

Algorithmen & Datenstrukturen

# Threads

**Abstraktion der nebenläufigen  
Programmierung**

Roger Diehl



# Inhalt

- Einführung
- Erzeugen und Starten von Threads
- Beenden von Threads
- Zusammenfassung

# Lernziele

- Sie kennen Vorteile und Nachteile von Nebenläufigkeit.
- Sie kennen das Mass für die Parallelisierung und die verschiedenen Sichtweisen von Amdahl und Gustafson.
- Sie kennen den Lebenszyklus von Threads.
- Sie kennen die zwei grundsätzlichen Arten, wie ein Java Thread implementiert werden kann und wissen warum man nur eine dieser Arten verwenden soll.
- Sie kennen mindestens drei Arten, wie ein Java Thread beendet werden kann.
- Sie können einen Thread aus einem andern Softwareteil beenden und wissen wie die InterruptedException anzuwenden ist.
- Sie können zu den obigen Lernzielen Code-Beispiele erstellen, nachvollziehen und modifizieren.

# Warum...

...ist das Schreiben von nebenläufiger Software so schwer?

- Zeitgleiche Abläufe beherrschen doch unseren Alltag.
- Wir arbeiten in Teams, koordinieren unsere Termine und übernehmen oder verteilen Aufgaben.
- In der Regel kommen wir mit dieser Art der Parallelität ganz gut zurecht.



# Abstraktion der nebenläufigen Programmierung

- Bei vielen nebenläufigen Konzepten ist es der Thread
  - der unabhängig von anderen agiert
  - durch Programmcode gesteuert wird
- Diese Beschreibungsweise hat ihren Ursprung in der sequenziellen Programmierung
  - bei der es genau einen Ablaufstrang gibt!
- Diese Parallelitätsabstraktion ist intuitiv **nicht** leicht zugänglich
- Wir denken im Alltag nicht in Threads!



# Buchtipp

## Nebenläufige Programmierung mit Java

Konzepte und Programmiermodelle für Multicore-Systeme

Prof. Dr. Jörg Hettel

Prof. Dr. Manh Tien Tran

August 2016

dpunkt.verlag

ISBN:

Print 978-3-86490-369-4

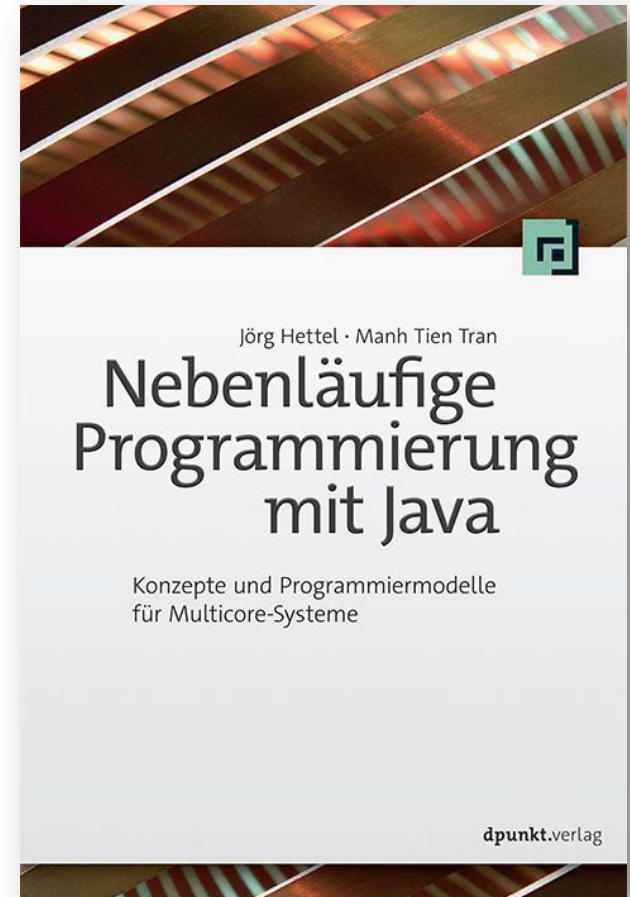
PDF 978-3-96088-012-7

ePub 978-3-96088-013-4

Mobi 978-3-96088-014-1

<http://webhome.hs-kl.de/~hettel/java-concurrency/>

<https://heise.de/-3331770>



# Buchtipp

## Concurrent Programming in Java, 2nd Edition

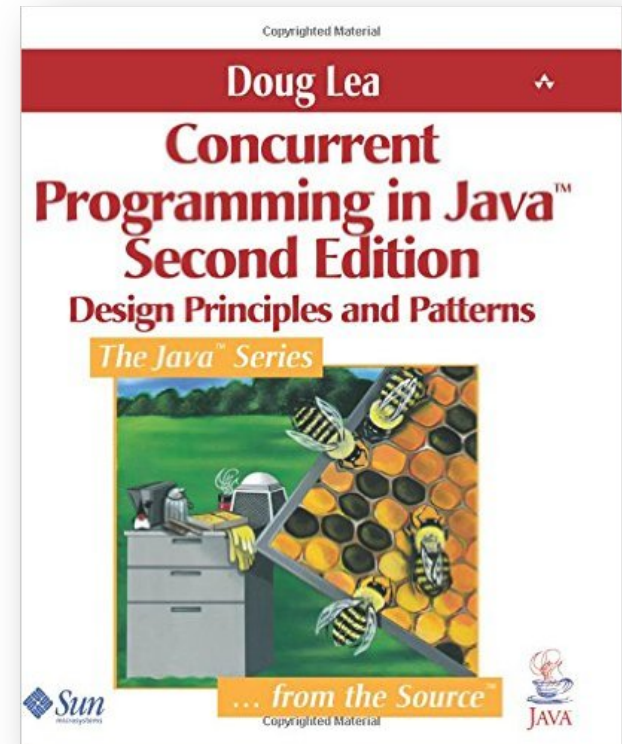
Design Principles and Patterns

Doug Lea

Oktober 1999

Addison-Wesley Professional

ISBN: 0-201-31009-0



<http://gee.cs.oswego.edu/dl/cpj/>

# Einführung



# Dimensionen der Parallelität

- Bei **Multinode-Systemen** wird die Aufgabe über verschiedene Rechner (Knoten) hinweg verteilt.
  - Jeder Knoten hat seinen eigenen Speicher und Prozessor (Beispiel <http://www.enterpriselab.ch>)
  - Verteilten Anwendungen
- Bei **Multiprocessor-Systemen** ist die Anwendung auf verschiedene Prozessoren verteilt.
  - Alle Prozessoren auf demselben Rechner (Mainboard)
  - Zugriff auf denselben Hauptspeicher
- Bei **Multicore-Systemen** befinden sich verschiedene Rechenkerne in einem Prozessor.
  - Zugriff auf den Hauptspeicher ist von allen Kernen gleich schnell
  - UMA-Architektur (Uniform Memory Access)

# Nebenläufigkeit und Parallelität

- siehe OOP Input → **O15\_IP\_Nebenläufigkeit; Folien 4-11**
- Zwei oder mehrere Aktivitäten (Tasks) heissen nebenläufig, wenn sie zeitgleich bearbeitet werden können.
  - Egal, ob zuerst der eine und dann der andere ausgeführt wird.
  - Egal, ob sie in umgekehrter Reihenfolge ausgeführt werden.
  - Egal, ob sie gleichzeitig erledigt werden.
  - Die Tasks haben keine kausale Abhängigkeit.
- Besitzt ein Rechner mehr als eine CPU oder Rechenkerne, kann die Nebenläufigkeit parallel auf der Hardwareebene realisiert werden.
  - Beschleunigung des Programms, wenn der zugehörige Kontrollfluss nebenläufige Tasks (Aktivitäten) beinhaltet.
- Das Abstraktionskonzept für Nebenläufigkeit bei Java ist der Thread
  - entspricht einem eigenständigen Kontrollfluss

# Vorteile / Nachteile von Nebenläufigkeit

- + Steigerung der Performance.
  - Beispielsweise kann auf mehreren CPUs das Sortieren eines grossen Arrays auf mehrere Threads verteilt werden.
- + Zur Verfügung stehende Rechenleistung wird voll ausgenutzt.
- + Durch Auslagerung von blockierenden Tätigkeiten in separate Threads kann die CPU in der Zwischenzeit andere Tasks erledigen.
- Programmcode mit Multithreading-Konzepten ist oft schwer zu verstehen und mit hohem Aufwand zu warten.
- Erschwertes Debugging, weil die CPU-Zuteilung an die Threads nicht deterministisch ist (bei jedem Programmstart anders).
- Parallel ablaufende Threads müssen koordiniert werden!
  - Vor allem bei Zugriff auf gemeinsame Daten

# Mass für die Parallelisierung

- Masszahl für den Performance-Gewinn ist der Speedup (Beschleunigung bzw. Leistungssteigerung)

$$S = \frac{T_{seq}}{T_{par}}$$

$T_{seq}$  Laufzeit mit einem Kern  
 $T_{par}$  Laufzeit mit mehreren Kernen

- Gesetz von Amdahl ([https://de.wikipedia.org/wiki/Amdahlsches\\_Gesetz](https://de.wikipedia.org/wiki/Amdahlsches_Gesetz))

$$S(N) = \frac{\text{Sequenzielle Laufzeit}}{\text{Parallele Laufzeit}} = \frac{1}{\frac{P}{N} + (1 - P)} = \frac{1}{(1 - P)}$$

P prozentuale, parallelisierbare Anteil

wenn N unendlich ist

N Anzahl Prozessoren

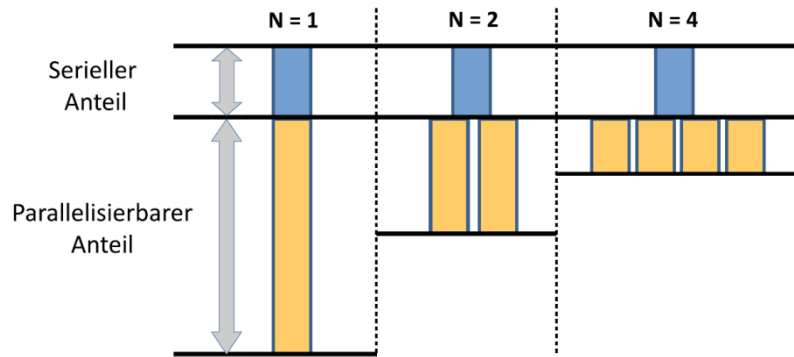
- Gesetz von Gustafson ([https://de.wikipedia.org/wiki/Gustafsons\\_Gesetz](https://de.wikipedia.org/wiki/Gustafsons_Gesetz))

$$S(N) = (1 - P) + N \cdot P$$

P prozentuale, parallelisierbare Anteil; N Anzahl Prozessoren

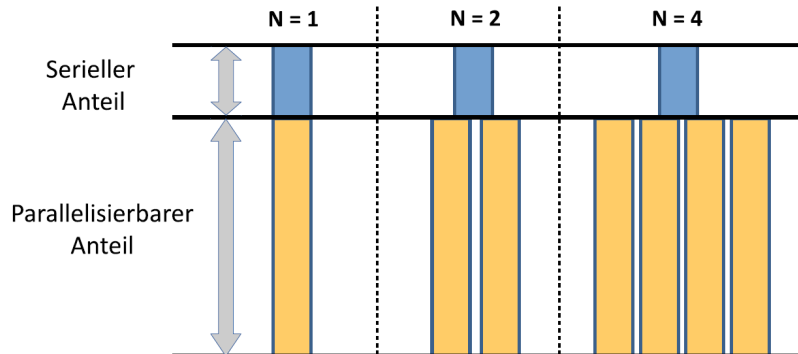
# Sichtweise von Amdahl und Gustafson

- Amdahl geht von einem fest vorgegebenen Programm bzw. einer fixen Problemgrösse (Fläche der gelben Balken bleiben gleich) aus.



Der nicht parallelisierbare Anteil begrenzt den Speedup

- Gustafson betrachtet eine variable Problemgrösse (Fläche der gelben Balken vergrössert sich) in einem festen Zeitfenster.



Die Vergrößerung des zu berechnenden Problems wirkt sich im Wesentlichen nur auf den parallelisierbaren Programmteil aus

**Die Anwendung ist skalierbar**

# **Erzeugen und Starten von Threads**

# Der main-Thread

- Eine Java-Anwendung wird in einer Java Virtual Machine (JVM) ausgeführt.
- Die JVM selbst entspricht einem Prozess des Betriebssystems.
- Zur Ausführung des Programms startet die JVM den main-Thread.
  - Die JVM startet auch noch weitere Threads (z.B. den GC)
- Wenn das Betriebssystem selbst Threads unterstützt, kann die JVM die Java-Threads auf sie abbilden.
  - Betriebssystem- bzw. OS-Threads
- Die Zuordnung der OS-Threads auf die Hardware-Threads übernimmt der Scheduler des Betriebssystems.
- Siehe OOP Input ➔ **O15\_IP\_Nebenläufigkeit; Folien 4-11**

## Beispiel: Attribute des main-Threads

- Zugriff auf den ausführenden Thread erhält man über die Klassenmethode `Thread.currentThread`

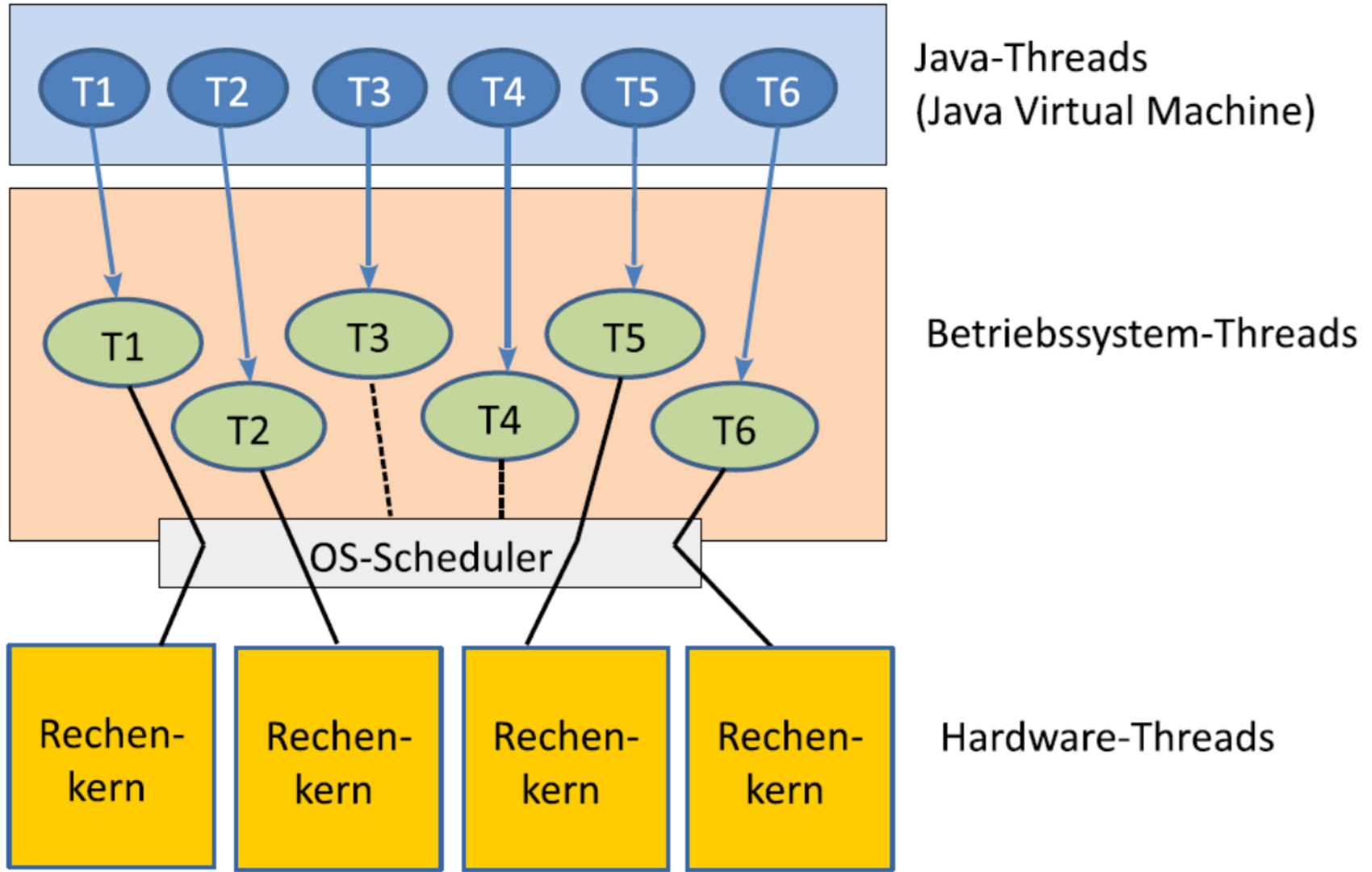
```
public static void main(final String[] args) {  
    // Anzahl der Prozessoren abfragen  
    final int nr = Runtime.getRuntime().availableProcessors();  
    System.out.println("Available processors " + nr);  
    // Eigenschaften des main-Threads  
    final Thread self = Thread.currentThread();  
    System.out.println("Name      : " + self.getName());  
    System.out.println("Priority  : " + self.getPriority());  
    System.out.println("ID       : " + self.getId());  
}
```



# Thread Erzeugung

- Es gibt zwei Arten um einen (Java) Thread zu erzeugen.
  - Ableitung der Klasse **Thread**
    - Erzeugen eines Thread Objekts der abgeleiteten Klasse\*
  - Implementation des Interfaces **Runnable**
    - Erzeugen eines Objekts der **Runnable** Klasse
    - Erzeugen eines Thread Objekts der Klasse **Thread**
- Das Thread Objekt muss mit der Methode **start** gestartet werden.
- Gemäss API Dokumentation: "In most cases, the Runnable interface should be used if you are only planning to override the **run()** method and no other Thread methods.«
- Bei Verwendung von **Runnable** wird konzeptuell klar zwischen dem Programmfluss (Thread) und der nebenläufig durchzuführenden Aufgabe (Task) unterschieden.

# Zuordnung der Java-Threads zu einzelnen Kernen



# Thread Erzeugung und Start mit Hilfe Runnable

- In der **run**-Methode sind die Anweisungen implementiert, die von einem Thread abgearbeitet werden.
- Instanzen der Task Klasse werden dem Thread-Objekt über den Konstruktor zugewiesen.

```
public final class MyTask implements Runnable {
```

Codeskizze

```
    @Override  
    public void run() {  
        //...Anweisungen - nebenläufig ausführen  
    }
```

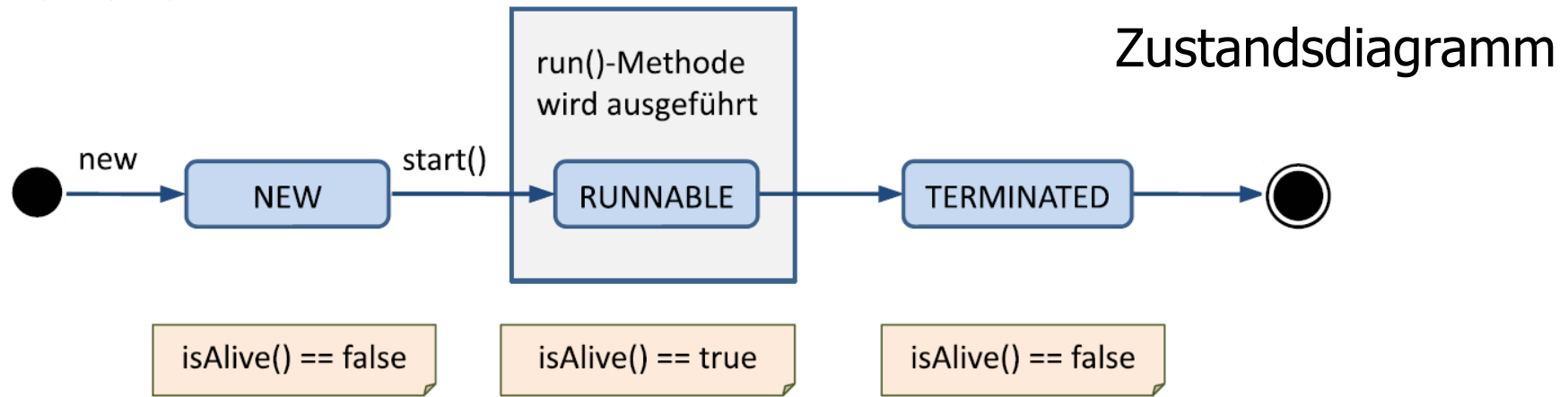
```
    public static void main(final String[] args) {  
        final MyTask myTask = new MyTask();  
        final Thread thread = new Thread(myTask, "MyTask-Thread");  
        thread.start();  
    }
```

```
}
```

Es empfiehlt sich, Threads immer einen Namen zuzuordnen - es erleichtert die Fehlersuche!

# Der Lebenszyklus von Threads

- Siehe OOP Input → **O15\_IP\_Nebenläufigkeit; Folien 10+11**
- Ein Java-Thread durchläuft während der Verwendung verschiedene Zustände:



- Ein Thread kann nur einmal gestartet werden kann - ein erneutes Starten ist nicht mehr möglich.
- Mithilfe der **isAlive**-Methode kann festgestellt werden, ob sich ein Thread Objekt im RUNNABLE-Zustand befindet (Rückgabe **true**) oder nicht (Rückgabe **false**).

# Praxistipp

- Die Methode **isAlive** verleitet dazu, dass man diese für das aktive Warten auf das Ende eines Threads einsetzt.
- Ein solches Warten verbraucht nur unnötig Ressourcen und sollte in der Praxis nicht angewendet werden!

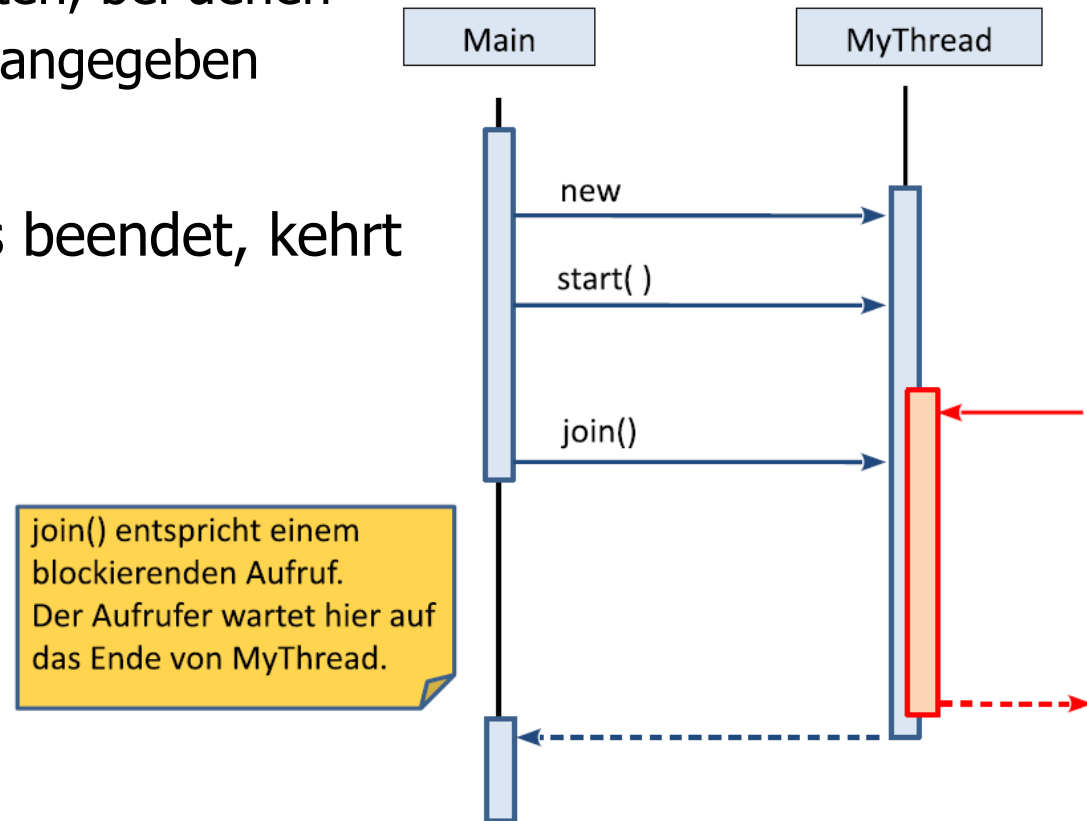
```
public static void main(String[] args) throws InterruptedException {  
    final MyTask myTask = new MyTask();  
    final Thread thread = new Thread(myTask, "MyTask-Thread");  
    thread.start();  
    while (thread.isAlive()) {  
        Thread.sleep(10);  
    }  
    for (int i = 0; i < 1000; i++) {  
        System.out.print("x");  
    }  
}
```

← Keine gute Idee!

Codeskizze

# Auf das Ende eines Threads warten

- Die Klasse **Thread** stellt die Methode **join** zur Verfügung.
- Der aufrufende Thread wartet so lange, bis der Ziel-Thread (hier **MyThread**) zum Ende kommt. Die Methode **join** ist blockierend.
  - Es gibt auch **join** Varianten, bei denen eine maximale Wartezeit angegeben werden kann.
- Ist der Ziel-Thread bereits beendet, kehrt **join** sofort zurück.



# **Beenden von Threads**

# Beenden eines Threads

- Der Thread Lebenszyklus besagt, dass mit dem Ende der Thread-Methode auch der Thread beendet wird.
- Ist in der Methode eine Endlosschleife programmiert, würde dies ein theoretisch nie endender Thread bilden.
- Allgemein ist ein Thread beendet, wenn eine der folgenden Bedingungen zutrifft:
  - Die **run**-Methode wurde ohne Fehler beendet.
  - In der **run**-Methode tritt eine Ausnahme (Exception) auf, welche die Methode beendet.
  - Wenn irgendwo **System.exit** aufgerufen wird.
  - Der Thread wurde von aussen aktiv beendet.



# Erwartung an das Beenden des Threads

- Der Thread soll zeitnah beenden
  - nachdem er benutzte Objekte in einen konsistenten Zustand gebracht hat, z.B.
    - gemeinsame Ressourcen bereinigt (z.B. auch Streams)
    - kritische Bereiche freigeben
  - und die anstehenden Aufgaben zu Ende geführt hat.
    - das kann unterschiedlich lange dauern
  - Thread soll auf keinen Fall eine neue Aufgabe beginnen.


# Aktives Beenden von Threads

- Grundsätzliche Möglichkeiten für das Beenden eines Threads durch einen anderen Programmteil:
  - erzwungen (forceful cancellation), heisst sofortiger Abbruch
  - verzögert (deferred cancellation), heisst Abbruch beim nächsten Abbruchpunkt
  - kooperativ (cooperative cancellation), heisst Thread wird gebeten zu beenden und muss sich dann selbst beenden, z.B. durch **return**
- Java verwendete bis zur Version 1.1 den erzwungen Abbruch mit der Methode **stop**. Seit Version 1.2 ist diese **deprecated** und sollte unter keinen Umständen benutzt werden.
- Java verwendet das kooperative Beenden.
  - Dies erfordert regelmässiges Nachfragen (Programmieraufwand).

# Idee für sicheres Beenden eines Threads

- Soll ein Thread aktiv beendet werden, so sollte er ordnungsgemäss seine `run`-Methode verlassen.

```
public final class StoppableTask implements Runnable {  
  
    private volatile Thread runThread;  
    private volatile boolean isStopped = false;  
  
    public void stopRequest() {  
        isStopped = true;  
        if (runThread != null) {  
            runThread.interrupt();  
        }  
    }  
  
    public boolean isStopped() {  
        return isStopped;  
    }  
    // ...  
}
```



Ein boolesches Attribut, das in der `run`-Methode regelmässig abgefragt wird.

Codeskizze

# Phasen für ein sauberes Beenden eines Threads

- **Initialisierungsphase:** Eigener Thread-Kontext einrichten.
- **Arbeitsphase:** Mehrere Aufgaben nacheinander erledigen und Attribut `isStopped` regelmässig überprüfen.
- **Aufräumphase:** Benutzte Objekte und Ressourcen werden in einen konsistenten Zustand gebracht.

```
// ...  
@Override  
public void run() {  
    // Initialisierungsphase  
    while (isStopped() == false) {  
        // Arbeitsphase  
    }  
    // Aufräumphase  
}
```

Codeskizze

Überprüfung von `isStopped` ist in eine separate Methode ausgelagert.

# Beenden eines Threads durch Interrupt

- In der vorherigen Codeskizze wurde vorsorglich **interrupt** aufgerufen.
- Die Methode **interrupt** setzt ein Interrupted-Flag des Threads, das eine Unterbrechungsanforderung signalisiert.
- Die Methode **isInterrupted** liest das Interrupted-Flag des Threads, gibt sie **true** zurück, wenn das Interrupted-Flag gesetzt wurde.
- Zusätzlich gibt es noch die Thread Klassenmethode **interrupted**. Sie stellt den Wert des Interrupted-Flag beim aktuellen Thread fest.
  - Das **Gefährliche** an dieser Methode ist, dass sie nach dem Aufruf das Interrupted-Flag immer auf **false** zurücksetzt!

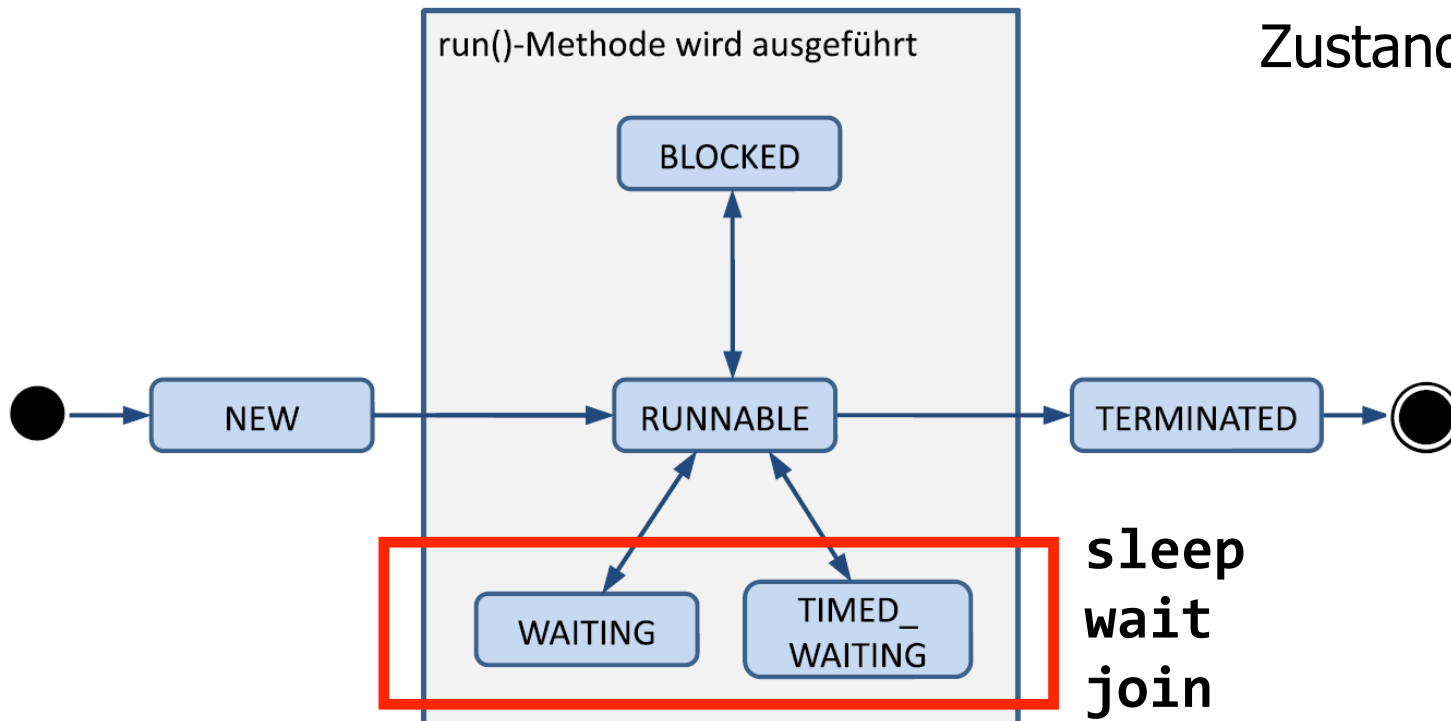
# Was passiert bei einem Interrupt?

- Befindet sich der Thread in einer blockierten Wartemethode, wird er durch **interrupt** geweckt.
- Ist er nicht im Wartemodus, stösst aber im weiteren Verlauf auf eine Wartemethode, wird diese gleich nach Betreten wieder verlassen.
- In beiden Fällen wird eine **InterruptedException** geworfen.
- Folgende Wartemethoden lösen eine **InterruptedException** aus
  - **sleep**
  - **wait**
  - **join**

# Der komplette Lebenszyklus eines Threads

- Ein Thread kann den WAITING- und TIMED\_WAITING-Zustand durch **interrupt** verlassen.
- Aus dem BLOCKED-Zustand kann der Thread durch **interrupt** nicht heraus geholt werden.

Zustandsdiagramm



# Sicheres Beenden eines Threads durch Interrupt-Status

- **Wichtig!** Mit dem Auslösen der Exception wird das Interrupt-Flag wieder auf `false` gesetzt.
  - ggf. muss das Interrupt-Flag wieder explizit gesetzt werden!

Codeskizze

```
// ...
@Override
public void run() {
    // Initialisierungsphase
    try {
        while (Thread.currentThread().isInterrupted() == false) {
            // Arbeitsphase
        }
    } catch (InterruptedException ex) {
        // Thread wurde in einer Wartemethode unterbrochen
    } finally {
        // Aufräumphase
    }
}
```



# Reaktion auf InterruptedException

- Keine gute Idee: Nichts tun.
  - leerer `catch` Teil
- Exception abfangen und terminieren.
  - üblich ist mindestens eine Exception Meldung auszugeben oder einen Log Eintrag machen, z.B. mit `LOG.info(...)`
  - im `catch` Teil die Beendung des Threads einleiten; z.B. ein `return` oder (besser) nochmals die Methode `interrupt` aufrufen, denn das **InterruptedException-Flag wurde zurückgesetzt!**
- Exception abfangen und eventuell nicht terminieren, wenn eine angefangene Arbeitsphase beendet werden soll.
  - im `catch` Teil den Thread nur aufgrund einer Entscheidung beenden

# Zusammenfassung

- Die Masszahl für den Performance-Gewinn bei nebenläufiger Ausführung ist der Speedup  $S = \frac{T_{seq}}{T_{par}}$
- Für die Thread Erzeugung das Interface **Runnable** verwenden.
  - Konzeptuelle Unterscheidung zwischen dem Programmfluss (Thread) und der nebenläufig durchzuführenden Aufgabe (Task).
  - In der **run**-Methode werden die Anweisungen implementiert.
- Das Thread Objekt muss mit der Methode **start** gestartet werden.
  - Nur einmal möglich - ein erneutes Starten geht nicht mehr.
- Java verwendet das kooperative Beenden mit Hilfe von **interrupt**.
  - Der Thread sollte ordnungsgemäss seine **run**-Methode verlassen.
- **InterruptedException** immer abfangen und am besten das Interrupted-Flag erneut setzen.

**Fragen?**