Zusammenfassung

**Lernziele**Programmierung von parallelen und nebenläufigen Systemen

* Grundlagen der nebenläufigen und parallelen Programmierung (Threads, Synchronisation, Korrektheits- und Fairnessbedingungen, Thread Pools, asynchrone Programmierung, Speichermodelle) kennen und verstehen.
* Entwurf und Implementation von nebenläufigen und parallelen Programmen in modernen verbreiteten Technologien (z.B. .NET C#, Java) umsetzen können.
* Datenstrukturen, Algorithmen und Design Patterns zur effizienten Parallelisierung (Lock-Free/Wait-Free Data Structures, Recursive Parallel, Producer/Consumer u.a.) kennen und anwenden können.
* Weitergehende Concurrency-Modelle und Technologien (Actors/CSP, Software Transactional Memory, Cluster-Parallelisierung mit MPI, GPU-Parallelisierung) kennen und einsetzen können.

**Lerninhalte**

*Multi-Threading und Synchronisation*

* Einführung in die nebenläufige/parallele Programmierung und zugrundeliegenden Systemarchitekturen
* Multi-Threading mit .NET und/oder Java
* Kritische Abschnitte und Synchronisationsmechanismen
* Monitor-Konzept und deren Umsetzung in gängigen Sprachen
* Spezifische Synchronisationsprimitiven (Semaphore, Reader-Writer Locks, Latches, Barrieren u.a.)
* Korrektheits- und Fairnessbedingungen; Problematik von Race Conditions, Deadlocks und Starvations

*Thread Pools und effiziente Parallelisierung*

* Thread Pools: Mechanismus, Eignung und Limitationen
* Task- und Daten-Parallelität
* Parallele Algorithmen (Sortierung, Suchen etc.)
* Asynchrone Programmierung
* GUI und Nebenläufigkeit
* Entwurfsmuster der Nebenläufigkeit (Producer/Consumer, Concurrent Pipelines, Reader/Writer, Recursive Parallel)
* Speichermodelle: Atomarität, Sichtbarkeit und Optimierung
* Lock-Free & Wait-Free Datenstrukturen

*Fortgeschrittene Nebenläufigkeitsmodelle*

* Verteilte Parallelisierung mit Actors/CSP
* Cluster Computing mit MPI
* Software Transactional Memory
* Vektorparallelisierung; GPU / Coprozessor-Parallelisierung

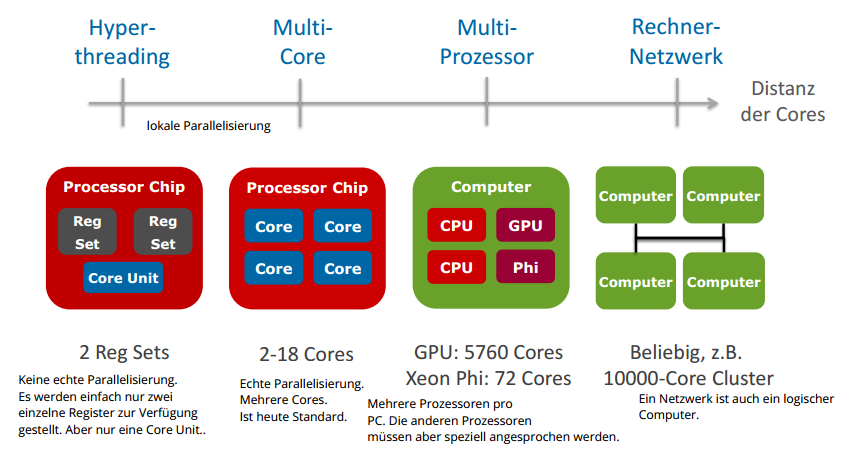
Übersicht des Modules

[Erste Woche 3](#_Toc475179144)

# Einführung

*Wieso eigentlich Parallele Programmierung?*  
Es führt zu einer Performance-Steigerung, da wir mehrere Prozessoren sowie die Netzwerkverteilung nutzen können. Zudem entspricht es der natürlichen Modellierung. Es gibt vieles das in einer asynchronen Ausführung stattfinden (wie Druckjob, Downloads, etc.). Wie auch ein Server mit mehreren Klienten-Sitzungen.

*«The Free Lunch is Over»*  
Bis 2003 hat circa alle 2 Jahre eine Verdoppelung der Prozessor-Taktraten stattgefunden. Damit wurden die Programme schneller ohne Code-Änderung. Seit 2003 stagnieren die Taktraten, dafür Hyperthreading und mehr Prozessorkerne. Wir müssen also die Programme so schreiben, dass Sie parallelisiert werden können.

Stufen der Parallelisierung  


*Parallelität (Parallelism)*Ist die Zerlegung eines Ablaufs in mehrere Teilabläufe, welche gleichzeitig auf mehreren Prozessoren laufen. Das Ziel sind schnellere Programme.

*Nebenläufigkeit (Concurrency)*Gleichzeitig oder verzahnt ausführbare Abläufe, welche auf gemeinsame Ressourcen zugreifen. Das Ziel ist es hier einfachere Programme zu haben.

Dasselbe Prinzip gilt für mehrere Threads/Prozesse, die interagieren.

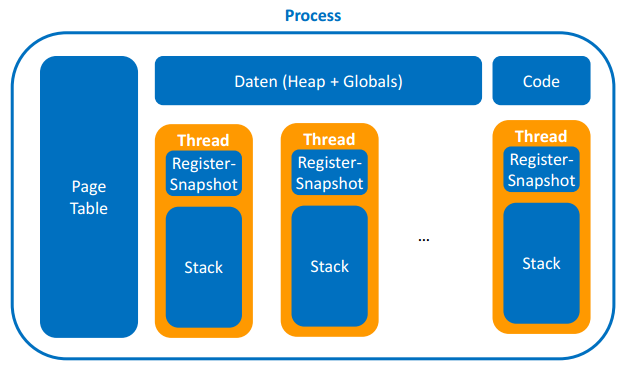
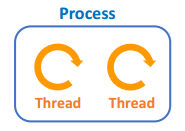
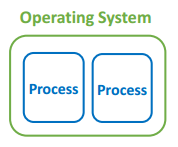
# Multi-Threading Grundlagen

## Parallelität des Betriebssystems

Es gibt dabei zwei verschiedene Arten von Parallelenabläufen.

Ein **Prozess** (Schwergewichtsprozess) ist eine parallel laufende Programm-Instanz im System. Pro Prozess ist ein eigener Adressraum vorhanden.

Ein **Thread** (Leichtgewichtsprozess) ist eine parallele Ablaufsequenz innerhalb eines Programms. Sie teilen sich den gleichen Prozess innerhalb des Prozesses.



### Speicherressourcen

Prozesse sowie jeder Thread brauchen auch Speicherressourcen. So muss zum Beispiel jeder Thread einen Stack abbilden.

### Thread-Implementationen

Es wird bei der Implementation zwischen zwei Arten von Threads unterschieden.

Der User-Level Thread ist im Prozess implementiert (Programm-Laufzeitsystem). Keine echte Parallelität durch mehrere Prozessoren.

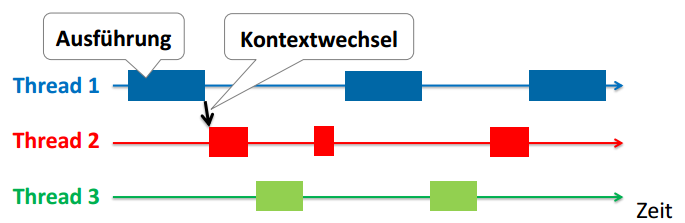
Der Kernel-Level Thread ist im Kernel implementiert (Multi-Core Ausnutzung). Der Kontextwechsel vom Prozess findet per SW-Interrupt (Kernel-Mode) statt.

Die Java-Threads der JVM werden eins zu eins auf Kernel-Threads abgebildet. Heutzutage wird in der Regel Kernel-Level Threading verwendet.

### Thread Scheduling

Es kommt zum Prozessor Sharing, da wir im Normalfall mehr Thread als Prozessoren ausführen. Bei einer Wartebedingung wird der Prozessor also an anderen bereiten Thread freigegeben.

### Prozessor Multiplexing

Gleichzeitig findet ebenfalls eine verzahnte Ausführung statt. Der Prozessor führt Instruktionen von mehreren Threads in abwechslungsweisen Teilsequenzen aus. Somit entsteht eine Illusion der Parallelität mehrerer Threads auch bei nur einem Prozessor (Quasiparallelität).

### Kontextwechsel

**Synchron (Warten auf Bedingung)**  
Thread wartet auf Bedingung. Reiht sich als wartend ein und gibt dann den Prozessor frei.

**Asynchron (Zeitablauf)**  
Nach einer gewissen Zeit soll der Thread den Prozessor freigeben. Somit wird verhindert, dass ein Thread dauerhaft den Prozessor belegt.

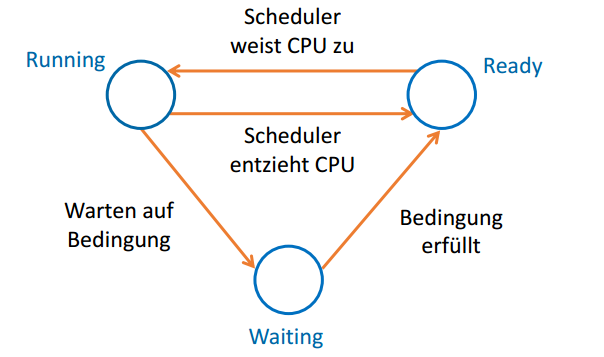
### Multi-Tasking

**Kooperativ**  
Threads müssen explizit beim Scheduler in Abständen Kontextwechsel synchron initiieren. Der Scheduler kann den laufenden Thread nicht unterbrechen.

**Preemptiv**  
Scheduler kann per Timer-Interrupt den laufenden Thread asynchron unterbrechen. Es findet Time-Sliced Scheduling statt. Jeder Thread besitzt den Prozessor für einen maximalen Zeitintervall.

Heutzutage in der Regel nur *preemptiv*.

### Thread Zustände



## Grundlagen der Java Threads

### Ausführungsmodell

#### JVM Thread Modell

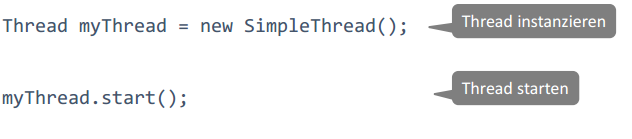
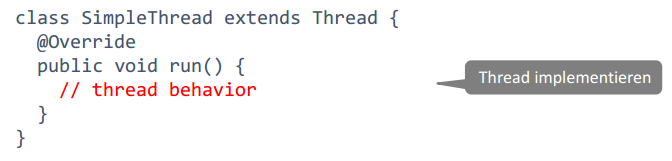
Java ist ein **Single Process System**. Die Java Virtual Machine (JVM) ist ein Prozess im Betriebssystem. Die JVM erzeugt beim Aufstarten den Main-Thread, welcher die statische Methode main() aufruft. Der Programmierer kann weitere Threads starten. Subsysteme /Laufzeitsysteme starten auch eigene Threads (z.B. Garbage Collector).

#### JVM Terminierung

Die JVM läuft, solange Thread laufen. Ausnahme sind dabei die Threads, welche als Daemon markiert werden. Die JVM warten nämlich nicht auf Daemon Thread (z.B. Garbage Collector). Daemon Threads brechen beim JVM Ende unkontrolliert ab. Eine Terminierung findet auch bei System.exit()/Runtime.exit() statt. Dies führt zu einer direkten Terminierung der JVM, was unsauber ist. Alle Threads werden unkontrolliert abgebrochen.

### Implementierung

#### Erzeugen eines Threads

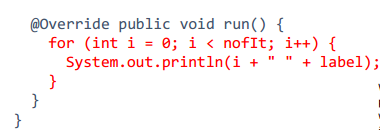
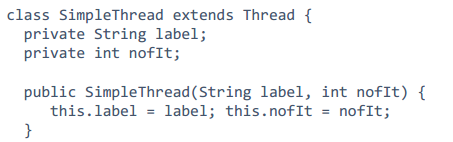
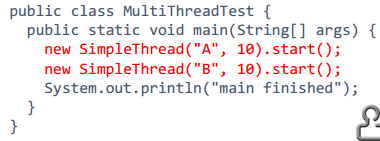


#### Start und Ende eines Threads

Ein richtiger Thread wird erst bei start() erzeugt. Start() führt die run()-Methode des Objektes aus.

Ein Thread endet beim Verlassen von run() und somit am Ende der Methode. Ein Return Statement kann mitgegeben werden. Zudem können unbehandelte Exceptions auftreten. Bei einer unbehandelten Exception laufen alle anderen Threads aber weiter.

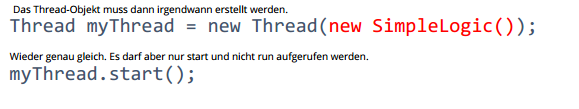
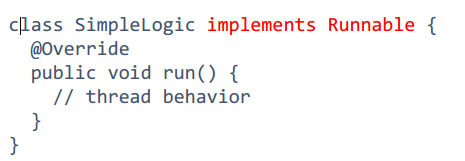
#### Multi-Thread Beispiel

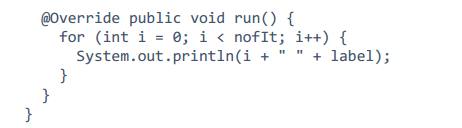
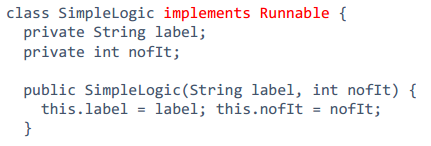
  
Pro Thread (run-Methode) wird eine Liste von 0A,1A und so weiter ausgeben. Wenn mit run() aufgerufen, dann passiert nichts nebenläufiges. Es werden dann keine Threads erstellt. Somit werden zuerst alle A’s, dann alle B’s und am Schluss die Ausgabe ausgegeben.

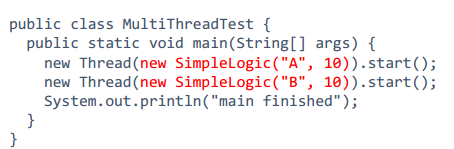
Mit start() sind A und B für sich geordnet. Es könnte aber auch sein dass alle A und dann alle B’s kommen. Die Ausgabe von Main kann auch irgendwann auftauchen. Die Reihenfolge kann sich daher von Ausführung zu Ausführung ändern 🡺 Nicht-Determinismus.

**Nicht-Determinismus**  
Threads laufen ohne Vorkehrung beliebig verzahnt oder parallel. Viele JVMs realisieren einzelne System Outputs ohne Verzahnung/Thread-Fehler – aber es ist nicht spezifiziert.

#### Runnable Implementierung

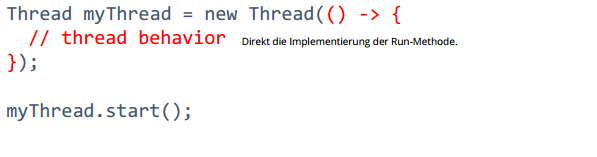
Es eine Alternativ zur Vererbung von Thread. Dies wenn andere Basisklasse benötigt wird (Single-Inheritance). Es findet eine Schnitstellenimplementierung des Interfaces statt.  


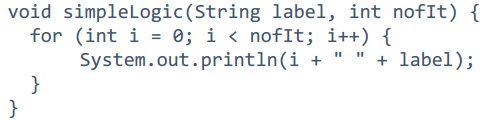
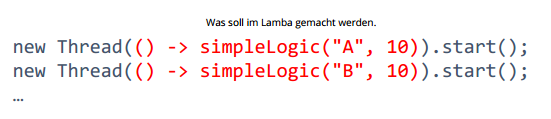
**Beispiel**  
Das Runnable Interface könnte man auch mit Lambdas benutzen.  




#### Kürzer mit Java Lambda

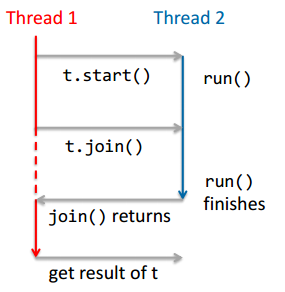
Es ist eine Ad-Hoc Implementierung einer Funktionsschnittstelle (1 abstrakte Methode). In diesem Fall ein Runnable Interface mit run(). Die Syntax ist (Parameterliste) => {Methodenrumpf}.

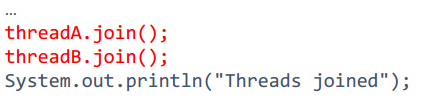
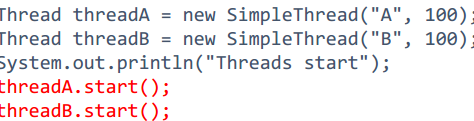


**Beispiel**  


Diese Variante ist komfortabler. Man braucht keinen Konstruktor um Parameter an die Run-Methode weiterzugeben. Grundsätzlich ist der gesamte Code um einiges einfacher.

### Join

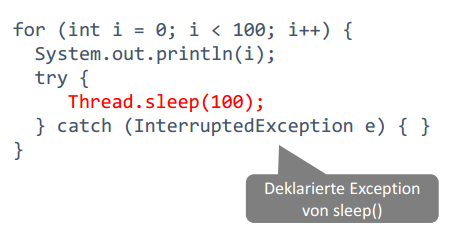
Ein Warten auf die Beendigung eines Threads. T.join() blockiert solange, bis t terminiert (Warten bis jemand anders fertig wird). Nach Join gilt !t.isAlive(). Der Zugriff auf die Variable von t ist erst nach dem Join sicher (z.B. für Resultate des Threads. Main Finished (Beispiel) soll somit erst kommen, wenn beide wirklich fertig sind. Beide Threads müssen dazu im Main gejoint werden.

**Beispiel**  


### Passivierung

Dies findet mit statischen Methoden der Thread-Klasse statt.

* **Thread.sleep(milliseconds)**
  + Der Laufende Thread geht in den Wartezustand. Nach Zeitablauf wird er wieder Ready. Muss dann aber zuerst wieder zugeordnet werden.
  + Er kann nur sich selbst schlafen legen und nicht die anderen. Mit Sleep sind die Verzahnungen meist besser.
* **Thread.yield()**
  + Der Laufende Thread gibt den Prozessor frei. Er wird aber direkt wieder ready.
  + Es provoziert mehr Thread-Wechsel und daher sollte eine stärkere Verzahnung resultieren (ist aber nicht immer so).

**Beispiel**  


### InterruptedException

Ist eine mögliche Exception bei blockierenden Aufrufen von sleep(), join() etc. Threads können auch von aussen unterbrochen werden mit myThread.interrupt().

Kooperatives Caneling sollte nur verwendet werden, wenn die Cancel-Policy des Threads bekannt ist. Oft wird die Exception zum Aufbrechen von Blockaden missbraucht. Meist hinterlässt dies inkonsistente Zwischenzustände oder die Exception wird ignoriert und es blockiert weiter.

### Weitere Thread Methoden

* Static Thread **currentThread**()
  + Gerade ausführende Thread-Instanz
* Void **setDaemon**(boolean on)
  + Thread als «Daemon» markieren (default ist false)
* Thread Name einstellen
  + String **getName**(), void **setName**(String name)
  + Thread(String name), Thread(Runnable r, String name)
  + Default: “Thread-0”, “Thread-1” etc.
  + Grundsätzlich nur für diagnostische Ausgaben von Vorteil

# Vorlesung Woche 2