Nebenläufigkeit: Einführung in die Thread-Programmierung

- Unterschied zwischen *Prozess* und *Thread* erkennen
- Wichtige *Begriffe* der Thread-Programmierung kennen
- Thread über eine `Runnable`-Objekt erzeugen und starten können
- Eigenen Thread als Erweiterung der Klasse `Thread` erzeugen können
- *Wichtige Methoden* der Klasse Thread verwenden können
- Die Verwendung der Methoden `currentThread()` und `sleep()` beherrschen
- Anwendungsmöglichkeiten von Threads in der GUI-Programmierung beherrschen
- Dem Problem das *Swing* nicht *threadsicher* ist begegnen können
- Anwendungsmöglichkeiten und Unterschiede zwischen `invokeLater()` und `invokeAndWait()` kennen

Unterschied Prozess – Thread, Begriffe

Prozess 2

Prozess 1

```
int a = 1;
```

- Besteht aus Programmcode und Daten
- Besitzt eigenen Adressraum
- Verfügt über Ressourcen (geöffnete Dateien, belegte Schnittstellen)

Thread 1

```
int b = 2;
```

- Führt eigentlichen Programmcode aus
- Verwendet zusammen mit anderen Threads denselben Adressraum

Thread 2

```
int c = 3;
```

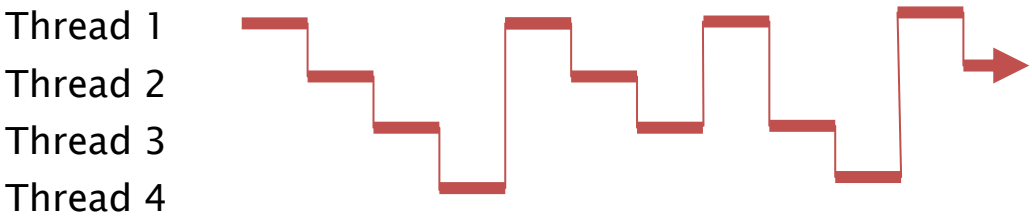
- Thread dtsch. *Ausführungsstrang*
- Prozesse tauschen Daten über *Shared-Memory* aus, Threads über ihre öffentliche Daten
- *Scheduler* weißt Threads Rechenzeit zu
- *Konkurrierende Zugriffe* bei gleichzeitigem Zugriff auf gemeinsame Ressourcen oder Daten ⇨ *Synchronisation*
- Synchronisation kann *Verklemmungen* (engl. *deadlocks*) hervorrufen
- Gemeinsam benutzte Bibliotheken (z.B. des Ein-/Ausgabesystems oder für Collections) sollten *threadsicher* sein. Swing ist nicht threadsicher, AWT schon (siehe hinten)

Parallele Threads

Mehrere Threads werden gleichzeitig auf mehreren Prozessoren ausgeführt

Verzahnte Threads

Mehrere Threads die stückweise abwechselnd auf einem Prozessor ausgeführt werden



Nebenläufige Threads

Threads die parallel oder verzahnt ausgeführt werden können

Threads erzeugen

Jeder laufende Thread ist eine Instanz der Klasse Thread

1. Möglichkeit: Thread erhält Befehlsobjekt vom Typ Runnable

```
public class CounterCommand implements Runnable
{
  private int counter = 0;
  @Override
  public void run() {
    String threadName = Thread.currentThread().getName();
    System.out.println(threadName + " started " + counter);
    for (int i = 0; i < 10; i++)
      System.out.println(threadName + " increment " + ++counter);
    System.out.println(threadName + " stopped " + counter);
  }
}
```

```
public static void main(String[] args) {
  Runnable r1 = new CounterCommand();
  Runnable r2 = new CounterCommand();
  Thread t1 = new Thread(r1);
  Thread t2 = new Thread(r2);
  t1.start();
  t2.start();
}
```

Console

<terminated> CommandCommandExe

```
Thread-0 started 0
Thread-0 increment counter 1
Thread-0 increment counter 2
Thread-0 increment counter 3
Thread-1 started 0
Thread-1 increment counter 1
Thread-1 increment counter 2
Thread-1 increment counter 3
Thread-1 increment counter 4
Thread-1 increment counter 5
Thread-1 increment counter 6
Thread-1 increment counter 7
Thread-1 increment counter 8
Thread-1 increment counter 9
Thread-1 increment counter 10
Thread-1 stopped 10
Thread-0 increment counter 4
Thread-0 increment counter 5
Thread-0 increment counter 6
Thread-0 increment counter 7
Thread-0 increment counter 8
Thread-0 increment counter 9
Thread-0 increment counter 10
Thread-0 stopped 10
```

- Befehlsobjekt wird dem Konstruktor von Thread übergeben
- `start()` erzeugt für Thread *Ablaufumgebung* und ruft selbständig genau einmal `run()` auf
- Jeder Thread kann nur einmal mit `start()` gestartet werden

Frage:

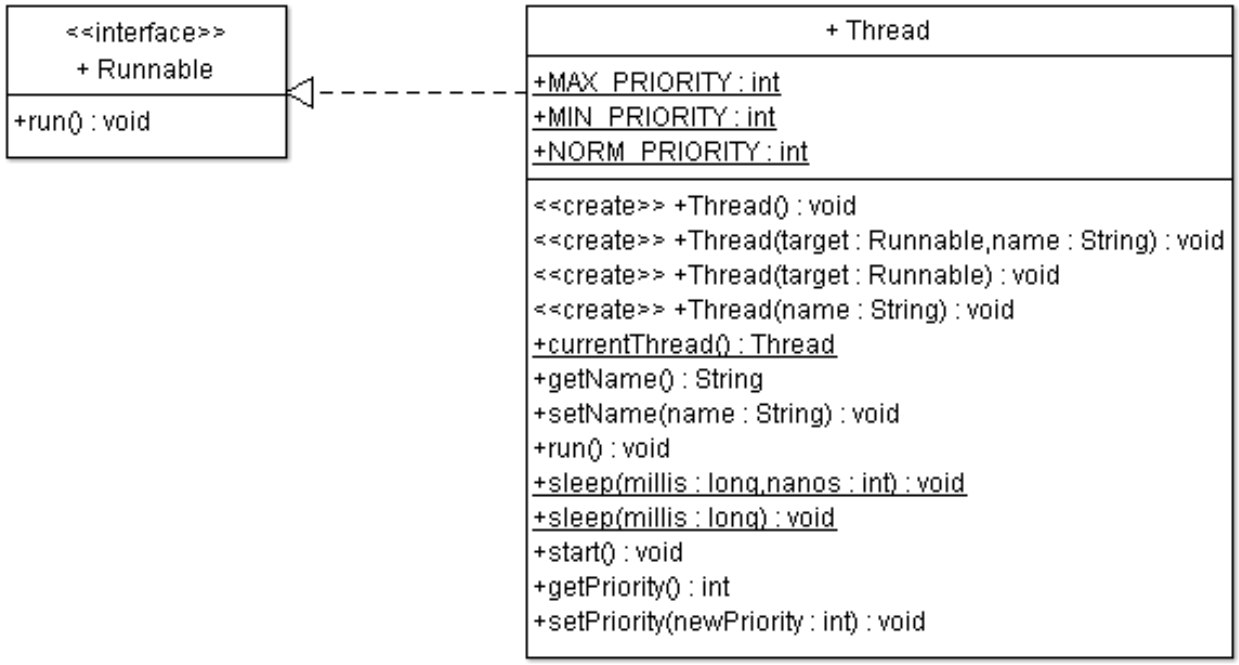
Wie würde `counter` hochgezählt wenn `t1` und `t2` *dasselbe Befehlsobjekt* übergeben wird?

Variante: Starten während der Erzeugung mit Namenssetzung

```
public class CounterCommand implements Runnable
{
  public CounterCommand(String name) {
    new Thread(this, name).start();
  }

  …
}
```

2. Möglichkeit: Erweiterung der Klasse Thread



[1]

- *Klasse* `Thread` implementiert *Interface* `Runnable`
- Klasse `Thread` enthält *leere* `run()`-Methode
- Statische Methode `currentThread()` liefert jenen Thread der diese Methode ausführt
- Statische Methode `sleep()` legt den Thread der `sleep()` ausführt *mindestens* für die angegebene Zeit schlafen

[1] Ausschnitt des *UML-Klassendiagrammes* der *Klasse* `Thread` und des *Interfaces* `Runnable`

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

Starten mit

```
CounterCommandThread t = new CounterCommandThread("Thread1");
t.start();
```

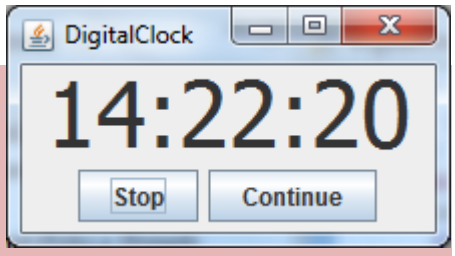
oder mit

```
new CounterCommandThread("Thread1").start();
```

Frage

Wie kann bei der Instanziierung des obigen Threads dieser automatisch gestartet werden?

Beispiel Digitaluhr




```
public class DigitalClock extends Thread
{
  private JLabel outputLabel = null;
  private boolean stopped = false;

  public DigitalClock(JLabel outputLabel) {
    this.outputLabel = outputLabel;
    this.start();
  }

  public boolean getStopped() {
    return stopped;
  }
  public void setStopped(boolean stopped) {
    this.stopped = stopped;
  }

  public void run() {
    while (true) {
      if (!this.stopped) {
        outputLabel.setText(
          new Date().toString().substring(11,19));
        try {
          Thread.sleep(1000);
        } catch (InterruptedException e) {
          ;
        }
      }
    }
  }
}
```

Problem:

Prozessorauslastung schnellt auf Maximum wenn `this.stopped == true`

Lösung:

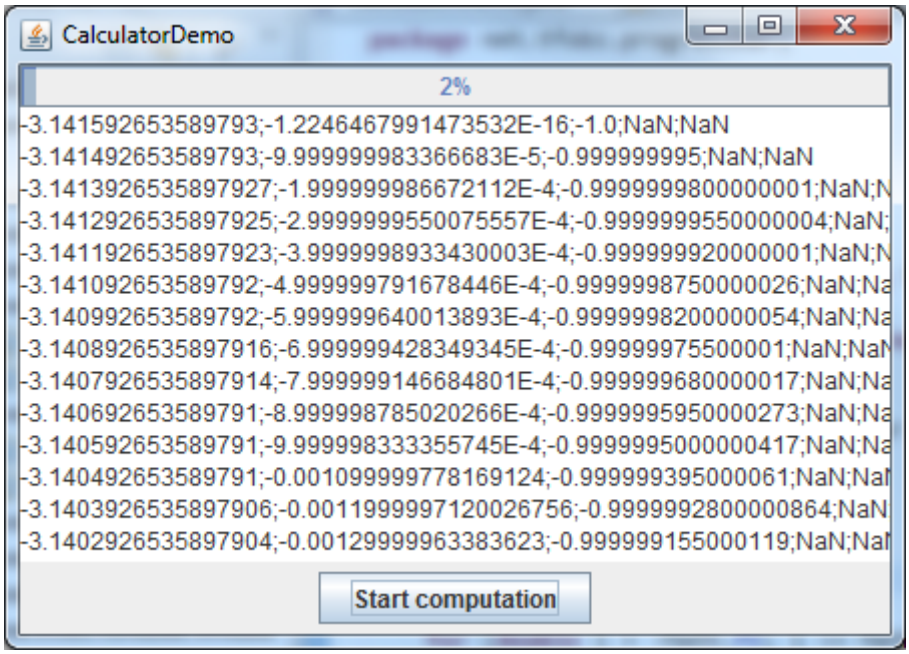
Das Schlafen außerhalb von `if` positionieren

Problem:

Thread wird aber immer noch nach einer Sekunde geweckt um zu kontrollieren, ob `this.stopped` gesetzt ist ⇨ `wait()`, `notify()` (siehe nächste Kapitel)

```
public class DigitalClockFrame extends JFrame
{
  private DigitalClock digitalClock = null;
  public DigitalClockFrame() {
    …
    JLabel clockLabel = new JLabel();
    this.digitalClock = new DigitalClock(clockLabel);
    JButton stop = new JButton("Stop");
    stop.addActionListener(
      new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          digitalClock.setStopped(true);
        }
      }
    );
    JButton continue = new JButton("Continue");
    continue.addActionListener(
      new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          digitalClock.setStopped(false);
        }
      }
    );
  }
}
```

Threads und Swing




```
public class CalculatorDemo extends JFrame
{
  private JProgressBar progressBar = null;
  private JTextArea textArea = null;

  public CalculatorDemo() {
    …
    JButton btnStart = new JButton("Start computation");
    btnStart.addActionListener(
      new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          new CalculatorThread(progressBar, textArea).start();
        }
      }
    );
  }
}
```

- Aufwändige, zeitraubende Berechnung wird in eigenem Thread gestartet damit in der Zwischenzeit Fenster auf Ereignisse reagieren kann
- Für jedes Fenster läuft eigener *Fenster-Thread* der Ereignisse in der *Event-Queue* nacheinander abarbeitet

```
public class CalculatorThread extends Thread
{
  private JProgressBar progressBar = null;
  private JTextArea textArea = null;

  public CalculatorThread(
    JProgressBar progressBar, JTextArea textArea) {
    this.progressBar = progressBar;
    this.textArea = textArea;
  }

  public void run() {
    int p = 0;
    for (DOUBLE i = -Math.PI; i <= Math.PI; i+=0.0001) {
      final double j = i;
      final int q = p;
      EventQueue.invokeLater(
        new Runnable() {
          public void run() {
            textArea.append(j + ";" + Math.sin(j) + ";…\n");
            progressBar.setValue(q);
          }
        }
      );
      p++;
    }
  }
}
```

Anmerkungen:

- Nur eine als `final` deklarierte *lokale Variable* ist in anonymer Klasse der Methode verwendbar
- Auch `DigitalClock` von vorher müsste angepasst werden

Problem:

Swing-Komponenten sind *nicht threadsicher* (wegen Geschwindigkeitseinbusen und Gefahr von Deadlock-Situationen)!

Lösung:

- `EventQueue.invokeLater()` reiht Auftrag zur Oberflächenmanipulation in Event-Queue ein. Auftrag ist in `Runnable`-Objekt definiert
- `EventQueue.invokeAndWait()` wartet bis Auftrag durch Event-Queue abgearbeitet wurde und kehrt erst dann zurück

ACHTUNG:

Event-Queue sollte nicht mit zu vielen Ereignissen bombardiert werden