

Multithreading

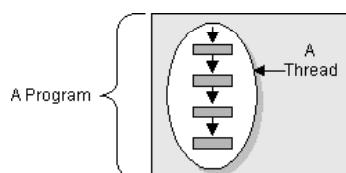
Motivation und Grundlagen
Threadzustände und Prioritäten
Synchronisation
Weiteres



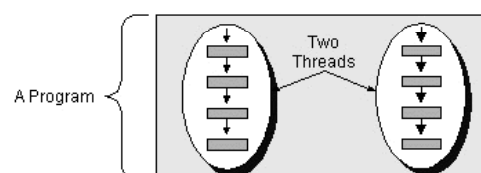
Multithreading

- Threads in Java sind quasiparallele Programmabläufe innerhalb einer VM
- Threads in Java sind mächtiges und bequemes Konzept, um nebenläufige Abläufe zu realisieren, wie zB:
 - Applikationslogik und Benutzerinteraktion bei GUI-Anwendungen
 - interaktive Anwendungen bei denen mehrere Aktivitäten gleichzeitig laufen sollen
 - Bedienung von mehreren gleichzeitigen Anforderungen am Server
 - Animationen mit mehreren aktiven Agenten
 - ...
- Threads in Java laufen in einem gemeinsamen Speicherbereich
- Threads können/müssen bei gemeinsam benutzten Objekten (Shared Memory) kooperieren.

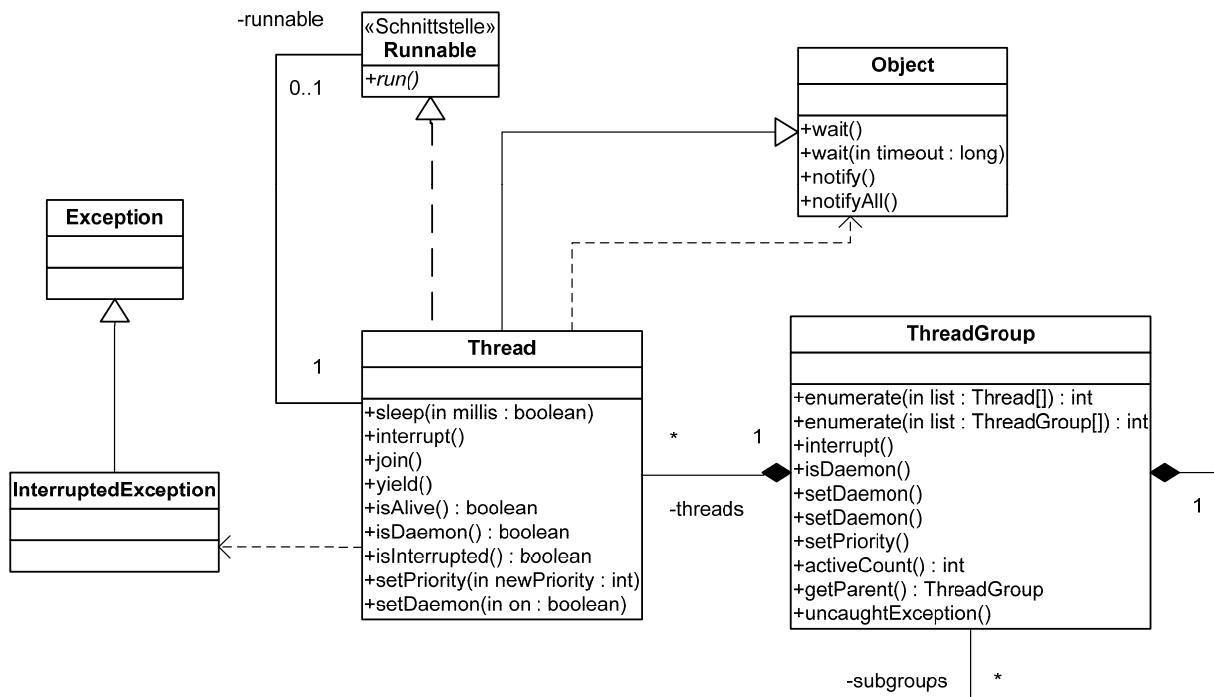
single-threaded



multi-threaded



Klassendiagramm (Auszug)



Überblick über Klassen

- **Thread:**
 - Thread-Objekte repräsentieren einen Thread
 - definiert Methoden für starten, unterbrechen, ...
 - definiert statische-Methoden, um
 - aktuell laufenden Thread zu steuern, zB ihn schlafen zu legen
 - Verwaltung aller aktuell existierenden Threads
- **Runnable:**
 - Interface **Runnable** definiert Methode `run()`, welche den von einem Thread auszuführenden Code enthält
 - **Thread** implementiert **Runnable**
 - **Thread** kann 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` und `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 (zB: 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 {
    public void addBall() {
        Ball b = new Ball(canvas);
        Thread 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-Objekt

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

Beispiel: BallThread

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

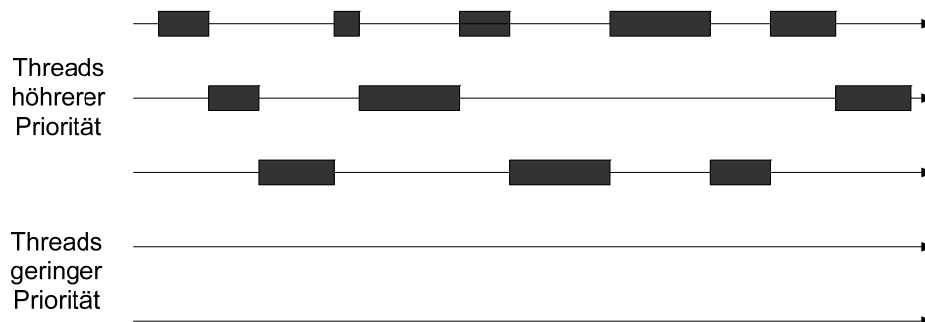
class Ball implements Runnable {
    ...

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



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)

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

- 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()) {  
        try {  
            // do something  
            sleep(1000);  
        } catch (InterruptedException e) { // Fange Interrupt in sleep  
            interrupt(); // Rufe nochmals interrupt() auf,  
                        // um interrupted() zu setzen  
        }  
    }  
    // terminieren  
}
```

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



Multithreading

Motivation und Grundlagen

Threadzustände und Prioritäten

Synchronisation

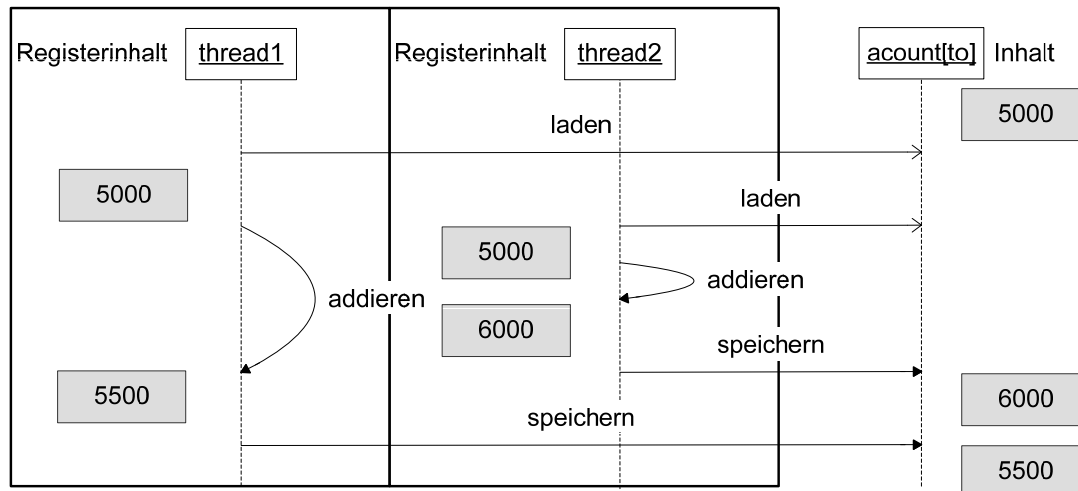
Weiteres



Wechselseitiger Ausschluss

- Greifen mehrere Threads gleichzeitig auf ein Objekt zu, muss wechselseitiger Ausschluss bei nicht-atomaren Operationen realisiert werden
- Beispiel: Banktransaktionen

```
void transfer(int kto, int amount) {  
    account[kto] = account[kto] + amount;  
}
```



Monitor

- Basisklasse `Object` realisiert `Monitor` für wechselseitiger Ausschluss
- `Monitor` 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 `synchronized` 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

```
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 Bank {  
    private Object accountLock = new Object();  
    private Object customerLock = new Object();  
    ...  
    void transfer(int from, int to, int amount) {  
        synchronized (accountLock) {  
            account[from] -= amount;  
            account[to] += amount;  
        }  
    }  
    void addCustomer(Customer customer) {  
        synchronized (customerLock) {  
            customers.add(customer);  
        }  
    }  
}
```



wai t und noti fy

- Soll ein synchronisierter Code nicht fortgesetzt werden, kann er den Lock zurückgeben und den Thread als „**wartend auf das Objekt**“ einreihen
- Obj ect stellt dazu Methoden zur Verfügung

wai t() der Thread wird als wartend auf das Objekt blockiert;
Lock auf das Objekt wird freigegeben

wai t(Long ti meout) wie wai t, zusätzlich erfolgt nach ti meout Millisekunden
ein Interrupt

noti fy() Es wird ein (!) auf das Objekt wartender Thread aufgeweckt

noti fyAl l () Es werden alle auf das Objekt wartenden Threads
aufgeweckt

Beispiel: Warten bis Konto gefüllt

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



Beispiel: Producer – Consumer (1)

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

```
public class ProducerConsumerApp! {
    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) {
            // nothing to do
        } finally {
            p.interrupt();
            c.interrupt();
        }
    }
}
```

```
// This buffer is NOT thread-safe!
public class Buffer {
    private 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 final Buffer buffer;

    public Producer(Buffer buffer) { this.buffer = buffer; }

    public void run() {
        int i = 0;

        while (!interrupted()) {
            try {
                synchronized (buffer) {
                    // Sperren des Puffers
                    while (!buffer.isEmpty()) {
                        // warte auf buffer leer
                        buffer.wait();
                    }
                    Object o = new Integer(i++);
                    buffer.put(o);
                    // Element produzieren
                    System.out.println("Produzent erzeugte " + o);
                    buffer.notifyAll();
                    // benachrichtige wartende Consumer
                }
                Thread.sleep((int) (100 * Math.random())); // schlafen
            } catch (InterruptedException e) {
                interrupt();
            }
        }
    }
}
```



Beispiel: Producer – Consumer (3)

```
class Consumer extends Thread {
    private final Buffer buffer;

    public Consumer(Buffer buffer) {
        this.buffer = buffer;
    }

    public void run() {
        while (!interrupted()) {
            try {
                synchronized (buffer) {
                    // Sperren des Puffers
                    while (buffer.isEmpty()) {
                        // warte auf buffer nicht leer
                        buffer.wait();
                    }
                    Object o = buffer.get();
                    // konsumieren
                    System.out.println("Konsument fand " + o);
                    buffer.notifyAll();
                }
                Thread.sleep((int) (100 * Math.random())); // schlafen
            } 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];  
        for (int 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;  
}
```



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) {  
            // nothing to do  
        }  
    }  
  
    private final Bank bank;  
    private final int fromAccount;  
    private final int maxAmount;  
}
```



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.

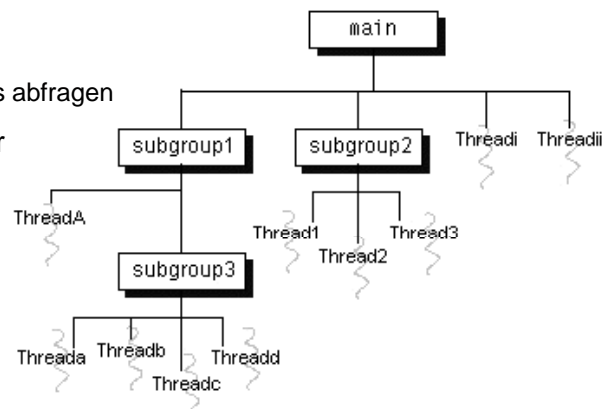
Beispiel:

```
Thread t = new Thread(...);  
t.start();  
// concurrent execution  
t.join(); // wait for termination of Thread t  
// continue
```



Thread-Gruppen

- Oft ist es sinnvoll und hilfreich Threads zu Gruppen zusammenzufassen, um diese gemeinsam behandeln zu können
 - zB: 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



Java 1.5 Klassen

- *Lock*
 - Ermöglicht den exklusiven zugriff auf eine Resource von mehreren Threads
- *TimeUnit*
 - Hilfsklasse zum Bestimmen von Zeitspannen
- *Executor, ExecutorService, Executors*
 - Hilfsklassen zum Ausführen von Aufgaben
- *Callable*
 - Beschreibt eine Aufgabe, wie *Runnable*
 - Allerdings mit generischem Rückgabewert und Exception
- *Future, FutureTask*
 - Ermöglicht Asynchrone Berechnungen.
- Weitere Klassen, siehe Pakete:
 - `java.util.concurrent`, `java.util.concurrent.atomic`, `java.util.concurrent.locks`

```
Lock l = ...
l.lock();
try {
    // access the resource
    // protected by this lock
} finally {
    l.unlock();
}
```



Java 1.5 Klassen, 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
        }
    }
}
```



Java 1.5 Klassen, 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;
    }
}
```



Java 1.5 Klassen, 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.");
    }
}
```



- Brian Goetz et al., Java Concurrency in Practice, Addison-Wesley, 2007
- 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



Programmbeispiele

- 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

Downloads:

- Die Beispiele zu Core Java 2 können von
<http://www.horstmann.com/corejava.html>
heruntergeladen werden
- Die Beispiele aus Handbuch der Java-Programmierung können von
<http://www.javabuch.de>
heruntergeladen werden

