

# Threads

## Programmiermethodik 2

**Zum Nachlesen:**

Christian Ullenboom: Java ist auch eine Insel, Kapitel 14.1-14.3  
[http://openbook.rheinwerk-verlag.de/javainsel9/javainsel\\_14\\_001.htm](http://openbook.rheinwerk-verlag.de/javainsel9/javainsel_14_001.htm)

# Wiederholung

- Streams
- Streams von Elementen
- (Streams von Daten)
- Serialisierung

# Ausblick



# Agenda

- Parallelität und Threads
- Erzeugen
- Beenden
- Weitere Methoden
- Timer
- Probleme

## Zum Nachlesen

- Kathy Sierra, Bert Bates: Java von Kopf bis Fuß, Kapitel 15 ab Abschnitt "Multithreading", O'Reilly-Verlag
- zur `wait()` / `notify()`: Oracle Java Tutorial "Guarded Blocks": <http://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html>, abgerufen am 11.04.2014
- Christian Ullenboom: Java ist auch eine Insel, 9., aktualisierte Auflage 2011, Galileo Computing, Kapitel 14.6: Synchronisation über Warten und Benachrichtigen, auch online verfügbar
- Brian Goetz: Java Concurrency in Practice, Addison Wesley



# Einführung

# Bisher: Sequenzieller Programmablauf

- typischer Ablauf:
  - Start in der main()-Methode
  - Erzeugen eines Objektes
  - Aufrufen einer Methode
  - dort Aufruf einer Methode eines anderen Objektes
  - Rückkehr aus der Methode
  - Aufruf einer weiteren Methode
  - ...
  - Erreichen des Endes der main()-Methode

# Sequentielle Programme

- Bestehen aus unteilbaren Anweisungen
- nur das Programm selbst ändert die Umgebung
- Ergebnisse sind zeitunabhängig, daher
  - ein geschlossenes System
  - wiederholbar
- Entdeckung von Fehlern ist relativ einfach



- parallele (nebenläufig arbeitende) Systeme haben diese Eigenschaften nicht!



# Motivation: Nebenläufigkeit

- viele Programme basieren hinter den Kulissen bereits stark auf Nebenläufigkeit
  - z.B. Grafische Benutzerschnittstellen
- Nebenläufigkeit = technische Bezeichnung für den Umstand, dass mehrere Tätigkeiten nebeneinander laufen können.
- tägliches Leben:
  - zwei Menschen arbeiten parallel an einer Aufgabe
  - Voraussetzung: Verständigung über Arbeitsteilung

# Der klassische Prozessbegriff

- Programm:
  - statische Darstellung eines Algorithmus (Handlungsanweisungen)
- Prozessor
  - aktives Organ, das ein Programm ausführt
- Umgebung
  - Menge der Dinge, die bei einer Programmausführung manipuliert werden
- Prozess:
  - Ausführung eines Programms auf einem Prozessor in einer Umgebung

# Prozesse und Prozessorkerne

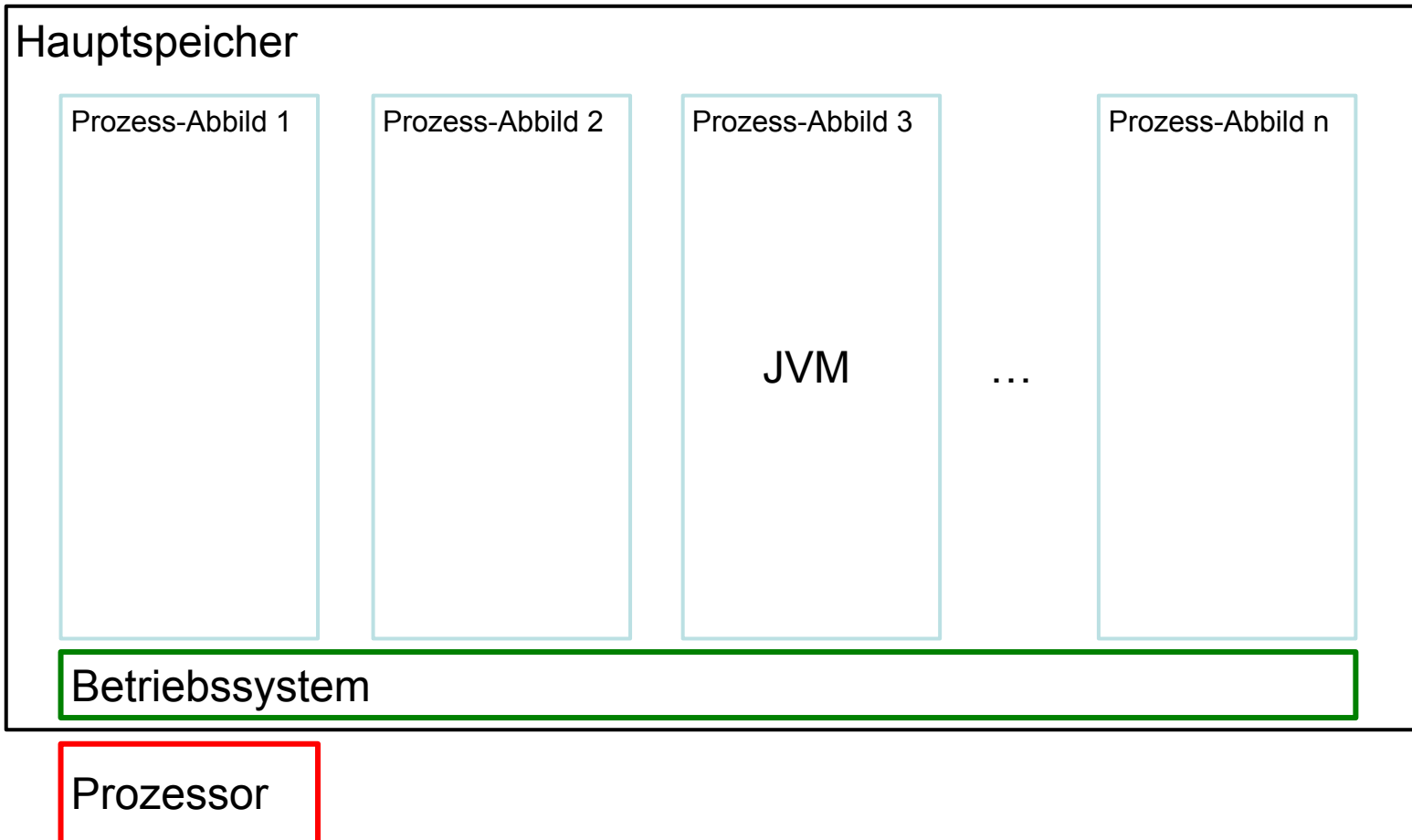
- Prozessorkern:
  - zentrale Ausführungseinheit eines Computers
- bis ca. zum Jahr 2005:
  - Computern verfügten über nur einen Kern
  - Nebenläufigkeit von Prozessen nur simuliert:
    - schneller Wechsel zwischen mehreren Prozessumgebungen
- heute:
  - meist tatsächlich mehrere Kerne
  - „echte“ Parallelität möglich



# Zeitscheibenverfahren

- aber auch heute:
  - meist mehr Prozesse als Kerne
  - Nebenläufigkeit durch Betriebssystem simuliert
- daher für uns Vereinfachung:
  - Betrachtung von Prozessoren mit einem Kern (=Prozessor)
- Teilen eines Prozessors in modernen Betriebssystemen:
  - Zeitscheibenverfahren
- zentrale Komponente des Betriebssystems (Scheduler), legt fest, welcher Prozess als nächstes den Prozessor bekommt
  - Details dazu in Veranstaltungen zu Betriebssystemen

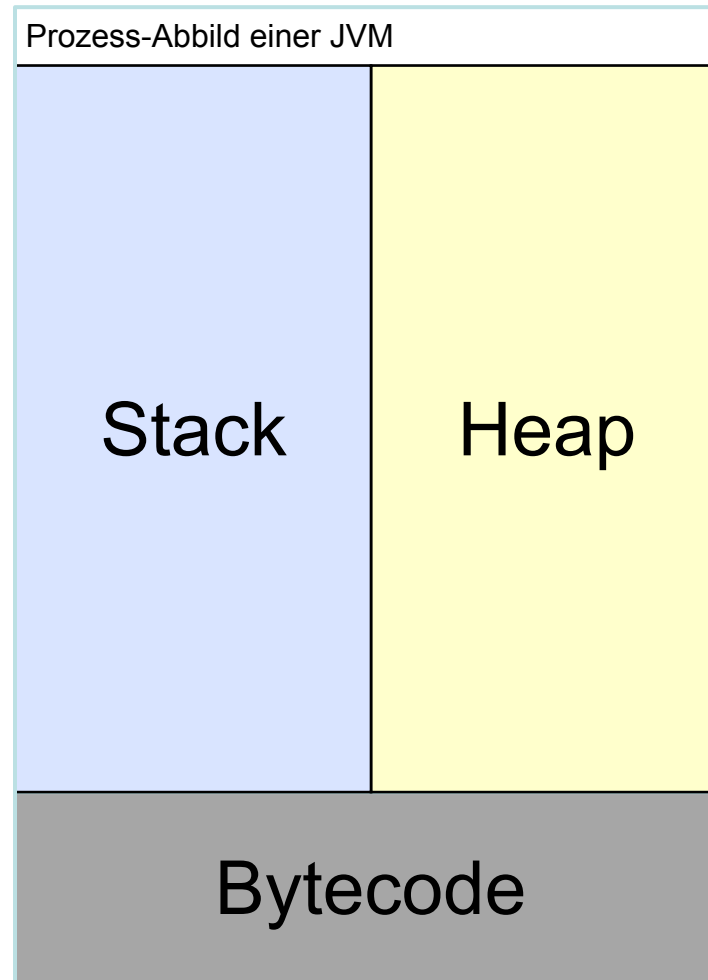
# Eine Prozessor für viele Prozesse



# Wie sieht eine Java-Prozess aus?

- drei Komponenten:
  - Stack
  - Heap
  - Bytecode
- Stack:
  - Information, welcher Befehl in welcher Methode des aktuellen Objekts gerade ausgeführt wird
- Heap
  - Information, welche Java-Objekte gerade existieren und wie sie miteinander über Referenzen verknüpft sind
- Bytecode
  - der Klassen, die das Programm ausmachen

# Zoom: Das Prozess-Abbild für eine JVM



# Java-interne Prozesse: Threads

- also: JVM = eigenständiger Prozess des Betriebssystems
  - kaum Schnittstellen mit anderen Prozessen
- aber: innerhalb einer JVM kann es aber auch parallele Prozesse geben "Mini-Java-Prozesse"
  - begrifflich zu unterscheiden, nennen wir die Java-internen Prozesse Threads.



Aus der API der Klasse Thread:

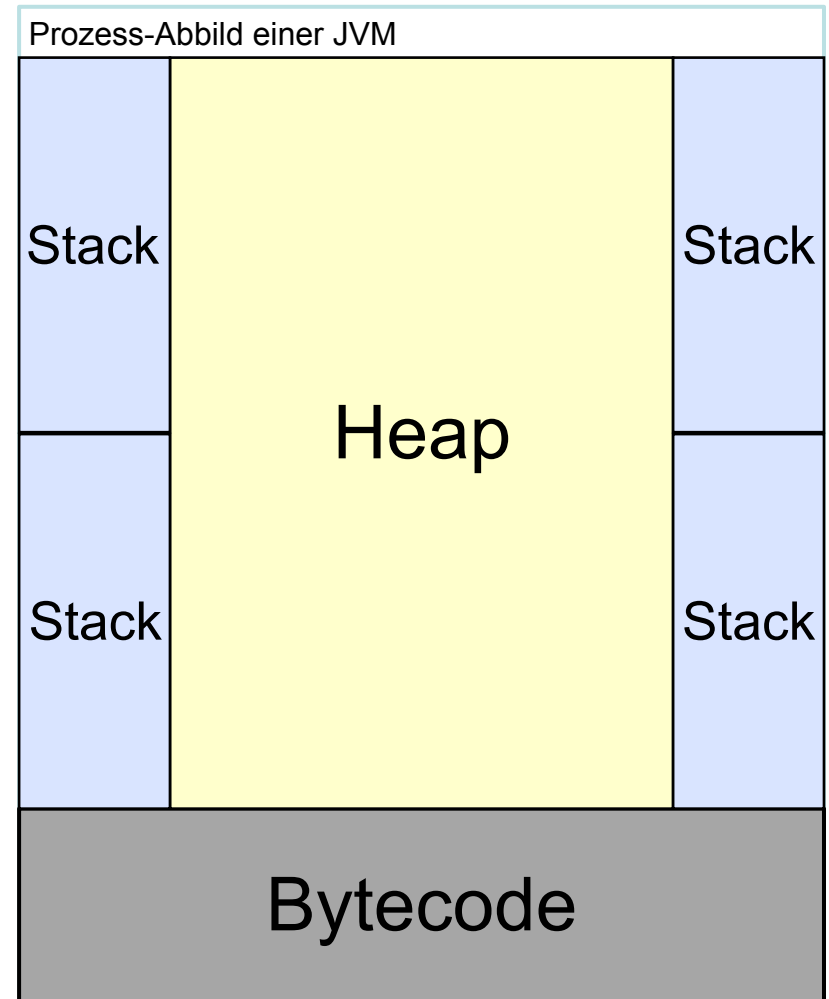
*"A thread is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently."*

- ein leichtgewichtiger Prozess.
- Threads eines Java-Programms haben oft einen engen fachlichen Zusammenhang



# Threads in der JVM

- Stack hält lokale Variablen und Methodenaufrufe
  - daher sind lokale Variablen thread-lokal
  - jeder Thread hat seine eigenen lokalen Variablen
- Heap hält die Objekte
  - daher sind Objektvariablen thread-global
  - potentiell für alle Threads (die eine Referenz auf ein Objekt haben) zugreifbar



# System-Threads in Java

- main()-Methode eines Java-Programms wird in einem speziellen Thread ausgeführt.
  - bisher: Java-Programm = nur dieser eine Thread
  - Endet mit Ende der main-Methode
- es gibt weitere System-Threads, beispielsweise für Garbage Collection



# Erzeugen

# Java-Threads

- Die main-Methode eines Java-Programms wird bereits durch einen Thread ausgeführt
  - ohne explizite Thread-Verwendung läuft nur der eine Haupt-Thread ab (main()-Thread)
- Zum Erzeugen weiterer Threads gibt es im Package java.lang ...
- die Klasse Thread
  - Einzige von einer abgeleiteten Klasse zu redefinierende Methode:
    - void run()
- das Interface Runnable (als Alternative)
  - Einzige zu implementierende Methode: void run()
  - muss mit dem Code für den eigenen Thread (re-)definiert werden

# Erster Weg: Von Thread Erben

- Vorgehen
  - eigene Klasse schreiben
  - von Thread erben
  - Methode run() überschreiben
  - Instanz erzeugen und mit start() starten

# Beispiel

```
public class ZaehlerThread extends Thread {  
    @Override  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            System.err.println(i);  
        }  
    }  
}
```

Ausgabe:

*Ende der main()-Methode.*

```
ZaehlerThread zaehlerThread = new ZaehlerThread();  
zaehlerThread.start();  
System.err.println("Ende der main()-Methode.");
```

0  
1  
2  
3  
4  
5  
6  
7  
8  
9

# Thread Starten

- Methode der Klasse Thread zum Starten eines neuen Threads: void start()
- erzeugt, initialisiert und startet den Thread auf Betriebssystem-Ebene
- ruft die Methode run() automatisch auf
- start() kehrt sofort zurück, der neue Thread läuft parallel
- Niemals run() direkt aufrufen!
  - ansonsten: nicht-parallel Ausführung der run()-Methode
  - dann weiter normaler Programmfluss (nachdem run()-Methode zurückgekehrt ist)

## **Zweiter Weg: Interface Runnable implementieren**

- eigene Klasse schreiben, die Interface Runnable implementiert
- Methode run() überschreiben
- Instanz der Klasse an Konstruktor-Aufruf von Thread übergeben
- Thread mit start() starten



# Beispiel

```
public class ZaehlerRunnable implements Runnable {  
    @Override  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            System.err.println(i);  
        }  
    }  
}
```

Ausgabe (und übriges Verhalten)  
identisch zur ersten Variante:

```
Thread zaehlerThread = new Thread(new ZaehlerRunnable());  
zaehlerThread.start();  
System.err.println("Main Ende");
```

0  
1  
2  
3  
4  
5  
6  
7  
8  
9

## Übung: Parallele Eingabe

- Erzeugen und starten Sie einen Java-Thread, der auf eine Anwender-Eingabe wartet (einen String) und diesen auf der Konsole ausgibt.
- Setzen Sie Ihre Implementierung mit beiden besprochenen Varianten um



**Beenden**

# Threads Beenden

- Methoden der Klasse Thread zum Beenden eines Threads
- „Normales“ Ende
  - run()-Methode ist fertig!
- Beenden durch einen anderen Thread:
  - void stop()
  - nicht mehr verwenden (unsicher)!!

# Threads Beenden

- Die stop()-Methode ist als deprecated gekennzeichnet

```
StopNichtVerwenden thread = new StopNichtVerwenden();
thread.start();
try {
    Thread.sleep(2000);
} catch (InterruptedException e) {
    System.err.println(e.getStackTrace());
}
thread.stop();
System.err.println("Thread gestoppt.");
```

- Was spricht gegen stop()?

## Was dann?

- Thread mitteilen, dass er sich bitte selber beenden sollte
- Thread entscheidet selber, wann und ob er das macht
- Mitteilung durch ein Interrupt

# Interrupt

- Thread mitteilen, dass Abbruch gewünscht ist:

`void interrupt()`

- setzt für den Thread ein Interrupt-Flag
- entspricht Setter einer Objektvariablen vom Typ boolean
- weckt Thread, falls er blockiert ist

- Thread-intern prüfen, ob Abbruch gewünscht ist:

`boolean isInterrupted()`

- liefert den Wert des Interrupt-Flags für den Thread
- entspricht Getter einer Objektvariablen vom Typ boolean
- muss vom Thread-Code abgefragt werden

# Beispiel

## Ausgabe (Auszug)

```
public class ZaehlerUnterbrechen extends Thread {  
    @Override  
    public void run() {  
        int zahl = 0;  
        while (!isInterrupted()) {  
            System.err.println("Zähler-Thread:" + zahl);  
            zahl++;  
        }  
        System.err.println("Zähler-Thread beendet.");  
    }  
}
```

```
public static void main(String[] args) {  
    ZaehlerUnterbrechen zaehlerThread = new ZaehlerUnterbrechen();  
    zaehlerThread.start();  
    for (int zahl = 0; zahl < 100; zahl++) {  
        System.err.println("main()-Zähler: " + zahl);  
    }  
    zaehlerThread.interrupt();  
    System.err.println("Ende der main()-Methode.");  
}
```

```
...  
Zähler-Thread:36  
Zähler-Thread:37main()-Zähler: 95  
main()-Zähler: 96  
main()-Zähler: 97  
main()-Zähler: 98  
main()-Zähler: 99  
Zähler-Thread:38  
Ende der main()-Methode.  
Zähler-Thread beendet.
```



# Threads Beenden

- Thread kann nach Beendigung nicht neu gestartet werden
- unabhängig davon ob er
  - von alleine fertig war (Ende der run()-Methode erreicht)
  - zum Ende "gebeten" wurde (durch Aufruf von interrupt())

# Schlafen

- Methoden der Klasse Thread zum Anhalten und Fortsetzen

```
static void sleep(long milliseconds) throws
```

```
    InterruptedException
```

```
static void sleep(long milliseconds, long nanoseconds)  
    throws InterruptedException
```

- hält den ausgeführten Thread für die angegebene Zeit an
- kann InterruptedException werfen
  - falls zwischendurch erfolgreicher Aufruf von interrupt() durch anderen Thread
  - weckt den ausführenden Thread → muss von diesem behandelt werden


# Schlafen

```
@Override
public void run() {
    int i = 0;
    while (!isInterrupted()) {
        System.err.println(i++);
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            System.err.println("Thread wurde durch Interrupt geweckt!");
            interrupt();
        }
    }
    System.err.println("Thread wird beendet!");
}
```

Interrupt-Ereignis in Exception  
gefangen - wird damit auf false  
zurückgesetzt.



neu setzen, damit  
die while-Schleife  
beendet wird



# Schlafen

- Achtung: catch-Aufruf setzt Interrupt-Flag auf false zurück
- sleep() kann auch außerhalb einer Klasse, die von Thread erbt verwendet werden
  - dann wird umschließender Thread schlafen gelegt
  - z.B. main()-Thread

## Übung: Stopper

- Schreiben Sie eine Klasse Stopper, die von Thread erbt.
- dieser Thread soll einen Zähler von 0 beginnend in Schritten von 10-5 hochzählen
- der Thread endet, wenn ihm interrupt() mitgeteilt wird
- Starten Sie den Thread in einer main()-Methode
- unterbrechen/enden Sie den Thread durch eine Tastatureingabe



## Weitere Methoden

# Weitere Methoden von Thread

- Java-Thread-Objekt des ausgeführten Threads

```
static Thread currentThread()
```

- Namen des Threads

```
String getName()
```

```
void setName(String name)
```

- liefert true, wenn der Thread gestartet, aber noch nicht beendet wurde

```
boolean isAlive()
```

# Join()

- hält den ausgeführten Thread an, bis dieser Thread (das Zielobjekt beim Aufruf) beendet ist

`void join() throws InterruptedException`



# Beispiel

```
public class ThreadsJoinIsAlive extends Thread {
```

```
@Override
```

```
public void run() {
```

```
    System.err.println("Name des ausgeführten Threads (in run()): "
```

```
        + Thread.currentThread().getName());
```

```
    for (int i = 0; i < 10; i++) {
```

```
        System.err.println("Thread-Zahl: " + i);
```

```
    }
```

```
    System.err.println("ThreadsJoinIsAlive beendet!");
```

```
}
```

```
public static void main(String[] args) {
```

```
    ThreadsJoinIsAlive testThread = new ThreadsJoinIsAlive();
```

```
    System.err.println("Name des ausgeführten Threads (in main()): "
```

```
        + Thread.currentThread().getName());
```

```
    System.err.println(
```

```
        "ThreadsJoinIsAlive isAlive() vor start()? " + testThread.isAlive());
```

```
    testThread.start();
```

```
    System.err.println(
```

```
        "ThreadsJoinIsAlive isAlive() nach start()? " + testThread.isAlive());
```

```
    try {
```

```
        testThread.join();
```

```
    } catch (InterruptedException e) {
```

```
        System.err.println("InterruptedException bei join()");
```

```
    }
```

```
    System.err.println(
```

```
        "ThreadsJoinIsAlive isAlive() Ende main()? " + testThread.isAlive());
```

```
    System.err.println("main()-Thread beendet");
```

```
}
```

```
}
```

Ausgabe:

*Name des ausgeführten Threads (in main()): main*

*ThreadsJoinIsAlive isAlive() vor start()? false*

*ThreadsJoinIsAlive isAlive() nach start()? true*

*Name des ausgeführten Threads (in run()): Thread-0*

*Thread-Zahl: 0*

*Thread-Zahl: 1*

*...*

*Thread-Zahl: 8*

*Thread-Zahl: 9*

*ThreadsJoinIsAlive beendet!*

*ThreadsJoinIsAlive isAlive() Ende main()? false*

*main()-Thread beendet*

# Priorität

- Möglichkeit, einem Thread eine Priorität zuzuweisen  
`void setPriority(int newPrio)`
- weist dem Thread eine Priorität zu
- kleinster möglicher Wert: `Thread.MIN_PRIORITY` (üblich: 1)
- größter möglicher Wert: `Thread.MAX_PRIORITY` (üblich: 10)
- neue Threads haben automatisch die Priorität des Eltern-Threads

# Priorität

- Achtung: Priorität ist eine "Empfehlung" an das Betriebssystem
- Je nach OS und nach Aufgabe der Threads ist der Effekt zu vernachlässigen

# Beispiel

```
public class Prioritaet extends Thread {
    private int anzahlAufrufe = 0;
    @Override
    public void run() {
        while (!isInterrupted()) {
            System.err.println("Anzahl Aufrufe: " + anzahlAufrufe);
            anzahlAufrufe++;
        }
    }
    public int getAnzahlAufrufe() {
        return anzahlAufrufe;
    }
    public static void main(String[] args) {
        List<Prioritaet> threads = new ArrayList<Prioritaet>();
        for (int i = 0; i < 100; i++) {
            boolean isLowPrio = Math.random() < 0.5;
            Prioritaet thread;
            if (isLowPrio) {
                thread = new Prioritaet();
                thread.setName("Niedrige Priorität");
                thread.setPriority(Thread.MIN_PRIORITY);
            } else {
                thread = new Prioritaet();
                thread.setName("Hohe Priorität");
                thread.setPriority(Thread.MAX_PRIORITY);
            }
            threads.add(thread);
        }
        for (int i = 0; i < threads.size(); i++) {
            threads.get(i).start();
        }
        try {
            Thread.sleep(300);
        } catch (InterruptedException e) {
            System.err.println(e.getStackTrace());
        }
        for (int i = 0; i < threads.size(); i++) {
            threads.get(i).interrupt();
        }
        for (int i = 0; i < threads.size(); i++) {
            System.err.println("Anzahl Aufrufe: (" + threads.get(i).getName() + "): "
                + threads.get(i).getAnzahlAufrufe());
        }
    }
}
```

## – Ausgabe (Auszug):

...

Anzahl Aufrufe: (Hohe Priorität): 55

Anzahl Aufrufe: (Hohe Priorität): 155

Anzahl Aufrufe: (Hohe Priorität): 82

Anzahl Aufrufe: (Hohe Priorität): 360

Anzahl Aufrufe: (Niedrige Priorität): 219

Anzahl Aufrufe: (Niedrige Priorität): 54

Anzahl Aufrufe: (Hohe Priorität): 154

Anzahl Aufrufe: (Niedrige Priorität): 241

Anzahl Aufrufe: (Hohe Priorität): 56

Anzahl Aufrufe: (Niedrige Priorität): 356

Anzahl Aufrufe: (Niedrige Priorität): 47

Anzahl Aufrufe: (Hohe Priorität): 68

Anzahl Aufrufe: (Niedrige Priorität): 40

...



## Timer

# Timer

- Aufgabe
    - Warte eine bestimmte Zeit ab und führe dann eine Methode aus
    - Timerablauf, "Timeout"
    - unabhängig vom ausführenden Thread! Es besteht die Möglichkeit, den Timer vor Ablauf zu stoppen ("cancel")
  - Schnittstelle
    - Interface TimeoutTask
    - Implementiert die auszuführende Methode
- `void run()`

# Timer

- Starten durch

```
void schedule(TimeoutTask task, long delay, long period)
```

- Stoppen durch

```
void cancel()
```

# Beispiel

```
class TimerBeispiel {  
    public static void main(String[] args) {  
        Timer timer = new Timer();  
        timer.schedule(new TimerTaskBeispiel(), 2000, 2000);  
        try {  
            Thread.sleep(11000);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        timer.cancel();  
        System.err.println("Fertig.");  
    }  
}
```

```
class TimerTaskBeispiel extends TimerTask {  
    @Override  
    public void run() {  
        System.err.println("Aktuelle Sekunde: " + LocalDateTime.now().getSecond());  
    }  
}
```

– Ausgabe (stellvertretend):

*Aktuelle Sekunde: 50*

*Aktuelle Sekunde: 52*

*Aktuelle Sekunde: 54*

*Aktuelle Sekunde: 56*

*Aktuelle Sekunde: 58*

*Fertig.*



# Übung: Fußball

- Implementieren Sie eine kleine Fußballsimulation bestehend aus Spielern (Klasse Spieler) und einem Tor(wart) (Klasse Keeper).
- Der Torwart hält keine Bälle, er kann sich nur Tore fangen (Methode void score() ). Er merkt sich die Anzahl der geschossenen Tore. Nach jedem Tor muss er sich zwischen 0 und 0.5 Sekunden erholen.
- Ein Spieler ist als Thread implementiert. Er hält eine Referenz auf das Tor. Er schießt permanent auf das Tor (10 x in Folge, dann hört er auf).
- In der Anwendung gibt es ein Keeper und 10 Spieler. Wenn die Anwendung startet, beginnen alle Spieler, auf das Tor zu schießen. Erst wenn alle Spieler fertig sind, soll die Hauptanwendung (main()-Thread) enden.

# Zusammenfassung

- Parallelität und Threads
- Erzeugen
- Beenden
- Weitere Methoden
- Timer

# Quellen

- Die Folien basieren zum großen Teil auf den Folien von Prof. Dr. Martin Hübner, Hochschule für Angewandte Wissenschaften Hamburg und folgendem Buch: Elisabeth Freeman, Eric Freeman, Kathy Sierra, Bert Bates: *Head First Design Patterns*, O'Reilly Media, 2004
- [1] Valerijs Kostreckis, *123rf.com/*, Bild-Nummer : 14007058, abgerufen: 24.10.2013
- [2] Wikipedia: Mutual Exclusion: [http://en.wikipedia.org/wiki/Mutual\\_exclusion](http://en.wikipedia.org/wiki/Mutual_exclusion), abgerufen am 31.10.2013
- [3] Dijkstra, E. W.: "*Solution of a problem in concurrent programming control*". Communications of the ACM 8 (9): 569
- [4] Christian Ullenboom: Java ist auch eine Insel, Galileo Computing, ISBN 978-3-8362-1506-0
- [5] Wikipedia: Philosophenproblem, <http://de.wikipedia.org/wiki/Philosophenproblem>, abgerufen am 22.3.2014