

D Überblick über die 3. Übung

D Überblick über die 3. Übung

- Threads - Überblick
- Synchronisation
- Probleme
- optimistische Nebenläufigkeitskontrolle
- einige neue Interfaces in Java 5

D.1 Threads

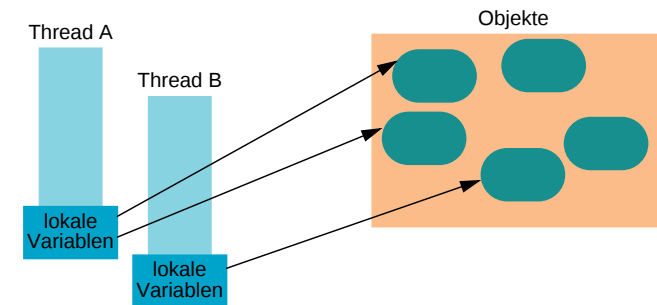
D.1 Threads

- Referenz:
 - ◆ D. Lea. *Concurrent Programming in Java - Design Principles and Patterns*. The Java Series. Addison-Wesley 1997.

1 Was ist ein Thread?

D.1 Threads

- Aktivitätsträger mit:
 - eigenem Instruktionszähler
 - eigenen Registern
 - eigenem Stack
- Alle Threads laufen im gleichen Adressbereich



2 Vorteile / Nachteile

D.1 Threads

- Vorteile:
 - ◆ Ausführen paralleler Algorithmen auf einem Multiprozessorrechner
 - ◆ durch das Warten auf langsame Geräte (z.B. Netzwerk, Benutzer) wird nicht das gesamte Programm blockiert
- Nachteile:
 - ◆ komplexe Semantik
 - ◆ Fehlersuche sehr schwierig
 - ◆ John Ousterhout: *Why Threads Are A Bad Idea (for most purposes)*.

3 Thread Erzeugung: Möglichkeit 1

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

■ Beispiel:

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

Test test = new Test();
test.start();
```

4 Die Methode sleep

- Ein Thread hat die Methode `sleep(long n)` um für `n` Millisekunden zu "schlafen".
- Der Thread kann jedoch verdrängt worden sein nachdem er aus dem `sleep()` zurückkehrt.

3 Thread Erzeugung: Möglichkeit 2

1. Das Interface `java.lang.Runnable` implementieren. Dabei muss eine `run()`-Methode erstellt werden.
2. Ein Objekt instantiiieren, welches das Interface `Runnable` implementiert.
3. Eine neue Instanz von `Thread` erzeugen, dem Konstruktor dabei das `Runnable`-Objekt mitgeben.
4. Am neuen Thread-Objekt die `start()`-Methode aufrufen.

■ Beispiel:

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

Test test = new Test();
Thread thread = new Thread(test);
thread.start();
```

5 Die Methode join

- Ein Thread kann auf die Beendigung eines anderen Threads warten:

```
workerThread = new Thread(worker);
...
workerThread.join();
worker.result();
```

6 Daemon-Threads

- Daemon-Threads werden für Hintergrundaktivitäten genutzt
- Sie sollen nicht für die Hauptaufgabe eines Programmes verwendet werden
- Sobald alle *nicht-daemon* Threads beendet sind, ist auch das Programm beendet.
- Woran erkennt man, ob ein Thread ein Daemon-Thread sein soll?
 - ◆ Wenn man keine Bedingung für die Beendigung des Threads angeben kann.
- Wichtige Methoden der Klasse **Thread**:
 - ◆ **setDaemon(boolean switch)**: Ein- oder Ausschalten der Daemon-Eigenschaft
 - ◆ **boolean isDaemon()**: Prüft ob ein Thread ein Daemon ist.

7 Die Klasse ThreadGroup

- Gruppe von verwandten Threads (**ThreadGroup**):
 - ◆ Eine Threadgruppe kann Threads enthalten und andere Threadgruppen.
 - ◆ Ein Thread kann nur Threads in der eigenen Gruppe beeinflussen.
- Methoden, die nur auf Threads der gleichen Gruppe angewendet werden können:
 - ◆ **list()**
 - ◆ **stop()**
 - ◆ **suspend()**
 - ◆ **resume()**

8 Zustände von Threads

```

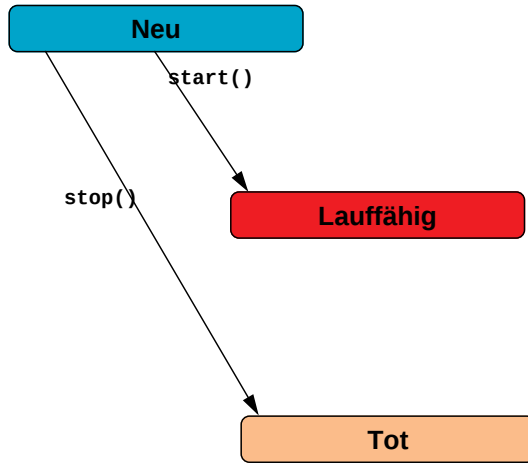
graph TD
    A[Neu]
  
```

8 Zustände von Threads (2)

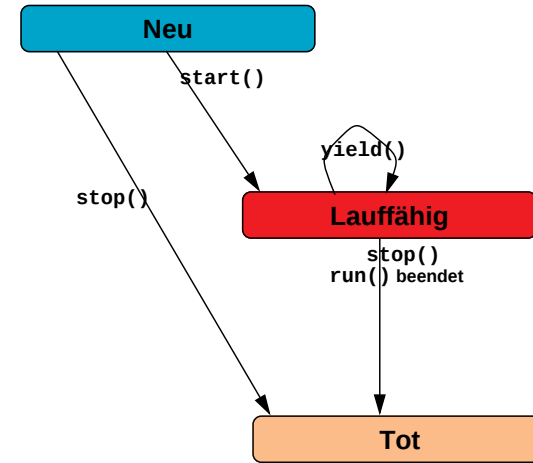
```

graph TD
    A[Neu] -- stop() --> B[Tot]
  
```

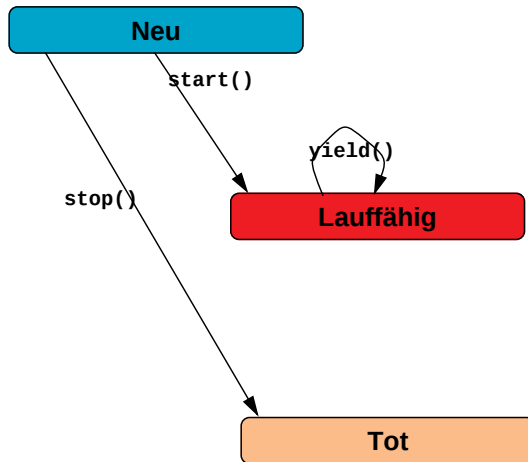
8 Zustände von Threads (3)



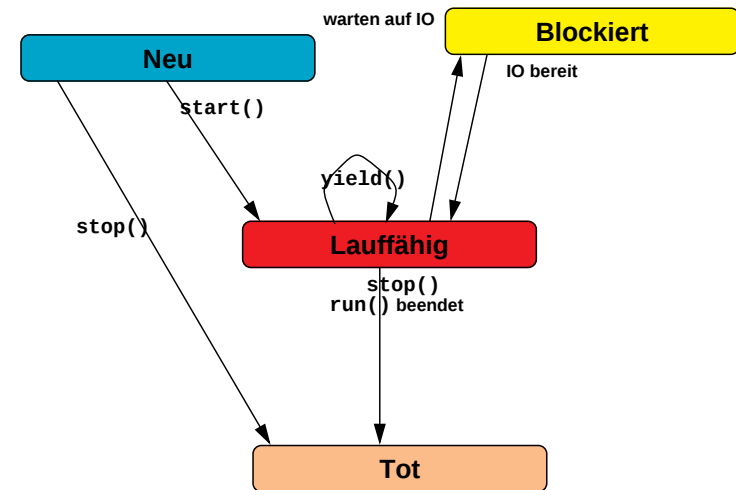
8 Zustände von Threads (5)



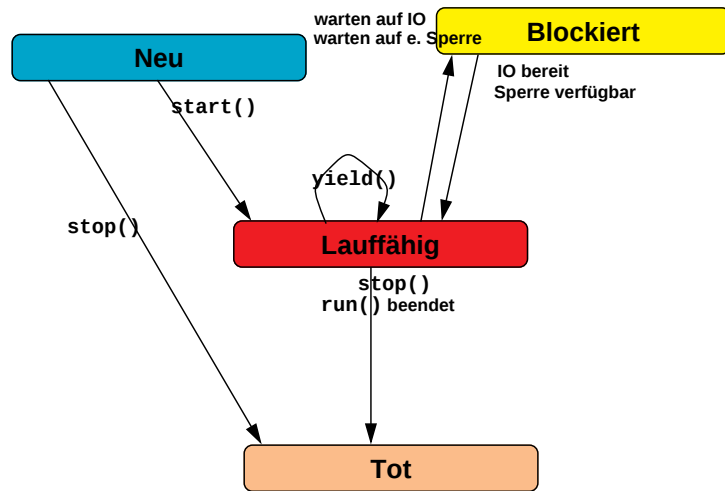
8 Zustände von Threads (4)



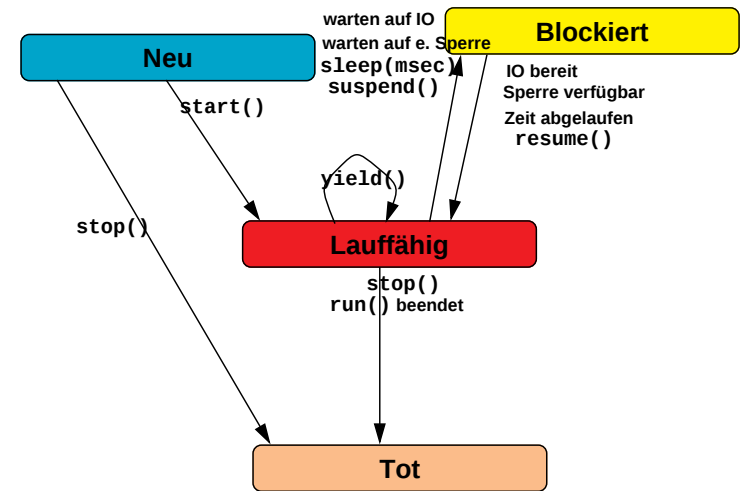
8 Zustände von Threads (6)



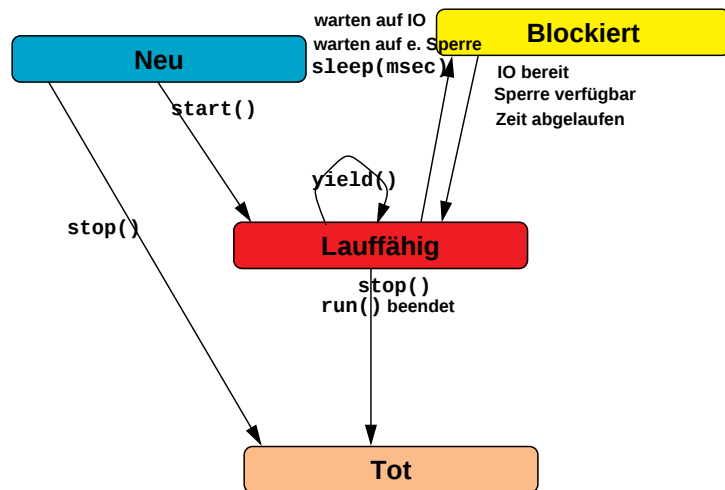
8 Zustände von Threads (7)



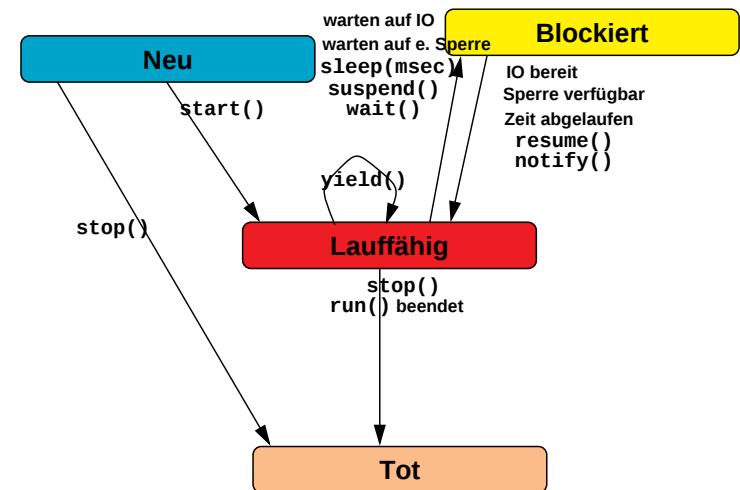
8 Zustände von Threads (9)



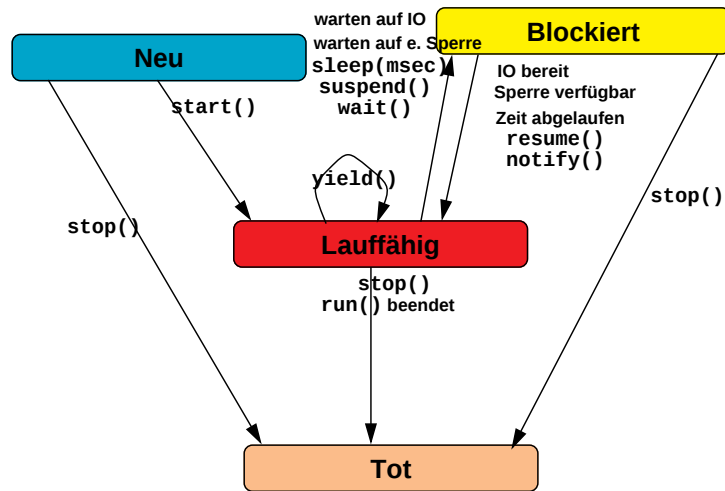
8 Zustände von Threads (8)



8 Zustände von Threads (10)



8 Zustände von Threads (11)



9 Veraltete Methoden der Klasse Thread

- **stop()**, **suspend()**, **resume()** sind seit Java 1.2 unerwünscht.
- **stop()** gibt alle Sperren des Thread frei - kann zu Inkonsistenzen führen
- **suspend()** und **resume()** können zu einem Deadlock führen:
 - ◆ suspend gibt keine Sperren frei
 - ◆ angehaltener Thread kann Sperren halten
 - ◆ Thread, der **resume** aufrufen will blockiert an einer Sperre

10 Multithreading Probleme

```

public class Test implements Runnable {
    public int a=0;
    public void run() {
        for(int i=0; i<100000; i++) {
            a = a + 1;
        }
    }

    public static void main(String[] args) {
        Test value = new Test();
        Thread t1 = new Thread(value);
        Thread t2 = new Thread(value);
        t1.start();
        t2.start();
        try {
            t1.join();
            t2.join();
        } catch (Exception e) {
            System.out.println("Exception");
        }
        System.out.println("Expected a=200000 but a="+value.a);
    }
}
  
```

Was ist das Ergebnis dieses Programmes?

zwei Threads erzeugen, beide Threads starten

auf Beendigung der beiden Threads warten

10 Multithreading Probleme (2)

- Ergebnis einiger Durchläufe: 173274, 137807, 150683
- 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 beteiligt sind (initial a=0):
 - ◆ T1-load:a=0,Reg1=0
 - ◆ T2-load:a=0,Reg2=0
 - ◆ T1-add:a=0,Reg1=1
 - ◆ T1-store:a=1,Reg1=1
 - ◆ T2-add:a=1,Reg2=1
 - ◆ T2-store:a=1,Reg2=1
- Die drei Operationen müssen *atomar* ausgeführt werden!

11 Das Schlüsselwort synchronized

- Jedes Objekt kann als Sperre verwendet werden.
- Um eine Sperren anzufordern und freizugeben wird ein **synchronized** Konstrukt verwendet.
- Methoden oder Blöcke können als **synchronized** deklariert werden:

```
class Test {
    public synchronized void m() { ... }
    public void n() { ...
        synchronized(this) {
            ...
        }
    }
}
```

- ein Thread kann eine Sperre mehrfach halten (rekursive Sperre)
- verbessertes Beispiel: **synchronized(this) { a = a + 1; }**

12 Wann soll synchronized verwendet werden?

- **synchronized** ist nicht notwendig:
 - ◆ wenn Code immer nur von einem Thread ausgeführt wird (single-threaded context)
 - ◆ für einfache get-Methoden (siehe Ausnahmen unten)
- **synchronized** sollte verwendet werden:
 - ◆ wenn Daten geschrieben werden
 - ◆ wenn mit dem Objekt Berechnungen durchgeführt werden (auch wenn der Zustand *nur gelesen* wird)
 - ◆ für get-Methoden, die **long** oder **double** Typen zurückliefern
 - ◆ für einfache get-Methoden, die blockieren sollen wenn eine Zustandsveränderung durchgeführt wird

13 Synchronisationsvariablen (Condition Variables)

- Thread muss warten bis eine Bedingung wahr wird.
- zwei Möglichkeiten:
 - ◆ aktiv (polling)
 - ◆ passiv (condition variables)
- Jedes Objekt kann als Synchronisationsvariable verwendet werden.

- Die Klasse **Object** enthält Methoden um ein Objekt als Synchronisationsvariable zu verwenden.

- ◆ **wait**: auf ein Ereignis warten

```
while(! condition) { wait(); }
```

- ◆ **notify**: Zustand wurde verändert, die Bedingung könnte wahr sein, einen anderen Thread benachrichtigen
- ◆ **notifyAll**: alle wartenden Threads aufwecken (teuer)

14 Warten und Sperren

- **wait** kann nur ausgeführt werden, wenn der aufrufende Thread eine Sperre an dem Objekt hält.
- **wait** gibt die Sperre frei bevor der Thread blockiert wird (atomar)
- beim Deblockieren wird die Sperre wieder atomar angefordert

15 Condition Variables - Beispiel

- PV-System: Bedingung: *count > 0*

```
class Semaphore {
    private int count;
    public Semaphore(int count) { this.count = count; }
    public synchronized void P() throws InterruptedException{
        while (count <= 0) {
            wait();
        }
        count--;
    }
    public synchronized void V() {
        count++;
        notify();
    }
}
```

15 Condition Variables - Beispiel (3)

- Worker:

```
class SpecificWorker implements Runnable, WorkerThread {
    public void run() {
        for(;;) {
            while(queue.empty()) // race condition!!
                synchronized (this) { wait(); }
            Customer customer = queue.next();
            // do something nice with customer
            // ...
        }
    }
    public void synchronized insertCustomer(Customer c) {
        queue.insert(c);
        this.notify();
    }
}
```

15 Condition Variables - Beispiel (2)

- Bestellsystem: ein Thread akzeptiert Kundenabfragen (**SecretaryThread**) ein anderer Thread bearbeitet sie (**WorkerThread**)
- Secretary:

```
class SecretaryThread implements Runnable {
    public void run() {
        for(;;) {
            Customer customer = customerLine.nextCustomer();
            WorkerThread worker = classify(customer);
            worker.insertCustomer(customer);
        }
    }
}

interface WorkerThread {
    public void insertCustomer(Customer c);
}
```

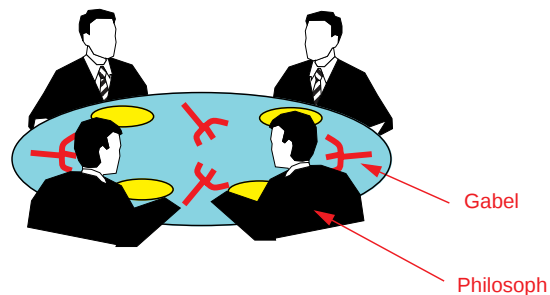
D.2 Korrektheit nebenläufiger Programme

- Safety: "Es passiert niemals etwas Schlechtes"
- Liveness: "Es passiert überhaupt etwas"

1 Safety

- gegenseitiger Ausschluss durch **synchronized**
- optimistische Nebenläufigkeitskontrolle

2 Deadlock: Das Philosophenproblem



denken → essen → denken → essen → ...

- ein Philosoph braucht beide Gabeln zum Essen
- alle Philosophen nehmen zuerst die rechte Gabel dann die linke → Verklemmung

3 Liveness

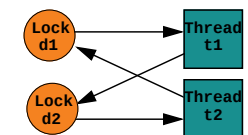
- keine Sprachunterstützung zur Deadlock-Verhinderung/Erkennung
- Deadlock-Beispiel:

```
class Deadlock implements Runnable {
    Deadlock other;
    void setOther(Deadlock other) { this.other = other; }
    synchronized void m() {
        try { Thread.sleep(1000); }
        catch (InterruptedException e) {}
        other.m();
    }
    public void run() { m(); }
}
```

3 Liveness (2)

- Verwendung, die zum Deadlock führt:

```
Deadlock d1 = new Deadlock();
Deadlock d2 = new Deadlock();
d1.setOther(d2); d2.setOther(d1);
Thread t1 = new Thread(d1);
t1.start();
Thread t2 = new Thread(d2);
t2.start();
try { t1.join(); t2.join(); } catch (InterruptedException e) {}
```



4 Deadlocks

■ Counter

```
class Counter {
    private int count = 0;

    public synchronized void inc() { count++; }

    public int getCount() {return count; }

    public void setCount(int count) { this.count = count; }

    public synchronized void swap(Counter counter) {
        synchronized (counter) { // Deadlock Gefahr
            int tmp = counter.getCount();
            counter.setCount(count);
            count = tmp;
        }
    }
}
```

5 Deadlock Vermeidung (2)

■ Ressourcen (Locks) werden atomar angefordert:

```
class Counter {
    ...
    static Object lock = new Object();
    public void swap(Counter counter) {
        synchronized (lock) {
            synchronized (this) {
                synchronized (counter) {
                    int tmp = counter.getCount();
                    counter.setCount(count);
                    count = tmp;
                }
            }
        }
    }
}
```

5 Deadlock - Vermeidung (1)

■ Verhinderung zyklischer Ressourcenanforderung, Ordnung auf Locks

```
class Counter {
    ...
    public void swap(Counter counter) {
        Counter first = this;
        Counter second = counter;
        if (System.identityHashCode(this)
            < System.identityHashCode(counter)) {
            first = counter;
            second = this;
        }
        synchronized (first) {
            synchronized (second) {
                int tmp = counter.getCount();
                counter.setCount(count);
                count = tmp;
            }
        }
    }
}
```

6 Nachteile der Java-Locks

- Methoden (lock, unlock) von Java-Locks sind unsichtbar, nur mit synchronized beeinflussbar
- kein Timeout beim Warten auf ein Lock möglich (Deadlock-Erkennung)
- Es können keine Unterklassen von Locks mit speziellem Verhalten erzeugt werden (Authentifizierung, Queueing-Strategien, ...)
- Locks können nicht referenziert werden (keine Deadlock-Erkennung oder Recovery möglich)

7 Optimistische Nebenläufigkeitskontrolle

- Vorteile:
 - ◆ keine Deadlocks
 - ◆ höhere Parallelität möglich
- Nachteile:
 - ◆ Designs werden komplexer
 - ◆ ungeeignet bei hoher Last
- Rollback/Recovery
 - ◆ Aktionen müssen umkehrbar sein, keine Seiteneffekte
 - ◆ zu jeder Methode muss es eine "Antimethode" geben
- Versioning
 - ◆ Methoden arbeiten auf shadow-Kopien des Objektzustandes
 - ◆ atomares commit überprüft, ob sich Ausgangszustand geändert hat (Konflikt) und setzt shadow-Zustand als neuen Objektzustand

7 Optimistisch - Beispiel (1)

- Counterzustand (Instanzvariablen) ist in separates Objekt ausgelagert (Memento Design-Pattern)

```
class CounterState {
    int count;
    CounterState(CounterState state) { ... }
    void inc() { ... }
    void dec() { ... }
    void swap(CounterState counter) { ... }
}
```

7 Optimistisch - Beispiel (2)

- Counter führt alle Operationen auf Kopie des Zustands aus
- am Ende wird die Kopie in einer atomaren Operation als Ist-Zustand gesetzt

```
class Counter {
    CounterState state;
    synchronized boolean commit(CounterState assumed,
                                CounterState next) {
        if (state != assumed) return false;
        state = next;
        return true;
    }
    void inc {
        do {
            assumed = state;
            next = new CounterState(assumed);
            next.inc();
        } while ( ! commit(assumed, next));
    }
}
```

D.3 Threads/Nebenläufigkeit und Java 5

1 java.util.concurrent.atomic

- atomare Operationen auf Variablen
- z.B.: **AtomicInteger**

```
AtomicInteger ai = new AtomicInteger(42);
ai.addAndGet(12);
```

- Versionen für alle primitiven Datentypen vorhanden
- auch für Arrays verfügbar
- **AtomicReference** für Referenzen

2 Lock - das Interface

- exklusive Sperre definiert im Paket: `java.util.concurrent.locks`
- Allgemeines Interface: **Lock**
- Sperre anfordern
 - ◆ `void lock()`
 - ◆ `void lockInterruptibly()` throws `InterruptedException`
 - ◆ `boolean tryLock()`
 - ◆ `boolean tryLock(long time, TimeUnit unit)` throws `InterruptedException`
- Sperre freigeben:
 - ◆ `void unlock()`
- "Condition"-variable für diese Sperre erzeugen
 - ◆ `Condition newCondition()`

2 Lock - Implementierungen

- **ReentrantLock** - eine Implementierungen von **Lock** enthält weitere sinnvolle Methoden:
- Wer hält die Sperre?
 - ◆ `Thread getOwner()`
- Wer wartet auf die Sperre?
 - ◆ `Collection<Thread> getQueuedThreads()`
- Wer wartet auf eine Bedingung?
 - ◆ `Collection<Thread> getWaitingThreads(Condition c)`

2 Condition

- Auf Signal warten
 - ◆ `void await()` throws `InterruptedException`
 - ◆ `void awaitUninterruptibly()`
 - ◆ `boolean await(long time, TimeUnit unit)` throws `InterruptedException`
 - ◆ `long awaitNanos(long nanosTimeout)` throws `InterruptedException`
 - ◆ `boolean awaitUntil(Date deadline)` throws `InterruptedException`
- Signalisieren
 - ◆ `void signal()`
 - ◆ `void signalAll()`

2 Threads / Executors

- **Runnable** mit Rückgabe und Exception: **Callable<V>**
 - ◆ `V call()` throws `Exception`

- Interface **Executor**:

```
// bisher:
new Thread(new RunnableTask()).start()

// Java 5:
Executor executor = anExecutor;
executor.execute(new RunnableTask1());
executor.execute(new RunnableTask2());
```

- Thread Erzeugung mittels **ThreadFactory**
 - ◆ `Thread newThread(Runnable r)`

2 Ein Executor zur asynchronen Methodenausführung

- Ausführung durch einen **ExecutorService**:

- ◆ `<T> Future<T> submit(Callable<T> task)`
- ◆ `Future<?> submit(Runnable task)`

- Ergebnis als Platzhalter: **Future<V>**

- ◆ `boolean cancel(boolean mayInterruptIfRunning)`
- ◆ `V get()`
- ◆ `V get(long timeout, TimeUnit unit)`
- ◆ `boolean isDone()`

2 Die Executor Fabrik Executors

- Klasse **Executors** mit statischen Methoden:

- ◆ `static ThreadFactory defaultThreadFactory()`
- ◆ `static ExecutorService newSingleThreadExecutor()`
- ◆ `static ExecutorService newFixedThreadPool(int nThreads)`
- ◆ `static ExecutorService newCachedThreadPool()`
- ◆ ...