Nebenläufigkeit: Zustände eines Theads

- Die verschiedenen Zustände eines Threads kennen und konkret abfragen können
- Wissen wie ein Thread sein Ende findet
- Auf das vorzeitige Abbrechen eines Threads reagieren können
- Einen laufenden Thread konsistent abbrechen können
- Wissen wie ein Thread auf das Ende eines anderen Threads warten kann
- Erkennen dass beim fertigen Abarbeiten des Hauptthreads die von ihm erstellten Threads noch weiter laufen
- Den Thread als Dämon definieren und die Eigenschaft eines Dämons ausnutzen können

Zustände eines Threads

NEW	Neuer Thread, noch nicht gestartet
RUNNABLE	Läuft in der JVM
BLOCKED	Wartet auf einen Monitor-Lock, wenn er etwa einen *synchronized Block* betreten möchte[1]
WAITING	Wartet auf ein `notify()`[1]
TIMED_WAITING	Wartet in einem `sleep()`
TERMINATED	Ausführung beendet

`Thread.State getState()` liefert den Zustand

`boolean` `isAlive()` `true` falls Thread in `run()`

```
static void sleep(long millis)
    throws InterruptedException
```

Thread wird mindestens für `millis` schlafen gelegt. Unterbricht anderer Thread den schlafenden ➔ Exception (siehe unten)

[1] Siehe nächstes Kapitel

Das Ende eines Threads

run() wurde ohne Fehler beendet

In run() tritt ein RuntimeException auf, welches Methode beendet. Andere Threads und VM laufen aber weiter

Thread wird von außen durch stop() radikal abgebrochen. run() wird abgebrochen **PROBLEMATISCH!!!**

VM wird bendet und nimmt alle Threads mit ins Grab. ■-Knopf in Eclipse

Abbruch eines Threads erkennen und darauf reagieren

Tritt bei Threadabarbeitung RuntimeException auf, wird UncaughtExceptionHandler darüber informiert, der bei Thread *registriert* wird

```
Thread t = new Thread() {
  @Override
  public void run() {
    throw new RuntimeException("Thread stopped");
  }
};

t.setUncaughtExceptionHandler(
  new UncaughtExceptionHandler() {
    @Override
    public void uncaughtException(Thread t, Throwable e) {
      System.out.println(t.getName() + " stopped with message " +
        e.getMessage());
    }
  }
);

t.start();
```

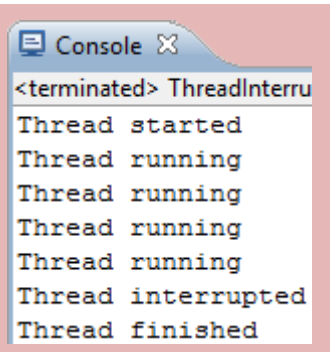
Exception-Handler für alle Threads

```
static void setDefaultUncaughtExceptionHandler(
  Thread.UncaughtExceptionHandler eh)
```

Thread mit `interrupt()` von außen bitten damit er sich beendet

`interrupt()`
setzt von außen in einem Thread ein internes Flag, das in `run()` durch `isInterrupted()` abgefragt werden kann

```
public class ThreadInterrupt extends Thread
{
  @Override
  public void run() {
    System.out.println("Thread started");
    while (!isInterrupted()) {
      System.out.println("Thread running");
      try {
        Thread.sleep(500)[2];
      } catch (InterruptedException e) {
        interrupt()*;
        System.out.println("Thread interrupted");
      }
    }
    System.out.println("Thread finished");
  }
}
```




```
ThreadInterrupt t = new ThreadInterrupt();

t.start();
Thread.sleep(2000);
t.interrupt();
```

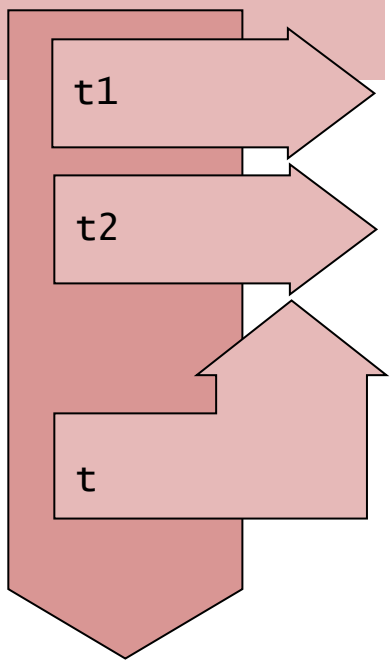
ACHTUNG: `interrupt()` bricht `run()` selbst nicht ab, dafür aber `sleep()`, `join()` und `wait()` und setzt Flag wieder auf `false` ➔ deshalb (*) nochmals aufrufen

[2] Siehe hinten *Local Drift*– und *Cumulative–Drift*–Problem

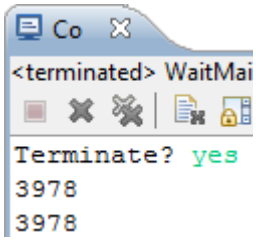
Auf das Ende von Threads warten mit `join()`

```
public class WaitThread extends Thread
{
  public long time = 0;

  @Override
  public void run() {
    long start = Calendar.getInstance().getTimeInMillis();
    while (!isInterrupted())
      try {
        Thread.sleep(1000);
      } catch (InterruptedException e) {
        interrupt();
      }
    time = Calendar.getInstance().getTimeInMillis() - start;
  }
}
```

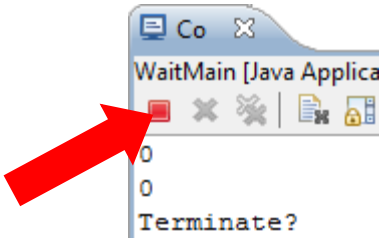
```
final WaitThread t1 = new WaitThread();
final WaitThread t2 = new WaitThread();
t1.start(); t2.start();

Thread t = new Thread() {
  @Override
  public void run() {
    TestScanner.readString("Terminate?");
    t1.interrupt(); t2.interrupt();
  }
};
t.start();

t1.join(); t2.join();
System.out.println(t1.time);
System.out.println(t2.time);
```



Mit `t1.join()` und `t2.join()`



Ohne `t1.join()` und `t2.join()`



ACHTUNG: Programm wird erst beendet, wenn letzter Thread stirbt (siehe nächster Punkt)

`void join(long millis) throws InterruptedException`
wie `join()`, doch wartet diese Variante höchstens `millis` und fährt dann mit Abarbeitung fort

Thread als Dämon starten

Serverthread arbeiten in Endlosschleife seine Aufgaben ab

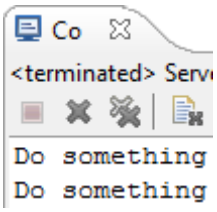
```
public class ServerThread extends Thread
{
  @Override
  public void run() {
    while (true)
     System.out.println("Do something");
  }
}
```

Wenn `main()` des Hauptthread fertig abgearbeitet ist, bedeutet dies nicht, dass auch Serverthread beendet wird, es sei denn Thread wird zum Dämon

```
public class ServerMain
{
  public static void main(String[] args) {
    ServerThread t = new ServerThread();
    t.setDaemon(true);
    t.start();
  }
}
```

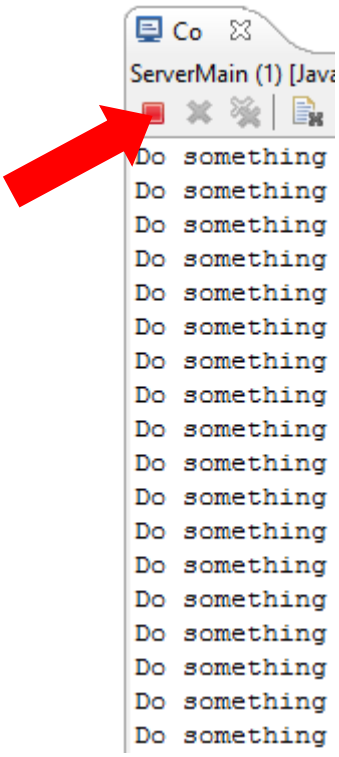
Hauptthread zieht seine Dämonen mit in den Tod

Mit `t.setDaemon(true)`



WICHTIG: Muss vor `start()` gemacht werden

Ohne



Weitere Möglichkeiten

`yield()`
Thread teilt der VM mit, dass er seine Rechenzeit dem Scheduler zurückgeben möchte. Scheduler belässt Thread aber immer noch in der Warteschlange, so dass dieser beim nächsten Durchlauf wieder drankommen kann

`setPriority()` und `getPriority()`
Thread wird Priorität (zw. 1 und 10, standard 5) zugeordnet die ausdrücken soll, wie viel Rechenzeit Thread relativ zu anderen erhalten soll. **ANMERKUNG:** Java macht sehr schwache Aussagen über die Bedeutung und Auswirkung von Thread-Prioritäten

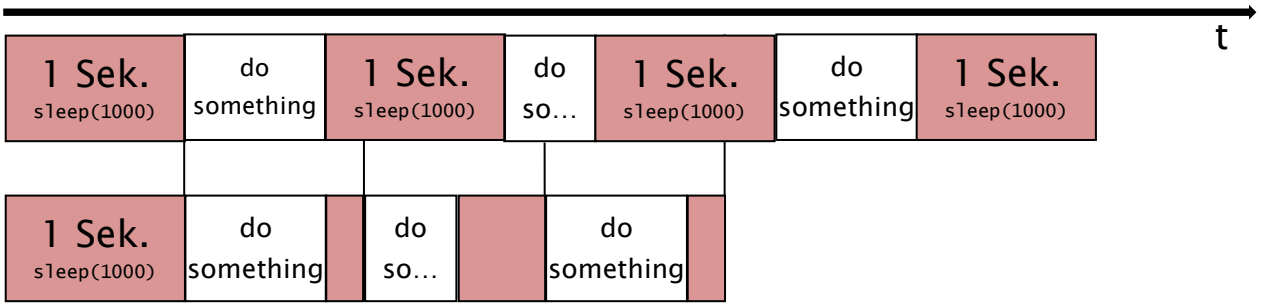
„Local Drift“- und „Cumulative Drift“-Problem

Local-Drift

`sleep(1000)` weckt nicht exakt nach 1000 Millisekunden auf, sondern etwas später

Cumulative Drift

Aufgabe soll jede Sekunde gestartet werden. Allein mit `sleep()` nicht lösbar denn



LÖSUNG: Anstelle von `sleep(1000)` wird `sleepUntil()` aufgerufen das sich berechnet, wie lange noch bis zur nächsten Sekunde gewartet wird

```
import java.time.Instant;
public class Test extends Object
{
  public static void sleepUntil(Instant until)
    throws InterruptedException {
    System.out.println("sleepUntil(" + until.getEpochSecond() + ",
       " + until.getNano() + ")");
    long now = Instant.now().toEpochMilli();
    if (until.toEpochMilli() - now > 0)
      Thread.sleep(until.toEpochMilli() - now);
  }
  public static void main(String[] args)
    throws InterruptedException {
    Instant until = Instant.now();
    for (int i = 0; i < 10; i++) {
      until = until.plusSeconds(1);
      sleepUntil(until);
      int workingMillis =
        (int)(Math.random() * 1000);
      System.out.println(
        "doSomething " + workingMillis);
      Thread.sleep(workingMillis);
    }
  }
}
```

```
sleepUntil(1579504461, 143000000)
doSomething 436
sleepUntil(1579504462, 143000000)
doSomething 840
sleepUntil(1579504463, 143000000)
doSomething 127
sleepUntil(1579504464, 143000000)
doSomething 942
sleepUntil(1579504465, 143000000)
doSomething 986
sleepUntil(1579504466, 143000000)
doSomething 939
sleepUntil(1579504467, 143000000)
doSomething 79
sleepUntil(1579504468, 143000000)
doSomething 648
sleepUntil(1579504469, 143000000)
doSomething 27
sleepUntil(1579504470, 143000000)
doSomething 138
```