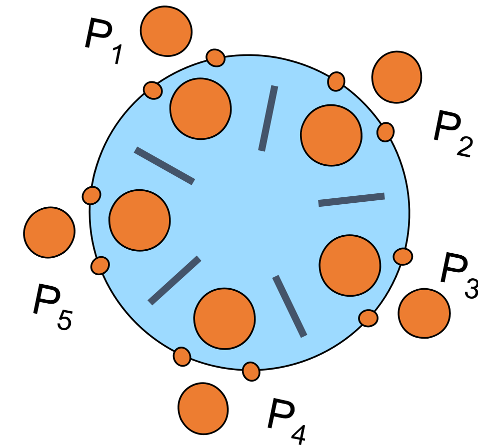
Vorlesung Programmieren
Thema 14:
Multithreading

Olaf Herden
Fakultät Technik
Studiengang Informatik

synchronized

notify



Stand: 05/2024

Gliederung

- Grundlagen
- Threads in Java
- Probleme nebenläufiger Programme (Konkurrenz)
- Synchronisation in Java mit `synchronized`
- Probleme nebenläufiger Programme (Kooperation)
- Synchronisation in Java mit `wait` und `notify`

Nebenläufigkeit

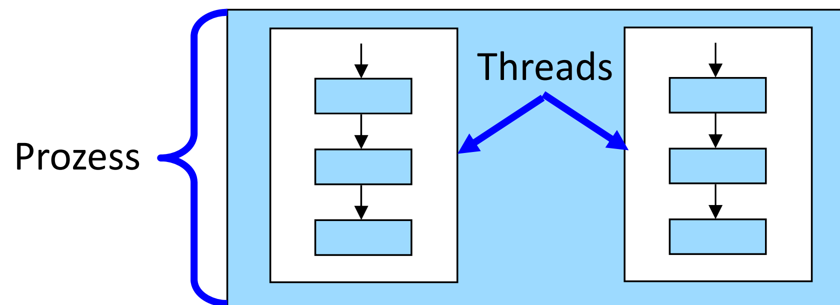
- Nebenläufigkeit bedeutet den gleichzeitigen (bzw. quasi-gleichzeitigen) Ablauf von zwei oder mehr Vorgängen
- Konzept in Java integriert
- Nicht-nebenläufige Programme besitzen eine einzige Befehlssequenz
- Nebenläufige Programme besitzen mehrere Befehlssequenzen (Prozesse oder Threads)

Prozesse und Threads (I)

- Prozess:
  - Folge von Befehlen
  - Von Prozessor ausführbar
  - Haben eigenen Adressraum, Programmzähler, Stapel und Registersatz
  - Prozessor kann schnell zwischen verschiedenen Prozessen umschalten (quasi-gleichzeitiger Ablauf)
- Thread:
  - Ähnlich einem Prozess
  - Eigenständiges Programmfragment, kann parallel zu anderen Threads ablaufen
  - Haben eigenen Programmzähler, Stapel und Registersatz
  - Besitzt einen der Zustände New, Ready, Running, Blocked oder Finished

Prozesse und Threads (II)

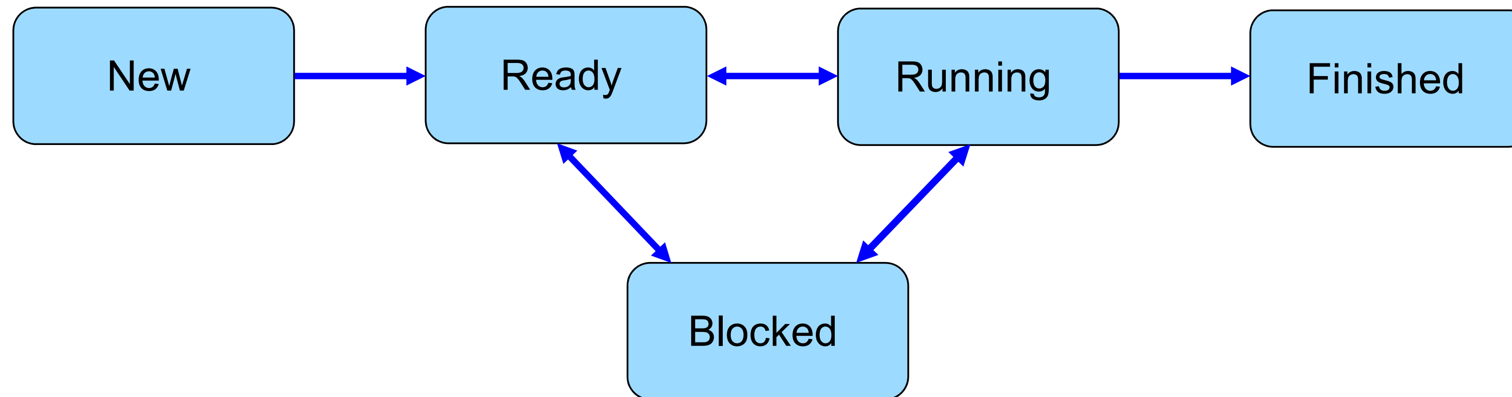
- Unterschied zwischen Thread und Prozess:
  - Threads teilen sich Adressraum, Prozesse sind voneinander getrennt
  - Erzeugen von Threads erheblich Ressourcen schonender als Erzeugen eines neuen Prozesses
  - Threads werden auch als leichtgewichtige Prozesse bezeichnet



Realisierung von Nebenläufigkeit

- Prozessor kann jeweils nur einen Prozess/Thread ausführen
- Mehrere Prozesse konkurrieren gleichzeitig um Zuteilung von Rechenzeit
- Scheduler: Komponente des Betriebssystems, die nach bestimmten Verfahren für Zuteilung des Prozessors sorgt
- In Java: Virtuelle Maschine mit eigenem Scheduler, koordiniert die Threads
- Dabei: Threads in verschiedenen Zuständen

Zustände von Threads



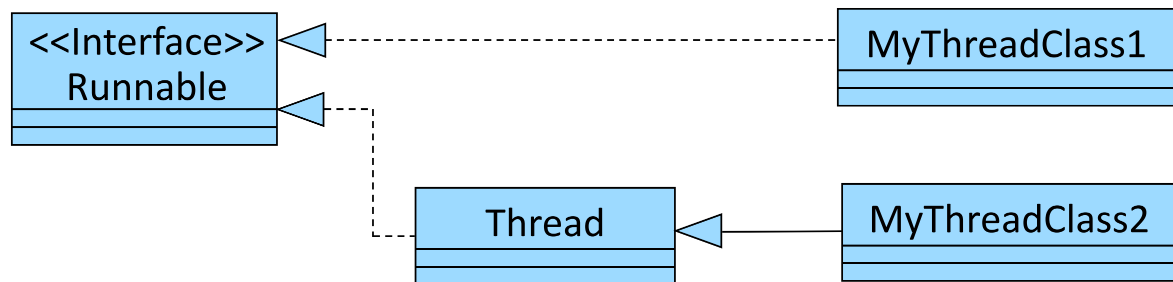
- New: Thread angelegt, aber noch nicht gestartet
- Ready: Thread gestartet, ist ablaufbereit, aber Scheduler hat anderen Threads Vorzug gegeben
- Running: Hat Ressourcen vom Scheduler zugeteilt bekommen, kann voranschreiten
- Blocked: Kann nicht weiter voranschreiten (Ursache: Ressource fehlt, Methode zum Schlafen ausgeführt, Warten auf Signal eines anderen Thread)
- Finished: Ende der Thread-Abarbeitung ist erreicht worden

Übersicht

- Grundlagen
- Threads in Java
- Probleme nebenläufiger Programme (Konkurrenz)
- Synchronisation in Java mit `synchronized`
- Probleme nebenläufiger Programme (Kooperation)
- Synchronisation in Java mit `wait` und `notify`

Realisierungsmöglichkeiten

- In Java zwei Möglichkeiten der Implementierung:
  - Ableitung von Klasse `Thread`
  - Implementierung Interface `Runnable`



- In beiden Fällen:
  - Methode `run()` für parallel auszuführenden Code
  - Aufruf der Befehlssequenz mit Methode `start()`

Klasse `Thread` (I)

- Befindet sich im Paket `java.lang`
- Bietet Basismethoden zur Erzeugung, Kontrolle und zum Beenden von Threads
- Wichtige Methoden:

Name	Bedeutung
`public Thread()`	Konstruktor zum Anlegen ohne Namen
`public Thread(String name)`	Konstruktor zum Anlegen mit Namen
`void run()`	Hauptmethode des Thread
`void start()`	Starten des Thread (Überführung Zustand Ready nach Running)
`static void sleep(long m)`	Thread legt sich für m Millisekunden schlafen (Überführung Zustand Running nach Blocked)
`void join()`	Warten auf das Zuendegehen eines Thread

Klasse `Thread` (II)

Name	Bedeutung
`void yield()`	Pausieren, um anderen Threads eine Chance zu geben
`boolean isAlive()`	Wahr, wenn Thread gestartet wurde, aber noch nicht beendet ist, d.h. in einem der Zustände Ready, Running oder Blocked ist
`void setPriority(int i)`	Setzt die Priorität eines Prozesses auf i
`int getPriority()`	Gibt die Priorität eines Prozesses zurück

- Für Prioritäten:
  - Drei Konstanten:
    - `MIN_PRIORITY = 1`
    - `MAX_PRIORITY = 10`
    - `NORM_PRIORITY = 5`
  - Folgende Konventionen:

Prioritätswert	Beschreibung
10	Krisenmanagement
7-9	Interaktiv, Ereignisgesteuert
4-6	E/A-gebunden
2-3	Hintergrundberechnung
1	Soll nur laufen, wenn nichts anderes kann

Klasse `Thread` (III)

- Erzeugung konkreter Thread:
  - Klasse von `Thread` ableiten
  - Methode `run()` überschreiben
- Aufruf Methode `start()` beginnt Ausführung des Thread, d.h. Start des Code der Methode `run()` als leichtgewichtiger Prozess

Beispiel

- Anlegen einer neuen Thread-Klasse:

```
(1) public class ZaehlenPerThread extends Thread{
(2)   public void run(){
(3)     int i = 0;
(4)     while (true){
(5)       System.out.println(i++);
(6)     }
(7)   }
(8) }
```

- Einsatz in einer Anwendung (Beispiel 1):

```
(1) public static void main(String[] args){
(2)   ZaehlenPerThread zaehler = new ZaehlenPerThread();
(3)   zaehler.start();
(4) }
```

- Führt zur Ausgabe: 1, 2, 3, 4, ...

Interface `Runnable` (I)

- Problem Klasse `Thread`: Soll Klasse auch von anderer Klasse abgeleitet werden ⇒ Mehrfachvererbung
- Dann: Verwendung Interface `Runnable`:
  - Definiert nur Methode `run()`
  - Thread-Erzeugung auf Basis von Runnable:
    - Anlegen neues `Runnable`-Objekt
    - Aufruf spezieller Konstruktor der Klasse `Thread` `public Thread (Runnable <Objektname>)`

Interface `Runnable` (II)

- Beispiel: Klasse `ZaehlenPerThread` mit folgendem Anfang:

```
(1) public class ZaehlenPerThread
(2)   extends <IrgendeineKlasse>
(3)   implements Runnable{…
```

- Anwendung (Beispiel 2):

```
(1) public static void main(String[] args){
(2)   ZaehlenPerThread zaehler = new ZaehlenPerThread();
(3)   Thread t = new Thread(zaehler);
(4)   t.start();
(5) }
```

Pause in Befehlsausführung

- Methode `public static void sleep(long m) throws InterruptedException`: Thread macht m Millisekunden Pause
- Pause kann durch InterruptedException von außen beendet werden
- Auch Methode `main()` intern Thread, somit möglich (Beispiel 3):

```
( 1) public static void main(String[] args){
( 2)   for (int i=0; i<=10; i++){
( 3)     System.out.println(i);
( 4)   }
( 5)   for (int i=0; i<=10; i++){
( 6)     System.out.println(i);
( 7)     try{
( 8)       Thread.sleep(1000);
( 9)     } catch (InterruptedException e) {}
(10) }}
```

- Erste Ausgabe schnell hintereinander, zweite Ausgabe Sekunde Verzögerung zwischen jeweils zwei Zahlen

Unterbrechen der Thread-Ausführung (I)

- Klasse `Thread` stellt Mechanismen zur Unterbrechung durch Senden einer Unterbrechungsanforderung zur Verfügung
- Thread ist selber dafür verantwortlich, auf Unterbrechungsanforderungen zu reagieren
- Methode `run()` muss in sinnvollen Zeitabständen prüfen, ob Unter-brechungsanforderung vorliegt
- Methoden:
  - **`public void`** `interrupt()`
    Signalisiert dem Thread von außen eine Unterbrechungsanforderung
  - **`public boolean`** `isInterrupted()`
    Mit dieser Methode kann Thread feststellen, ob Unterbrechungsanforderung vorliegt
- Beispiel:
  - Thread startet ab 1 zu zählen
  - Kommt von außen eine Unterbrechungsanforderung, hört er auf

Unterbrechen der Thread-Ausführung (II)

- Unterbrechbarer Thread muss in der `run()`-Methode Unterbrechung realisieren (Beispiel 4):

```
( 1) public class UnterbrechbarerZaehler extends Thread{
( 2)   public void run(){
( 3)     int i = 0;
( 4)     while (true){
( 5)       if (isInterrupted()){break;}
( 6)       System.out.println(i++);
( 7)     }
( 8)     System.out.println("Ich bin unterbrochen worden!");
( 9)   }
(10) }
```

Unterbrechen der Thread-Ausführung (III)

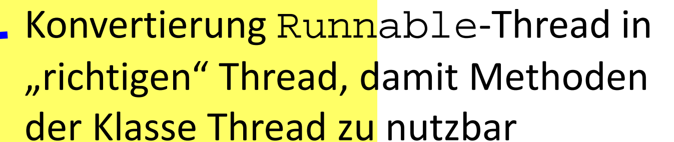
- `main()`-Methode startet den Thread und bricht ihn nach fünf Sekunden ab:

```
(1) public static void main(String[] args){
(2)   UnterbrechbarerZaehler z = new UnterbrechbarerZaehler();
(3)   z.start();
(4)   try{
(5)     Thread.sleep(5000);
(6)   } catch (InterruptedException e) {}
(7)    z.interrupt();
(8) }
```

Unterbrechen der Thread-Ausführung (IV)

- Unterbrechungen bei Runnable: Methoden der Klasse `Thread` (z.B. `isInterrupted`) bei Realisierung mit Interface `Runnable` nicht unmittelbar zugreifbar
- Klasse `Thread` besitzt hierfür statische Methode `public static Thread currentThread()`, die aktuellen Thread zurückliefert
- Modifiziertes Beispiel abbrechbarer Zähler (Beispiel 5):

```
( 1) public class UnterbrechbarerZaehler implements Runnable{
( 2)   public void run(){
( 3)     int i = 0;
( 4)     while (true){
( 5)       Thread aktuellerThread = Thread.currentThread();
( 6)       if (aktuellerThread.isInterrupted()){ break;}
( 7)       System.out.println(i++);
( 8)     }
( 9)     System.out.println("Ich bin unterbrochen worden!");
(10)   }
(11) }
```



Parallele Zähler (I)

- Bisherige Beispiele: Keine Nutzung Nebenläufigkeit
- Nun sollen zwei Zähler (quasi-)parallel voranschreiten
- Klasse ThreadZaehler (Beispiel 6):

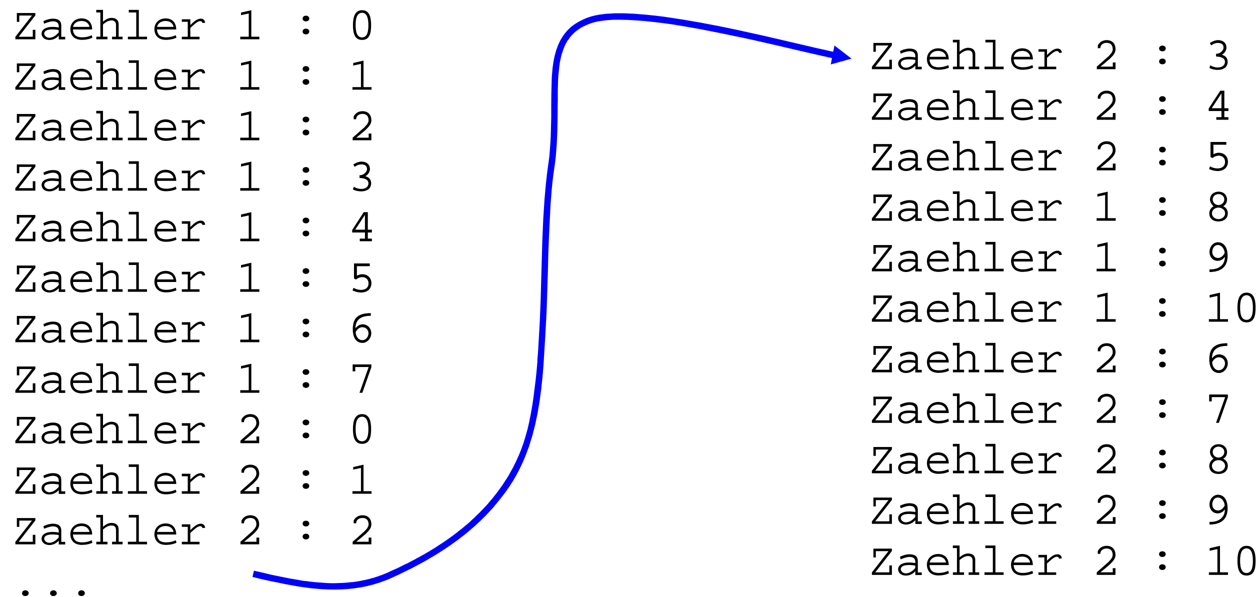
```
(1) public class ThreadZaehler extends Thread{
(2)   private String name;
(3)   public ThreadZaehler(String s){name = s;}
(4)   public void run(){
(5)     for (int i=0; i<=10; i++){
(6)       System.out.println(name + " : " + i);
(7)     }
(8)   }
(9) }
```

Parallele Zähler (II)

- Aufruf aus `main()`-Methode:

```
(1) ThreadZaehler z1 = new ThreadZaehler("Zaehler 1");
(2) ThreadZaehler z2 = new ThreadZaehler("Zaehler 2");
(3) z1.start();
(4) z2.start();
```

- Beispiel Ausgabe:



Übersicht

- Grundlagen
- Threads in Java
- Probleme nebenläufiger Programme (Konkurrenz)
- Synchronisation in Java mit `synchronized`
- Probleme nebenläufiger Programme (Kooperation)
- Synchronisation in Java mit `wait` und `notify`

Interleaving (I)

- „Ungünstige“ Abfolge der Elementaroperationen
  - Beispiel: Ein Konto K_1 habe einen Stand von 100€, Transaktion T_1 bucht 10€ ab, T_2 bucht 100€ nach K_1
  - Erwartetes Resultat bei sequentieller Ausführung (T_1 vor T_2 oder T_2 vor T_1): K_1 hat Wert von 190€
  - Abarbeiten der Anweisung (stand = stand + wert) geschieht in drei Teilschritten: Lesen des alten Werts, Ausführen der Rechenoperation, Schreiben des neuen Wertes
  - Im parallelen Fall kann folgendes Szenario auftreten:

Zeit	T_1	T_2	Wert
t_1			100
t_2		read(stand)	100
t_3	read(stand)	stand=stand+100	100
t_4	stand=stand-10	write(stand)	200
t_5	write(stand)		90

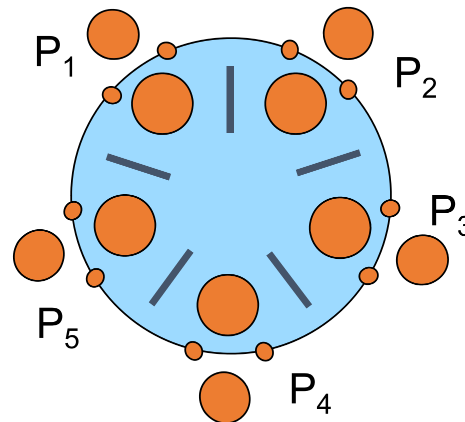
- Das Resultat ist 90 statt der erwarteten 190!

Interleaving (II)

- Abhilfe durch Synchronisation: Transaktionen müssen aufeinander warten, d.h. T_2 darf erst lesen, wenn T_1 geschrieben hat!
- Problem heißt Interleaving (synonym: Verzahnung oder Verschränkung)
- Vermeidung Probleme wie oben, aber neues Problem möglich

Deadlock (I)

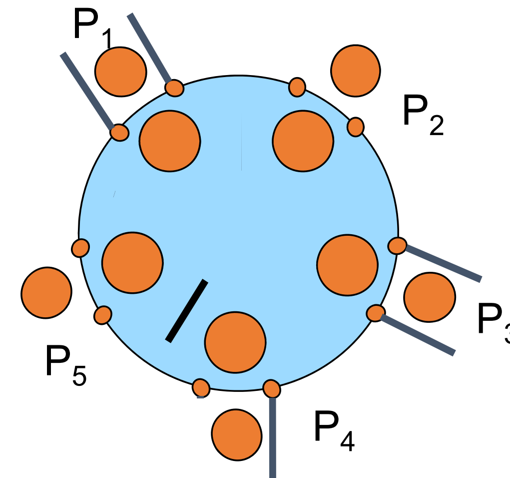
- Deadlock: zwei oder mehrere Transaktionen warten wechselweise aufeinander
- Beispiel: Dining Philosophers



  - Fünf Philosophen sitzen an einem Tisch und tuen nichts außer essen und denken
  - Zwischen zwei Philosophen liegt jeweils ein Stäbchen
  - Um essen zu können, braucht ein Philosoph beide neben ihm liegende Stäbchen
  - Es gilt: Erst das linke Stäbchen aufnehmen, dann das rechte

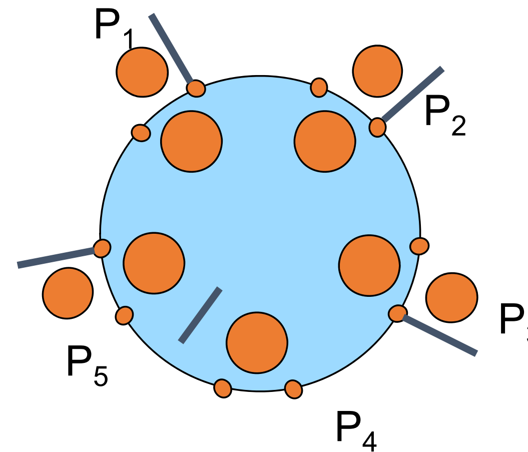
Deadlock (II)

- Beispiel:
  - P_1 nimmt linkes Stäbchen
  - P_1 nimmt rechtes Stäbchen
  - P_4 nimmt linkes Stäbchen
  - P_4 nimmt rechtes Stäbchen
  - P_4 legt rechtes Stäbchen ab
  - P_3 nimmt linkes Stäbchen
  - P_3 nimmt rechts Stäbchen
  - P_4 legt linkes Stäbchen ab
  - ...



Deadlock (III)

- Beispiel:
  - P_1 nimmt linkes Stäbchen
  - P_2 nimmt linkes Stäbchen
  - P_3 nimmt linkes Stäbchen
  - P_4 nimmt linkes Stäbchen
  - P_5 nimmt linkes Stäbchen
  - Deadlock



Übersicht

- Grundlagen
- Threads in Java
- Probleme nebenläufiger Programme (Konkurrenz)
- Synchronisation in Java mit `synchronized`
- Probleme nebenläufiger Programme (Kooperation)
- Synchronisation in Java mit `wait` und `notify`

Beispiel (I)

- Zwei Zähler sollen gemeinsame Variable hochzählen (Beispiel 7):

```
( 1) public class SynchroProblem extends Thread{
( 2)   static int grenze = 0;
( 3)   public void run(){
( 4)     while(grenze<20){
( 5)       int i = grenze;
( 6)       // Kritischer Abschnitt Beginn
( 7)       try{// Simuliert zeitaufwendige Berechnungen
( 8)         Random r = new Random();
( 9)         int j = r.nextInt(1000);
(10)         sleep(j);
(11)       }
(12)       catch(InterruptedException e){}
(13)       i++;
(14)       grenze = i;
(15)       // Kritischer Abschnitt Ende
(16)       System.out.println(i);
(17)     }
(18)   }
(19) }
```

Beispiel (II)

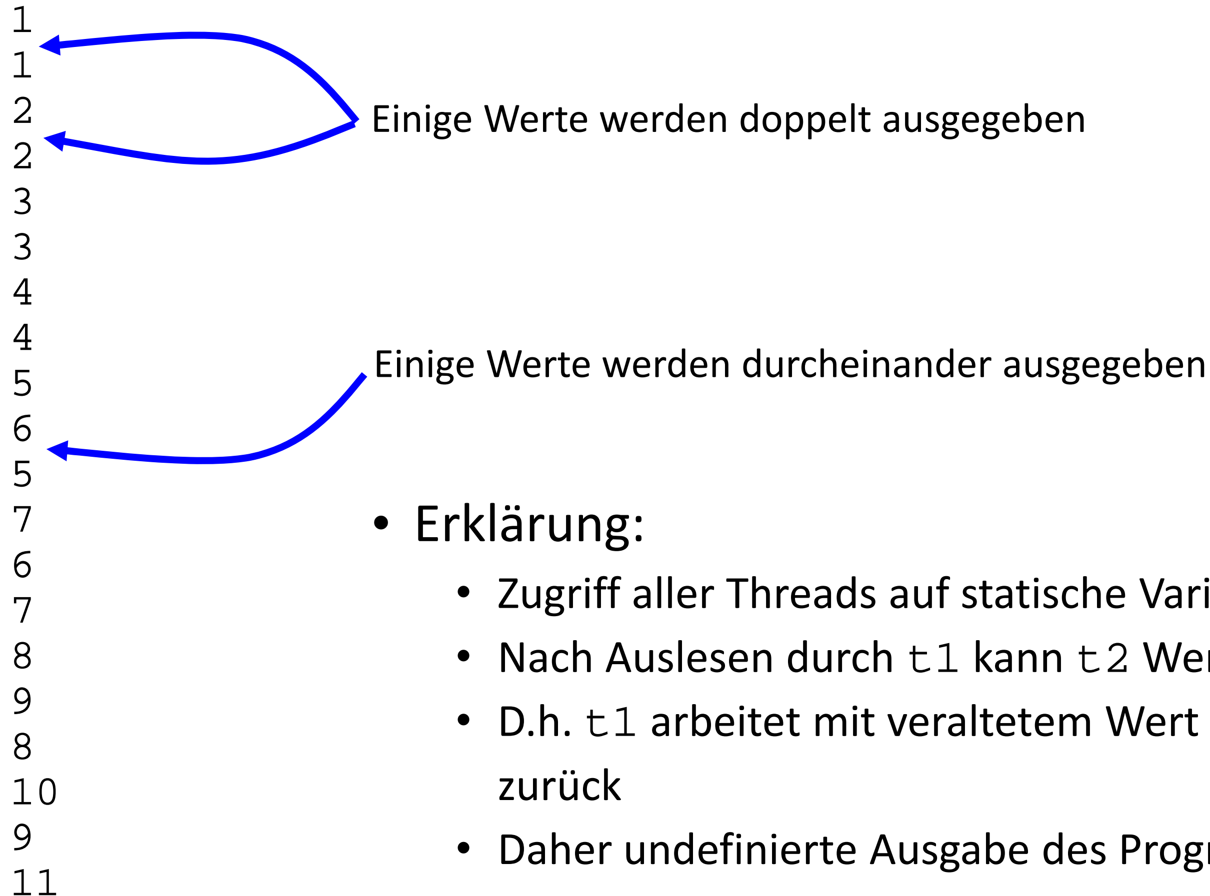
- Anlegen und Starten der Threads in Applikation:

```
(1) public static void main(String[] args){
(2)   Thread t1 = new SynchroProblem();
(3)   Thread t2 = new SynchroProblem();
(4)   t1.start();
(5)   t2.start();
(6) }
```

- Erwartet: Ausgabe der Zahlen von 1 bis 20

Beispiel (III)

- Stattdessen: Mögliche Ausgabe des Programms



- Erklärung:
  - Zugriff aller Threads auf statische Variable `grenze`
  - Nach Auslesen durch `t1` kann `t2` Wert schreiben
  - D.h. `t1` arbeitet mit veraltetem Wert und schreibt diesen zurück
  - Daher undefinierte Ausgabe des Programms

Schlüsselwort `synchronized` (I)

- Betreten kritischer Abschnitte (Programmabschnitte, in denen mehrere Threads schreibend auf die gleiche Ressource zugreifen) zu einem Zeitpunkt nur von einem Thread
- Synchronisation in Java durch sog. Monitore
- Monitor:
  - Kapselung kritischer Abschnitt mit Hilfe automatisch verwalteter Sperre
  - Beim Betreten des Monitors: Setzen Sperre
  - Beim Verlassen des Monitors: Sperre zurücknehmen
  - Bei Eintritt in Monitor Sperre durch anderen Thread gesetzt $\Rightarrow$ Aktueller Thread muss warten, bis Sperre freigegeben
- Realisierung: Schlüsselwort `synchronized`
- Wirkungsbereich `synchronized`: komplette Methode oder Block

Schlüsselwort `synchronized` (II)

- Erster Ansatz: Definition kritischer Methode als `synchronized` (Beispiel 7b):

```
(1) public synchronized void run()
(2) // Alles andere wie im letzten Beispiel
```

- Resultat (Beispiel):

```
1
1
2
2
3
3
...
```

- Problem:
  - Methode Instanzmethode
  - Korrektes Zählen innerhalb eines Thread funktioniert
  - Aber kein gemeinsamer Zähler für beide Threads

Schlüsselwort `synchronized` (III)

- Definition nur des kritischen Abschnitts als `synchronized` (Beispiel 7c):

```
( 1) while(grenze<20){
( 2)   // Kritischer Abschnitt Beginn
( 3)   synchronized(getClass()){
( 4)     int i = grenze;
( 5)     try{
( 6)       Random r = new Random();
( 7)       int j = r.nextInt(1000);
( 8)       sleep(j);
( 9)       // Simuliert zeitaufwendige Berechnungen
(10)     }
(11)     catch(InterruptedException e){}
(12)     i++;
(13)     if (grenze<20){
(14)       grenze=i;
(15)       System.out.println(i);
(16)     }
(17)   } // Kritischer Abschnitt Ende
(18) }
```

Nur kritischer Bereich als `synchronized` markiert

Schlüsselwort `synchronized` (IV)

- Führt zu erwartetem Resultat:

```
1
2
3
...
```

Synchronisation auf Methodenebene (I)

- Bei Synchronisierung ganzer Methode muss diese als statisch deklariert sein (mehrere Threads nutzen diese)
- Direkte Anwendung auf Methode funktioniert nicht:

```
(1) public static synchronized void run()
```

führt zur Fehlermeldung `run() kann nicht überschrieben werden`

- Daher: Auslagerung kritischer Abschnitt in separate Methode, diese als statisch deklarieren (Beispiel 7d):

```
(1) static synchronized void synchronisierteMethode(){
(2)   // Kritischer Abschnitt Beginn
(3)     ... "Wie im letzten Beispiel"
(4)   grenze=i; // Kritischer Abschnitt Ende
(5)   System.out.println(i);
(6) }
```

Synchronisation auf Methodenebene (II)

- Diese kann nun in `run()` aufgerufen werden:

```
(1) public void run(){
(2)   while(grenze<20){
(3)     synchronisierteMethode();
(4)   }
(5) }
```

- Führt ebenfalls zu erwartetem Resultat:

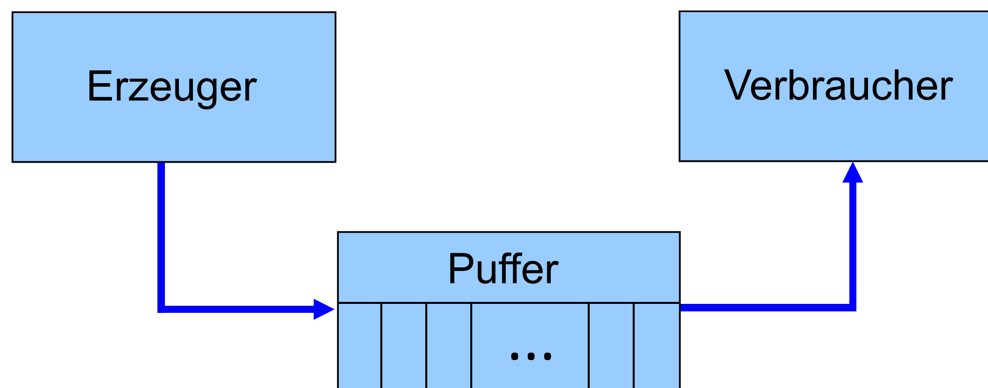
```
1
2
3
...
```

Übersicht

- Grundlagen
- Threads in Java
- Probleme nebenläufiger Programme (Konkurrenz)
- Synchronisation in Java mit `synchronized`
- Probleme nebenläufiger Programme (Kooperation)
- Synchronisation in Java mit `wait` und `notify`

Motivation

- Erzeuger-Verbraucher-Problem:
  - Thread 1 erzeugt etwas (hier: `Integer`-Zahlen) und schreibt es in Puffer (hier: `Vector`-Objekt)
  - Thread 2 verbraucht diese Werte durch Lesen aus Puffer und Ausgabe des Wertes
- Anwendungsbeispiel:
  - Erzeuger Benutzer*innen, produzieren Druckaufträge
  - Puffer Druckerwarteschlange
  - Verbraucher Drucker, Abarbeiten der Aufträge



Ohne Synchronisation (I)

- Erster Ansatz: Erzeuger-Verbraucher ohne Synchronisation (Beispiel 8)
- Klasse Puffer:

```
(1) import java.util.Vector;
(2) public class Puffer{
(3)   static Vector myPuffer = new Vector();
(4) }
```

- Klasse Erzeuger:

```
( 1) public class Erzeuger extends Thread{
( 2)   private int i = 0;
( 3)   public void run(){
( 4)     while (true){
( 5)         synchronized(Puffer.myPuffer){
( 6)           Puffer.myPuffer.set(0,new Integer(i++));
( 7)         }
( 8)     try{sleep(100);}
( 9)     catch(InterruptedException e){}
(10)   }
(11) }
```

Ohne Synchronisation (II)

- Klasse Verbraucher:

```
( 1) public class Verbraucher extends Thread{
( 2)   public void run(){
( 3)     while (true){
( 4)       synchronized(Puffer.myPuffer){
( 5)          System.out.println(Puffer.myPuffer.get(0));
( 6)        }
( 7)       try{sleep(50);}
( 8)       catch(InterruptedException e){}
( 9)     }
(10)   }
(11) }
```

Ohne Synchronisation (III)

- Klasse Anwendung:

```
( 1) ...
( 2) Puffer.myPuffer.add(new Integer(0));
( 3) Thread myErzeuger = new Erzeuger();
( 4) Thread myVerbraucher = new Verbraucher();
( 5) myErzeuger.start();
( 6) myVerbraucher.start();
```

Ohne Synchronisation (IV)

- Ausgabe:

```
0
0
1
1
2
2
3
3
4
4
...
```

- Erklärung/Problem:
  - Verbraucher-Thread schneller als Erzeuger-Thread
  - Daher doppelte Ausgabe der Zahlen
- Lösung:
  - Synchronisation Threads
  - „Verbraucher muss warten, bis wieder was zu verbrauchen“

Übersicht

- Grundlagen
- Threads in Java
- Probleme nebenläufiger Programme (Konkurrenz)
- Synchronisation in Java mit `synchronized`
- Probleme nebenläufiger Programme (Kooperation)
- Synchronisation in Java mit `wait` und `notify`

Schlüsselworte `wait` und `notify` (I)

- Synchronisationsprimitive `wait()` und `notify()`
- Definiert in Klasse `Object`
- Jedes Objekt Warteliste mit Threads, die vom Scheduler unterbrochen wurden und auf Ereignis warten, um fortgesetzt werden zu können
- Aufruf von `wait()` und `notify()` nur bei Sperrung des Objekts, d.h. innerhalb `synchronized`-Block
- `wait()` gibt Sperre (temporär) frei, stellt aufrufenden Thread in Warteliste des Objekts
- Dadurch Unterbrechung Prozess, im Scheduler als wartend markiert
- `notify()` entfernt einen (beliebigen) Thread aus der Warteliste des Objekts, stellt die (temporär) aufgehobenen Sperren wieder her und führt ihn dem normalen Scheduling zu

Schlüsselworte `wait` und `notify` (II)

- Erzeuger-Verbraucher-Lösung muss folgendermaßen erweitert werden:
  - Synchronisation über das Puffer-Objekt
  - Verbraucher muss Methode `wait()` ausführen und damit warten, bis der Erzeuger den Puffer mit einem `notify()` freigibt
- Erweiterung des Erzeugers (nach Schreiben in Puffer) (Beispiel 9):

```
(1) ...
(2) try{Puffer.myPuffer.notify();}
(3) catch(IllegalMonitorStateException e){}
(4) ...
```

- Erweiterung des Verbrauchers (vor dem Lesen aus Puffer):

```
(1)  ...
(2) try{Puffer.myPuffer.wait();}
(3) catch(InterruptedException e){}
(4) ...
```

Schlüsselworte `wait` und `notify` (III)

- Diese Erweiterungen führen zur gewünschten Ausgabe:

```
1
2
3
4
...
```

- Anmerkung:
  - Neben `notify()` gibt es eine Methode `notifyAll()`
  - Diese veranlasst nicht nur einen, sondern alle wartenden Threads zum Weiterlaufen

Zusammenfassung (I)

- Grundlagen
  - Nebenläufigkeit
  - Prozesse und Threads
  - Threadzustände

- Threads in Java
  - Klasse Thread
  - Interface Runnable
  - Unterbrechungen

- Probleme nebenläufiger Programme (Konkurrenz)
  - Interleaving
  - Deadlock

Zusammenfassung (II)

- Synchronisation in Java mit `synchronized`
  - Schlüsselwort `synchronized`
  - Kritischer Abschnitt
  - Synchroniastion auf Methodenebene
- Probleme nebenläufiger Programme (Kooperation)
  - Erzeuger-Verbraucher-Problem
- Synchronisation in Java mit `wait` und `notify`
  - Schlüsselworte `wait` und `notify`

Aufgaben (I)

- **Aufgabe 1**

  Mehrere Threads sollen über eine gemeinsame Variable kommunizieren.
  Diese soll einen Anfangswert haben, jeder Thread erhöht oder erniedrigt die Variable um einen bestimmten Wert (dieser Wert und die Art der Operation werden beim Anlegen des Threads als Parameter übergeben). Beendet werden soll das Programm, wenn die gemeinsame Variable auf 0 heruntergezählt ist. Der aktuelle Schritt soll auf der Konsole dokumentiert werden.

Aufgaben (II)

- **Aufgabe 2**

  In einem System arbeiten eine Reihe von Benutzer*innen, die von Zeit zu Zeit Druckaufträge eines gewissen Umfangs erzeugen. Diese werden in einer Druckerwarteschlange verwaltet. Zwei Drucker stehen zur Verfügung und arbeiten die Aufträge ab.

  Modellieren Sie das Problem als Erzeuger-Verbraucher-Problem und realisieren Sie es in Java.