

Multi-Threading

Thread Definition sprachlich :

Kommt aus dem Englischen und heißt Faden oder Strang

Thread Definition Wissenschaft :

Ausführungsstrang

Thread Definition in seiner Bedeutung :

Ein Thread ist ein Teil eines Programms, der eigenständig läuft, während der Rest des Programms etwas anderes tut. Dies nennt man auch Multitasking, da das Programm mehr als eine Aufgabe (Task) zur selben Zeit ausführen kann.

Zeitgleich werden mehrere Anweisungen abgearbeitet

WANN SETZT MAN THREADS EIN ?

Threads sind ideal für alles was viel Rechenzeit in Anspruch nimmt und kontinuierlich ausgeführt wird, wie unser Bomberman-Projekt, Wo halt viele Grafiken geladen, gelöscht, verschoben und neugezeichnet werden

Wenn man also die Arbeitslast auf viele Threads packt wird die Arbeit insgesamt schneller und leichter von statten gehen.

Insbesondere muss es aussehen als ob alles gleichzeitig passiert, naja Wie würde das den aussehen wenn Bomberman die Ausführungen alle Hintereinander (sequentiell) abarbeiten würde ?

WIE SCHREIBT MAN EIN PROGRAMM MIT THREADS ?

Man implementiert Threads in Java mithilfe der Klasse Thread.

Die einfachste Anwendung von Threads besteht darin die Ausführung eines Programms anzuhalten und für eine bestimmte Zeit pausieren zu lassen.

Dazu ruft man die Methode *sleep(long)* auf. Die Dauer wird in Millisekunden als Argument übergeben. Diese Methode wirft eine Ausnahme.

```
try {  
    Thread.sleep(3000);  
} catch (InterruptedException ie) {  
    // Man braucht hier nichts zu tun  
}
```

WIE KANN EINE KLASSE THREADS BENUTZEN ?

Die Klasse die Threads benutzen soll muss eine Schnittstelle Runnable Implementieren.

Dies macht man indem man hinter die Klasse“ implements Runnable“ schreibt.

```
public class KlassenName implements Runnable {  
    public void run() {  
        // ...  
    }  
}
```

Wenn eine Klasse eine Schnittstelle implementiert müssen wir auch all ihre Methoden implementieren , deshalb die Methode run().
Jetzt muss man eine Referenz auf ein Objekt der Thread-Klasse erzeugen.

WIE ERZEUGT MAN EINEN THREAD ?

Threads werden erzeugt, indem der Konstruktor *Thread(Object)* aufgerufen wird, wobei das Objekt übergeben wird, das in einem Thread ablaufen soll.

```
public class KlassenName {  
    public static void main(String[] args) {  
        MyTask t = new MyTask();  
        Thread thread = new Thread(t);  
        . . .  
        . . .  
        . . .  
    }  
}
```

MyTask ist unsere Klasse die wir in einem Thread laufen lassen möchten und unsere Instanz t schieben wir in den Thread Konstruktor.

WANN BEGINNT EIN THREAD ?

Ein Thread beginnt, wenn seine *start()*-Methode aufgerufen wird.

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

Der Aufruf der *start()*-Methode eines Threads führt zum Aufruf einer weiteren Methode – der *run()*-Methode, die in dem nebenläufigen Objekt präsent sein muss.

Die *run()*-Methode ist das Herz einer nebenläufigen Klasse. In unserem Bomberman würden in ihr z. B. die Veränderungen stehen, die das betreffen, was die Paint- Methode zeichnet.

WIE KANN MAN EINEN THREAD SCHLIESSEN ?

Einen Thread schließt man mit der *stop()*-Methode

thread.stop();

Oder man schreibt sich selber eine solche Methode.

Dies ist sogar besser, da die *stop()*-Methode veraltet ist und Fehler verursachen kann. Mehr zur *stop()*-Methode in der folgenden API-doc .

[org.w3c.dom.cs](#)
[org.w3c.dom.ls](#)
[org.xml.sax](#)
[org.xml.sax.ext](#)
[org.xml.sax.helpers](#)

[TextLayout](#)
[TextListener](#)
[TextMeasurer](#)
[TextOutputCallback](#)
[TextSyntax](#)
[TextUI](#)
[TexturePaint](#)
[Thread](#)
[Thread.State](#)
[Thread.UncaughtExceptionHandler](#)
[THREAD POLICY](#)
[ThreadDeath](#)
[ThreadFactory](#)
[ThreadGroup](#)
[ThreadInfo](#)
[ThreadLocal](#)
[ThreadMXBean](#)
[ThreadPolicy](#)

stop

[@Deprecated](#)

```
public final void stop()
```

Deprecated. This method is inherently unsafe. Stopping a thread with `Thread.stop` causes it to unlock all of the monitors that it has locked (as a natural consequence of the unchecked `ThreadDeath` exception propagating up the stack). If any of the objects previously protected by these monitors were in an inconsistent state, the damaged objects become visible to other threads, potentially resulting in arbitrary behavior. Many uses of `stop` should be replaced by code that simply modifies some variable to indicate that the target thread should stop running. The target thread should check this variable regularly, and return from its `run` method in an orderly fashion if the variable indicates that it is to stop running. If the target thread waits for long periods (on a condition variable, for example), the `interrupt` method should be used to interrupt the wait. For more information, see [Why are Thread.stop, Thread.suspend and Thread.resume Deprecated?](#).

Also diese sollten wir nicht verwenden deswegen gehen wir nicht weiter darauf ein.

Wenn wir schon bei Api-doc sind hier die Seite in der man sich das anschauen kann

<http://docs.oracle.com/javase/6/docs/api/>

```
public class MyTask extends Thread{

    @Override
    public void run() {
        long start = System.currentTimeMillis();

        int bestMove = Integer.MIN_VALUE;
        Random random = new Random();
        for (int i = 1; i <= 100; i++) {
            for (int j = 0; j < 8000; j++) {
                for (int k = 0; k < 8000; k++) {
                    int move = random.nextInt();
                    if (move > bestMove) {
                        bestMove = move;
                    }
                }
            }
            System.out.println("Thinking... " + i + "%");
        }
    }
}
```

Die Klasse MyTask erbt von Thread und die beinhaltet die Methode run. diese müssen wir implementieren. In der Methode sind 3 Schleifen. Die Äußere zeigt den Fortschritt in % und in der innersten wird eine zufällige Zahl in move gespeichert anschl. wird die höchste der Zahlen in bestMove gespeichert. Der Time-Stamp berechnet uns die Dauer in Millisekunden.

```
public class MyTask extends Thread{

    @Override
    public void run() {
        long start = System.currentTimeMillis();

        int bestMove = Integer.MIN_VALUE;
        Random random = new Random();
        for (int i = 1; i <= 100; i++) {
            for (int j = 0; j < 8000; j++) {
                for (int k = 0; k < 8000; k++) {
                    int move = random.nextInt();
                    if (move > bestMove) {
                        bestMove = move;
                    }
                }
            }
            System.out.println("Thinking... " + i + "%");
        }

        long stop = System.currentTimeMillis();
        long time = stop - start;
        System.out.println(getName() + " FINISHED AFTER " + time / 1000.
            + " seconds.");
    }
}
```

Das lassen wir uns dann ausgeben. Soviel zur run-Methode jetzt benötigen wir einen Instanz davon.

Hier erzeugen wir nun unsere Instanz und starten den Thread mit t.start(). Start ist auch von Thread geerbt.

```
public class MultiThreadingDemo {  
  
    public static void main(String[] args) {  
        MyTask t = new MyTask();  
        t.start();  
    }  
}
```

Vorsicht wenn wir t.run() schreiben würden dann haben wir keinen Thraed erzeugt sondern einen normalen Methoden-Aufruf. Der Compiler würde dann ganz normal zur Methode run gehen und abarbeiten.

<http://download.oracle.com/javase/6/docs/api/>

start

```
public void start()
```

Causes this thread to begin execution; the Java Virtual Machine calls the `run` method of this thread.

The result is that two threads are running concurrently: the current thread (which returns from the call to the `start` method) and the other thread (which executes its `run` method).

It is never legal to start a thread more than once. In particular, a thread may not be restarted once it has completed execution.

Throws:

[IllegalThreadStateException](#) - if the thread was already started.

See Also:

[run\(\)](#), [stop\(\)](#)

Diese Methode ruft die `run` Methode vom Thread auf und schließt sich dann.

Bei **Throws** sehen wir das wir in unserem Code eine Exception behandeln müssen.

Damit wir jetzt sehen das mehrere Threads „parallel“ laufen müssen wir noch im Main-Thread auch noch eine Ausgabe machen.

```
public class MultiThreadingDemo {  
  
    public static void main(String[] args) {  
        MyTask t = new MyTask();  
        t.start();  
  
        while(true){  
            System.out.println("MAIN-THREAD LÄUFT...");  
        }  
    }  
}
```

Hier wird jetzt endlos ein Text ausgegeben.

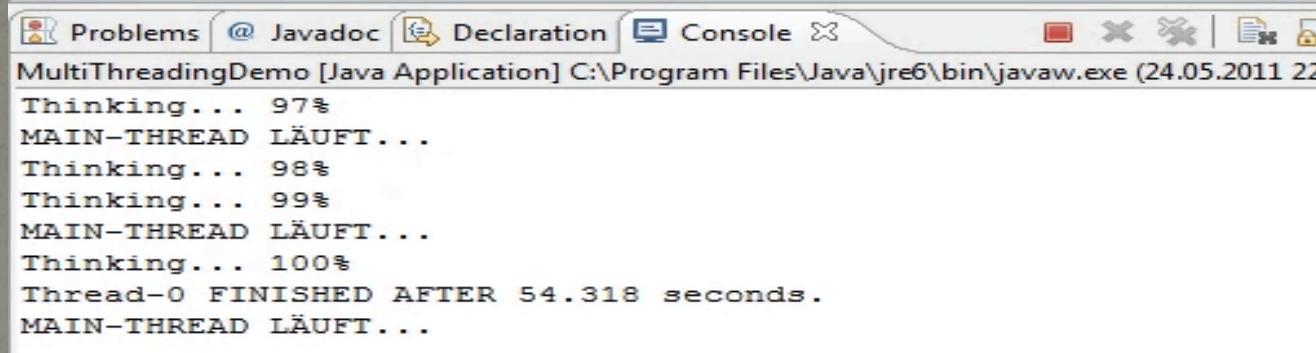
Jetzt haben einen Main-Thread der die Konsole mit der Ausgabe zu bombt.
Deshalb machen wir nach jeder Ausgabe 1 sec. Pause

```
public class MultiThreadingDemo {  
  
    public static void main(String[] args) {  
        MyTask t = new MyTask();  
        t.start();  
  
        while(true){  
            System.out.println("MAIN-THREAD LÄUFT...");  
            try {  
                Thread.sleep(1000);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

Diese Pause realisiert man mit der sleep Methode und für die müssen wir eine Exception behandeln.

So jetzt können wir das Ganze starten.

```
public class MultiThreadingDemo {  
    public static void main(String[] args) {  
        MyTask t = new MyTask();  
        t.start();  
  
        while(true){  
            System.out.println("MAIN-THREAD LÄUFT...");  
            try {  
                Thread.sleep(1000);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```



The screenshot shows an IDE interface with a code editor and a console window. The code editor contains the Java code for `MultiThreadingDemo`. The console window displays the execution of the program, showing two threads: the main thread and a `MyTask` thread. The main thread prints "MAIN-THREAD LÄUFT..." every second. The `MyTask` thread prints "Thinking..." and its progress percentage (97%, 98%, 99%, 100%). After the task is completed, it prints "Thread-0 FINISHED AFTER 54.318 seconds." and continues to print "MAIN-THREAD LÄUFT...".

```
Problems @ Javadoc Declaration Console  
MultiThreadingDemo [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (24.05.2011 22:55)  
Thinking... 97%  
MAIN-THREAD LÄUFT...  
Thinking... 98%  
Thinking... 99%  
Thinking... 100%  
MAIN-THREAD LÄUFT...  
Thread-0 FINISHED AFTER 54.318 seconds.  
MAIN-THREAD LÄUFT...
```

Und hier sehen wir einen Auszug wie die zwei Threads laufen. Main-Thread wird ausgeführt (while) dann pausiert er und dann läuft unser MyTask Thread. Man sieht es ist nicht Sequentiell.

```
public class MyTask implements Runnable{

    @Override
    public void run(){
        long start = System.currentTimeMillis();

        int bestMove = Integer.MIN_VALUE;
        Random random = new Random();
        for (int i = 1; i <= 100; i++) {
            for (int j = 0; j < 8000; j++) {
                for (int k = 0; k < 8000; k++) {
                    int move = random.nextInt();
                    if (move > bestMove) {
                        bestMove = move;
                    }
                }
            }
        }
        System.out.println("Thinking... " + i + "%");
    }
}
```

Hier sehen wir eine zweite Möglichkeit die run Methode zu Implementieren. Statt von Thread zu erben implementiert man Das Interface Runnable .

```
public class MultiThreadingDemo {  
  
    public static void main(String[] args) {  
        MyTask t = new MyTask();  
        Thread thread = new Thread(t);  
        thread.start();  
  
        while(true){  
            System.out.println("MAIN-THREAD LÄUFT...");  
            try {  
                Thread.sleep(1000);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

Dann muss man die start Methode ändern da wir ja nicht mehr von Thread Erben.

Stattdessen erzeugen wir ein Objekt von Thread und rufen einen Konstruktor auf , dem Runnable übergeben wird.

Dann rufen wir auf der Instanz die start Methode auf.

Threads stoppen

Es gibt 2 Arten

1.

Threads kann man stoppen dazu gibt es den stop Aufruf .
Würde an unserem Beispiel so aussehen

thread.stop();

2.

Oder man schreibt eine eigene Methode die an einem
Bestimmten Platz einen thread stoppt.

```

private boolean alive;

@Override
public void run(){
    alive = true;
    long start = System.currentTimeMillis();

    int bestMove = Integer.MIN_VALUE;
    Random random = new Random();
    for (int i = 1; i <= 100; i++) {
        for (int j = 0; j < 8000; j++) {
            for (int k = 0; k < 8000; k++) {
                if(!alive){
                    System.out.println("Stopping!");
                    return;
                }
                int move = random.nextInt();
                if (move > bestMove) {
                    bestMove = move;
                }
            }
        }
        System.out.println("Thinking... " + i + "%");
    }

    long stop = System.currentTimeMillis();
    long time = stop - start;
    System.out.println(Thread.currentThread().getName() + " FINISHED AFTER "
        + " seconds.");
}

public void cancel(){
    alive = false;
}

```

Die 2te Möglichkeit ist die richtige und die Erste ist die Falsche, da diese wie wir zu Anfang gesehen haben veraltet ist.
Hier können wir die cancel Methode aufrufen und dies setzt alive auf false und in der innersten Schleife wird in der if-Abfarge rausgesprungen

ANNOMALIEN

Das sind Probleme die die Multithreading entstehen können, da mehrere Threads sich die selben Daten teilen.

Threads haben einen Lokalen Speicher auf dem sie ihr Arbeit auf Daten ausführen, die sie vom Globalen Speicher haben.

So kann es passieren das die Daten die im Globale Speicher sind Nicht mehr aktuell sind, da die aktuellen im Thread sind.

Man muss also denn Thread zwingen die Daten in den Globalen Speicher zu packen. Somit hätten andere Threads Zugang auf die aktuellen Daten .

Dies geschieht mit einem Keyword namens
synchronized() {...}

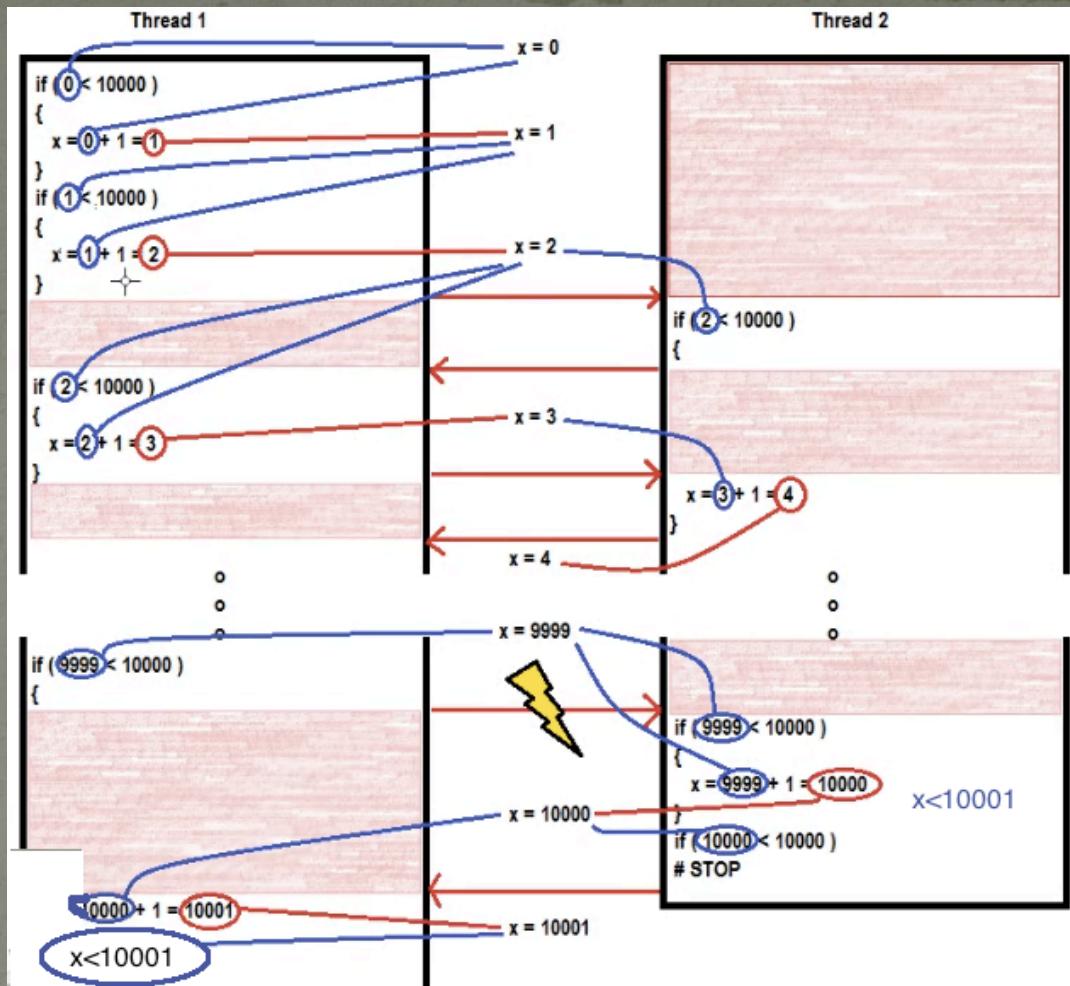
In die runden Klammern kommt eine statische Variable eines Objekts. Dieser funktioniert wie ein Schlüssel. Der Thread der ihn hat darf arbeiten

und nachdem die Daten aktualisiert wurden, bekommt ein anderer den Schlüssel ...

DEADLOCK

Deadlock können durch Synchronisationen entstehen.
Deshalb sollte man nicht willkürlich synchronized benutzen.

Als anschauliches Beispiel wäre der Strassenverkehr.
Um Unfälle zu vermeiden gibt es eine Rechts vor Links Regel.
Diese wäre dann unser synchronized.
Wenn aber an einer Kreuzung von jeder Seide ein Fahrzeug kommt,
Muss jeder warten bis der von Rechts gefahren ist. Das wäre dann
Unser Deadlock.



Ein Beispiel für 2 Threads die sich die Variable x teilen. Das Problem ist das das Betriebssystem entscheidet wer arbeiten und wer pausieren muss.