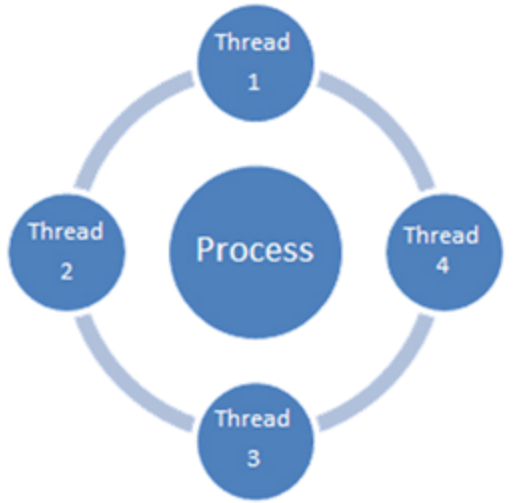
Orhan Konak
HTW Berlin, 25.10.2024

MULTITHREADING IN JAVA



Java Multithreading

Agenda

- Motivation: Was ist Multithreading?
- Klasse Thread und Interface Runnable
- Methode *join* und Parallelisierung von Algorithmen
- Daemon und User Threads
- Synchronisierung mit *synchronized*
- Erzeuger / Verbraucher-Problem und die Methoden *wait*, *notify*, *notifyAll*
- Zustände eines Java-Threads

Agenda

- Motivation: Was ist Multithreading?
- Klasse Thread und Interface Runnable
- Methode *join* und Parallelisierung von Algorithmen
- Daemon und User Threads
- Synchronisierung mit *synchronized*
- Erzeuger / Verbraucher-Problem und die Methoden *wait, notify, notifyAll*
- Zustände eines Java-Threads

Motivation

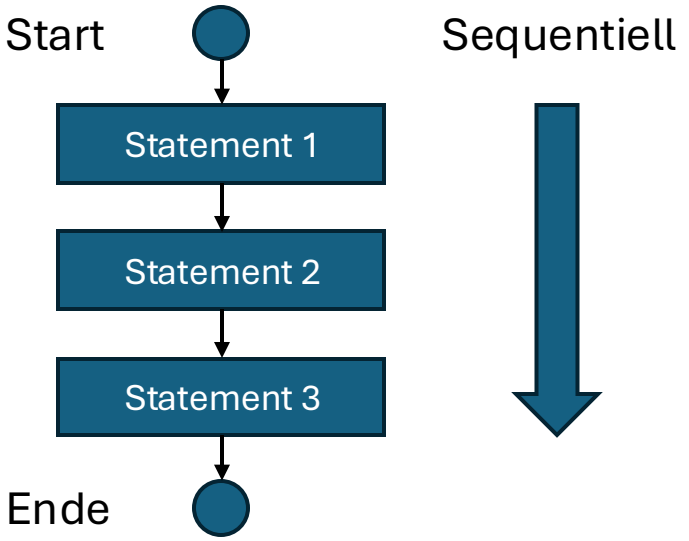
Sequentielle Programmierung:

- Lineare Abfolge von Anweisungen
- Operationen werden nacheinander ausgeführt
- Kein Überlappen von Aufgaben – eine Aufgabe wird erst beendet, bevor die nächste beginnt
- Leicht verständlich und einfach zu debuggen

→ Vorteil: Gut geeignet für einfache Programme und Single-Core-Systeme

→ Nachteile bei rechenintensiven oder zeitkritischen Aufgaben, die mehr Zeit beanspruchen

```java
// Sequentielles Programm
class SequentialExample {
  public static void main(String[] args) {

    // 1. Lade Datei
    load();
    // 2. Verarbeite Datei
    process();
    // 3. Speichere Resultat
    save();
    // 4. Zeige Ergebnis an
    display();
  }
}
```



Multitasking

- **Multitasking** ein Prozess ist, bei dem mehrere Aufgaben gleichzeitig ausgeführt warden, z.B.:
  - Tippen in Word und gleichzeitiges starten einer Musik-App → beide Aufgaben werden als Prozesse bezeichnet
  - Tippen in Word + gleichzeitige Rechtschreibprüfung → Word unterteilt in Unterprozesse → Threads

- **Multitasking wird auf zwei Arten erreicht**:
  - **Multiprocessing** : Prozessbasiertes Multitasking ist ein schwergewichtiger Prozess und belegt verschiedene Adressräume im Speicher → beim Wechsel von einem Prozess zu einem anderen wird einige Zeit benötigt, auch wenn diese sehr kurz ist
  - **Multithreading** : Threadbasiertes Multitasking ist ein einfacher Prozess und belegt denselben Adressraum → Kommunikationskosten beim Umschalten gering

Motivation

Ein Koch führt eine einzige Aufgabe aus

Single Threading

Derselbe Koch mit mehreren Armen führt verschiedene Aufgaben gleichzeitig aus

Multithreading

Der Koch wird von zwei weiteren Köchen unterstützt, und jeder arbeitet an einer eigenen Aufgabe

Multiprocessing

Definitionen von Begriffen

PROZESSOR

Ein **Prozessor** (CPU) ist die zentrale Recheneinheit eines Computers, die Anweisungen ausführt

PROZESS

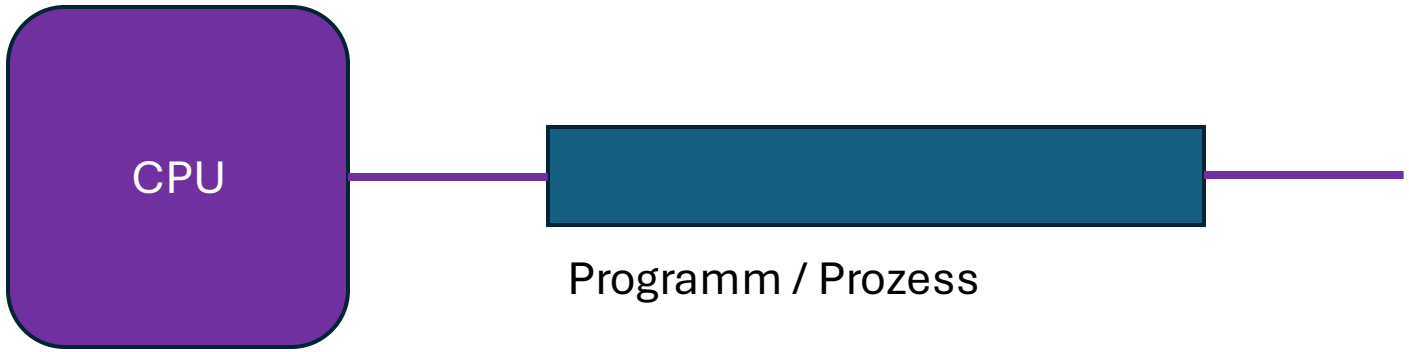
Ein **Prozess** ist ein laufendes Programm, das Ressourcen wie Speicher und CPU-Zeit nutzt

THREAD

Ein **Thread** ist die kleinste Ausführungseinheit innerhalb eines Prozesses, die parallel zu anderen Threads laufen kann, um Aufgaben effizienter zu bearbeiten

Motivation

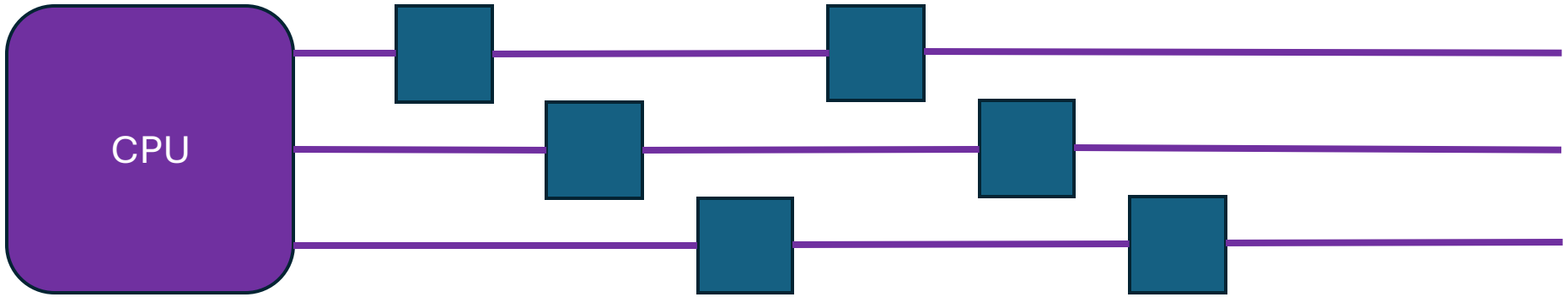
Singletasking in der frühen Datenverarbeitung



Ein CPU / Computer kann nur
ein Programm (Prozess)
gleichzeitig ausführen

Motivation

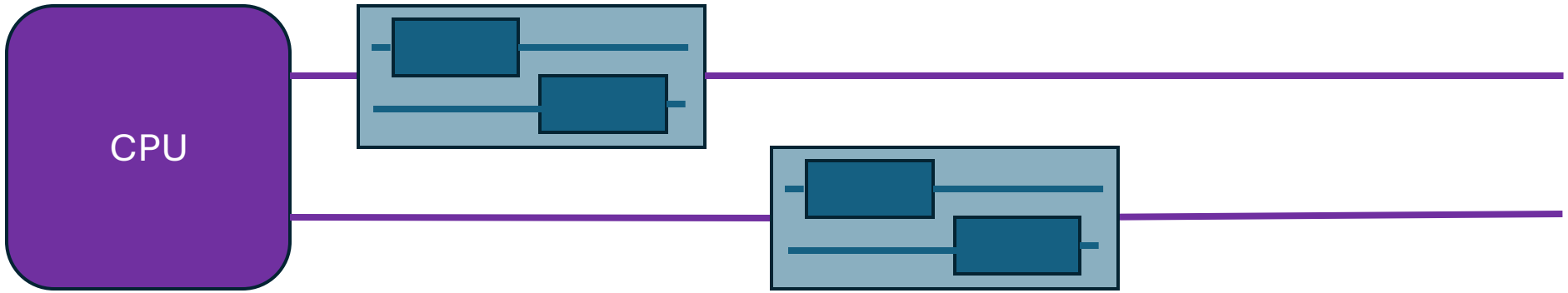
Multitasking in der frühen Datenverarbeitung



Ein CPU / Computer kann mehrere Programme (Prozess) gleichzeitig ausführen – in dem es zwischen den einzelnen Programmen für kurze Zeit hin- und herschaltet

Motivation

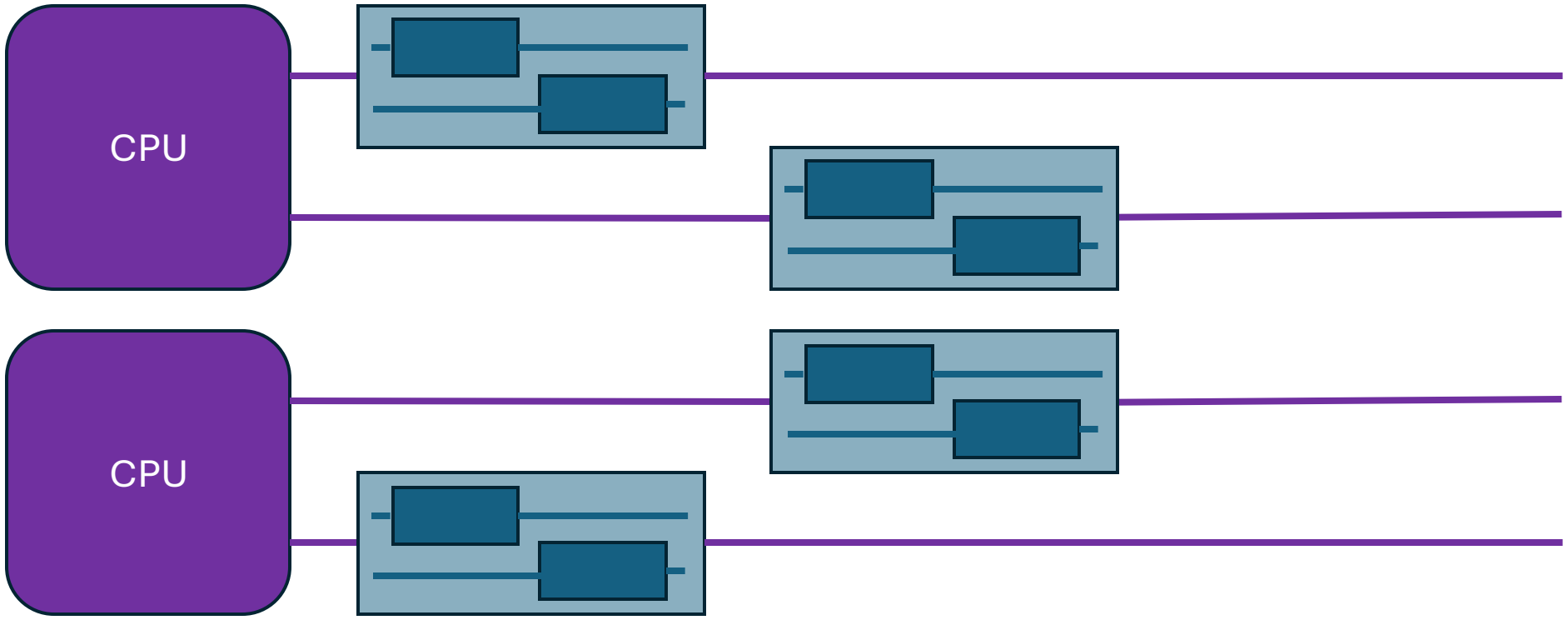
Multithreading



Ein CPU / Computer kann mehrere Programme (Prozesse) gleichzeitig ausführen – mit mehreren Threads

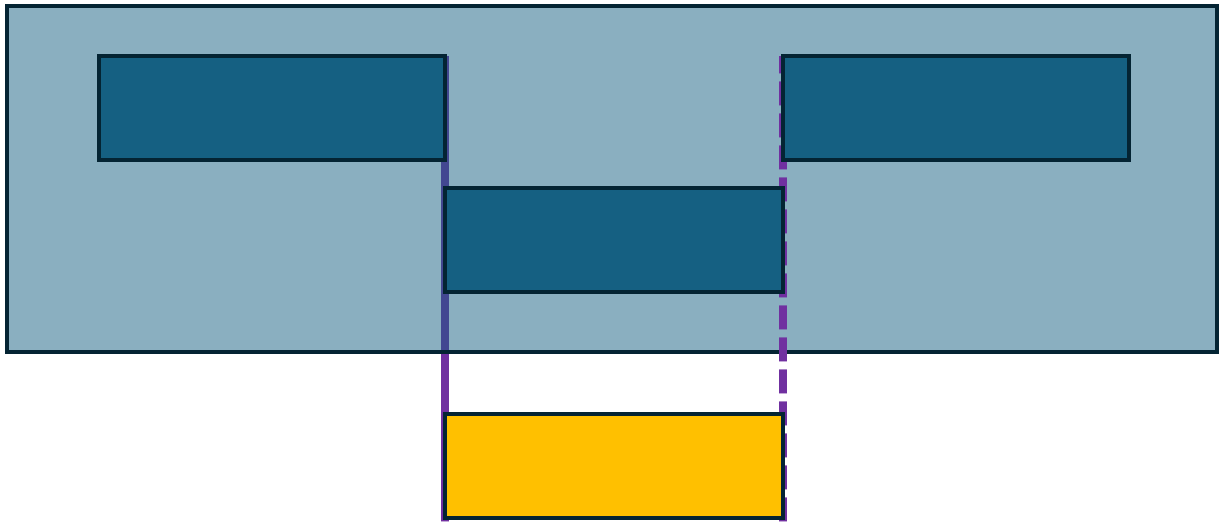
Motivation

Multithreading mit mehreren CPUs



Motivation

Warum Multithreading?



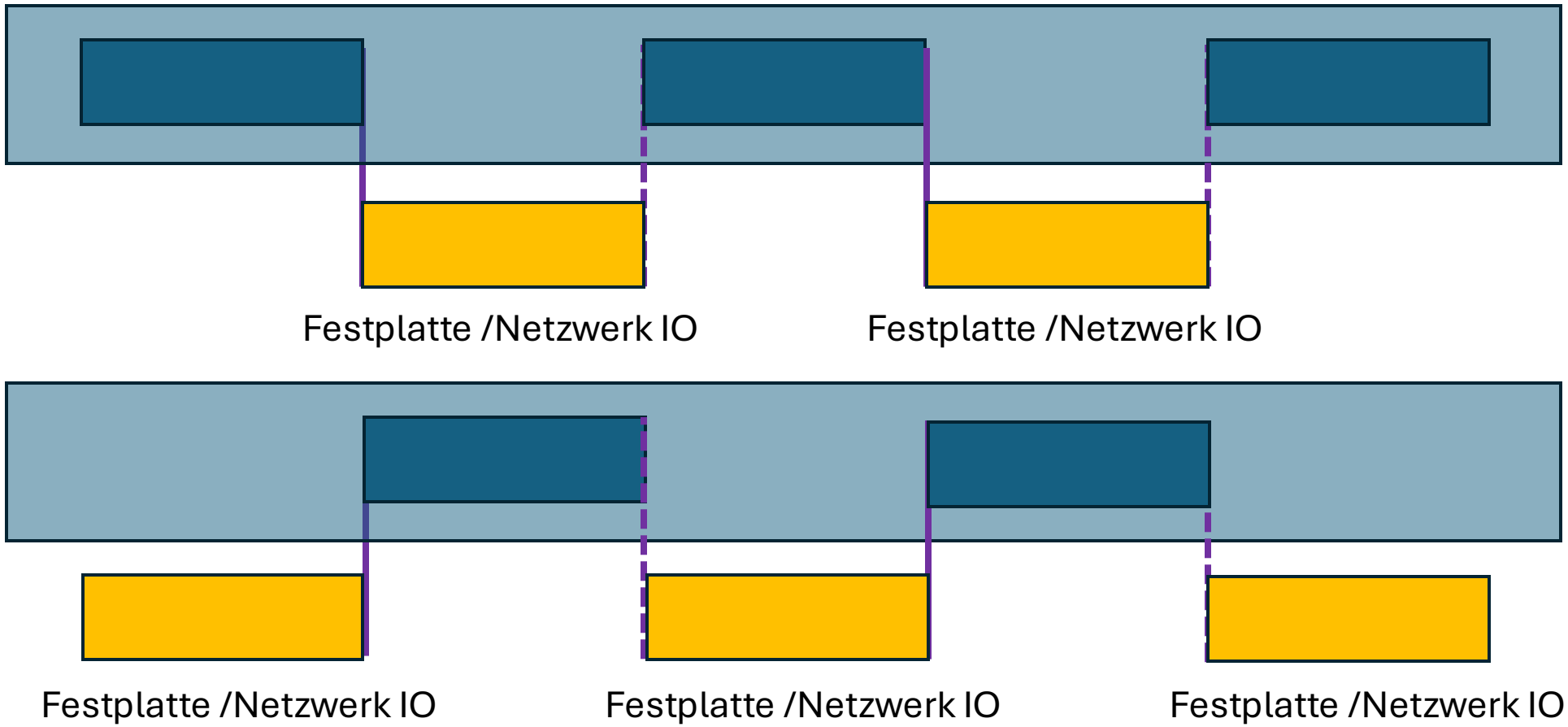
von der
Festplatte oder
Netzwerk lesen

- bessere CPU-Auslastung

Motivation

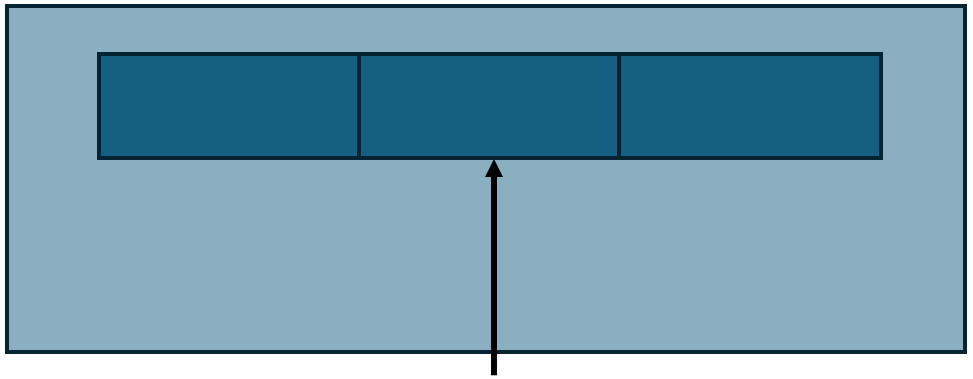
Warum Multithreading?

- bessere IO Auslastung



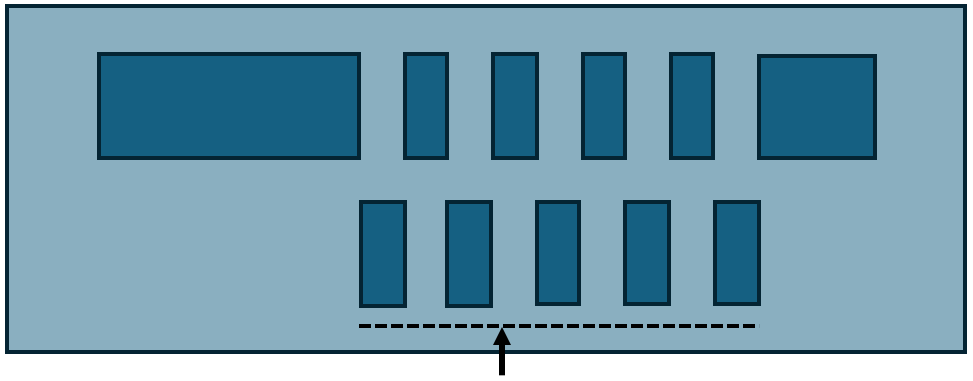
Motivation

Warum Multithreading?

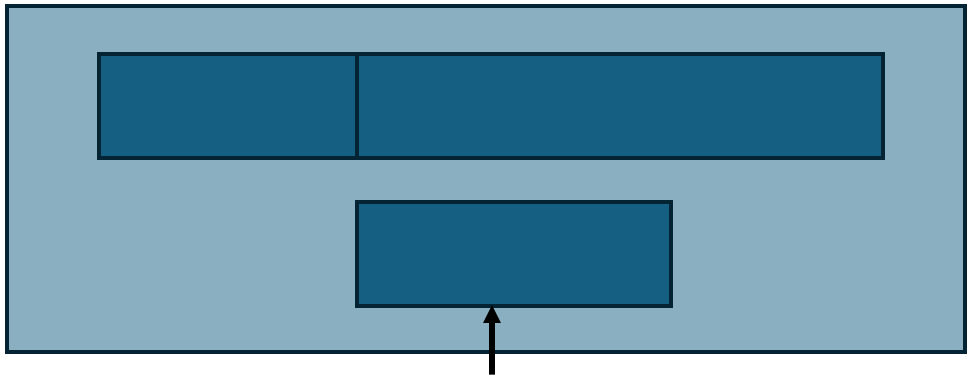


schwere, langwierige Aufgabe

- Höhere Anwendungs-Reaktionsfähigkeit



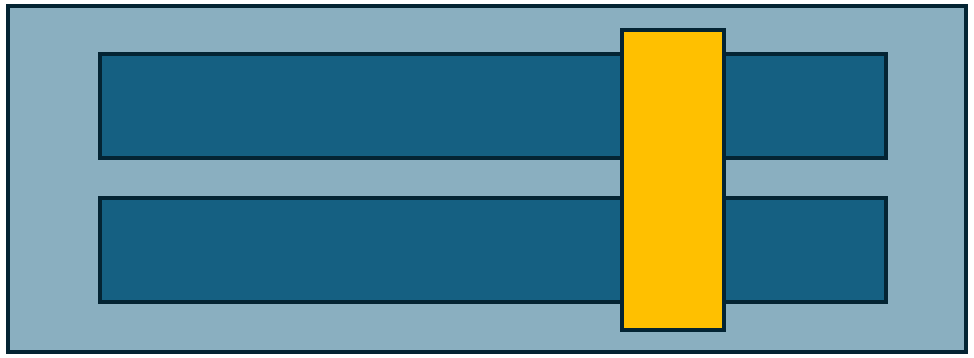
schwere, langwierige Aufgabe in einem separatem Thread

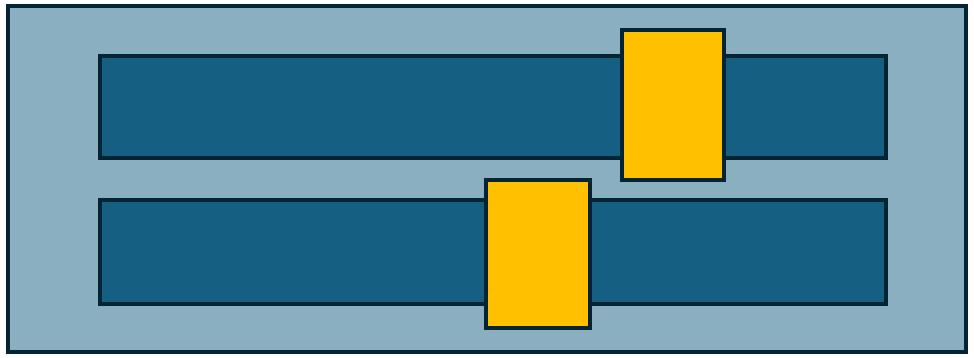


schwere, langwierige Aufgabe

Motivation

Warum Multithreading?





- **Bessere Ressourcenausnutzung**
  Mehrere Threads können gleichzeitig CPU-Kerne nutzen, wodurch die Hardware effizienter arbeitet
- **Parallele Ausführung**
  Aufgaben können parallel bearbeitet werden, was die Gesamtleistung verbessert
- **Schnellere Reaktion**
  In Anwendungen mit Benutzerschnittstellen ermöglicht Multithreading, dass die UI nicht einfriert, während andere Aufgaben im Hintergrund laufen
- **Verkürzte Wartezeiten**
  Threads können auf I/O-Operationen (z.B. Dateizugriffe, Netzwerk) warten, ohne den gesamten Prozess zu blockieren
- **Aufgabenaufteilung**
  Komplexe Aufgaben können in kleinere, unabhängige Teilaufgaben zerlegt werden, die gleichzeitig bearbeitet werden

Agenda

- Motivation: Was ist Multithreading?
- Klasse Thread und Interface Runnable
- Methode *join* und Parallelisierung von Algorithmen
- Daemon und User Threads
- Synchronisierung mit *synchronized*
- Erzeuger / Verbraucher-Problem und die Methoden *wait, notify, notifyAll*
- Zustände eines Java-Threads

Threads in Java

- Es gibt in Java zwei Arten Threads zu erstellen

Klasse, die von Thread erbt

```
class MyThread extends Thread {
  @Override
  public void run() {
    // mein Code ...
  }
}
```

Interface Runnable

```
class MyRunnable implements Runnable {
  @Override
  public void run() {
    // mein Code: ...
  }
}
```

Zur Erinnerung: Interface

- **Interface** ist eine vollständig „abstrakte Klasse“ um verwandte Methoden mit leeren Körpern zu gruppieren
- Beispiel:

```
// interface
interface Animal {
  public void animalSound(); // interface Methode (hat keinen Body)
  public void run(); // interface Methode (does keinen Body)
}
```

- Um auf die Schnittstellenmethoden zugreifen zu können, muss die Schnittstelle von einer anderen Klasse mit dem Schlüsselwort **implements** (anstelle von **extends**) „implementiert“ (also sozusagen geerbt) werden.:

```
// Interface
interface Animal {
  public void animalSound();
  public void sleep();
}

class Pig implements Animal {
  public void animalSound() {
  System.out.println(”Das Schwein sagt: wee wee");
  }
  public void sleep() {
  System.out.println("Zzz");
  }
}

class Main {
  public static void main(String[] args) {
  Pig myPig = new Pig();
  myPig.animalSound();
  myPig.sleep();
  }
}
```

Einführendes Beispiel

- Definition einer Thread-Klasse
- Jeder Thread durchläuft den in der **run-Methode** definierten Code

- Es warden 5 Thread-Objekte definiert, die mit **start() nebenläufig** gestartet warden
- Der **start-Aufruf** eines Threads bewirkt seinen **run-Aufruf**
- Auch die main-Methode läuft als eigener Thread
- Damit laufen 6 Threads nebenläufig

```java
// Klasse, die von Thread erbt
class CountingThread extends Thread {
  private final String threadName;

  // Konstruktor für den Thread, um den Namen zu setzen
  public CountingThread(String name) {
    this.threadName = name;
  }

  @Override
  public void run() {
    // Iteriere von 0 bis 99
    for (int i = 0; i < 100; i++) {
      System.out.println(threadName + ": " + i);
    }
  }
}

public class MultiThreadExample {
  public static void main(String[] args) {
    // Erstelle 5 Threads
    for (int i = 1; i <= 5; i++) {
      // Jeder Thread bekommt einen eindeutigen Namen, z.B. "Thread-1", usw.
      CountingThread thread = new CountingThread("Thread-" + i);
      thread.start(); // Startet den Thread
    }
    System.out.println("Main ist fertig"); // Main Thread
  }
}
```

```
Main ist fertig
Thread-3: 0
Thread-3: 1
Thread-2: 0
Thread-5: 0
Thread-5: 1
Thread-4: 0
Thread-5: 2
...
Thread-1: 7
Thread-1: 8
Thread-1: 9
Thread-2: 5
Thread-3: 15
Thread-4: 6
Thread-4: 7
Thread-4: 8
Thread-4: 9
Thread-4: 10
Thread-5: 9
...
Thread-4: 20
Thread-4: 21
Thread-4: 22
Thread-4: 23
Thread-4: 24
Thread-1: 25
Thread-1: 26
Thread-1: 27
Thread-3: 84
...
```

Konsolenausgabe

Threads und Nebenläufigkeit

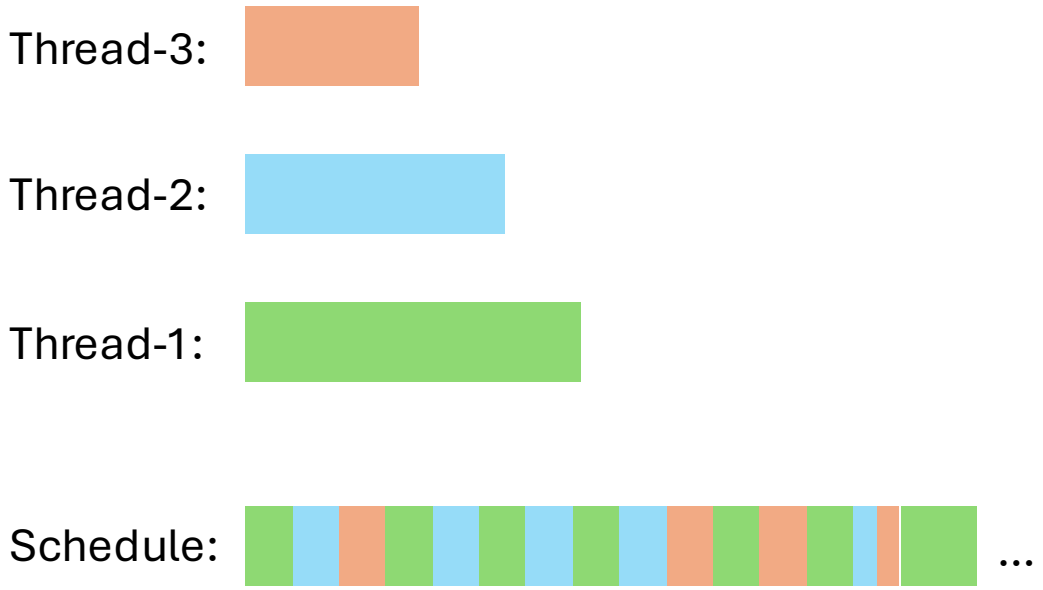
- Ein **Thread** ist eine Folge von Anweisungen, die nebenläufig ausgeführt werden können
- **Nebenläufigkeit (concurrency)** bedeutet:
  - (echte) **Parallelität**: die Threads laufen auf verschiedenen Prozessoren gleichzeitig ab
  - **Pseudo-Parallelität**: die Threads laufen auf genau einem Prozessor ab, wobei die Threads mit einer hohen Taktrate ständig gewechselt werden. Es wird eine Gleichzeitigkeit vorgetäuscht
- Jeder Thread besitzt einen **eigenen Laufzeitkeller** (Stack) für Methodenaufrufe und Speicherung lokaler Variablen
- Wichtig: die Threads können Zugriff auf gemeinsame Daten haben. Dazu muss der Zugriff geeignet synchronisiert warden (später)

Scheduling

- Ein Thread kann nur eine Operation ausführen, wenn ihm ein Prozessor (CPU) zur Ausführung zugeteilt worden ist
- Im Allgemeinen gibt es mehr Threads als CPUs
- Der **Scheduler** verwaltet die verfügbaren CPUs und teilt sie den Threads zu
- Bei verschiedenen Programmläufen kann diese Zuteilung verschieden aussehen!
- Es gibt verschiedene Strategien, nach denen sich Scheduler richten können (**Betriebssysteme**). Z.B.:
  - Zeitscheibenverfahren
  - Naives Verfahren

Zeitscheibenverfahren

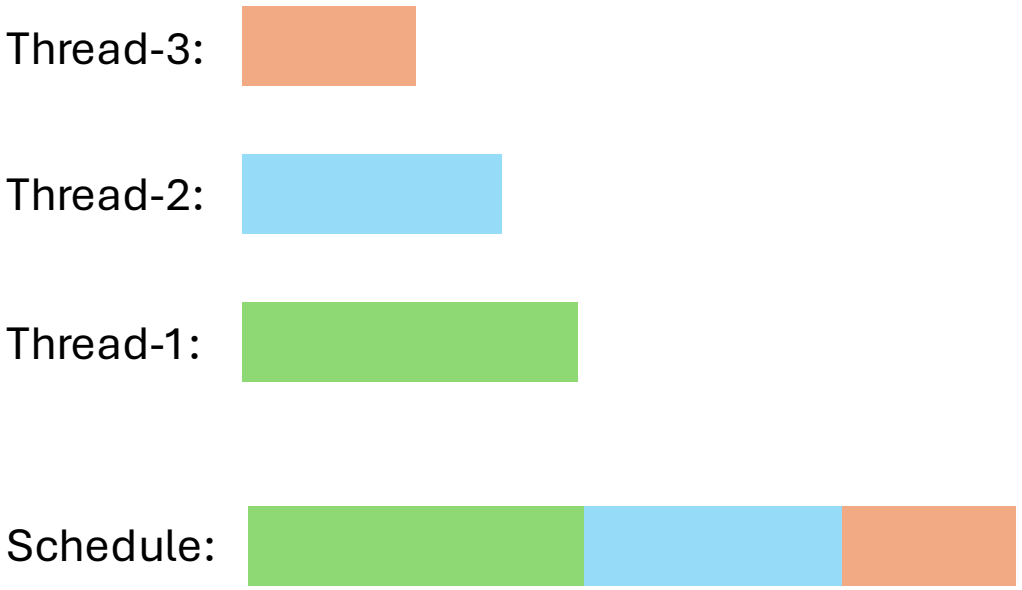
- Ein Thread erhält eine CPU nur für eine bestimmte Zeitspanne (Time Slice), in der er rechnen darf
- Danach wird er unterbrochen. Dann darf ein anderer



- Ein Zeitscheiben-Scheduler versucht, jeden Thread fair zu behandeln, d.h. ab und zu Rechenzeit zuzuordnen – egal, welche Threads sonst noch Rechenzeit beanspruchen
- Kein Thread hat jedoch Anspruch auf einen bestimmten Time-Slice
- Für den Programmierer sieht es so aus, als ob sämtliche Threads „echt“ parallel ausgeführt werden, d.h. jeder über eine eigene CPU verfügt

Naives Scheduling

- Erhält ein Thread eine CPU, darf er laufen, so lange er will …
- Gibt er die CPU wieder frei, darf ein anderer Thread arbeiten …



Beispiel Scheduling

```
public class SchedulingExample {
  public static void main(String[] args) {
    (new Start()).start();
    (new Start()).start();
    (new Start()).start();
    System.out.println("main is running ...");
    while (true);
  }
// Ende der Klasse Start
}

class Start extends Thread {
  public void run() {
  System.out.println("I'm running ...");
  while (true);
  }
}
```

Ausgabe

Naives Scheduling

main is running ...

main wird nie fertig → die anderen Threads erhalten keine Chance, sie **verhungern**

Faires Scheduling mit Zeitscheibenverfahren

```
I'm running ...
main is running ...
I'm running ...
I'm running ...
```

- Java legt nicht fest, wie intelligent der Scheduler ist
- Die aktuelle Implementierung unterstützt **faires** Scheduling

Erzeugung von Threads durch Erweiterung der Klasse Thread

- Die Klasse **Thread aus java.lang** wird erweitert, indem die Methode run() überschrieben wird
- Der Aufruf der Methode **start()** der Klasse Thread bewirkt, dass die Java Virtual Machine (JVM) die run-Methode als Thread nebenläufig ausführt

```
// Klasse, die von Thread erbt
class MyThread extends Thread {

  @Override
  public void run() {
    // mein Code: ...
  }
}
```

```
class ThreadExample {
  public static void main(String[] args) {
    Thread t = new MyThread();
      t.start(); // Startet den Thread
  }
}
```

Erzeugung von Threads durch Implementierung des Interface Runnable

- Das **Interface Runnable aus java.lang** enhält nur die Methode **run()**. Runnable ist ein **funktionales Interface**
- Das Interface Runnable wird durch eine eigene Runnable-Klasse implementiert
- Ein Thread lässt sich dann mit Hilfe eines **Thread-Konstruktors** definieren, indem ein Objekt der Runnable-Klasse als Parameter übergeben wird
- Das Thread-Objekt wird dann mit der Methode **start()** gestartet

```
@FunctionalInterface
interface Runnable {
  void run();
}
```

```
class MyRunnable implements Runnable {
  public void run() {
    // mein Code: ...
  }
}
```

```
class ThreadExample {
  public static void main(String[] args) {
    Thread t = new Thread(new MyRunnable());
      t.start();
  }
}
```

Beispiel mit Runnable

Erklärung:

1. **MultiThreadExample** ist die Hauptklasse, in der das Programm startet
2. Eine **for-Schleife** wird verwendet, um 5 Threads zu erstellen und zu starten. Jeder Thread wird durch die CountingTask-Klasse repräsentiert
3. **CountingTask** implementiert das Runnable-Interface. Das run()-Methode definiert den Code, der in jedem Thread ausgeführt wird
4. Jeder Thread gibt seinen Namen (threadName) und die aktuelle Iterationsnummer (i) aus

```
public class MultiThreadExample {
  public static void main(String[] args) {
    // Erstelle 5 Threads
    for (int i = 1; i <= 5; i++) {
      // Jeder Thread bekommt einen eindeutigen Namen, z.B. "Thread-1", usw.
      Thread thread = new Thread(new CountingTask("Thread-" + i));
      thread.start(); // Startet den Thread
    }
    System.out.println("Main ist fertig");
  }
}

class CountingTask implements Runnable {
  private final String threadName;

  // Konstruktor für den Thread, um den Namen zu setzen
  public CountingTask(String name) {
    this.threadName = name;
  }

  @Override
  public void run() {
    // Iteriere von 0 bis 99
    for (int i = 0; i < 100; i++) {
      System.out.println(threadName + ": " + i);
    }
  }
}
```

Runnable-Objekte als Lambda-Ausdrücke

- Da Runnable ein funktionales Interface ist, dürfen Lambda-Ausdrücke als Runnable-Objekte verwendet werden
- Damit ist eine prägnante Schreibweise möglich:

```
class MultiThreadLambda {
  public static void main(String[] args) {
    Runnable myRun = () -> {
      System.out.println("myRun läuft");
    };
    Thread t = new Thread(myRun);
    t.start();
  }
}
```

- Noch kürzer:

```
class MultiThreadLambda {
  public static void main(String[] args) {
    new Thread (() -> {
      System.out.println("myRun läuft");
    }).start();
  }
}
```

Agenda

- Motivation: Was ist Multithreading?
- Klasse Thread und Interface Runnable
- Methode *join* und Parallelisierung von Algorithmen
- Daemon und User Threads
- Synchronisierung mit *synchronized*
- Erzeuger / Verbraucher-Problem und die Methoden *wait, notify, notifyAll*
- Zustände eines Java-Threads

Mit join auf Beendigung von Threads warten

- Mit der Methode join() der Klasse Thread wird solange gewartet, bis der Thread zu Ende gelaufen ist
- join kann eine InterruptedException werfen

```
public class MultiThreadJoin {
  public static void main(String[] args)throws InterruptedException{
    Thread t = new Thread();
    t.start();
    // irgendwelche Berechnungen des main-Threads: ...
    t.join(); // Warte, bis der Thread t fertig ist
    // ... weitere Berechnungen des main-Threads
  }
}
```

- Mit dem start-join-Konzept lassen sich sehr einfach Daten-parallele Algorithmen realisieren (d.h. Daten lassen sich in unabhängige Teile zerlegen und nebenläufig bearbeiten)

Unser Eingangsbeispiel mit join

```java
public class MultiThreadJoin {
  public static void main(String[] args) throws InterruptedException {
    // Erstelle 5 Threads
    for (int i = 1; i <= 5; i++) {
      // Jeder Thread bekommt einen eindeutigen Namen, z.B. "Thread-1", usw.
      Thread thread = new Thread(new CountingTask("Thread-" + i));
      thread.start(); // Startet den Thread
      thread.join(); // Warte, bis der Thread fertig ist
    }
    System.out.println("Main ist fertig");
  }
}

class CountingTask implements Runnable {
  private final String threadName;

  // Konstruktor für den Thread, um den Namen zu setzen
  public CountingTask(String name) {
    this.threadName = name;
  }

  @Override
  public void run() {
    // Iteriere von 0 bis 99
    for (int i = 0; i < 100; i++) {
      System.out.println(threadName + ": " + i);
    }
  }
}
```

```
Thread-1: 0
Thread-1: 1
Thread-1: 2
...
Thread-2: 0
Thread-2: 1
Thread-2: 2
...
Thread-3: 0
Thread-3: 1
Thread-3: 2
...
Thread-4: 0
Thread-4: 1
Thread-4: 2
...
Thread-5: 0
Thread-5: 1
Thread-5: 2
...
Thread-5: 99
Main ist fertig
```

Konsolenausgabe

Beispiel: paralleles Befüllen eines Feldes

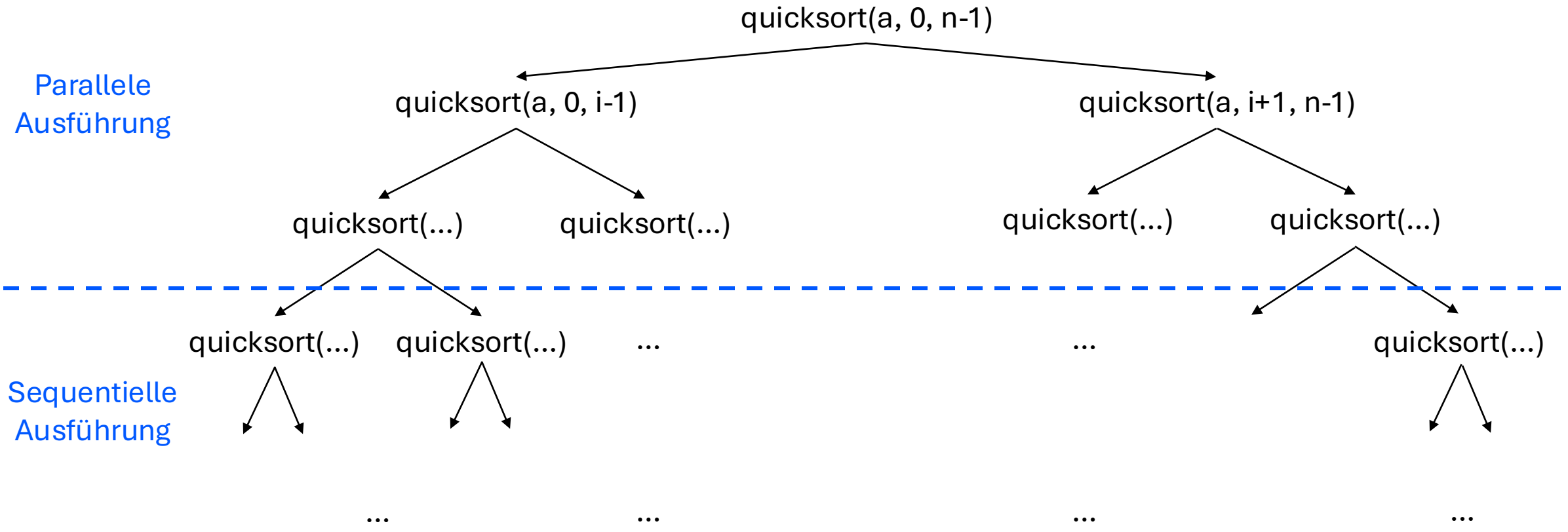
```
import java.util.Arrays;

class RandomizeArrayThread extends Thread {
  private final double[ ] a;
  private final int li;
  private final int re;
  public RandomizeArrayThread (double[ ] a, int li, int re) {
    this.li = li;
    this.re = re;
    this.a = a;
  }

  @Override
  public void run() {
    for (int i = li; i < re; i++)
    a[i] = Math.random();
  }
}
```

- Die run-Methode befüllt ein Feld a von a[li] bis a[re-1] mit zufälligen Zahlen
- a, li und re werden als Parameter beim Konstruktor übergeben

```
public class JoinApplication {
  public static void main(String[ ] args) throws InterruptedException {
    int N = 1000;
    double[ ] a = new double[N];
    Thread t1 = new RandomizeArrayThread(a, 0, N/2);
    Thread t2 = new RandomizeArrayThread(a, N/2, N);
    t1.start();
    t2.start();
    t1.join(); // Warte bis t1 zu Ende
    t2.join(); // Warte bis t2 zu Ende
    System.out.println(Arrays.toString(a));
    System.out.println("Alles fertig");
  }
}
```

- Der main-Thread startet zwei parallele Threads t1 und t2, die zwei unabhängige Teile des Felds a mit zufälligen Zahlen initialisieren
- Danach wartet der main-Thread, bis beide Threads t1 und t2 zu Ende gelaufen sind

Beispiel: paralleles QuickSort (1)



Nur QuickSort-Aufrufe bis zur Rekursionstiefe d = 2 einschl. sollen parallel ausgeführt werden

Beispiel: paralleles QuickSort (2)

```
public class ParallelQuickSort {
 // der eigentliche Code
}
```

```
public static void sort(int[] a) {
  int maxDepth = 2; // maximale Rekursionstiefe für paralleles Sortieren
  Thread sortThread = new QuickSortThread(a, 0, a.length - 1, maxDepth);
  sortThread.start();
  try {
    sortThread.join(); // Warten, bis der Thread fertig ist
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
}
```

Übergeordnete Sortiermethode startet einen Thread und wartet auf sein Ende

```
// Thread-Klasse für paralleles QuickSort
class QuickSortThread extends Thread {
  private int[] a;
  private int li;
  private int re;
  private int maxDepth;

  public QuickSortThread(int[] a, int li, int re, int maxDepth) {
    this.a = a;
    this.li = li;
    this.re = re;
    this.maxDepth = maxDepth;
  }

  @Override
  public void run() {
    if (li >= re) return;

    int i = partition3Median(a, li, re);

    if (maxDepth <= 0) {
      // Sequentielles QuickSort, wenn die maximale Tiefe erreicht ist
      quickSort(a, li, i - 1);
      quickSort(a, i + 1, re);
    } else {
      Thread tli = null;
      Thread tre = null;
      if (li < i - 1) {
        tli = new QuickSortThread(a, li, i - 1, maxDepth - 1);
        tli.start();
      }

      if (i + 1 < re) {
        tre = new QuickSortThread(a, i + 1, re, maxDepth - 1);
        tre.start();
      }

      try {
        if (tli != null) tli.join(); // Warten, bis der linke Thread fertig ist
        if (tre != null) tre.join(); // Warten, bis der rechte Thread fertig ist
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
// nächste Folie bitte ...
```

Das Runnable-Objekt wird mit den QickSort-Parametern initialisiert

Partitionierung mit 3-Median-Strategie

Paralles QuickSort

Beispiel: paralleles QuickSort (2)

```
// ... weiter geht's
  // Partitionierung mit 3-Median-Strategie
  private int partition3Median(int[] a, int li, int re) {
    int pivot = a[re];
    int i = li - 1;
    for (int j = li; j < re; j++) {
      if (a[j] < pivot) {
        i++;
        swap(a, i, j);
      }
    }
    swap(a, i + 1, re);
    return i + 1;
  }

  // Hilfsmethode zum Tauschen von Elementen
  private void swap(int[] a, int i, int j) {
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
  }

  // Sequentielles QuickSort
  private void quickSort(int[] a, int li, int re) {
    if (li < re) {
      int i = partition3Median(a, li, re);
      quickSort(a, li, i - 1);
      quickSort(a, i + 1, re);
    }
  }
}
```

```
// Testmethode
public static void main(String[] args) {
  int[] array = {9, 7, 5, 11, 12, 2, 14, 3, 10, 6};
  System.out.println("Unsortiertes Array:");
  for (int num : array) {
    System.out.print(num + " ");
  }
  System.out.println();

  sort(array); // Paralleles QuickSort starten

  System.out.println("Sortiertes Array:");
  for (int num : array) {
    System.out.print(num + " ");
  }
}
```

Unsortiertes Array:
9 7 5 11 12 2 14 3 10 6
Sortiertes Array:
2 3 5 6 7 9 10 11 12 14

Konsolenausgabe

Mit join auf Beendigung von Threads warten

- CPU-Zeiten auf iMac Studio M2 (12 Kerne)
- n = 100 Mio. int-Zahlen

Arrays.sort():	4.2 sec
Arrays.parallelSort():	0.9 sec (6.2 sec bei n = 10^9)
QuickSortThread (depth = 8)	1.1 sec (12 sec bei n = 10^9)
Stream.sort()	4.5 sec
Stream.parallel().sort()	1.0 sec
Python Liste	32 sec
Python numpy-Feld	9.5 sec

Agenda

- Motivation: Was ist Multithreading?
- Klasse Thread und Interface Runnable
- Methode *join* und Parallelisierung von Algorithmen
- Daemon und User Threads
- Synchronisierung mit *synchronized*
- Erzeuger / Verbraucher-Problem und die Methoden *wait, notify, notifyAll*
- Zustände eines Java-Threads

Daemon und User Threads

- Es gibt zwei Arten von Threads in Java:
  - **Daemon-Threads**
  - **User-Threads**
- Beim Start eines Java-Programms läuft der Main-Thread (aus der *main()*-Methode) sofort los. Von ihm aus können weitere Threads gestartet werden. In der Regel ist der Main-Thread der letzte, der beendet wird, da er abschließende Aufgaben ausführt
- **Daemon-Threads** sind Hintergrundthreads mit niedriger Priorität, z. B. der Garbage Collector
- Die JVM beendet Daemon-Threads automatisch, sobald alle User-Threads abgeschlossen sind
- **User-Threads** laufen normalerweise bis zum Ende ihrer Aufgabe, während Daemon-Threads von der JVM beendet werden, wenn keine User-Threads mehr aktiv sind

Beispiel Daemon Threads

```
public class UserDaemonThreads {
  public static void main(String[] args) {
    Thread bgThread = new Thread(new DaemonHelper());
    Thread usrThread = new Thread(new UserHelper());
    bgThread.setDaemon(true);

    bgThread.start();
    usrThread.start();
  }
}

class DaemonHelper implements Runnable {
  @Override
  public void run() {
    int counter = 0;
    while (counter < 500) {
      try {
        Thread.sleep(1000);
      } catch (InterruptedException e) {
        throw new RuntimeException(e);
      }
      counter++;
      System.out.println("Daemon helper running ...");
    }
  }
}

class UserHelper implements Runnable {
  @Override
  public void run() {
    try {
      Thread.sleep(5000);
    } catch (InterruptedException e) {
      throw new RuntimeException(e);
    }
    System.out.println("User thread done with execution.");
  }
}
```

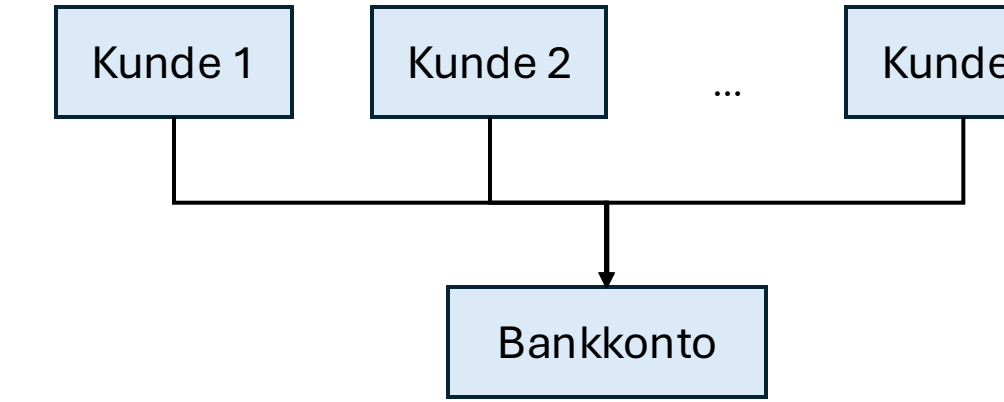
```
Daemon helper running ...
Daemon helper running ...
Daemon helper running ...
Daemon helper running ...
User thread done with
execution.
```

Konsolenausgabe

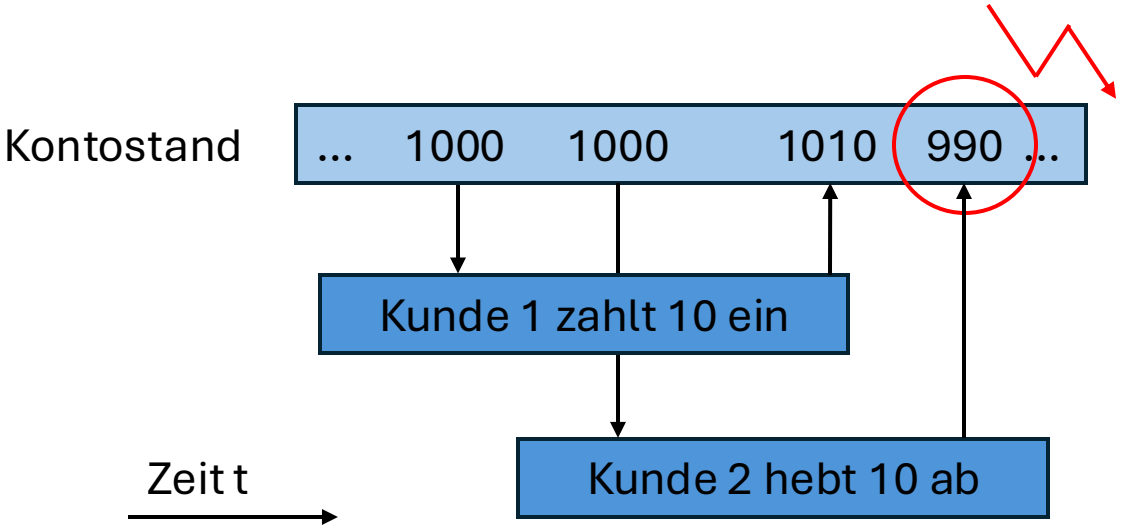
Agenda

- Motivation: Was ist Multithreading?
- Klasse Thread und Interface Runnable
- Methode *join* und Parallelisierung von Algorithmen
- Daemon und User Threads
- Synchronisierung mit *synchronized*
- Erzeuger / Verbraucher-Problem und die Methoden *wait, notify, notifyAll*
- Zustände eines Java-Threads

Problem bei nebenläufigem Zugriff auf gemeinsame Daten – Race Conditions



Verschiedene Kunden greifen auf ein gemeinsames Konto zu



Nebenläufiger Zugriff auf dasselbe Konto kann zu Inkosistenzen führen

Beispiel mit wechselseitigem Ausschluss

Race Condition: Wettlaufsituationen, bei der der zeitliche Ablauf das Ergebnis bestimmt. Beispiel:

- Paralleler Zugriff auf ein Bankkonto (aktueller Kontostand 1000€)
- Prozess A: Erhöht den Kontostand um 10€
- Prozess B: Verringert den Kontostand um 10€

Zeitpunkt	**Prozess A**	**Kontostand**		**Prozess B**
1	Aktuellen Kontostand lesen (1000€)	1000€		Aktuellen Kontostand lesen (1000€)
2	Neuen Kontostand berechnen (1010€)	1000€		Neuen Kontostand berechnen (990€)
3	Neuen Kontostand schreiben	1010€	990€	Neuen Kontostand schreiben (Prozess B schreibt kurz nach Prozess A)

Problem bei nebenläufigem Zugriff: Beispiel

```
static class BankAccount {
  private int balance;
  public BankAccount(int initialBalance) {balance = initialBalance; }
  public int getBalance() {return balance; }
  public void deposit(int amount) {balance += amount; }
}

static class Customer extends Thread{
  private BankAccount account;
  private int amount;
  public Customer(BankAccount a, int d) { account = a; amount = d; }
  public void run() {
  for (int i = 0; i < 1000; i++) account.deposit(amount);
  }
}

public static void main(String[] args) throws InterruptedException {
  BankAccount a = new BankAccount(1000);
  Thread kunde1 = new Customer(a, +10);
  Thread kunde2 = new Customer(a, -10);
  kunde1.start(); kunde2.start();
  kunde1.join(); kunde2.join();
  System.out.println(a.getBalance());
}
```

Bankkonto mit Startguthaben
balance = initialBalance

Kunde führt 1000 Buchungen durch

Bankkonto mit Startguthaben
balance = 1000 definieren

Es warden 2 Kunden gestartet.
Kunde 1 zahlt 1000x 10 ein.
Kunde 2 hebt 1000x 10 ab.

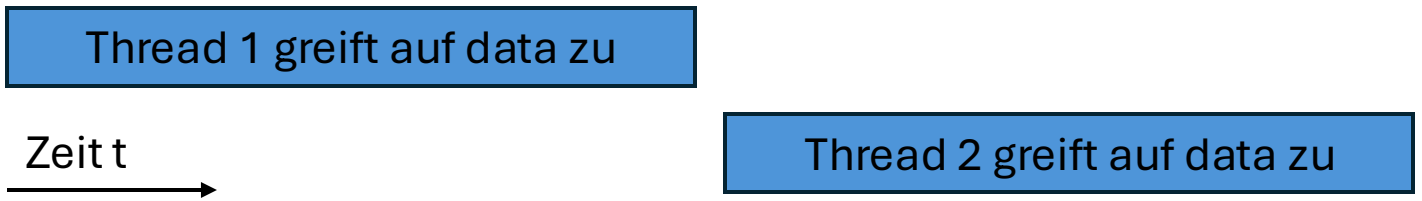
Kontostand hat nicht immer den erwarteten Wert balance = 1000!

Synchronisierung mit synchronized-Methode

- Bei Eintritt in eine **synchronized-Methode** wird das Objekt **gesperrt** und bei Austritt wieder **freigegeben** (**locking Mechanismus**)
- Zu einem Zeitpunkt darf daher höchstens ein Thread auf ein gemeinsames Objekt mit einer synchronized-Methode zugreifen
- Der Thread, der ein gesperrtes Objekt bearbeiten möchte, wird **blockiert**, bis das Objekt wieder freigegeben wird
- Beachte: auf verschiedene Objekte darf gleichzeitig zugegriffen werden

```
class GemeinsameDaten {
  ...
  public snychronized ... zugriff1(...) { ... }
  public snychronized ... zugriff2(...) { ... }
  ...
}
```

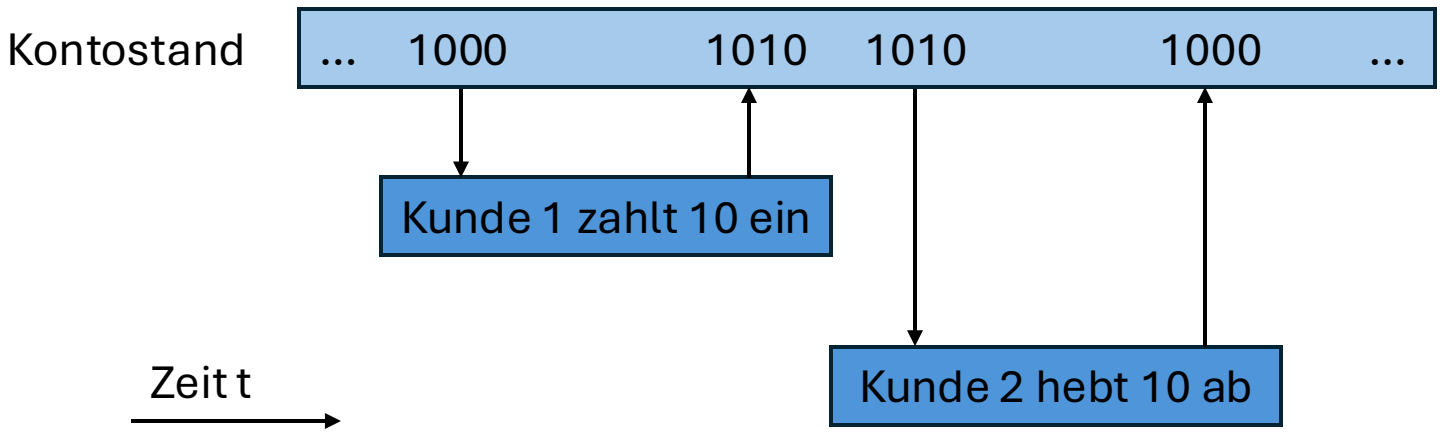
```
GemeinsameDaten data = new GemeinsameDaten();
```



Threads greifen auf gemeinsame Daten nicht gleichzeitig zu!

Problem bei nebenläufigem Zugriff auf gemeinsame Daten

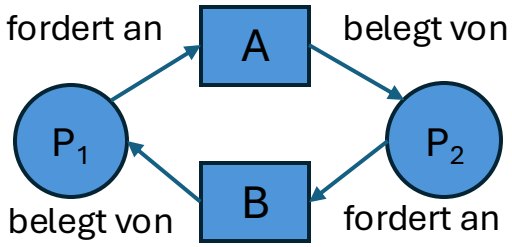
```
static class BankAccount {
  private int balance;
  public BankAccount(int initialBalance) {balance = initialBalance; }
  public synchronized int getBalance() {return balance; }
  public synchronized void deposit(int amount) {balance += amount; }
}
```



Kunde 2 wird so lange blockiert, bis die Buchung von Kunde 1 erledigt ist

Deadlock

Ein **Deadlock** entsteht, wenn zwei oder mehr Threads auf Ressourcen warten, die von anderen Threads blockiert werden, sodass keiner der Threads weiterarbeiten kann → Verklemmung



P_i ... Prozesse
A, B ... Ressourcen

```
class DeadlockExample {
  private final Object lock1 = new Object();
  private final Object lock2 = new Object();

  public void method1() {
    synchronized (lock1) {
      System.out.println("Thread 1: Holding lock 1...");

      try { Thread.sleep(10); } catch (InterruptedException e) {}

      System.out.println("Thread 1: Waiting for lock 2...");
      synchronized (lock2) {
        System.out.println("Thread 1: Holding lock 1 & 2...");
      }
    }
  }

// ...
```

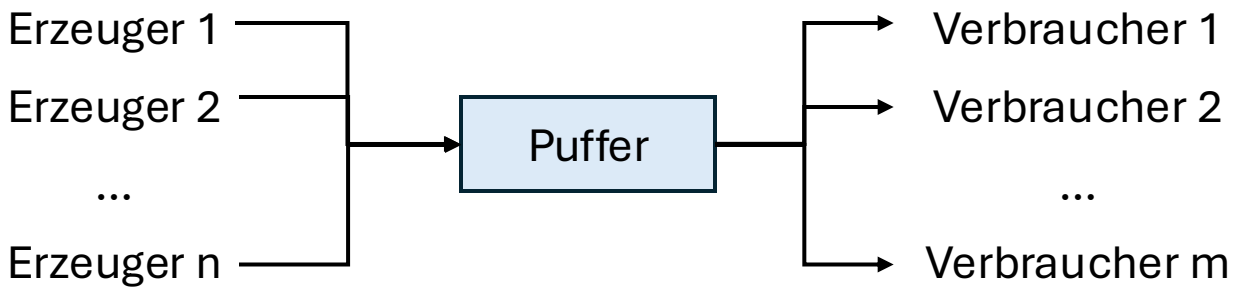
```
// ...
  public void method2() {
    synchronized (lock2) {
      System.out.println("Thread 2: Holding lock 2...");

      try { Thread.sleep(10); } catch (InterruptedException e) {}

      System.out.println("Thread 2: Waiting for lock 1...");
      synchronized (lock1) {
        System.out.println("Thread 2: Holding lock 1 & 2...");
      }
    }
  }
}

public class Main {
  public static void main(String[] args) {
    DeadlockExample deadlock = new DeadlockExample();

    Thread t1 = new Thread(deadlock::method1);
    Thread t2 = new Thread(deadlock::method2);

    t1.start();
    t2.start();
  }
}
```

Agenda

- Motivation: Was ist Multithreading?
- Klasse Thread und Interface Runnable
- Methode *join* und Parallelisierung von Algorithmen
- Daemon und User Threads
- Synchronisierung mit *synchronized*
- Erzeuger / Verbraucher-Problem und die Methoden *wait*, *notify*, *notifyAll*
- Zustände eines Java-Threads

Erzeuger / Verbraucher-Problem

- Es gibt verschiedene **Erzeuger-Threads**, die Daten erzeugen und in einen Puffer (z.B. eine Queue) schreiben
- Es gibt verschiedene **Verbraucher-Threads**, die Daten vom Puffer holen und verarbeiten
- Zugriff auf Puffer muss **synchronisiert** werden
- Verbraucher-Threads müssen **warten**, falls Puffer leer ist
- Falls Erzeuger-Threads Daten im Puffer ablegt, dann müssen wartende Verbraucher **benachrichtigt** und aktiviert warden
- Zusätzlich kann der Puffer begrenzte Kapazität haben, so dass auch Erzeuger eventuell warten müssen und vom Verbraucher benachrichtigt werden müssen



Methoden wait, notify, notifyAll

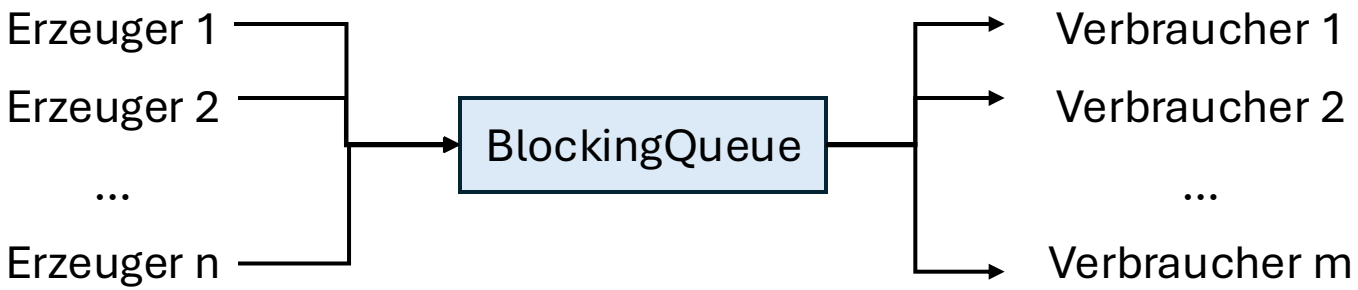
- Mit der Methode **wait** wird ein Thread solange in den **Wartezustand** gesetzt, bis eine Bedingung B erfüllt ist. **wait erfolgt in einer Schleife**, da bei Aktivierung des Threads Bedingung erneut geprüft werden muss
- Mit der Methode **notifyAll** werden **alle wartenden Threads wieder aktiviert**
- Mit **notify** wird **irgendein wartender Thread aktiviert**
- wait und notifyAll (notify) sollten in synchronized-Methoden aufgerufen werden, da auf gemeinsame Daten zugegriffen wird
- wait, notify und notifyAll sind in in der **Klasse Object** definiert
- **Wichtig**: Die hier vorgegebenen Muster für die Benutzung von wait, notify und notiyfAll sollten befolgt werden!

```
snychronized void doWhenCondition() {
  while (!B)
  wait();
  // Zugriff auf gemeinsame Daten:
  // ...
}
```

```
snychronized void changeCondition() {
  // Zugriff auf gemeinsame Daten:
  // ...
  // Bedingung B kann sich nun geändert haben.
  // Daher wartende Threads benachrichtigen,
  // um Bedingung B neu zu prüfen:
  notifyAll(); // oder notify();
}
```

Beispiel Queue (1)

- Verschiedene **Erzeuger-Threads** schreiben Daten in eine Queue
- **Verbraucher-Threads** holen die Daten aus der Queue
- Verbraucher-Threads müssen **warten (Methode wait)**, falls die Queue leer ist
- Sobald ein Erzeuger-Thread Daten in die Queue schreibt, wird irgendein Verbraucher mit **notify** aktiviert



Beispiel Queue (2)

```java
import java.util.LinkedList;
import java.util.Queue;

// BlockingQueue-Klasse
class BlockingQueue {
  private final Queue<Integer> queue = new LinkedList<>();

  // synchronized Methode zum Hinzufügen eines Elements zur Queue
  public synchronized void add(int x) {
    queue.add(x);
    notify(); // Benachrichtigt wartende Threads, dass ein neues Element hinzugefügt wurde
  }

  // synchronized Methode zum Entfernen eines Elements aus der Queue
  public synchronized int remove() throws InterruptedException {
    while (queue.isEmpty()) {
      wait(); // Wartet, bis ein Element in der Queue vorhanden ist
    }
    return queue.poll(); // Holt das erste Element aus der Queue
  }
}
```

Nur Verbraucher-Threads können im Warte-Zustand sein.

Es genügt, irgendein wartenden Verbraucher-Thread zu aktivieren.

Daher: notify (und nicht notifyAll)

Beispiel Queue (3)

Producer-Thread schreibt 100 Zahlen in die BlockingQueue

```
// Producer-Klasse (Erzeuger-Thread)
class Producer extends Thread {
  private final BlockingQueue bq;
  private final int start;

  public Producer(BlockingQueue bq, int start) {
    this.bq = bq;
    this.start = start;
  }

  @Override
  public void run() {
    // Schreibt 100 Zahlen in die BlockingQueue
    for (int i = start; i < start + 100; i++) {
      bq.add(i);
    }
  }
}
```

Consumer-Thread holt 150 Zahlen aus der BlockingQueue und gibt sie aus

```
// Consumer-Klasse (Verbraucher-Thread)
class Consumer extends Thread {
  private final BlockingQueue bq;
  private final String name;

  public Consumer(BlockingQueue bq, String name) {
    this.bq = bq;
    this.name = name;
  }

  @Override
  public void run() {
    // Holt 150 Zahlen aus der BlockingQueue und gibt sie aus
    for (int i = 0; i < 150; i++) {
      try {
        int value = bq.remove();
        System.out.println(name + ": " + value);
      } catch (InterruptedException ex) {
        Thread.currentThread().interrupt();
      }
    }
  }
}
```

Beispiel Queue (4)

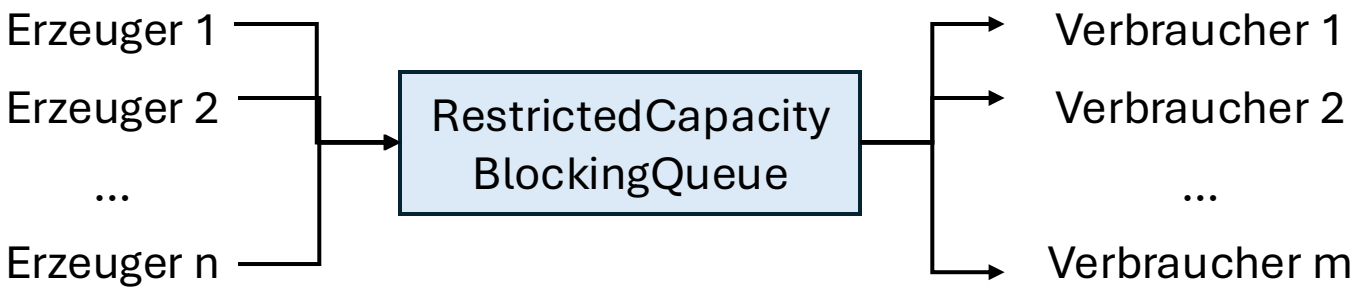
```
// Hauptklasse zur Ausführung des Producer-Consumer-Beispiels
public class ProducerConsumerExample {
  public static void main(String[] args) {
    BlockingQueue bq = new BlockingQueue();

    // Erzeuge drei Producer-Threads
    Producer p1 = new Producer(bq, 0);
    Producer p2 = new Producer(bq, 1000);
    Producer p3 = new Producer(bq, 1000000);

    // Erzeuge zwei Consumer-Threads
    Consumer c1 = new Consumer(bq, "Consumer 1");
    Consumer c2 = new Consumer(bq, "Consumer 2");

    // Starte alle Producer- und Consumer-Threads
    p1.start();
    p2.start();
    p3.start();
    c1.start();
    c2.start();
  }
}
```

Es werden 3 Producer-Thread gestartet, die insgesamt 300 Zahlen in die BlockingQueue schreiben

Es werden 2 Consumer-Threads gestartet, die insgesamt 300 Zahlen aus der BlockingQueue holen und ausgeben

Beispiel mit kapazitätsbegrenzter Queue (1)

- Verschiedene **Erzeuger-Threads** schreiben Daten in eine **kapazitätsbegrenzte Queue**
- **Verbraucher-Threads** holen die Daten aus der Queue
- Verbraucher-Threads müssen **warten (Methode wait)**, falls die **Queue leer** ist. Sobald ein Erzeuger-Thread Daten in die Queue schreibt, werden **alle wartenden Threads** mit **notifyAll** aktiviert
- Erzeuger-Threads müssen **warten (Methode wait)**, falls die **Queue voll** ist. Sobald ein Verbraucher-Thread Daten aus der
- Queue holt, werden **alle wartenden Threads** mit **notifyAll** aktiviert



Beispiel mit kapazitätsbegrenzter Queue (2)

```
class RestrictedCapacityBlockingQueue {
  private final Queue<Integer> queue = new LinkedList<>();
  private final int cap = 5; // Kapazität der Queue

  // synchronized Methode zum Hinzufügen eines Elements zur Queue
  public synchronized void add(int x) {
    while (queue.size() >= cap)
      wait();
    queue.add(x);
    System.out.println("added: " + queue.size());
    notifyAll(); // Benachrichtigt wartende Threads, dass ein neues Element hinzugefügt wurde
  }

  // synchronized Methode zum Entfernen eines Elements aus der Queue
  public synchronized int remove() throws InterruptedException {
    while (queue.isEmpty()) {
      wait(); // Wartet, bis ein Element in der Queue vorhanden ist
    }
    int x = queue.poll(); // Holt das erste Element aus der Queue
    System.out.println("removed: " + queue.size());
    notifyAll();
    return x;
  }
}
```

Hier muss wenigstens ein Consumer-Thread aktiviert werden

Hier muss wenigstens ein Producer-Thread aktiviert werden

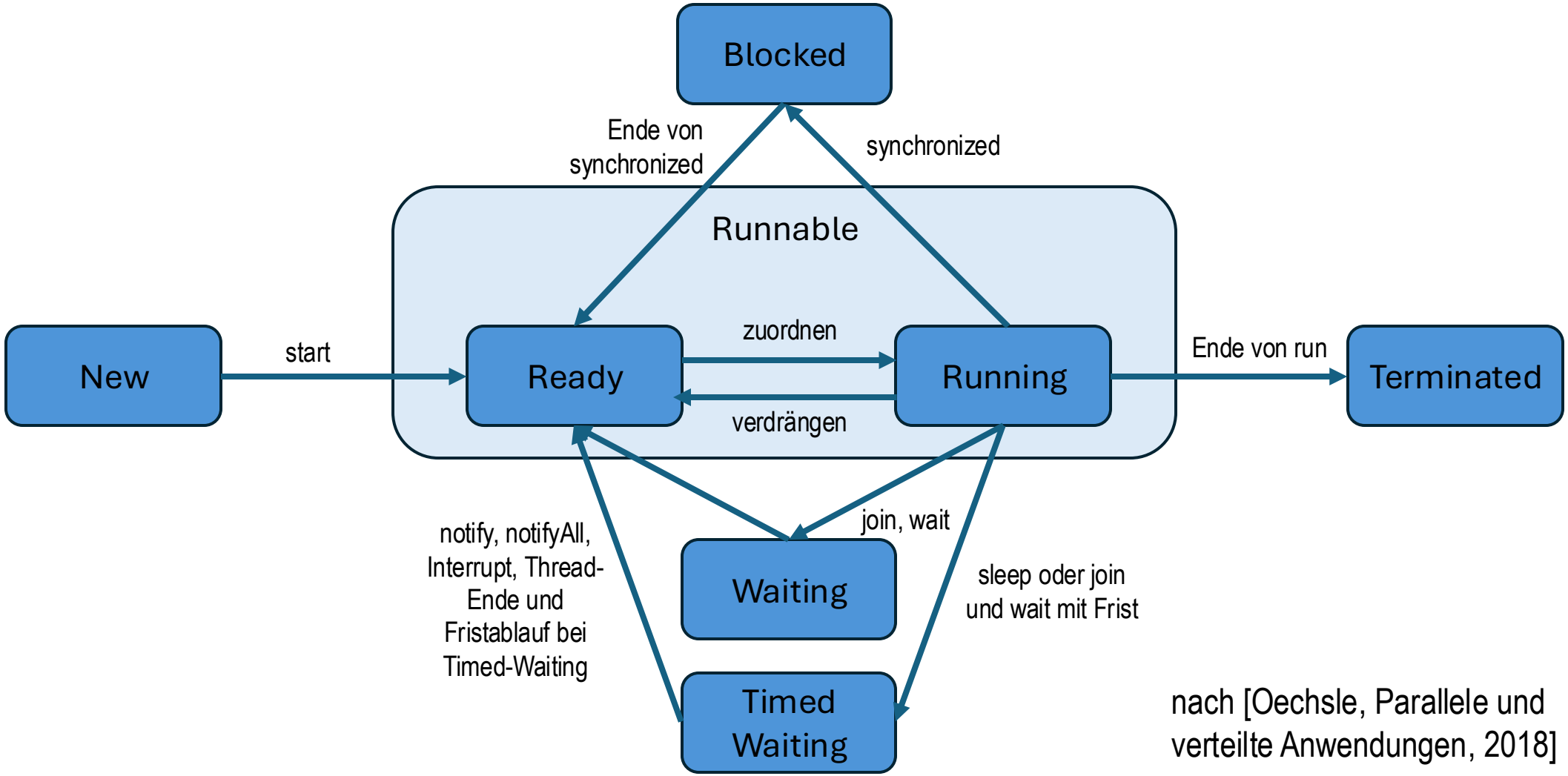
Da die Aktivierung irgendeines Threads nicht genügen würde, werden alle Threads aktiviert.

Daher: notifyAll (und nicht notify)

Agenda

- Motivation: Was ist Multithreading?
- Klasse Thread und Interface Runnable
- Methode *join* und Parallelisierung von Algorithmen
- Daemon und User Threads
- Synchronisierung mit *synchronized*
- Erzeuger / Verbraucher-Problem und die Methoden *wait, notify, notifyAll*
- Zustände eines Java-Threads

Zustände eines Java-Threads

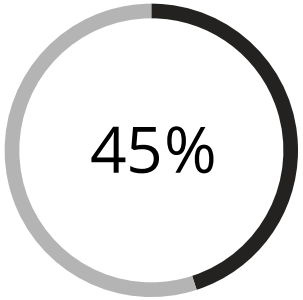


nach [Oechsle, Parallele und verteilte Anwendungen, 2018]

Zusammenfassung

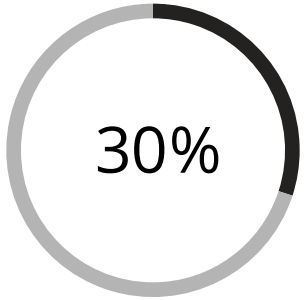
- Multithreading:
  - Bessere Leistung als bei parallelen Programmen mit mehreren Prozessen
  - Maximale Ausnutzung der CPU-Zeit
  - Spart Zeit durch parallele Aufgabenverarbeitung
- Methode join
- Synchronisierung mit synchronized
- Race Condition & Deadlock
- Thread-Kommunikation (wait, notify, notifyAll)
- Lebenszyklus (new, runnable, running, blocked, terminated, …)

Zukunft von Multithreading



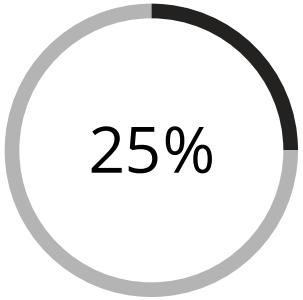
Cloud-Anwendungen

Diese Statistik zeigt einen Anstieg der Nachfrage nach Multithreading-Fähigkeiten in den letzten Jahren. Unternehmen suchen aktiv nach Entwicklern, die mit Multithreading-Techniken vertraut sind.



Steigende Nachfrage

Die Nutzung von Multithreading in Cloud-Anwendungen hat zugenommen, was schnelle und skalierbare Lösungen bietet. Diese Tendenz wird durch die ständig wachsende Verbindung von Geräten und Benutzern vorangetrieben.



Fortschrittliche Frameworks

Mit der Entwicklung neuer Frameworks bleibt Multithreading ein zentrales Thema in der Softwareentwicklung. Die Verbesserung von Multithreading-Techniken wird zukünftig die Effizienz und Benutzererfahrung weiter steigern.

Weiterführende Quellen

Mail:
konak@uni-potsdam.de

Videos

- Multithreading for Beginners (5:55 Std)
- Java Concurrency and Multithreading – Introduction (Serie von 26 Videos)

Folien

- Vorlesungsfolien zu Threads (HTWG Konstanz)
- Folien Threads (TU München)

Bücher

- Java Concurrency In Practice – GitHub (PDF)

Zum Nachmachen:
Heutige Vorlesungsfolien und Code findet ihr in GitHub

https://github.com/konakion/MyJavaThreads