

Multithreading



Multithreading

- Motivation und Grundlagen
- Threadzustände und Prioritäten
- Synchronisation
- Weiteres



Überblick über Klassen

- Thread:
 - Thread-Objekte repräsentieren einen Thread
 - definiert Methoden für starten, unterbrechen, ...
 - definiert statische-Methoden, um
 - aktuell laufenden Thread zu steuern, z.B. ihn schlafen zu legen
 - Verwaltung aller aktuell existierenden Threads
- Runnable:
 - Interface Runnable definiert Methode run(), welche den von einem Thread auszuführenden Code beinhaltet
 - Thread implementiert Runnable
 - Thread kann aber auch mit einem Runnable-Objekt erzeugt werden
- Object:
 - in Klasse Object ist ein Monitor implementiert, d.h. jedes Objekt in Java kann zur Thread-Synchronisation verwendet werden
 - wesentlichen Methoden sind wait und notify, bzw. notifyAll
- ThreadGroup: Für das Bilden von Gruppen von Threads
- InterruptedException: Exception geworfen bei Unterbrechung



Erzeugen, Starten, Ablauf eines Threads (1)

Variante: Ableiten von Thread

- Thread wird abgeleitet (z.B. BallThread) und run() von Runnable überschrieben
- Thread-Objekt wird mit new erzeugt
- Thread wird mit start() gestartet und damit run() ausgeführt

- Der Thread läuft bis zum Ende der Methode run() und stirbt dann
- Die static-Methode sleep(long millis) erlaubt es, den aktuellen Thread für eine gegebene Zeit „Schlafen zu legen“

Achtung:

sleep wirft InterruptedException und muss daher mit try/catch-Anweisung geklammert werden

Beispiel: BallThread

```
class BounceFrame extends JFrame {
    public void addBall() {
        Ball b = new Ball(canvas);
        canvas.add(b);
        BallThread thread = new BallThread(b);
        thread.start();
    }
    ...
}

class BallThread extends Thread {
    public BallThread(Ball aBall) {
        b = aBall;
    }

    public void run() {
        try {
            for (int i = 1; i <= 1000; i++) {
                b.move();
                Thread.sleep(5);
            }
        } catch (InterruptedException e) { ... }
    }

    private Ball b;
}
```



Erzeugen, Starten, Ablauf eines Threads (2)

Variante: Eigenes Runnable e-Objekt

- Die Methode run() wird in einem eigenen Runnable e-Objekt implementiert (im Bsp. Ball)
- Thread-Objekt wird mit new erzeugt, wobei das Runnable e-Objekt als Parameter übergeben wird
- Thread wird mit start() gestartet und damit run() des Runnable e-Objekts ausgeführt

Beispiel: BallThread

```
class BounceFrame extends JFrame {
    public void addBall() {
        Ball b = new Ball(canvas);
        canvas.add(b);
        Thread thread = new Thread(b);
        thread.start();
    }
    ...
}

class Ball implements Runnable {
    ...

    public void run() {
        try {
            for (int i = 1; i <= 1000; i++) {
                move();
                Thread.sleep(5);
            }
        } catch (InterruptedException e) { ... }
    }
    ...
}
```



Multithreading

- Motivation und Grundlagen
- Threadzustände und Prioritäten
- Synchronisation
- Weiteres



Thread-Zustände

neu: wurde gerade erzeugt und noch nicht gestartet

lauffähig:

aktiv: wird gerade ausgeführt

bereit: kann ausgeführt werden und wartet auf Zuteilung des Prozessors

blockiert:

schlafend: wurde mit sleep() gelegt

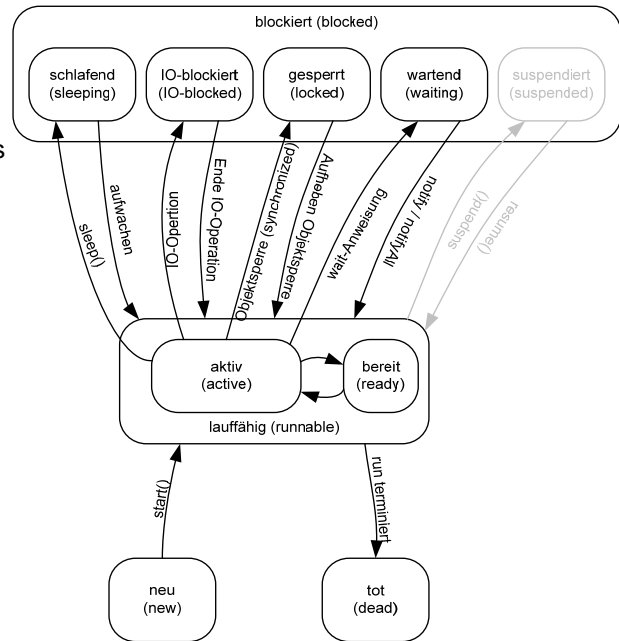
IO-blockiert: wartet auf Beendigung einer IO-Operation

wartend: wurde mit wait() in den wartenden Zustand versetzt

gesperrt: wartet auf die Aufhebung einer Objekt-Sperre

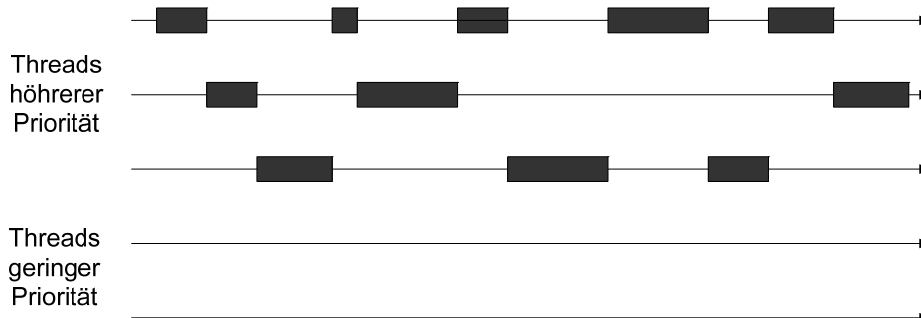
suspendiert: durch suspend() vorübergehend blockiert
Achtung: ist veraltet und sollte nicht verwendet werden

tot: run()-Methode hat terminiert



Scheduling und Prioritäten

- Die lauffähigen Threads müssen sich den Prozessor zur Ausführung teilen; sie konkurrieren um die Zuteilung des Prozessors
- Java legt keine Zuteilungsstrategie fest; diese ist abhängig vom Laufzeitsystem



Scheduling und Prioritäten (2)

- Mit der Methode

void setPriority(int priority)

kann man einem Thread eine Priorität für die Zuteilung geben

- Prioritätswerte liegen zwischen `MIN_PRIORITY = 0` und `MAX_PRIORITY = 10` mit `NORM_PRIORITY = 5` als Standardwert

Beispiel: BounceExpress (siehe Bsp.: v2. v2c1. BounceExpress in CoreJava)

```
public void addBall(int priority, Color color)
{
    Ball b = new Ball(canvas, color);
    canvas.add(b);
    BallThread thread = new BallThread(b);
    thread.setPriority(priority);
    thread.start();
}
```

Ball Threads mit höherer
Priorität werden bevorzugt
bewegt

- Mit der Methode

static void yield()

kann ein Thread seine Kontrolle des Prozessors abgeben und anderen die Chance zur Zuteilung geben.



Unterbrechung und Terminieren

- Unterbrechen von Threads durch

void interrupt()

d.h., befindet sich der Thread in einem blockierten Zustand, wird eine `InterruptedException` geworfen und der Thread aktiviert.

- Mit static-Methode

static boolean interrupted()

wird für den aktuellen Thread der Interrupt-Status abgefragt und rückgesetzt (!)

- Interrupts sind für die außerordentliche Terminierung eines Threads wichtig

```
public void run() {
    while (!interrupted()) { // fortsetzen
        try {
            // do something
            sleep(1000);
        } catch (InterruptedException e) { // Fange Interrupt in sleep
            interrupt(); // Rufe nochmals interrupt() auf,
            // um interrupted() erneut zu setzen
        }
    }
    // terminieren
}
```

Anmerkung: Dieses Vorgehen kann die gefährliche `stop`-Anweisung ersetzen, die nicht mehr verwendet werden soll



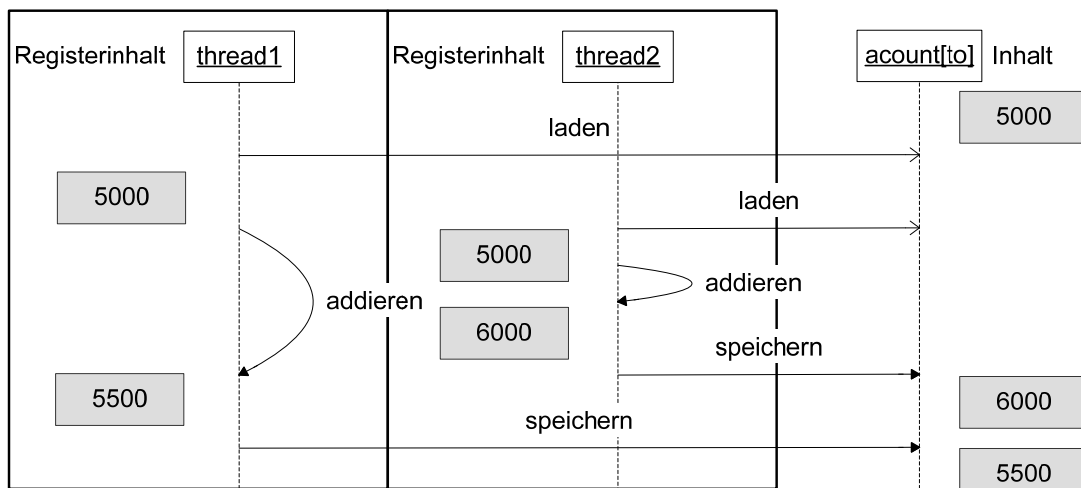
- Motivation und Grundlagen
- Threadzustände und Prioritäten
- Synchronisation
- Weiteres



Wechselseitiger Ausschluss

- Greifen mehrere Threads gleichzeitig auf ein Objekt zu, muss wechselseitiger Ausschluss bei nicht-atomaren Operationen realisiert werden
- Beispiel: Banktransaktionen
 - In einer Bankapplikation wird zwischen Konten Geld transferiert; Addieren und Subtrahieren sind nicht atomar

```
void transfer(int from, int to, int amount)
{
    accounts[from] -= amount;
    accounts[to] += amount;
}
```



Locks

- Basisklasse `Object` bietet Locks für wechselseitiger Ausschluss
- Jedes `Object` hat einen Schlüssel (Lock) und verwaltet eine Queue
 - Threads können Lock anfordern
 - werden eventuell in der Queue als wartend auf den Lock gespeichert
 - bei Freiwerden des Locks erhält nächster in Queue den Lock und kann Ausführung fortsetzen
- damit kann ein beliebiger Code (aber insbesondere eine Methode des Objekts) unter exklusivem Zugriff auf das Objekt ausgeführt werden
- dies passiert, indem man eine Methode oder einen Block als **synchronisiert** deklariert



synchronized Methode

- Wird eine Methode als `synchronisiert` deklariert, muss bei Ausführung der Methode der Lock des Objekts (`this`) erhalten werden
- Ist dieser nicht verfügbar, kann die Methode nicht begonnen und es muss auf die Zuteilung des Locks gewartet werden
- Bei statischen Methoden wird auf das Klassenobjekt synchronisiert

Methode gelockt auf
`this`-Objekt

```
class Bank {  
    ...  
    synchronized void transfer(int from, int to, int amount) {  
        accounts[from] -= amount;  
        accounts[to] += amount;  
    }  
    ...  
}
```



synchronized Block

- Blöcke können auf ein beliebiges Objekt *synchronized* werden
- Block kann nur betreten werden, wenn man den Lock des Objekts hat

```
class Consumer extends Thread {  
    private Object o = new Object();  
    ...  
    public void run() {  
        while (true) {  
            synchronized (o) {  
                // gelockt auf Objekt o  
            }  
        }  
    }  
}
```

Block gelockt
auf Objekt o



wait und notify

Methoden bei Object:

- **wait** gibt Lock auf Objekt **vorübergehend frei** und reiht den Thread als „**wartend auf das Objekt**“ ein

`void wait() throws InterruptedException`
der Thread wird als wartend auf das Objekt blockiert; Lock auf das Objekt wird freigegeben

`void wait(long timeout) throws InterruptedException`
wie `wait`, aber wartet höchstens `timeout` Millisekunden

- **notify/notifyAll** wecken wartende Threads auf; diese laufen nach Wiederherstellung des Locks weiter

`notify()` Es wird ein (!) auf das Objekt wartender Thread aufgeweckt; Thread läuft nach Wiederherstellung des Locks weiter

`notifyAll()` wie `notify`; es werden alle auf das Objekt wartenden Threads aufgeweckt

Beispiel: Warten bis Konto gefüllt

```
public synchronized void transfer(int from, int to, int amount)  
    throws InterruptedException {  
    while (accounts[from] < amount)  
        wait();  
    accounts[from] -= amount;  
    accounts[to] += amount;  
    notifyAll();  
}
```



Beispiel: Producer – Consumer (1)

- Producer produziert Elemente und schreibt sie in den Puffer
- Consumer konsumiert produzierte Elemente aus dem Puffer

```
public class ProducerConsumerAppl {  
  
    public static void main(String[] args) {  
  
        Buffer buffer = new Buffer();  
  
        Producer p = new Producer(buffer);  
        Consumer c = new Consumer(buffer);  
  
        p.start(); // Starten Producer  
        c.start(); // Starten Consumer  
  
        try {  
            Thread.sleep(10000);  
        } catch (InterruptedException e) {  
        } finally {  
            p.finish();  
        }  
    }  
}
```

```
public class Buffer {  
  
    Object obj = null;  
  
    public void put(Object o) {  
        obj = o;  
    }  
  
    public Object get() {  
        Object o = obj;  
        obj = null;  
        return o;  
    }  
  
    public boolean isEmpty() {  
        return obj == null;  
    }  
}
```



Beispiel: Producer – Consumer (2)

```
class Producer extends Thread {  
  
    private Buffer buffer;  
  
    public Producer(Buffer buffer) { this.buffer = buffer; }  
  
    public void run() {  
        int i = 0;  
        Object o;  
        while (!interrupted()) {  
            try {  
                synchronized (buffer) {  
                    while (!buffer.isEmpty()) {  
                        buffer.wait();  
                    }  
                    o = new Integer(i++);  
                    buffer.put(o);  
                    System.out.println("Produzent erzeugte " + o);  
                    buffer.notifyAll();  
                }  
                Thread.sleep((int) (100 * Math.random())); // schlafen  
            } catch (InterruptedException e) {  
                interrupt();  
            }  
        }  
    }  
  
    public void finish() {  
        interrupt();  
    }  
}
```



Beispiel: Producer – Consumer (3)

```
class Consumer extends Thread {  
  
    private Buffer buffer;  
  
    public Consumer(Buffer buffer) {  
        this.buffer = buffer;  
        setDaemon(true);  
    }  
  
    public void run() {  
        while (!interrupted()) {  
            try {  
                synchronized (buffer) {  
                    while (buffer.isEmpty()) {  
                        buffer.wait();  
                    }  
                    Object o = buffer.get();  
                    System.out.println("Konsument fand " + o);  
                    buffer.notifyAll();  
                }  
                Thread.sleep((int) (100 * Math.random()));  
            } catch (InterruptedException e) {  
                interrupt();  
            }  
        }  
    }  
}
```



Beispiel: BankAccounts (1)

- Transferieren von Geld zwischen Konten in mehreren Threads

```
public class SynchBankTest {  
  
    public static void main(String[] args) {  
        Bank b = new Bank(NACCOUNTS, INITIAL_BALANCE);  
        int i;  
        for (i = 0; i < NACCOUNTS; i++) {  
            TransferThread t = new TransferThread(b, i,  
                INITIAL_BALANCE);  
            t.setPriority(Thread.NORM_PRIORITY + i % 2);  
            t.start();  
        }  
    }  
  
    public static final int NACCOUNTS = 10;  
    public static final int INITIAL_BALANCE = 10000;  
}
```



Beispiel: BankAccounts (2)

```
class Bank {  
  
    public Bank(int n, int initialBalance) {  
        accounts = new int[n];  
        int i;  
        for (i = 0; i < accounts.length; i++)  
            accounts[i] = initialBalance;  
        ntransacts = 0;  
    }  
  
    public synchronized void transfer(int from, int to, int amount)  
        throws InterruptedException {  
        while (accounts[from] < amount)  
            wait();  
        accounts[from] -= amount;  
        accounts[to] += amount;  
        ntransacts++;  
        notifyAll();  
    }  
  
    ...  
  
    private final int[] accounts;  
    private long ntransacts = 0;  
}
```



Beispiel: BankAccounts (3)

```
class TransferThread extends Thread {  
  
    public TransferThread(Bank b, int from, int max) {  
        bank = b;  
        fromAccount = from;  
        maxAmount = max;  
    }  
  
    public void run() {  
        try {  
            while (!interrupted()) {  
                int toAccount = (int) (bank.size() * Math.random());  
                int amount = (int) (maxAmount * Math.random());  
                bank.transfer(fromAccount, toAccount, amount);  
                sleep(1);  
            }  
        }  
        catch (InterruptedException e) {}  
    }  
  
    private Bank bank;  
    private int fromAccount;  
    private int maxAmount;  
}
```



Lock-Objekte (ab Java 1.5)

- Ab Java 1.5 gibt es zusätzlich explizite Lock-Objekte
- sind flexibler und mächtiger als Locks bei Object

```
import java.util.concurrent.lock.*;  
class Bank {  
    ...  
    synchronized void transfer(int from, int to, int amount) {  
        bankLock.lock();  
        try {  
            accounts[from] -= amount;  
            accounts[to] += amount;  
        } finally {  
            bankLock.unlock();  
        }  
        ...  
    }  
    private Lock bankLock = new ReentrantLock();  
}
```



Lock-Objekte: Möglichkeiten

- tryLock: Abfragen, ob Lock verfügbar

```
Lock lock = ...;  
if (lock.tryLock()) {  
    try {  
        // manipulate protected state  
    } finally {  
        lock.unlock();  
    }  
} else {  
    // perform alternative actions  
}
```

- tryLock mit Timeout

```
if (lock.tryLock(50, TimeUnit.SECONDS)) ... {  
    ...  
}
```

- lockInterruptibly: Lock mit Möglichkeit der Unterbrechung

```
lock.lockInterruptibly() {  
    try {  
        // manipulate protected state, allow interrupt  
    } catch (InterruptedException e) { ...  
    } finally { lock.unlock(); }  
}
```



Bedingungen bei Lock-Objekten (ab Java 1.5)

- Lock-Objekte erlauben die Erzeugung beliebiger Condition-Objekte
 - ➔ effizienter, weil sehr selektive Bedingungen

```
class Bank {  
  
    public Bank(int n, double initialBalance) {  
        bankLock = new ReentrantLock();  
        sufficientFunds = bankLock.newCondition();  
        ...  
    }  
  
    public void transfer(int from, int to, double amount) throws InterruptedException {  
        bankLock.lock();  
        try {  
            while (accounts[from] < amount)  
                sufficientFunds.await();  
            accounts[from] -= amount;  
            accounts[to] += amount;  
            sufficientFunds.signalAll();  
        } finally {  
            bankLock.unlock();  
        }  
    }  
    ...  
    private Lock bankLock;  
    private Condition sufficientFunds;  
}
```



Beispiel Lock, TimeUnit

```
public class Test {  
    public static void main(String[] args) {  
        ReentrantLock lock = new ReentrantLock();  
        Thread susi = new Thread(new User(lock)); susi.setName("Susi");  
        Thread maxi = new Thread(new User(lock)); maxi.setName("Maxi");  
        susi.start(); maxi.start();  
    }  
}  
  
class User implements Runnable {  
    private ReentrantLock lock;  
    public User(ReentrantLock lock) { this.lock = lock; }  
    public void run() {  
        for (int i = 0; i < 10; ++i) {  
            try {  
                if (lock.tryLock(10, TimeUnit.MILLISECONDS)) {  
                    System.out.printf("%s: Got the lock.%n",  
                        Thread.currentThread().getName());  
                    // do something  
                } else {  
                    System.out.printf("%s: Someone else uses the lock.%n",  
                        Thread.currentThread().getName());  
                }  
            } catch (InterruptedException ign) { ign.printStackTrace(); }  
            finally { if (lock.isHeldByCurrentThread()) { lock.unlock(); } }  
            // do something  
        }  
    }  
}
```



Read/Write Locks

- Oft ist es sinnvoll, wenn viele Lesen und nur wenige Schreiben wollen

```
public class Bank {  
    private ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();  
    private Lock readLock = rwl.readLock();  
    private Lock writeLock = rwl.writeLock();  
  
    public double getTotalBalance() {  
        readLock.lock();  
        try { ...  
        finally { readLock.unlock(); }  
    }  
  
    public void transfer(...) {  
        writeLock.lock();  
        try { ... }  
        finally { writeLock.unlock(); }  
    }  
    ...  
}
```



Multithreading

- Motivation und Grundlagen
- Threadzustände und Prioritäten
- Synchronisation
- Weiteres



- Oft ist es notwendig einen Thread zu erzeugen und den Ablauf mit diesem abzustimmen
- Die Anweisung `join` erlaubt es, auf einen Thread zu warten, bis dieser terminiert ist.

Beispiel:

```
Thread t = new
  Thread() { // anonyme Subklasse von Thread
    public void run() {

      /* Thread Operationen */

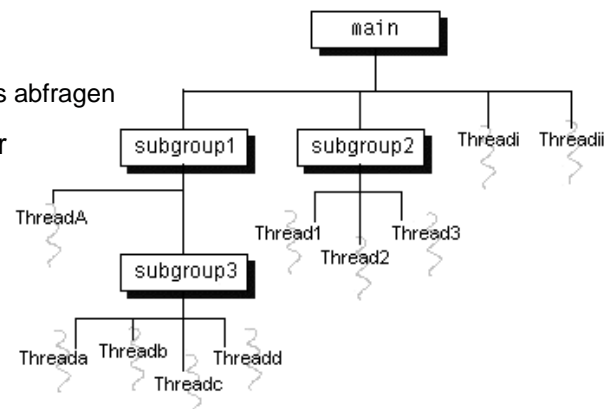
    }
  };

t.start();
t.join(); // warte bis Thread t terminiert
// setze fort
```



Thread-Gruppen

- Oft ist es sinnvoll und hilfreich Threads zu Gruppen zusammenzufassen, um diese gemeinsam behandeln zu können
 - z.B. alle Threads in der Gruppe zu unterbrechen
- Dazu dient die Klasse `ThreadGroup`, mit der eine hierarchische Gruppierung von Threads erfolgen kann
- `ThreadGroup` erlaubt:
 - Unterbrechen aller Threads in der Gruppe
 - Exceptions, die innerhalb eines enthaltenen Threads auftreten, zu behandeln
 - Zugriff auf die enthaltenen Threads
 - Informationen über die enthaltenen Threads abfragen
- Threads können bei ihrer Erzeugung einer `ThreadGroup` zugeordnet werden



Ende der Applikation und Daemon-Threads

- Eine Applikation wird beendet, wenn alle ihre Threads terminiert (tot) sind
- Eine Ausnahme bilden dabei aber die sogenannten Daemon-Threads; diese werden automatisch beendet, wenn der letzte Nicht-Daemon-Thread einer Applikation terminiert hat
- Daemon-Threads verwendet man daher für Hilfsdienste
- Threads können durch Setzen der daemon-Property mit
`void setDaemon(boolean on)`
zu Daemon-Threads gemacht werden



Veraltete Methoden

stop:

- Mit `stop()` kann man einen Thread „töten“; er wird sofort terminiert
- sollte nicht verwendet werden, weil dadurch jede Aktion sofort beendet wird und dadurch inkonsistente Zustände der bearbeiteten Objekte entstehen können
- Beispiel `transfer()` bei Bank: Es wird zwar von einem Konto noch abgeboben aber auf das andere Konto nicht mehr gebucht

suspend / resume:

- Mit `suspend()` kann ein Thread vorübergehend blockiert und mit `resume()` wieder aufgeweckt werden
- dabei gibt er aber Locks von Objekten nicht frei (der Lock kann erst wieder frei gegeben werden, wenn der Thread mit `resume()` wieder aufgeweckt wird)
- Dadurch können sehr leicht Deadlocks entstehen, die Verwendung von `suspend` wird nicht empfohlen



Thread-sichere Collections

- Collections in `java.util` nicht Thread-sicher
 - d.h. Zugriffe sind nicht *synchronized*
- Collections API erlaubt zu jeder Collection-Klasse eine *synchronized* Version zu erzeugen
 - jeder Zugriff sperrt gesamte Collection

```
Collection c = Collections.synchronizedCollection(myCollection);  
Set s = Collections.synchronizedSet(new HashSet());  
SortedSet s = Collections.synchronizedSortedSet(new TreeSet());  
List list = Collections.synchronizedList(new ArrayList());  
Map m = Collections.synchronizedMap(new HashMap());  
SortedMap m = Collections.synchronizedSortedMap(new TreeMap());
```

- Java 1.5 bietet effizientere Implementierungen die jeweils nur einen Teil sperren

```
java.util.concurrent.ConcurrentHashMap<K, V>  
java.util.concurrent.ConcurrentLinkedQueue<E>
```



Weiteres bei Java 1.5

- `TimeUnit`
 - Hilfsklasse zum Bestimmen von Zeitspannen
- `Executor`, `ExecutorService`, `Executors`, `TreadPools`
 - Hilfsklassen zum Ausführen von Aufgaben
- `Callable<T>`
 - Beschreibt eine Aufgabe, wie *Runnable*
 - Allerdings mit generischem Rückgabewert und Exception
- `Future`, `FutureTask`
 - Ermöglicht Asynchrone Berechnungen.
- `BlockingQueue<T>`
 - Erlaubt die Synchronisation von Producer/Consumer über Queue
- `ConcurrentLinkedList<T>`, `ConcurrentHashMap<K, V>`
 - Thread-sichere, hocheffiziente Collections
- Weitere Klassen, siehe Pakete:
 - `java.util.concurrent`, `java.util.concurrent.atomic`,
`java.util.concurrent.locks`



Beispiel *Callable*, *FutureTask*

```
public class Test {
    public static void main(String[] args)
        throws InterruptedException, ExecutionException {
        Adder adder = new Adder(1, 2);
        FutureTask<Integer> addTask = new FutureTask<Integer>(adder);
        new Thread(addTask).start();
        System.out.printf("Task started (%d).\n", System.currentTimeMillis());
        while (!addTask.isDone()) {
            Thread.sleep(100);
            System.out.printf("Task still in progress (%d).\n",
                System.currentTimeMillis());
        }
        System.out.printf("Task finished (%d), result: %d\n",
            System.currentTimeMillis(), addTask.get());
    }
}

class Adder implements Callable<Integer> {
    private Integer i1, i2;
    public Adder(Integer i1, Integer i2) {
        this.i1 = i1; this.i2 = i2;
    }
    public Integer call() throws Exception {
        Thread.sleep(1000);
        return i1 + i2;
    }
}
```



Beispiel *Executor*

```
public class Test {
    public static void main(String[] args)
        throws InterruptedException, ExecutionException {
        ExecutorService e = Executors.newFixedThreadPool(10);
        Future<Integer> x = e.submit(new X());
        e.submit(new Y());
        Future<String> ys = e.submit(new Y(), "Done");
        System.out.printf("x: %d, ys: %s\n", x.get(), ys.get());
    }
}

class X implements Callable<Integer> {
    public Integer call() throws Exception {
        System.out.println("X started.");
        Thread.sleep(1000);
        System.out.println("X done.");
        return 42;
    }
}

class Y implements Runnable {
    public void run() {
        System.out.println("Y started.");
        try { Thread.sleep(500); }
        catch (InterruptedException ign) { ign.printStackTrace(); }
        System.out.println("Y done.");
    }
}
```



BlockingQueue<T>

- BlockingQueue<T> realisiert FIFO-Queue mit Blockieren
- Schnittstelle mit unterschiedliche Zugriffsmethoden
 - unterschiedliche Reaktion bei Lesen von leerer Queue und Schreiben in volle Queue

	<i>Throws exception</i>	<i>Special value</i>	<i>Blocks</i>	<i>Times out</i>
Insert	add(e)	offer(e)	put(e)	offer(e, time, unit)
Remove	remove()	poll()	take()	poll(time, unit)
Examine	element()	peek()	<i>not applicable</i>	<i>not applicable</i>

- Mehrere Implementierungen
 - ArrayBlockingQueue: Array-Implementierung mit Größenbeschränkung
 - LinkedBlockingQueue: LinkedList-Implementierung, Größenbeschränkung optional
 - PriorityQueue: Queue sortiert nach Priorität, unbeschränkt
 - DelayQueue: Queue, wobei Elemente zeitlich verzögert zur Verfügung gestellt werden
 - SynchronousQueue: Lese und Schreibzugriffe der Thread müssen gleichzeitig erfolgen



Beispiel: Producer/Consumer mit BlockingQueue<T>

```
import java.util.concurrent.*;
public class BlockingQueueTest {
    static final int CAPACITY = 10;
    public static void main(String[] args) {
        BlockingQueue<Integer> queue = new ArrayBlockingQueue<Integer>(CAPACITY);
        new Producer(queue).start();
        new Consumer(queue).start();
        ...
    }
}
```

```
class Producer extends Thread {
    private BlockingQueue<Integer> buffer;
    public void run() {
        int i = 0;
        while (!interrupted()) {
            try {
                buffer.put(i++);
                ...
            } catch (InterruptedException e) { interrupt(); }
        }
    }
}
```

```
class Consumer extends Thread {
    private BlockingQueue<Integer> buffer;
    public void run() {
        while (!interrupted()) {
            try {
                Integer i = buffer.take();
                ...
            } catch (InterruptedException e) { interrupt(); }
        }
    }
}
```



API Zusammenfassung (1)

public class Thread implements Runnable

A *thread* is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.

Thread()

Allocates a new Thread object.

Thread(Runnable target)

Allocates a new Thread object with a Runnable object .

Thread(ThreadGroup group, Runnable target)

Allocates a new Thread object with a Runnable object and assigns it to the group.

boolean isAlive()

Tests if this thread is alive.

boolean isDaemon()

Tests if this thread is a daemon thread.

boolean isInterrupted()

Tests whether this thread has been interrupted.

static boolean holdsLock(Object obj)

Returns true if and only if the current thread holds the monitor lock on the specified object.

void run()

Run method called when started. Does nothing by default. Calls the run method of the Runnable-Object if set.

void setDaemon(boolean on)

Marks this thread as either a daemon thread or a user thread.

void setPriority(int newPriority)

Changes the priority of this thread.

static void sleep(long millis)

Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.

public void interrupt()

Interrupts this thread.

public final void join() throws InterruptedException

Waits for this thread to die.

static void yield()

Causes the currently executing thread object to temporarily pause.



API Zusammenfassung (2)

public interface Runnable

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called run.

void run()

When an object implementing interface Runnable is used to create a thread, starting the thread causes the object's run method to be called in that separately executing thread.

public class InterruptedException extends Exception

Thrown when a thread is waiting, sleeping, or otherwise paused for a long time and another thread interrupts it using the interrupt method in class Thread.

public class Object

Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.

void wait()

Causes current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.

void wait(long timeout)

Causes current thread to wait until either another thread invokes the

void notify()

Wakes up a single thread that is waiting on this object's monitor.

void notifyAll()

Wakes up all threads that are waiting on this object's monitor.



API Zusammenfassung (3)

`public class ThreadGroup extends Object`

A thread group represents a set of threads. In addition, a thread group can also include other thread groups.

`int activeCount()`

Returns an estimate of the number of active threads in this thread group.

`int enumerate(Thread[] list)`

Copies into the specified array every active thread in this thread group and its subgroups.

`int enumerate(Thread[] list, boolean recurse)`

Copies into the specified array every active thread in this thread group.

`int getMaxPriority()`

Returns the maximum priority of this thread group.

`ThreadGroup getParent()`

Returns the parent of this thread group.

`void interrupt()`

Interrupts all threads in this thread group.

`void setDaemon(boolean daemon)`

Changes the daemon status of this thread group.

`void setMaxPriority(int pri)`

Sets the maximum priority of the group.

`void uncaughtException(Thread t, Throwable e)`

Called by the Java Virtual Machine when a thread in this thread group stops because of an uncaught exception.



Literatur

- Java Tutorial, Multithreading,
<http://java.sun.com/docs/books/tutorial/essential/threads/index.html>
- Horstmann, Cornell, Core Java 2, Band 2 - Expertenwissen, Markt und Technik, 2002:
Kapitel 1
- Krüger, Handbuch der Java-Programmierung, 3. Auflage, Addison-Wesley, 2003,
<http://www.javabuch.de>: Kapitel 22



- Erzeugen, Starten, Ablauf von Threads:
 - ➔ Core Java 2: v2ch1. BounceThread. j ava
- Scheduling und Prioritäten:
 - ➔ Core Java 2: v2ch1. BounceExpress. j ava
- Synchronisation:
 - ➔ Core Java 2: v2ch1. SynchBankTest. j ava

