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

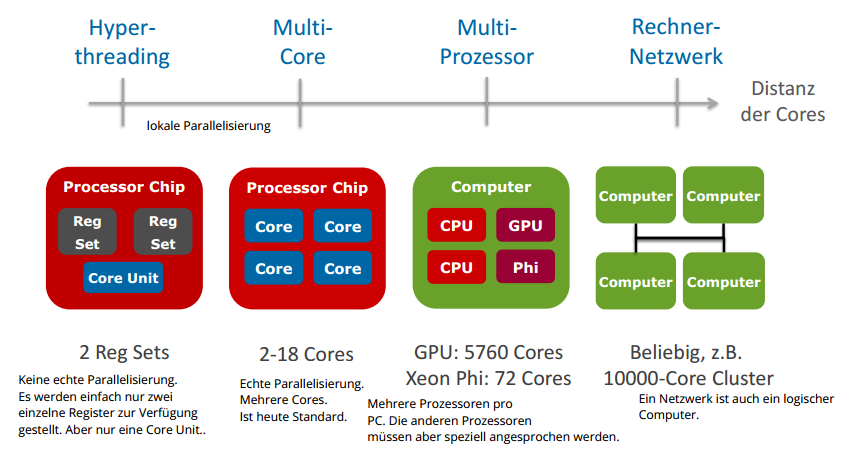
Übersicht des Modules

[Erste Woche Fehler! Textmarke nicht definiert.](#_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.

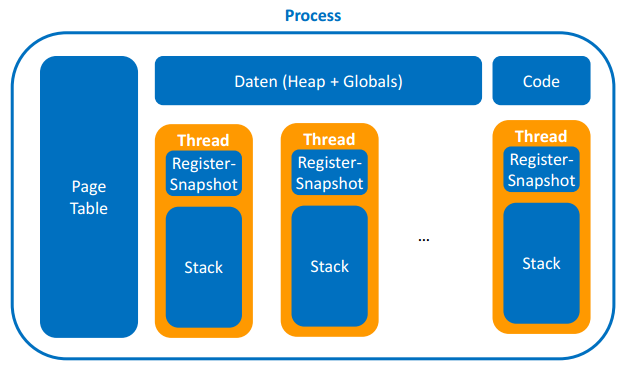
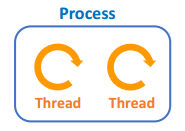
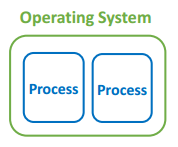
# Multi-Threading Grundlagen

## 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.

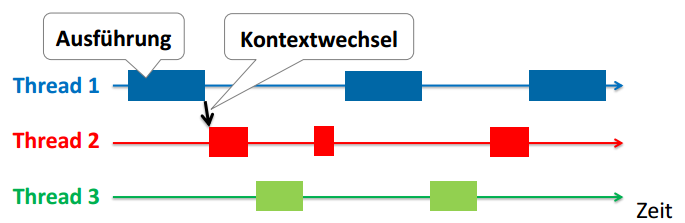
Der Kernel-Level Thread ist im Kernel implementiert (Multi-Core Ausnutzung). Der Kontextwechsel vom Prozess findet per SW-Interrupt (Kernel-Mode) statt.

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.

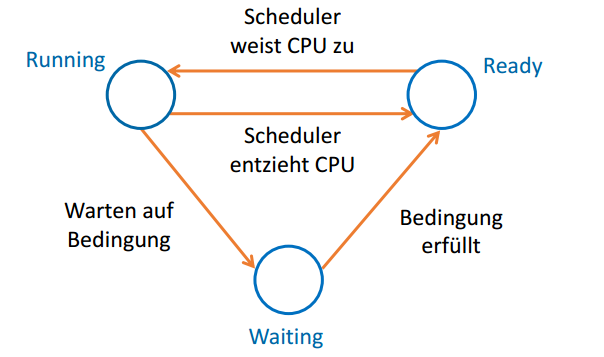
### Multi-Tasking

**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

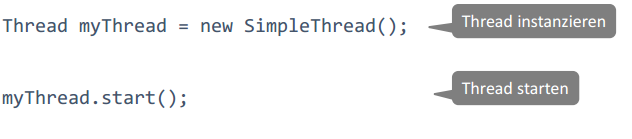
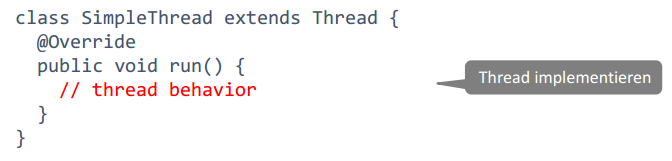
Java ist ein **Single Process System**. Die Java Virtual Machine (JVM) ist ein Prozess im Betriebssystem. Die JVM erzeugt beim Aufstarten den Main-Thread, welcher die statische Methode main() aufruft. Der Programmierer kann weitere Threads starten. Subsysteme /Laufzeitsysteme starten auch eigene Threads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

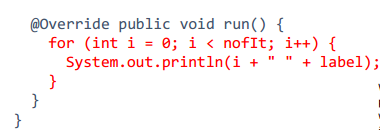
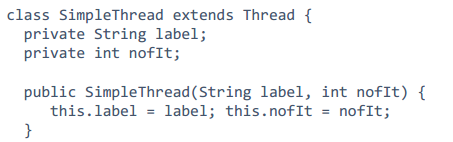
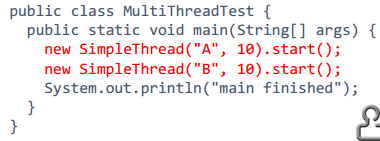


#### Start und Ende eines Threads

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.

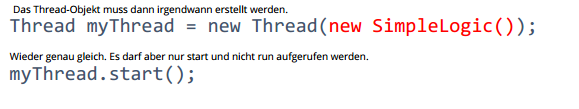
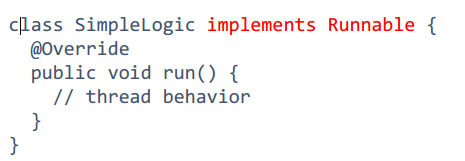
#### Multi-Thread Beispiel

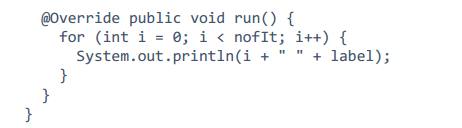
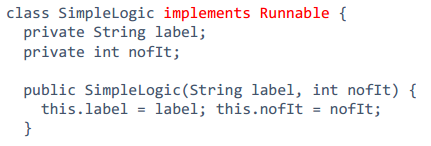
  
Pro Thread (run-Methode) wird eine Liste von 0A,1A und so weiter ausgeben. Wenn mit run() aufgerufen, dann passiert nichts nebenläufiges. Es werden dann keine Threads erstellt. Somit werden zuerst alle A’s, dann alle B’s und am Schluss die Ausgabe ausgegeben.

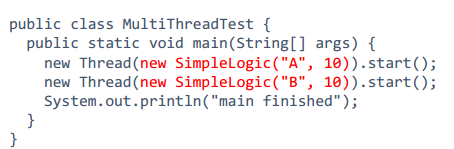
Mit start() sind A und B für sich geordnet. Es könnte aber auch sein dass alle A und dann alle B’s kommen. Die Ausgabe von Main kann auch irgendwann auftauchen. Die Reihenfolge kann sich daher von Ausführung zu Ausführung ändern 🡺 Nicht-Determinismus.

**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

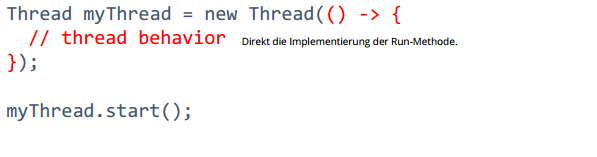
Es eine Alternativ zur Vererbung von Thread. Dies wenn andere Basisklasse benötigt wird (Single-Inheritance). Es findet eine Schnitstellenimplementierung des Interfaces statt.  


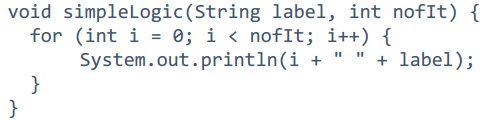
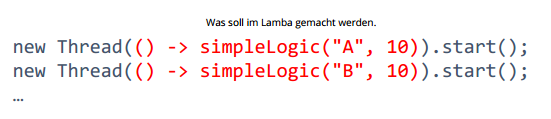
**Beispiel**  
Das Runnable Interface könnte man auch mit Lambdas benutzen.  




#### Kürzer mit Java Lambda

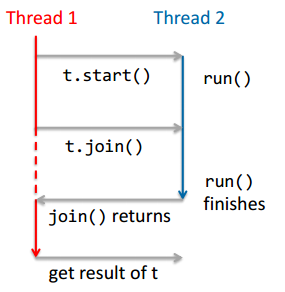
Es ist eine Ad-Hoc Implementierung einer Funktionsschnittstelle (1 abstrakte Methode). In diesem Fall ein Runnable Interface mit run(). Die Syntax ist (Parameterliste) => {Methodenrumpf}.

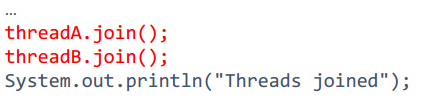
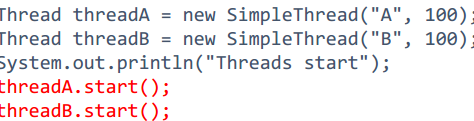


**Beispiel**  


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

### Join

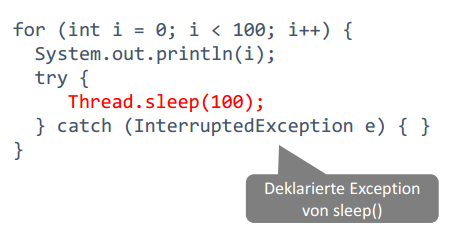
Ein Warten auf die Beendigung eines Threads. T.join() blockiert solange, bis t terminiert (Warten bis jemand anders fertig wird). Nach Join gilt !t.isAlive(). Der Zugriff auf die Variable von t ist erst nach dem Join sicher (z.B. für Resultate des Threads. Main Finished (Beispiel) soll somit erst kommen,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

# Monitor Synchronisation

Thread.currentThread().join() 🡺 Es wird herausgefunden, wer er gerade selbst ist. Wir rufen nun einen join auf uns selber auf und somit kommt es zu einem Stehenbleiben (ewig warten), da er ja auf sich selber wartet, bis er beendet wird. Dieser Fall wird aber nie eintretten.

## Thread Synchronisation

Ohne Vorkehrung laufen Threads beliebig verzahnt oder parallel. Oft muss aber die Nebenläufigkeit der Threads beschränkt werden. Die Synchronisation ist die Einschränkung der Nebenläufigkeit.

**Fälle der Synchronisation**  
*Gegenseitiger Ausschluss*  
Nur einer darf die Kaffeemaschine auf einmal benutzen. Oder nur eine Vorlesung zur gleichen Zeit im gleichen Raum.

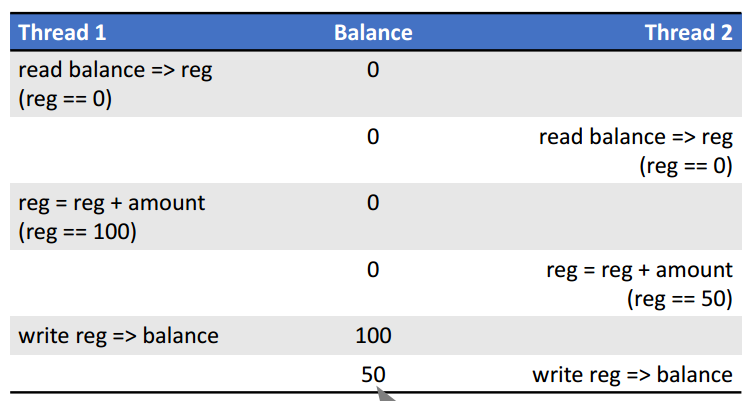
*Warten auf Bedingungen*  
Post erst abholen, wenn Sie angeliefert worden ist. Oder am Telefon erst reden, wenn die Gegenseite abgenommen hat.

### Gemeinsame Ressourcen

Threads teilen sich einen Adressraum und damit den Heap. Threads können daher gleichzeitig auf dasselbe zugreifen. Wie die Instanzvariablen, Statische Variablen, Elemente in einem Array oder lokale Variablen (in Java 8 nicht mehr). Die Zugriffe müsse daher genügend synchronisiert werden, sonst kommt es zu unkontrollierten oder falschen Interaktionen zwischen Threads (Race Conditions).

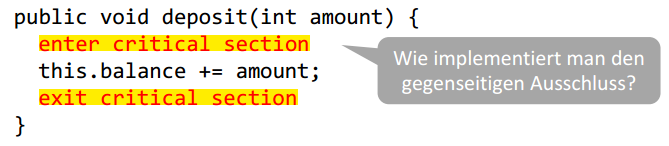
#### Race Condition Beispiel

|  |  |
| --- | --- |
|  |  |

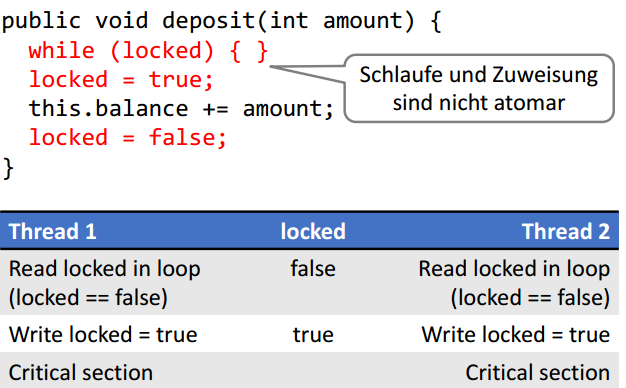
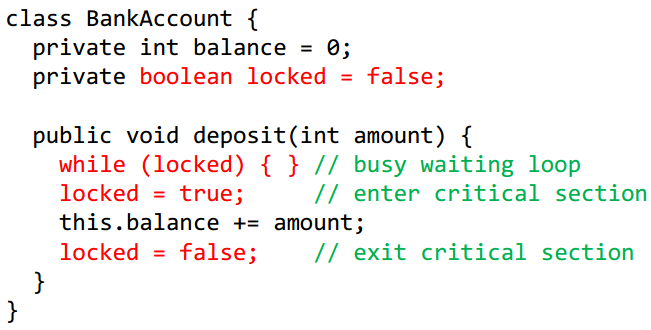
Folgende Tabelle zeigt, dass auch etwas anders resultieren kann. Und dies Obwohl wird eigentlich 150 erwartet haben. (Wir haben ja gesamthaft 150 eingezahlt.

### Kritische Abschnitte

Desposit() aus dem vorgehenden Beispiel ist der kritische Abschnitt (Critical Section). Dieser sollte nur von einem Thread zur gleichen Zeit ausführbar sein. Es braucht daher einen gegenseitigen Ausschluss (Mutal Exclusion).



#### Naiver Ansatz

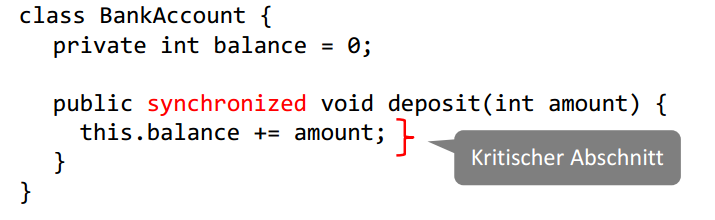


Bei dieser Implementation kommt es aber genau zum gleichen Fehler. Die Sachen sind nicht zusammengeschweisst und können aufgeteilt werden. Es findet kein gegenseitiger Ausschluss statt.

### Gegenseitiger Ausschluss

Die korrekte Implementierung ist nicht trivial. Algorithmen wie Dekker, Peterson oder Lamport’s Bakery. Es könnten auch Atomare Compare-and-Set (CAS) Instruktionen verwendet werden. Zudem muss auch die Weak Memory Consistency beachtet werden (Sehe ich Änderungen anderer CPU’s oder nicht, Memory Fences). Des Weiteren stehen noch Effizienzfragen im Raum. Busy Waiting zu teuer für lange Wartezeiten oder 1 CPU und die Warteschlange zu teuer für sehr kurzes Warten bei Multi-GPU. Daher sollten in jedem Fall die vorgefertigten Synchronisationsmechanismen verwendet werden.

#### Java Synchronized Methoden

Für Methoden steht der Modifier «synchronized» zur Verfügung. Der Body der Methode ist ein kritischer Abschnitt und wird unter gegenseitigen Ausschluss ausgeführt.

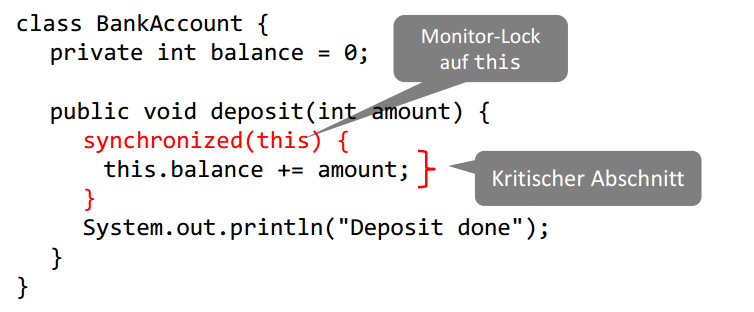
#### Funktionsweise von Synchronized

Jedes Objekt hat einen Lock (Monitor-Lock). Maximal ein Thread kann denselben Lock haben. Der synchronized Block belegt den Lock des Objektes. Bei Eintritt wird Lock besetzt – sonst warten, bis frei. Bei Austritt wird der Lock wieder freigegeben.

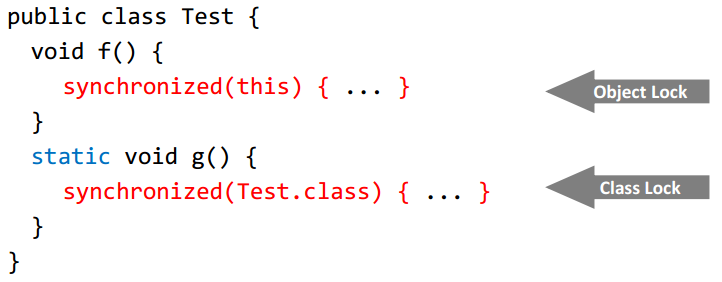
#### Beispiel Monitor Lock

#### Synchronized Statement Block

Mit synchronized (object) {statements} kann explizit angegeben werden, welche Instanz geblockt werden soll.



#### Äquivalente Verwendungen

Bei Arten sind gleichwertig und es ist egal wie man es verwendet.

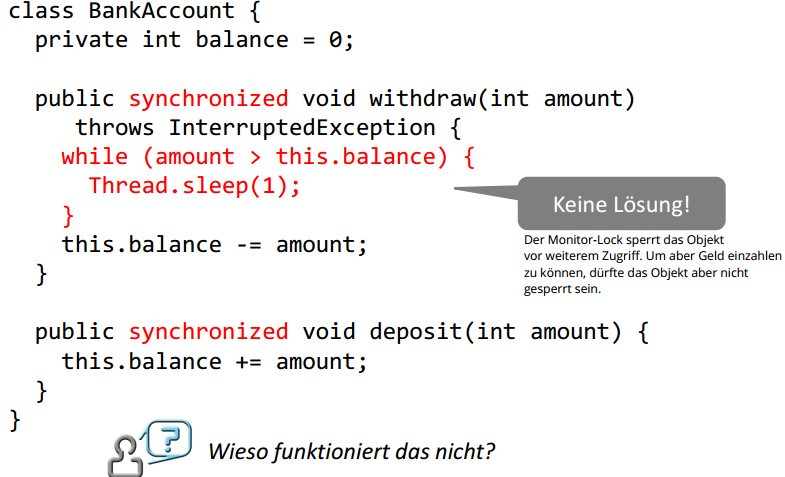
#### Exit aus Synchronized Block

Der Lock wird bei jedem Exit freigegeben. Ein Exit findet am Ende des Blocks statt, bei einem Return Statement oder bei einer Unbehandelten Exception.

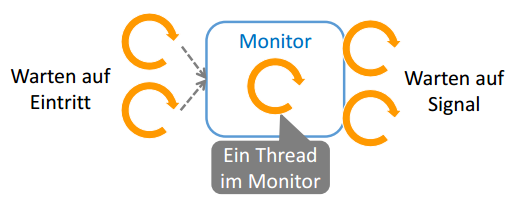
#### Rekursive Locks

Der gleiche Thread kann durch geschachtelte Aufrufe denselben Monitor-Lock mehrfach beziehen. Der Lock wird erst beim letzten Release freigegeben werden. Ich muss aber mitzählen wie tief ich bereits bin.

#### Naiver Ansatz – Warten auf Bedingung

Wieso? Der Monitor-Lock sperrt das Objekt vor weiteren Zugriff. Um aber Geld einzahlen zu können, dürfte das Objekt nicht gesperrt sein. Sleep() und yield() geben den Monitor-Lock nicht frei. Der andere Thread kann Bedingung gar nie erfüllen. Ein Pollen in Zeitabständen ist ineffizient. Viele unnötige Versuche (Kontextwechsel und Rechenzeit). Zudem hohe Verzögerung, bis die erfüllte Bedingung erkannt wird.

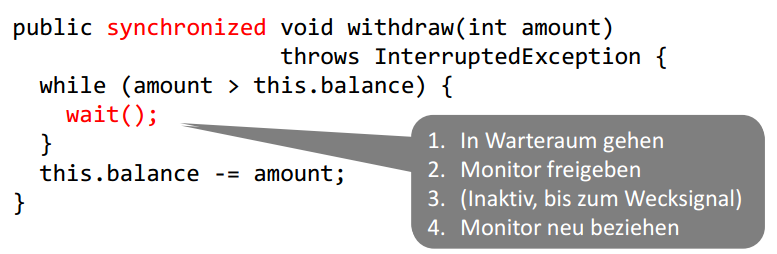
### Java Monitor Konzept

Dieses Konzept stammt von Brinch Hansen und Hoare. Das Objekt hat einen internen gegenseitigen Ausschluss. Nur ein Thread operiert zur gleichen Zeit im Monitor. Non-private Methoden sind alle synchronized, gearbeitet wird mit privaten Variabeln. Mit dem Wait & Signal Mechanismus können Threads um Monitor auf die Bedingung warten. Zudem können Threads die Bedingungen signalisieren und wartende Threads aufwecken.

#### Beispiel

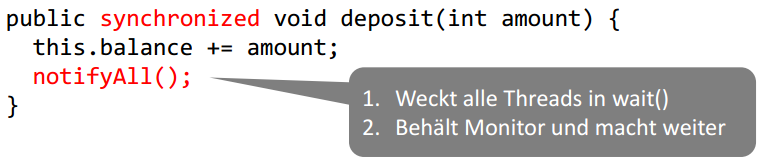
#### Wie funktioniert das?

Wait() gibt den Monitor-Lock temporär frei. Damit kann ein anderer Thread die Bedingung im Monitor erfüllen.

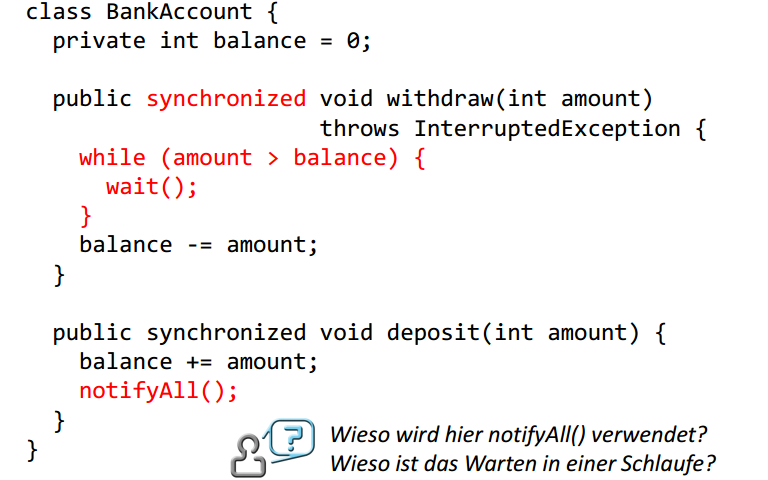


#### Wecksignal

Es wird vom siganlsieren einer Bedingung im Monitor gesprochen. Notify() weckt einen beliebigen wartenden Thread im Monitoir. notifyAll() weckt alle im Monitor wartende Threads. Kein Effekt, falls kein Thread wartet.



#### Analyse des Beispiels

Es wird notifyAll verwendet, da auch mehrere wartende Threads möglich sind.

Das Warten ist in einer Schlafe, da sonst die Rechenzeit wieder abgegeben wird.

### Wait & Signal

#### Pauschales Wait & Signal

Es gibt keine Unterscheidung zwischen verschiedenen semantischen Bedingungen. Wartende müssen selber schauen, ob sie ein Signal interessiert (ihre Bedingung erfüllt ist).

#### Java Monitor - Details

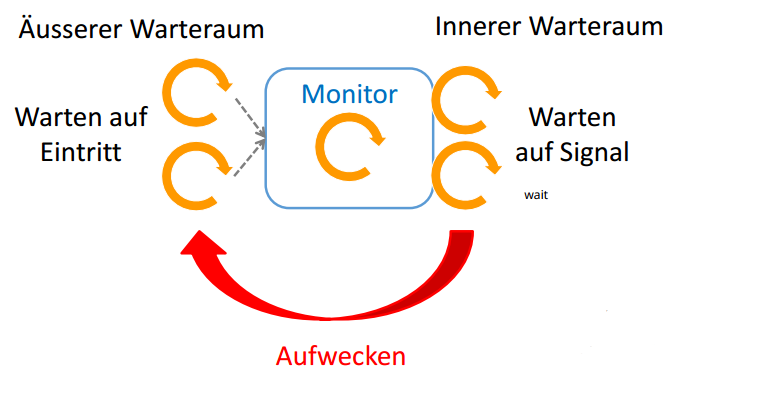
Wait(), notify() und notifyAll() nur in einem synchronized Block verwendbar. Sonst eine IllegalMonitorStateException.

Bei wait() mit rekursiven Locks werden alle gehaltenen Locks auf dem Objekt temporär freigegeben. Aber nur auf dem Objekt und nicht auf anderen Objekten.

#### Gründe zum Aufwachen aus dem wait()

* Notify(), notifyAll()
* InterruptedExcetpion
* Spurious Wakeup
  + Fälschliches Aufwecken in vereinzelten Betriebssystemen
  + Schlechtes Design 🡺 OS Implementierung vereinfacht, dafür Benutzung kompliziert.

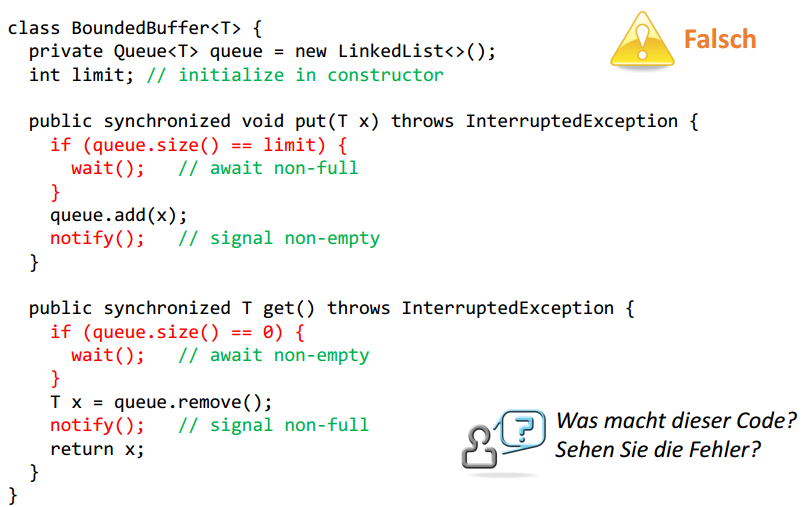
#### Java Monitor Funktionsweise

Wenn ich aufgeweckt werde, muss ich natürlich wieder neuanstehen. Er bekommt nicht direkt den Monitor.

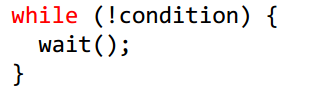
#### Signal and Continue

Der signalisierende Thread behält den Monitor. Nach notify()/notifyAll() läuft er im Monitor weiter. Der Aufgeweckte Thread kommt nicht direkt in den Monitor. Der Siganlisierende Thread ist ja noch drin. Daher der Kompromiss. Der kommt in den äusseren Warteraum. Der aufgeweckte Thread muss also neu um den Monitor-Eintritt kämpfen wie alle anderen eintrittswilligen Threads.

### Fallstricke beim Java Monitor

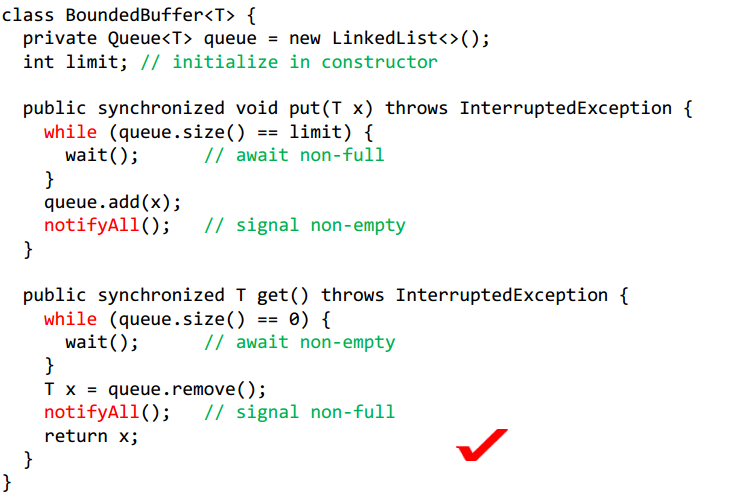


**Falle 1 – Wait mit if**  
Aufgeweckter Thread muss neu um Montor-Eintritt kämpfen. Anderer Thread kann vor ihm in den Monitor eintreten (ihn überholen) und seine Bedingung invalidieren. Eventuell kommt es auch zu einem Spurious Wakeup. Lösung 🡺 Eine Wartebeindung in der Schlaufe testen.



**Falle 2 – Single Notify**Mehrere semantische Warebedingungen «nicht leer», «nicht voll». Es können sich so Bedingung für beide Threads anstauen. Notify() weckt irgendein Thread im Warteraum auf. Der Aufgeweckte Thread wartet evtl. auf andere Bedingung. Es kommt dann kein Thread dran, der weitermachen kann.

* notifyAll() verwenden.

**Korrekte Lösung**  


#### Monitor Effizienz Problem

Bei mehreren Warebedingungen weckt notifyAll() prophylaktisch immer alle auf. Alle treten nacheinander in den Monitor ein. Für einen ist die Bedingung erfüllt, alle anderen rufen wieder wait() auf. Es kommt zu vielen Kontext-Wechsel und hohen Syncrhonisationskosten.

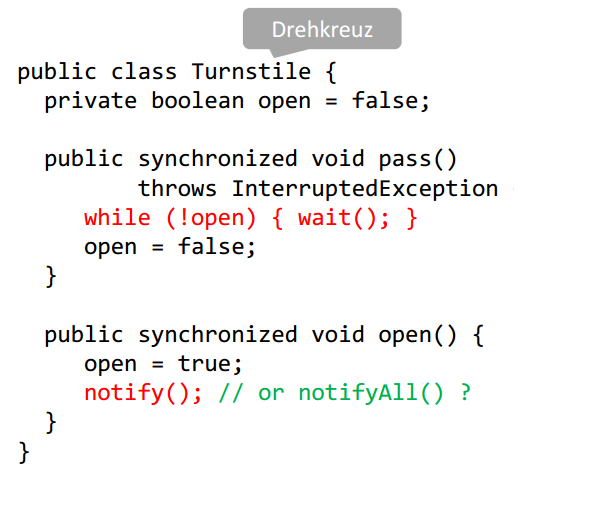
#### Monitor Fairness Probleme

* **Java** – Keine garantierte FIFO-Warteschlange
  + Notify() weckt irgendein Thread im Warteraum
  + Einige Threads können so nie dran kommen.
  + Auch ein Grund für notifyAll().
* **Signal-and-Continue**
  + Aufgeweckte Threads kommen in den äusseren Warteraum
  + Können kontinuierlich überholt werden
  + Auch bei FIFO-Warteschlange (.NET) nicht garantiert fair.

# Spezifische Synchrinisationsprimitiven

## Letzte Vorlesungswoche

### Quiz

Braucht es ein Notify? Braucht es die wait()-Schlaufe?

Ja ein Single Notify reicht aus, da wir nicht nicht mehrere Bedingungen haben. NotifyAll() haben wir damals eingesetzt, da mehrere Bedingungen vorhanden waren.

### Wann reicht ein Single Notify?

1. Nur eine semantische Bedingung (Uniform Waiters)
   1. Bedingung interessiert jeden wartenden Thread
2. Bedingung gilt jeweils nur für einen (One-In/One-Out)
   1. Nur ein einziger wartender Thread kann weitermachen

* Fairness-Problem in Java: Weckt beliebigen Thread. Dies ist der Grund für notifyAll() – garantiert aber auch keine Fariness.

### Monitor Diskussion

**Vorteile** Sehr mächtiges Konzept, jede Synchronisation damit realisierbar, OO Einbettung (Synchronisation in Objekte gekapselt).

**Nachteil** Nicht für alles optimal, muss Logik selber implementieren, Effizientprobleme (notifyAll() bei mehreren Conditions), Fairnessprobleme (Signal and Continue, kein FIFO)

## Semaphor

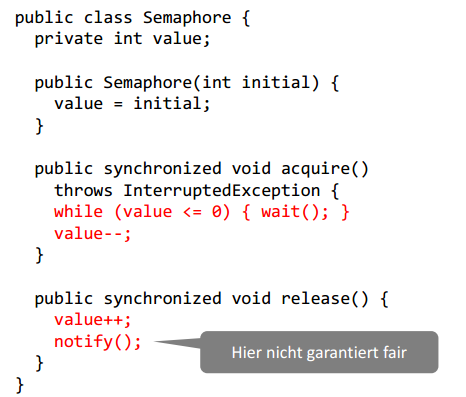
Das Konzept kommt von E.W. Dijkstra. Die Semaphor kommt vom Eisenbahn-Signal. Dabei handelt es sich um eine Vergabe einer beschränken Anzahl freier Ressourcen. (Freie Zugstrecke für einen, Freie Parkplätze für mehrere, Kuchenstücke für mehrere/einen).

Es ist ein Objekt mit Zähler. Der Zähler ist die Anzahl noch freier Ressourcen.

Methode **acquire**() bezieht die freie Ressoruce. Wartet, falls keine verfügbar ist (Zähler <= 0). Ansonsten wird der Zähler dekrementiert.

Methode **release**() gibt die Ressource frei und inkrementiert den Zähler.

### Nachbau mit einem Monitor



### Arten von Semaphoren

**Allgemeine Semaphore**   
Zähler zwischen 0 bis N, mit new Semaphore(N), Bis zu N Threads können gleichzeitig akquiriert haben, Für Quotas, Service Throttling etc.

**Binäre Semaphore**  
Zähler nur 0 oder 1, mit new Semaphore(1), für einen gegenseitigen Ausschluss (1=offen, 0=geschlossen).

In Java kann der Zähler auch negativ initialisiert werden. Nicht so in anderen Systemen (Windows, .NET).

### Faire Semaphore

Wird mit new Semaphore(N, true) erstellt. Es benutzt die FIFO-Warteschlange für eine Fairness. Diese Variante ist aber langsamer als die unfaire Variante. Default ist unfair.

### Anwendung

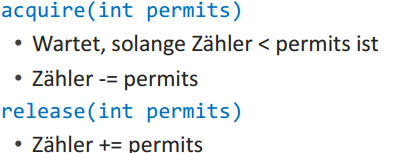
|  |  |
| --- | --- |
| Semaphore mit synchronized **Zeit**: 4 Sekunden | Nur mit Semaphore **Zeit**: 1.2 Sekunden (mit/oder ohne fairen Semaphoren). Diese Variante ist schneller da keine Schleifen benötigt werden und kein pauschales Aufwecken stattfindet. |

### In anderen Systemen

**Synonyme Operationen**  
acquire 🡺 decrement 🡺 wait 🡺 P («probeer”)  
release 🡺 increment 🡺 signal 🡺 V (“verhoog”)

Es gibt es auch im Betriebssystem für die Inter-Prozess Synchronisation (Betriebssysteme 1 und 2). In Java dient es der Intra-Prozess Synchronisation. Benutzt intern keine Betriebssystem-Semaphore da zu teuer, sondern eine übliche Thread-Synchronisation über den Heap.

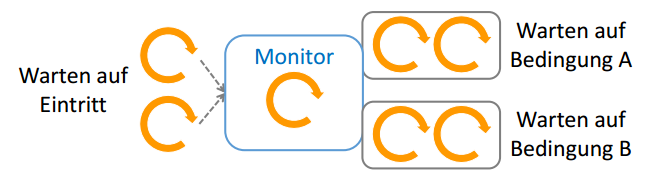
Es ist auch möglich mehr als eine Ressource anzufordern und freizugeben.



### Diskussion

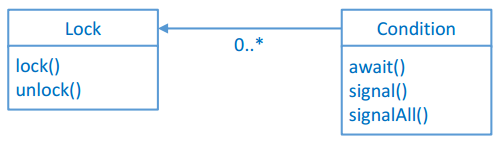
Es ist sehr mächtig. Beliebige Synchronisation implementierbar. Aber relativ low-level. Wir wollen beim Buffer die ineffiziente notifyAll() vermeiden. Zudem ist eine Signalisierung der spezifischen Bedindungen gewünscht. Nicht leer ⬄ lowerLimit > 0 und Nicht voll ⬄ upperLimit > 0

## Lock & Condition

Monitor mit mehreren Wartelisten für verschiedene Bedingungen.

Hintergrund  
Es soll unabhängig vom eingebauten Java Monitor sein. Also kein synchronized, wait(), notify(), notifyAll(). Sondern spezifische Synchronisationsprimitiven über die API. Diese kann benutzt werden, um eigenen Monitor zu bauen. 🡺 Package java.util.concurrent.locks.

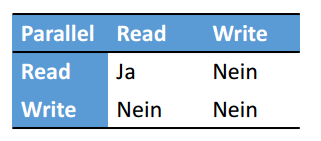
### Primitiven

Das Lock-Objekt ist eine Sperre für den Eintritt in den Monitor (Äussere Warteliste). Das Condition-Objekt ist «Wait & Signal» für bestimmte Bedingungen (also die Innere Warteliste). Mehrere Conditions sind pro Lock (daher Monitor) möglich.

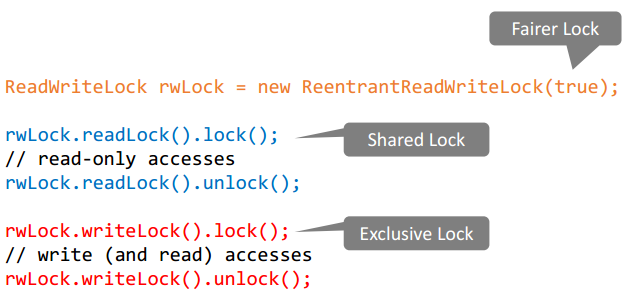
### Buffer mit Lock & Conditions

|  |  |
| --- | --- |
|  |  |

## Read-Write Lock

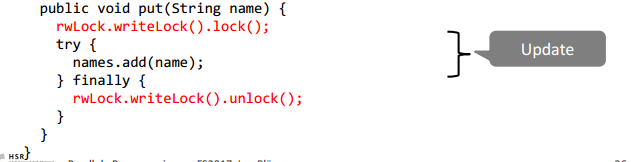
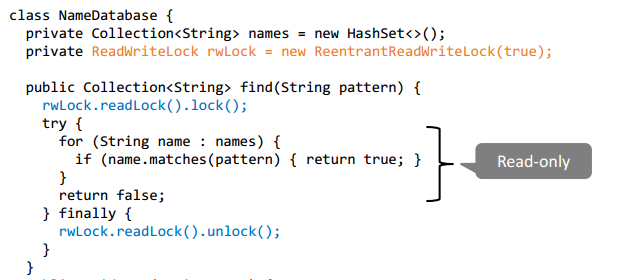
Ein gegenseitiger Ausschluss ist unnötig streng für rein lesende Abschnitte. Read-Write erlauben parallele Lese-Zugriffe (Reader), beim Schreib-Zugriff findet aber ein gegenseitiger Ausschluss statt (Writer).

### Verwendung

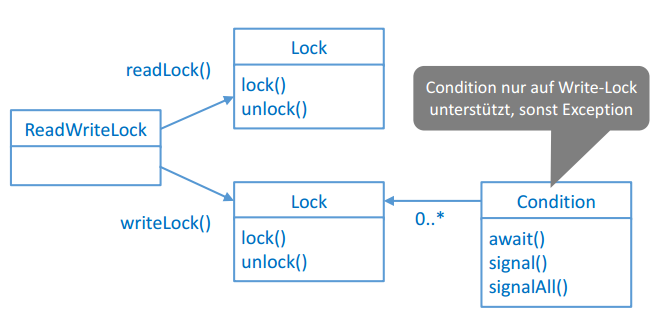


Falls ein schreibender Zugriff in Abschnitt involviert ist, muss der Write Lock verwendet werden.

### Beispiel



### Read-Write Lock mit Conditions

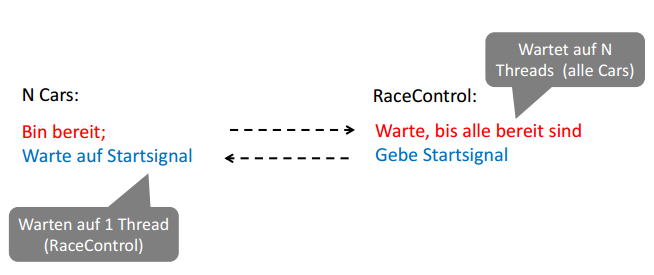
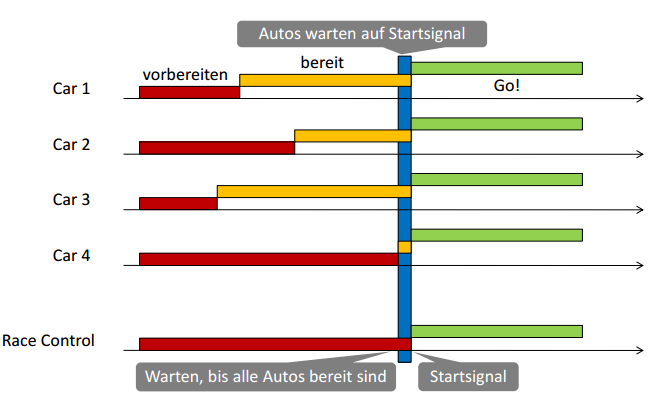
Macht es Sinn, die Condition nur auf Write-Locks zu gestatten? Beim Read macht man eigentlich gar keine Modifikation. Daher ja. Es gibt aber auch Fälle, wo der Reader auf etwas wartet. Daher nicht unbedingt immer.

## Zwischenstand

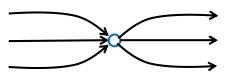
**Bisherigen Synchronisationsprimitiven**  
Schutz von Shared Ressourcen bei Multi-Threading (Monitor, Semaphore, Lock & Condition, Read-Write Lock)

**Weitere Synchronisationsprimitiven**  
Zeitlicher Synchronisationspunkt von mehreren Threads (CountDownLatch, CyclicBarrier, Pharser)

### Beispiel «Autorennen»



### Synchronisationspunkt

Anzahl von Threads warten auf eine Bedingung. Nach der Erfüllung der Bedingung laufen alle weiter. Beispiele: Autos warten auf Startsignal, Player warten alle aufeinander.

## Count Down Latch

Eine Synchronisationsprimitive mit Count Down Zähler. Threads können warten, bis Zähler <= 0 ist. Await() wartet das der Count Down 0 ist. Threads können auch runterzählen, mit countDown() wird der Zähler um 1 dekrementiert. Es gibt Ähnlichkeiten zur Semaphore. Diese blockiert bei <= 0, während der Count Down Latch bei >0 blockiert.

### Beispiel

|  |  |
| --- | --- |
| **Bespiel mit CountDown Latch** | **Analoge Lösung mit Semaphoren** |

### Details

Latches synchronisieren die Anzahl von Threads auf die Anzahl von Ereignissen. Ein Ereignis ist eine Count Down Operation, eine Synchronisation, ist ein Warten bis der Count Down Zähler <= 0 wird.

Auslösen der Ereignisse findet durch beliebige Thread statt, an einer beliebigen Stelle. Der countDown() blockiert nie.

Latches sind nur einmalig verwendbar. Der Count Down Zähler lässt sich nicht wieder hochsetzen. Neue Synchronisation == Neue Latch Instanz.

## Cyclic Barrier

Treffpunkt für eine fixe Anzahl von Threads. Die Threads warten, bis alle angekommen sind. Sie warten am Treffpunkt mit await(). Die Anzahl treffender Threads muss vorgegeben sein. Await() blockiert, bis so viele Threads await() aufgerufen haben wie es sollten. Der Cyclic Barrier ist wiederverwendbar. Therads können sich in mehreren Runden bei der gleichen Barriere synchronisieren.

|  |  |
| --- | --- |
| **Beispiel mit Cyclic Barrier** | **Wiederholte Barriere** |

Die Anzahl Teilnehmer wird beim Konstruktor festleget und ist nicht mehr änderbar. Mit int getParties() lässt sich die Zahl abrufen. Das Passieren der Barriere geschieht mit int await(). Als Rückgabe wert folgt die Anzahl noch fehlender Threads bei der Barriere. Wenn gleich 0 wird die Barriere geöffnet und die Wartenden aufgeweckt.

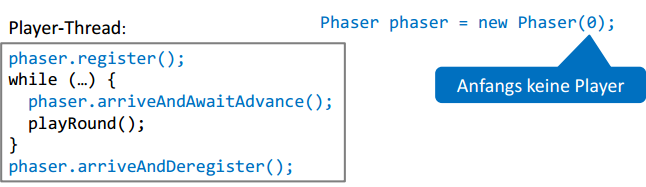
**«Broken Barrier»** Problem: Exception in await(), z.B. InterruptedException. Es sind dann alle betroffen und es kommt zu einer BrokenBarrierException.

### Latch versus Barriere



## Phaser

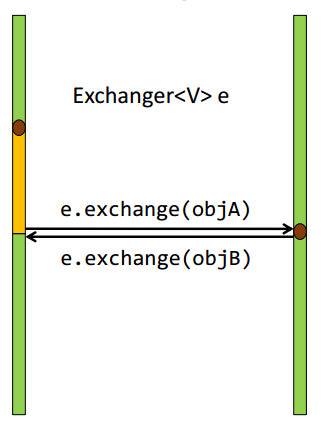
Verallgemeinerter Cyclic Barrier. Mit arriveAndAWaitAdvance() kann die Barriere passiert werden. Ein Teilnehmer kann sicher später anmelden bzw. abmelden. Mit register() bzw. arriveAndDeregister(). Dies wird erst bei der nächsten Warterunde wirksam.



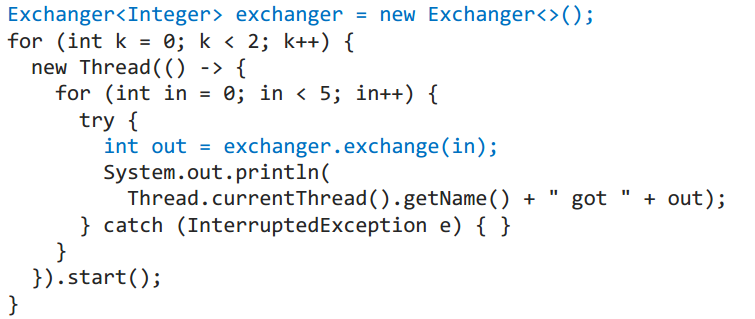
## Rendez-Vous

Ist eine Barriere mit Informationsaustausch, bei welchem nur 2 Parteien involviert sind. Zwei Threads treffen sich und tauschen Objekte aus. Ohne Austausch mit **new CyclickBarrier(2)** und mit Austausch **Exchanger.exchange(something).**

## Exchanger

V exchange(V x) blockiert, bis ein anderer Thread auch exchange() aufruft. Es liefert das Argument x des jeweils anderen Threads.

### Beispiel



# Gefahren der Nebenläufigkeit

Nebenläufige Programmierung birgt Risiko neuer Arten von Programmierfehler. Fehler, die es bei Single-Threading so nicht gibt. Sie können sporadisch oder selten auftreten und sind daher schwierig durch Tests zu finden.

## Unterschiede

### Java Monitor ⬄ Lock & Conditions

In L&C hat man mehrere Warteräume und kann damit gezielter Warten. Zudem ist technisch eine Fairness mit einer FIFO-Queue implementiert (ist aber nicht garantiert).

### Semaphore ⬄ CountDownLatch

Der CountDownLatch kann nur einmal verwendet werden (nicht reziklierbar). Beide zählen herunter. Bei der Semaphore ist die Zahl der Durchlässe fix. Die Fairness bei der Semaphore möglich und beim CountDownLatch immer automatisch gegeben.

### CountDownLatch ⬄ CyclicBarrier

Die CyclicBarrier ist reziklierbar

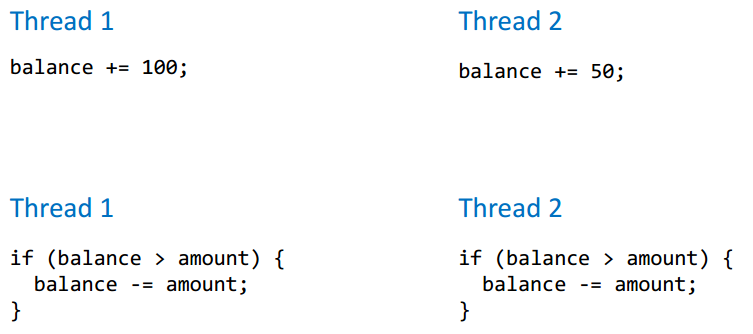
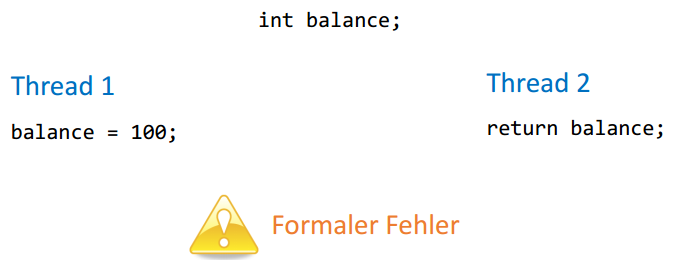
## Fehler der Nebenläufigkeit

### Race Conditions

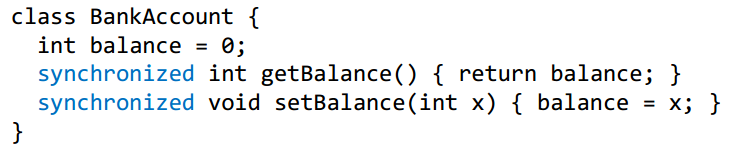
Mehrere Threads greifen auf gemeinsame Ressourcen ohne genügende Synchronisation zu. So sind falsche Resultate oder Verhalten möglich. Je nach Thread-Verzahnung und zeitlicher Ausführung. Ursache ist oft ein Data Race, aber nicht immer!

#### Data Race

Ein unsynchronisierter Zugriff auf den gleichen Speicher (dieselbe Variable oder Array-Element). Es ist mindestens ein schreibender Zugriff (Read-Write, Write-Read, Write-Write) vorhanden.



#### Race Condition ohne Date Race



**Mehrere Threads führen folgendes aus**  
account.setBalance(account.getBalance() + 100)

In diesem Falle sind Lost Updates möglich, da beide 0 + 100 machen und somit nur 100 und nicht 200 dabei herauskommt. (also kein atomare Inkrementieren).

Die Critical Sections sind dabei nicht geschützt. Die Low-Level Data Races sind zwar mit Synchronisation eliminiert, aber es sind nicht genügend grosse synchronisierte Blöcke.

#### Kombinationen

|  |  |  |
| --- | --- | --- |
|  | **Race Condition** | **Keine Race Condition** |
| **Data Race** | Fehlerhaftes Programmverhalten | Programm verhält sich zwar korrekt, dennoch formal falsch |
| **Kein Data Race** | Fehlerhaftes Programmverhalten | Richtig |

#### Einfach alles synchronisieren?

Nein, das hilft nichts. Race Conditions sind auch mit Synchronisation möglich. Weitere Nebenläufigkeitsfehler werden später noch gezeigt. Zudem entstehen Synchronisationskosten. Die Synchronisation ist relativ teuer. Zum Beispiel verhindert die Cache Invalidierung die Optimierung.

#### Verzichtbare Fälle der Synchronisation

In folgenden Fällen kann man auf eine Synchronisation verzichten.

* Immutability (Unveränderlichkeit)
  + Objekte mit nur lesendem Zugriff
* Confinement (Einsperrung)
  + Objekt gehört nur einem Thread zu einer Zeit.

##### Immutable Objects

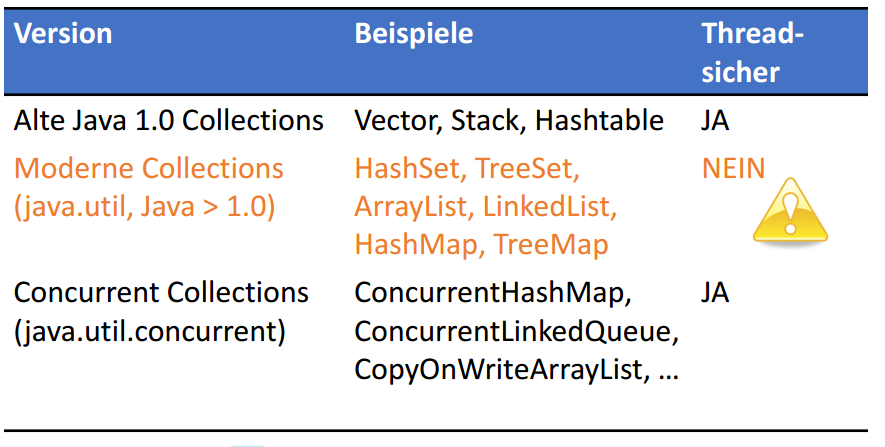
Dabei sind alle Instanzvariablen alle final (Primitive Datentypen oder Referenz auf wiederum Immutable Objekte). Die Methoden haben nur einen lesenden Zugriff. Der Konstruktor initialisiert die Instanzvariablen. Nach dem Konstruktor kann das Objekt ohne Synchronisation von Threads verwendet werden.

##### Confinement

Die Struktur garantiert, dass das Objekt nur durch einen Thread zur gleichen Zeit zugeriffen wird. Entweder durch Thread Confinement (Objekt nur über Referenzen von einem Thread erreichbar) oder über Object Confinement (Objekt in anderem bereits synchronisierten Objekt eingekapselt).

|  |  |
| --- | --- |
| **Thread Confinement** | **Objekt Confinement** |

#### Begriff «Thread-Safety»

Klassen / Methoden, welche intern synchronisiert sind. Also keine Race Conditions innerhalb des Codes und der kritische Abschnitt nur pro Methode erfüllt. Es gibt aber keinen kritischen Abschnitt über mehrere Methodenaufrufe und daher sind andere Nebenläufigkeitsfehler möglich.

Die Thread-Safety bei den Java Collections sieht so aus (Bild oben). Die Modernen Collections sind nicht thread-Sicher, da die Synchronisation oft nicht nötig ist und so unnötige Kosten entstehen. Zudem ist die Synchronisation meist ungenügend (Die Elemente und die Iteration der Elemente) ist nicht synchronisiert.

##### Concurrent Collections

Sind die effizienten Thread-sicheren Collections, welche für eine starke Nebenläufigkeit geeignet sind. Sie haben aber schwach konsistente Iteratoren. Ich sehe nebenläufige Updates bei der Iteration vielleicht nicht.

##### Achtung – Verstecktes Multi-Threading

* Finalizers (Laufen über separaten Finalizer-Thread)
* Timers (Handler durch separatem Thread ausgeführt, ausser GUI)
* Externe Libaries und Frameworks

Daher Gefahr von Race Conditions.

### Deadlocks

«Gegenseitiges Aussperren von Threads»

Definition  
Einige Threads sperren sich gegenseitig so, dass keinen von denen weitermachen kann. Ein Programm mit einem potentiellen Deadlock ist inkorrekt, da dieses Blockieren plötzlich stattfinden kann.

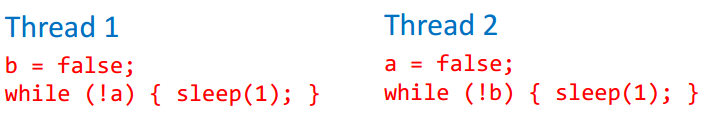
#### Deadlock – Nested Locks

|  |  |
| --- | --- |
|  |  |

#### Anderes Deadlock Beispiel

|  |  |
| --- | --- |
|  |  |

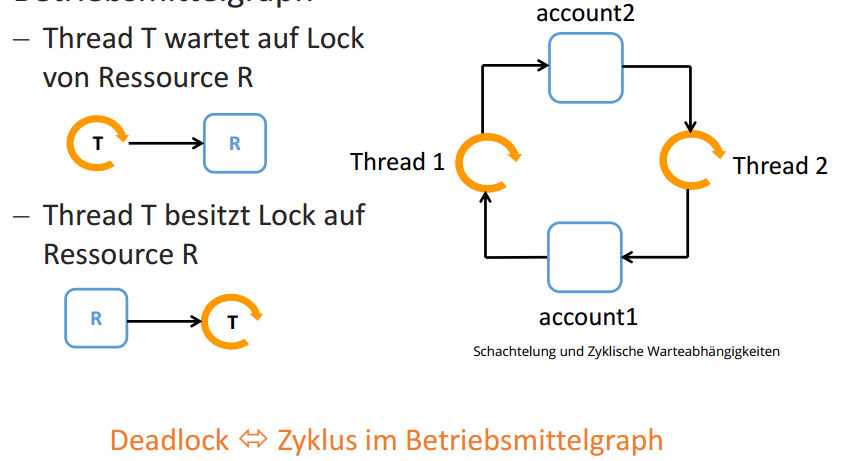
#### Spezialfall Livelocks



Threads haben sich gegenseitig permanent blockiert, führen aber noch Warteinstruktionen aus und verbrauchen somit CPU während dem Deadlock.

#### Erkennung

Dies lässt sich mit einem Betriebsmittelgraph erkennen.



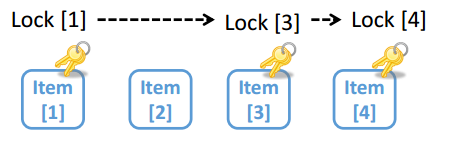
#### Voraussetzung

Alle der folgenden vier Bedingungen müssen zutreffen

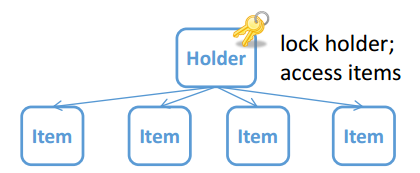
* Geschachtelte Locks
* Zyklische Warteabhängigkeiten
* Gegenseitiger Ausschluss (Locks)
* Sperren ohne Timeout/Abbruch

#### Vermeidung

**Lineare Ordnung der Ressourcen einführen**  
Also nur geschachtelt in aufsteigender Reigenfolge sperren. Zum Beispiel die Konten mit aufsteigender Nummer. Dies eliminiert zyklische Warteabhängigkeiten.



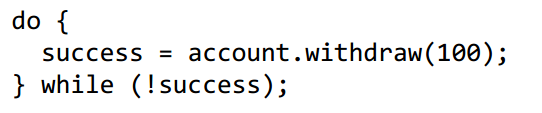
**Grobgranulare Locks wählen**  
Dies wenn lineare Ordnung nicht möglich oder sinnvoll ist. Zum Beispiel die gesamte Bank sperren bei einem Kontenzugriff. Es eliminiert die Schachtelung von Locks.



### Starvation

«Kontinuierliche Fortschrittsbehinderung von Threads wegen Fairness-Probleme»

Ein Thread kriegt nie die Chance, auf eine Ressource zuzugreifen. Dies obwohl die Ressource immer wieder frei wird (kein Deadlock oder Livelock). Andere Threads können ihn aber dauernd überholen und Ressource wegschnappen. Im folgenden Beispiel ist Starvation möglich:



Starvation ist ein Liveness/Fairnessproblem.

#### Vermeidung

Faire Synchronisationskonstruke  
Länger wartende Threads mit erfüllter Bedingung haben Vortritt. Dies kann zum Beispiel durch das Einschalten der Fairness bei der Java Semaphore, Lock & Condition sowie Read-Write Lock erreicht werden. Der Java Monitor hat aber ein Fairness-Problem. Er ist sehr starvation-anfällig, vor allem bei vielen Threads.

#### Thread Prioritäten

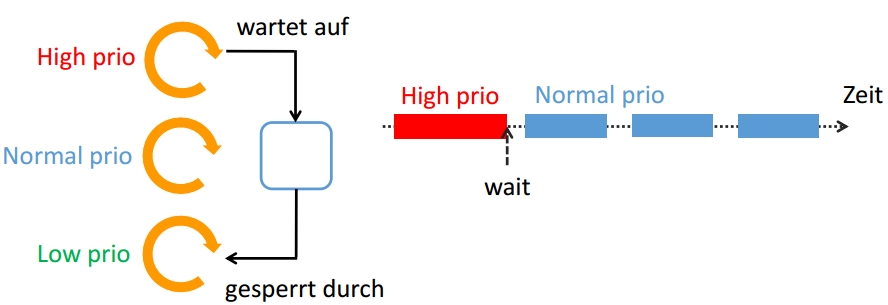
Die Prioritäten lassen sich mit myThread.setPriority(priority) setzen.  


Das Scheduling ist vom Betriebssystem abhängig. Meist haben hochprioritäre Threads immer Vorrang. Preemption findet unter Threads mit gleicher Priorität statt. Zu beachten ist, dass Windows nur 6 Prioritäten hat.

Starvation ist mit Thread Prioritäten dennoch möglich.

#### Priority Inversion

Ein Hoch prioritärer Thread wartet auf Bedingung von tief prioritärem Thread. Normale Threds können dazwischen laufen. Tief und hoch prioritärer Thread werden verhungern.



## Parallelität Korrektheitskriterien

**Keine Race Conditions (Safety)**  
Kritische Abschnitte auf gemeinsame Ressourcen sind genügend synchronisiert.

**Keine Deadlocks (Safety)**Threads können sich nicht gegenseitig für unbeschränkte Zeit sperren.

**Keine Starvation (Liveness)**Wenn ein Thread auf eine Bedingung wartet, soll er nach einer bestimmten Zeit fortschreiten können, sofern die Bedingung genügend oft erfüllt wird.

# Thread Pools

## Thread-Pool

### Konzept und Funktionsweise



**Task Queue**  
Tasks implementieren potentiell parallele Arbeitspakete. Auszuführende Tasks werden in Warteschlange eingereiht.

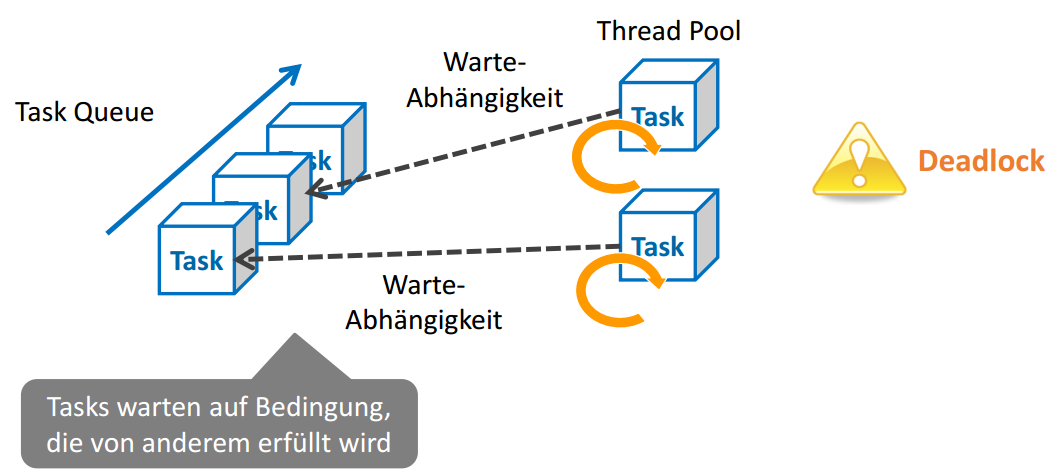
**Thread Pool**  
Eine beschränke Anzahl von Worker-Threads. Sie holen Tasks aus der Warteschlange und führen Sie aus.

### Vorteile

* **Beschränke Anzahl von Threads**
  + Denn zu viele Threads verlangsamen das System oder überschreiten den verfügbaren Speicher.
* **Recycling der Threads**
  + Spare Thread-Erzeugung und Freigabe
* **Höhere Abstraktion**
  + Trennung der Task-Beschreibung (Problem Space) von der Task-Ausführung (Machine Space).
* Die **Anzahl von Threads pro System sind konfigurierbar** 🡺 #Worker Threds = #Prozessoren + #I/O Aufrufe.

Somit entsteht ein neuer «Free Lunch» wenn man Programme mit Tasks modelliert laufen diese automatisch schneller auf parallelen Maschinen und ermöglichen eine Ausschöpfung der Parallelität ohne hohe Thread-Kosten.

### Einschränkungen

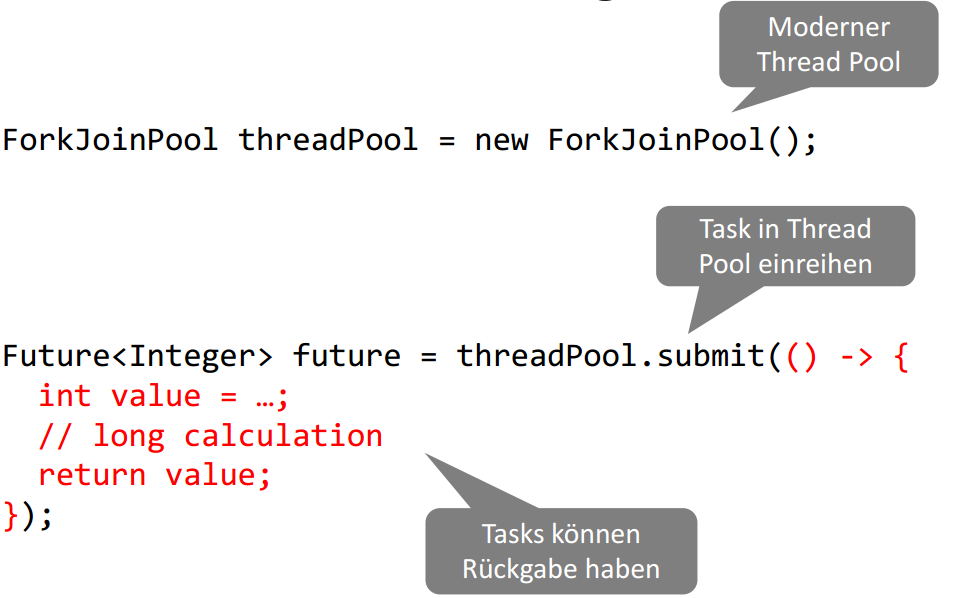


Die Tasks dürfen nicht aufeinander warten. Denn der Task muss zu Ende laufen, bevor der Worker Thread anderen Taks ausführen kann. Die Ausnahme sind hier die geschachtelten Tasks (Sub-Tasks).

### Fork & Join Pool

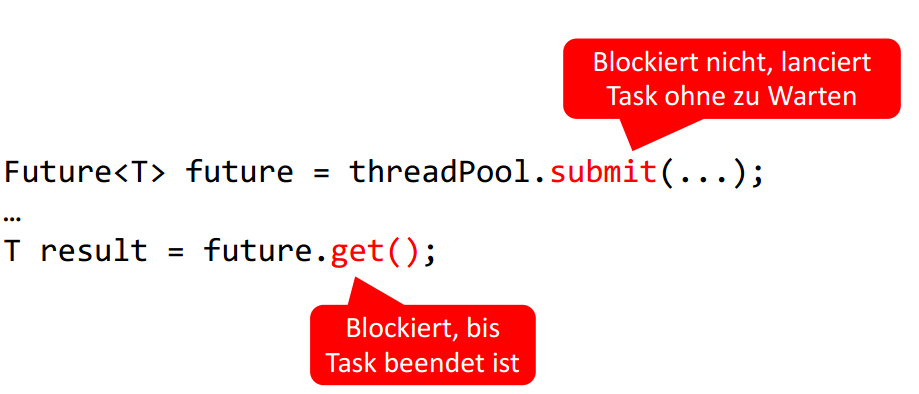
Der Fork-Join Pool (seit Java 7 & () unterstützt rekursive Aufgaben und hat eine effiziente Implementierung. Die einfachen Executors (seit Java 5) haben keine rekursiven Aufgaben und sind nicht besonders optimiert.

#### Lancierung



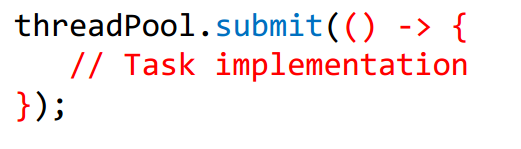
#### Future

**Konzept**  
Das Future repräsentiert ein zukünftiges Resultat. Es ist ein Proxy auf ein Resultat, dass evtl. noch nicht bekannt ist, weil die Berechnung noch läuft. Es muss das Ende der Berechnung abgewartet werden, bevor das Resultat zurückgegeben wird.

**Verwendung**  


**Details**  
*Fehler Propagierung*  
Wenn der Task mit einer unbehandelten Exception beendet wird, lässt sich mit get() dann die ExcetionException lieferen. Die ursprüngliche Exception ist darin geschachtelt (Cause).

*Task Abbruch*  
Dieser findet mit cancel(boolean mayInterruptIfRunning) statt. Dabei wird der Tasks aus der Warteschlange herausgenommen. Dies geschieht kooperativ (Bricht den laufenden Task nicht einfach ab). Optional gibt es einen Interrupt auf den Worker-Thread bei laufendem Task.

**Fire and Forget**  
Tasks starten, ohne später das Resultat abzuhölen. Der Submitter interessiert sich dabei nicht für das Resultat/Task-Ende. Zum Beispiel für einen Task ohne Resultat. Die unbehandelten Exceptions im Task werden ignoriert.  


**Beispiele**

|  |  |
| --- | --- |
| **Zählen von Primzahlen (Sequentiell = Ausgangslage)** | **Paralleles Zählen** |
| Mit nur 2 Mal parallel ist natürlich das Optimum noch nicht erreicht. Dafür weitere parallelisierungsschritte ist daher die Rekursion geeignet. | |
|  |  |

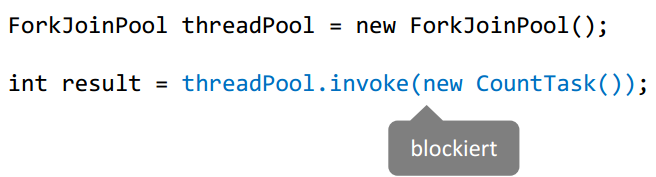
#### Rekursive Tasks

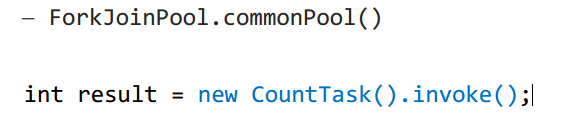
Diese Tasks können Untertasks starten und abwarten. Sie sind verboten in den alten Java Thread Pools, da es dort zu Deadlooks kommt.

Die Tasks erben von RecursiveTask<T> und haben daher folgende Methoden.

* T compute() – Task Implementierung
* fork() – Starte als Sub-Task in einem anderen Task
* T join() – Warte auf Task-Ende und frage Resultat ab
* T invoke() – Ein Sub-Tasks starten und abwarten
* invokeAll() – Mehrere Sub-Tasks starten und abwarten

Falls es kein Rückgabewert hat, kann man auch RecursiveAction verwenden. Dann mit void anstatt T.

**Ausführung**  
*Expliziter Thread-Pool*  


Standard Pool (Java 8)  


**Konkreter Ausbau**

|  |  |
| --- | --- |
|  | Aber Achtung. Keine Über-Parallelisierung. Sonst wird es wieder langsam. |

#### Reihenfolge



#### Internals

Die Fire-And-Forget laufen evtl. nicht zu Ende, denn die Worker Threads laufen als Daemon-Threads.

Der automatische Parallelitätsgrad  
Standardmässig entspricht die Anzahl der Worker Threads im Pool der Anzahl der Prozessoren. Nachher findet dynamisches Hinzufügen und Wegnehmen von Threads statt. Das Ziel ist es immer genügend aktiv/laufende Worker-Threads zu haben, dies ist jedoch nicht garantiert.

Der Common Pool verhindet Engpässe durch zu viele Thread Pool. Der Parallelitätsgrad ist zum Teil tiefer als die Anzahl der Prozessoren.

#### Work Stealing Verfahren

Verteile Queues führen zu weniger Contention. Somit streiten sich weniger Threads um die gleichen Looks und sind viel effizienter.

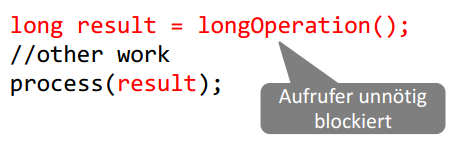
Die globale Queue funktioniert nach dem FIFO-Prinzip und verteilt zwischen den verschiedenen Queues.

Neue Sub-Tasks gelangen nach dem LIFO-Prinzip zuvorderst in die lokale Task-Queue. Dies ist der Grund für die Schachtelung der fork/joins.

## Asynchrone Programmierung

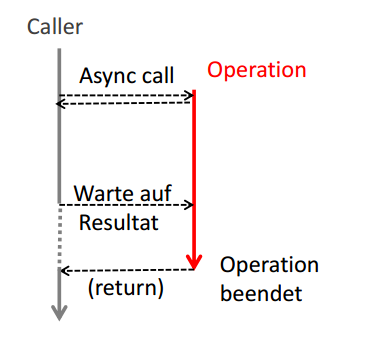
### Unnötige Synchronität

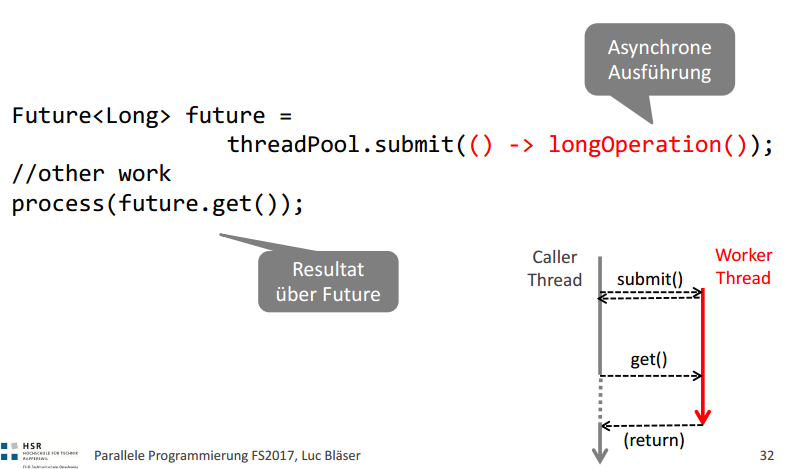
Meisten sind vielfach unnötige blockierende Methodenaufrufe vorhanden. Zum Beispiel bei langlaufenden Rechnungen oder I/O Aufrufen.



### Asynchroner Aufruf

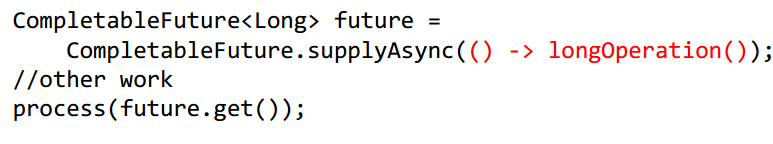
Der Aufrufer soll während der Operation weiterarbeiten können.

Klassisch würde die Operation in einen Thread oder einen Thread Pool auslagert werden.



### Moderne Asynchronität (Java 8)

Da wird die asynchrone Aufgabe im Standard Pool gestartet. runAsync(), falls es kein Rückgabetyp gibt.

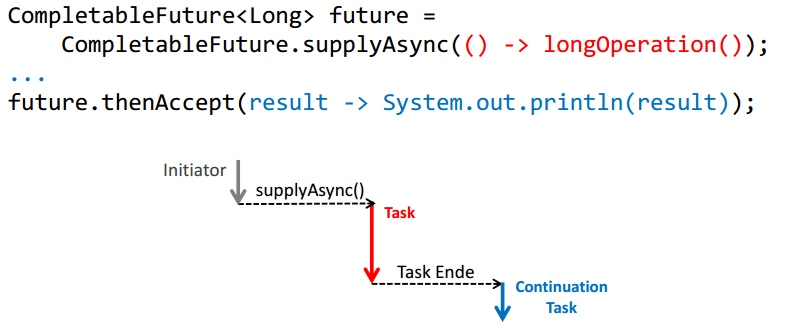


### Ende des asynchronen Aufrufs

**Caller-zentrisch (Pull)**  
Der Caller wartet auf das Task Ende und holt sich das Resultat. Z.B. das Future abfragen.

**Callee-zentrisch (Push)**  
Asynchrone Operation informiert direkt über das Resultat. Z.B. Completion Callback (Continuation, Promise).

### Continuation

Dabei wird eine Folgeaufgabe an eine asynchrone Aufgabe angehängt. Die Ausführung findet statt, sobald der vorgängige Task fertig ist.

## Task Continuations

### Ausführung der Continuation

Findet durch einen beliebigen Thread statt. Initiator, wenn das Future bereits ein Resultat hat.

**thenAccecpt()** für Handler ohne Rückgabe  
**thenApply()** für Funktion mit Rückgabe

* Continuation läuft potentiell in einem anderen Thread. Daher die Synchronisation beachten.

### Continuation-Style Programming

Die Tasks verketten via deren CompletableFuture.

### Multi Continuation

