Threads und Synchronisation

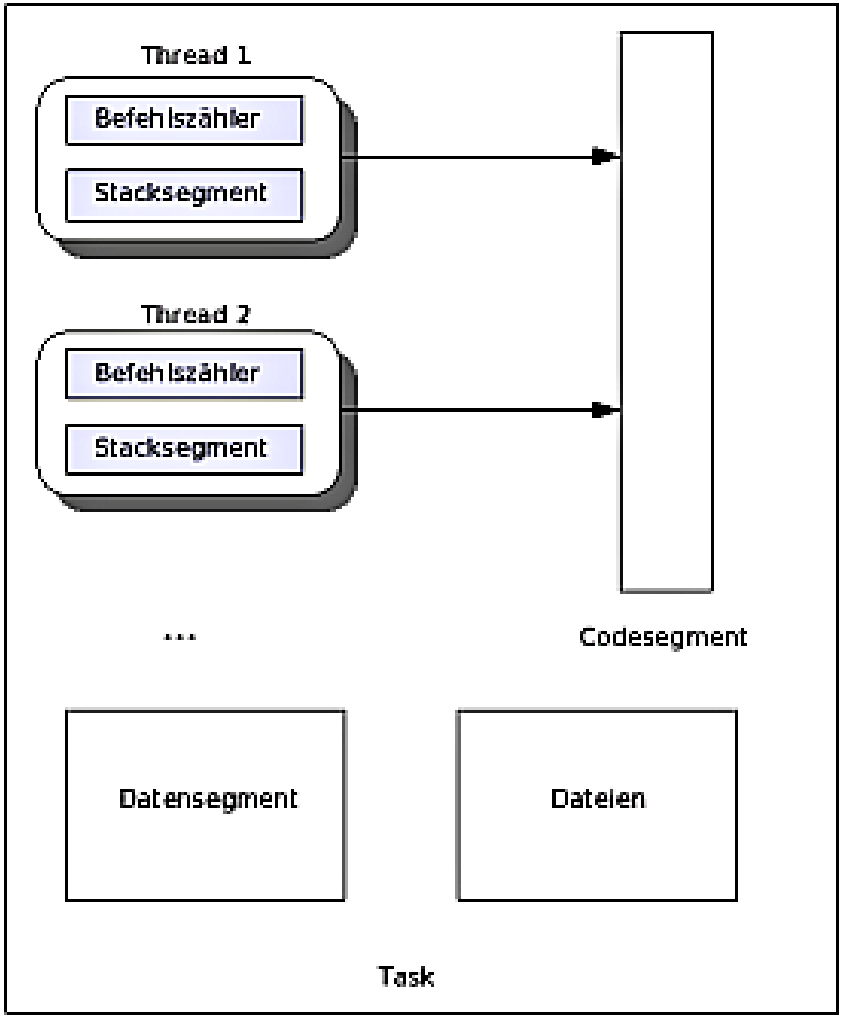
Vorlesungsinhalt

- Threads und Prozesse
- Nebenläufigkeit
- Synchronisation
- Monitore
- Semaphore
- Aufgabe

Threads und Prozesse

Process vs. Thread

- Ein Thread ist ein sequentieller Abarbeitungslauf eines Prozesses (Task) und teilt sich mit den anderen vorhandenen Threads (multithreading)
  - das Codesegment
  - das Datensegment (Ausnahme: Thread local Variablen)
  - die verwendeten Dateideskriptoren
- Threads innerhalb des gleichen Prozesses verwenden voneinander unabhängige Aufrufstapel (Callstacks)



Warum Threads?

- Desktop Anwendungen
  - Entkopplung von der Hauptanwendung (z.B. Rechtschreibprüfung)
  - Sicherung der Reaktionsfähigkeit des Systems (Progress Bar)
- Server Anwendungen
  - Datenbankzugriffe mehrerer Benutzer verwalten
  - Webzugriffe mehrerer Anfragen behandeln
  - Bereitstellung von Services
  - Sicherstellung der Resourcenzugriffe im Teilhaberbetrieb

Leider nutzen die meisten Server basierten Frameworks Threads in einer Form, die dem Anwender nicht vollständig transparent bleibt. Deshalb kommt auch der Durchschnittsprogrammierer nicht um das Thema herum. Das ist schade, weil durch falsche Threadprogrammierung nichtdeterministisches Verhalten entsteht und Fehler auftreten, deren Lokalisierung und Beseitigung sehr schwer sein kann.
Nette Story: Mars Rover Pathfinder

Threads in Java

- Java erlaubt keinen direkten Zugriff auf die Betriebssystem Threads
- Indirekter Zugriff über JVM und Thread Klasse möglich.
- Synchronisation von Threads über Java Mechanismen möglich
  - Monitore
  - Semaphore
  - Executors (Threadpools)
  - Parallel Streams
- Verwendung von Threads erfordert besondere Sorgfalt. Nicht alle Klassen der Java Klassenbibliothek sind Thread-safe.

Erzeugung nebenläufiger Verarbeitung

- `MyThread` erbt von Thread
- `MyThread` implementiert die Methode void run()
- start() erzeugt einen neuen BS Thread, startet diesen Thread und übergibt der Methode run() die Kontrolle
- die run() Methode läuft unabhängig und nebenläufig zum main Thread

Thread Erzeugung

```
class MyThread extends Thread {
  public void run(){
    System.out.println("Ein Thread");
  }
}

class Main{
  public static void main(String[] s) {
    MyThread t = new MyThread();
    t.start();
  }
}
```

Thread Erzeugung 2. Möglichkeit

- `MyRunner` implementiert das Interface Runnable
- Die Methode run() beschreibt was zu tun ist
- Dem Thread wird ein Runnable Objekt übergeben
- start() erzeugt einen neuen BS Thread, startet ihn und übergibt run() die Kontrolle
- start() wird beendet, während run() läuft!
- die run() Methode läuft unabhängig und nebenläufig

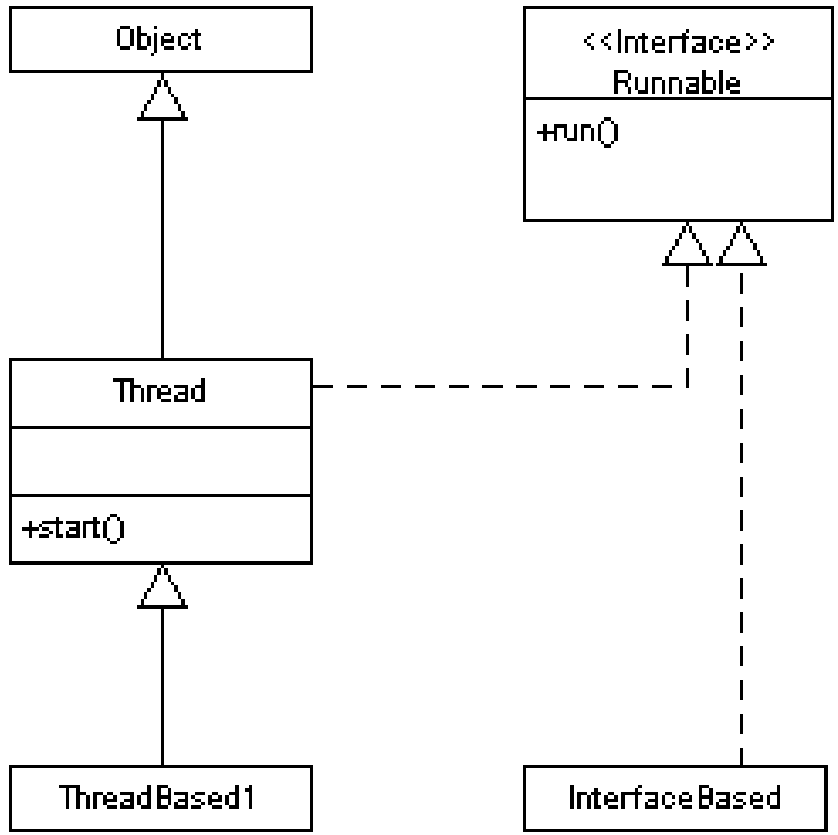
Alternative Threaderzeugung

```
class MyRunner implements Runnable {
  public void run(){
    System.out.println("Auch Thread");
  }
}

class Main{
  public static void main(String[] s) {
    Thread t = new Thread(new MyRunner());
    t.start();
  }
}
```

Klassendiagramm

- Eine Instanz des Interfaces Runnable repräsentiert die Verarbeitung eines nebenläufigen Ausführungsfadens. Sie fordert ausschließlich die Implementierung der Methode run().
- Eine Instanz Klasse Thread repräsentiert die Betriebs-systemresource selbst. Auch Thread implementiert Runnable, aber die Methode run() soll **nicht** aufgerufen werden, da dann die Verarbeitung nicht nebenläufig ausgeführt wird, sondern synchron zum Aufruf. Nur start() startet eine nebenläufige Aktion.

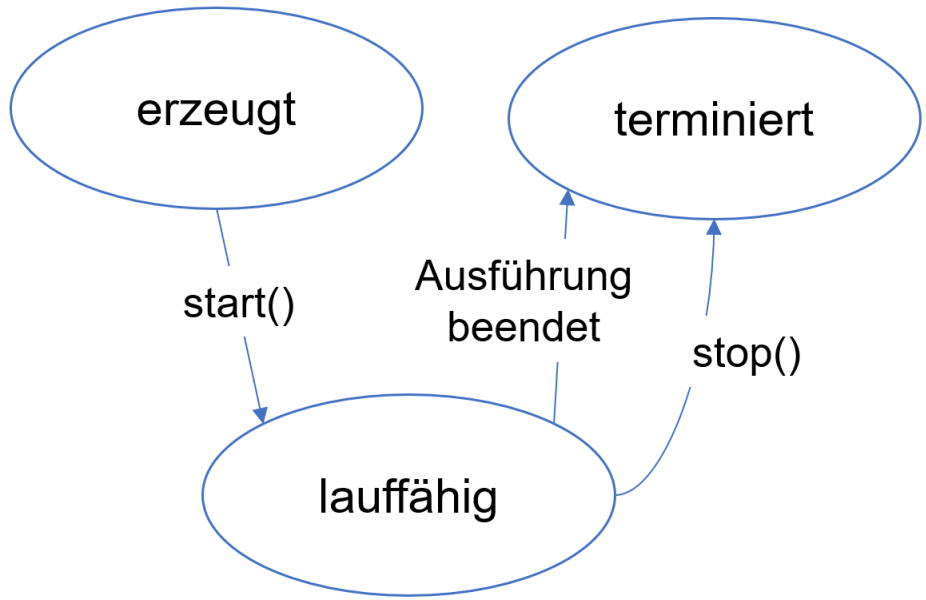


Gegenüberstellung der Methoden

	extends Thread	**implements Runnable**
Vorteile	in run() können Methoden der Klasse Thread genutzt werden	Klasse kann von problemspezifischen Klassen erben
Nachteile	Java kennt keine Mehrfachvererbung, Einsatzmöglichkeit begrenzt	schwieriger zu referenzieren für andere Objekte, Thread-Methoden nur mit Thread.currentThread() ausführbar

Wann wird ein Thread beendet?

- run()-Methode wurde beendet
- In run() tritt nicht selbst gefangene Exception auf
- Thread bekommt Nachricht stop() (schlecht: bearbeitete Objekte können in einem inkonsistenten Zustand sein)
- Besser: Nachricht interrupt() schicken, Thread kann mit isInterrupted() prüfen, ob er unterbrochen wurde



Unterbrechen von Threads

Thread mit interrupt beenden

```
class InterruptSample extends Thread {
  public void run() {
    while (!isInterrupted()) {
      for(int i=0; i<2000000; i++)
        System.out.println("Arbeite␣hart:␣"+i);
    }
  }
}
public class Main {
  public static void main(String[] args) throws Exception {
    InterruptSample t = new InterruptSample();
    t.start();
    t.interrupt();
  }
}
```

Nebenläufigkeit

Definition Nebenläufigkeit

- Nebenläufigkeit: Parallele Ausführung von Anweisungen auf einem oder mehreren Prozessoren.
- Organisation der Parallelität erfolgt explizit durch den Programmierer in geeigneter Hochsprache.
- Nebenläufige Ausführung ist auf eine einzige physische Maschine beschränkt. Diese kann jedoch mehrere vollständige CPUs enthalten.
- Die Literatur trifft die Unterscheidung zwischen Parallelität und Nebenläufigkeit nicht immer trennscharf und gleichermaßen eindeutig.

Nebenläufigkeit in Java

- Bei Java-Threads handelt es sich um eine Möglichkeit der nebenläufigen Ablaufsteuerung, da diese explizit (konkret: in Form von API-Aufrufen) durch den Applikationsprogrammierer festgelegt wird.
- Durch die Nebenläufigkeit tritt der codierte Kontrollfluß in den Hintergrund, zugunsten eines durch den Programmierer nicht beeinflußbaren Nichtdeterminismus.
- Es obliegt ausschließlich dem Prozessorzeit zuteilenden Betriebsystem, in welcher Reihenfolge die einzelnen Anweisungen ausgeführt werden.
- Eine einmal eingetretene Ausführungsreihenfolge ist nicht wieder reproduzierbar.

Demo Nebenläufigkeit

Hoch- und Runterzählen

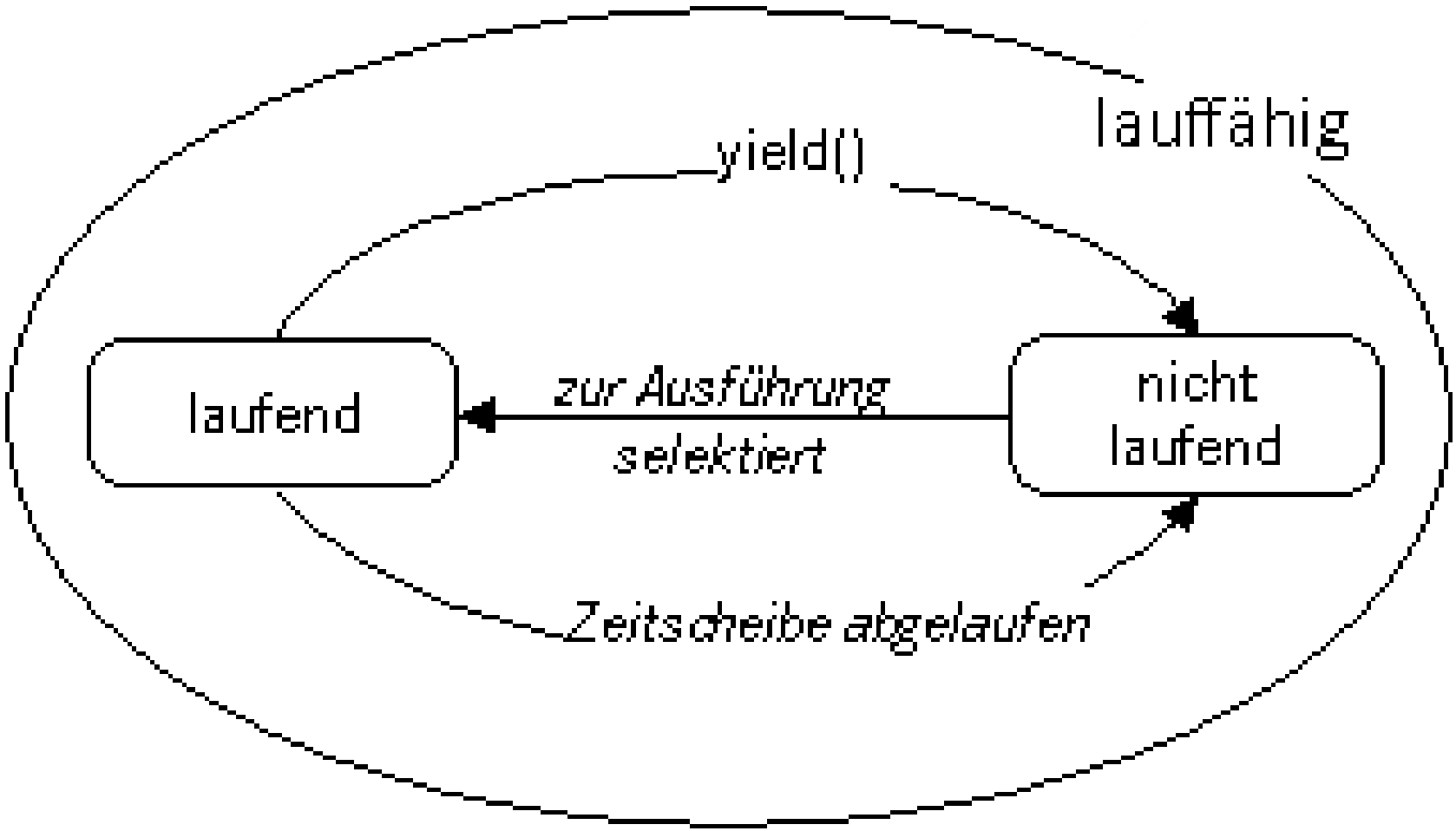
```
class Value {
    private int i=0;
    public void increment(){i++; System.out.print("␣"+i);}
    public void decrement(){i--; System.out.print("␣"+i);}
}
class Incrementer extends Thread { //Decrementer analog
    private Value val;
    public Incrementer(Value v){val=v;}
    public void run(){for(int i=0;i<100;i++) val.increment();}
}
public class Main {
    public static void main(String[] s) {
        Value v=new Value();
        new Incrementer(v).start(); new Decrementer(v).start();
    }
}
```

Beispielausgaben

\eclipse\workspace\ThreadTest>java Main
-1 -2 -3 -4 -5 -6 -7 -8 -9 -10 -11 -12 -13 -14 -15 **-16** -15 -14 -13 -12 -11 -10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 **52** 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 **-1** 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 **31** 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

\eclipse\workspace\ThreadTest>java Main
-1 -2 -3 -4 -5 -6 -7 -8 -9 -10 -11 -12 -13 -14 -15 **-16** -15 -14 -13 -12 -11 -10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 **35** 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 -1 -2 -3 -4 -5 -6 -7 -8 -9 -10 -11 -12 -13 -14 **-15** -14 -13 -12 -11 -10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 **34** 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 1 1 10 9 8 7 6 5 4 3 2 1 0

Lauffähige Threads



Steuerung der Zeitzuteilung

Steuerung der Priorität vom Thread t mit

- t.setPriority(int) - ändert die Priorität des Threads
  - Thread.MIN_PRIORITY = 1
  - Thread.MAX_PRIORITY = 10
  - Standard - Thread.NORM_PRIORITY = 5
- Achtung: Keine Reaktion des Systems garantiert
- Bei Prioritäten größer als 5 kann es mit der IDE-Steuerung zu Verzögerungen kommen
- t.getPriority() - gibt die Priorität des Threads aus

Koordination nebenläufiger Programme

Eine der Hauptschwierigkeiten im Einsatz von Threads liegt in dem Nichtdeterminismus, der auf der nicht vorhersagbaren Zuteilung von Prozessorzeit durch den Scheduler beruht. Bei der Nutzung von nebenläufiger Verarbeitung treten Situationen auf, die eine definierte Reihenfolge der Verarbeitung erfordern. Zur Steuerung der Reihenfolge stehen im Java API verschiedene Möglichkeiten zur Verfügung, die

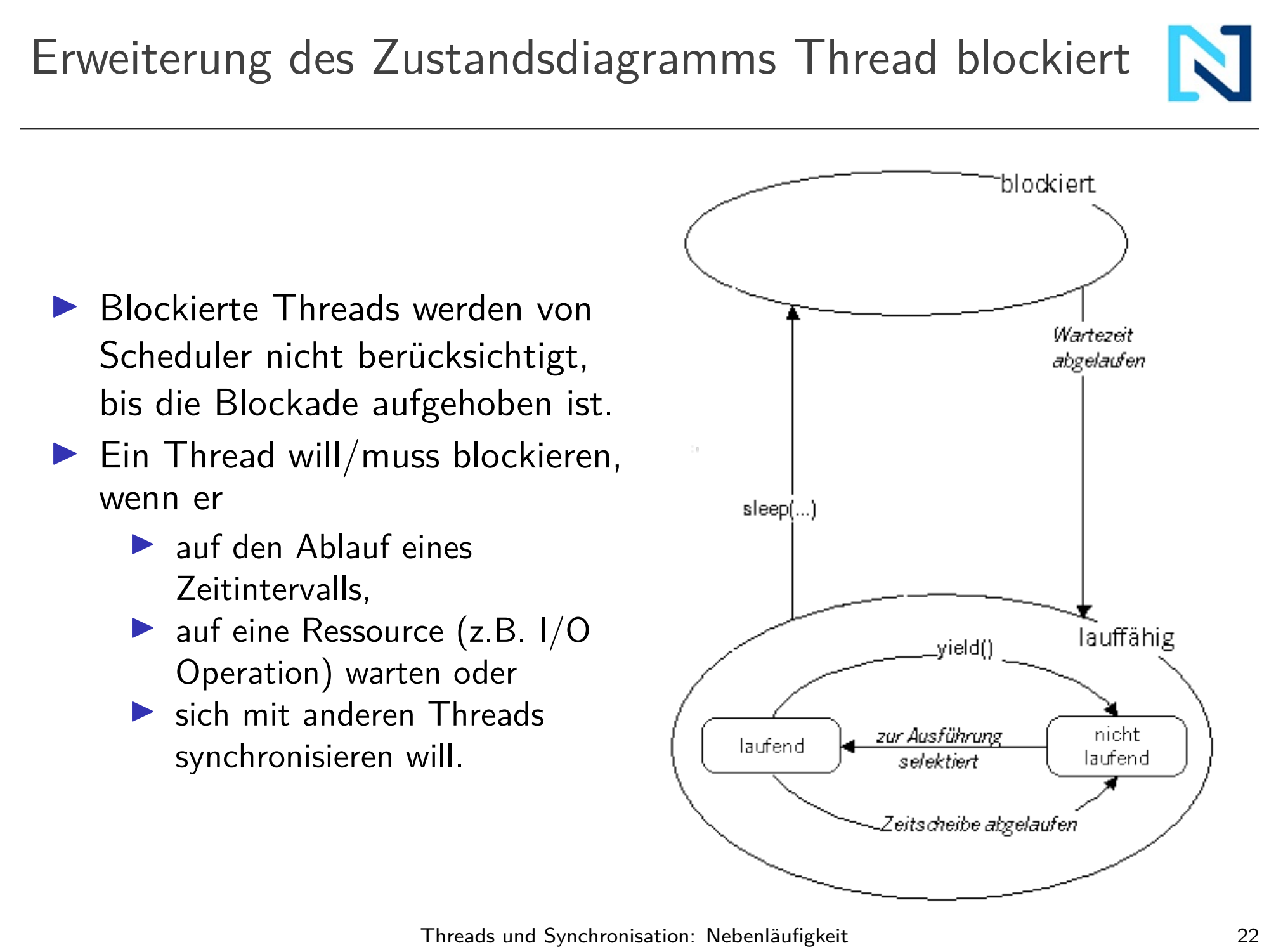
- Verzögerung eines Threads mit Hilfe von sleep()
- Warten auf das Beenden eines Threads mit join()
- Realisierung des wechselseitigen Ausschlusses mit Hilfe von Monitoren wait() und notify() sowie mit Semaphoren acquire() und release().

Thread Verzögerung

Beispiel Uhr

```
public class KleinerTicker implements Runnable {
    private int sec=0;
    public void run() {
        while(true){
            System.out.println((sec++)+ "␣Sekunden");
            try {
                Thread.sleep(1000); //1000 Millisekunden
            } catch (InterruptedException e) {
                System.out.println("Interrupted");
            }
        }
    }
    public static void main(String[] args) {
        new Thread(new KleinerTicker()).start();
    }
}
```

Erweiterung des Zustandsdiagramms Thread blockiert



- Blockierte Threads werden von Scheduler nicht berücksichtigt, bis die Blockade aufgehoben ist.
- Ein Thread will/muss blockieren, wenn er
  - auf den Ablauf eines Zeitintervalls,
  - auf eine Ressource (z.B. I/O Operation) warten oder
  - sich mit anderen Threads synchronisieren will.

Interrupts bei blockierten Threads

```
class Aktion extends Thread {
  public void run() {
    while (!isInterrupted()) {
      System.out.println("Bin␣aktiv");
      try {
        System.out.println("Geh␣schlafen");
        Thread.sleep(2000);
      } catch (InterruptedException e) {
        System.out.println("Aufgeweckt");
      }
    }
} }
public class Main {
  public static void main(String[] args) throws Exception {
    Aktion t = new Aktion(); t.start(); Thread.sleep(3000);
    t.interrupt();
} }
```

Wie werden blockierte Threads interrupted?

```
class Aktion extends Thread {
  public void run() {
    while (!isInterrupted()) {
      System.out.println("Bin␣aktiv");
      try {
        System.out.println("Geh␣schlafen");
        Thread.sleep(2000);
      } catch (InterruptedException e) {
        interrupt(); //Setzen des Interrupt Status!
        System.out.println("Aufgeweckt");
      }
    }
} }
public class Main {
  public static void main(String[] args) throws Exception {
    Aktion t = new Aktion(); t.start(); Thread.sleep(3000);
    t.interrupt();
} }
```

interrupt()

- Die Methode `interrupt` für ein Objekt der Klasse Thread, das sich in einem lauffähigen Zustand befindet, setzt ein Flag, das mit isInterrupted() abgefragt werden kann.
- Die Methode `interrupt` für ein Objekt der Klasse Thread, das sich im blockierten Zustand befindet, versetzt den Thread in einen lauffähigen Zustand (meist nicht laufend) mit einer InterruptedException. Das Interrupted-Flag wird dabei **nicht** gesetzt.
- isInterrupted() gibt den Zustand des Interrupted Flag wieder. **Ein Aufruf verändert den Status nicht**.
- Als Alternative zu isInterrupted() bei Verwendung der Interface basierten Konstruktionsmethode kann man die Klassenmethode Thread.interrupted() verwenden. Sie gibt den Interrupt Status des aktuellen Threads wieder und **setzt das Flag zurück**.

Warten auf das Ende einer Threadverarbeitung

So geht es nicht

```
class Erzeuger extends Thread {
  private StringBuffer speicher;
  public Erzeuger(StringBuffer s){speicher=s;}
  public void run(){speicher.append("x");}
}
public class Main {
  public static void main(String[] args) {
    StringBuffer speicher=new StringBuffer(":");
    Thread t1 = new Erzeuger(speicher);
    t1.start();
    Thread t2 = new Erzeuger(speicher);
    t2.start();
    System.out.println("Ergebnis:␣"+speicher);
  }
}
```

- Ein Erzeuger erzeugt ein x und legt es im Speicher ab
- Ausgabe ist :xx oder :x oder :

Warten auf das Ende einer Threadverarbeitung 2/2

Mit `join()` auf Ende der Threads warten

```
class Erzeuger extends Thread {
  private StringBuffer speicher;
  public Erzeuger(StringBuffer s){speicher=s;}
  public void run(){speicher.append("x");}
}
public class Main {
  public static void main(String[] args) {
    StringBuffer speicher=new StringBuffer(":");
    Thread t1 = new Erzeuger(speicher);
    t1.start();
    Thread t2 = new Erzeuger(speicher);
    t2.start();
    t1.join();
    t2.join();
    System.out.println("Ergebnis:␣"+speicher);
} }
```

- Ausgabe ist :xx

Synchronisation

Demo Lost Update 1/2

Nebenläufiger Zugriff auf Datei

```
public class Increment extends Thread {

  public static void main(String args[]) throws Exception {
    Thread[] threads = new Thread[5];
    new DataOutputStream(new FileOutputStream("testfile")).writeInt(0);
    for (int i=0; i<5; i++){
      threads[i] = new Increment();
      threads[i].start();
    }
    for (int i=0; i<5; i++) threads[i].join();
    System.out.println("counter value: "+
      new DataInputStream(new FileInputStream("testfile")).readInt());
  }

  public void run(){
    for (int i=0; i<10; i++) addOne();
  }
```

Demo Lost Update 2/2

Class Increment Teil 2

```
  public void addOne(){
    try{
      DataInputStream dis =
        new DataInputStream(new FileInputStream("testfile"));
      int value = dis.readInt();
      System.out.println("thread␣named␣"+
        Thread.currentThread().getName()+"␣read:␣"+value);
      value++;
      DataOutputStream dos =
        new DataOutputStream(new FileOutputStream("testfile"));
      dos.writeInt( value );
      System.out.println("thread␣named␣"+
        Thread.currentThread().getName()+"␣wrote:␣"+value);
      dis.close(); dos.close();
    } catch (IOException ioe){ioe.printStackTrace();}
  }
}
```

Beispielausgabe

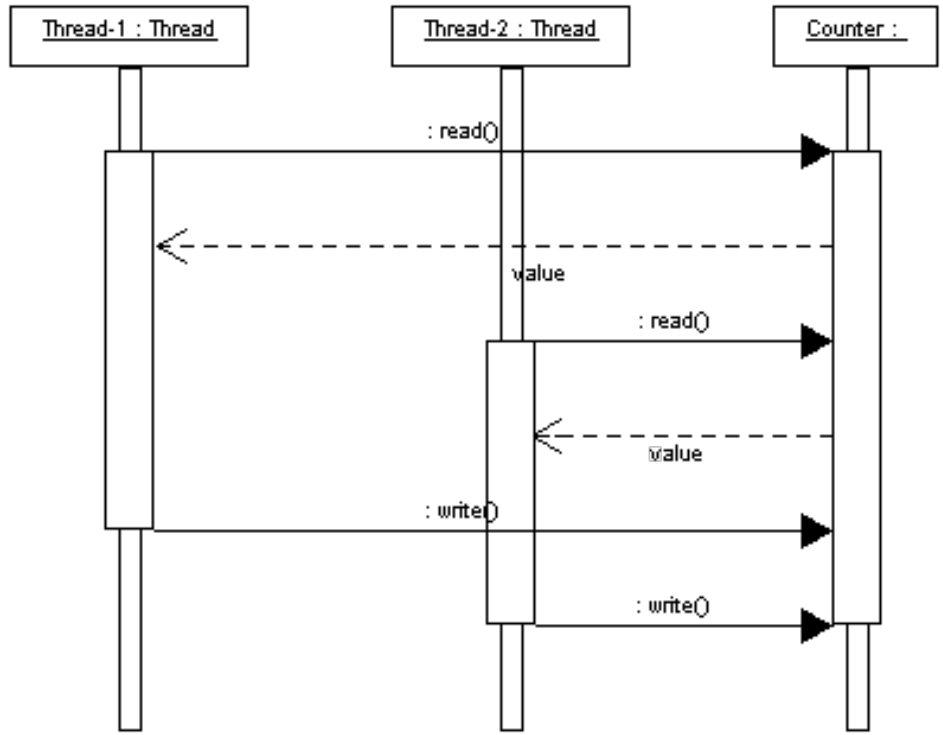
Verhalten: Das Beispiel definiert eine nebenläufige Operation, durch welche ein int-Zählerstand aus einer Datei gelesen und um eins erhöht in dieselbe Datei zurückgeschrieben wird. **Beobachtung:** Entgegen der intuitiven Erwartung weist der Zählerstand in der Datei nach erfolgreichem Ende der Ausführung aller erzeugten Threads nicht den vermuteten Wert von 10*Anzahl Threads auf.

Lese und Schreiboperationen durcheinander

```
thread named Thread-0 wrote: 10
thread named Thread-2 read: 10
thread named Thread-4 read: 10
thread named Thread-3 read: 4
thread named Thread-1 read: 9
thread named Thread-2 wrote: 11
thread named Thread-0 read: 10
thread named Thread-1 wrote: 10
```

Was ist passiert?

Analyse: Durch Unterbrechung eines Threads nach dem Einlesen des Wertes – noch vor dem Rückschreiben des inkrementierten Zählerstandes – kann ein anderer Thread zur Ausführung gelangen, durch den nochmals der bereits gelesene Zählerstand aus der Datei verarbeitet wird.
Später wird durch beide Threads derselbe erhöhte Variablenwert rückgeschrieben; ein Erhöhungsvorgang ist daher verloren, da er auf veralteten (da bereits (teil-)verarbeiteten) Daten beruht.



Definitionen

Die sich zeitlich überscheidenden Threads haben auf inkonsistente Datenstände Zugriff erlangt. Dies muss jedoch in der Ausführung nicht immer der Fall sein.

Definition: Race Condition

Hängt das Ergebnis der gesamten Programmausführung von der Abarbeitungsreihenfolge der Einzelthreads ab, spricht man von einer **race condition**.

Definition: Atomar

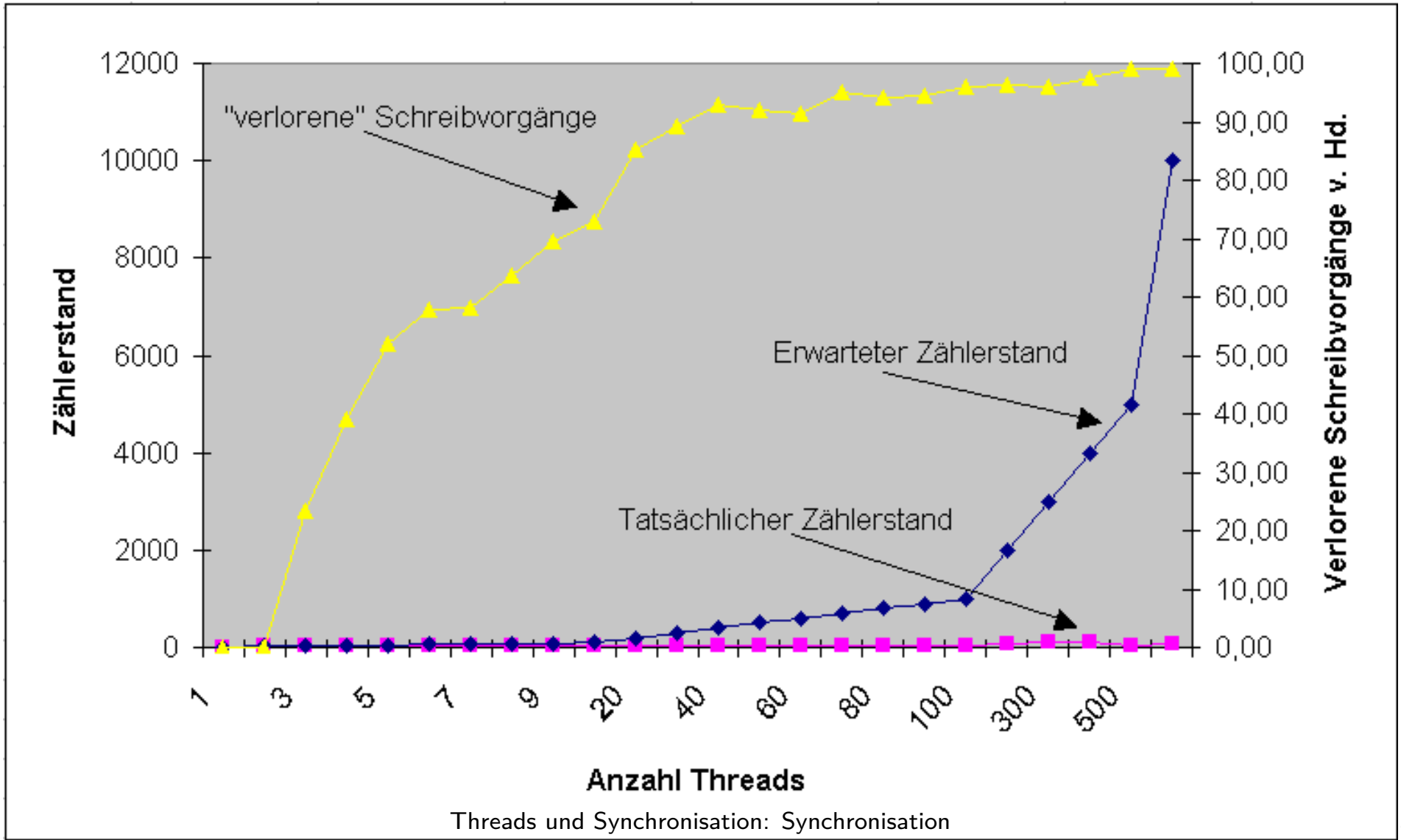
Eine Routine heißt **atomar**, wenn sie nicht in separate kleinere Einheiten unterteilt werden kann, die während ihrer Ausführung unterbrochen werden können.

Definition: Kritischer Bereich

Eine Folge von Anweisungen, deren Ausführung nicht nebenläufig erfolgen kann oder sollte, heißt **kritischer Bereich**.

Lost Update Statistik

Die Effekte von race conditions wirken sich bei niedriger Parallelität fast nicht, bei hoher Parallelität aber sehr stark aus.



Monitore

Wechselseitiger Ausschluss

Definition: Wechselseitiger Ausschluss

Wechselseitiger Ausschluss (mutual exclusion) stellt sicher, dass sich zu jedem Zeitpunkt höchstens eine Programmeinheit (in unserem Falle: Thread) in einem kritischen Abschnitt befindet.

- Hierfür wird der Abschnitt durch einen Sperrmechanismus gesichert, der überwunden werden muss (d.h. die Sperre wird gesetzt) bevor die Ausführung der kritischen Anweisungen gestattet wird. Nach dem Ausführungsende (d.h. Verlassen des kritischen Abschnitts) wird die Sperre wieder freigegeben.
- Während der Ausführungszeit erfolgende weitere Zugriffsversuche werden an der Sperre blockiert und verzögert bis die in Ausführung befindliche Programmeinheit den kritischen Abschnitt verlassen hat.
- Durch wechselseitigen Ausschluss der nebenläufigen Ausführung von Programmteilen entsteht **Illusion einer atomaren Anweisung**.

Monitore nach C.A.R. Hoare

Grundidee

- Methoden oder Blockbearbeitungen werden überwacht, es gibt jeweils einen Schlüssel, der zur Bearbeitung abgeholt und nach der Bearbeitung zurückgegeben wird.

Umsetzung in Java

- Durch Anbringung des Schlüsselwortes **synchronized** in der Signatur einer Methode wird die virtuelle Maschine veranlasst ein so gekennzeichnete Methode nicht nebenläufig auszuführen.
- Alle Aufrufe werden strikt serialisiert und ggf. verzögert bis die Methode zum alleinigen Zugriff zur Verfügung steht.

Monitore in Java

- Ein Monitor ist die Kapselung eines kritischen Bereiches mit Hilfe einer automatisch verwalteten Sperre
- Jedes von der JVM angelegte Objekt verfügt über eine Sperre
- Schlüsselwort **synchronized** markiert Methoden in einer Klasse, für die ein Thread vor Ausführung die Sperre verlangen muss

Beispiel Monitor

```
class Sample {
  synchronized void say(String words){
    // nur durch einen Thread
    // zur Zeit ausgeführt
  }
}
```

Sperre wird von einem Objekt angefordert

- Exemplar-Methode ⇒ Exemplar (Objekt) mit dem die Methode aufgerufen wird
- Klassen-Methode ⇒ Class-Objekt der entsprechenden Klasse
- In synchronized Blöcken ⇒ durch spezifiziertes Objekt

Beispiel synchronized Block

```
class Sample {
 void say(String words){
   synchronized(this) {
    // kritischer Bereich
   }
}}
```

- Benutzen mehrere Methoden/synchronized Blöcke **dasselbe** Sperrobjekt, kann zur Zeit immer nur **einer** der Methoden/synchronized Blöcke ausgeführt werden.

Beispiel Synchronisation

```
class Zaehlen extends Thread{
  public static long i=0;

  public static synchronized void erhoehen(){i++;}

  public void run(){
    for(int j=0;j<200000;j++) erhoehen();
  }

  public static void main(String[] args)
                      throws InterruptedException {
    Zaehlen[] z = new Zaehlen[10];
    for (int i=0;i<10;i++) z[i]=new Zaehlen();
    for (int i=0;i<10;i++) z[i].start();
    for (int i=0;i<10;i++) z[i].join();
    System.out.println(Zaehlen.i);
  }
}
```

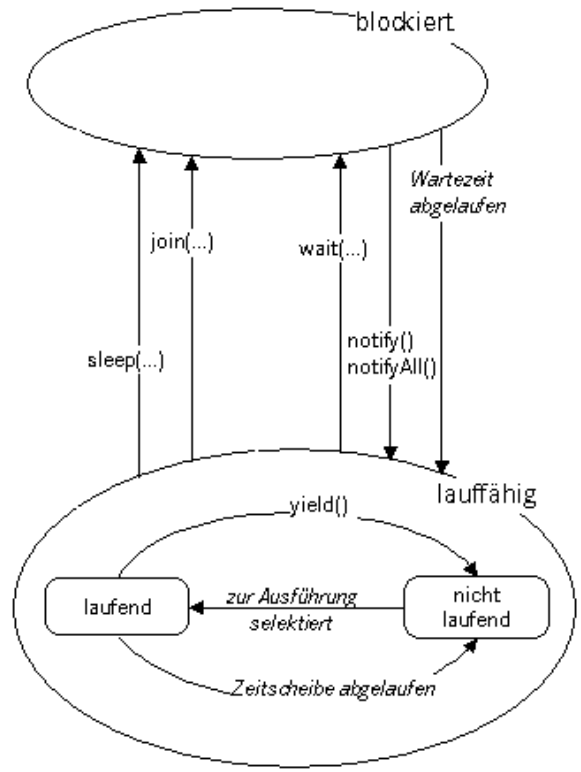
- erhoehen() ist kritischer Bereich
- mögliche Ausgaben ohne synchronized:
  1.511.194
  1.823.645
  1.149.064
- Ausgabe mit synchronized:
  2.000.000
- mit synchronized dauert es länger

Erzeuger-Verbraucher Probleme

- Auch bekannt als: Leser-Schreiber-Problem
- Typische Beziehung: Genau ein oder mehrere Produzenten stellen eine Ware oder Dienstleistung zur Verfügung, die von genau einem oder mehreren Konsumenten entgegengenommen werden.
- Erzeugen und Verbrauchen greift auf einen kritischen Bereich zu - typischerweise ein Speicherbereich, in dem das erzeugte Produkt hinterlegt und das zu verbrauchende entnommen wird.
- Hinterlegen und Entnehmen sind kritische Bereiche, die wechselseitig ausgeschlossen werden sollten.
- Doch was, wenn der Speicherbereich leer ist, und ein Teil entnommen werden soll? Und was wenn der Speicherbereich voll ist, und ein Produkt hinterlegt werden muss?
- wait() und notify() stellen weitere Synchronisationsmechanismen zur Verfügung, damit ein Erzeuger auf einen freien Platz oder ein Verbraucher auf ein hinterlegtes Produkt warten kann.

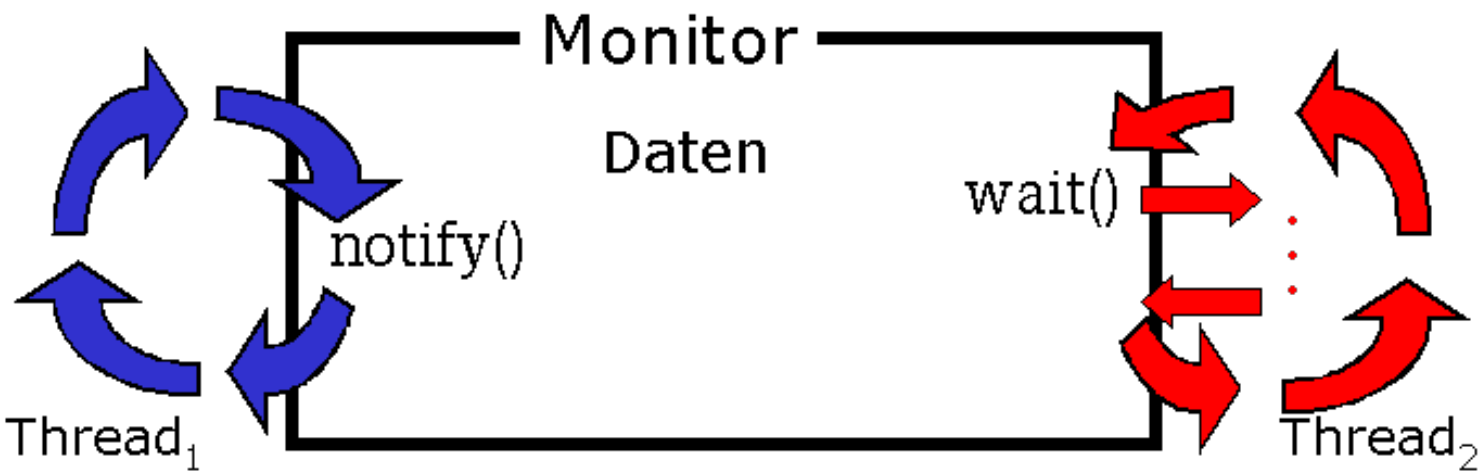
wait()

- object.wait() darf nur aufgerufen werden, wenn der Thread die Sperre zu object besitzt, z.B. innerhalb einer synchronized Methode.
- Methodenaufruf wait() bewirkt: Thread unterbricht Bearbeitung des synchronisierten Blocks und geht in den blockiert Zustand
- Sperre für den synchronisierten Block wird freigegeben
- erst wenn ein anderer Thread notify() aufruft, wird der blockiert Zustand wieder verlassen, und der Thread ist lauffähig
- der Thread läuft unter der Kontrolle des Monitors im wechselseitigen Ausschluss weiter
- wait() muss in einem try-catch-Block stehen



notify()

- object.notify() darf nur aufgerufen werden, wenn der Thread die Sperre zu object besitzt, z.B. innerhalb einer synchronized Methode.
- **ein** mit wait() in den schlafenden Zustand versetzter Thread, kann wieder die Sperre des entsprechenden Objektes anfordern
- notify() aufrufender Thread läuft normal weiter und gibt in Folge schließlich die Sperre frei, indem es einen synchronisierten Bereich verlässt
- aufgeweckter Thread kann (muss aber nicht) als nächstes die Sperre erhalten und führt die Bearbeitung fort



Beispiel vereinfachter Spooler

```
public class Spooler {
  private ArrayList v = new ArrayList();
    public synchronized void write(Object o) {
    if (v.size() == 3) try {
      System.out.println("writer␣waits"); wait();
    } catch (InterruptedException e) { }
    v.add(o); System.out.println("Adding:␣"+o+"␣Level:␣" +(v.size()));
    notify();
  }

  public synchronized Object read() {
    if (v.size() == 0) try {
      System.out.println("reader␣waits"); wait();
    } catch (InterruptedException e) { }
    Object ergebnis = v.remove(0);
    System.out.println("Level:␣"+v.size()+"␣Removing:␣"+ergebnis);
    notify();
    return ergebnis;
  }
}
```

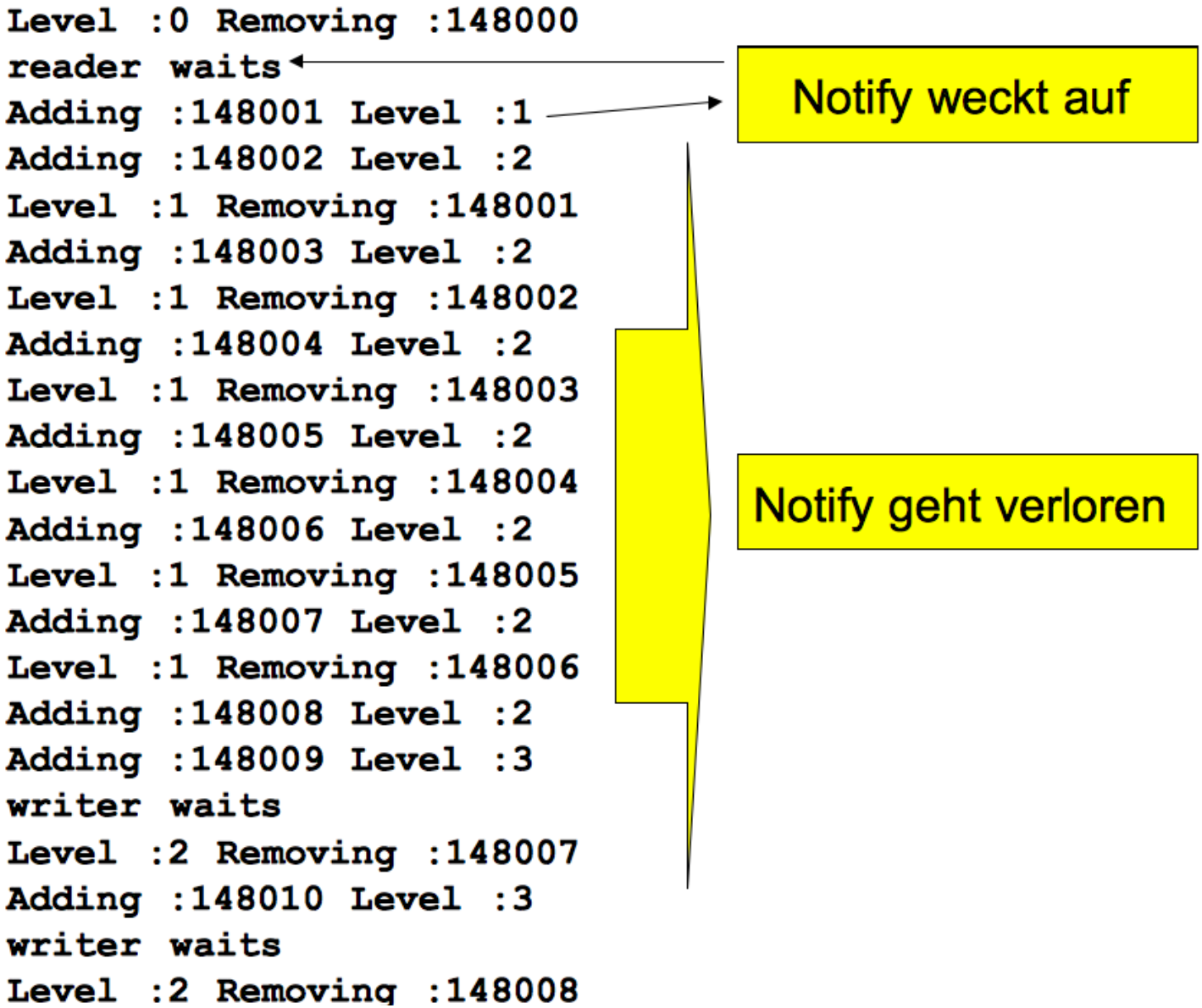
Beispiel vereinfachter Spooler

```
public class Producer implements Runnable {
  private Spooler spooler;
  public Producer(Spooler s) { spooler = s;}
  public void run() { int i=0; while(true) spooler.write(i++);}
}

public class Consumer implements Runnable {
  private Spooler spooler;
  public Consumer(Spooler s) { spooler = s; }
  public void run() { while(true) spooler.read(); }
}

public class ProducerConsumer {
  public static void main(String[] args) {
    Spooler s = new Spooler();
    new Thread(new Producer(s)).start();
    new Thread(new Consumer(s)).start();
  }
}
```

Ausgabe Spooler



Semaphore

Klassische Synchronisation nach Dijkstra

- Semaphore als Mechanismus für die Prozesssynchronisation wurden von Edsger W. Dijkstra konzipiert und 1965 in seinem Artikel Cooperating sequential processes vorgestellt.
- Semaphore unterstützen zwei Operationen
  - P (Passeren)
  - V (Vrijgeven)
- Binäre Semaphore: Es kann immer nur ein Thread P aufgerufen, dann muss V aufgerufen werden, bevor P aufgerufen werden kann. Andernfalls blockiert P bis zum Aufruf von V.
- Zählende Semaphore haben einen Zähler, der durch V hochgezählt und durch P heruntergezählt wird. Bevor der Zähler negativ wird, werden P Aufrufe blockiert.

Semaphore in Java

```
public class ConnectionLimiter {
  private final Semaphore semaphore;
  private ConnectionLimiter(int maxConcurrentRequests) {
    semaphore = new Semaphore(maxConcurrentRequests);
  }
  public URLConnection acquire(URL url) throws InterruptedException,
                                               IOException {
    semaphore.acquire();
    return url.openConnection();
  }
  public void release(URLConnection conn) {
    try {
      // clean up here
    } finally {
      semaphore.release();
    }
  }
}
```

Aufgabe

Aufgabe 1/2

- Legen Sie ein neues Projekt in Ihrer IDE für die folgende Aufgabe an.
- Schreiben Sie eine Klasse `Account` mit Methoden zum Einzahlen, Abheben und Kontostand abfragen (Jede Änderung des Kontostands soll auch auf der Konsole ausgegeben werden).
- Schreiben Sie einen Thread `Transferer`, der in einer Schleife kontinuierlich einen Betrag von einem Konto auf ein anderes Konto überweist.

Aufgabe 2/2

- Schreiben Sie einen Thread `BalanceChecker`, der für zwei gegebene Konten immer wieder überprüft, ob die Summe der Beträge der Konten konstant bleibt und sonst das Programm abbricht (wenn es sein muss mit System.exit(0), aber besser über Threadunterbrechung). Das Testergebnis soll immer ausgegeben werden.
- Schreiben Sie eine Klasse Main, die einen `Transferer` mit zwei Konten und dazu einen `BalanceChecker` mit den gleichen Konten startet. Bricht der Prüfer Ihr Programm ab? (Er sollte! Warum?)
- Ändern Sie Ihr Programm unter Nutzung von synchronized-Blöcken so ab, dass es nicht abbricht.
- Entwickeln Sie eine Lösung, welche Semaphoren statt synchronzied-Blöcken nutzt.

Zusatzaufgabe

Erzeuger Verbraucher Problem

- Clonen Sie aus gitlab die Übung `erzeuger-verbraucher`
- Lösen Sie die in der README.md beschriebene Aufgabestellung.