

# Multithreading

(((( itanius informatik ))))

*Java Foundation Track by Carsten Bokeloh*

## Multithreading

- Grundlagen
- join, yield, interrupt
- Dämonen
- synchronized
- wait & notify
- java.util.concurrent

# Grundlagen

- ❖ Threads ermöglichen Multitasking innerhalb des Programms
- ❖ paralleles(Mehrprozessor) oder quasi-paralleles(Einprozessor) Ablauen bestimmter Programmabschnitte
- ❖ Greifen mehrere Prozesse gleichzeitig auf ein Objekt zu muß man Mechanismen zur Hand haben, um dies gegebenenfalls zu unterbinden um Datenkonsistenz zu gewährleisten.

# Threads

- ❖ Threads können vom Programmierer nur eingeschränkt konfiguriert werden, da letzten Endes das Betriebssystem ihren Ablauf bestimmt.
- ❖ Warteschlange, wo sie dann auf Zuteilung von Prozessorzeit warten
- ❖ Im Ursprung gab es zwei Mechanismen, um aus dem main-Thread andere Threads zu starten

# Interface Runnable

java.lang.Runnable	
Returntyp	Name der Methode
<b>void</b>	<b>run()</b> 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.

# Interface Runnable

```
public class EasyRunnable implements Runnable{  
    @Override  
    public void run() {  
        System.out.println("I am running....");  
    }  
}
```

```
public class EasyRunnableStart {  
    public static void main(String[] args) {  
        Thread thread = new Thread(new EasyRunnable());  
        thread.start();  
    }  
}
```

# Thread

- ❖ Die Klasse Thread implementiert dieses Interface Runnable und stellt die Methode start() bereit. start() teilt dem Betriebssystem mit, daß mit der Methode run() ein eigener Prozeß gestartet werden soll und ruft dann run() auf. Nur über diesen Umweg läuft run() in einem eigenem Prozeß ab.

# Thread

<b>java.lang.Thread</b>	
<b>Returntyp</b>	<b>Name der Methode</b>
<b>void</b>	<b>start()</b> Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.

# Thread

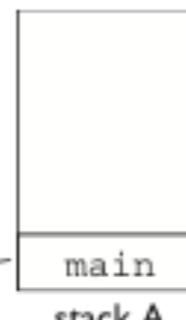
```
public class EasyThread extends Thread{
    public EasyThread(String name) {
        super(name);
    }

    @Override
    public void run()
    {
        System.out.println("I am running ... "+getName());
    }
}
```

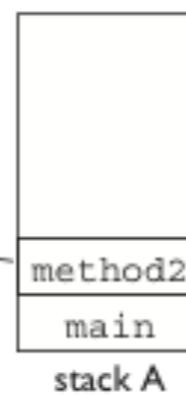
```
public class EasyThreadStart {
    public static void main(String[] args)
    {
        EasyThread easyThread = new EasyThread("Nomen est omen");
        easyThread.start();
    }
}
```

# Einen Thread starten

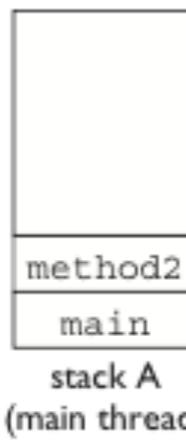
```
public static void main(String [] args) {  
    // running  
    // some code  
    // in main()  
    method2();  
    // running  
    // more code  
}  
  
static void method2() {  
    Runnable r = new MyRunnable();  
  
    Thread t = new Thread(r);  
  
    t.start();  
    // do more stuff  
}
```



1) main() begins

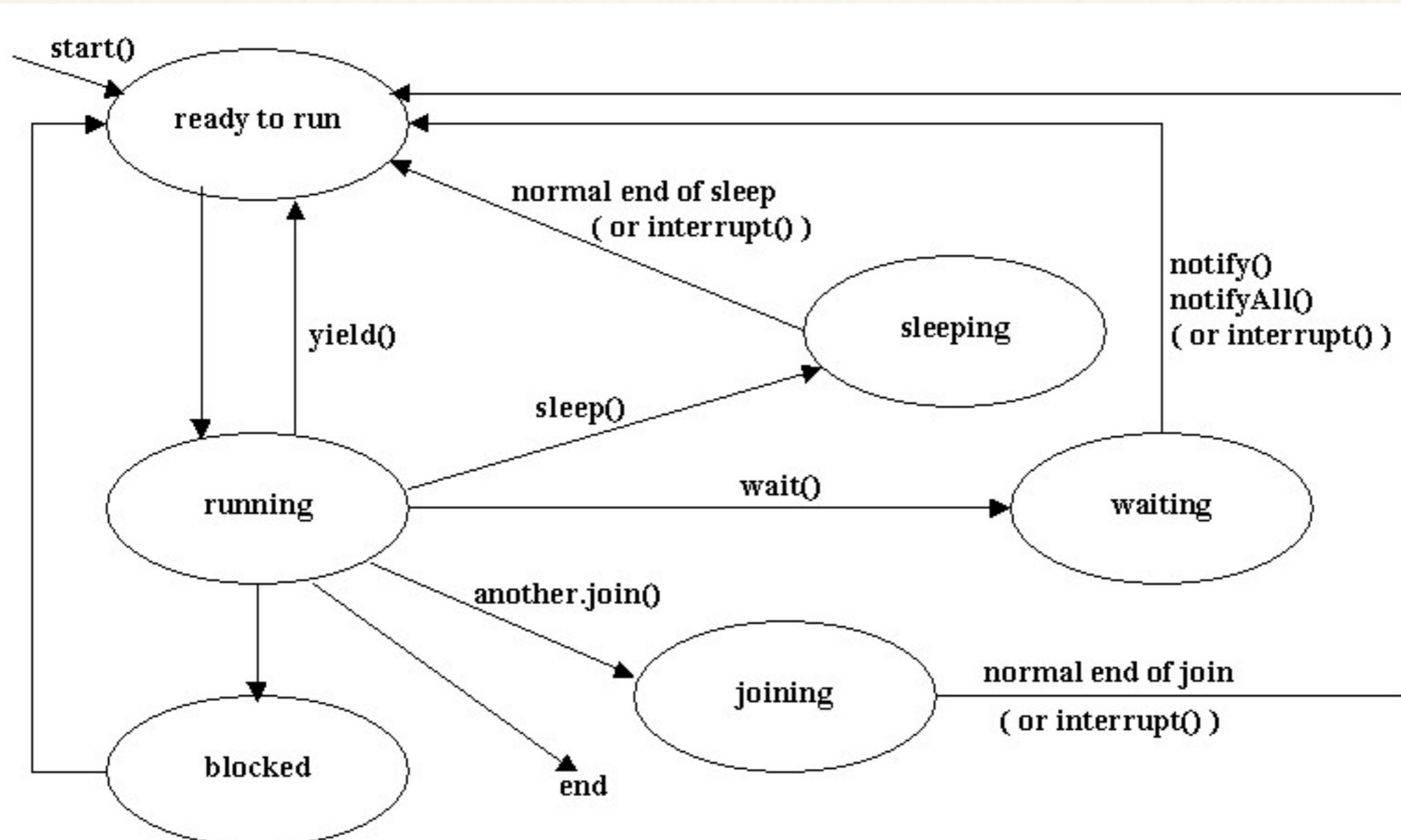


2) main() invokes method2()

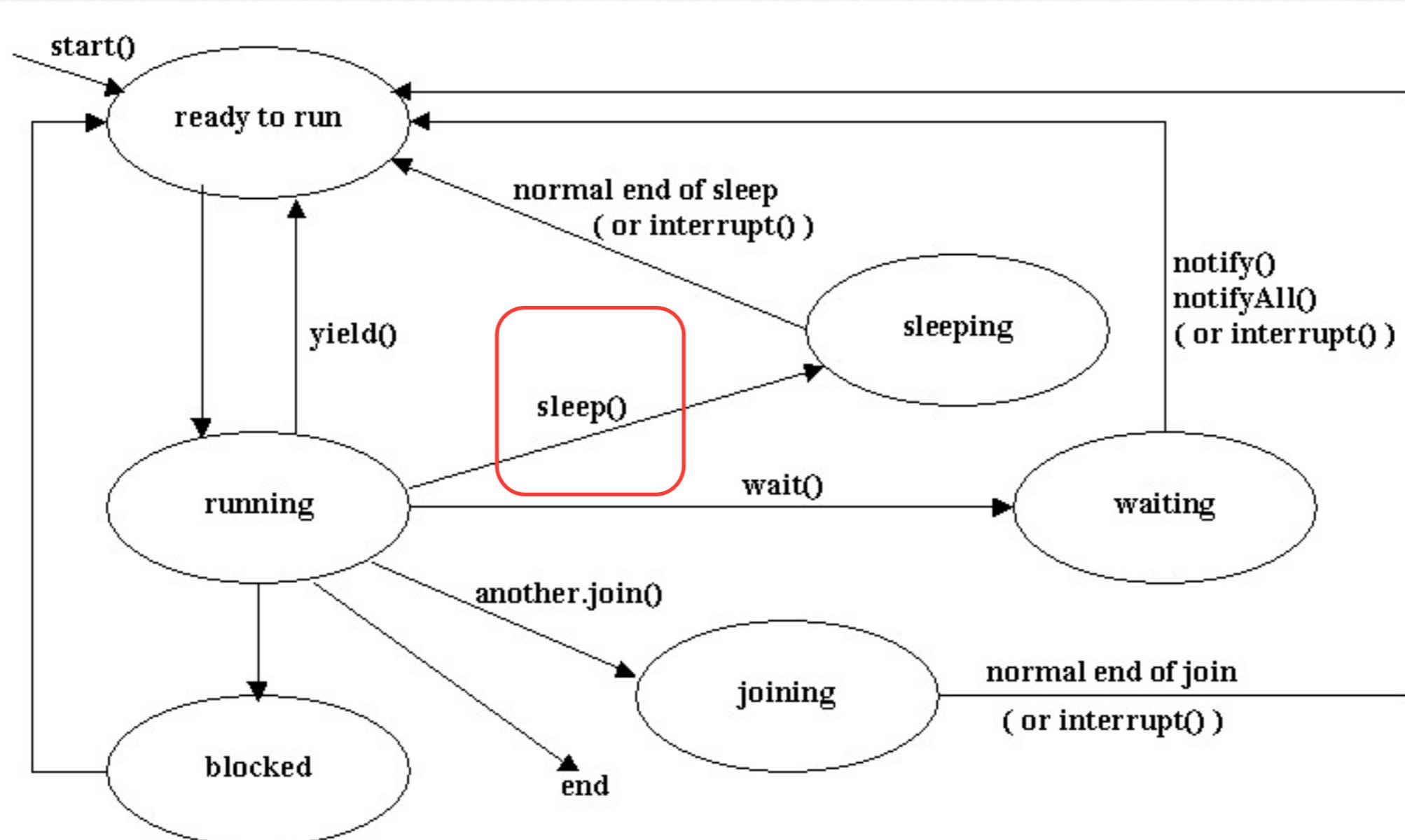


3) method2() starts a new thread

# Threadzustände



# Threadzustände



# sleep

```
public class ThreadSleepDemo implements Runnable {
    Thread t;
    public void run() {
        for (int i = 10; i < 13; i++) {

            System.out.println(Thread.currentThread().getName() + " " + i);
            try {
                // thread schläft für 5000 Millisekunden
                Thread.sleep(5000);
            } catch (Exception e) {
                System.out.println(e);
            }
        }
    }

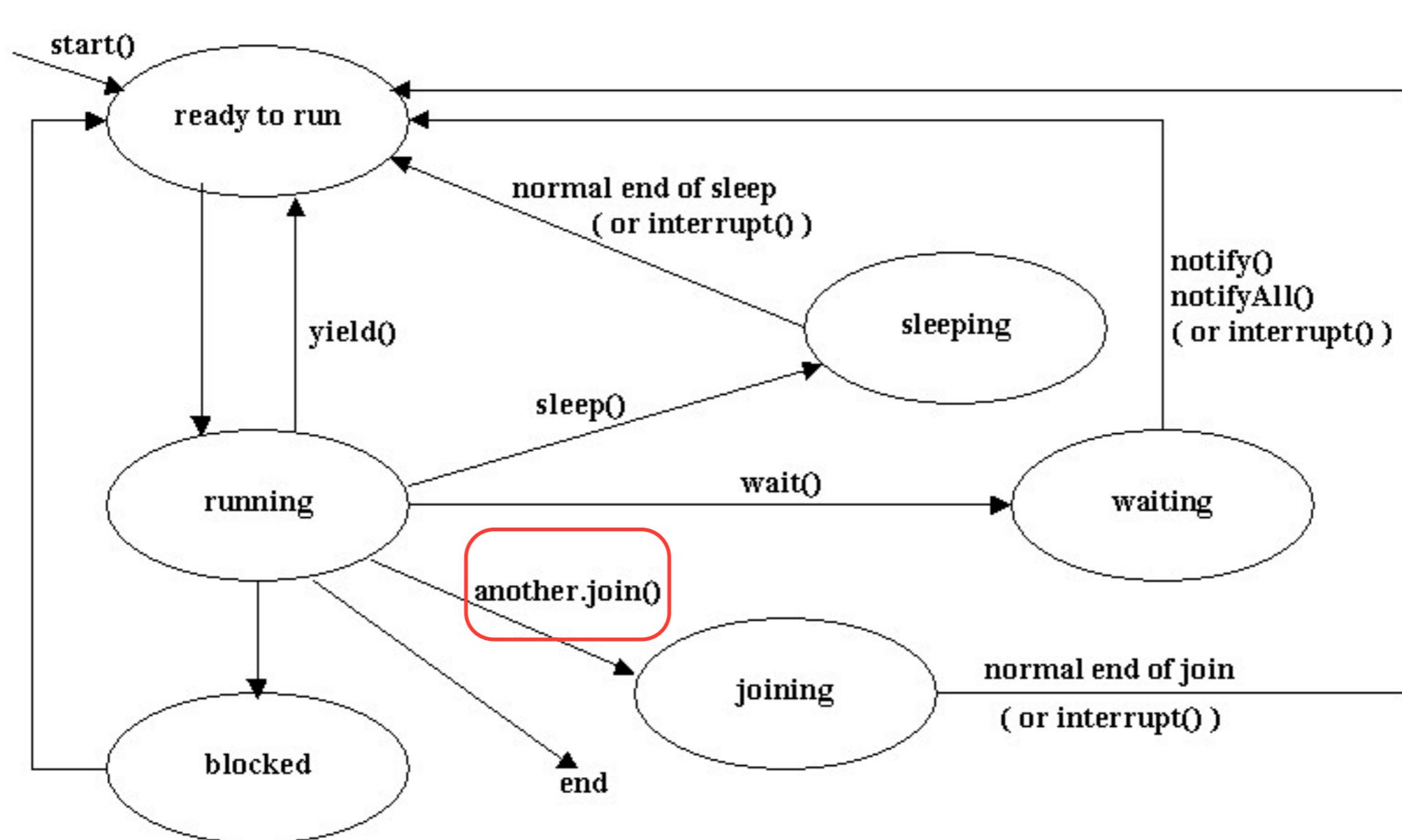
    public static void main(String[] args) throws Exception {
        Thread t = new Thread(new ThreadSleepDemo());
        // Start 1 thread
        t.start();

        Thread t2 = new Thread(new ThreadSleepDemo());
        // Start 2. Thread
        t2.start();
    }
}
```

# Übung

- ❖ UebMultithreading.implementingRunnable /  
Uebung1.txt

# Threadzustände



## join,join(long mills)

- ❖ Thread A kann auf die Beendigung eines anderen Threads B warten
- ❖ join() blockiert also den weiteren Ablauf von A
- ❖ bestimmte Zeit warten über Angabe von millis
- ❖ Um auf einen anderen Thread warten zu können muß der Wartende den Thread als Javaobjekt kennen

# join

# join

```
public class Child extends Thread {  
    private String data = "Start Wert";  
  
    public Child() {  
        System.out.println(Thread.currentThread().getName());  
    }  
  
    /**  
     * @return the data  
     */  
    public String getData() {  
        return data;  
    }  
  
    /* läuft in einem eigenen Prozeß */  
    @Override  
    public void run() {  
        System.out.println(Thread.currentThread().getName());  
        data = "Neuer Wert";  
    }  
}
```

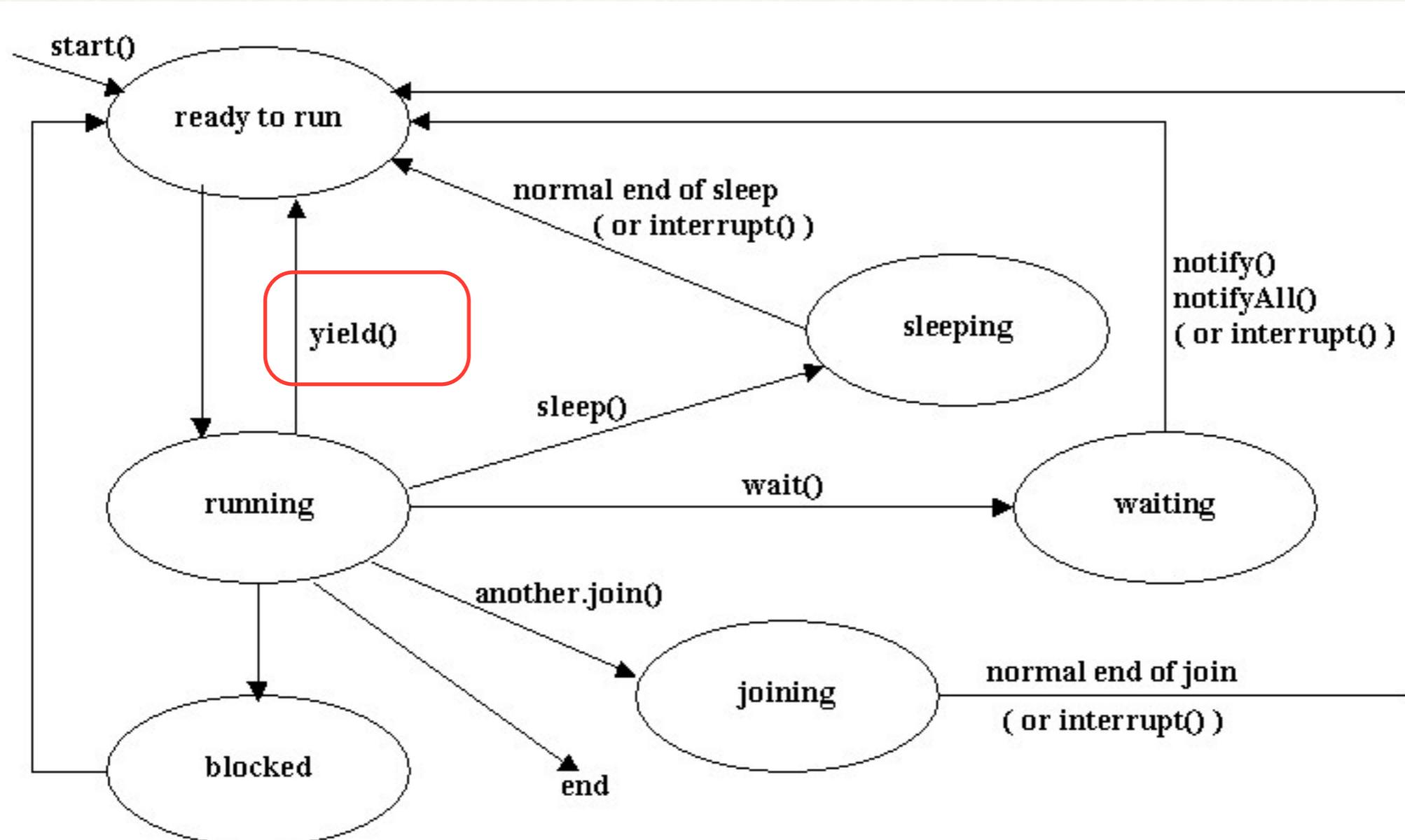
# join

```
public class ParentChild {  
    public static void main(String[] args) {  
        Child myThread = new Child();  
        System.out.println(myThread.getData());  
        myThread.start();  
  
        try {  
            myThread.join(1); // 1 millisekunde warten  
        } catch (InterruptedException ex) {  
        }  
        System.out.println(myThread.getData());  
        // myThread.start(); // java.lang.IllegalThreadStateException  
    }  
}
```

# Übung

- ❖ UebMultithreading.join/Uebung1.txt
- ❖ UebMultithreading.join/Uebung2.txt

# Threadzustände



# yield

- ❖ ein Prozess gibt bekannt, dass er auf Abarbeitung verzichtet
- ❖ Prozess zieht sich zurück und reiht sich wieder in die Warteschlange ein.
- ❖ aber: nur Vorschlag. Kann vom Scheduler ignoriert werden.

# yield

```
class Yield extends Thread
{
    @Override
    public void run()
    {
        for(int i=0; i<10; i++)
        {
            System.out.print(Thread.currentThread().getName());
            yield();
        }
        System.out.println();
    }
}
```

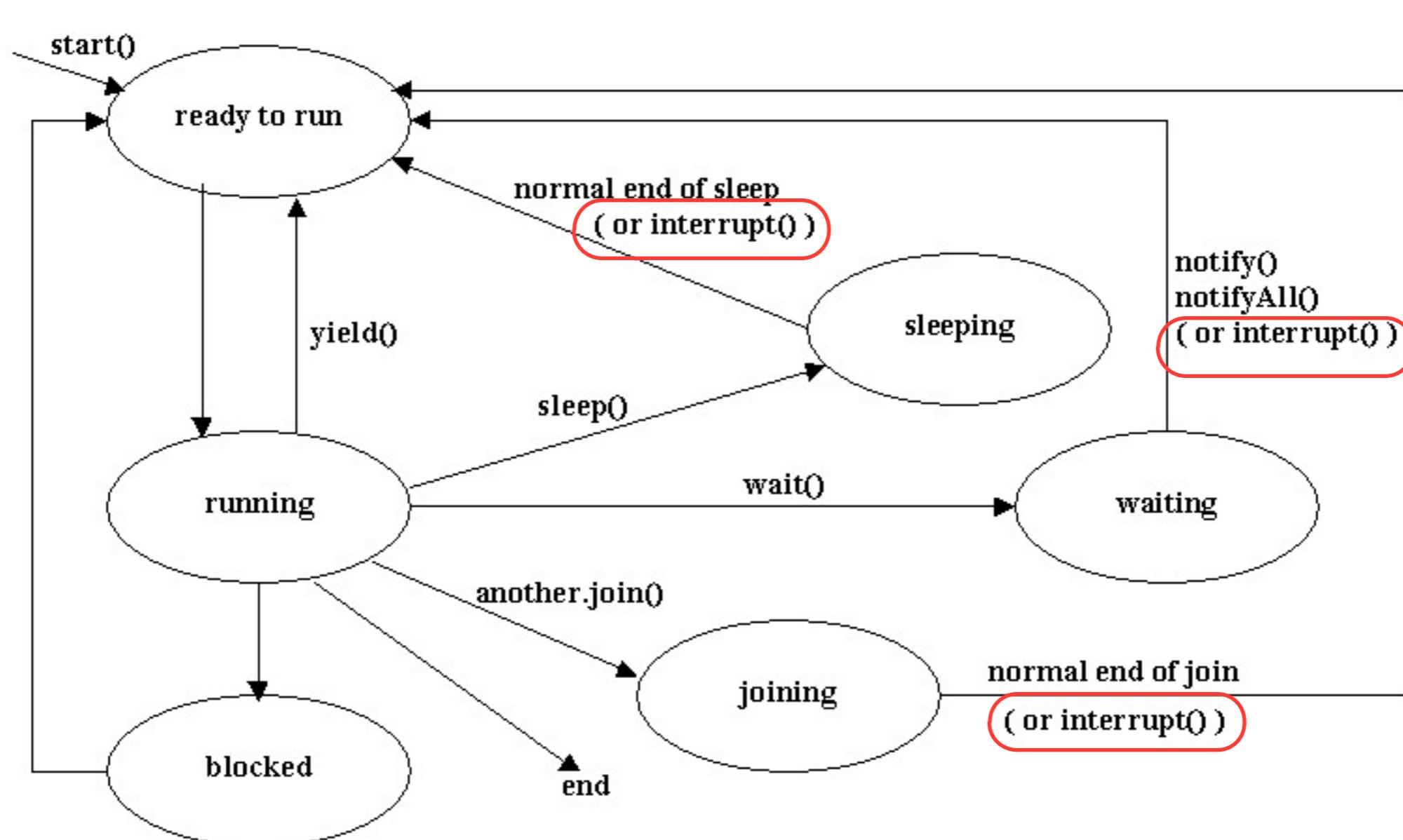
# YieldDemo

```
public class YieldDemo
{
    public static void main(String[] args)
    {
        Yield yield1 = new Yield();
        Yield yield2 = new Yield();
        yield1.setName("1");
        yield2.setName("2");
        yield1.start();
        yield2.start();
    }
}
```

# Übung

- ❖ UebMultithreading.yield/Uebung1.txt

# Threadzustände



## Beenden mit interrupt

- ❖ `interrupt()` selbst beendet keinen Thread
  - ❖ Objektmethode setzt in einem (anderen) Thread-Objekt ein Flag, dass es einen Antrag gab, den Thread zu beenden.
  - ❖ Flag lässt sich mit `isInterrupted()` abfragen.
    - ❖ i.d.R geschieht dies innerhalb einer Schleife

## Beenden mit interrupt

- ❖ boolean isInterrupted
  - ❖ gibt den Status des Interruptflags zurück und verändert diesen Status NICHT.
- ❖ boolean interrupted(statisch)
  - ❖ interrupted() gibt den Status des Interruptflags zurück und setzt diesen auf false, falls er true war.  
D.h. nach einem Aufruf dieser Methode steht das Interruptflag immer auf NICHT unterbrochen.

# Beenden mit interrupt

```
/**  
 * Die Objektmethode interrupt() setzt in einem (anderen) Thread-Objekt  
 * ein Flag, dass es einen Antrag gab, den Thread zu beenden. Sie beendet aber den Thread  
 * nicht, obwohl es der Methodename nahelegt.  
 * Dieses Flag lässt sich mit der Objektmethode isInterrupted() abfragen.  
 * In der Regel wird dies innerhalb einer Schleife geschehen, die darüber bestimmt  
 * , ob die Aktivität des Threads fortgesetzt werden soll.  
 * Die statische Methode interrupted() ist zwar auch eine Anfragemethode  
 * und testet das entsprechende Flag des aktuell laufenden Threads, wie  
 * Thread.currentThread().isInterrupted(), aber zusätzlich löscht es den  
 * Interrupt-Status auch, was isInterrupted() nicht tut.  
 * Zwei aufeinanderfolgende Aufrufe von interrupted() führen daher zu einem  
 * false, es sei denn, in der Zwischenzeit erfolgt eine weitere Unterbrechung.  
 */  
public class InterruptedThread {  
    public static void main(String[] args) {  
        // 1. Thread erzeugen  
        Thread count = new Counter();  
        // 2. Start des Thread  
        count.start();  
        for(int i = 0; i < 100; i++) { // Schleifendurchlauf bis i == 5 // unterbricht den Counter class thread bei // i==5  
            if(i == 5){  
                try {  
                    Thread.sleep(5000);  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
            } if(i == 5){ // Aufruf der interrupt methode auf| // count object  
                count.interrupt(); }  
        }  
    }  
}
```

# Beenden mit interrupt

```
public class Counter extends Thread{
    public void run() {
        int i = 0;
        while(!isInterrupted()){
            System.out.println("Count is " + i++);
            try {
                Thread.sleep(1000);
            } catch(InterruptedException e) {
                break;
            }
        }
        System.out.println("Thread Interrupted");
    }
}
```

## Dämonen

- ❖ Thread muß nicht beendet sein bei Mainthread endet.  
Threads können länger laufen als der Mainthread.
- ❖ Möglichkeit Thread so zu konfigurieren, daß er bei Programmende automatisch beendet wird.
- ❖ sog. Dämonthread,
  - ❖ `setDaemon(boolean)`
- ❖ JVM beendet dann diesen Thread automatisch, wenn der letzte (Nichtdämon-) Thread sich beendet hat.

# Dämon

```
public class Daemon extends Thread
{
    public Daemon()
    {
        this.setDaemon(true);
    }

    @Override
    public void run()
    {
        for(int i = 0; i>=0 ; i++)
        {
            System.out.println("damon");
            try
            {
                sleep(500);
            }
            catch(InterruptedException ex)
            {
                System.out.println(ex);
            }
        }
        System.out.println("end damon");
    }
}
```

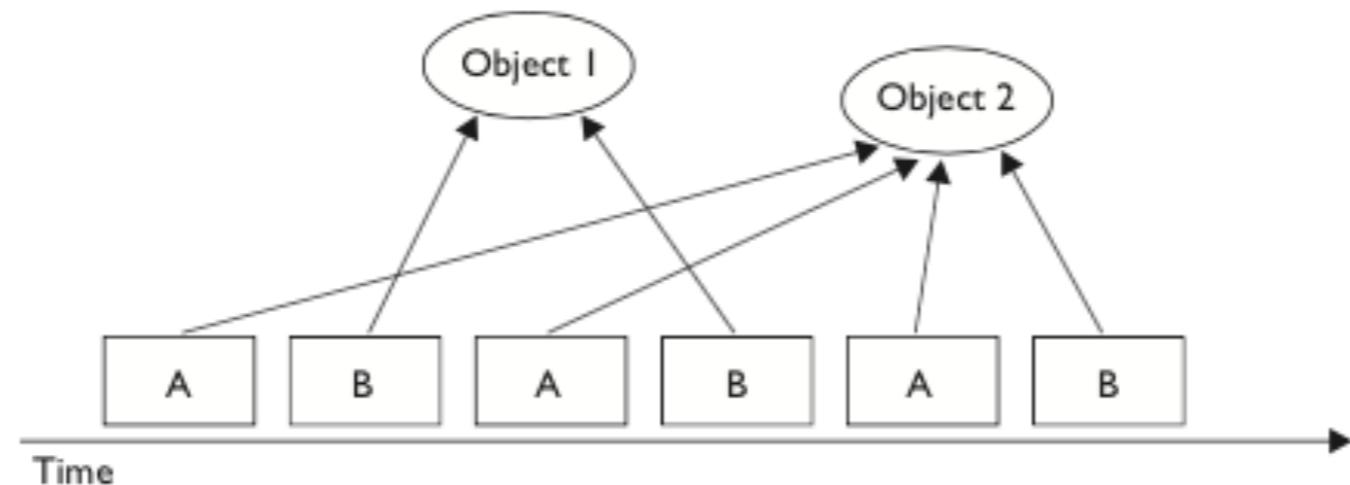
# NoDaemon

```
public class NoDaemon extends Thread
{
    @Override
    public void run()
    {
        for(int i = 0; i < 5; i++)
        {
            System.out.println("no damon");
            try
            {
                sleep(1000);
            }
            catch(InterruptedException ex)
            {
                ex.printStackTrace();
            }
        }
        System.out.println("end nodamon");
    }
}
```

# Main

```
public class Main
{
    public static void main(String[] args)
    {
        Daemon daemon = new Daemon();
        NoDaemon noDaemon = new NoDaemon();
        daemon.start();
        noDaemon.start();
        System.out.println("end main");
    }
}
```

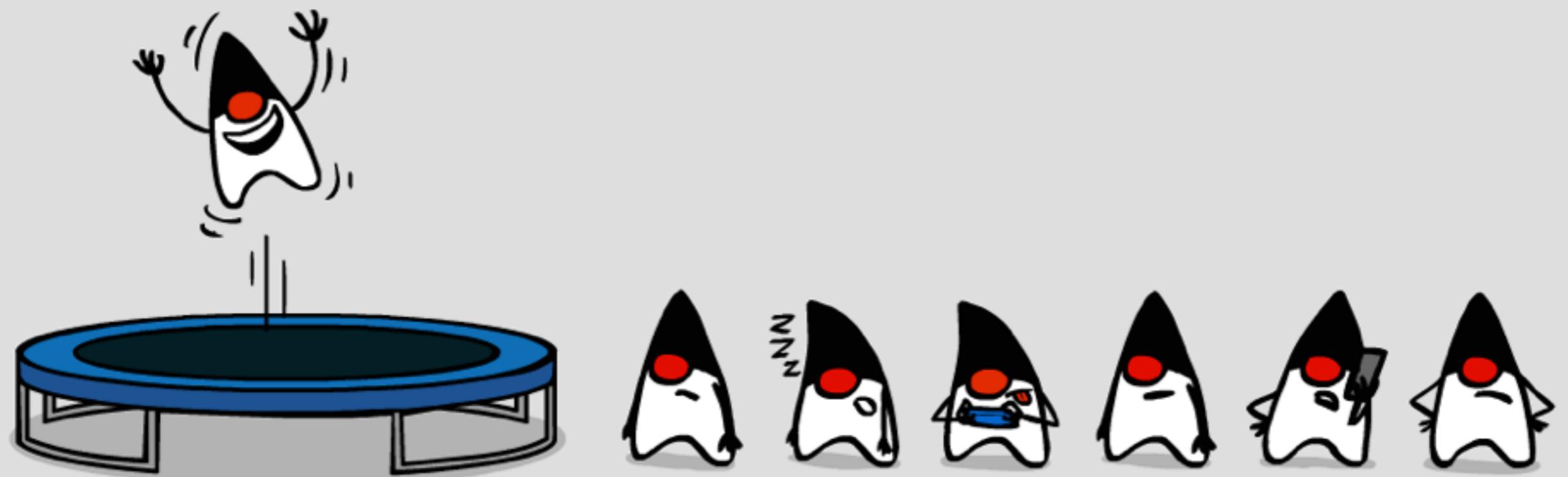
# Probleme beim parallelen Zugriff



Thread A will access Object 2 only

Thread B will access Object 1, and then Object 2

synchronized



# synchronized

- ❖ Mit synchronized kann man einen bestimmten Codebereich schützen oder auch eine ganze Methode. Im ersten Fall spricht man von einem synchronized-Block.

# synchronized

```
class Table {

    void printTable(int n) {
        // Mit synchronized kann man einen bestimmten Codebereich schützen
        // oder auch eine ganze Methode.
        // Im ersten Fall spricht man von einem synchronized-Block.
        // Dieser Bereich wird nun durch das angegebene Objekt geschützt.
        // Sehr oft ist dieses Objekt this,
        // da man den Codebereich des eigenen Objekts schützen will.
        synchronized (this) { // synchronized block
            for (int i = 1; i <= 5; i++) {
                System.out.println(n * i);
                try {
                    Thread.sleep(400);
                } catch (Exception e) {
                    System.out.println(e);
                }
            }
        }
    } // end of the method
}
```

# Deadlock



# Deadlock

```
public static Object lock1 = new Object();
public static Object lock2 = new Object();

public static void main(String args[]) {

    ThreadDemo1 T1 = new ThreadDemo1();
    ThreadDemo2 T2 = new ThreadDemo2();
    T1.start();
    T2.start();
}

private static class ThreadDemo1 extends Thread {
    public void run() {
        synchronized (lock1) {
            System.out.println("Thread 1: Holding lock 1...");
            try { Thread.sleep(10); }
            catch (InterruptedException e) {}
            System.out.println("Thread 1: Waiting for lock 2...");
            synchronized (lock2) {
                System.out.println("Thread 1: Holding lock 1 & 2...");
            }
        }
    }
}
private static class ThreadDemo2 extends Thread {
    public void run() {
        synchronized (lock2) {
            System.out.println("Thread 2: Holding lock 2...");
            try { Thread.sleep(10); }
            catch (InterruptedException e) {}
            System.out.println("Thread 2: Waiting for lock 1...");
            synchronized (lock1) {
                System.out.println("Thread 2: Holding lock 1 & 2...");
            }
        }
    }
}
```

# Deadlock Solution

```
public static Object Lock1 = new Object();
public static Object Lock2 = new Object();

public static void main(String args[]) {

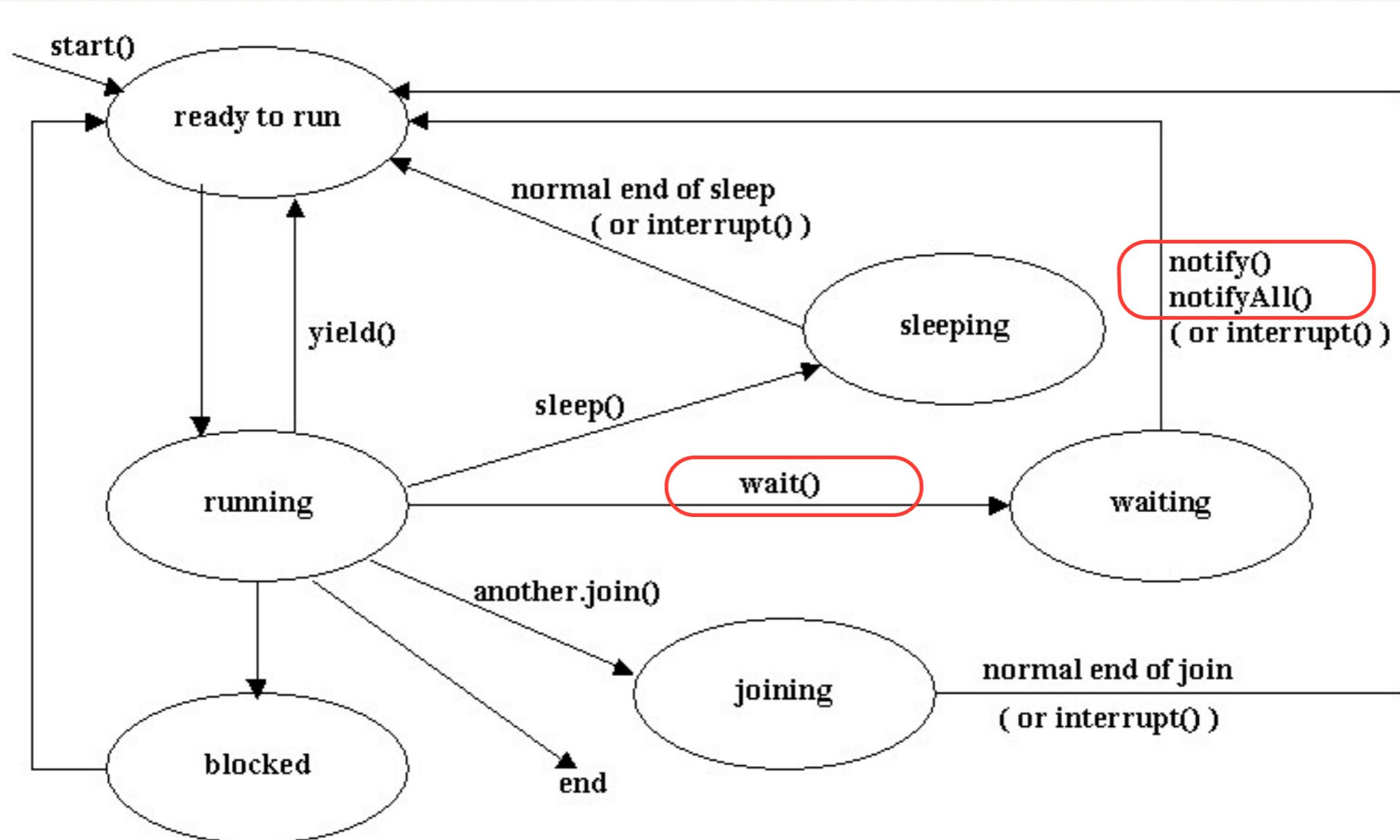
    ThreadDemo1 T1 = new ThreadDemo1();
    ThreadDemo2 T2 = new ThreadDemo2();
    T1.start();
    T2.start();
}

private static class ThreadDemo1 extends Thread {
    public void run() {
        synchronized (Lock1) {
            System.out.println("Thread 1: Holding lock 1...");
            try { Thread.sleep(10); }
            catch (InterruptedException e) {}
            System.out.println("Thread 1: Waiting for lock 2...");
            synchronized (Lock2) {
                System.out.println("Thread 1: Holding lock 1 & 2...");
            }
        }
    }
}
private static class ThreadDemo2 extends Thread {
    public void run() {
        synchronized (Lock1) {
            System.out.println("Thread 2: Holding lock 1...");
            try { Thread.sleep(10); }
            catch (InterruptedException e) {}
            System.out.println("Thread 2: Waiting for lock 2...");
            synchronized (Lock2) {
                System.out.println("Thread 2: Holding lock 1 & 2...");
            }
        }
    }
}
```

# Übung

- ❖ UebMultithreading.synchronize/Uebung1.txt

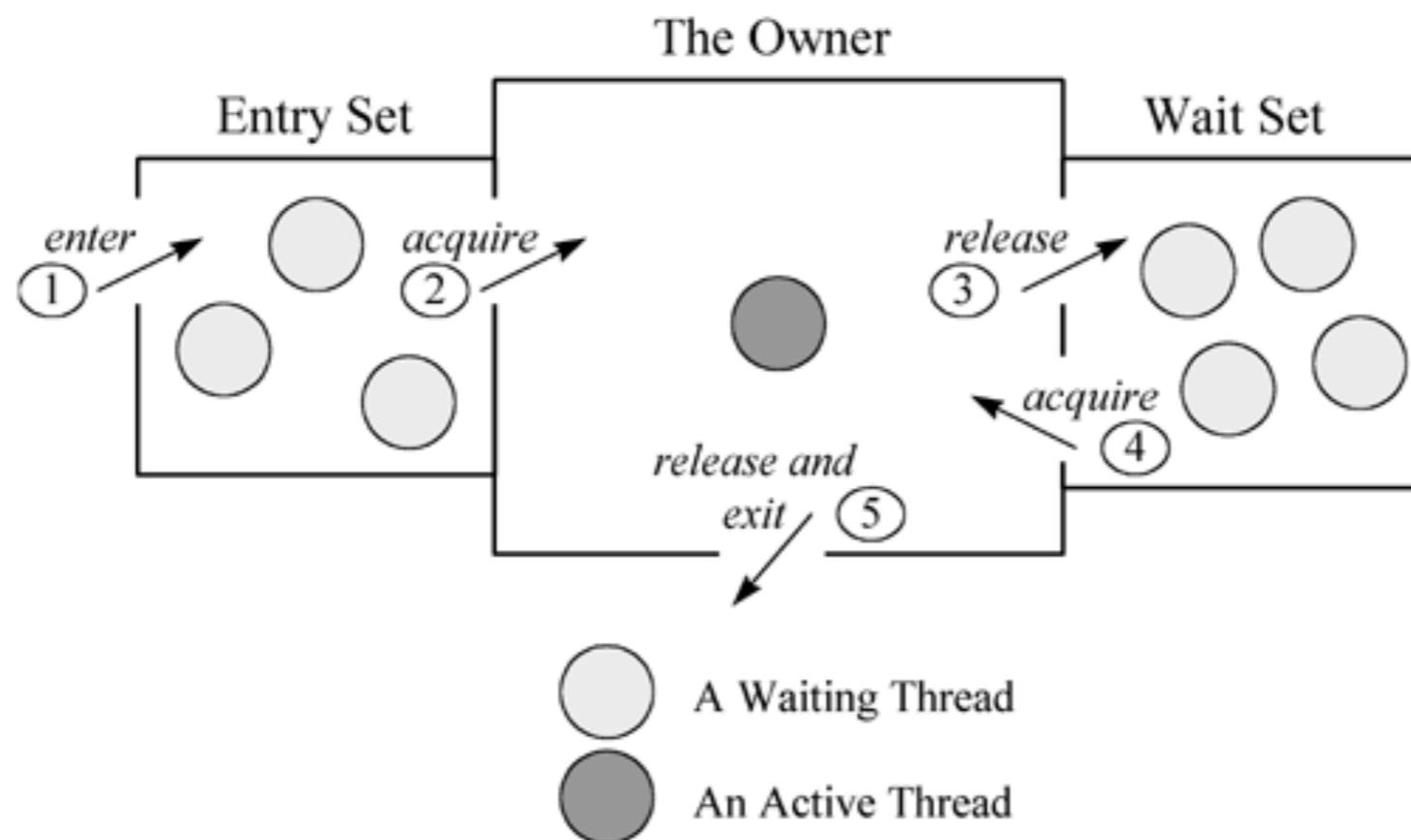
# Threadzustände



## wait & notify(Interprozesskommunikation)

- ❖ Methoden von Object
- ❖ können aber nur in synchronisierten Codeabschnitten aufgerufen werden
- ❖ mit wait geht ein Thread in einen Wartezustand
- ❖ mit notify bzw. notifyAll wird er beendet
- ❖ in Ausnahmefällen kann wait auch durch eine Interruptaufforderung beendet werden.
- ❖ wait gibt Lock zurück

# Interprozesskommunikation



# Interprozesskommunikation

- ❖ Eintritt des Thread, um einen Lock zu erwerben
- ❖ Der Lock ist zu einem Thread zugehörig
- ❖ Thread geht über in den Wartezustand, sobald wait() auf dem Objekt aufgerufen wird. Andernfalls gibt der Thread den Lock wieder ab.
- ❖ Sobald notify oder notifyAll aufgerufen wird, geht der Thread wieder in den runnable state.
- ❖ Nach Beendigung des Task, gibt der thread den Lock wieder frei und verlässt den Monitor state von dem Objekt.

```
public class WaiterThread1 extends Thread
{
    public WaiterThread1()
    {
    }

    public void run()
    {
        for(int i = 0; i < 3; i++)
        {
            System.out.println(new Date());
            goToSleep(1000);
        }
        synchronized(this)
        {
            try
            {
                System.out.println("waiter waits");
                this.wait();
            }
            catch(InterruptedException ex)
            {
                System.out.println(ex);
            }
        }
        System.out.println("waiter ended waiting");
        System.out.println(new Date());
    }

    private void goToSleep(int millis)
    {
        try
        {
            Thread.sleep(millis);
        }
        catch(InterruptedException ex)
        {
        }
    }
}
```

## MainWaiterThread1

```
public class MainWaiterThread1
{
    public static void main(String[] args)
    {
        WaiterThread1 waiter = new WaiterThread1();
        waiter.start();

        System.out.println("main goes to sleep");
        try
        {
            Thread.sleep(5000);
        }
        catch(InterruptedException ex)
        {
        }
        System.out.println("main ends sleeping, notifies waiter");

        synchronized(waiter)
        {
            waiter.notify();
        }
    }
}
```

# Waiter

```
public class MainWaiterThread {  
    public static void main(String[] args) {  
        Object ob = new Object();  
        //3 Waiter  
        WaiterThread wt1 = new WaiterThread(ob, 1, 3);  
        WaiterThread wt2 = new WaiterThread(ob, 2, 3);  
        WaiterThread wt3 = new WaiterThread(ob, 3, 3);  
        wt1.start();  
        wt2.start();  
        wt3.start();  
  
        try {  
            TimeUnit.SECONDS.sleep(5);  
        } catch (InterruptedException ex) {  
        }  
  
        synchronized (ob) {  
            //1 notify  
            ob.notify();  
            //ob.notifyAll();  
        }  
    }  
}
```

# WaiterThread

```
public class WaiterThread extends Thread {
    private final Object ob;
    private final int n; // nummer des Threads
    private final int count; // Gesamtzahl der Threads

    /**
     */
    public WaiterThread(Object ob, int n, int count) {
        this.ob = ob;
        this.n = n;
        this.count = count;
        this.setDaemon(true);
    }

    @Override
    public void run() {
        for (int i = 0; i < count; i++) {
            System.out.println("waiter-" + n + " " + new Date());
            try {
                //Thread.sleep(1000) vs. TimeUnit.SECONDS.sleep(1)
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException ex) {
            }
        }
        synchronized (ob) {
            try {
                System.out.println("waiter-" + n + " waits");
                ob.wait();
            } catch (InterruptedException ex) {
                System.out.println(ex);
            }
        }
        System.out.println("waiter-" + n + " ends waiting");
        System.out.println(new Date());
    }
}
```

# MainWaiter

```
public class MainWaiterThread {
    public static void main(String[] args) {
        Object ob = new Object();
        //3 Waiter
        WaiterThread wt1 = new WaiterThread(ob, 1, 3);
        WaiterThread wt2 = new WaiterThread(ob, 2, 3);
        WaiterThread wt3 = new WaiterThread(ob, 3, 3);
        wt1.start();
        wt2.start();
        wt3.start();

        try {
            TimeUnit.SECONDS.sleep(5);
        } catch (InterruptedException ex) {
        }

        synchronized (ob) {
            //1 notify
            ob.notify();
        }
    }
}
```

# Übung

- ❖ UebMultithreading.waitnotify / Uebung1.txt

## Nachteile beim Thread Runnable

- ❖ Schon beim Erzeugen eines Thread-Objektes muss das Runnable-Objekt im Thread Konstruktion übergeben werden.
- ❖ Wird start() auf dem Thread-Objekt zweimal aufgerufen, so führt der zweite Aufruf zu einer Ausnahme. Immer neuer Thread nötig.
- ❖ Thread beginnt mit der Abarbeitung des Programmcodes vom Runnable direkt nach dem Aufruf von start(). Keine spätere Ausführung.

Wünschenswert wäre eine Abstraktion, die das Ausführen des Runnable-Programmcodes von der technischen Realisierung(etwa den Threads) trennt.

# Interface Callable

- ❖ call Methode bietet mehr Möglichkeiten als run, da sie einen Returnwert hat und eine Exception werfen kann.
- ❖ Der Returntyp der Methode call() wird über den generischen Typ des Interfaces Callable festgelegt.

# Interface Callable

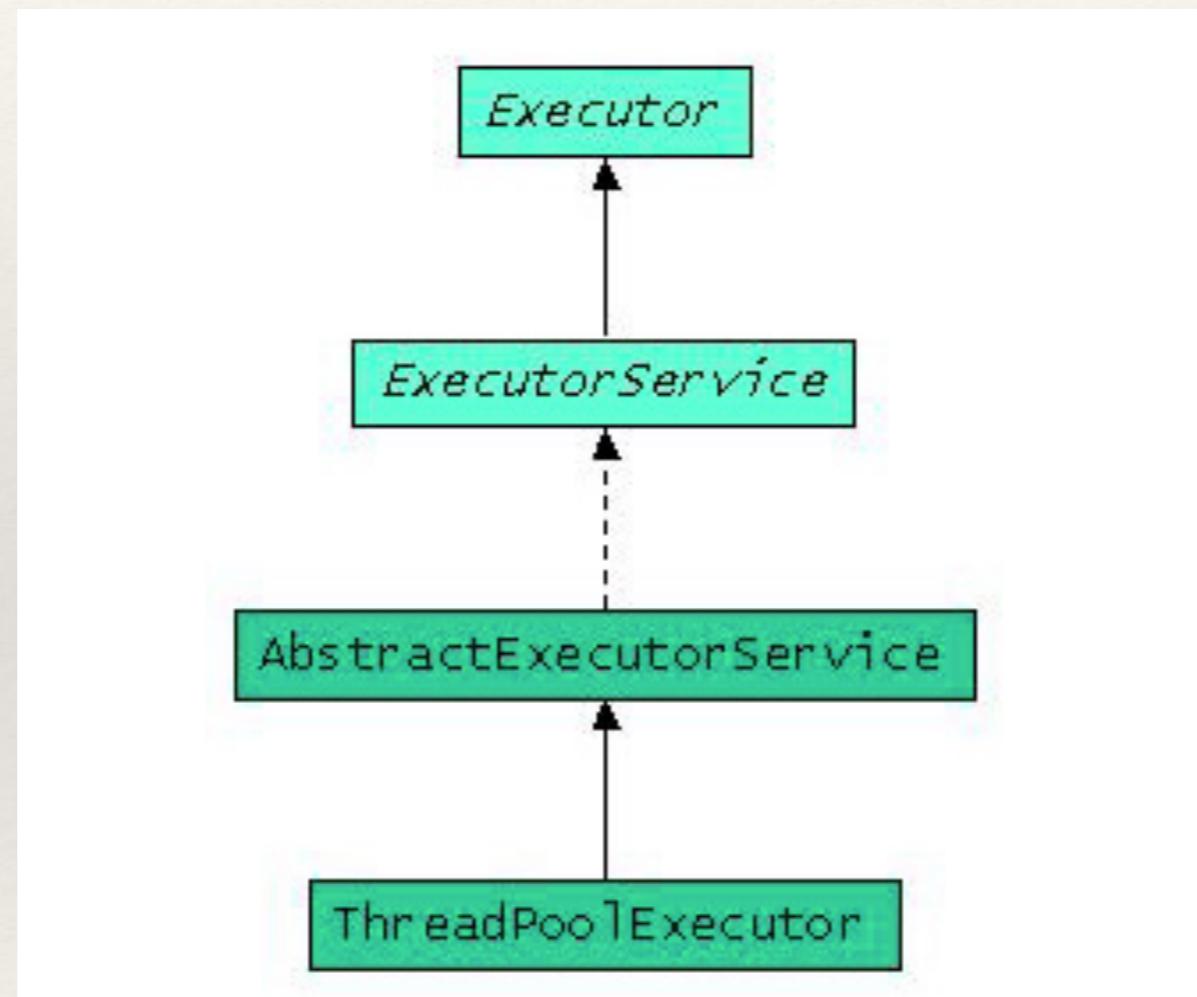
## **java.util.concurrent.Callable<V>**

Returntyp	Name der Methode
V	<b>call()</b> Computes a result, or throws an exception if unable to do so.

# ExecutorService

- ❖ ein Callable kann nur mit Hilfe eines ExecutorService starten.
- ❖ Ein ExecutorService stellt eine Methode submit(Callable<T> task) bereit. -> Start des Callable
- ❖ submit gibt Future<T> zurück
- ❖ dieser Typ besitzt u.a eine Methode get() und genau diese Methode liefert uns den Returnwert von call().
- ❖ Nach dem Absetzen von submit() kann man das Callable nicht mehr beeinflussen.

# Executor



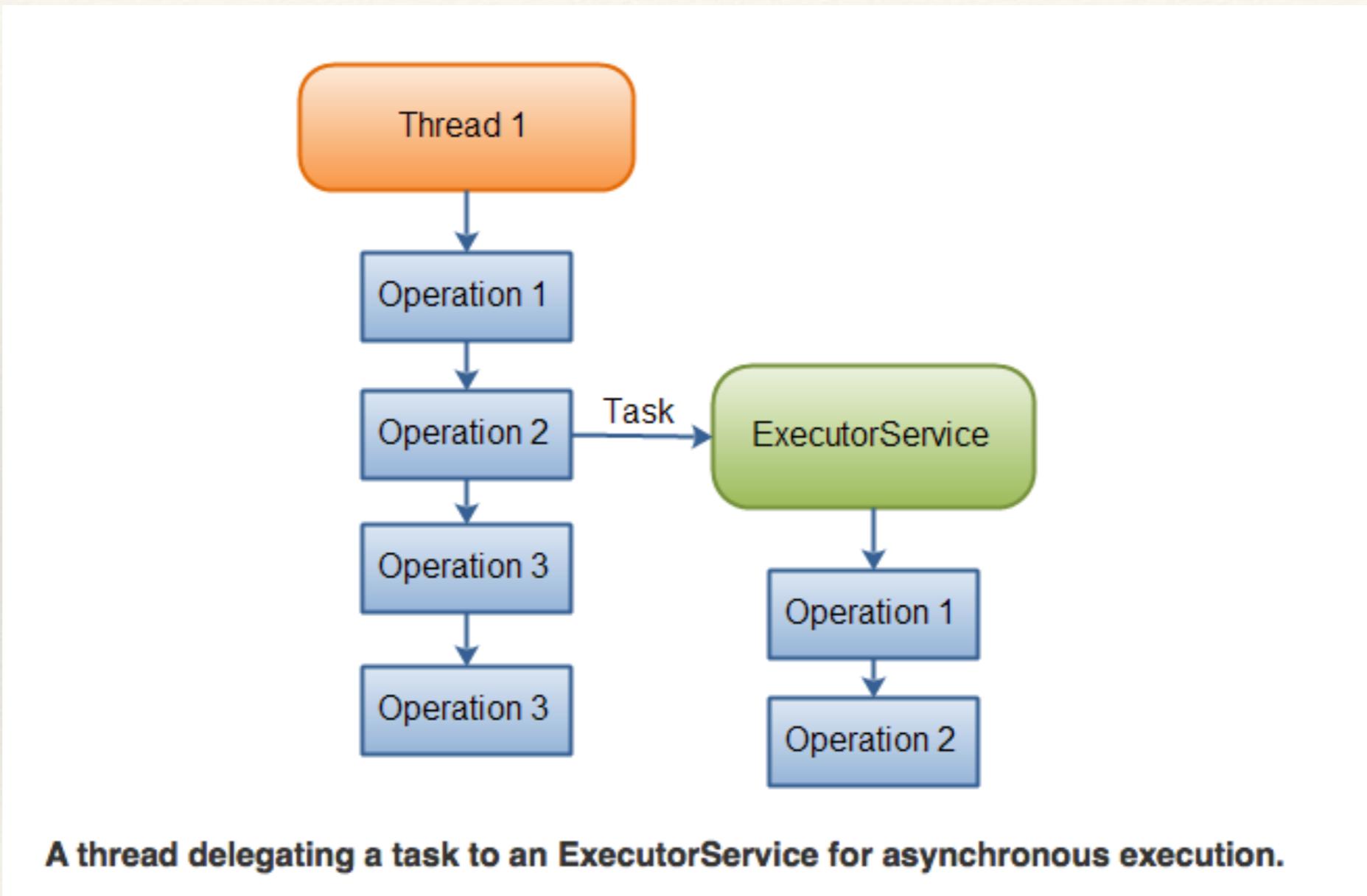
# Interface Executor

```
public interface Executor {  
  
    /**  
     * Executes the given command at some time in the future. The command  
     * may execute in a new thread, in a pooled thread, or in the calling  
     * thread, at the discretion of the {@code Executor} implementation.  
     *  
     * @param command the runnable task  
     * @throws RejectedExecutionException if this task cannot be  
     * accepted for execution  
     * @throws NullPointerException if command is null  
     */  
    void execute(Runnable command);  
}
```

# ExecutorService

- ❖ Das Interface ExecutorService bietet also zum einen Service Management Funktionalität an, um den Service zu beenden, seinen Status abzufragen, usw. Die Namen der Methoden sind dabei selbsterklärend: shutdown(), shutdownNow(), awaitTermination(), isShutdown(), isTerminated().

# ExecutorService



## Future<V>

- ❖ wann ist eine asynchrone Tätigkeit(Download in Swing) fertig.
- ❖ Future Pattern übernimmt Synchronisation zwischen Thread und Ergebnisübergabe
  - ❖ V get(), isDone(), isCancelled()

# FutureTask<V>

```
public class HelloWorldCount implements Callable<Integer> {
    public Integer call() throws InterruptedException {
        long end = new Date().getTime() + 1000;

        int i;
        for (i = 0; new Date().getTime() < end; i++) {
            System.out.println("HelloWorld! ");
            if (Thread.interrupted())
                throw new InterruptedException();
        }
        return new Integer(i);
    }

    static public void main(String[] argv) {
        FutureTask<Integer> t = new FutureTask<Integer>(new HelloWorldCount()); // 1
        new Thread(t).start(); // 2

        try {
            System.out.println("Die Anzahl war: " + t.get());
        } catch (ExecutionException e) {
            Throwable th = e.getCause();
            if (th == null)
                System.out.println("HelloWorldCount wurde aus unbekannten Gründen abgebrochen. ");
            else if (th instanceof InterruptedException)
                System.out.println("HelloWorldCount wurde abgebrochen. ");
            else
                System.out.println("HelloWorldCount wurde mit folgendem Fehler: " + th + " abgebrochen. ");
        } catch (InterruptedException e) {
            System.out.println("HelloWorldCount wurde abgebrochen. ");
        }
    }
}
```

## Callable example

```
public class MyCallable implements Callable<Long>{
    @Override
    public Long call() throws Exception {
        long sum = 0;
        for (long i = 0; i <= 100; i++) {
            sum += i;
        }
        return sum;
    }

}
```

## FutureTask<V>

- ❖ implementiert Runnable und Future<V>
- ❖ Vorstellbar als ein intelligenter Wrapper für ein Runnable oder ein Callable

## FutureTask<V>

- ❖ Konstruieren lässt sich eine FutureTask<V> entweder mit einem Callable<V>, dessen Ergebnis sie beim Future<V>.get() zurückliefert, oder mit einem Runnable und einem vordefinierten Ergebniswert beliebigen Referenztyps, welcher beim Future<T>.get() zurückliefert wird.
  - ❖ FutureTask(Callable<V> callable)
  - ❖ FutureTask(Runnable runnable, V result) - da Runnable kein Ergebnis

```
public class CallableFutures {
    private static final int NTHREDS = 10;

    public static void main(String[] args) {

        ExecutorService executor = Executors.newFixedThreadPool(NTHREDS);
        List<Future<Long>> list = new ArrayList<Future<Long>>();
        for (int i = 0; i < 20000; i++) {
            Callable<Long> worker = new MyCallable();
            Future<Long> submit = executor.submit(worker);
            list.add(submit);
        }
        long sum = 0;
        System.out.println("Size:"+list.size());
        // now retrieve the result
        for (Future<Long> future : list) {
            try {
                // Wartet, bis die asynchrone Tätigkeit abgeschlossen ist,
                // und gibt dann ihr Ergebnis zurück.
                // Dabei ist V der Typparameter des generischen Interface Future<V>.
                // Das heißt, es wird ein Ergebnisse vom einem beliebigen
                // Referenztypen zurückgeliefert.
                // Falls die Tätigkeit mit interrupt() abgebrochen wurde,
                // wird die InterruptedException geworfen.
                // Falls die Ausführung der Tätigkeit mit einer Exception abgebrochen
                // wurde, wird die ExecutionException geworfen.
                // Sie enthält die Orginal-Exception, die zum Abbruch geführt hat.
                sum += future.get();
            } catch (InterruptedException e) {
                e.printStackTrace();
            } catch (ExecutionException e) {
                e.printStackTrace();
            }
        }
        System.out.println("Summe:"+sum);
        executor.shutdown();
    }
}
```

# Übung

- ❖ UebMultithreading.callable/Uebung1.txt

# Lock

- ❖ Ein Thread sperrt ALLE synchronized Methoden, sobald er eine synchronized Methode abarbeitet.
- ❖ pro Object gibt es nur ein Lock
- ❖ sobald der Monitor diesen einen Lock vergeben, kann zwangsläufig kein anderer Thread mehr auf synchronized-Methoden oder Blöcke dieses Objektes zugreifen.
- ❖ Mit der im Package concurrent eingeführten Klasse Lock kann man sozusagen lokale Locks bzw. Monitore einrichten, die etwa nur eine Methode locken oder auch nur bestimmte Codeteile. Andere Methoden oder Codeteile sind dann von diesem Lock nicht betroffen.

# Lock

```
public class TableLock {  
    /*  
     * Lock ist ein Interface. Die Standardimplementierung ist bislang die  
     * Klasse ReentrantLock. Da Lock aber ein Interfaces ist, kann man eigene  
     * Implementierungen nach speziellen Bedürfnissen schreiben. Die Methoden  
     * des Interfaces bieten zudem mehr Flexibilität als das Schlüsselwort  
     * synchronized.  
     */  
    private Lock lock = new ReentrantLock();  
  
    void printTable(int n) {  
        /*  
         * Mit tryLock kann man falls kein Lock erhältlich ist,  
         * in der Zwischenzeit andere Aufgaben erledigen.  
         * Im Gegensatz zu lock() blockiert tryLock()  
         * in keinem Fall (siehe obigen API-Auszug).  
         */  
        // if (lock.tryLock()) {  
        try {  
            lock.lock();  
            for (int i = 1; i <= 5; i++) {  
                System.out.println(n * i);  
                try {  
                    Thread.sleep(400);  
                } catch (Exception e) {  
                    System.out.println(e);  
                }  
            }  
        } finally {  
            lock.unlock();  
        }  
        // }  
    }  
}/// end of the method
```

# WaiterThread

```
public class WaiterThread extends Thread {  
    private Lock lock;  
    private Condition condition;  
  
    public WaiterThread(Lock lock, Condition condition){  
        this.lock = lock;  
        this.condition = condition;  
    }  
  
    public void run(){  
        for(int i = 0; i < 3; i++){  
            System.out.println(new Date());  
            goToSleep(1000);  
        }  
        lock.lock();  
        try{  
            System.out.println("waiter waits");  
            condition.await();  
        }  
        catch(InterruptedException ex) {  
            System.out.println(ex);  
        }  
        finally{  
            lock.unlock();  
        }  
        System.out.println("waiter ended waiting");  
        System.out.println(new Date());  
    }  
  
    private void goToSleep(int millis){  
        try{  
            Thread.sleep(millis);}  
        catch(InterruptedException ex)  
        {}  
    }  
}
```

# Main

```
public class Main
{
    public static void main(String[] args)
    {
        Lock lock = new ReentrantLock();
        Condition condition = lock.newCondition();

        WaiterThread waiter = new WaiterThread(lock, condition); // waiter.start();

        System.out.println("main goes to sleep");
        try
        {
            TimeUnit.SECONDS.sleep(5);
        }
        catch(InterruptedException ex)
        {
        }
        System.out.println("main ends sleeping, signals waiter");
        lock.lock();
        try
        {
            condition.signal();
        }
        finally
        {
            lock.unlock();
        }
    }
}
```