Semaphore semaphore = new Semaphore(10);
semaphore.acquire();
try {
  // Hier können sich maximal 10 Threads aufhalten
} finally {
  semaphore.release();
}
```

Kann als Schutz vor Überlastung dienen

`drainPermits` braucht sämtliche verbleibenden Permits auf (liefert die Anzahl als Rückgabewert)

Optionaler Parameter für `acquire` und `release`: Anzahl von Permits (z.B. zur Regulierung unterschiedlich belastender Aufgaben)

Countdown Latch

Ein Countdown-Zähler, der eine Sperre freigibt, sobald 0 erreicht ist

```
CountDownLatch latch = new CountDownLatch(5);
// Hiervon werden 5 Thread-Instanzen gestartet
public void run() {
  // Unabhängige Berechnung
  latch.countDown();
  latch.await();
  // Nun haben alle Threads ihre Berechnung abgeschlossen
}
```

Anwendungsbeispiele:

- Wenn *n* Threads eine Operation vollzogen haben
- Wenn eine Operation *n* Mal erfolgt ist

Barriers

Hält Threads an der „Barriere“ an, bis *n* Threads diese erreicht haben

```
CyclicBarrier barrier = new CyclicBarrier(5);
// Hiervon werden 5 Thread-Instanzen gestartet
public void run() {
  // Unabhängige Berechnung
  barrier.await();
  // Nun haben alle Threads ihre Berechnung abgeschlossen
}
```

Nach erreichen des Barriere-Kriteriums wird der Zähler wieder auf den Ausgangswert zurückgesetzt, kann also erneut verwendet werden („Cyclic“).

Konstruktor-Variante mit `Runnable`-Instanz: Diese wird ausgeführt, nachdem der Schwellwert erreicht ist, aber bevor die Threads weiterlaufen dürfen (z.B. zum Einsammeln und Verrechnen von Ergebnissen).

Java-Speichermodell

Beschreibt Interaktion von Threads mit dem Speicher

Programmablauf folgt *innerhalb eines Threads* der „happens-before“-Semantik: Das Programm muß so ausgeführt werden, daß die *Beobachtungen* dem im Code notierten Ablauf entsprechen.

Bedeutet salopp: So lange es keinen Unterschied macht, können Anweisungen auch in anderer Reihenfolge ausgeführt werden. Folgendes wäre äquivalent:

```
int a=5;
int b=10;
```

```
int b=10;
int a=5;
```

Java-Speichermodell

Dieses Modell erlaubt es, thread-lokale Caches gemeinsamer Variablen zu haben

volatile, Lock-Benutzung, Beginn und Ende eines kritischen Abschnitts: Memory-Barrier, „Happens-before“-Bezug auf gesamten vorigen Code

Performance-Einbußen bei Synchronisierung

- Verhinderung von Parallelität
- Zwang des Speicherabgleichs
- Synchronisierte vs. unsynchronisierte Methoden: Laut Literatur Faktor 100!

Welche Ergebnisse sind möglich?

Initial sind A=0 und B=0. Welche Werte können r1 und r2 annehmen?

Thread 1	Thread 2
1: r2 = A;	3: r1 = B;
2: B = 1;	4: A = 2;

Mögliche Ausführungsreihenfolgen (nicht vollständig):

- 1 - 2 - 3 - 4: r2=0, r1=1
- 3 - 4 - 1 - 2: r1=0, r2=2
- 1 - 3 - 2 - 4: r2=0, r1=0
- Aber auch: 2 - 3 - 4 - 1: r1=1, r2=2 (Umstellen der Reihenfolge)

Quelle: Java Language Specification

volatile / j.util.concurrent.atomic.*

Schlüsselwort `volatile`: Immer synchronisierte Variable

- Lesen oder Schreiben ist atomar und stets aktuell
- Schreibzugriff happens-before folgenden Lesezugriffen
- volatileVar++ ist nicht threadsafe!

Hilfsklassen wie `AtomicInteger` besitzen atomare Operationen wie `addAndGet`, `getAndSet`, etc.

Memory Barrier zwischen den Threads, welche auf dieselbe Variableninstanz zugreifen

Double-checked locking

Vermeintlich gute Optimierungs-Idee:

```
private Resource resource = null;
public static Resource getResource() {
  if (resource==null) {
    synchronized (Resource.class) {
      if (resource==null) {
        resource = new Resource();
      }
    }
  }
  return resource;
}
```

Problem Reordering, mögliche Ausführungsreihenfolge:
Speicher allokieren, Referenz zuweisen, Konstruktor aufrufen.

Anderer Thread denkt nun, es wäre eine gültige Ressource vorhanden und ruft u.U. nicht initialisiertes Objekt auf

Quelle: Java World, Wikipedia

Double-checked locking korrigiert

```
private volatile Resource resource = null;
public static Resource getResource() {
  if (resource==null) {
    synchronized (Resource.class) {
      if (resource==null) {
        resource = new Resource();
      }
    }
  }
  return resource;
}
```

Volatile: Erzwingt einheitliche Sicht und vollständige Ausführung der Zuweisung

Allerdings nicht sehr performant wg. Memory Barrier bei Zugriff auf initialisierte Variable

Double-checked locking performant

```
private static class ResourceHelper {
  static Resource resource = new Resource();
}

public static Resource getResource() {
  return ResourceHelper.resource;
}
```

WTF?

JVM sorgt dafür, dass statische Members (thread safe) nur 1x initialisiert werden

Klassen werden erst bei der ersten Verwendung geladen und damit auch erst dann initialisiert

Ergebnis: Lazy, thread-safe singleton

Zusammenfassend…

Im Hinterkopf behalten:

- Sperrkonzept (synchronized, ReadWriteLock, …) - Vorsicht beim Mischen!
- Es geht um Instanzen! (auf welcher Instanz werden Sperren ausgeführt)
- Es gibt außer `synchronized` in Methodendefinition keine „Sperre auf Instanz“, welche Methodenaufrufe unmittelbar verhindert
- Scope („atomare Klammer“): Was muss alles 'rein?
- Locking Order

Schwierig :-)

Some people, when confronted with a problem, think, "I know, I'll use threads," and then two they hav erpoblesms.

Some people, when confronted with a problem, think "I know, I'll use multithreading". Nothhw tpe yawrve o oblems.

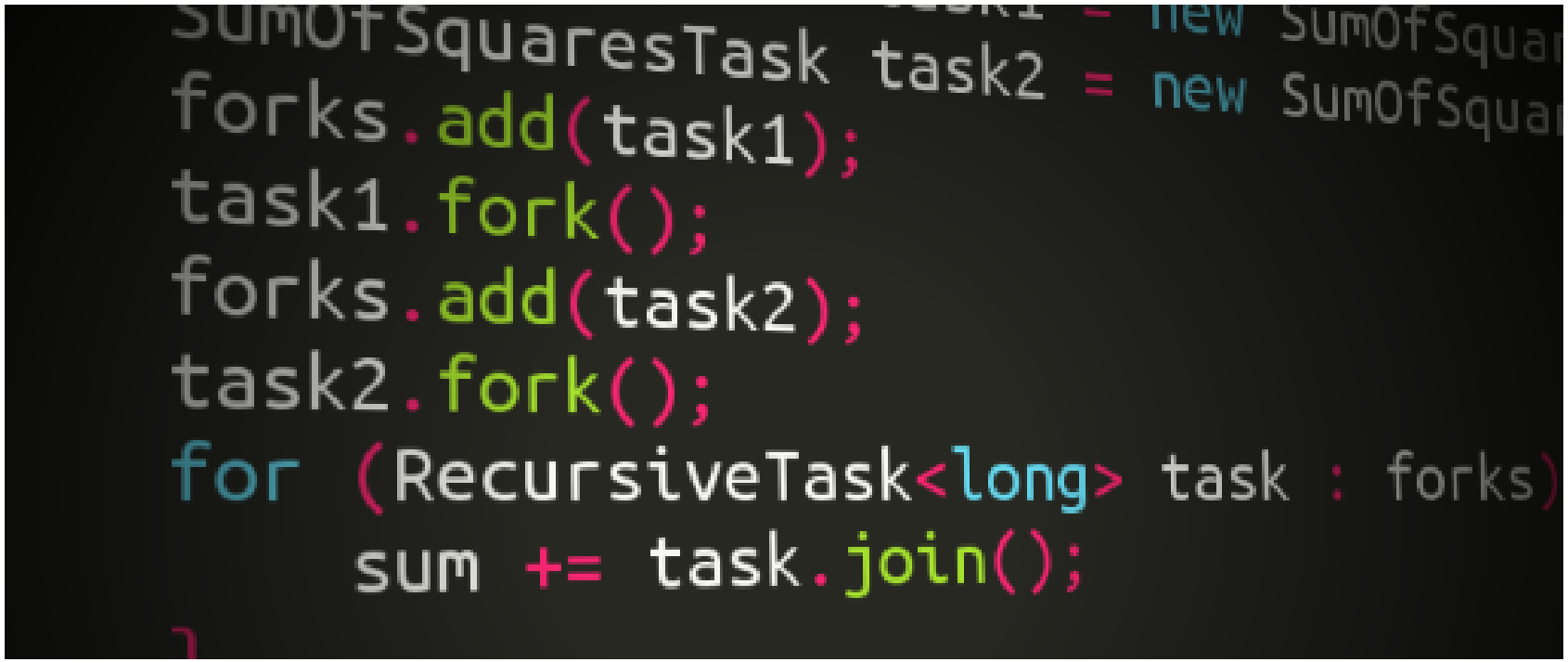
Some people, when confronted with a problem, think, "I know, I'll use mutexes." Now they have

Unit Testing?

Generell gesprochen: Fehlanzeige.

Diverse Projekte, allesamt inaktiv:

- MultithreadedTC: Einfügen von `waitForTick`, Weiterschalten im Unit Test
- IMUnit: Weiterentwicklung von MultithreadedTC
- cJUnit: Suche nach Interleavings, Durchprobieren aller Möglichkeiten



Parallelisierungs-Ansätze

Kriterien

Partitionierung, Granularität

Datenlokalität, Concurrency-Grad

Kommunikationsaufwand

Blockieren von Ressourcen

Mögliche Flaschenhälse

Datenpartitionierung

Teilen der Daten in parallel bearbeitbare Teilprobleme

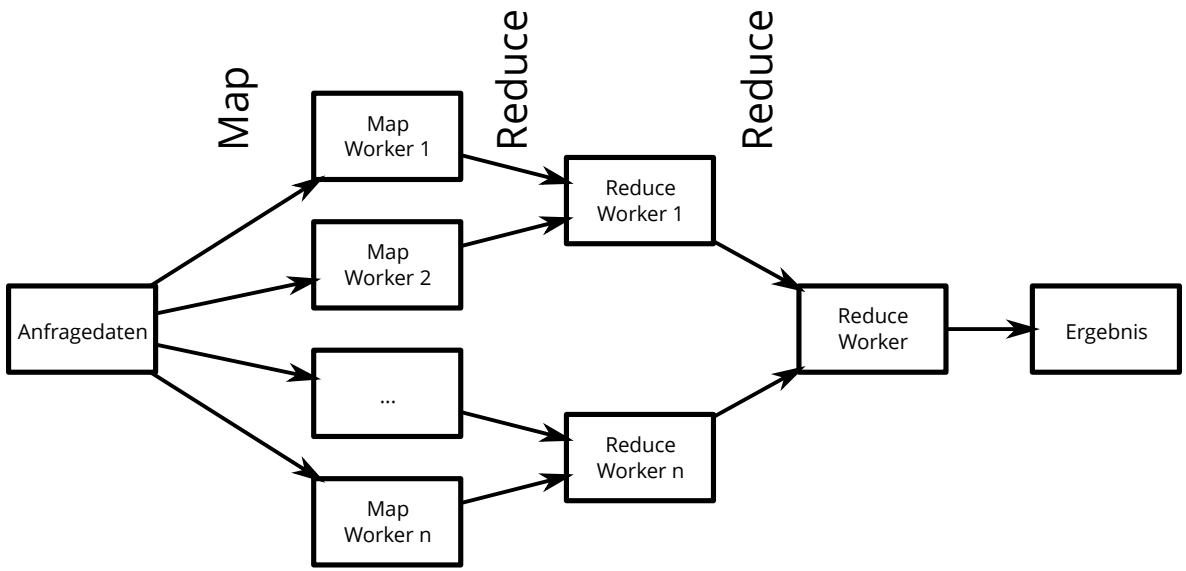
- Geometrische Zerlegung (*n* Teile gleicher Größe, etc.)
  Beispiel: Merge Sort
- Rekursive Zerlegung (z.B. Divide-and-Conquer-Algorithmen)
  Beispiel: Quick Sort
- Iterative Zerlegung (z.B. Breitensuche)

Taskpartitionierung

Parallele Verarbeitung von Arbeitsschritten

- ...im einfachsten Fall: Mehrere Useranfragen gleichzeitig
- Innerhalb einer Aufgabe: Parallele Bearbeitung unterschiedlicher Teilaufgaben auf derselben Datenmenge
  Beispiel: Unterschiedliche statistische Auswertungen derselben Daten
- Sequentielle Zerlegung einer Aufgabe, um unterschiedliche Partitionierungsstrategien anzuwenden
  Beispiel: Erst Daten sortieren (Datenpartitionierung), dann verschiedene Auswertungen parallel

Map-Reduce

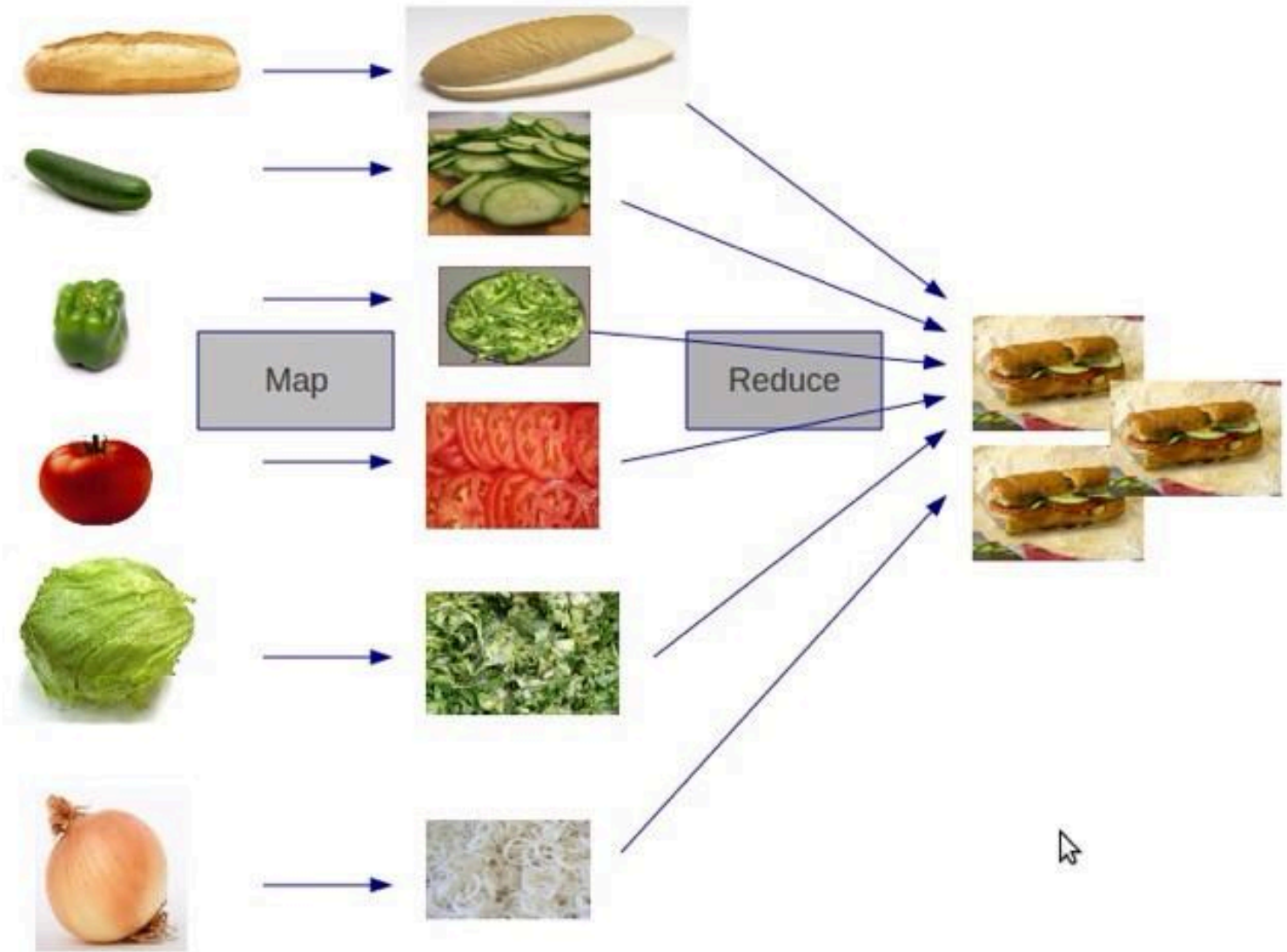


Map: Aufteilen (Partitionieren) einer Anfrage in Teilprobleme

Reduce: Zusammenfassen (Aggregieren) der Teilprobleme

Aggregation u.U. mehrstufig

Map-Reduce



Map-Reduce

Bekannt durch „Google-Paper“ (und -Patent)

„Wunderwaffe“ gegen riesige Datenmengen

...aber schon vergleichsweise banale Dinge lassen sich damit abbilden (z.B. Datenbank-Sharding)

Für Effizienz benötigt:

- Cachen von Teilergebnissen (Map-Schritt)
- Map-Schritt muss Datenmenge drastisch reduzieren
- Hohe Datenlokalität: Map-Worker muss möglichst autark arbeiten können

Grenzen der Skalierbarkeit

Parallelisierung = Nutzen von mehr Ressourcen gleichzeitig

Kann man die Parallelisierung beliebig weit treiben?

Teile eines Programms sind oft nicht parallelisierbar

- Synchronisation
- Verteilen von Aufgaben
- Einsammeln von Ergebnissen
- Zusammenfassen von Teilergebnissen
- ...

STAND BACK

I'M GOING TO TRY

SCIENCE

Amdahl's Law

Sei $T(x)$ die Laufzeit auf X parallelen Einheiten

Beschleunigung (Speedup) auf N Einheiten (im Optimalfall):
$S(N) = \frac{T(1)}{T(N)}$

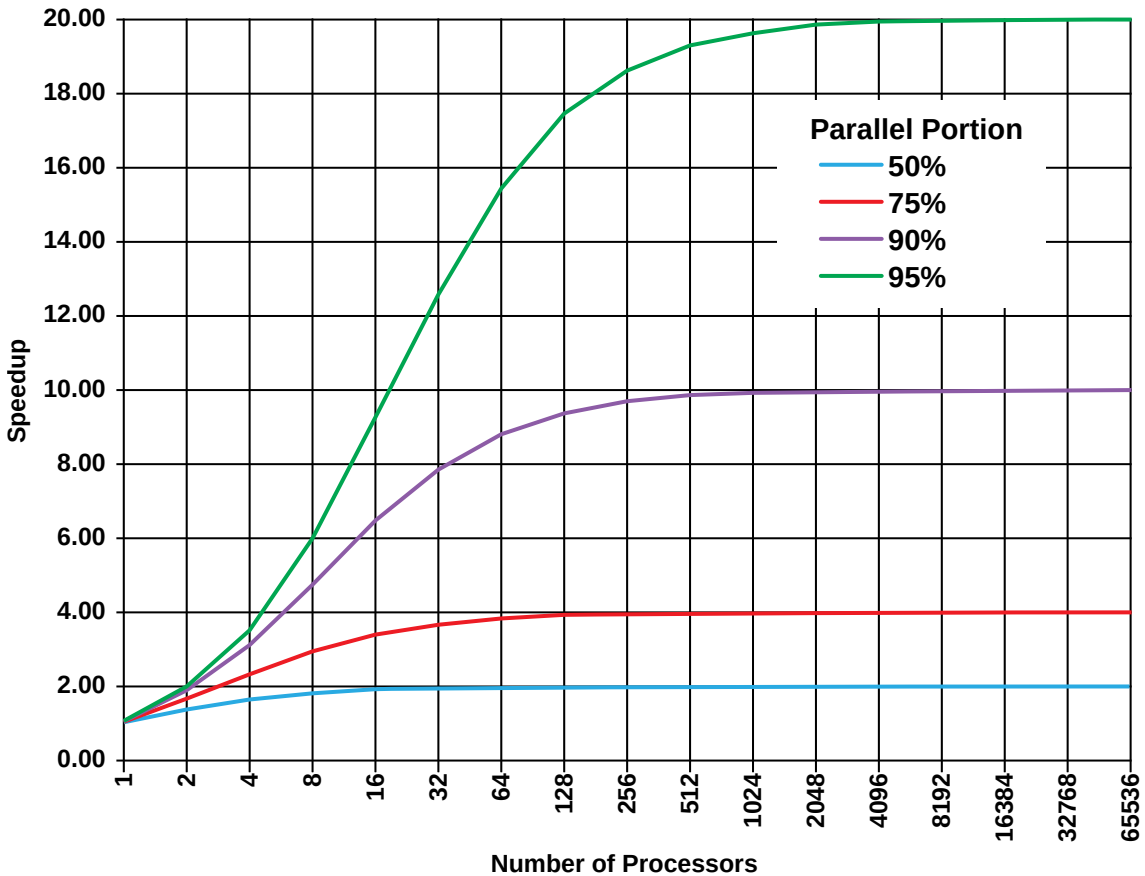
Nun habe ein Programm einen parallelen Anteil von P (0..1)

Parallelisierbarer Anteil: $\frac{P}{N}$

Sequentieller Anteil: $1 - P$

$$S(N)_{max} \leq \frac{1}{(1-P) + \frac{P}{N}}$$

Amdahl's Law

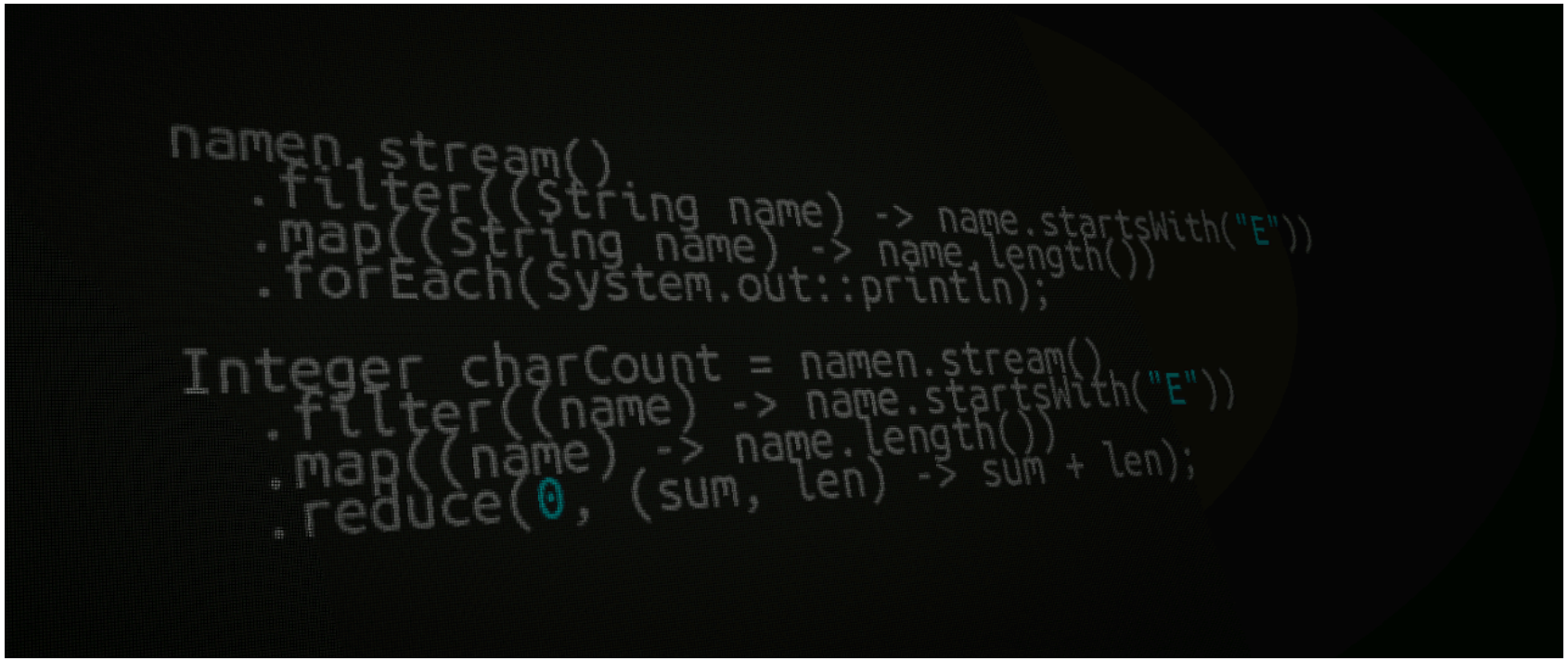

20.00
18.00
16.00
14.00
12.00
10.00
8.00
6.00
4.00
2.00
0.00
Speedup
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
Number of Processors
Parallel Portion
50%
75%
90%
95%

Fazit aus Amdahl's Law

„Unendliches Skalieren“ nicht möglich

Überlegen, welche Programmteile sich überhaupt parallelisieren lassen

Keine „premature optimization“



Exkurs: Lambda Expressions und Streams

Motivation

Methodenaufruf ≈ „Code hier einfügen“

Was tun mit „Code drumherum einfügen“?

- Das „Innere“ eines Threads
- Sortieralgorithmus: Vergleichsfunktion
- GUI-Callbacks (Event Listeners)
- „Callback-Kommunikation“ nach außen, z.B. Fortschrittsanzeige
- …

Motivation

Einige solcher Situationen: Als Sprachfeature in Java

- `synchronized`-Blöcke
- auto-closeable mit `try()`

Rest: Interfaces und meist anonymous inner classes

Beispiel: Fortschrittsanzeige

```
public interface ProgressUpdater {
  public void updateProgress(int percentComplete);
}

public void doSomething(ProgressUpdater progress) {
  progress.updateProgress(0);
  for (int i=0; i<10; i++) {
    // calculate…
  progress.updateProgress(i*10);
  }
}

doSomething(new ProgressUpdater() {
  @Override public void updateProgress(int percentComplete) {
    System.out.println(percentComplete + "% erledigt");
  }
});
```

Vorteil: Ausgabe für Kommandozeile, GUI, Unittest, … passend austauschbar

Lambda: Ersatz für anonymous inner class

Ohne Lambda:

```
doSomething(new ProgressUpdater() {
  @Override public void updateProgress(int percentComplete) {
    System.out.println(percentComplete + "% erledigt");
  }
});
```

Mit Lambda:

```
doSomething((int percentComplete) ->
  System.out.println(percentComplete + "% erledigt"));
```

Syntatic Sugar für eine anonyme innere Klasse mit einer Methode.

Nur 1 Statement: Geschweifte Klammern können wegfallen.

Typen-Inferenz

```
@FunctionalInterface
public interface ProgressUpdater {
  public void updateProgress(int percentComplete);
}
```

Ist das Interface entsprechend annotiert, werden die Typen automatisch aus der Methodensignatur übernommen.

Aus...

```
doSomething((int percentComplete) ->
  System.out.println(percentComplete + "% erledigt"));
```

...wird damit:

```
doSomething((percentComplete) ->
  System.out.println(percentComplete + "% erledigt"));
```

Method references

Neue Syntax: Methode referenzieren durch Notation mit `::`.
Beispiel: Die Methode `println` von `System.out` wird mit `System.out::println` beschrieben.

Solche Methodenreferenzen können für Lambdas eingesetzt werden

Voraussetzung: Methodensignatur stimmt mit Signatur des Interfaces überein.

Möchte man im Beispiel nur die Zahl ausgeben:

```
doSomething((percentComplete) -> System.out.println(percentComplete));
```

...kann man schreiben als:

```
doSomething(System.out::println);
```

Beispiel-Einsatz: Threads und Runnable

Bisher:

```
Thread t = new Thread(new Runnable() {
  @Override public void run() {
    …
  }
});
```

Mit Lambdas:

```
Thread t = new Thread(() -> {
    …
});
```

Java 8 Streams

„Fluent Interface“ für Operationen auf Datenströmen

Verkettung von Operationen auf „Listen“ – analog zur Unix-Pipe

Collection-Interface (Wurzelklasse der Collections-API) hat neue Methode `stream()`

Stream (→ intermediate → ...) → terminal

- intermediate: Liefert nach Operation wieder einen Stream. Streams sind *lazy*, d.h. die Operationen werden erst durch Konsumieren des Streams ausgeführt
- terminal: Bildet das Endergebnis, konsumiert Stream

Ein Stream kann nur 1x konsumiert werden (Strom-Semantik)

Beispieldaten:

```
class Name {
    final String firstName, lastName;
    public Name(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
}

final List<Name> namen = Arrays.asList(new Name("Hugo", "Hansel"),
        new Name("Achim", "Schneider"), new Name("Alfred", "Neumann"),
        new Name("Egon", "Schmidt"), new Name("Dau", "Jones"));
```

Wandeln in Liste mit Strings der Namen:

```
List<String> namensliste = namen.stream()
  .map((name) -> name.firstName + " " + name.lastName)
  .collect(Collectors.toList());
```

`map`: Umwandeln der Elemente von einem Typ in einen anderen

`collect`: Sammeln mittels Collector in neuen Container

Bearbeiten der ersten drei Elemente:

```
List<String> kurzeliste = namen.stream()
        .limit(3)
        .map((name) -> name.firstName + " " + name.lastName)
        .collect(Collectors.toList());
```

`limit`: „Kürzen“ des Streams auf die gewünschte Zahl (wichtig für „endlose“ Streams)

Alle Elemente Anzeigen

```
namen.stream()
        .filter((name) -> name.lastName.startsWith("S"))
        .map((name) -> name.lastName + ", " + name.firstName)
        .forEach(System.out::println);
```

`forEach`: Alternative zur `for`-Loop

Sortieren der Elemente:

```
namen.stream()
        .sorted((o1, o2) -> o1.lastName.compareTo(o2.lastName))
        .map((name) -> name.lastName + ", " + name.firstName)
        .forEach(System.out::println);
```

`sorted`: Sortieren (verschiedene überladene Varianten)

Iterator als Terminalelement:

```
Iterator<String> it = namen.stream()
        .map((name) -> name.firstName + " " + name.lastName)
        .iterator();
it.forEachRemaining(System.out::println);
```

Stream wird erst beim Traversieren des Iterators konsumiert

Generieren einer Map:

```
Map<String, Name> nameMap = namen.stream()
        .collect(Collectors.toMap(
                (name) -> name.lastName,
                (name) -> name
        ));
```

Collector benötigt zwei Closures (für Key und Value)

Streams auf Maps:

```
nameMap.entrySet().stream()
        .map((entry) -> entry.getValue().firstName)
        .forEach(System.out::println);
```

Map besitzt direkt keine `stream`-Methode, es muss auf `keySet`, `valueSet` oder `entrySet` gearbeitet werden.

Parallelisierung

Parallele Stream-Ausführung mittels `parallelStream()` möglich

„Stückeln“ des Streams in Teilstreams, welche von mehreren Threads ausgeführt werden

Benutzt wird der aktuelle Thread sowie Threads aus `ForkJoinPool.commonPool`

Wichtig: Nicht zwingend schneller!

Overhead durch Verteilung + Zusammenführung

Belastung/Überlastung des Threadpools

Paralleles Sortieren:

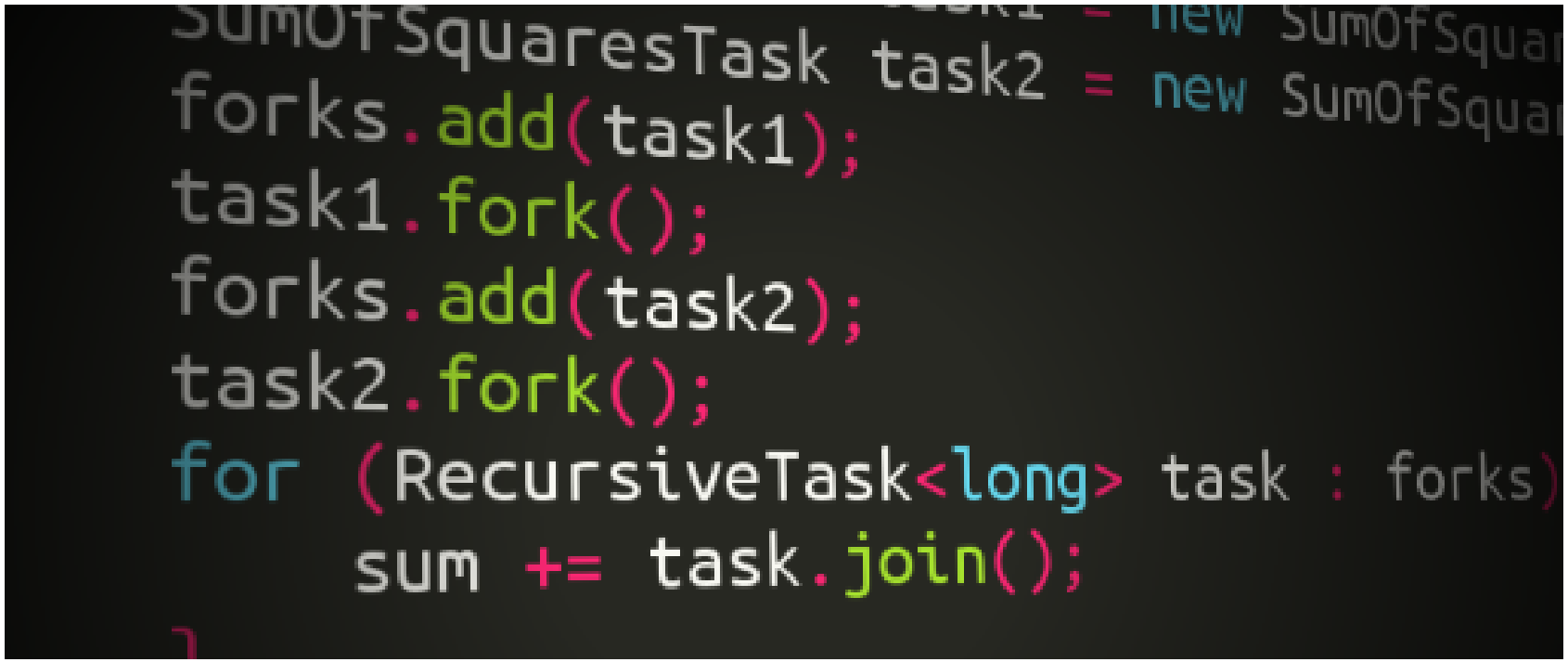
```
List<Integer> zahlen = Arrays.asList(20,19,18,…,3,2,1);

zahlen.parallelStream()
        .sorted()
        .forEach(System.out::println);
```

…ergibt Durcheinander!

Wenn Reihenfolge des Endergebnisses relevant, `forEachOrdered` verwenden:

```
zahlen.parallelStream()
        .sorted()
        .forEachOrdered(System.out::println);
```

`collect` und `iterator` berücksichtigen Ordnung



Threadpools

Ressourcenproblem…

Mit steigender Anzahl Threads:

- Rechenzeit muss immer weiter aufgeteilt werden
- Folge: Jeder Einzelthread braucht immer mehr Zeit
- Overhead durch Thread-Wechsel
- Viele Threads machen potentiell Cache-Effekte kaputt

Anlegen von Threads: Gewisser Overhead!

Längst nicht so schlimm wie beim Anlegen von Prozessen…

…aber tödlich z.B. bei Ansätzen wie „ein neuer Thread pro Request“ in einem Webserver

Threadpools

Entkopplung zwischen auszuführenden Tasks und ausführenden Threads

Verwaltet Worker-Threads, denen Arbeit zugeteilt werden kann

Nach Abschluß der Aufgabe wird dem Thread vom Threadpool neue Arbeit zugeteilt

Queue für wartende Aufträge

Threadpools

Einfaches Modell: Fixe Anzahl an Threads

Verfeinertes Modell:

- Mindestanzahl an Threads werden vorgehalten
- Bis zu einer Maximalzahl werden bei Bedarf zusätzliche erzeugt
- Nach gewisser Idle-Time werden Threads beendet (sofern die Mindestanzahl nicht unterschritten wird)

Threadpools in Java

Erzeugen eines einfachen Threadpools fixer Größe: `Executors.newFixedThreadPool(size)`

`ExecutorInterface` generische Schnittstelle

Aktionen ausführen bzw. in die Warteschlange einreihen: `submit(task)`

- Interface `Runnable` für Tasks ohne Rückgabewert
- Interface `Callable` für Tasks mit Rückgabewert

Geordnetes Herunterfahren mit `shutdown`: Keine neuen Tasks, Blockieren bis Queue leer

Futures in Java

`Future`: Zugriffsmöglichkeit auf ein zukünftiges Ergebnis

Erlaubt einen Rückgabewert/Zugriff auf z.B. das Ergebnis eines Hintergrundprozesses

- `get()` liefert das Ergebnis (blockiert, falls dieses noch nicht vorliegt)
  - Variante: `get(timeout, unit)` mit Timeout
  - `ExecutionException` zeigt an, daß Methode mit einer Exception verlassen wurde
- `isDone` zeigt an, ob ein Ergebnis bereits vorliegt
- `cancel` bietet die Option, einen geschedulten oder bereits angestoßenen Vorgang abzubrechen (ist optional und selbst zu implementieren)

Programmieraufgabe: Futures

Ergänze das Java-Programmtemplate threadpool.zip, welches „aufwendige Berechnungen“ über einen Threadpool verteilt und die Ergebnisse über Futures einsammelt.

Erzeuge einen Threadpool statischer Größe mit 5 Threads (siehe `java.util.concurrent.Executors`)

Weise diesem 10 Aufgaben zu, welche jeweils 1 Sekunde warten und dann das Resultat 42 zurückliefern sollen

Summiere die Ergebnisse und gib dieses aus.

Was passiert, wenn eine abgeschickte Aufgabe mittels `cancel` abgebrochen wird?

Fragen zu Threadpools

Wie groß sollte ein Threadpool sein? (CPUs maximal ausreizen bei möglichst geringem Overhead durch Thread-Verwaltung)

Wie lange wartet eine Aufgabe in der Warteschlange, bis ihr ein Worker zugeteilt wird?

STAND BACK

I'M GOING TO TRY

SCIENCE

Abschätzung Threadpool-Größe

Ohne I/O oder andere blockierende Operationen: Threads nutzen CPU zu 100%, also 1 Thread pro Core.

Abschätzung Thread-Zahl: n Cores, p Prozent aktive Rechenzeit, $1 - p$ Prozent Warten auf I/O

$$n * \frac{1}{p}$$

Abschätzungen bei Warteschlangen

„Wie lange warte ich im Schnitt, bis ich Antwort bekomme?“

Wenn mehr Aufgaben zu erledigen sind als gleichzeitig bearbeitet werden können: Warteschlange (typischerweise)

Welchen Einfluß haben hierbei...

- Einfluß der Bearbeitungsstrategie (Warteschlange? Stapel? Priority Queue? ...)
- Statistische Verteilung der Anzahl der eingehenden Anfragen
- Statistische Verteilung der Bearbeitungszeit
- ...

Little's Law

Die durchschnittliche Anzahl Anfragender L in einem stabilen System ist gleich der durchschnittlichen effektiven Ankunftsrate λ (Lambda) multipliziert mit der durchschnittlichen Bearbeitungszeit W

$L = \lambda W$

Stabiles System: Durchschnittliche Exit-Rate ≥ λ (intuitiv: Das System muß langfristig mit der Zahl der Anfragen klarkommen)

Effektive Ankunftsrate: Ankunftsrate der Anfragen, die tatsächlich zur Bearbeitung kommen (kein Abbruch durch Timeout o.ä.)

Little's Law - Implikationen

Aussagen „über eine gewisse Zeit“, keine Betrachtung von kurzfristigen Effekten (intuitiv: Ausreißer durch die Verteilung „mitteln sich heraus“)

Interessante Aspekte:

- unabhängig von der stochastischen Verteilung von λ und W
- unabhängig vom Abarbeitungs-Modus (LIFO, FIFO, ...)
- gilt für ein Gesamtsystem (z.B. App-Server) genauso wie für einen Auschnitt daraus (z.B. einzelner Thread eines Webservers, Mitglied eines Clusters, etc.)

Die Intuition entspricht glücklicherweise den mathematischen Tatsachen :-)

Threadpools mit Work Stealing

Problem, wenn Threads aus Threadpool blockierend auf Ergebnisse warten, die ebenfalls mit dem selben Threadpool berechnet werden

Beispiel Divide and Conquer:

- Aufteilen in Teilaufgaben, erstellen neuer Tasks für diese Teilaufgaben
- Warten auf Teilaufgaben = Blockieren des Threads
- Deadlock, sobald Threadpool erschöpft

Lösung: Temporäre „Rückgabe“ des Threads an den Threadpool, bis das Ergebnis vorliegt

ForkJoinPool (Auszug)

Bringt `ForkJoinTasks` zur Ausführung.

`ForkJoinTask`: Analog zu `Runnable`/`Callable`, nur mit Zugriff auf Pool (weitere Tasks starten). Implementierungen: `RecursiveAction` (analog `Runnable`), `RecursiveTask` (analog `Callable`), `CountedCompleter` (seit Java 8).

Methode `compute`: Überschreiben mit eigener Berechnung

Innerhalb von `compute`:

- Neue Instanz *`i`* für Teilberechnung anlegen
- `i.fork()`: Teilberechnung wird ausgeführt/in Warteschlange eingereiht
- `i.join()`: Wartet auf Teilergebnis

Codebeispiel: Verzeichnisgröße

Eine Javaanwendung, welche die Anzahl der Dateien und den von ihnen belegten Plattenplatz bestimmt.

- Bestimmung die Daten von Unterverzeichnissen durch Rekursion
- Parallelisierung dieser Rekursion durch einen Thread Pool mit Work Stealing

Java 8 CompletionStage

„Future on Steroids“ → 50 Methoden!

Erlaubt Verkettung von Futures - asynchrone Ausführung, wenn anderer asynchroner Task abgeschlossen ist

Benötigt zur Ausführung expliziten Executor, oder benutzt `ForkJoinPool.commonPool()`

Task ohne Rückgabewert ausführen:
`CompletableFuture.runAsync(Runnable r)`

Task mit Rückgabewert ausführen:
`CompletableFuture.supplyAsync(Supplier<U> s)`

Blockierendes Warten auf Ergebnis: `join()`

Einfache Verkettung

`thenApply(...)` führt Lamba aus, nachdem Future erfüllt wurde. Lambda-Eingabeparameter = Ausgabe der vorigen CompletionStage

`thenAccept(...)` wie `thenApply`, nur ohne Rückgabewert

`thenRun(...)` wie `thenAccept`, nur zusätzlich ohne Eingabewert

Varianten `thenApplyAsync`, `...Async`: Ausführung nicht im selben Thread, sondern ein anderer (via Threadpool, ggfs. sogar anderer Executor)

`exceptionally(...)` Lambda wird gerufen, falls CompletionStage eine Exception wirft

Verkettung mehrerer Werte

`thenCompose(…)` macht aus einem `CompletableFuture<CompletableFuture<…>>` wieder ein `CompletableFuture<…>`

`thenCombine(future2, (a,b) -> …)` kombiniert zwei Futures, ruft das Lambda mit beiden Ergebnissen, sobald beide vorliegen

`applyToEither(future2, …)` Lambda wird ausgeführt, sobald *eines* der beiden Futures erfüllt ist

`CompletableFuture.allOf(…)`, `CompletableFuture.anyOf(…)`: Erzeugt ein Future, welches erfüllt ist, wenn alle/eines der x Futures erfüllt sind

Codebeispiel: Map-Reduce

Java-Anwendung soll drei Kenngrößen (Anzahl Studierende, Durchschnittsalter, maximale Zeichenlänge des Nachnamens) einer Namensdatenbank bestimmen.

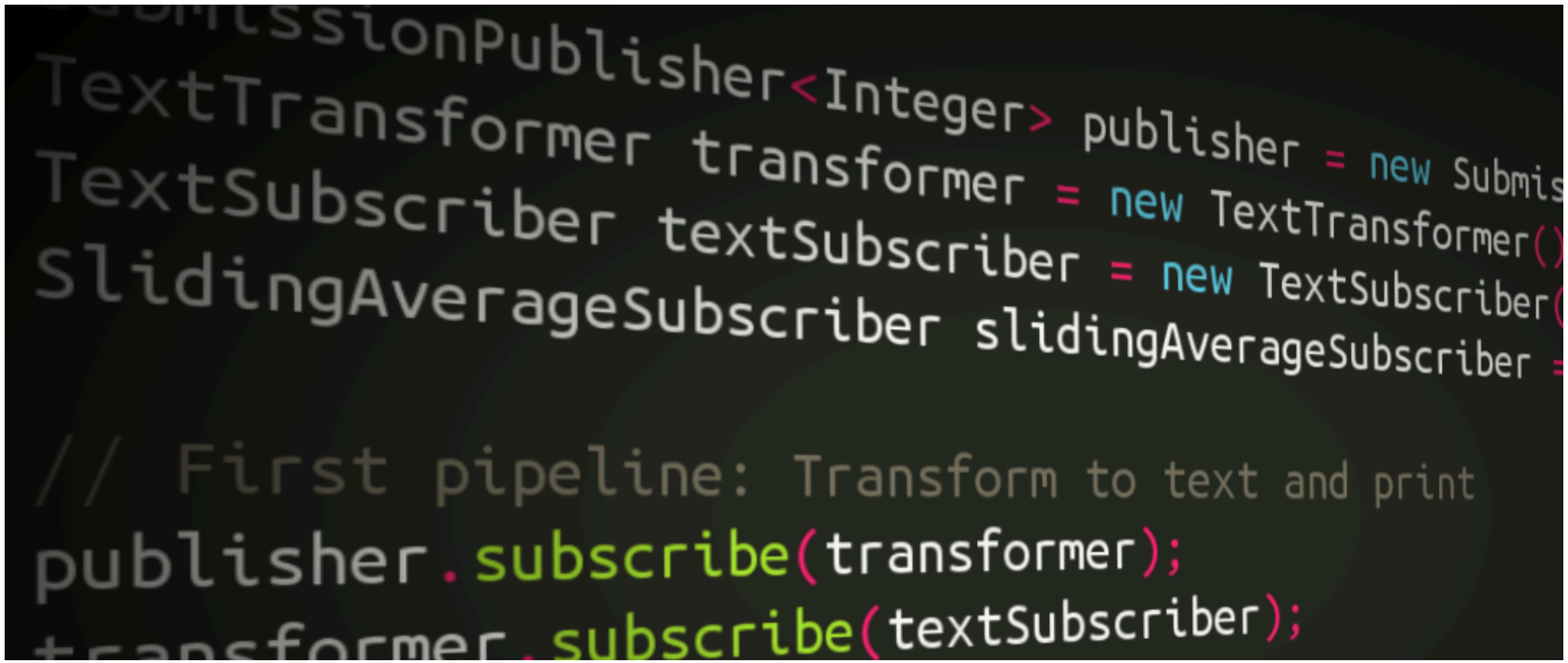
- 4 Threads sollen parallel das Namensarchiv abfragen
- Die Ergebnisse sollen in einer Liste gesammelt werden
- Auf dieser Liste sollen parallel die Auswertungen erfolgen

Programmieraufgabe: Verzeichnisgröße mit CompletionStages

Ergänze das Java-Programmtemplate dirsize2.zip, welches die Anzahl der Dateien und den von ihnen belegten Plattenplatz bestimmt.

- Bestimme die Daten von Unterverzeichnissen durch Rekursion
- Parallelisiere diese Rekursion mit Hilfe von `CompletableFutures`

Für die Rekursion ist die mitgelieferte Methode `sequence` hilfreich.



Reactive Streams

Reactive Streams

CompletableFuture: Erlaubte einmalige Datenpipeline. Reactive Streams: (Grob) Pendant für wiederverwendbare Pipelines bzw. kontinuierliche Datenströme

Asynchron, nicht-blockierend

Schutz vor Überlastung (Pull-Prinzip)

Basisinterface quasi-standardisiert (reactive-streams.org), verfügbar in verschiedenen Programmiersprachen (definiert Basisprimitive; Bibliotheken bieten höhere Funktionen)

Back Pressure (Pull-Prinzip)

Daten werden nicht von "vorne" "durchgedrückt" (Push), sondern "von hinten" "angezogen"

Folge: Kontrolle über den Datenfluss (wie viel, wie schnell) durch die Konsumenten

Auf den "Druck von Daten" wird ein "Gegendruck" ausgewirkt, so dass die Pipeline immer nur so viele Daten anfordert, wie sie auch konsumieren kann/will

Komponenten

Publisher: Produzent von neuen Daten

Subscriber: Konsument von Daten

Processor: Mittelteil einer Pipeline - ist sowohl Konsument als auch Produzent. In Java: Bedient beide Interfaces.

Subscription: Verknüpfungs-Informationen, welche in Subscriber beim Zusammenhängen der Pipeline gereicht werden

Publisher-Interface

`subscribe`: Fügt neuen Subscriber (Datenkonsument) hinzu

Neue Werte müssen allen Subscribern zugeführt werden

Java: SubmissionPublisher

Konkrete Implementierung: Hier können mittels `submit` neue Werte in die Pipeline eingefügt werden

Sorgt für Ausführung in einem Threadpool

Subscriber-Interface

`onSubscribe(subscription)`: Wenn eine neue Subscription eingerichtet wurde. Subscription lokal merken!

`onNext(item)`: Wird für jedes eingehende Item aufgerufen

`onComplete()`: Publisher beendet (regulär) die Kommunikation

`onError(throwable)`: Pulisher beendet Kommunikation aufgrund von Exception

Subscriber: Abruf von Daten

Pull-Prinzip: Subscriber fordert Daten an: `subscription.request(count)` Liefert (höchstens) count Items

Wichtig: Nach `onSubscribe` und `onNext` entsprechend `subscription.request` aufrufen! Sonst werden keine Daten nachgereicht

Je nach Implementierung: Puffer zwischen Publisher und Subscriber

Programmieraufgabe: Reactive Map-Reduce

Ergänze das Java-Programmtemplate reactivemapreduce.zip, welches drei Kenngrößen (Anzahl Studierende, Durchschnittsalter, maximale Zeichenlänge des Nachnamens) einer Namensdatenbank bestimmt.

Die Daten sollen im Sekundentakt dem Stream übergeben werden. Bei jeder Berechnung soll der aktuelle Stand angezeigt werden.