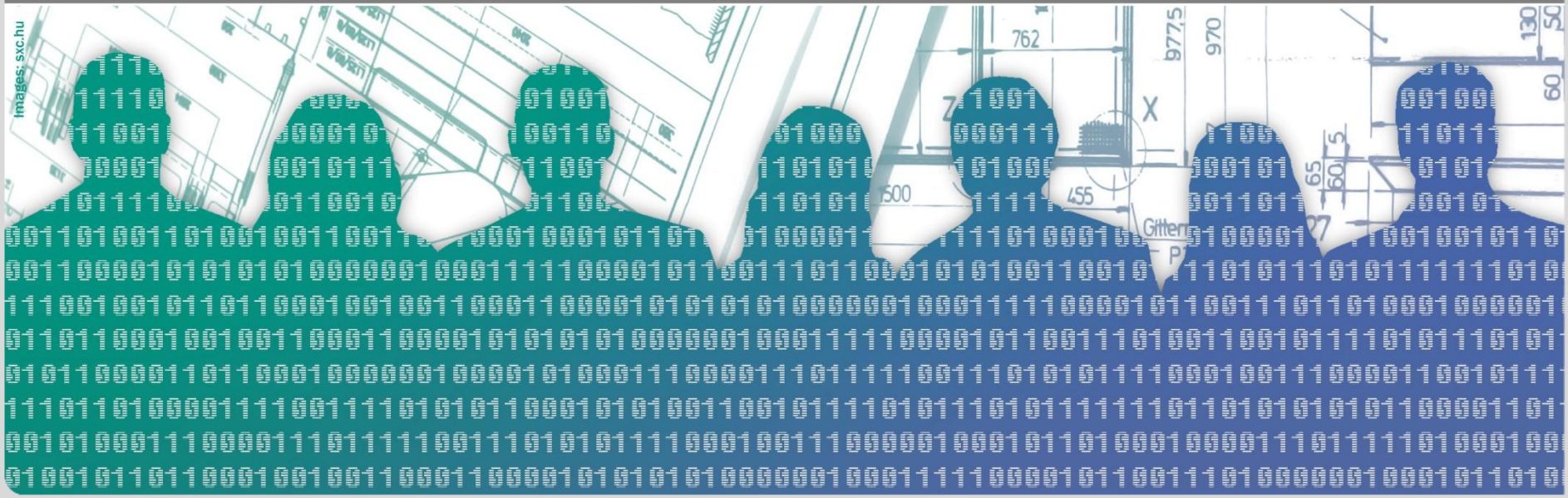
Kapitel 4.2.1 - Parallelität in Java

SWT I – Sommersemester 2010

Walter F. Tichy, Andreas Höfer, Korbinian Molitorisz

IPD Tichy, Fakultät für Informatik



Überblick

- Erzeugen von Kontrollfäden
- Konstrukte zum Schützen kritischer Abschnitte
- Konstrukte für Warten und Benachrichtigung

- **Hinweis:** Alle hier vorgestellten Techniken gelten für Java 1.5 und neuer.

Karlsruhe Institute of Technology
Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

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 **Kontrollfaden** hat
  - seinen eigenen Keller und Programmzähler.
  - Zugriff auf den gemeinsamen Hauptspeicher (Halde).
  - möglicherweise lokale Kopien von Daten des Hauptspeichers (aufgrund der Caches)

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

Konstrukte zum Erzeugen von Parallelität (1)

- Seit Java 1.0 gibt es eingebaute Klassen und Schnittstellen für parallele Programmierung:
  - Interface `java.lang.Runnable`

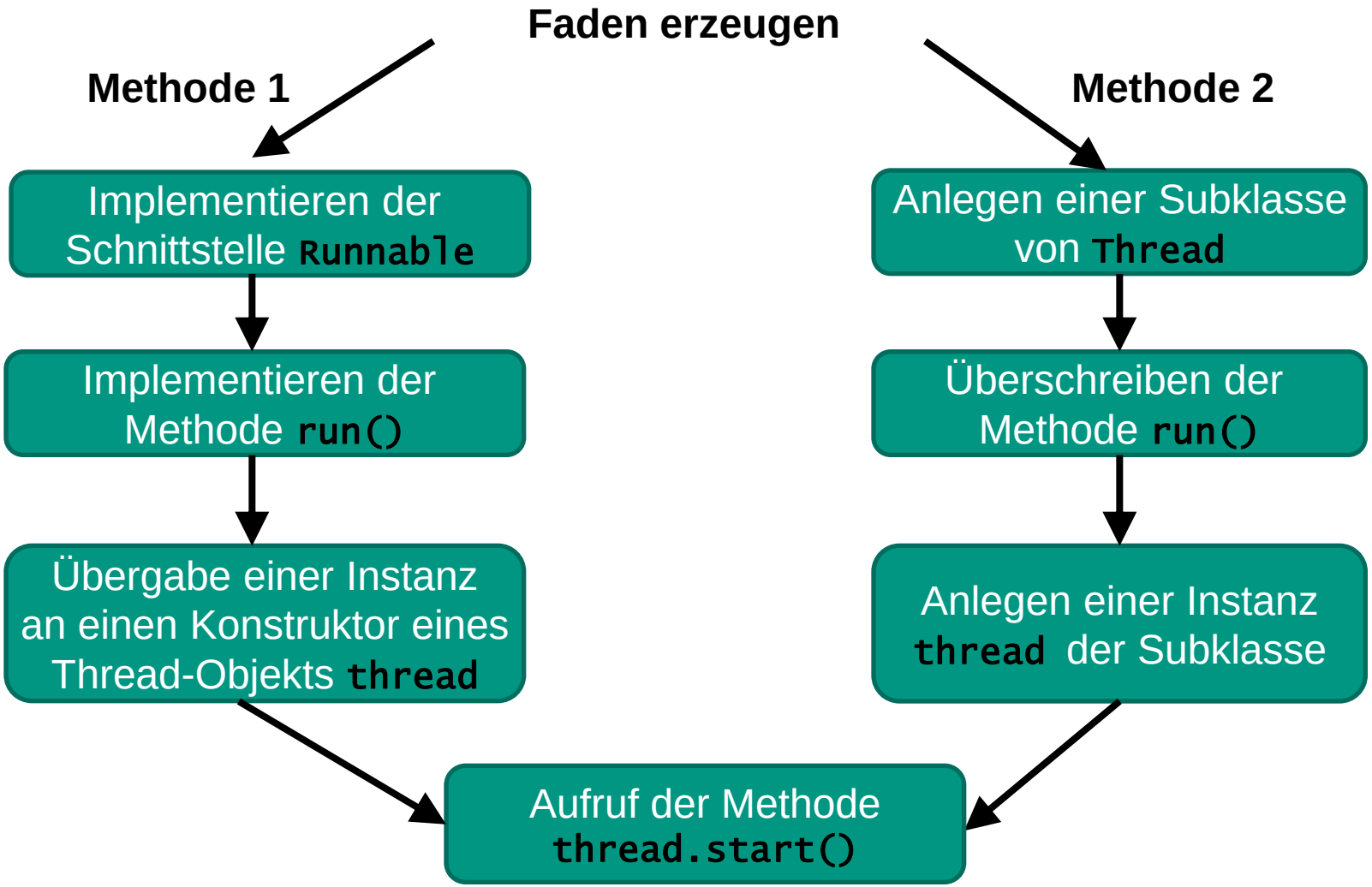
```
public interface Runnable {
  public abstract void run();
}
```

  - Klasse `java.lang.Thread`

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


Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

Konstrukte zum Erzeugen von Parallelität (2)



Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

Konstrukte zum Erzeugen von Parallelität – Beispiel (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

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

Konstrukte zum Erzeugen von Parallelität – Beispiel (2)

- Klasse, die von `Thread` erbt:

```
class ComputeRun extends Thread {
 long min, max;

 ComputeRun (long min, long max) {
   this.min = min; this.max = max;
 }

 public void run () {
   // Parallele Aufgabe
 }
}
```

- Erzeuge und starte Kontrollfaden:

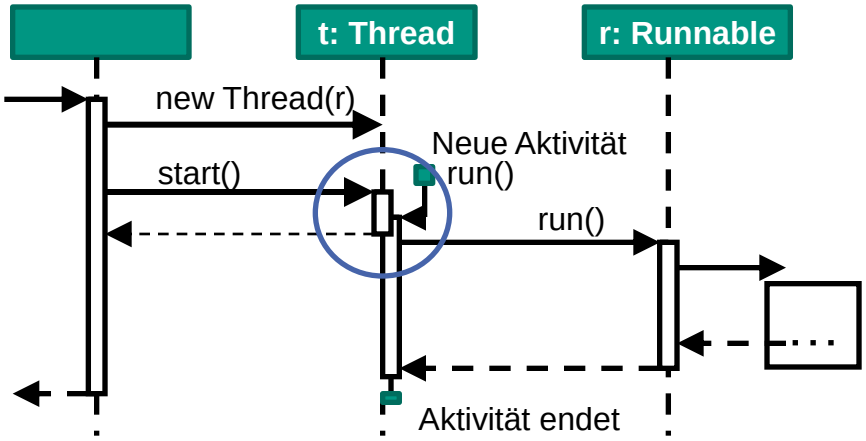
```
ComputeThread t = new ComputeThread(1, 20);
t.start();
```


Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

Konstrukte zum Erzeugen von Parallelität – Beispiel (3)

- Auch Konstrukt mit anonymer, inneren Klasse möglich:

```
Thread t = new Thread (
  new Runnable() {
    public void run() {
      //Parallele Aufgabe
    }
  }
);
t.start();
```



- `Thread` implementiert selbst die Schnittstelle `Runnable`.
- Beim Start (`start()`) der neuen Aktivität führt diese als erstes die `run()`Methode des eigenen Thread-Objekts aus.
- Die Implementierung von `Thread.run()`wiederum ruft `run()`an dem an den Konstruktor übergebenen `Runnable`Objekt auf.

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

Runnable vs. Thread

- 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 über das Netzwerk versendet werden (`Thread` ist nicht serialisierbar).

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

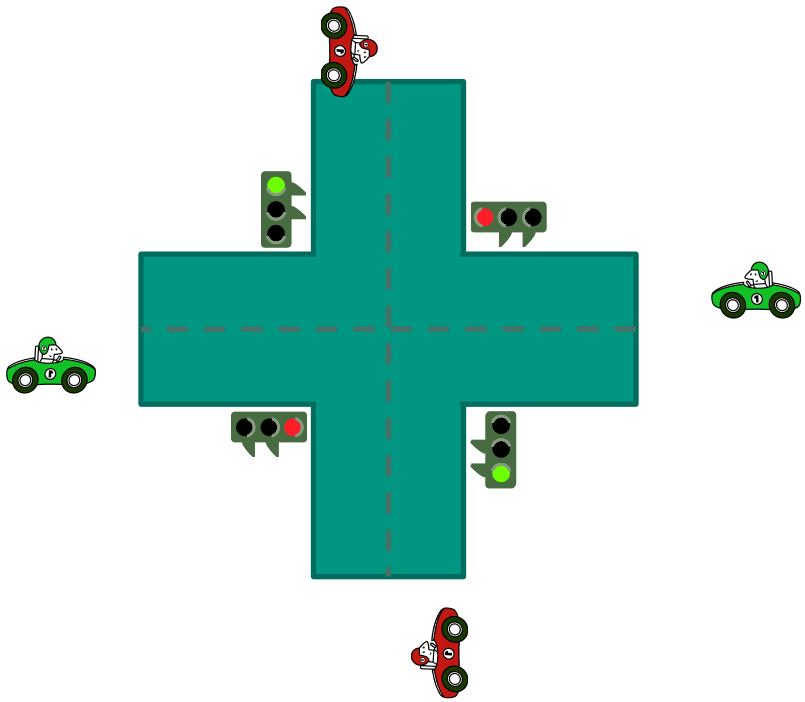
Koordination

- Grundlegende Koordinationsmechanismen sind in die Sprache eingebaut.
  - Wechselseitiger Ausschluss (engl. Mutual exclusion):
    - 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.

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

Koordination: Beispiel

- Eine Straßenkreuzung mit Ampelanlage:



Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

Koordination: Wofür?

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

```
int globalvar = 1;

…

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

- Kann `globalvar` negativ werden?

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

Koordination: Wofür?

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

Thread 1	Thread 2
	`// globalvar == 1`
`if (globalvar > 0) {`	
	`if (globalvar > 0) {`
	`  globalvar--;`
	`}`
`  globalvar--;`	
`}`	

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

Koordination: Kritische Abschnitte (1)

- Ein Bereich, in dem ein Zugriff auf einen gemeinsam genutzten Zustand stattfindet, ist ein **kritischer Abschnitt** (engl. *critical section*).
- Um Wettlaufsituationen 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.

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

Koordination: Kritische Abschnitte (2)

- **Atomarität:** Java garantiert, dass Zugriffe auf einzelne Variablen atomar erfolgen.
  - Ausgenommen sind Variablen vom Typ **`double`** und **`long`** (wegen 64 Bit statt 32 Bit)
- Zugriffe auf mehrere Variablen, 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).

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

Koordination: Monitor (1)

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

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

Koordination: Monitor (2)

- 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.

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

Koordination: Monitor (3)

- 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()
```


Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

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-Verifizierer (u.a.) die Schachtelung der Monitor-Anforderungen und -Freigaben überprüft.

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

Koordination: Monitore & Synchronisation (2)

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

```
/*synchronisierter Block*/
synchronized (obj) {
  // kritischer
  // Abschnitt
}
```

```
/*synchronisierte Methode*/
synchronized void foo(){
  // ganze Methode
  // kritischer Abschnitt
}
```

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

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

Koordination: Monitore & Synchronisation (3)

- Bei der Ausführung eines synchronisierten Blocks…
  - der Monitor des zugehörigen Objekts wird betreten: `enter()`
  - die Anweisungen des Block werden ausgeführt
  - der Monitor wird 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`

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

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.
- Die selbe Aktivität kann einen Monitor beliebig oft betreten (sinnvoll z.B. bei Rekursion).

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

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?

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

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.
  - Illustration: Puffer begrentzer Kapazität

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

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();
}
```

Falsch!

Karlsruhe Institute of Technology
Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

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.

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

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.

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

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();
```


Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

Koordination: Konstrukte für Warten und Benachrichtigung (2)

- Um `wait` und `notify` aufrufen zu können, muss die aktuelle Aktivität mit `synchronized` 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` ausgelöst.
- In synchronisierten Methoden ist der Monitor `this`, und statt `this.{wait|notify|notifyAll}()` kann `wait()` alleine aufgerufen werden.

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

Koordination: Konstrukte für Warten und Benachrichtigung (3)

- `wait()` versetzt aktuellen Faden in einen Wartezustand, bis ein Signal bei diesem Objekt eintrifft. Folgendes passiert:
  - Die aktuelle Aktivität wird „schlafengelegt“, d.h gibt den Prozessor ab.
  - 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 nun um den Monitor konkurrieren.
- Variante: `wait(long timeout, [int nanos])` beschränkt die Wartezeit auf den angegebenen Wert und kehrt dann zurück.
  - Die Wartezeit kann dennoch länger ausfallen, weil u. U. auf die Freigabe des Monitors gewartet werden muss.

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

Koordination: Konstrukte für Warten und Benachrichtigung (4)

- `notify()` und `notifyAll()` schicken Signale an wartende Aktivitäten des betreffenden Objekts. Folgendes passiert:
  - „Schlafende“ Aktivitäten befinden sich in einer mit dem Monitor assoziierten Warteschlange.
    - `notify()` schickt Signal an irgendeine Aktivität aus dieser Warteschlange.
    - `notifyAll()` schickt Signal an alle Aktivitäten dieser Warteschlange.
  - „Aufgeweckte“ Aktivitäten werden aus der Warteschlange entfernt und konkurrieren erneut um den Monitor
    - Zunächst müssen alle warten, bis die Aktivität, die `notify[All]` aufgerufen hat, den Monitor freigibt

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

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();
}
```


Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

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.
- Warten auf Signale alleine kann zu Fehlern führen, z.B.
  - `wait()` ohne Wächterbedingung
    - FALSCH: ... **`synchronized(obj) {wait(); }`** ...
  - `notify()` ohne Zustandsänderung
    - FALSCH: ... **`synchronized(obj) {notify(); }`** ...
  - Warum?

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

Koordination: Sicherheitshinweise & Faustregeln (2)

- Auszug aus der Sprachspezifikation von Java (S. 581)
  - „Implementations are permitted, although not encouraged, to perform **"spurious wake-ups"** –
    - to remove threads from waitsets and
    - Thus enable resumption without explicit instructions to do so.
    - Notice that this provision necessitates the Java coding practice of using wait only within loops that terminate only when some logical condition that the thread is waiting for holds.“
- Spurious: unecht, falsch, überzählig.
- D.h. Aktivitäten können zufällig aufgeweckt werden, ohne dass sich am Zustand eines Objektes etwas geändert hat.

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

Koordination: Sicherheitshinweise & Faustregeln (3)

- Prüfe immer vor und nach dem Warten auf die Bedingung!
  - Verwende `wait()` immer in einer Schleife!
    - FALSCH: ... `if (! istBedingung) {wait(); }` ...
  - Gründe
    - Vorher: Kontrollfaden nicht unnötigerweise „schlafen legen“
    - Nachher: Nicht sichtbar, warum ein Signal geschickt wurde. „Aufwachen“ könnte durch ein „falsches“ Signal verursacht 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.

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

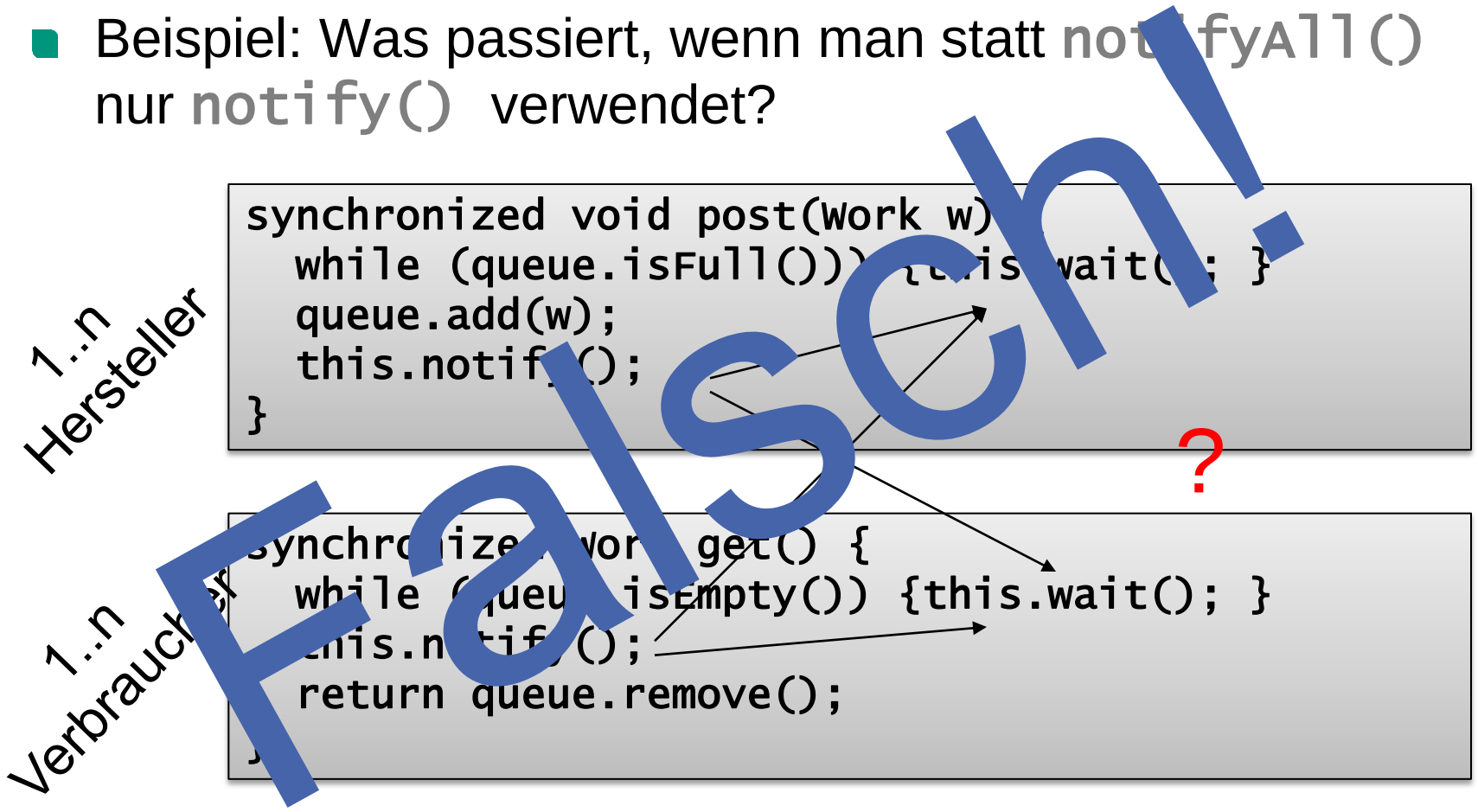
Koordination: Sicherheitshinweise & Faustregeln (4)

- 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.

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

Koordination: Sicherheitshinweise & Faustregeln (5)

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



Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

Koordination: Sicherheitshinweise & Faustregeln (6)

- 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.

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

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? Naiv: Sonderfall einführen, zum Beispiel eine null-Referenz in die Liste der Arbeitspäckchen einstellen.
    - Aber: Was ist, wenn null schon „besetzt“, oder einfach ein gültiger Wert ist?
- Einfacher mit Java-Sprachkonstrukten: Man schickt den betreffenden Aktivitäten eine Unterbrechung (`InterruptedException`).

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

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.

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

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, löst `wait()` eine `InterruptedException` aus.
  - Die Unterbrechung ist selbst nicht unterbrechbar
  - Die `InterruptedException` muss deklariert und weitergeleitet oder abgefangen und behandelt (`catch`) werden.

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

Unterbrechung (4) - Faustregeln

- Wenn für eine `InterruptedException` keine vernünftige Behandlung angeben werden kann, sollte man sie mit `throw` weiterleiten
  - Das gilt wie für jede andere Ausnahmebedingung auch…
  - Der umgebende Code bzw. Aufrufer sollte 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 „gefangen“ werden.
    also nicht:

```
try {
  wait();
} catch (InterruptedException ex)
{/*nichts*/}
```


Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

Verklemmungen (1)

- Trotz `synchronized` können noch weitere Probleme auftreten.
- Verklemmung, Blockaden
  (engl. *deadlock*, *deadly embrace*)
  - Blockade, die durch eine zyklische Abhängigkeit hervorgerufen wird.
  - Führt dazu, dass alle beteiligten Fäden im Wartezustand verharren.

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

Verklemmungen (2)

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

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) { rechne();}
    }
  }
});

Thread t2 = new Thread(new Runnable() {
  pubilc  void  run(){
    synchronized (resource2) {
      synchronized (resource1) { rechne();}
    }
  }
});

t1.start(); // sperrt resource1
t2.start(); // sperrt resource2; jetzt beide blockiert!
```

Reparaturvorschläge?

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

Beispiel: Vektoraddition (1)

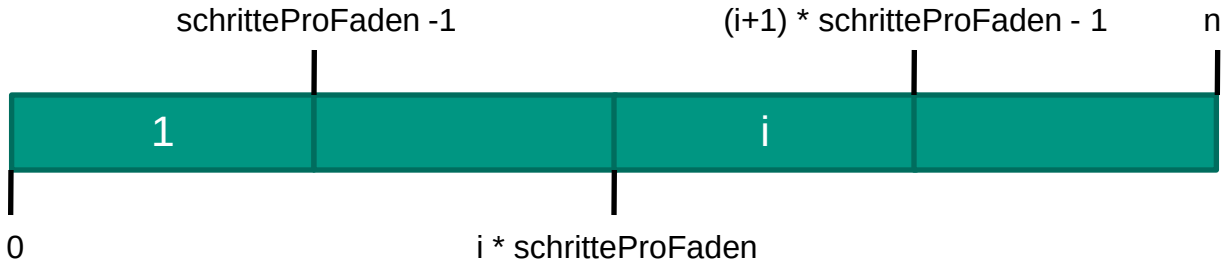
- Wir wollen eine parallele Vektoraddition durchführen.
- Wichtigster Bestandteil: Die Klasse Arbeiter, die zwei (Teil-) Vektoren elementweise addiert und das Ergebnis in einem dritten Vektor speichert.

```
class Arbeiter implements Runnable {
  private int[] a, b, c;
  private int links, rechts;

  publicArbeiter (int[] a, int[] b, int[] c, int links, int rechts) {
    this.a = a; this.b = b; this.c = c;
    this.links = links;  this.rechts = rechts;
  }

  public void run() { // addiert Teilvektoren im Segment [links..rechts)
    for (int i = links; i < rechts; i++) {
      c[i] = a[i] + b[i];
    }
  }
}
```


Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

Beispiel: Vektoraddition (2)

- **Illustration:** Funktionsweise der parallelen Vektoraddition
- Jeder Faden `i` bearbeitet ein Segment der Länge `schritteProFaden`.



Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

Beispiel: Vektoraddition (2)

- Die Main-Methode definiert zunächst
  - Die Vektoren,
  - Konstanten, um den Indexbereich der Vektoren aufzuteilen.

```
public static void main(String[] args) {

  int[] a = … // füllen
  int[] b = … // füllen

  assert a.length == b.length;
  int[] c = new int[a.length];

  final int anzahlFäden = 10;
  final int n = a.length;
  final int schritteProFaden = (int) Math.ceil((double) n / anzahlFäden);
```


Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

Beispiel: Vektoraddition (3)

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

```
Thread[] mannschaft = new Thread[anzahlFäden];

for (int i = 0; i < anzahlFäden; ++i) {
  int links = i * schritteProFaden;
  int rechts = Math.min((i + 1) * schritteProFaden, n);

  mannschaft[i] =
    new Thread(new Arbeiter(a, b, c, links, rechts));
  mannschaft[i].start(); //Faden i läuft nun
}
```


Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

Beispiel: Vektoraddition (4)

- Nun muss auf die Beendigung der Fäden gewartet werden. Das geschieht mit `join()`
  - Anmerkung: `join()` ist auch über `InterruptedException` unterbrechbar.

```
for (Thread thread : mannschaft) {
  try {
    thread.join(); /* wartet, bis Faden endet */
  } catch (InterruptedException ex) {
    System.err.println(”Unerwartete Unterbrechung" +
                       ”beim Warten auf Arbeiter");
  }
}

//Jetzt das Ergebnis in c[] benutzen
```


Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

Ergänzungen in der Java-Bibliothek java.util.concurrent

- Die Verwendung von expliziten bzw. feingranularen Sperren ist oft fehleranfällig.
- Viele der Datenstrukturen (z.B. Schlangen) aus dem sequenziellen Fall sind im parallelen Fall nicht verwendbar, wenn mehrere Fäden simultan darauf arbeiten (engl. *not thread-safe*).
- `java.util.concurrent` stellt weitere Klassen zur parallelen Programmierung zur Verfügung.

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

java.util.concurrent

- Nützlicher Funktionalität, die immer wieder gebraucht wird.
- Kategorien
  - **Mengen** (engl. *collections*)
  - **Konstrukte zur asynchronen Ausführung, Thread Pools**
  - **Synchronisierer** (engl. *synchronizers*)

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

java.util.concurrent – ConcurrentCollections

- `java.util.concurrent` bietet zusätzliche Mengen-Typen sowie atomare Operationen für sichere Nutzung im parallelen Fall, z.B.:
  - `ConcurrentHashMap`
  - `ConcurrentLinkedQueue`
  - `CopyOnWriteArrayList` (CopyOnWrite: jede Schreiboperation kopiert die Daten, d.h. schon begonnene Iteratoren werden nicht gestört.)
  - `CopyOnWriteArraySet` (ebenfalls kopierend)
- Einige benutzen aus Performanzgründen keine Monitore, sondern explizite Sperren und `volatile`-Modifizierer
  - (`volatile` bewirkt, dass eine Variable nicht im Zwischenspeicher belassen wird, sondern immer aktualisiert wird. Die parallel ausgeführten Fäden sehen somit immer den korrekten Variablenwert, da er vor jeder Benutzung aus dem Speicher gelesen und nach einer Änderung sofort wieder zurückgeschrieben wird)

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

java.util.concurrent – Schlangen (1)

- Blockierende Schlangen
  - Implementieren `BlockingQueue`-Schnittstelle
  - Datenaustausch zwischen Fäden gemäß dem Erzeuger-Verbraucher-Muster
  - Blockierende Methoden: `put`, `take`
    - Erzeuger blockiert, wenn Schlange voll; Verbraucher blockiert, wenn Schlange leer
  - Auch zeitlich beschränke Blockierung möglich
    - `offer/poll` blockieren nur für eine bestimmte Zeit

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

java.util.concurrent – Schlangen (2)

- Verschiedene Implementierungen
  - `ArrayBlockingQueue`
    basiert auf Array mit fester Größe
  - `DelayQueue`
    ordnet Elemente nach Wartezeiten an
  - `LinkedBlockingQueue`
    basiert auf verketteter, linearer Liste
  - `PriorityBlockingQueue`
    ordnet Elemente gemäß einer Vergleichsoperation
  - `SynchronousQueue`
    basiert auf Schlange mit Kapazität 0

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

java.util.concurrent – Semaphore

Werden zur Synchronisation zwischen Fäden verwendet

- **Semaphore**
  - Wird mit einer Anzahl von „Genehmigungen“ initialisiert.
  - `acquire` blockiert, bis eine „Genehmigung“ verfügbar ist und erniedrigt anschließend Anzahl der „Genehmigungen“ um 1 (oder Parameterwert)
  - `release` erhöht Anzahl der „Genehmigungen“ um 1 (oder Parameterwert).

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

Beispiel: Semaphore

```
public class Semaphore {
  private int count;
  public synchronized void aquire()
    throws InterruptedException {

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

- Aufwändigere Varianten sind denkbar.
- Java 5 enthält eine Implementierung in `java.util.concurrent`.

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

java.util.concurrent – Barrierensynchronisation

- **CyclicBarrier**
  - Synchronisiert Gruppe von n Fäden.
  - Fäden rufen `await()`-Methode der Barriere auf, die so lange blockiert, bis n Fäden warten.
  - Danach wird den Fäden erlaubt, ihre Ausführung fortzusetzen (die Barriere wird zurückgesetzt).
  - Erweiterung: Zusätzliche Runnable-Methode wird ausgeführt, wenn der Letzte von n Fäden `await()` aufgerufen hat.
  - Beim Vektor-Beispiel statt `join()`-Schleife einsetzbar.
  - Zyklische Barriere, weil sie wiederholt benutzt werden kann.

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825

Mehr zu diesem Thema …

- Im Java Tutorial von Sun:
  http://java.sun.com/docs/books/tutorial/essential/concurrency/
- In der Java Dokumentation:
  http://java.sun.com/javase/6/docs/api/
- Im Buch „Java ist auch eine Insel“:
  http://openbook.galileocomputing.de/javainsel8/javainsel_11_001.htm

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH) Forschungsuniversität • gegründet 1825