

Verteilte Systeme – Übung

Grundlagen: Java

Sommersemester 2022

Laura Lawniczak, Tobias Distler

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)

www4.cs.fau.de



Lehrstuhl für Verteilte Systeme
und Betriebssysteme



FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Java

Collections & Maps

Threads

Kritische Abschnitte

Koordinierung

Java

Collections & Maps

- Package: `java.util`
- Gemeinsame Schnittstelle: `Collection`
- Datenstrukturen
 - Menge
 - Schnittstelle: `Set`
 - Implementierungen: `HashSet`, `TreeSet`, ...
 - Liste
 - Schnittstelle: `List`
 - Implementierungen: `LinkedList`, `ArrayList`, ...
 - Warteschlange
 - Schnittstelle: `Queue`
 - Implementierungen: `PriorityQueue`, `LinkedBlockingQueue`, ...
- Tutorial



The Java Tutorials, Trail: Collections

<http://docs.oracle.com/javase/tutorial/collections/index.html>

- Verfügbare Algorithmen (Beispiele)
 - Maximums- (`max()`) bzw. Minimumsbestimmung (`min()`)
 - Sortieren (`sort()`)
 - Überprüfung auf Existenz gemeinsamer Elemente (`disjoint()`)
 - Erzeugung zufälliger Permutationen (`shuffle()`)
- Beispiel
 - Implementierung

```
Integer[] values = { 1, 2, 3, 4, 5, 6 };  
  
List<Integer> list = new ArrayList<Integer>(values.length);  
Collections.addAll(list, values);  
  
System.out.println("Before: " + list);  
Collections.shuffle(list);  
System.out.println("After: " + list);
```

- Ausgabe eines Testlaufs

```
Before: [1, 2, 3, 4, 5, 6]  
After:  [4, 2, 1, 6, 5, 3]
```

- Allgemeine Schnittstelle für Datenstrukturen zur Verwaltung von Schlüssel-Wert-Paaren
- Eigenschaften
 - Maximal ein Wert pro Schlüssel (→ keine Duplikate)
 - Interner Aufbau bestimmt durch gewählte Implementierung
 - HashMap
 - TreeMap
 - ...
- Beispiel

```
Map<String, Integer> telBook = new HashMap<String, Integer>();  
telBook.put("Alice", 123456789);  
telBook.put("Bob" , 987654321);  
[...]
```

```
Integer aliceNumber = telBook.get("Alice");  
System.out.println("Alice's number: " + aliceNumber);
```

Java

Threads

Variante 1: Unterklasse von `java.lang.Thread`

■ Vorgehensweise

1. Unterklasse von `Thread` erstellen
2. `run()`-Methode überschreiben
3. Instanz der neuen Klasse erzeugen
4. An dieser Instanz die `start()`-Methode aufrufen

■ Beispiel

```
class VSThreadTest extends Thread {  
    @Override  
    public void run() {  
        System.out.println("Test");  
    }  
}
```

```
Thread test = new VSThreadTest();  
test.start();
```


Variante 2: Implementieren von `java.lang.Runnable`

■ Vorgehensweise

1. `run()`-Methode der `Runnable`-Schnittstelle implementieren
2. `Runnable`-Objekt erstellen
3. Instanz von `Thread` mit Hilfe des `Runnable`-Objekts erzeugen
4. Am neuen `Thread`-Objekt die `start()`-Methode aufrufen

■ Beispiel

```
class VSRunnableTest implements Runnable {  
    @Override  
    public void run() {  
        System.out.println("Test");  
    }  
}
```

```
Runnable test = new VSRunnableTest();  
Thread thread = new Thread(test);  
thread.start();
```

Variante 3: Java Lambda Ausdrücke [seit Java 8]

■ Vorgehensweise:

1. Erzeugung von Thread-Instanz und Beschreibung der `run()`-Methode mittels Lambda
2. Am neuen Thread-Objekt die `start()`-Methode aufrufen

■ **Einschränkung:** Kein Zustand (z.B. globale Variablen) möglich

■ Beispiel

```
class VSLambdaTest {
    private int x = 10;

    public void lambdaTest() {
        Thread test = new Thread(() -> {
            System.out.println("Test " + this.x);
        });
        test.start();
    }
}
```

- Ausführung für einen bestimmten Zeitraum aussetzen

- Mittels sleep()-Methoden

```
static void sleep(long millis) throws InterruptedException;
```

```
static void sleep(long millis, int nanos) throws InterruptedException;
```

- Legt aktuellen Thread für millis Millisekunden (und nanos Nanosekunden) „schlafen“

- **Achtung:**

- Es ist nicht garantiert, dass der Thread exakt nach der angegebenen Zeit seine Ausführung fortsetzt
- Von Präzision der Systemzeit/des Schedulers abhängig (Linux: 1ms, Windows (default): 15ms)

- Synchronisierung mit anderen Threads (siehe Kapitel „Koordinierung“ ab Folie 17)

■ Regulär

- return aus der run()-Methode
- Ende der run()-Methode

■ Abbruch nach expliziter Anweisung

- Aufruf der interrupt()-Methode (durch einen anderen Thread)

```
public void interrupt();
```

- Führt zu

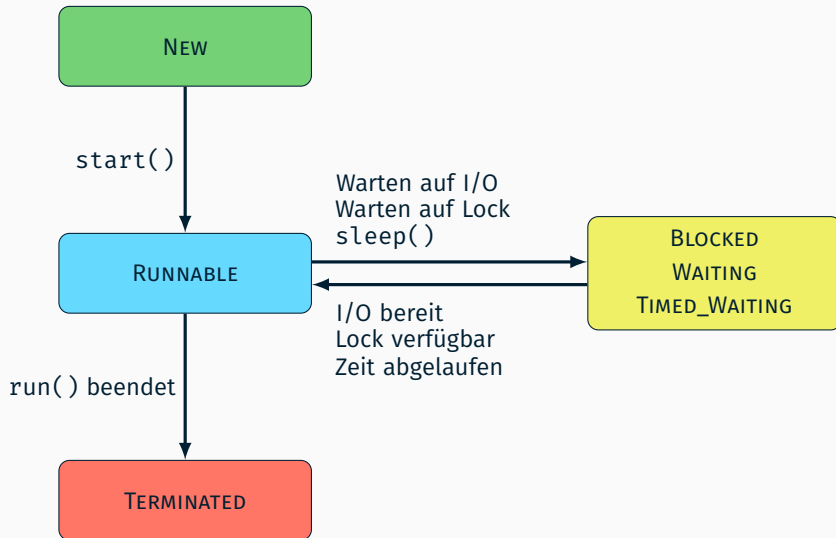
- einer InterruptedException, falls sich der Thread gerade in einer unterbrechbaren blockierenden Operation befindet
- einer ClosedByInterruptException, falls sich der Thread gerade in einer unterbrechbaren I/O-Operation befindet
- dem Setzen einer Interrupt-Status-Variable, die mit isInterrupted() abgefragt werden kann, sonst.

Wichtig: Threads können sich in Java aktiv der Unterbrechung *widersetzen* (z.B. Fangen & Ignorieren von InterruptedExceptions). Man kann sie von außerhalb also nicht zum Beenden *zwingen*.

■ Auf die Terminierung eines Threads warten mittels join()-Methode

```
public void join() throws InterruptedException;
```

Thread-Zustände in Java



Java

Kritische Abschnitte

```
public class VSCounter implements Runnable {
    public int a = 0;

    public void run() {
        for(int i = 0; i < 1000000; i++) {
            a = a + 1;
        }
    }

    public static void main(String[] args) throws Exception {
        VSCounter value = new VSCounter();
        Thread t1 = new Thread(value);
        Thread t2 = new Thread(value);

        t1.start();
        t2.start();

        t1.join();
        t2.join();
        System.out.println("Expected a = 2000000, " +
            "but a = " + value.a);
    }
}
```

- Ergebnisse einiger Durchläufe: 1 732 744, 1 378 075, 1 506 836
- Was passiert, wenn $a = a + 1$ ausgeführt wird?

```
LOAD a into Register
ADD 1 to Register
STORE Register into a
```

- Mögliche Verzahnung wenn zwei Threads T_1 und T_2 beteiligt sind

0. $a = 0$;

1. T_1 -LOAD: $a = 0$, $Reg_1 = 0$

2. T_2 -LOAD: $a = 0$, $Reg_2 = 0$

3. T_1 -ADD: $a = 0$, $Reg_1 = 1$

4. T_1 -STORE: $a = 1$, $Reg_1 = 1$

5. T_2 -ADD: $a = 1$, $Reg_2 = 1$

6. T_2 -STORE: $a = 1$, $Reg_2 = 1$

⇒ Die drei Operationen müssen jeweils **atomar** ausgeführt werden!

Synchronisieren ist notwendig, falls Atomizität erforderlich

1. Der Aufruf einer (komplexen) Methode muss atomar erfolgen

- Eine Methode enthält mehrere Operationen, die auf einem konsistenten Zustand arbeiten müssen
- Beispiele:
 - „a = a + 1“
 - Listen-Operationen (add(), remove(),...)

2. Zusammenhängende Methodenaufrufe müssen atomar erfolgen

- Methodenfolge muss auf einem konsistenten Zustand arbeiten
- Beispiel:

```
List list = new LinkedList();  
[...]  
int lastObjectIndex = list.size() - 1;  
Object lastObject = list.get(lastObjectIndex);
```

■ Standardansatz in Java

- Kennzeichnung eines kritischen Abschnitts mittels `synchronized`-Block
- Verknüpfung eines kritischen Abschnitts mit einem *Sperrobjekt*
- Ein Sperrobjekt kann nur von jeweils einem Thread gehalten werden

```
public void foo() {  
    [...] // unkritische Operationen  
    synchronized(<Sperrobjekt>) {  
        [...] // kritischer Abschnitt  
    }  
    [...] // unkritische Operationen  
}
```

■ Hinweise

- Jedes `java.lang.Object` kann als Sperrobjekt dienen
- Ein Thread kann dasselbe Sperrobjekt mehrfach halten (rekursive Sperre)

■ Mögliche Lösung für das Zähler-Beispiel

```
synchronized(this) { a = a + 1; }
```

■ Alternativen: Semaphore, ReentrantLock

- Ersatzschreibweise für einen methodenweiten `synchronized`-Block
- Sperrobjekt
 - Statische Methoden: `Class`-Objekt der entsprechenden Klasse
 - Sonst: `this`

```
class VExample {  
    synchronized public void foo() {  
        [...] // kritischer Abschnitt  
    }  
    public void bar() {  
        synchronized(this) {  
            [...] // kritischer Abschnitt  
        }  
    }  
}
```

- Beachte
 - Alle `synchronized`-Methoden einer Klasse nutzen dasselbe Sperrobjekt
 - Ansatz nur sinnvoll, falls Methoden tatsächlich in Konflikt stehen

- Klasse `java.util.Collections`
 - Statische Wrapper-Methoden für `Collection`-Objekte
 - Synchronisation kompletter Datenstrukturen

- Methoden

```
static <T> List<T> synchronizedList(List<T> list);  
static <K,V> Map<K,V> synchronizedMap(Map<K,V> map);  
static <T> Set<T> synchronizedSet(Set<T> set);  
[...]
```

- Beispiel

```
List<String> list = new LinkedList<String>();  
List<String> syncList = Collections.synchronizedList(list);
```

- Beachte

- Synchronisiert **alle** Zugriffe auf eine Datenstruktur
- Kein Schutz von zusammenhängenden Methodenaufrufen

■ Ansatz

- Ersatz-Klassen für problematische Datentypen
- Atomare Varianten häufig verwendeter Operationen
- Operation für atomares *Compare-and-Swap* (CAS)

■ Verfügbare Klassen

- Versionen für primitive Datentypen: `Atomic{Boolean,Integer,Long}`
- Arrays: `AtomicIntegerArray`, `AtomicLongArray`
- Referenzen: `AtomicReference`, `AtomicReferenceArray`
- ...

■ Beispiel

```
AtomicInteger ai = new AtomicInteger(47);
int newValueA = ai.incrementAndGet();
int newValueB = ai.getAndIncrement();
int oldValue = ai.getAndSet(4);
boolean success = ai.compareAndSet(oldValue, 7);
```

Java

Koordinierung

■ Problemstellung

- Rollenverteilung zwischen Threads (z. B. Produzent/Konsument)
 - Threads müssen sich abstimmen, um eine gemeinsame Aufgabe zu lösen
- Mechanismen zur Koordinierung erforderlich

■ Standardansatz in Java

- Ein Thread wartet darauf, dass ein Ereignis eintritt
- Der Thread wird mittels einer *Synchronisationsvariable* benachrichtigt

■ Hinweise

- Jedes `java.lang.Object` kann als Synchronisationsvariable dienen
- Um andere Threads per Synchronisationsvariable zu benachrichtigen, muss ein Thread innerhalb eines `synchronized`-Blocks dieser Variable sein

■ Methoden

`wait()` Auf eine Benachrichtigung warten

`notify()` Benachrichtigung an **einen** wartenden Thread senden

`notifyAll()` Benachrichtigung an **alle** wartenden Threads senden

■ Variablen

```
Object syncObject = new Object(); // Synchronisationsvariable
boolean flag = false;             // Ereignis-Flag
```

■ Auf Erfüllung der Bedingung wartender Thread

```
synchronized(syncObject) {
    while(!flag) {
        syncObject.wait();
    }
}
```

■ Bedingung erfüllender Thread

```
synchronized(syncObject) {
    flag = true;
    syncObject.notify();
}
```