Multithreading

PROG 2: Einführung in die Programmierung für Wirtschaftsinformatiker

Dr.-Ing. Steffen Helke
Technische Universität Berlin
Fachgebiet Softwaretechnik

3. Juni 2013

Übersicht

- Threads in Java
- Datenaustausch zwischen Threads
- Synchronisation von Threads

Teil IV der Vorlesung PROG 2

Multithreading

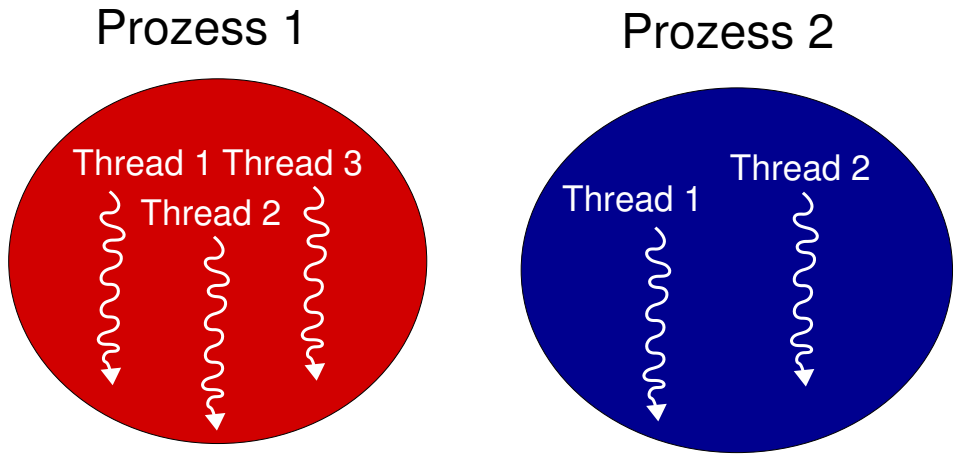
Motivation

Quelle: *Inhalt & Gestaltung nach Vorlesungsfolien von Peter Pepper und Odej Kao, TU Berlin*
Methodische- und Praktische Grundlagen der Informatik 4 (MPGI 4), WS 2010/11 bzw. WS 2011/12

Nebenläufige Programme mit Threads

Definition Threads

- auch leichtgewichtige Prozesse
- mehrere parallel ablaufende Aktivitäten innerhalb eines Prozesses



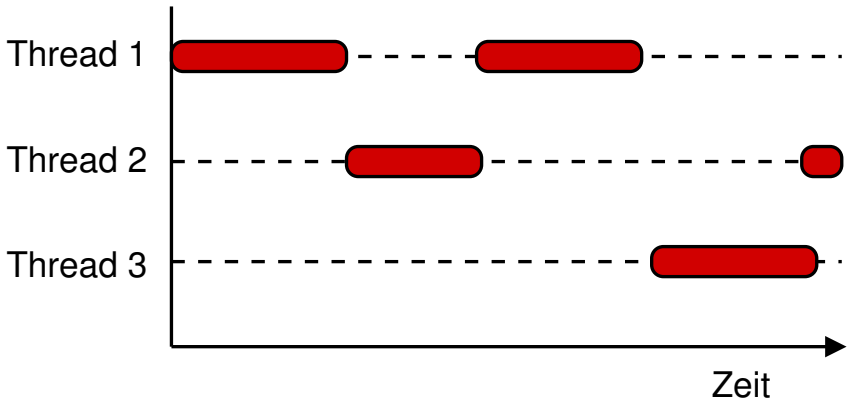
Technische Besonderheiten

- jeder Thread realisiert einen sequentiellen Kontrollfluss, d.h. besitzt seinen eigenen Programmzähler
- im Gegensatz zu Prozessen gemeinsame Ressourcen, d.h. Threads besitzen gemeinsamen Adressraum

Prozessorbelegung für Threads

Realisierung

- nebenläufige Programme führen mehrere Rechenschritte gleichzeitig (simultan) durch
- Abarbeitung erfolgt verschränkt, oft auf einem Prozessor



Vorteile von Threads

- schnelles Umschalten zwischen Anwendungen auf unterschiedlichen Threads
- gemeinsame Nutzung von Ressourcen ist einfach

Probleme beim Einsatz von Threads

Nachteile

- Programmausführung zeigt *oft nicht-deterministisches Verhalten* (schwierig beim Test nebenläufiger Programme)
- *Konflikte beim Schreiben gemeinsamer Ressourcen*, wie z.B. beim Zugriff auf gemeinsame globale Variable oder eine Datei

Lösung

- Einsatz verschiedener Synchronisationsmechanismen, um zumindest teilweise deterministisches Systemverhalten zu erzwingen
- nur kontrollierten Zugriff auf gemeinsame Variablen erlauben, z.B. mit Hilfe von Semaphoren

Beispiel zum Einsatz von Multithreading

Wozu sind Threads auf einem Prozessor sinnvoll?

- oft ist der Prozessor nicht ausgelastet
- z.B. bei der Eingabe von Daten durch einen Benutzer
- Idee: Analysen im Hintergund durchführen lassen

Beispiel Computerspiel (Sequentielle Abarbeitung)

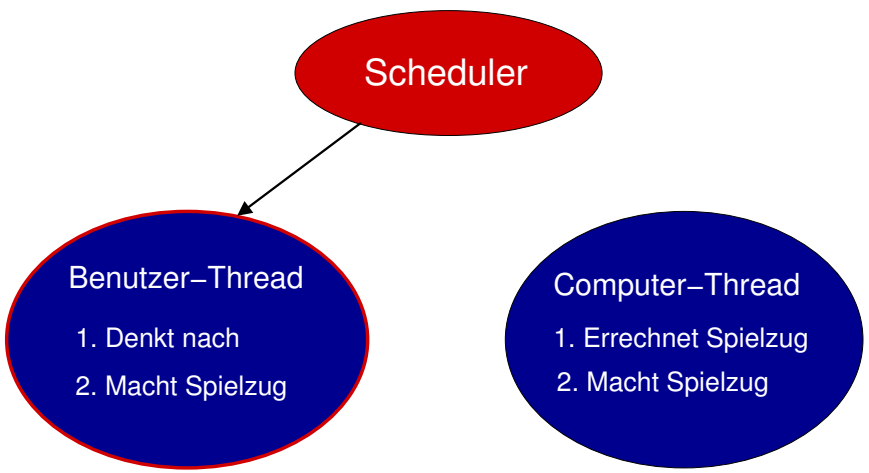
1. Spieler (Benutzer) denkt über nächsten Spielzug nach
2. Spieler gibt Spielzug ein
3. Computer errechnet nächsten Spielzug
4. Computer macht seinen Spielzug

Computerspiel (Nebenläufige Abarbeitung)

1. Spieler (Benutzer) denkt über nächsten Spielzug nach
2. Spieler gibt nächsten Spielzug ein

1. Computer errechnet/plant nächsten Spielzug
2. Computer macht seinen Spielzug

Ressourcen-Zuteilung durch einen Scheduler

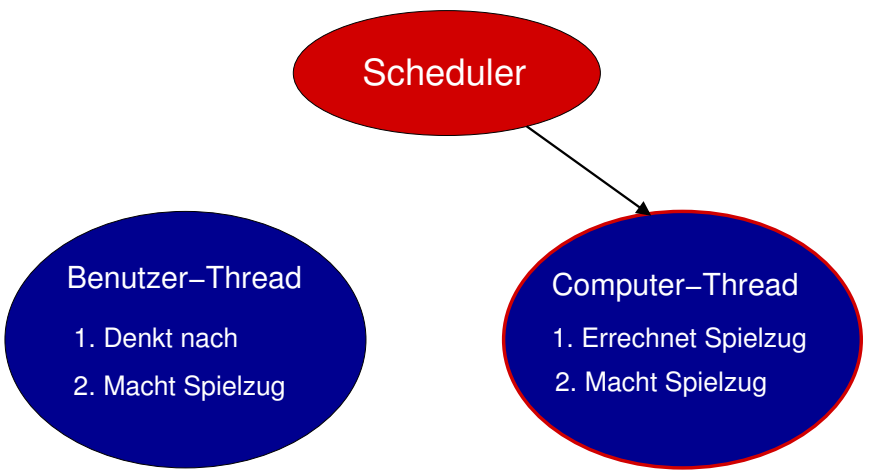


Computerspiel (Nebenläufige Abarbeitung)

1. Spieler (Benutzer) denkt über nächsten Spielzug nach
2. Spieler gibt nächsten Spielzug ein

1. Computer errechnet/plant nächsten Spielzug
2. Computer macht seinen Spielzug

Ressourcen-Zuteilung durch einen Scheduler

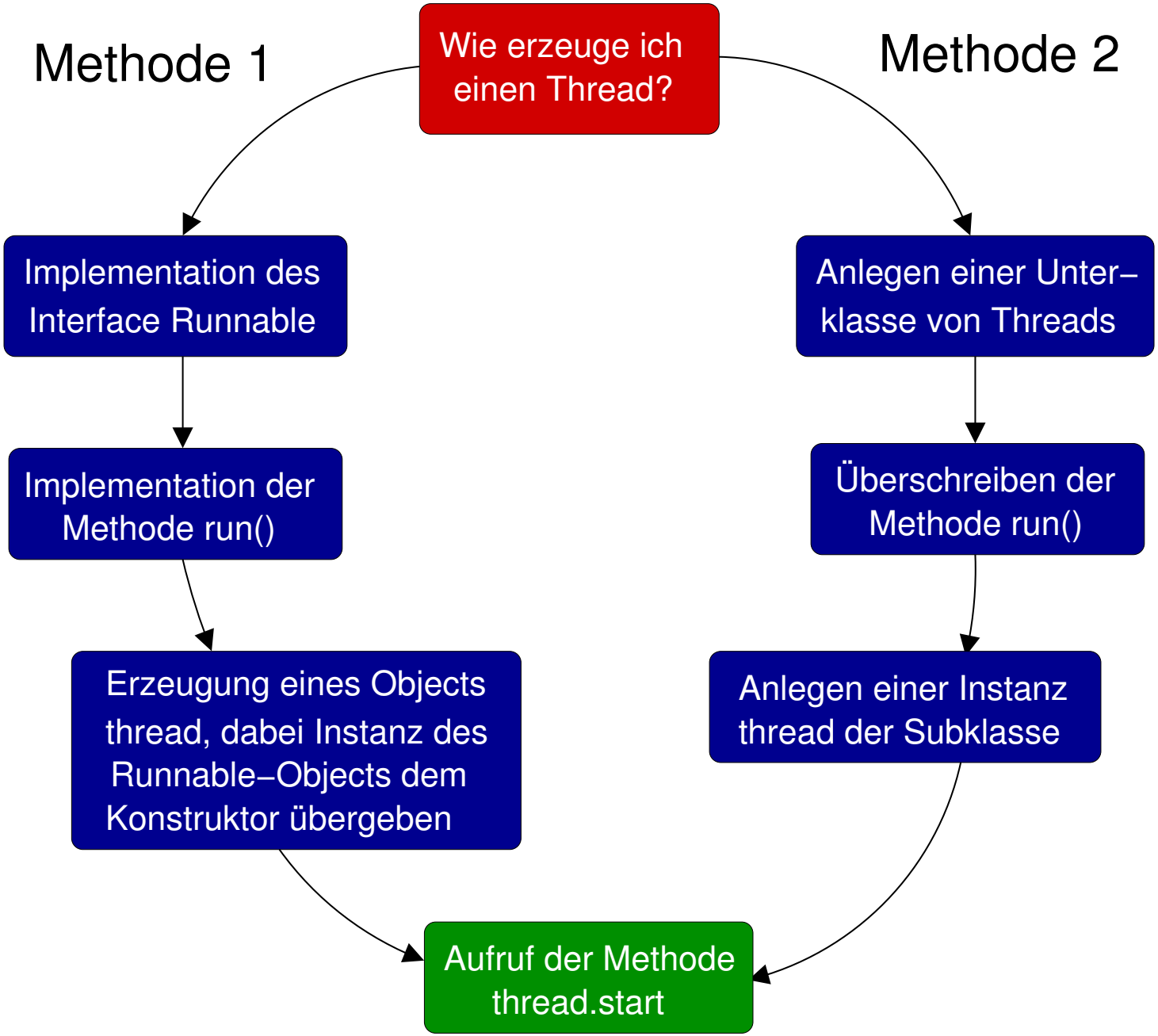


Teil IV der Vorlesung PROG 2

Multithreading

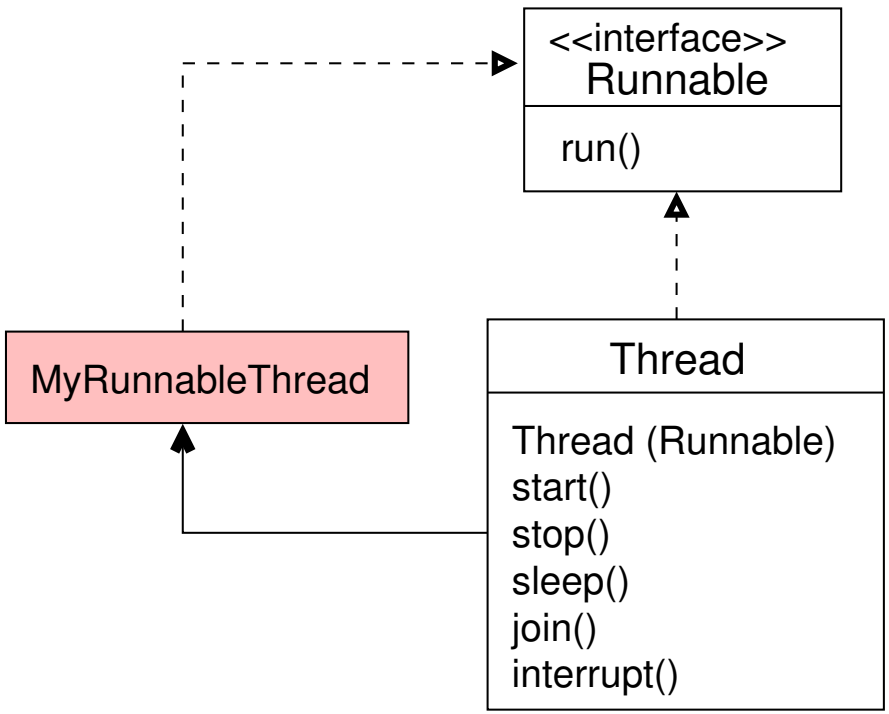
Threads in Java

Quelle: *Inhalt & Gestaltung nach Vorlesungsfolien von Peter Pepper und Odej Kao, TU Berlin*
Methodische- und Praktische Grundlagen der Informatik 4 (MPGI 4), WS 2010/11 bzw. WS 2011/12

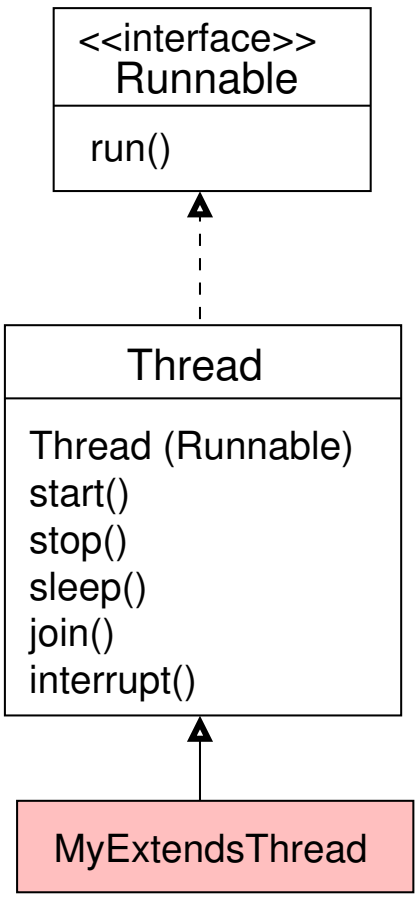

Wie erzeuge ich einen Thread?
Methode 1
Methode 2
Implementation des Interface Runnable
Implementation der Methode run()
Erzeugung eines Objects thread, dabei Instanz des Runnable–Objects dem Konstruktor übergeben
Anlegen einer Unter–klasse von Threads
Überschreiben der Methode run()
Anlegen einer Instanz thread der Subklasse
Aufruf der Methode thread.start

Varianten zur Erzeugung von Threads in Java

Methode 1



Methode 2



Methode 1: Erzeugung von Threads

```
import java.util.Random;

public class RunnableThread implements Runnable  {
  private int sleepTime;
  private Random generator = new Random();

  public RunnableThread() {
    sleepTime = generator.nextInt( 1000 ); }
  public void run() {
    // Thread geht schlafen
    try { Thread.sleep(sleepTime);}
    catch (InterruptedException exception) {...}
    // Thread ist wieder aufgewacht
  }

  public static void main(String[] args) {
    // Erzeuge neue Threads
    Thread thread1 = new Thread(new RunnableThread());
    Thread thread2 = new Thread(new RunnableThread());
    // Starte neue Threads
    thread1.start(); thread2.start();   ... }
}
```

Methode 2: Erzeugung von Threads

```
import java.util.Random;

public class HelloThread extends Thread  {
  private int sleepTime;
  private Random generator = new Random();

  public void run() {
    sleepTime = generator.nextInt(1000);
    // Thread geht schlafen
    try { Thread.sleep(sleepTime);}
    catch (InterruptedException exception) {...}
    // Thread ist wieder aufgewacht
  }

  public static void main(String[] args) {
    // Erzeuge neue Threads
    HelloThread thread1 = new HelloThread();
    HelloThread thread2 = new HelloThread();
    // Starte neue Threads
    thread1.start(); thread2.start(); ...
  }
}
```

Welche Methode ist zu bevorzugen?

Thread-Erzeugung: Methode 1 vs. Methode 2

- Methode 2 mit dem Erweitern der Thread Klasse durch Vererbung wird als ungünstiger eingeschätzt
- Grund: Es können künstliche bzw. konstruierte Klassenhierarchien entstehen

➜ Methode 1 mit der Implementierung von Interfaces ist zu bevorzugen

➜ Methode 1 kann auch mit *e.execute(Runnable-Object)* gestartet werden, wobei *e* hier ein *Executor*-Objekt ist

➜ *Executor*-Objekte sind zur Umsetzung von Thread-Pools (Abarbeitung unterschiedlicher Runnable-Objekte) gedacht

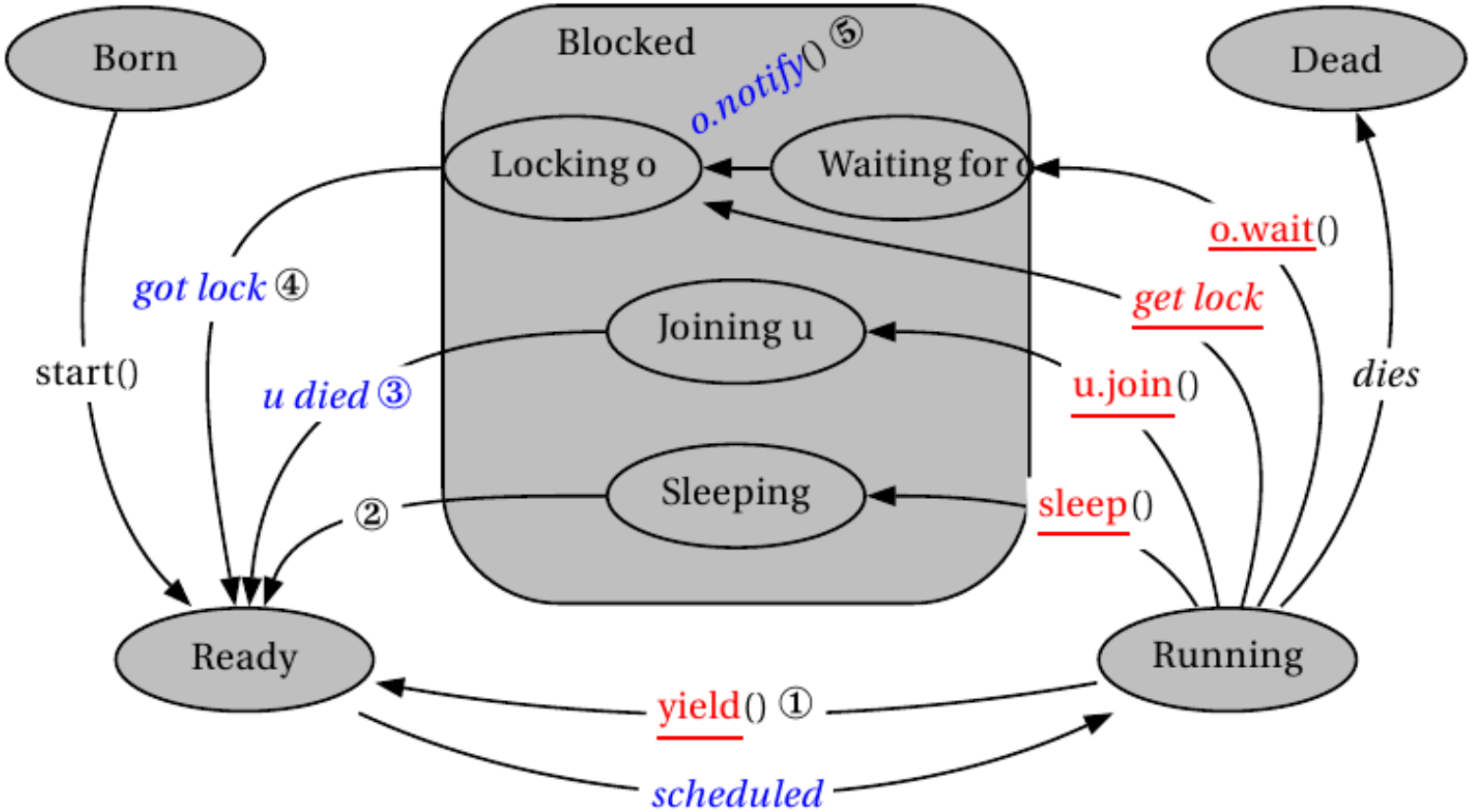
Erzeugung von Threads mit Executor

```
import java.util.Random;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class RunnableThread implements Runnable {
  private int sleepTime;
  private Random generator = new Random();

  public RunnableThread() { ... }

  public static void main(String[] args) {
    // Erzeuge neue RunnableThreads
    RunnableThread thread1 = new RunnableThread();
    RunnableThread thread2 = new RunnableThread();
    // Erzeuge Manager fuer die drei Threads
    ExecutorService
        threadExecutor = Executors.newCachedThreadPool();
    // Starte neue Threads
    threadExecutor.execute(thread1);   ... }
    // Beende den Starter-Thread
    threadExecutor.shutdown();
}
```

Thread-Zustände: Abfrage mit getstate()

1. **NEW**
   - Thread ist bereits erstellt, aber *start()* noch nicht aufgerufen
2. **RUNNABLE**
   - Thread wird gerade ausgeführt
3. **BLOCKED**
   - Thread wird nicht ausgeführt, da er auf eine Ressource wartet
4. **WAITING**
   - Thread wird nicht ausgeführt, da *Object.wait()* oder *Thread.join()* aufgerufen wurde
5. **TIMED_WAITING**
   - Thread wird nicht ausgeführt, da *Thread.sleep()*, *Object.wait()* oder *Thread.join()* mit Timeout aufgerufen wurde
6. **TERMINATED**
   - Ausführung ist beendet, d.h. *run()* wurde komplett abgearbeitet oder durch Auslösen einer Exception beendet

Die Zustände eines Threads



rot = vom Thread selbst ausgeführt

blau = von einem anderen Thread ausgeführt

Stoppen von Threads

Intention

- Threads vor Ende der Abarbeitung beenden
- Umsetzung durch Aufruf der Methode *stop()*
- Achtung: Methode ist *deprecated*, d.h. nicht verwenden!

```
public class HelloThread extends Thread  {
  private int sleepTime;
  private Random generator = new Random();
  public void run() { ... }

  public static void main(String[] args) {
    // Erzeuge neuen Thread
    HelloThread thread1 = new HelloThread();
    // Starte neuen Thread
    thread1.start(); ...
    // Frühzeitiges Beenden des Threads
    thread1.stop();
}}
```

Alternative: Unterbrechen von Threads

- durch Aufruf von *interrupt()* wird dem Thread-Objekt mitgeteilt, dass es sich beenden soll (setzen eines Flags)
- mit der Methode *isInterrupted()* kann thread-Objekt nachschauen, ob es sich beenden soll

```
public class InterruptThread extends Thread  {
  private int sleepTime; ...
  public void run() {
    while (!this.isInterrupted()) {
     try { System.out.println("Hello World");
           Thread.sleep(sleepTime); }
 // entsteht bei Aufruf von interrupt in einer sleep-Phase
       catch (InterruptedException e) {
           this.interrupt(); }
  public static void main(String[] args) {
    InterruptThread thread1 = new InterruptThread();
    thread1.start(); ...
    // Unterbrechen des Threads
    thread1.interrupt(); }}
```

Warten auf einen Thread mit join

- Aufruf von *join()* auf einem Thread-Objekt *t* bewirkt, dass der aufrufende Thread mit der Abarbeitung wartet, bis *t* abgearbeitet ist

```
public class JoinThread extends Thread  {
  private int sleepTime; ...
  public void run() {
     try { System.out.println("Hello World");
           Thread.sleep(sleepTime); }
     catch (InterruptedException e) { ... }
  public static void main(String[] args) {
    JoinThread thread1 = new JoinThread();
    thread1.start(); ...
    // Warten auf Abarbeitung des Threads
    try { thread1.join(); }
    catch (InterruptedException e) {
        e.printStackTrace(); }
}
```

Teil IV der Vorlesung PROG 2

Multithreading

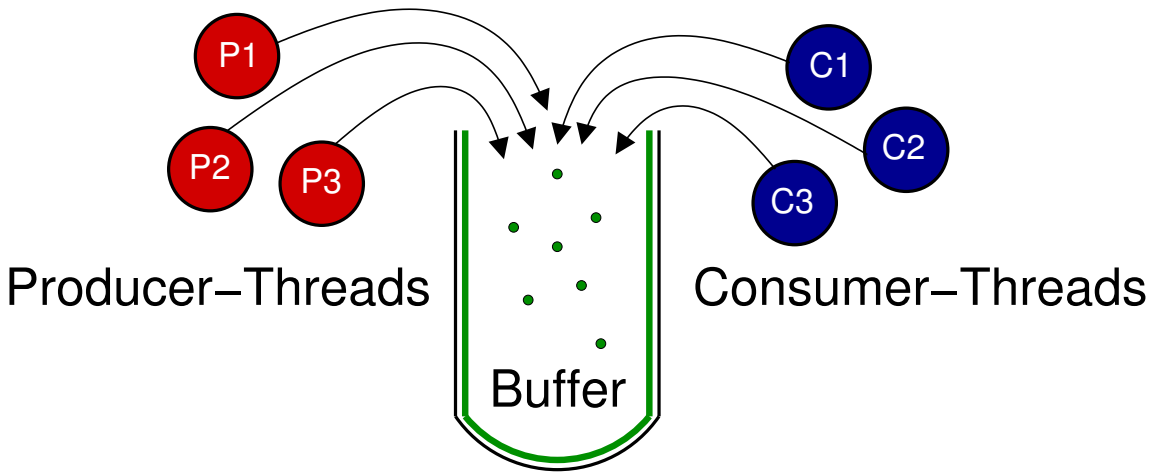
Producer-Consumer-Problem

Quelle: *Inhalt & Gestaltung nach Vorlesungsfolien von Andreas Rausch, TU Clausthal, Programmierkurs SS 2011*

Producer-Consumer-Problem

Problembereich

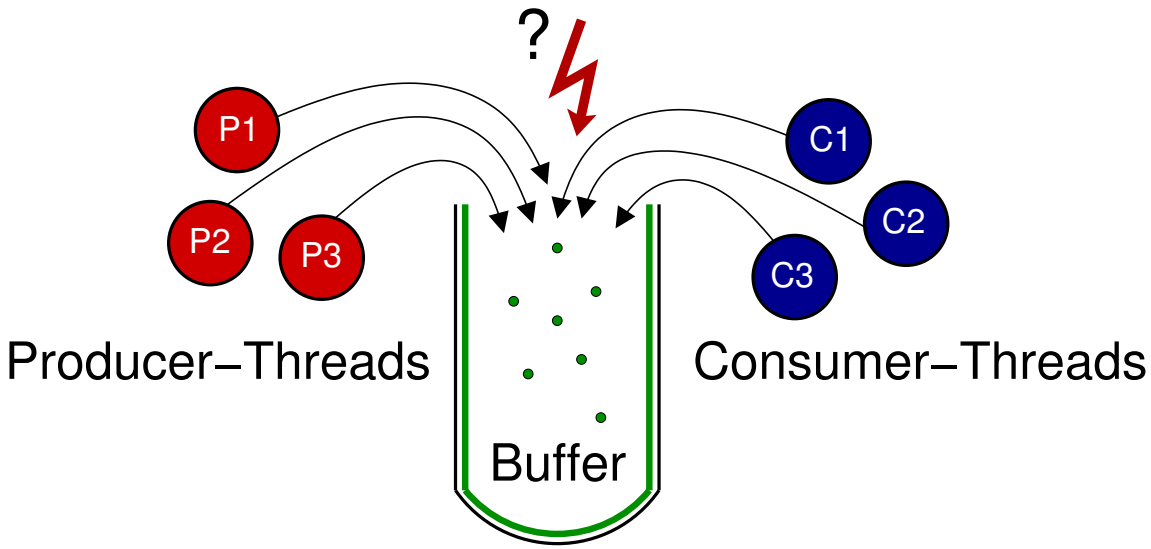
- eine feste Anzahl von Threads (Producer) erzeugen Elemente für gemeinsame Datenstruktur (Buffer)
- eine feste Anzahl von Threads (Consumer) entnehmen Elemente aus der gemeinsam genutzten Datenstruktur
- die Datenstruktur hat eine beschränkte Kapazität



Konfliktvermeidung

Synchronisationsbedarf

- der Lese- bzw. Schreibvorgang eines Threads darf nicht durch andere Threads unterbrochen werden (Inkonsistenzen)
- zugreifende Consumer-Threads werden blockiert, wenn die Datenstruktur leer ist
- zugreifende Producer-Threads werden blockiert, wenn die Datenstrukur voll ist



Gemeinsame Datenstruktur (Buffer)

```
public class Buffer {
  private Vector<String> queue;
  final int MAX = 3;

  public Buffer() {
    queue = new Vector<String>(); }

  public void transferElementToBuffer (String element) {
      if (queue.size() >= MAX) {
        throw new RuntimeException ("Kein Platz"); }
     queue.add(element);
     System.out.println(element + " wurde abgelegt");
  }
  public String takeElementFromBuffer () {
      if (queue.isEmpty()) {
        throw new RuntimeException ("Puffer leer"); }
     String element = queue.remove(0);
     System.out.println(element + " wurde entnommen");
     return element;
  }
}
```

Producer-Klasse zum Auffüllen des Buffers

```
public class Producer extends Thread {
 private Buffer buffer;
 private int sleepTime;

 public Producer (Buffer buffer, int sleepTime) {
   this.buffer = buffer;
   this.sleepTime = sleepTime; }

 public void run() {
   for (int i = 0; i < 10; i++) {
      try {
            Thread.sleep(sleepTime);
            String element = new String("Element " + i);
             buffer.transferElementToBuffer(element);
      } catch (InterruptedException e) {
              e.printStackTrace(); }
   }
  }
}
```

Consumer-Klasse zum Reduzieren des Buffers

```
public class Consumer extends Thread {
  private Buffer buffer;
  private int sleepTime;

  public Consumer (Buffer buffer, int sleepTime) {
    this.buffer = buffer;
    this.sleepTime = sleepTime; }

    public void run() {
      for (int i = 0; i < 10; i++) {
        try {
            Thread.sleep(sleepTime);
             buffer.takeElementFromBuffer();
        } catch (InterruptedException e) {
             e.printStackTrace();}
      }
    }
}
```

Consumer-Producer-Klasse

```
public class ConsumerProducerProblem {
  public static void main (String[] args) {
     Buffer buffer = new Buffer();
     Producer producer = new Producer(buffer, 200);
     Consumer consumer = new Consumer(buffer, 1000);

     producer.start();
     consumer.start();
  }
}
```

Probleme

- Exceptions werden sowohl beim Entnehmen, als auch beim Einfügen geworfen
- Zugriffe sind nicht atomar (Gefahr von Inkonsistenzen)

➜ zusätzliche Synchronisation nötig

Lösungsansatz: Einsatz von Semaphoren

Idee

- Mittel zur Synchronisation von nebenläufig ausgeführten Threads mit gemeinsamen Speicher
- Erfinder: Edsger W. Dijkstra

Bestandteile einer Semaphore

- Zähler
- Warteschlange
- Methode *P()* zum Dekrementieren des Zählers
- Methode *V()* zum Inkrementieren des Zählers

Abarbeitung beim Einsatz von Semaphoren

Vorgehen

1. Initialisierung des Zählers (maximal zugelassene Anzahl an Threads im kritischen Bereich)
2. vor Zugriff auf kritischen Bereich muss ein Thread t die Methode *P()* aufrufen
3. ist Zähler bereits 0, wird Thread t in Warteschlange eingefügt, sonst bekommt er Zugriff auf kritischen Bereich
4. ist Thread t fertig, so ruft er Methode *V()* auf
5. falls Threads in der Warteschlange, erhält der nächste Zugriff

Semaphoren in Java

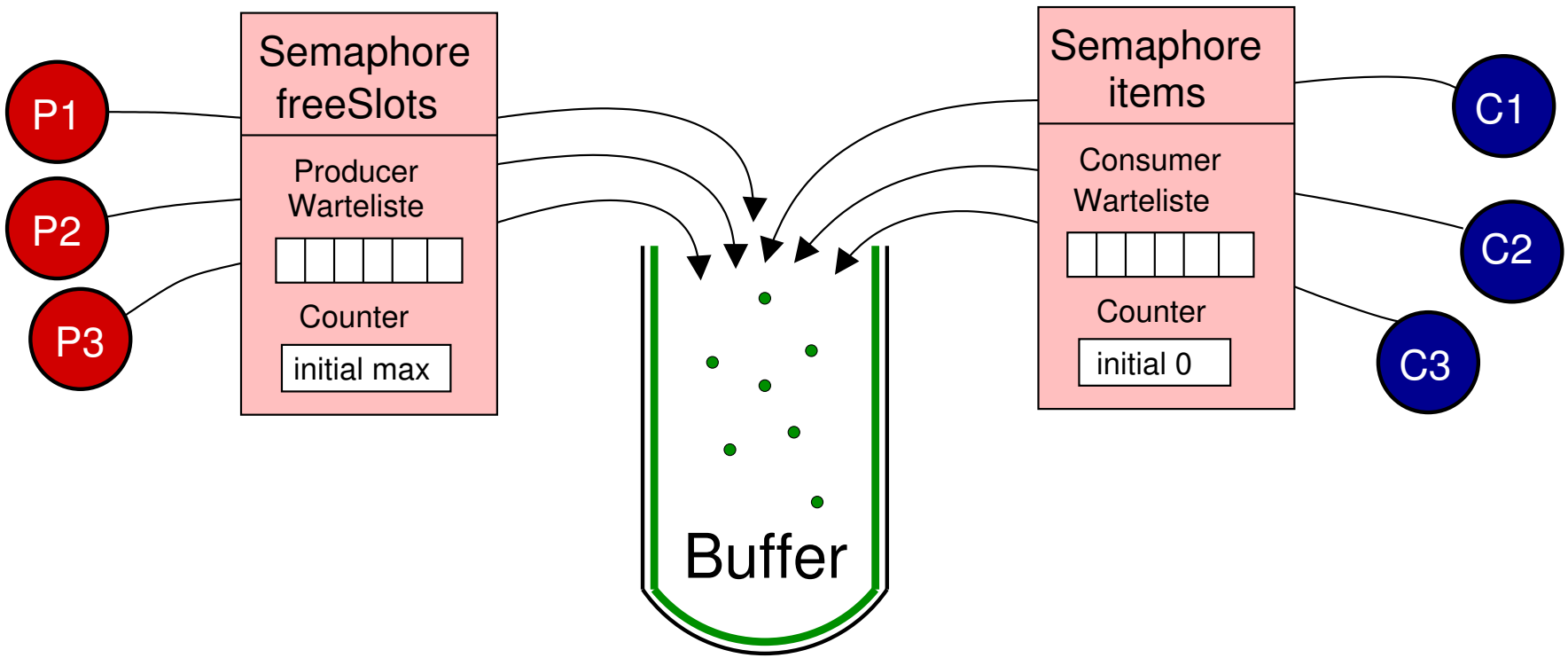
Notation

- Methoden *P()* und *V()* werden umbenannt
- zusätzliche Einführung von Methoden, zum Inkrementieren oder Dekrementieren um mehr als 1 vornehmen zu können

```
// Dekrementieren des internen Zählers
void acquire();

// Inkrementieren des internen Zählers
void release();

// Dekrementieren um mehrere Schritte
void acquire(int permits);

// Inkrementieren um mehrere Schritte
void release(int permits);
```

Semaphoren für Producer-Consumer-Beispiel

- Zugriffskontrolle mit Hilfe von zwei Semaphoren
- Producer-Kontrolle mit *freeSlots* (initial *max*)
- Consumer-Kontrolle mit *items* (initial 0)

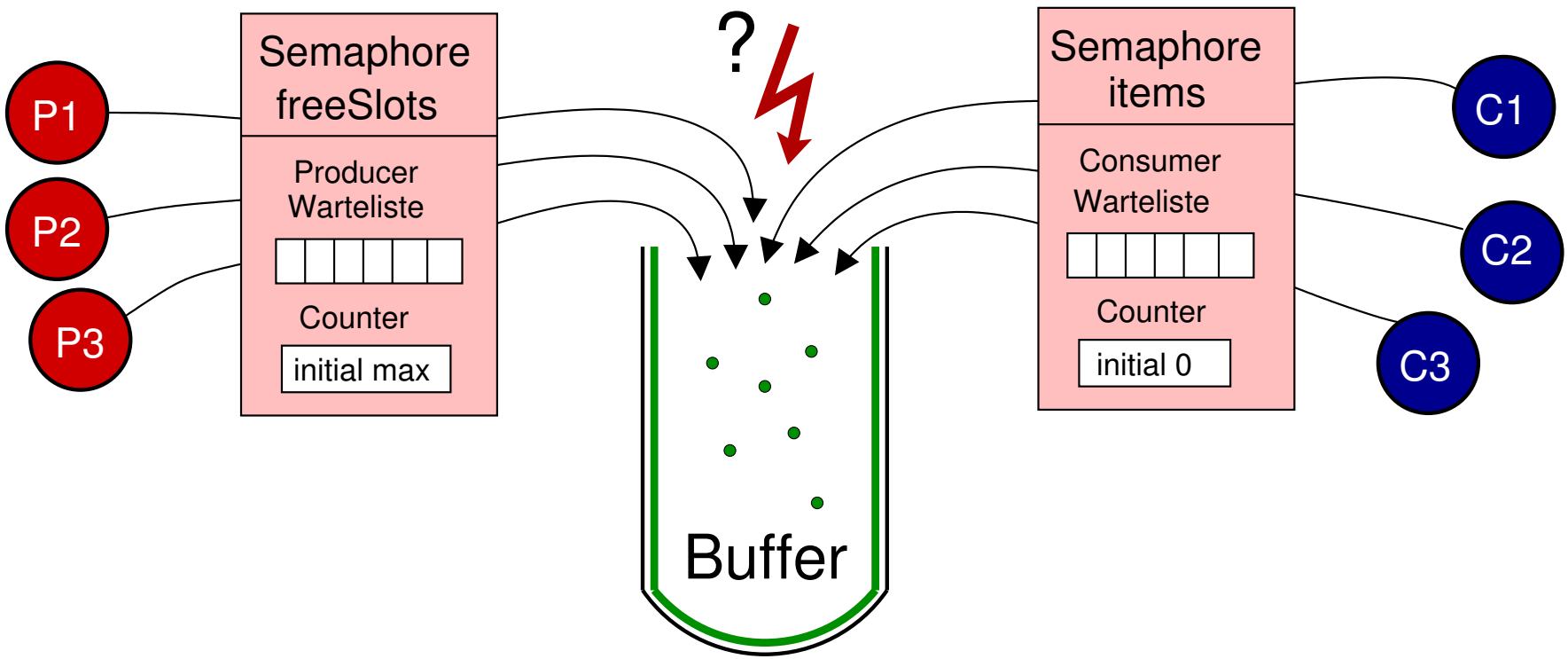


Datenstruktur mit Semphore (Buffer)

```
public class SemaphoreBuffer {
  private Vector<String> queue;
  private Semaphore items;
  private Semaphore freeSlots;
  final int MAX = 2;

  public SemaphoreBuffer() {
    queue = new Vector<String>();
    this.items = new Semaphore(0);
    this.freeSlots = new Semaphore(MAX);
  }
  public void transferElementToBuffer (String element) {
   try {
       freeSlots.acquire();
   } catch (InterruptedException e) {
       e.printStackTrace(); }
   queue.add(element);
   System.out.println(element + " wurde abgelegt");
   items.release();
  }
  public String takeElementFromBuffer () { ... }
}
```

Motivation für zusätzliches Monitoring

- Semaphoren verhindern nur Producer-Zugriff bei vollem bzw. Consumer-Zugriff bei leerem Puffer
- gleichzeitiger Zugriff von zwei Threads kann weiter zu Lese- oder Schreibkonflikten führen



Monitore in Java

- zusätzlich zu Semaphoren gibt es in Java Monitore
- Idee: kritischer Programmteil darf nur von einem Thread zu einem Zeitpunkt betreten werden
- Konzept: Setzen einer Sperre beim Betreten des kritischen Bereichs, wollen andere Threads auch in den Bereich, so müssen sie warten
- Umsetzung: Schlüsselwort *synchronized* vor eine Methode oder einen Programmblock setzen
- Hinweis: Sind mehrere Methoden mit *synchronized* markiert, so kann nur genau ein Thread gleichzeitig nur genau eine dieser Methoden zu einem Zeitpunkt ausführen

Monitoring mit synchronized

Probleme

- Gefahr von Deadlocks, wenn alle Prozesse auf einander warten müssen
- Beispiel: alle Producer/Consumer sind mit *synchronized* markiert

Lösung

- Entfernen der Semaphoren in der Implementierung
- Hinzufügen der Methoden *wait()* und *notify()*

Datenstruktur mit synchronized (Buffer)

```
public class MonitoredBuffer {
  private Vector<String> queue;
  final int MAX = 2;
  public MonitoredBuffer() {
    queue = new Vector<String>(); }

  public synchronized void
                  transferElementToBuffer (String element) {
    while (queue.size() == MAX) {
  // Warten auf notify-Aufruf eines anderen Threads
  // in dieser Wartezeit dürfen andere Threads
  // die synchronized-Methode ausführen
      try {
          wait();
      } catch (InterruptedException e) { ... }
   }
   queue.add(element);
   System.out.println(element + " wurde abgelegt");
  // Wenn Element hinzugefügt, anderen Thread aufwecken
   notify();
  }
  public String takeElementFromBuffer () { ... }}
```