08-Threads-01

Objektorientierte Programmierung | Matthias Tichy

Software Engineering
Programming Languages

universität ulm

Lernziele

- Eigenschaften von Threads
- Arbeiten mit Threads
- Synchronisierung von Zugriffen auf Ressourcen

Eigenschaften von Threads

Beispiele für Nebenläufigkeit

- **Betriebssysteme**
  - Prozesse verschiedener Nutzer
- **Technische Systeme**
  - Nebenläufige Steuerung und Regelung technischer Systeme, meist in Echtzeit (z.B. Elektronik-Kompo-nenten im Flugzeug)
  - permanentes Reagieren auf ankommende Nachrichten
- **Systeme auf Parallelrechnern**
  - rechenintensive Simulationen (Wettervorhersage)

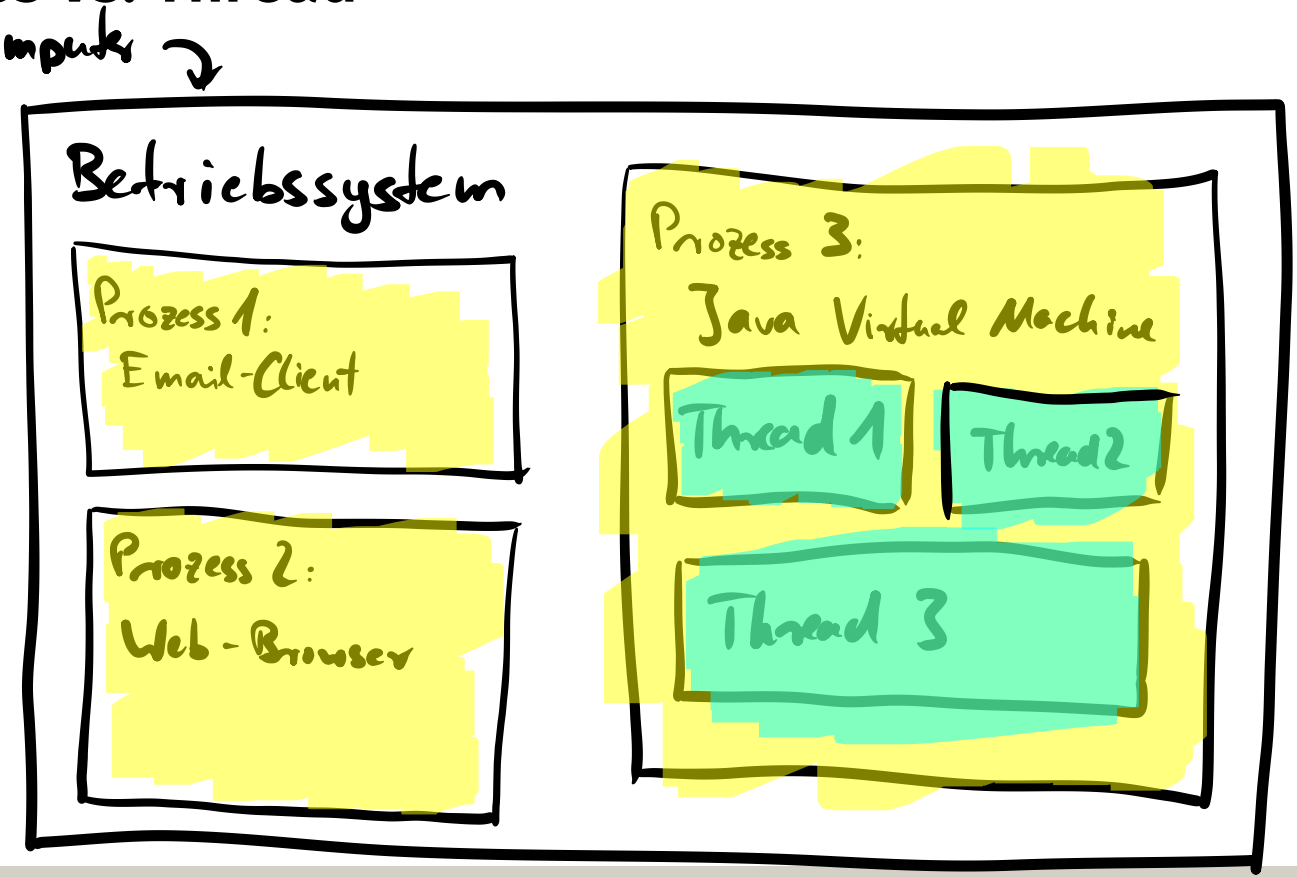
Beispiele für Nebenläufigkeit

- **Web Services**
  - Annahme und Ausführung von Dienstaufträgen auf Server
- **Graphische Oberflächen**
  - Beispiel: Web-Browser: Holen von HTML-Seiten, Holen von Bildern über das Internet, Reaktion auf Benutzereingaben, Anzeige am Bildschirm, ...
  - Permanente Reaktion auf ankommende Events
  - wie bei technischen Systemen aber ohne Echtzeitanforderung

Begriffsdefinitionen

- **Sequentielles System**
  - Rechner führt eine einzige Sequenz von Anweisungen aus
- **Nebenläufiges System (*Concurrent System*)**
  - mehrere Ströme von Anweisungen, die unabhängig voneinander abgearbeitet und entweder parallel oder pseudoparallel ausgeführt werden
  - **Parallelität:** Anweisungen können echt gleichzeitig aus-geführt werden (z.B.: Mehrprozessor-Systeme, Rechnernetze
  - **Pseudoparallelität**: Durch häufigen Wechsel zwischen den Abläufen wird Parallelität simuliert
- **Verteiltes System (*Distributed System*)**
  - räumliche (oder auch konzeptionelle Aufteilung) der einzelnen Komponenten eines Systems

Prozess vs. Thread



Prozess vs. Thread

- **Prozess**
  - Vorgang mit eigenen Ressourcen (z.B. Adressraum)
  - wird vom Betriebssystem verwaltet
- **Thread**
  - läuft innerhalb eines Prozesses und nutzt dessen Ressourcen
  - wird vom Benutzerprogramm (z.B. Java Virtual Machine) verwaltet
  - Leichtgewichtiger Prozess *(engl. Lightweigt Process)*
- bei theoretischer Betrachtung Unterschied irrelevant

Eigenschaften

- **Synchronisation: Koordination nebenläufiger Abläufe**
  - regelt den Zugriff auf gemeinsame Ressourcen
- **Verklemmung (Deadlock)**
  - Jeder Ablauf wartet auf eine Bedingung, die nur ein anderer (ebenfalls wartender) Ablauf beseitigen kann
- **Aushungerung (Starvation)**
  - Ein Ablauf kommt nie dran
- **Faire Zuteilung von Ressourcen**
  - Jedem wartenden Ablauf wird eine benötigte Ressource irgendwann zugeteilt

Interaktion zwischen Prozessen

- durch Zugriff auf gemeinsame Ressourcen
- durch Nachrichtenaustausch

- Beispiel: Viele Köche in einer Küche
  - Köche benötigen Zugriff auf Quirl, Backofen, Dampfgarer, etc. ➔ falls belegt, warten
  - Geschirrspüler nicht als gemeinsames Betriebsmittel, sondern Mitarbeiter für den Betrieb verantwortlich
    - Schmutzige Geschirr als Nachricht an Mitarbeiter ("Bitte spülen!")
    - Koch kann weiterkochen oder warten

Zugriff auf gemeinsame Ressourcen

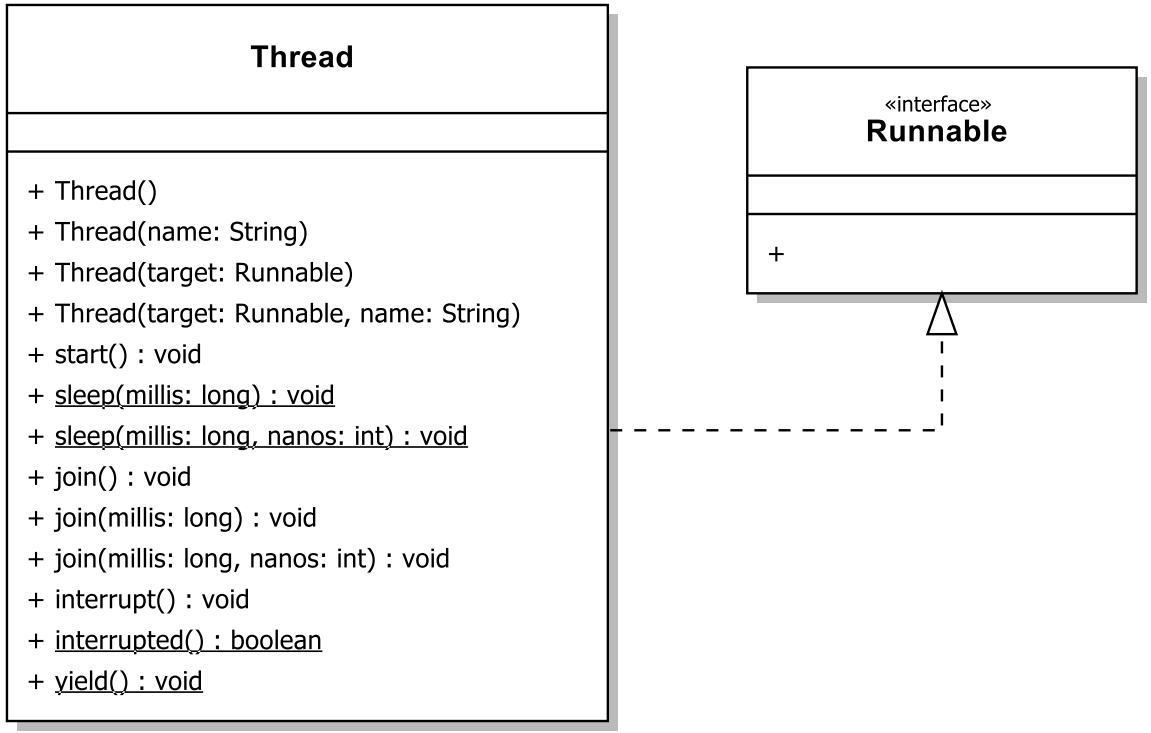
- In unserem Fall: Variablen, Speicherbereiche, ..
- Zugriff muss synchronisiert werden, um
  - konsistenten Zustand zu gewährleisten
  - Ressource nach bestimmter Strategie zu verteilen
- Klassische Synchronisationskonzepte:
  - Semaphore
  - Monitore

Zugriff auf gemeinsame Ressourcen

- In unserem Fall: Variablen, Speicherbereiche, ..
- GANZ WICHTIG:
  - Gemeinsame Ressourcen soweit wie möglich minimieren!
  - Seiteneffektfrei programmieren.
  - Lieber Daten austauschen als auf gemeinsame Daten zugreifen
  - Genau überlegen/dokumentieren, wer/wann auf die gemeinsamen Ressourcen zugreifen darf!
  - NICHT: Einfach wild (random) mit dem `synchronized` Keyword (für Monitore) auf Code drauf hauen!

Arbeiten mit Threads

Threads in Java

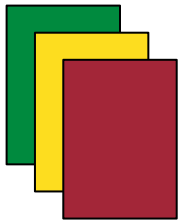


Vererbung von Thread

```
public class SimpleThread1 extends Thread {

   public void run() {
      System.out.println("Hello, concurrent world!");
   }

   public static void main(String[] args) {
      SimpleThread1 t = new SimpleThread1();
      t.start();
   }
}
```

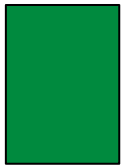
Realisierung von Runnable

```
public class SimpleThread2 implements Runnable {

   public void run() {
      System.out.println("Hello, concurrent world!");
   }

   public static void main(String[] args) {
      SimpleThread2 runner = new SimpleThread2();
      Thread t = new Thread(runner);
      t.start();
   }
}
```

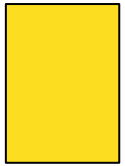


Warum zwei Möglichkeiten?

Es gab Streit zwischen den Entwicklern von Java

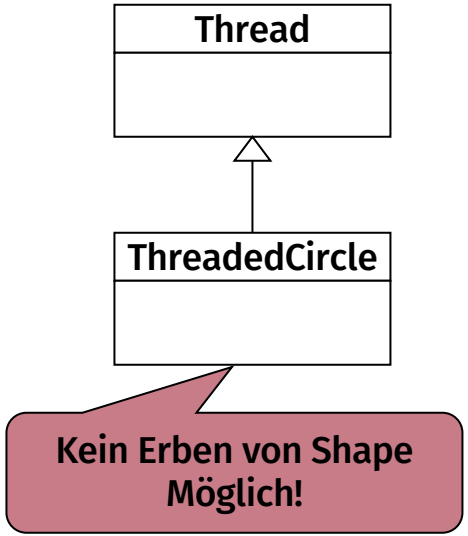


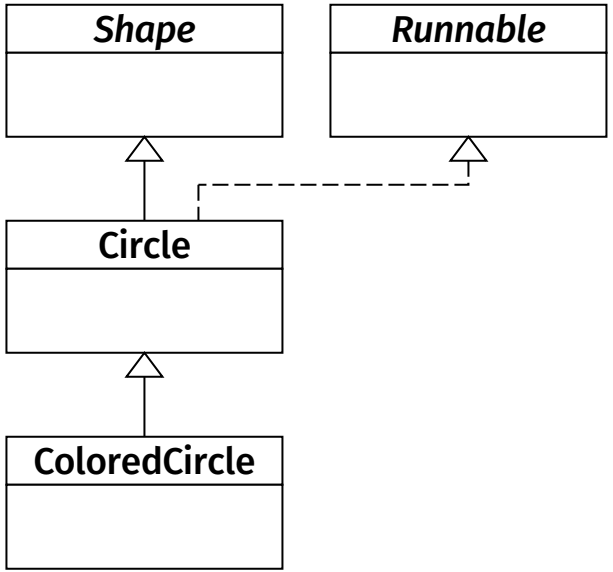
Weil es in Java keine Mehrfachvererbung gibt



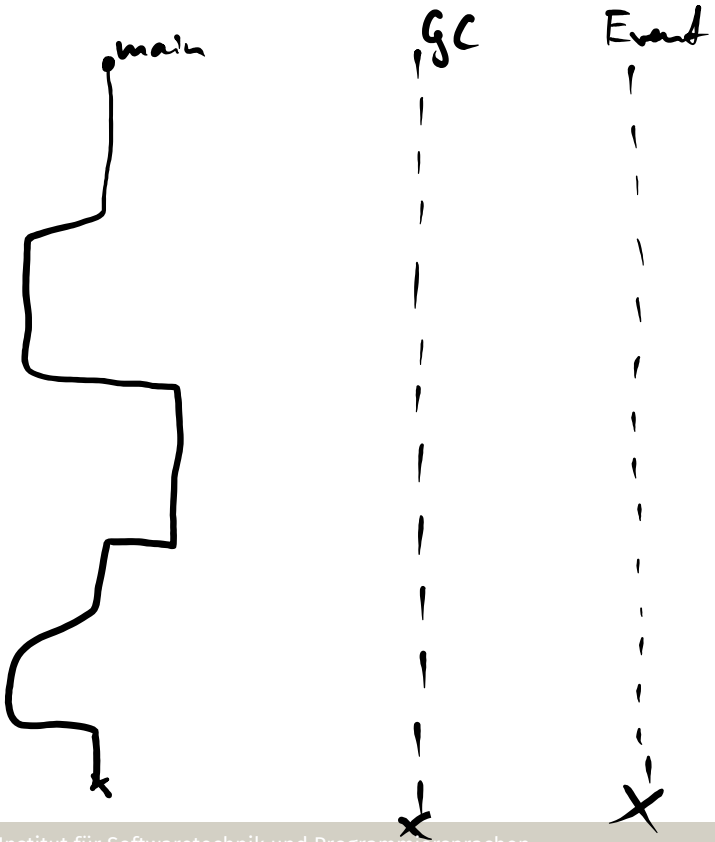
Es sind ja gar nicht wirklich zwei Möglichkeiten, da `Thread` ja auch nur `Runnable` implementiert

Darum


Thread
ThreadedCircle
Kein Erben von Shape
Möglich!


Shape
Runnable
Circle
ColoredCircle

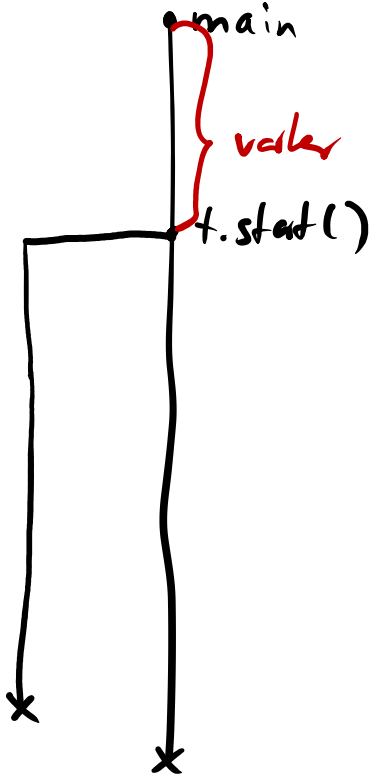
Fäden ziehen



Fäden ziehen

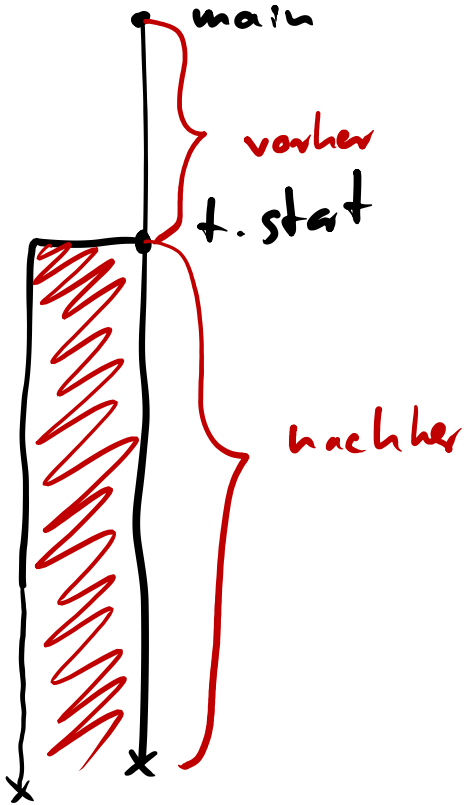
Threads/…/OnlyMain.java

Fäden ziehen



Threads/…/MainAndT.java

What happens before?

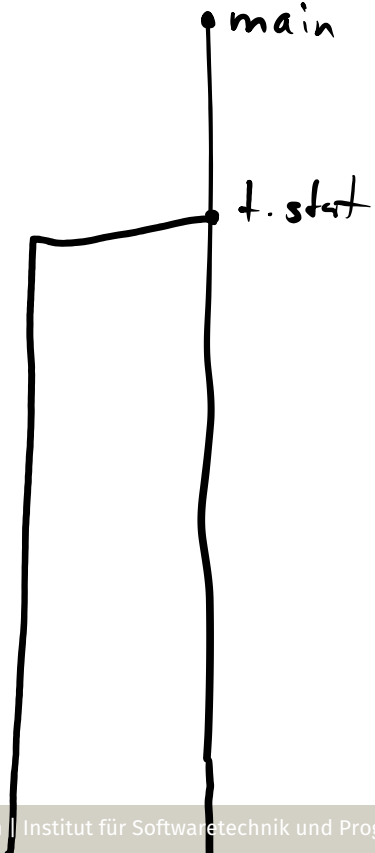


Threads/…/MainAndT.java

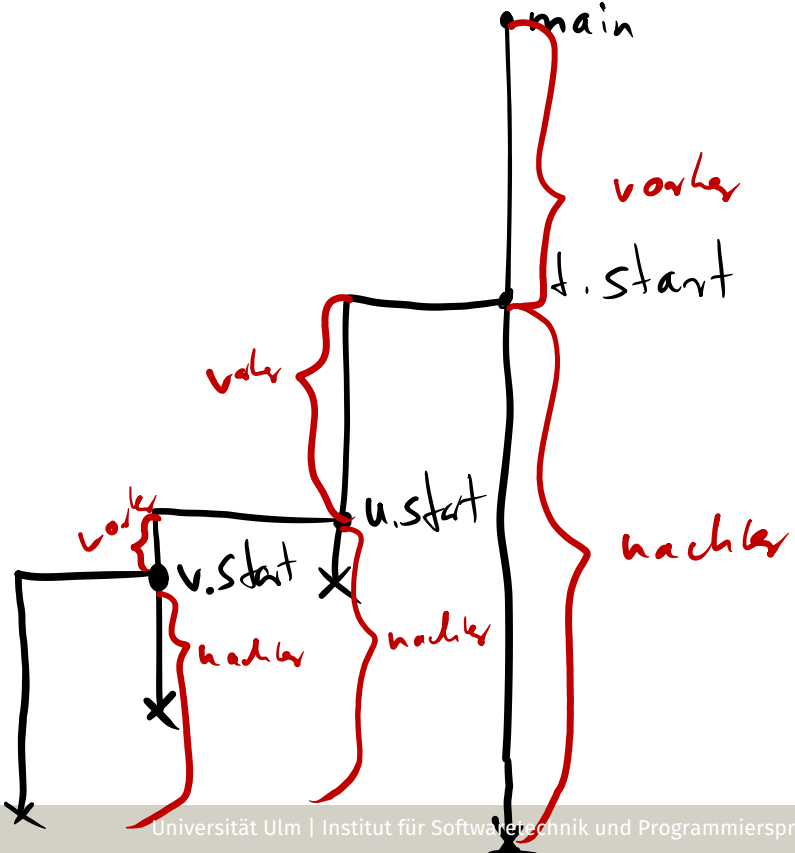
Der unendliche Faden

Threads/…/MainWhile.java

zwei unendliche Fäden

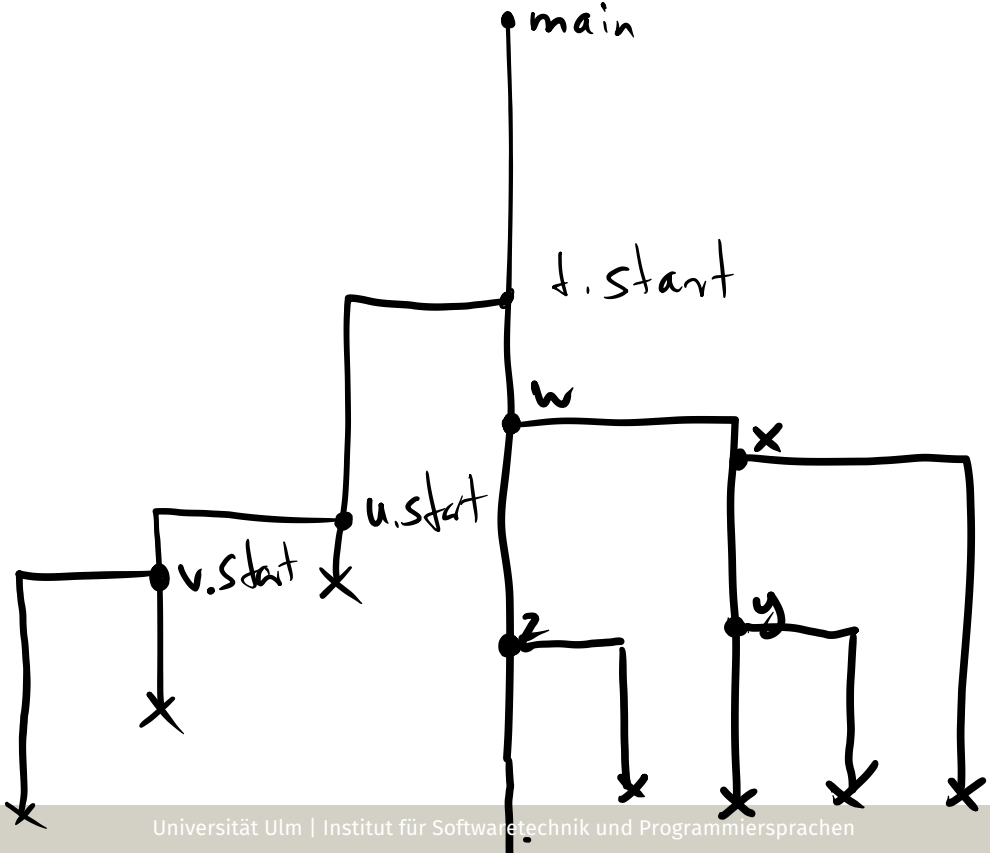


mehrere Fäden



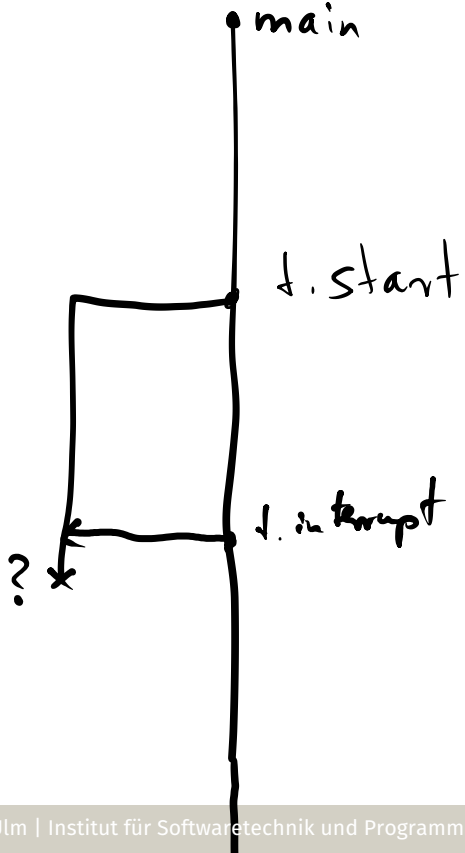
Threads/…/MainTUV.java

viele Fäden



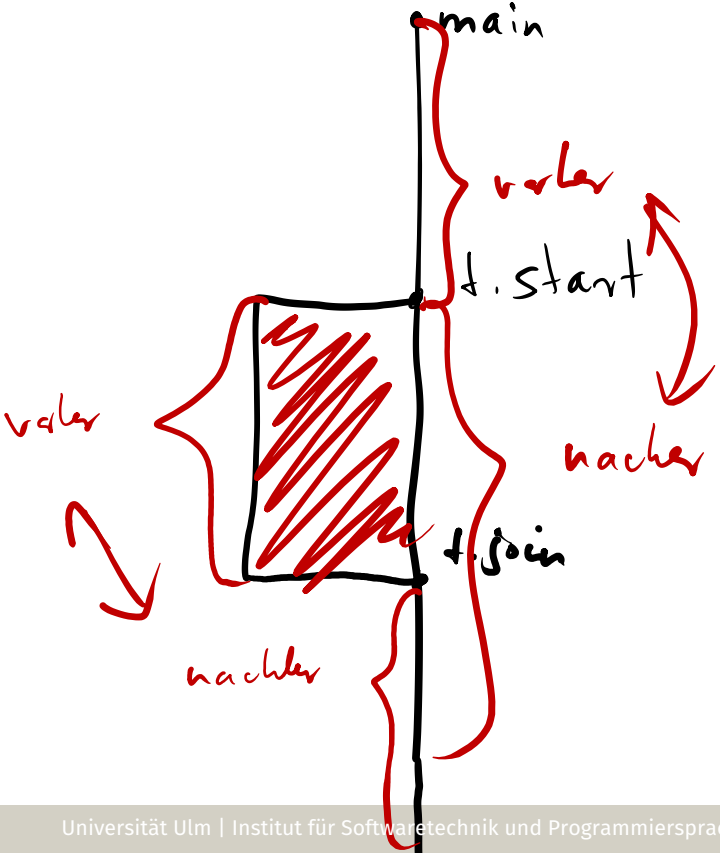
Threads/…/ManyThreads.java

Bitte aufhören



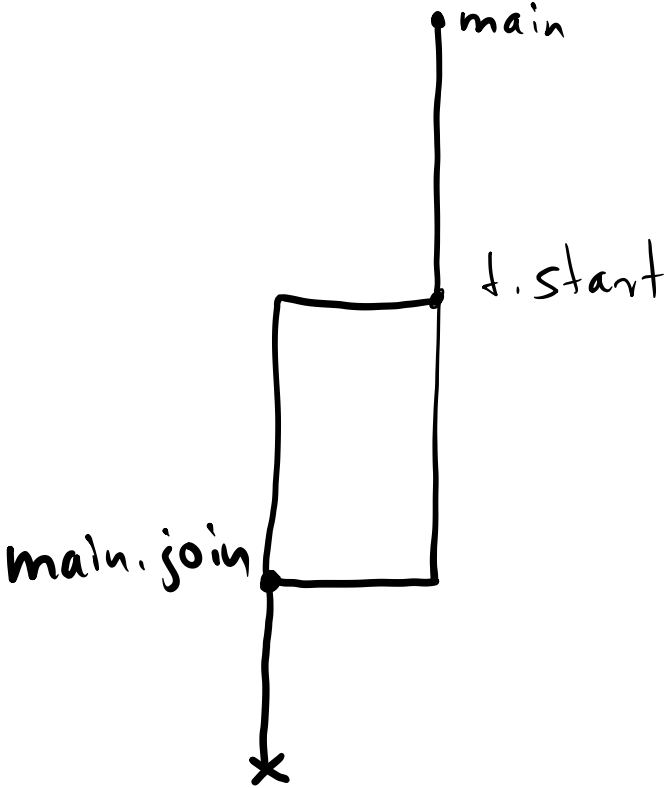
Threads/…/Interrupt1.java

Zöpfe flechten



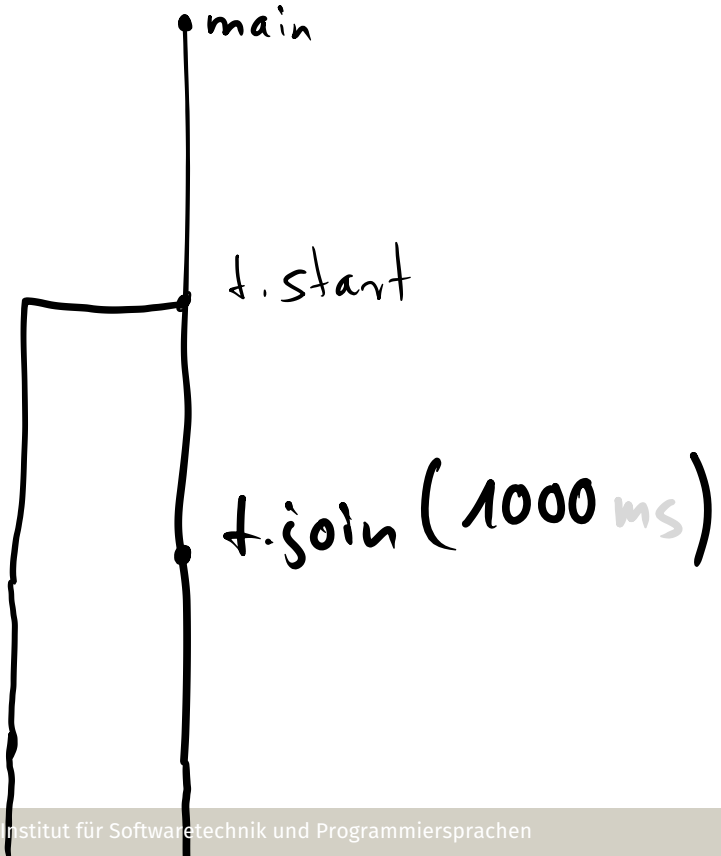
Threads/…/Join1.java

Zöpfe flechten

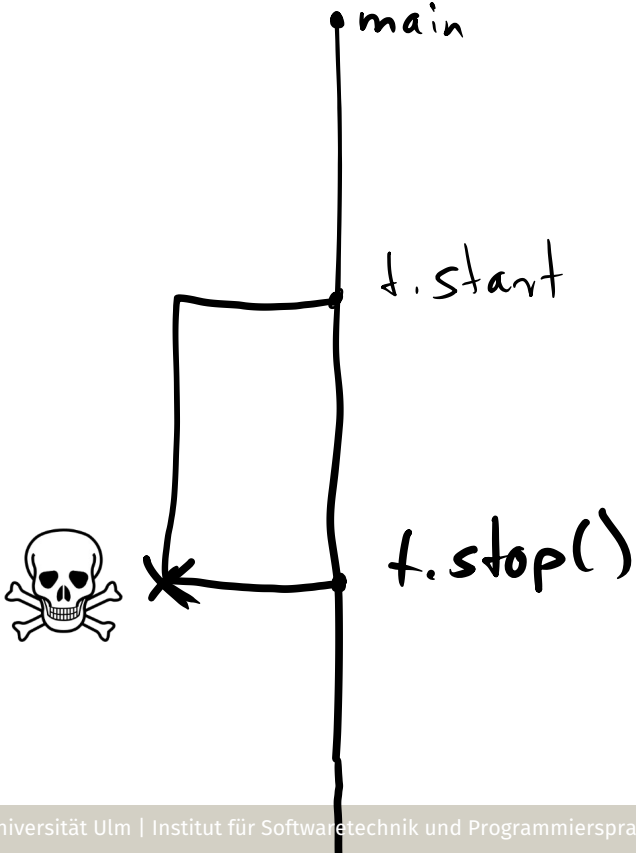


Threads/.../Join2.java

Zöpfe flechten



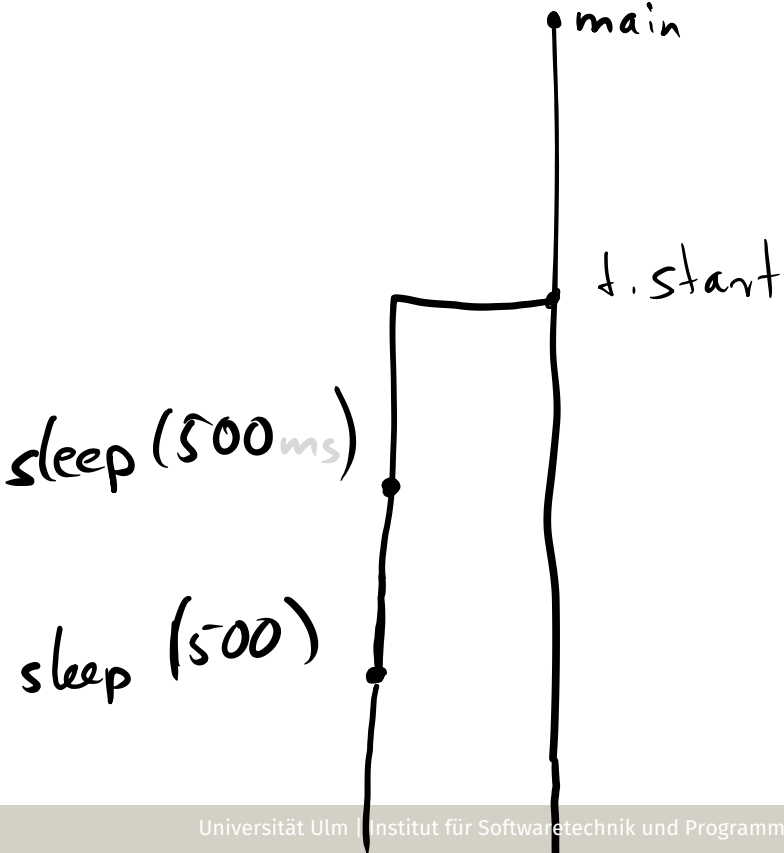
Jetzt reicht's!



Probleme mit Thread.stop()

- Thread wird einfach "abgeschossen", also während seiner Arbeit unterbrochen
- Inkonsistente Daten in anderen Objekten können die Folge sein
- "Abschießen" könnte mit "ThreadDeath-Exception" abgefangen werden, aber man weiß nicht, wo die Exception auftrat ➔ schwierig aufzuräumen!
- → Nie Benutzen, Deprecated.

gepflegtes Nichtstun



Sleep / yield

- Sleep:
  - Zeitangabe nicht exakt
  - wird bei Unterbrechung (`interrupt`) sofort mit `InterruptedException` abgebrochen
- Yield:
  - Rechenzeit weitergeben
- Siehe:
  - https://infinitescript.com/2014/09/difference-between-wait-and-sleep-yield-in-java/

Wettrennen

Synchronisierung der Zugriffe auf Ressourcen

Kontobeispiel

Kontobeispiel

- Probleme, Konsistenzfehler zu finden, da Fehlverhalten nicht immer auftritt

- Ein Befehl in Java wird auf mehrere JVM-Befehle aufgeteilt, die wiederum beliebig unterbrochen werden können

Probleme

- Funktioniert lange, aber nicht für immer:
  - Zwischen
    ```
    while (locked) ;
    ```
    und
    ```
    locked = true;
    ```
    können bereits andere Threads die Sperre passiert haben

- Sog. busy waiting sollte auf alle Fälle vermieden werden, da es sehr viel Rechenzeit kostet

Memory Consistency Errors

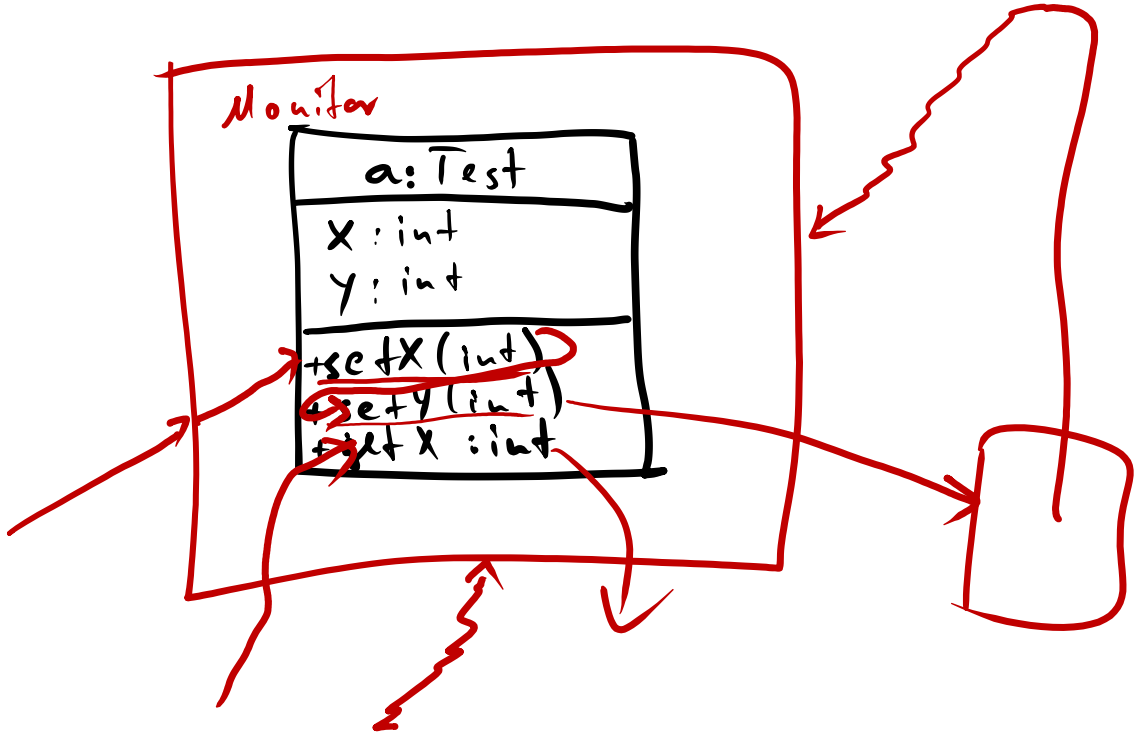
- **Können** auftreten bei unkontrollierten, nicht atomaren Schreibzugriff auf gleichen Speicherbereich
  ➔ Synchronisation

- in Java: Synchronisation über Monitore

Monitore

- Jedes Objekt hat einen Monitor
- In einem Monitor darf sich immer nur genau ein Thread befinden
- Andere Threads, die Zugriff auf diesen Monitor haben möchten, müssen warten

Monitore



Monitore in Java

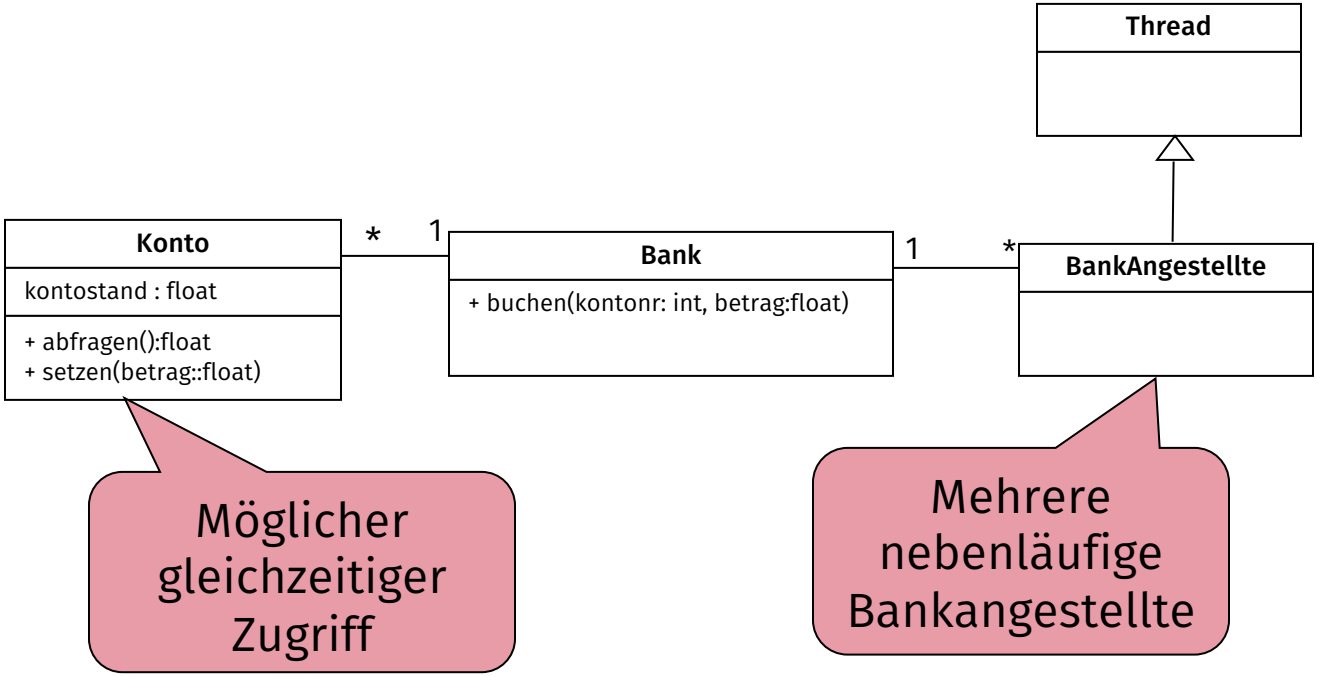
- Synchronisation von Methoden über **`synchronized`** Modifizierer
- jegliche Zugriffe auf eine Instanz werden dann von der VM synchronisiert
- Genau ein Monitor für jedes Objekt
  ➔ wenn Monitor einmal betreten, dann alle anderen synchronized Methoden des ganzen Objekts für alle anderen Zugriffe gesperrt

Eintrittsinvarianz (Reentrancy)

- Was, wenn eine synchronized Methode eine andere synchronized Methode innerhalb des gleichen Objekts aufrufen möchte?

- Kein Problem, da Java-Monitore reentrant

Beispiel Bank 2.0

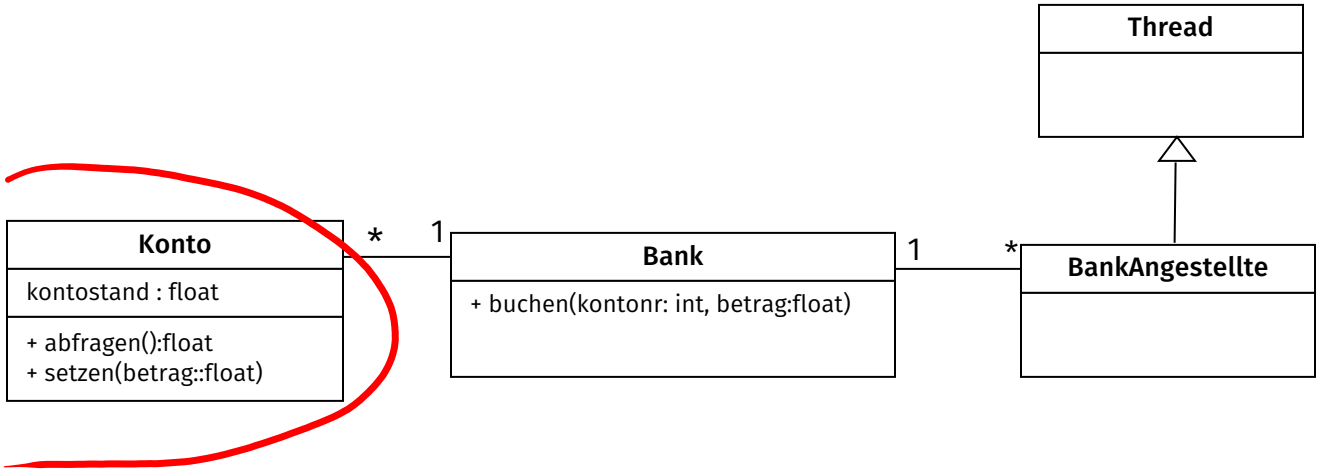


Beispiel Bank 2.0

- Besonderheiten:
  - Thread-Objekt startet selbst seinen Thread im Konstruktor

- Synchronisation erfolgt bei `buchen()`, damit ist komplette Bank bei einer Buchung gesperrt
- Zwei Transaktionen auf zwei verschiedenen Konten können nicht gleichzeitig durchgeführt werden

Beispiel Bank 2.0



Feinere Synchronisation

- Bei jedem Eintritt Blockierung des gesamten Objekts ➔ Verhinderung von Nebenläufigkeit
- Feingranularere Synchronisation wünschenswert
- Synchronisation von Methodenabschnitten über **`synchronized(`*`Object`*`)`**

Feinere Synchronisation

- Methoden-Synchronisierung nur Spezialfall

```
public synchronized void m() {
  …
}
```

ist äquivalent zu

```
public void m() {
  synchronized(this) {
    …
  }
}
```

Vorsichtiger Umgang mit Sperren

- synchronize-Statements sind teuer
  - Sperrmechanismus kostet Zeit
  - Verklemmungsgefahr

- Regel:
  Wenn von mehreren Threads auf ein Objekt zugegriffen wird und mindestens ein Thread den Zustand des Objekts verändert, dann müssen alle Methoden, die schreibend oder lesend auf den Objektzustand zugreifen, synchronisiert werden.

Programmierhinweise

- Synchronisation nur in Methoden, nie auf Attributen
  ➔ Vorsicht mit public-Attributen, besser set- und get-Methoden verwenden
- Ausnahme: volatile
  - Markiert Feld als synchronisiert
  - Normalerweise lokale Kopie für jeden Thread mit asynchronem Abgleich
  - **volatile** erzwingt Abgleich bei jedem Zugriff
  - **Problem: read/write (z.B. lost updates, dirty reads)**
  - Siehe: http://tutorials.jenkov.com/java-concurrency/volatile.html

Programmierhinweise

- Sperren gelten nur für synchronized-Methoden, d.h. non-synchronized-Methoden weiterhin für alle Threads zugreifbar

  ➔ gut überlegen, welche Methoden synchronized sein müssen, ohne paranoid zu werden

Programmierhinweise

- Manchmal gleiche Sperre für Bereiche, die auf unterschiedliche Teile des Objektzustands zugreifen
  ➔ getrennte Sperren für voneinander unabhängige Bereiche

- mehrere (leere) Lock-Objekte zur Trennung der entsprechenden Lock-Bereiche

Bemerkungen zu synchronized

- `synchronized` als Methodenmodifizierer wird nicht vererbt, wenn die Subklasse die entsprechenden Methoden der Superklasse überschreibt
- Interfaces können Methoden nicht als `synchronized` deklarieren
- Konstruktoren können nicht als `synchronized` deklariert werden
  - Aber: Block-Synchronisation kann im Rumpf verwendet werden

Lernziele

- Threads vs. Prozesse
- Arbeiten mit Threads
- Synchronisierung der Zugriffe auf Ressourcen