

Universität Karlsruhe (TH)

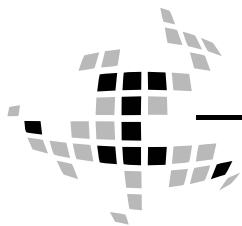
Forschungsuniversität · gegründet 1825

Parallelität und Koordination in Java

Prof. Dr. Walter F. Tichy

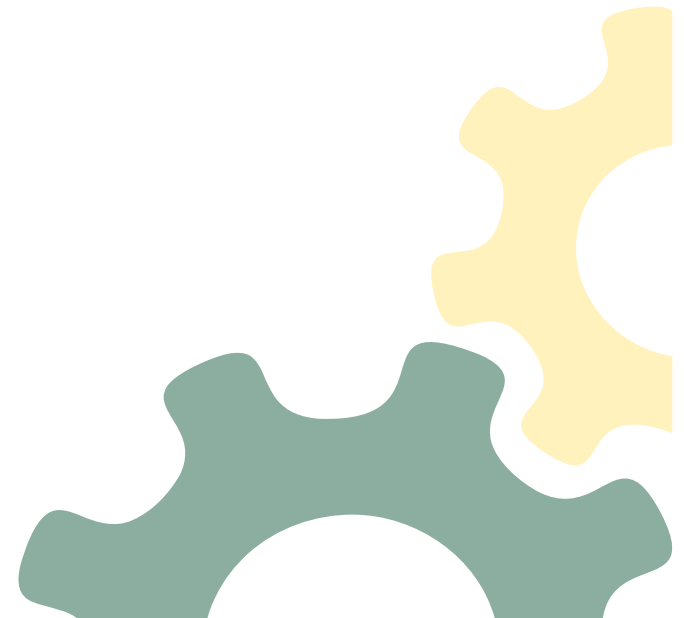
Dr. Victor Pankratius

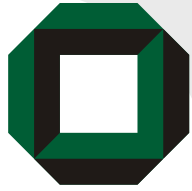
Ali Jannesari



Fakultät für **Informatik**

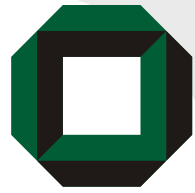
Lehrstuhl für Programmiersysteme





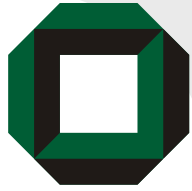
Agenda

1. Überblick
2. Konstrukte zum Erzeugen von Parallelität
3. Koordination
 - Wofür?
 - Kritische Abschnitte
 - Monitore
 - Monitore & Synchronisation
 - Monitore & Signalisierung
 - Konstrukte für Warten und Benachrichtigung
 - Sicherheitshinweise & Faustregeln
4. Unterbrechung von Fäden
5. Verklemmungen
6. Beispiele für parallele Programme in Java
7. Ausblick



Überblick

- Java bietet von Hause aus Unterstützung für parallele Programme.
- Programmiermodell: **Kontrollfäden** mit **gemeinsam genutztem** Speicher.
 - Javas virtuelle Maschine kann mehrfädige Programme ausführen
 - Kontrollfäden können bei Bedarf gestartet und wieder beendet werden.
 - Kontrollfäden werden bei aktuellen VMs auf native Kontrollfäden / Prozesse abgebildet.
 - Ausnutzung mehrerer vorhandener Prozessoren / Kerne
 - Javas Sprachdefinition enthält Primitive zur Synchronisation
 - Javas Bibliotheken enthalten Klassen und Methoden, um Kontrollfäden zu starten sowie für Synchronisation



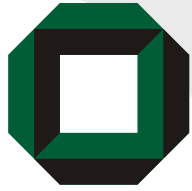
Konstrukte zum Erzeugen von Parallelität

- Eingebaute Klassen und Schnittstellen:
 - Interface **java.lang.Runnable**
 - *repräsentiert eine nebenläufig ausführbare Aufgabe*

```
public interface Runnable {  
    public abstract void run();  
}
```

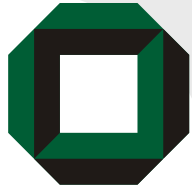
- Klasse **java.lang.Thread**
 - *repräsentiert einen nebenläufigen Kontrollfaden*

```
public class Thread implements Runnable {  
    public Thread(String name);  
    public Thread(Runnable target)  
    public void start();  
    public void run();  
    ...  
}
```

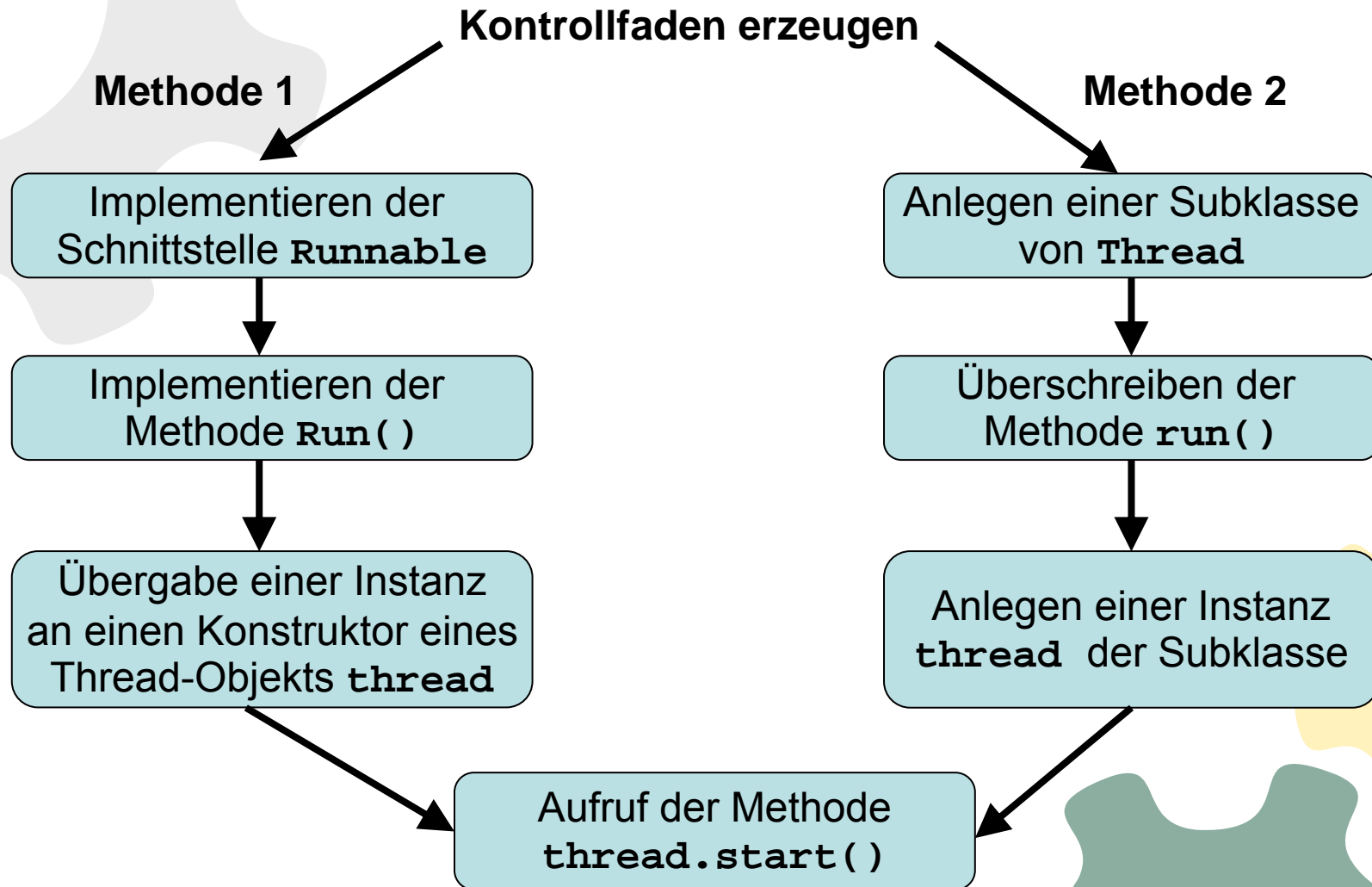


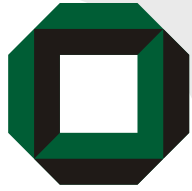
Konstrukte zum Erzeugen von Parallelität

- Erzeugen und Starten eines neuen, nebenläufigen Kontrollfadens ist nicht durch eine Java-Bibliothek realisierbar, sondern braucht die **Unterstützung der VM**.
- Der neue **Programm-faden** hat
 - seinen eigenen **Stapel** und **Programmzähler**.
 - Zugriff auf den **gemeinsamen Hauptspeicher**.
 - möglicherweise **lokale Kopien** von Daten des Hauptspeichers.



Konstrukte zum Erzeugen von Parallelität





Konstrukte zum Erzeugen von Parallelität

Beispiel: Methode 1

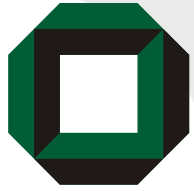
- Klasse, die `Runnable` implementiert:

```
class ComputeRun implements Runnable {
    long min, max;
    ComputeRun(long min, long max) {
        this.min = min; this.max = max;
    }
    public void run() {
        // Parallele Aufgabe
    }
}
```

- Erzeuge und starte Kontrollfaden:

```
ComputeRun c = new ComputeRun(1, 20);
new Thread(c).start();
```

- **Starten** des neuen Kontrollfadens. Erst das erzeugt die neue **Aktivität!**
- Die Methode `start()` **kehrt sofort zurück**, der neue Kontrollfaden **arbeitet nebenläufig weiter**.
- **Kein Neustart**: `start()` darf **nur einmal** aufgerufen werden.
- `run()` nicht direkt aufrufen



Konstrukte zum Erzeugen von Parallelität

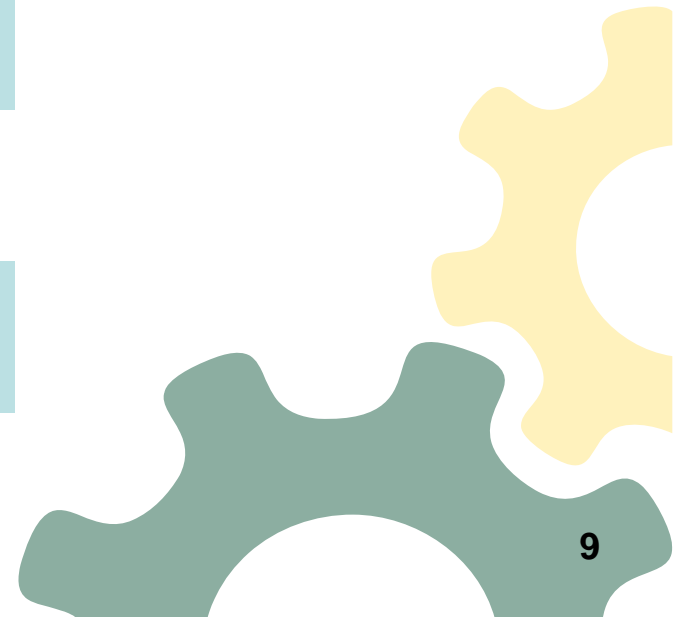
Beispiel: Methode 2

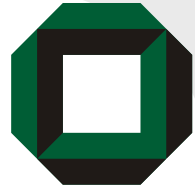
- Klasse, die von Thread erbt:

```
class ComputeThread extends Thread {  
    long min, max;  
    ComputeThread(long min, long max) {  
        this.min = min; this.max = max;  
    }  
    public void run() {  
        // Parallele Aufgabe  
    }  
}
```

- Erzeuge und starte Kontrollfaden:

```
ComputeThread t = new ComputeThread(1,10);  
t.start();
```

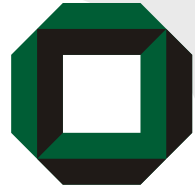




Konstrukte zum Erzeugen von Parallelität

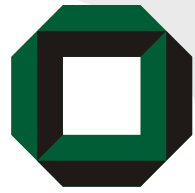
Methode 1 vs. Methode 2

- Warum eine eigene Klasse, die Runnable implementiert?
 - Das Überschreiben von `run()` in einer Unterklasse von `Thread` hätte denselben Effekt.
- Aber: Bessere Modularisierung bei der Verwendung der Schnittstelle `Runnable`:
 - Die **Kapselung** der Aufgabe in einem eigenen Objekt (bzw. einer eigenen Klasse) macht diese mit weniger Overhead in einem sequenziellen Kontext verfügbar.
 - Bei **Verteilung**: Die Aufgabe kann übers **Netzwerk** versendet werden (`Thread` ist nicht serialisierbar).



Koordination

- Grundlegende Koordinationsmechanismen sind in die Sprache eingebaut.
 - Wechselseitiger **Ausschluss**:
 - Markierung kritischer Abschnitte, die nur von einer Aktivität gleichzeitig betreten werden dürfen.
 - **Warten** auf "Ereignisse" und **Benachrichtigung**
 - Aktivitäten können auf Zustandsänderungen warten, die durch andere Aktivitäten verursacht werden.
 - Aktivitäten informieren andere, wartende Aktivitäten über Signale.
 - **Unterbrechungen**:
 - Eine Aktivität, die auf ein nicht (mehr) eintretendes Ereignis wartet, kann über eine Ausnahmebedingung abgebrochen werden.

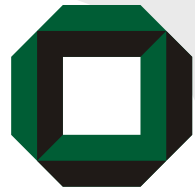


Koordination: Wofür?

- Zwei Aktivitäten führen den folgenden Code parallel aus.
Beide haben Zugriff auf dieselbe "globale" Variable `globalVar`.

```
if (globalVar > 0) {  
    globalVar --;  
}
```

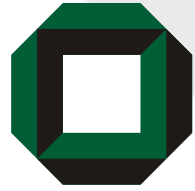
- Kann `globalVar` negativ werden?



Koordination: Wofür?

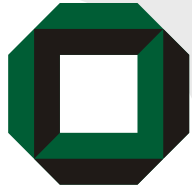
- Antwort: Ja, eine Wettlaufsituation (race condition) kann eintreten. Speicherzugriffe der Aktivitäten werden in **irgendeiner** Reihenfolge ausgeführt.
- Die folgende denkbare Ausführungsreihenfolge ist **kritisch**:

Thread 1	Thread 2
	<code>// globalVar == 1</code>
<code>if (globalVar > 0) {</code>	
	<code>if (globalVar > 0) {</code>
	<code> globalVar--;</code>
	<code>}</code>
<code> globalVar--;</code>	
<code>}</code>	



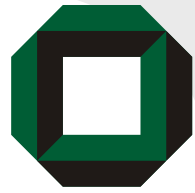
Koordination: Kritische Abschnitte

- Ein Bereich, in dem ein **Zugriff** auf einen gemeinsam genutzten Zustand stattfindet, ist ein **kritischer Abschnitt** (critical Section).
- Um **Wettlaufsituationen** (race conditions) zu vermeiden, müssen solche kritischen Abschnitte **geschützt** werden.
 - Nur eine Aktivität darf einen kritischen Abschnitt gleichzeitig bearbeiten.
 - Vor dem Betreten eines kritischen Abschnitts muss sichergestellt sein, dass ihn keine andere Aktivität ausführt.



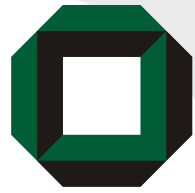
Koordination: Kritische Abschnitte

- **Atomarität:** Java garantiert, dass Zugriffe auf einzelne Felder atomar erfolgen.
 - **Ausgenommen** sind Felder vom Typ **double** und **long** (wg. 64 statt 32 Bit)
- Zugriffe auf mehrere Felder, oder aufeinanderfolgende Lese- und Schreiboperationen (wie im Beispiel) werden nicht atomar ausgeführt.
 - Vorsicht auch bei „**i++**“. Sieht atomar aus, ist es aber nicht!
- Auch wenn der Zugriff atomar erfolgt, ist im Allgemeinen **dennoch eine Synchronisation** erforderlich, um die möglicherweise von einer Aktivität **lokal zwischengespeicherten** Speicherbereiche sicher mit dem Hauptspeicher abzugleichen (siehe spätere Folien).



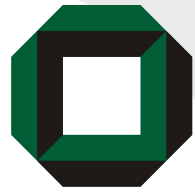
Koordination: Monitor

- In Java können **Monitore** zum Schutz kritischer Abschnitte verwendet werden (zur Idee vgl. auch Hoare 1974).
- Monitor hat Daten und Operationen
 - Ein Monitor bietet u.a. diese zwei Operationen an:
 - **enter()** „betritt“ den Monitor.
 - **exit()** „verlässt“ den Monitor.



Koordination: Monitor

- Prinzip:
 - Mit dem Aufruf von `enter()` **besetzt** eine Aktivität einen **freien** Monitor.
 - Versucht eine Aktivität, einen schon besetzten Monitor zu betreten, wird sie solange **blockiert**, bis der Monitor wieder freigegeben wird.
 - Der Monitor bleibt besetzt, bis die Aktivität schließlich `exit()` aufruft.
 - **Dieselbe** Aktivität kann einen Monitor mehrfach betreten.



Koordination: Monitor

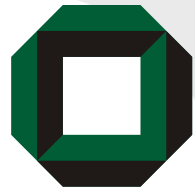
- Mit Monitoren können Wettlaufsituationen in kritischen Abschnitten vermieden werden:
 1. betrete den Monitor: `enter()`
 2. führe den kritischen Abschnitt aus
 3. verlasse den Monitor: `exit()`

```
"monitor.enter()"
```

```
    if (globalVar > 0) {  
        globalVar --;  
    }
```

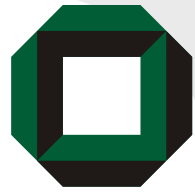
```
"monitor.exit()"
```

nicht ganz Java
Syntax...



Koordination: Monitore & Synchronisation (1)

- In Java kann **jedes Objekt** als Monitor verwendet werden.
 - nicht jedoch primitive Typen wie `int`, `float`, ...
- Die virtuelle Maschine kennt zwei Befehle
 - `0xC2 monitorenter <object-ref>`
 - `0xC3 monitorexit <object-ref>`
- Diese beiden Befehle dürfen nur paarweise in einem Block auftreten.
 - Dies beugt dem "Vergessen" der Monitorfreigabe vor.
 - Vor dem Ausführen des Bytecodes wird durch den Bytecode-Verifier (u.a.) die Schachtelung der Monitor-Anforderungen und -Freigaben überprüft.



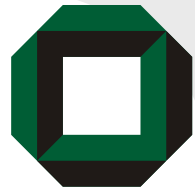
Koordination: Monitore & Synchronisation (2)

- In der Sprachdefinition wird die paarweise Verwendung von Monitor-Anforderung und -Freigabe durch eine Blocksyntax erzwungen:

```
/*synchronized block*/  
synchronized (obj) {  
    // critical  
    // section  
}
```

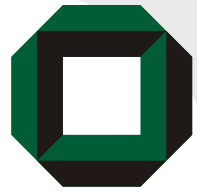
```
/*synchronized method*/  
synchronized void foo(){  
    // whole method is a  
    // critical section  
}
```

- eine synchronisierte Methode äquivalent zu einer Methode, deren Rumpf in einen an **this** (bzw. dem Klassenobjekt im Fall einer statischen Methode) synchronisierten Block eingeschlossen ist.



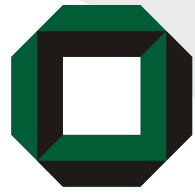
Koordination: Monitore & Synchronisation (3)

- Bei der Ausführung eines **synchronisierten** Blocks...
 - wird der Monitor des zugehörigen Objekts betreten: `enter()`
 - wird der Block ausgeführt
 - wird der Monitor verlassen: `exit()`
- Synchronisation ist **immer an ein Objekt** gebunden
 - bei einem `synchronized(x)`-Statement:
das angegebene **Objekt x**
 - bei einer synchronisierten Instanz-Methode `a.foo()`:
das aktuelle **Objekt this**
 - bei einer synchronisierten statischen Methode `A.sfoo()`:
das Klassen-**Objekt: A.class**



Koordination: Monitore & Synchronisation (4)

- Was passiert, wenn eine Aktivität versucht, einen **besetzten** Monitor zu **betreten**?
 - Die Aktivität wird **ununterbrechbar** blockiert.
 - Es gibt **kein Entkommen**, außer durch **Freigabe** des Monitors.
 - Es gibt keinen Test derart: "**wouldBlock(obj)**".
 - Warum nicht?
 - Aber: es gibt **Thread.holdsLock(Object)**
 - Statische Methode, mit der eine Aktivität selbst herausfinden kann, ob sie einen Monitor hält.
- **Dieselbe** Aktivität kann einen Monitor **beliebig oft** betreten (sinnvoll z.B. bei Rekursion).

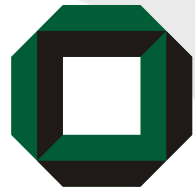


Koordination: Monitore & Synchronisation (5)

- Wie lässt sich die Wettlaufsituation aus dem Beispiel reparieren?

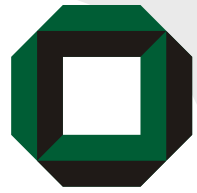
```
synchronized (someObject) {  
    if (globalVar > 0) {  
        globalVar --;  
    }  
}
```

- Die möglichen Ausführungsreihenfolgen der zwei Aktivitäten werden eingeschränkt.
 - Welche Möglichkeiten bleiben übrig?



Koordination: Monitore & Signalisierung (1)

- Wechselseitiger Ausschluss ist nicht genug:
 - Die **Abarbeitung einer Aufgabe** kann vom Fortschritt einer anderen Aktivität **abhängen**.
 - Beispiel: Muster „Hersteller – Verbraucher“
 - Die „Hersteller“ stellen **Aufträge** in eine Schlange.
 - Mehrere „**Verbraucher**“ nehmen die Aufträge entgegen und **führen sie aus**.
 - Ein **Verbraucher** kann nur weitermachen, wenn die Schlange **nicht leer** ist.
 - Der **Hersteller** muss anhalten, wenn die Schlange **voll** ist.



Koordination: Monitore & Signalisierung (2)

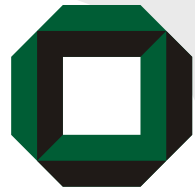
- Hersteller/Verbraucher – was ist hier falsch?
- Innerhalb derselben Klasse:

// Hersteller:

```
synchronized void post(Work w) {  
    while (queue.isFull()) { /*NOP*/ }  
    queue.add(w);  
}
```

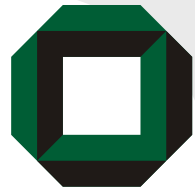
// Verbraucher:

```
synchronized Work get() {  
    while (queue.isEmpty()) { /*NOP*/ }  
    return queue.remove();  
}
```



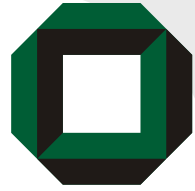
Koordination: Monitore & Signalisierung (3)

- **Gute Idee:** Der Zugriff auf die Schlange ist synchronisiert.
- **Schlechte Idee:** Der Hersteller und Verbraucher warten „aktiv“, falls sie nicht weiterarbeiten können.
 - Verschwendung von Rechenzeit!
- **Schlechte Idee:** Der Verbraucher hält den Monitor besetzt, solange er wartet.
 - Der Hersteller kann **niemals** neue Arbeit in die Schlange einstellen, weil er beim Versuch, den **Monitor** zu betreten, **blockiert**.
 - Für den Verbraucher gilt umgekehrt dasselbe.



Koordination: Monitore & Signalisierung (4)

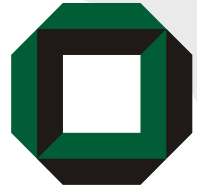
- Reparatur des Beispiels
 - Falls eine **Wächterbedingung fehlschlägt**, muss eine Aktivität ihre **Rechenzeit abgeben**.
 - Die **Kontrolle** sollte möglichst **direkt** an eine Aktivität **weitergegeben** werden, die weiterrechnen kann.
 - Während eine **Aktivität** an einer Wächterbedingung **wartet**, muss sie den **Monitor freigeben**.



Koordination: Konstrukte für Warten und Benachrichtigung (1)

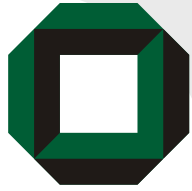
- Zur Erinnerung: Jedes Objekt kann Monitor sein...
- Methoden dazu in `java.lang.Object`

```
• public final void wait()  
  throws InterruptedException;  
  
• public final void wait(long timeout)  
  throws InterruptedException;  
  
• public final void wait(long timeout, int nanos)  
  throws InterruptedException;  
  
• public final void notify();  
  
• public final void notifyAll();
```



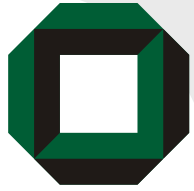
Koordination: Konstrukte für Warten und Benachrichtigung (2)

- Um die gezeigten Methoden aufrufen zu können, muss die aktuelle Aktivität den zugehörigen Monitor bereits betreten haben.
 - Dies lässt sich nicht (immer) durch den Übersetzer überprüfen. Bei Verstoß gegen diese Regel wird zur Laufzeit eine **IllegalMonitorStateException** geworfen.
- In synchronisierten Methoden ist der Monitor **this**, und statt **this.{wait|notify|notifyAll}()** kann einfach nur **wait() / ...** aufgerufen werden.



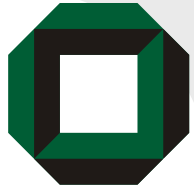
Koordination: Konstrukte für Warten und Benachrichtigung (3)

- `wait()` wartet solange, bis ein Signal bei dem Objekt eintrifft.
 - Folgendes passiert:
 - Die aktuelle Aktivität wird „schlafen gelegt“.
 - Die Aktivität wird in eine (VM-interne) Warteschlange für den Monitor des Objekts, an dem `wait()` aufgerufen wird, eingereiht.
 - Der betreffende Monitor wird während des Wartens freigegeben.
 - Alle anderen Monitore bleiben besetzt!
 - Andere Aktivitäten können den Monitor (einzeln) betreten.
 - Variante: `wait(long timeout, [int nanos])` beschränkt die Wartezeit auf den angegebenen Wert.
 - Die Wartezeit kann dennoch länger ausfallen, weil u. U. auf die Freigabe des Monitors gewartet werden muss.



Koordination: Konstrukte für Warten und Benachrichtigung (4)

- `notify()` und `notifyAll()` schicken Signale an die an diesem Monitor wartenden Aktivitäten.
- Folgendes passiert:
 - Wenn sich in der zugehörigen Warteschlange des betreffenden Monitors eine Aktivität befindet, wird ihr ein Signal geschickt.
 - `notify()` schickt Signal an irgendeine Aktivität aus dieser Warteschlange
 - `notifyAll()` schickt Signal an alle Aktivitäten dieser Warteschlange
 - Der oder die betreffenden Aktivitäten müssen den Monitor wieder betreten...
 - Das bedeutet, dass sie mindestens solange warten müssen, bis jene Aktivität, die `notify[All]()` aufgerufen hat, den Monitor freigibt.
 - ... und können (nacheinander) den Code des synchronisierten Blocks bzw. der synchronisierten Methode weiter ausführen.



Koordination: Konstrukte für Warten und Benachrichtigung (5)

- **Reparatur des Beispiels:**

Warten an Wächterbedingungen, Signale bei Bedingungsänderungen.

1..n
Hersteller

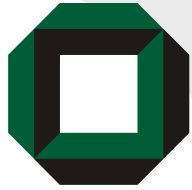
```
synchronized void post(Work w) {  
    while (queue.isFull()) {this.wait(); }  
    queue.add(w);  
    this.notifyAll();  
}
```

Geht auch an andere wartende
Hersteller

Geht an wartende Verbraucher

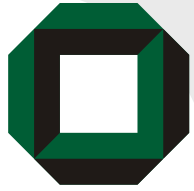
1..n
Verbraucher

```
synchronized Work get() {  
    while (queue.isEmpty()) {this.wait(); }  
    this.notifyAll();  
    return queue.remove();  
}
```



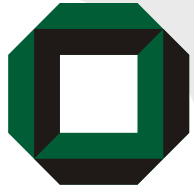
Koordination: Sicherheitshinweise & Faustregeln (1)

- Ein mit `notify()`/`notifyAll()` geschicktes Signal erreicht nur eine Aktivität, die beim Absenden schon wartet!
 - Das Signal wird nicht beim Monitor gespeichert
 - Das Signal hat keinen Effekt, wenn niemand wartet
- Verbinde Signale mit Zustand!
 - `wait()` ohne Wächterbedingung ist falsch (zu 99.9%).
 - FALSCH: ... `synchronized(obj) {wait(); } ...`
 - `notify()` ohne Zustandsänderung ist falsch (zu 99.9%).
 - FALSCH: ... `synchronized(obj) {notify(); } ...`



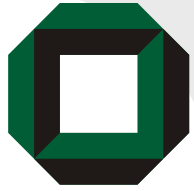
Koordination: Sicherheitshinweise & Faustregeln (2)

- Prüfe immer vor und nach dem Warten auf die Bedingung!
 - Verwende `wait()` immer in einer Schleife!
 - FALSCH: ... `if (! pCondition) {wait(); } ...`
- Gründe
 - Vorher: Kontrollfaden nicht unnötigerweise „schlafen legen“
 - Nachher: Nicht sichtbar, `warum` ein Signal geschickt wurde. „Aufwachen“ könnte durch ein „falsches“ Signal getriggert worden sein. Beispielsweise schickt Hersteller vereinfachend mit `notifyAll()` Signale an Hersteller UND Verbraucher. Hersteller sollen aber aus `wait()` immer nur dann entkommen, wenn Schlage nicht voll ist.



Koordination: Sicherheitshinweise & Faustregeln (3)

- In einem robusten Programm können alle `notify()`-Aufrufe durch `notifyAll()` ersetzt werden.
- Wenn der Monitor von außen zugänglich (**public**) ist, könnte jemand anderes ...
 - ... **Signale stehlen** (dadurch, dass er weitere Aktivitäten warten lässt).
 - ... unerwartet **mehr Signale** schicken.
- Kugelsicher: Immer `notifyAll()` verwenden.
 - Einige VMs machen keinen Unterschied zwischen `notify()` und `notifyAll()`!
 - Das wird von der JVM-Spezifikation ausdrücklich erlaubt.



Koordination: Sicherheitshinweise & Faustregeln (4)

- **Beispiel:** Was passiert, wenn man statt `notifyAll()` nur `notify()` verwendet?

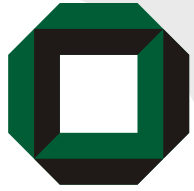
```
synchronized void post(Work w) {  
    while (queue.isFull()) {this.wait(); }  
    queue.add(w);  
    this.notify();  
}
```

1..n
Hersteller

```
synchronized Work get() {  
    while (queue.isEmpty()) {this.wait(); }  
    this.notify();  
    return queue.remove();  
}
```

1..n
Verbraucher

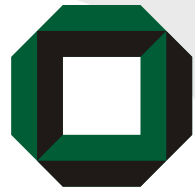




Koordination: Sicherheitshinweise & Faustregeln (5)

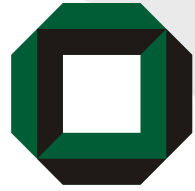
Antwort zu „nur `notify()`“

- Das Programm kann **blockieren**, oder **suboptimal** arbeiten.
- Beispiel: Die Schlange mit den Arbeitspäckchen ist leer.
 - Ein von einem Verbraucher geschicktes Signal kann einen anderen wartenden **Verbraucher** anstatt einen **Hersteller** treffen.
 - Dieser Verbraucher „**verschluckt**“ diese Signal, der Hersteller „schläft“ weiter.
 - Gleiches gilt entsprechend für Signale von Herstellern.



Unterbrechung (1)

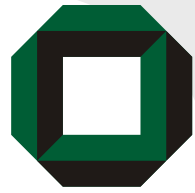
- **Problem:** Wie beendet man eine Aktivität, die auf Signale wartet, die nicht mehr eintreffen werden?
- *Illustration am letzten Beispiel:*
 - *Die Verbraucher fordern „bis in alle Ewigkeit“ Arbeit an.*
 - *Nach Beendigung der Hersteller trifft aber keine neue Arbeit mehr ein.*
 - *Wie löst man dieses Problem? Schlechte Lösung: Sonderfall einführen, zum Beispiel eine `null`-Referenz in die Liste der Arbeitspäckchen einstellen.*
 - *Wie viele solche Päckchen braucht man?*
 - *Was ist, wenn `null` schon „besetzt“, oder einfach ein gültiger Wert ist?*
- **Saubere Lösung:** Man schickt den betreffenden Aktivitäten eine Unterbrechung (`InterruptedException`).



Unterbrechung (2)

```
synchronized void post(Work w) {  
    while (queue.isFull()) {  
        try {  
            this.wait();  
        } catch (InterruptedException ex) {  
            // ??  
        }  
    }  
}
```

- Fehlt eine Ausnahmebehandlung, wird der Übersetzer dies bemängeln.
- Ausnahmebehandlung richtet sich an den Kontrollfaden, nicht an den Monitor

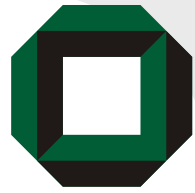


Unterbrechung (3)

- Eine Unterbrechung wird direkt an eine **Aktivität** geschickt:

```
Thread t;  
t.interrupt();
```

- Wenn t derzeit nicht wartet, wird die Unterbrechung beim **nächsten Aufruf** einer **wait()**-Methode durch die Aktivität t signalisiert.
 - D.h. die Unterbrechungsanforderung geht nicht verloren
- Wurde eine Aktivität unterbrochen, wirft **wait()** eine **InterruptedException**.
 - Die Unterbrechung ist nicht preemptiv
 - Unterbrechung ist typsicher!
 - Die InterruptedException muss deklariert (**throw**) oder abgefangen (**catch**) werden.
 - Es gelten dieselben Regeln wie bei **notify()**, nur dass direkt in den **catch()**-Block verzweigt wird.

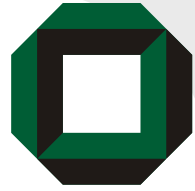


Unterbrechung (4) - Faustregeln

- Wenn für eine `InterruptedException` keine vernünftige Behandlung angegeben werden kann, sollte man sie deklarieren (d.h. mit `throw` weiter werfen!)
 - Das gilt wie für jede andere Ausnahmebedingung auch...
 - Der umgebende Code bzw. Aufrufer sollte besser damit umgehen können.
- Verwende Unterbrechungen, um Aktivitäten sauber zu beenden (anstatt Sonderfälle zu konstruieren).
 - Das geht nur, wenn diese Unterbrechungen nicht in Trivial-Behandlungsroutinen ohne Effekt „weggefangen“ werden.

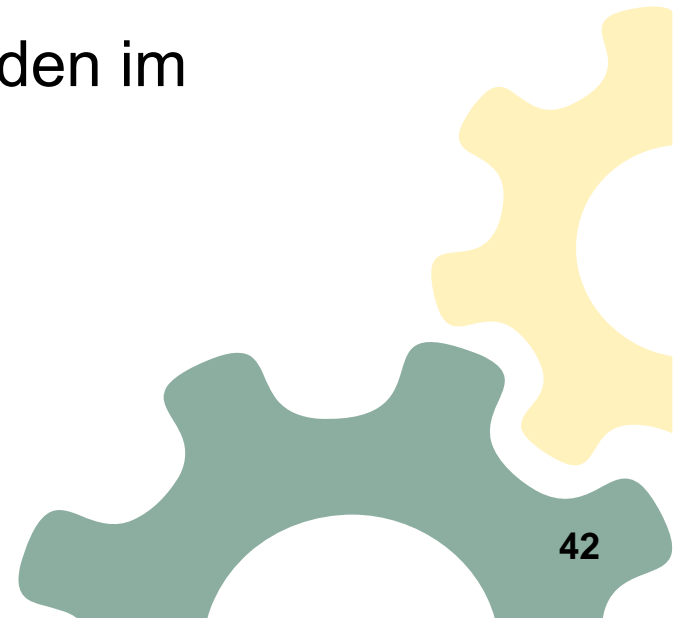
also **nicht**:

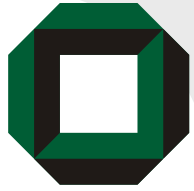
```
try {  
    wait();  
} catch (InterruptedException ex) {}
```



Verklemmungen (1)

- Trotz **synchronized** können noch weitere Probleme auftreten
- **Verklemmung (Deadlock)**
 - Blockade, die durch eine zyklische Abhängigkeit hervorgerufen wird
 - Führt dazu, dass alle beteiligten Fäden im Wartezustand verharren



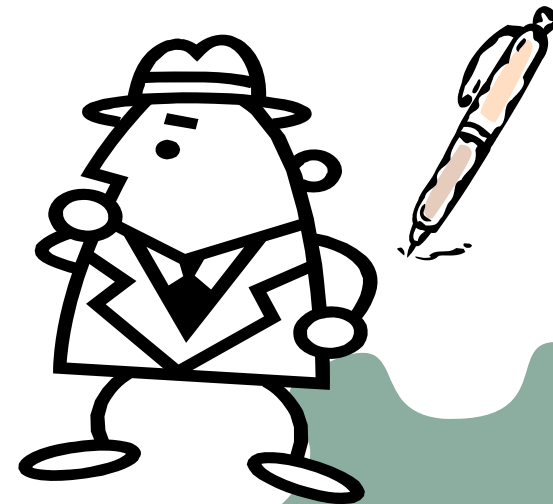
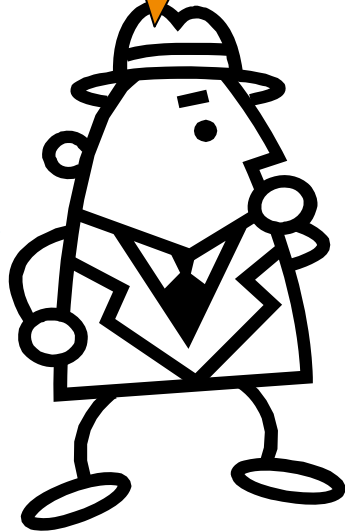
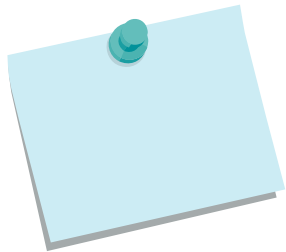


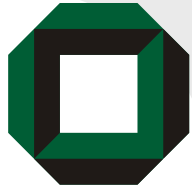
Verklemmungen (2)

Ich will
schreiben

Geben Sie mir
den Stift!

Geben Sie mir
das Papier!





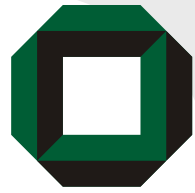
Verklemmungen (3)

```
//Beispiel für Verklemmung
final Object resource1 = new Object();
final Object resource2 = new Object();

Thread t1 = new Thread(new Runnable() {
    public void run(){
        synchronized(resource1) {
            synchronized(resource2) {compute();}
        }
    }
});

Thread t2 = new Thread(new Runnable() {
    public void run(){
        synchronized(resource2) {
            synchronized(resource1) {compute();}
        }
    }
});
t1.start(); // sperrt resource1
t2.start(); // sperrt resource2; jetzt beide blockiert!
```

Reparaturvorschläge?



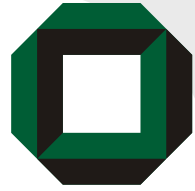
Beispiel: Vektoraddition (1)

- Wir wollen eine parallele Vektoraddition durchführen.
- Wichtigster Bestandteil: Die Klasse `Worker`, die zwei (Teil-) Vektoren elementweise addiert und das Ergebnis in einem dritten Vektor speichert.

```
class Worker implements Runnable {
    private int[] a, b, c;
    private int left, right;

    public Worker(int[] a, int[] b, int[] c, int left, int right) {
        this.a = a; this.b = b; this.c = c;
        this.left = left; this.right = right;
    }

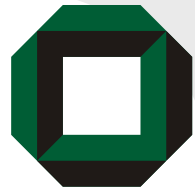
    public void run() {
        for (int i = left; i < right; ++i) {
            c[i] = a[i] + b[i];
        }
    }
}
```



Beispiel: Vektoraddition (2)

- Die `main`-Methode definiert zunächst
 - die zu addierenden Vektoren,
 - ein paar Konstanten,
 - Warum wird bei der Berechnung von `perThread` aufgerundet?
 - den Ergebnisvektor.

```
public static void main(String[] args) {  
  
    int a[] = ... // fill  
    int b[] = ... // fill  
    assert a.length == b.length;  
  
    final int nThreads = 10; // or whatever  
  
    final int nVecSize = a.length;  
    final int perThread = (int) Math.ceil((double) nVecSize / nThreads);  
  
    int[] c = new int[nVecSize];  
}
```



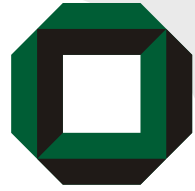
Beispiel: Vektoraddition (3)

- Es wird ein Feld angelegt, das Referenzen auf die zu startenden Aktivitäten hält.
- In der Schleife werden die Aktivitäten jeweils mit dem Indexbereich, den sie bearbeiten sollen, konstruiert und anschließend gestartet.
 - Erklären Sie, wofür bei der Berechnung von `right` die Minimumfunktion eingesetzt wird.
 - Kann `left >= right` werden? Stellt das ein Problem dar? Was machen die Aktivitäten, für die das der Fall ist?

```
Thread[] worker = new Thread[nThreads];

for (int i = 0; i < nThreads; ++i) {
    int left = i * perThread;
    int right = Math.min((i + 1) * perThread, nVecSize);

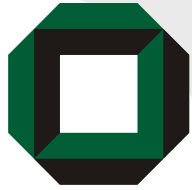
    worker[i] = new Thread(new Worker(a, b, c, left, right));
    worker[i].start(); // Thread i runs from now on
}
```



Beispiel: Vektoraddition (4)

- Nun müssen die Teilergebnisse wieder eingesammelt werden.
 - Anmerkung: `join()` ist auch durch `InterruptedException` durch einen weiteren Kontrollfaden unterbrechbar

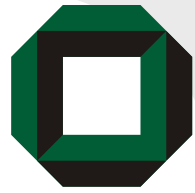
```
for (Thread thread : worker) {  
    try {  
        thread.join();  
    } catch (InterruptedException ex) {  
        System.err.println("Unexpected Interrupt " +  
            "while waiting for Thread ");  
    }  
}  
  
// now use the result in c[]
```



Beispiel: Semaphore

```
public class Semaphore {  
    private int count;  
  
    public synchronized void acquire()  
        throws InterruptedException {  
        while (count <= 0) wait();  
        --count;  
    }  
  
    public synchronized void release() {  
        ++count;  
        notifyAll();  
    }  
  
    public Semaphore(int capacity){ count = capacity; }  
}
```

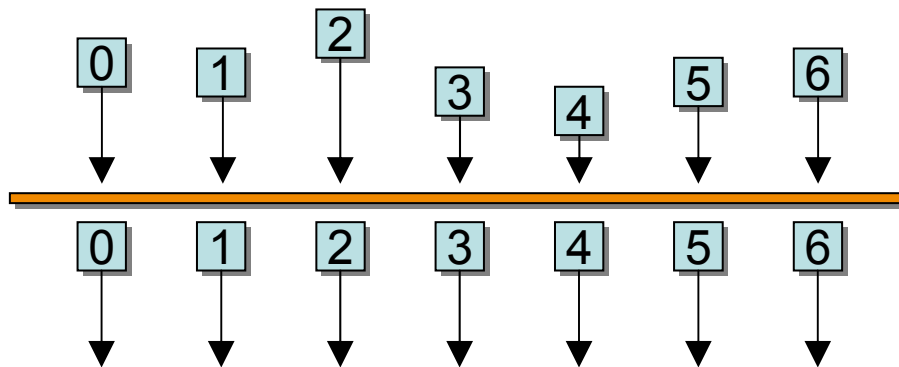
- Aufwändigere Varianten sind denkbar 😊.
- Java5 enthält eine Implementierung in `java.util.concurrent`.

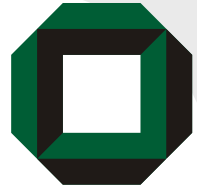


Beispiel: Barriere (1)

- Barrierensemantik:
 - Die Barriere wird **initialisiert** mit der Anzahl der Aktivitäten, die teilnehmen sollen.
 - Alle beteiligten Aktivitäten halten eine **Referenz** auf die Barriere: `barrier`.
 - Alle beteiligten Aktivitäten rufen `barrier.sync()` auf.

- **Keine** Aktivität verlässt die `sync()` Methode, **bevor nicht alle anderen** Aktivitäten die ebenfalls `sync()` aufgerufen haben.



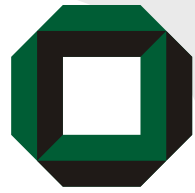


Beispiel: Barriere (2)

- Beispiel für Verwendung einer Barriere:

```
Barrier b = new Barrier(3); //global
                               //3 teilnehmende Fäden

//in jedem Kontrollfaden
for (int i=0; i<localMax; ++i){
    // Berechnungen durchführen
    b.sync();
    // Tausche Daten aus
}
```

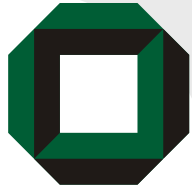


Beispiel: Barriere (3)

- Erster Versuch

```
public class Barrier {  
  
    final int cnt; /* Anzahl teilnehmender Kontrollfäden */  
    int arrived = 0;  
  
    public synchronized void sync() throws InterruptedException {  
        arrived++;  
        if (arrived == cnt) {  
            arrived = 0; notifyAll();  
        } else {  
            while (arrived > 0) wait();  
        }  
    }  
}
```

- Warum wird diese Barriere nicht funktionieren?
- Hinweis: Denken Sie daran, dass diese Barriere in einer Schleife verwendet werden kann.

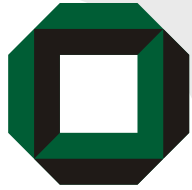


Beispiel: Barriere (4)

```
public class Barrier {  
  
    final int cnt; /* Anzahl teilnehmender Kontrollfäden */  
    int arrived = 0;  
  
    public synchronized void sync()  
        throws InterruptedException {  
        arrived++;  
        if (arrived == cnt) {  
            arrived = 0; notifyAll();  
        } else {  
            while (arrived > 0) wait();  
        }  
    }  
}
```

Beispiel mit 2 Fäden

- cnt=2; arrived=0
- T1.sync()
 - arrived=1 → else → wait()
- T2.sync()
 - //1. Durchlauf seiner for-Schleife
 - arrived=2 → if → notifyAll()
- T1 & T2 konkurrieren wieder um Monitor
 - T2 „gewinnt“, während T1 noch im while „am Ende“ vom wait() ist
- T2.sync()
 - //2. Durchlauf seiner for-Schleife
 - arrived=1 → else → wait()
 - T1 will weitermachen, befindet sich noch im while, Bedingung trifft aber wieder zu → wait()
- **Beide im wait()!**
Wer kann nun die Bedingung ändern?
→ War so nicht beabsichtigt!



Beispiel: Barriere (5)

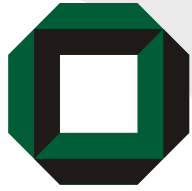
```
public class Barrier {
    private final int cnt;
    private int arrived = 0;
    private boolean state = true;

    public Barrier(int cnt) {
        this.cnt = cnt;
    }

    public synchronized void sync() throws InterruptedException {
        arrived++;
        if (arrived == cnt) {
            arrived = 0;
            state = !state; notifyAll();
        } else {
            boolean stateBeforeWait = state;
            while (stateBeforeWait == state)
                wait();
        }
    }
}
```

Der letzte stellt das
Fähnchen um

Aufgeweckte Fäden
können nicht mehr in
der While-Schleife
„hängen“



Ausblick: `java.util.concurrent`

- Viele Klassen der Standard-Bibliothek sind nicht in einer parallelen Umgebung verwendbar (nicht *thread safe*)
- Ab Java 1.5:
`java.util.concurrent` stellt weitere Klassen zur nebenläufigen Programmierung zur Verfügung
 - **Neue Sperr-Konstrukte**
z.B. explizite Sperren, Semaphore, Barrieren, Exchanger, ...
 - **Synchronisierte Datenstrukturen**
z.B. (blockierende) Schlangen, Listen, ...
 - **Organisation von Fäden**
z.B. ThreadPools, ...
 - **Futures**
 - ...