Skript
Informatik B

Objektorientierte Programmierung in Java

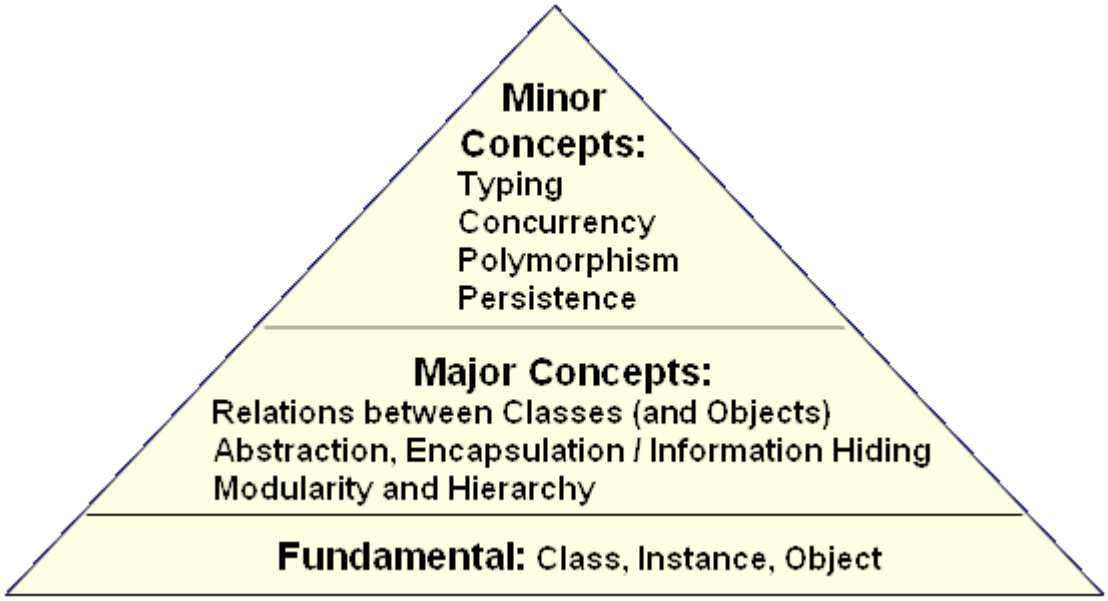
Sommersemester 2011

- Teil 7 -

Inhalt

… wird schrittweise erweitert

Kapitel 9:
Nebenläufigkeit



[Boo94]

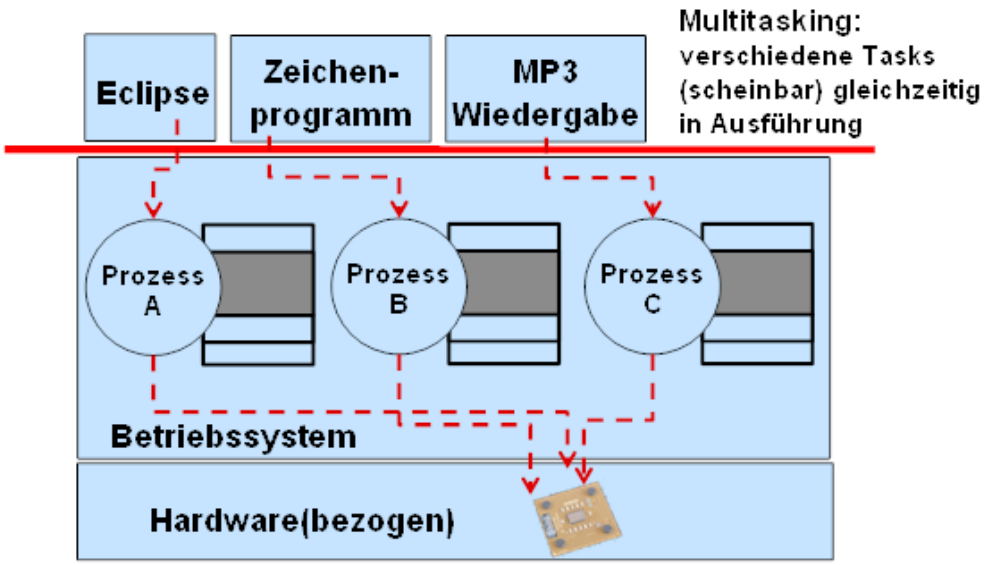
9.1 Grundlagen der Nebenläufigkeit

Nebenläufigkeit: Ereignisse sind nebenläufig, wenn sie in keiner kausalen Beziehung zueinander stehen.
Bemerkung: Kausale Beziehung meint, dass ein Ereignis Ursache des anderen ist.

Nebenläufige Ereignisse sind daher parallelisierbar.
Achtung: Obwohl eindeutig zu unterscheiden, werden die Begriffe “nebenläufig” und “parallel” umgangssprachlich oft synonym verwendet.

Nebenläufigkeit in der OO-Welt:
- Objekte sind weitgehend zueinander unabhängige und damit nebenläufige Einheiten. Die Grundidee der OO ist die Vorstellung von einem Programm, das aus vielen, untereinander unabhängig operierenden Objekten besteht, die sich von Zeit zu Zeit Nachrichten zusenden.
- Objekte können außerdem intern selbst nebenläufige Aktionen beinhalten.

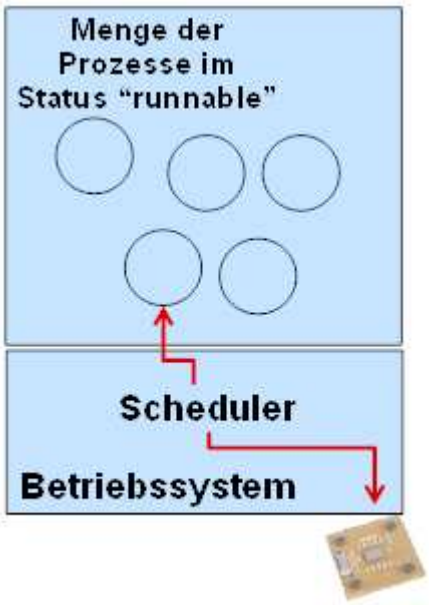
Ausnutzung von Nebenläufigkeit
a) durch Einsatz von Threads oder
b) in verteilten Systemen.



- Multitasking: Es können mehrere Tasks (scheinbar) gleichzeitig ausgeführt werden. Eine Task (Programm) kann auf einen oder mehrere Prozess(e) abgebildet werden.

- Multiprocessing: Mehrere Prozesse können gleichzeitig aktiv laufen.
  Der *Scheduler* (Ablaufplaner) bringt die Prozesse auf die Prozessoren. In der Regel gilt: Anzahl Prozesse > Anzahl Prozessoren.

- Einprozessormaschine: Prozesse laufen nur scheinbar gleichzeitig, d.h. wir haben eine sogenannte Pseudoparallelität bzw. Quasi-Parallelität.



Der Scheduler schaltet zwischen lauffähigen Prozessen hin und her. Das Umschalten geschieht meist ca. alle paar Millisekunden. Der Scheduler entscheidet je nach internem Algorithmus welcher Prozess wie lange läuft (Einplanung). Es gibt verschiedene Einplanungsalgorithmen. Durch die wechselnde Ausführung der Prozesse wird dem Benutzer eine parallele Ausführung vorgespielt. Von echter Parallelität sprechen wir, falls mehrere Prozessoren bzw. CPUs (Central Processing Unit) zur Verfügung stehen und die Verarbeitung wirklich zeitgleich läuft.

9.2 Threads

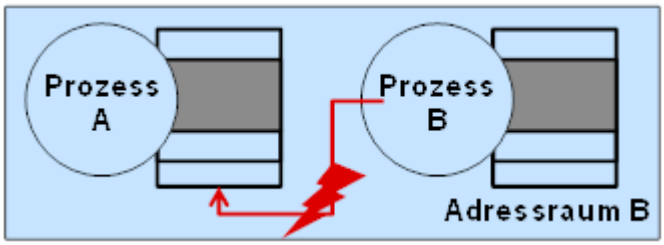
Prozess: Ein Programm in Ausführung bildet sich auf einen Prozess auf Betriebssystemebene ab. Ein Prozess ist eine elementare Ablauf- und Verwaltungseinheit im Rechner und besteht unter anderem aus Programmcode, Daten (im zugeordneten Adressraum) und Kontext. Der Prozesskontext umfasst Informationen wie z.B. Programmzähler, Prozessstatus, Priorität, Registerbelegung, belegte Ressourcen wie z.B. geöffnete Dateien oder belegte Schnittstellen.

Bemerkung:
Register sind spezielle Speicherbereiche, die innerhalb des Prozessors direkt mit der eigentlichen Recheneinheit verbunden sind und vom Prozessor für die Recheneingabe, -ausgabe und Zwischenspeicherung verwendet werden.

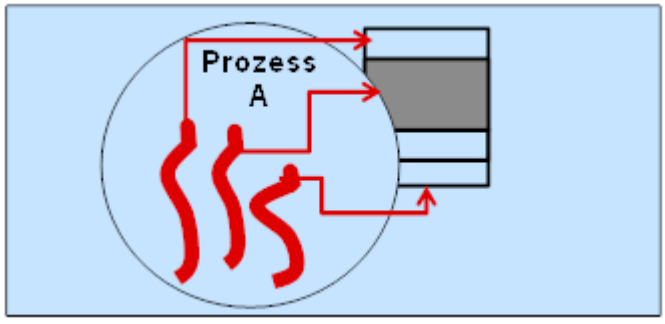
Prozesse sind also die im Betriebssystem laufende Repräsentation eines Programms. Sie genügen uns aber noch nicht, denn **für Prozesse gelten folgende Nachteile:**

- Der Kontextwechsel ist sehr aufwendig.
  Ein Kontextwechsel geschieht, wenn ein Prozess von der CPU genommen wird und ein anderer Prozess auf die CPU eingeplant wird und Rechenzeit erhält. Bei einem solchen Prozesswechsel muss unter anderem der Speicher (Register), mit dem die CPU arbeitet, für den aktuellen Prozess (d.h. mit den Daten im Adressraum des Prozess) neu belegt werden. Diese Speicherdaten sind Teil des Prozesskontext.

- Die Kommunikation zwischen Prozessen (Interprozesskommunikation) ist aufwendig.
  Jeder Prozess hat einen eigenen, getrennten bzw. geschützten Adressraum, auf den andere Prozesse nicht zugreifen können. Für die Trennung sorgt das Betriebssystem.
  Weil Prozesse nicht auf die Adressräume der anderen Prozesse zugreifen können, ist die Interprozesskommunikation zwar gut abgesichert aber aufwendig.



Abhilfe: Threads
Threads heißen auch Threads of Control, Fäden, Ausführungsstränge, leichtgewichtige Prozesse.
Für Threads ist ein schneller Kontextwechsel möglich (deshalb „leichtgewichtig“). Das kommt daher, dass z.B. deutlich weniger Verwaltungsinformationen in den Registern ausgetauscht werden müssen.



Mehrere Threads in einem Prozess laufen alle mit demselben Adressraum.
Threads eines Prozesses können jeweils auf die Daten dieses Adressraums zugreifen. Sie teilen sich die Ressourcen (geöffnete Dateien) und sind untereinander nicht geschützt.

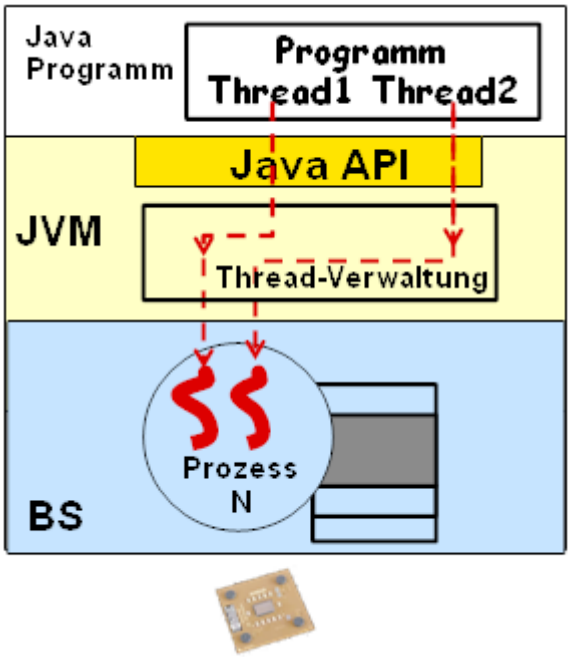
Beispiele für Einsatzgebiete von Threads

- Vermeidung von Blockierung und ungenutzten Wartezeiten:
  - z.B. eine Benutzerschnittstelle soll permanent auf Eingaben reagieren können,
  - z.B. blockierende Schnittstellen in der Netzwerkprogrammierung.
- Ereignisse, die nach einer bestimmten Zeit ausgeführt werden sollen (z.B. Diashow) und damit eine permanente Zeitüberwachung benötigen.
- Echte Parallelisierung zur Ausnutzung von Ressourcen (v.a. vorhandene Prozessoren).

Es erfordert vorherige Planung, um (Quasi-)Parallelität richtig auszunutzen.
Die Planung kann sehr aufwendig sein und erfordert teils spezielle parallele Algorithmen.

Threads in Java:

- Threads wurden ursprünglich in und für Betriebssysteme(n) eingeführt.
- In Java werden Threads auch auf höherer Sprachebene direkt als Sprachkonstrukt unterstützt.
- Die korrekte Funktionsweise dieser Threads ist in der Sprachspezifikation festgelegt, nicht aber das “Wie”.
- Umsetzungsalternativen:
  a. Simulation durch die JVM,
  b. Abbildung auf die nativen Threads des Betriebssystems (falls vorhanden).



Bemerkungen:

- Die meisten Betriebssysteme unterstützen heute Threads nativ (d.h. originär im Betriebssystem verankert).
- Die Java-Sprachspezifikation legt die genaue Umsetzung von Threads nicht fest, um die Unabhängigkeit zur Plattform sicherzustellen und gleichzeitig maximalen Nutzen von den Möglichkeiten der Plattform ziehen zu können.
- Ähnlich zu I/O müssen plattformspezifische Strukturen (hier: Threads) eine Repräsentation in den verschiedenen Abstraktionsschichten besitzen. Diese Abstraktionen kapseln die plattformspezifischen Gegebenheiten.

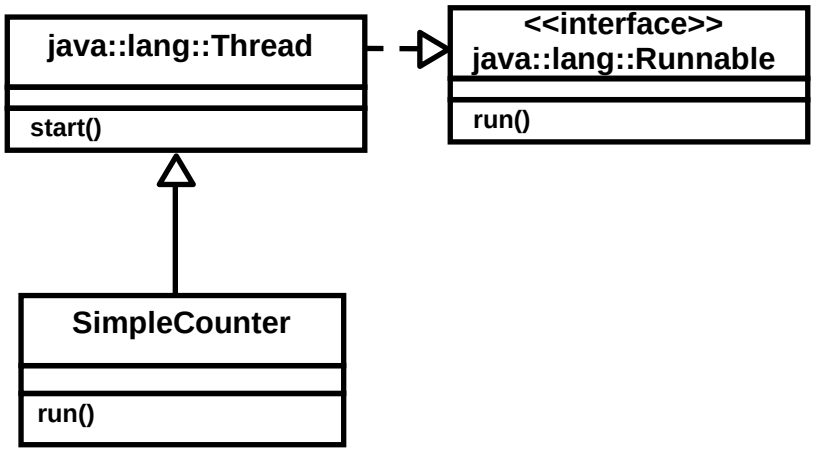
9.3 Java Threads

9.3.1 Wir bauen einen Java-Thread

In Java sind grundsätzlich drei verschiedene Wege zur Thread-Erzeugung möglich.

Variante 1: Vererbung

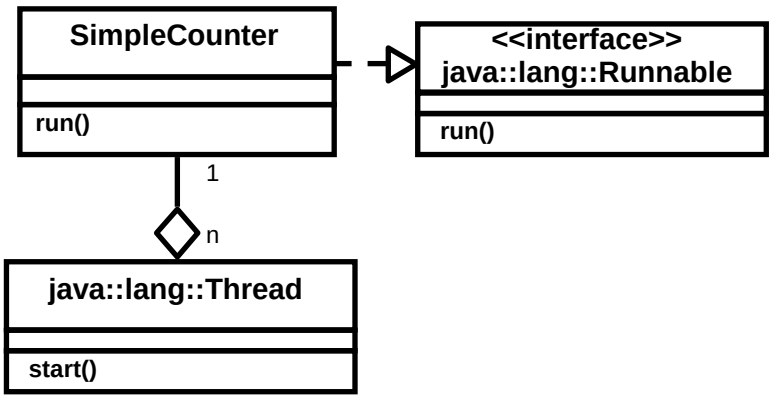
- Wir implementieren die nebenläufige Anweisungsfolge in einer run()-Methode einer Klasse (z.B. SimpleCounter).
- Diese Klasse erbt von Klasse Thread.



- Wir instanziieren das Objekt (das durch Vererbung auch ein Objekt vom Typ Thread ist) und starten die nebenläufige Aktivität mit start().

Variante 2: Aggregation

- Wir verpacken die nebenläufige Anweisungsfolge in einer run()-Methode eines Objekts vom Typ Runnable.
  Das Objekt mit der run()-Implementierung nennt man oft kurz "Runnable(-Objekt)".
  Im Beispiel ist das Runnable-Objekt eine Instanz der Klasse SimpleCounter.



- Dieses Objekt übergeben wir einem neuen Thread-Objekt (im Konstruktor).
- Die nebenläufige Aktivität des Thread-Objekts starten wir mit start().
- Bemerkung: Die Beziehung zwischen Thread und SimpleCounter könnte in Variante 2 auch eine Using-Beziehung sein (in Abhängigkeit von der Problemstellung).

Entwurfsentscheidung:
Im Sinne des Verantwortlichkeits- und Substitutionsprinzips sollte man in einem guten Entwurf die flexiblere Variante 2 bevorzugen. Das nebenläufige Arbeitspaket ist dabei gut gekapselt und unabhängig. Es wird zur Abarbeitung an einen Thread übergeben. Nur wenn wir wirklich die Thread-Funktionsweise an sich erweitern möchten, wählen wir die Vererbung (Variante 1).

start() und run():
In Java muss ein Thread mit start() gestartet werden. Nur dann sorgt der Thread dafür, dass der Programmcode innerhalb der run()-Methode des Runnable-Objekts (z.B. SimpleCounter) nebenläufig abgearbeitet wird.
Wird statt start() nur run() aufgerufen, ist die Abarbeitung nicht nebenläufig. Entwickler und Nutzer bemerken das eventuell nicht, da der Unterschied oft „nur“ in der Laufzeit des Programms liegt.

Einschränkungen für Variante 1 und 2:

- Methode start() kann an einem Thread-Objekt immer nur einmal aufgerufen werden, d.h. ein Runnable kann nicht mehrmals nacheinander gestartet werden:

```
Thread t = new Thread(SimpleCounter);
t.start(); t.start();
```

→ IllegalThreadStateException

- Es können nicht mehrere Runnable-Objekte nacheinander in einem Thread–Objekt abgearbeitet werden.

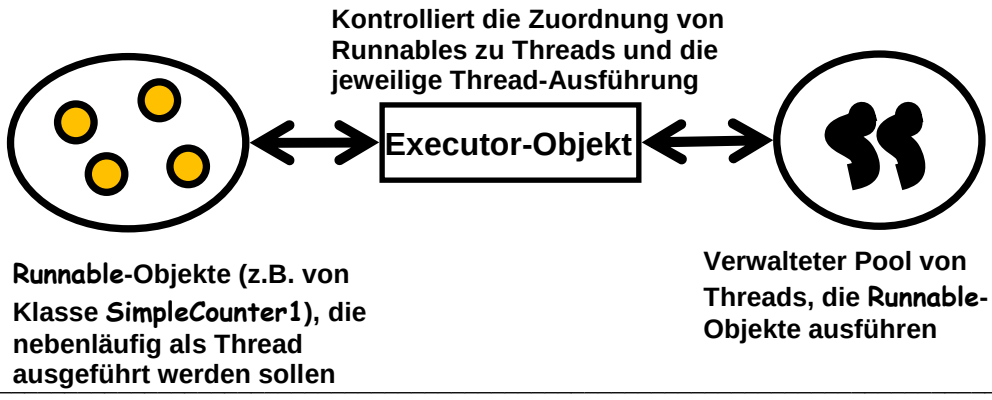
Abhilfe: Klasse java.util.concurrent.Executor (seit Java5)

Variante 3: Executor als “intelligente” Thread-Kontrolle

Die Thread-Erzeugung und Zerstörung ist teuer (kostet Performance).
Das *Executor* Prinzip hilft daher beim Umgang mit mehreren Threads und deren Ausführungen. Der Excecutor entkoppelt von den Details der Thread-Nutzung und des Scheduling und erlaubt die Thread-Wiederverwendung.

Ein Executor–Objekt wird aus der Klasse Executors mit verschiedenen Strategien zur Ausführung von Threads erzeugt.

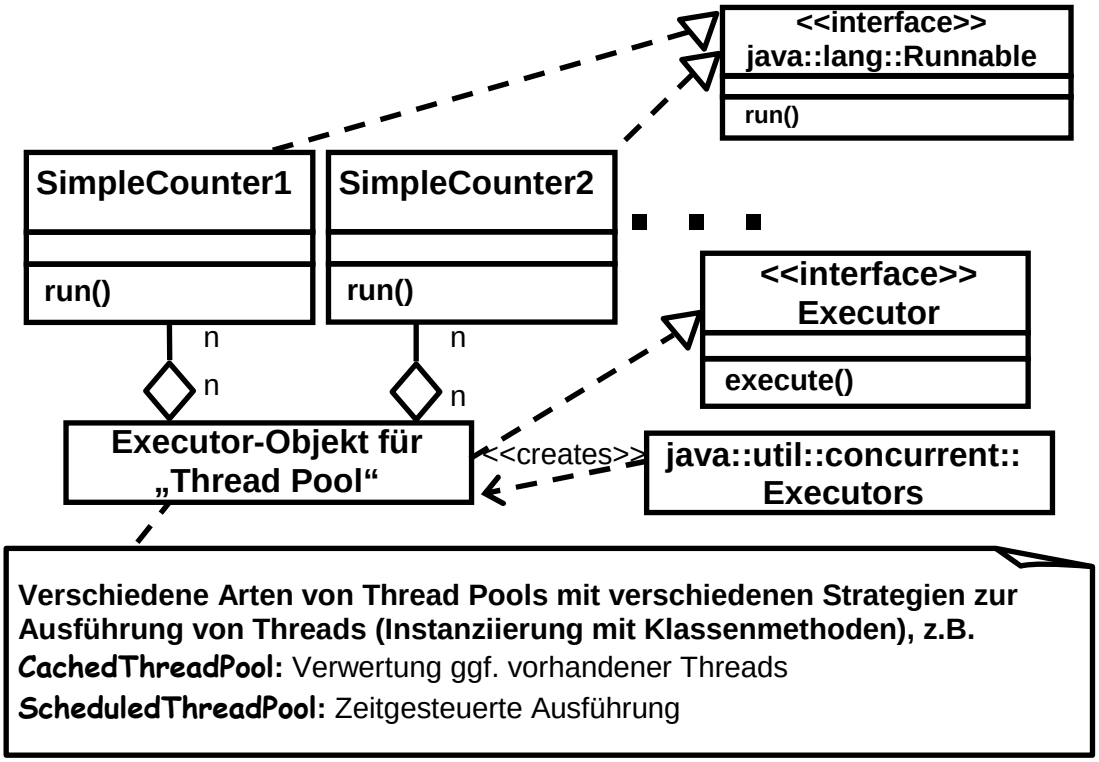
Ein Executor-Objekt verwaltet Runnable-Objekte und Threads und sorgt für die passende Zuordnung und Abarbeitung verschiedener Runnable-Objekte.



Erzeugung:
Wir beauftragen die Klasse Executors (in Java im Paket java.util), einen bestimmten, nach einer bestimmten Strategie arbeitenden Executor zu erzeugen, z.B.

```
Executor executor = Executors.newCachedThreadPool();
```

Das Ergebnis ist ein Objekt, das die Schnittstelle Executor (bzw. dessen Sub-Interface ExecutorService) implementiert. Das Interface Executor schreibt die Methode execute() vor. Wir erzeugen einen Executor also nicht mit einem Konstruktor, sondern mit einer Methode, die uns eine Instanz zurückgibt



Zuordnung von Runnables und Ausführung:
Wir führen unsere Runnable-Objekte mittels Methode execute() unter der Kontrolle des erzeugten Executor-Objekts aus:

```
executor.execute(new MyRunnable1());
executor.execute(new MyRunnable2());
executor.execute(new MyRunnable3());
```

Im Beispiel zum Bild:

```
executor.execute(new SimpleCounter1());
executor.execute(new SimpleCounter2());
```

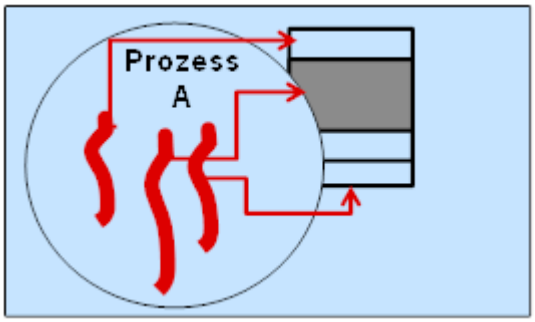
Soll ein Executor keine weiteren Threads mehr annehmen, kann dieser geschlossen werden:

```
executor.shutdown();
```

Nach dem Schließen eines Executor-Objekts werden die bereits vorhandenen Threads nicht abgebrochen, sondern weiter verarbeitet. Neue Anfragen werden aber nicht mehr angenommen.

9.3.2 Beenden eines Threads

Ein Programm bzw. ein Prozess wird erst dann beendet, wenn der letzte Thread beendet ist.



Ein Thread kann beendet werden durch:

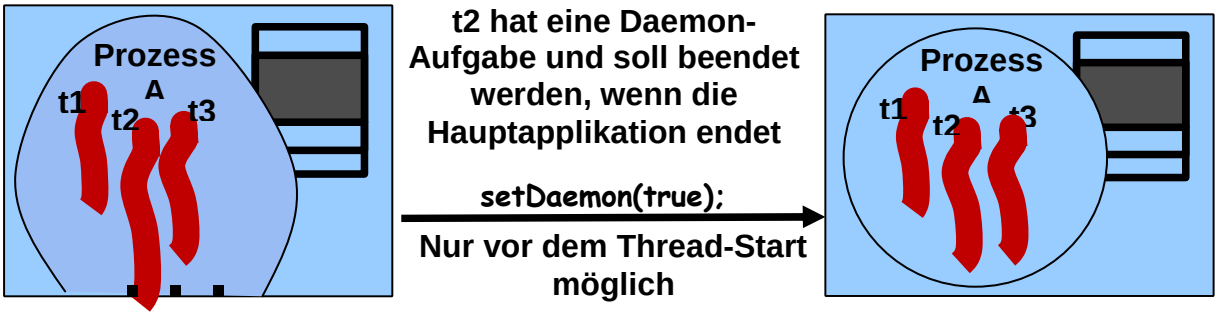
- Ein normales Ende von `run()`,
- eine RuntimeException,
- das Ende der virtuellen Maschine,
- direkt von außen gesteuert: `stop()` (deprecated!)
  Da der Thread zu irgendeinem Zeitpunkt unterbrochen wird, endet er gegebenenfalls in einem unsicheren Zustand.
- indirekt von außen gesteuert als Folge einer Zustandsänderung: `interrupt()` als Signal, das der Thread kontrolliert verarbeitet.

Bemerkung zum Begriff „deprecated“:
Eine Schnittstelle oder ein Merkmal ist in einer Sprache als „deprecated“ markiert, wenn es nicht mehr verwendet werden sollte. In zukünftigen Versionen wird das als deprecated bezeichnete Konzept eventuell nicht mehr unterstützt. Aus Gründen der Rückwärtskompatibilität (d.h. der Funktionsfähigkeit mit älteren Programmen) ist das Konzept nicht jetzt schon entfernt.

9.3.3 Spezielle Threads

(1) Daemons (Dämonen)

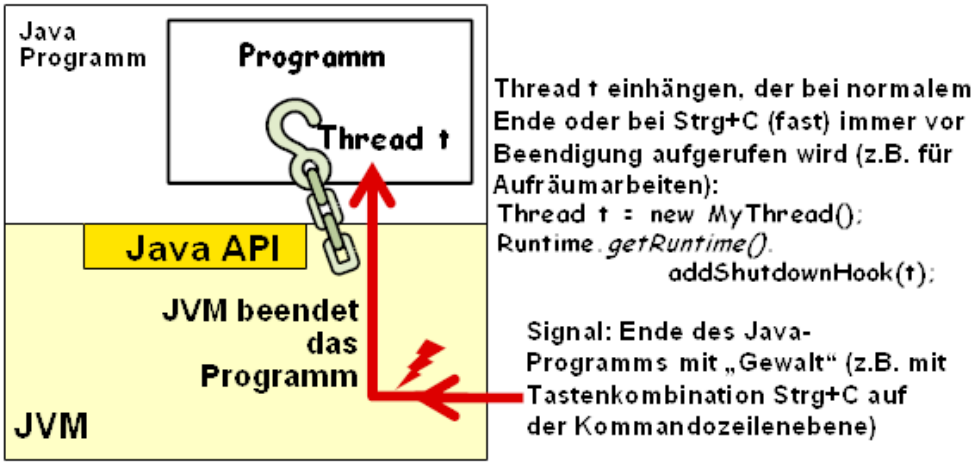
Ein Daemon (Dämon) ist eine Endlosschleife, die im Hintergrund etwas überwacht, auf Aufträge wartet oder sonst eine permanente Aufgabe erfüllt.
Der Einsatz von Threads ist für solche Aufgabenstellungen ist ganz typisch.
Mit dem Ende der Hauptapplikation brauchen wir einen solchen Daemon-Thread nicht mehr.



Enthält ein gestarteter Thread eine Endlosschleife wird dieser grundsätzlich nie beendet.
Der Thread würde also immer weiter laufen, auch wenn die Hauptapplikation beendet ist.
Dies ist nicht immer beabsichtigt, da die Dienstleistung des Threads im Hintergrund nach Beenden der Applikation nicht mehr gefragt ist. Auch ein solcher Endlos-Thread sollte dann beendet werden. Dazu kann ein Thread als Dämon gekennzeichnet werden. In Java geschieht

dies vor dem Starten des Threads mit der Nachricht `setDaemon(true)`. Standardmäßig ist ein Thread kein Dämon.
Dämon bezeichnet eigentlich ein Geisterwesen, einen Schutzgeist, Mischwesen (Chimäre). Die Bezeichnung spielt darauf an, dass der Thread seine Aufgabe (wie ein Geist) im Hintergrund erfüllt.

(2) ShutdownHook



Ein Java-Programm kann normal zu Ende gehen oder mit „Gewalt" beendet werden; dazu wird z.B. die Tastenkombination Strg+C auf der Kommandozeile eingegeben.
Dabei wird ein Signal an die JVM geschickt und das Programm wird beendet.

Will man noch vor der Beendigung des Programms z.B. Aufräumarbeiten erledigen, kann man einen Thread einhängen, der die Aufgabe übernimmt.

```
Thread t = ...;
Runtime.getRuntime().addShutdownHook(t);
```

Der Thread `t` wird dann (fast) immer vor Beendigung eines Programms ausgeführt, insofern es normal beendet wurde, oder das Signal durch die Tastenkombination Strg+C geschickt wurde.

Achtung: Der ShutdownHook-Thread wird bei einem gewaltsamen Ende in Eclipse oder mit dem Windows Taskmanager nicht ausgeführt.

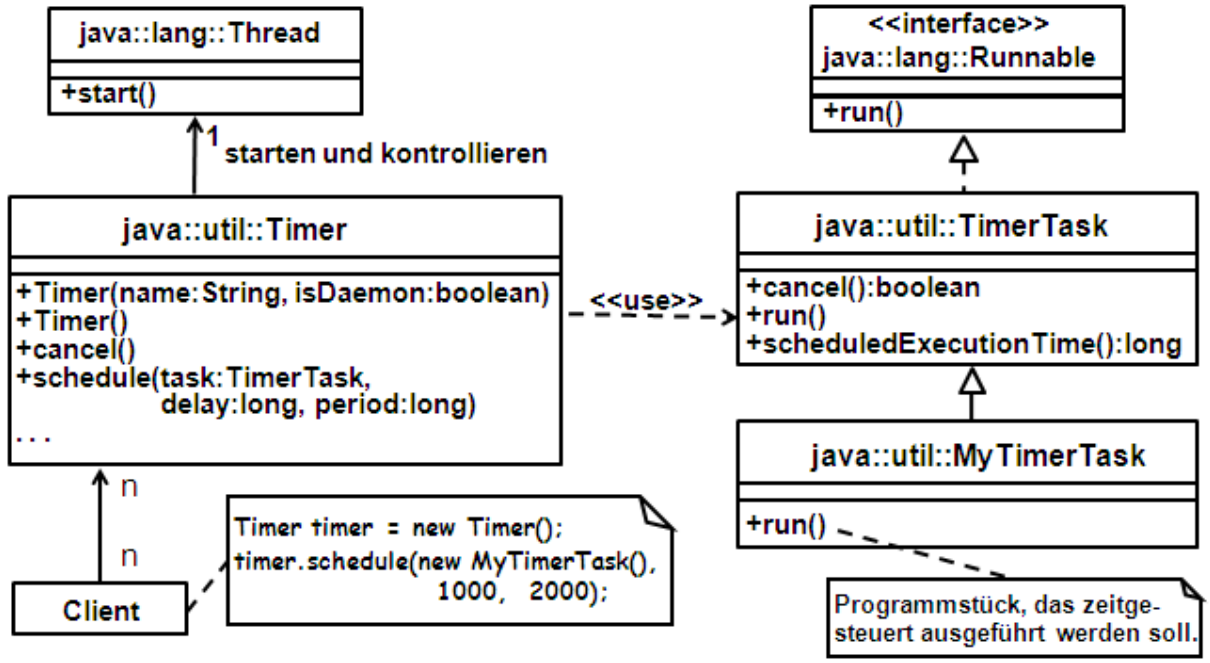
(3) Timer-Thread für eine zeitgesteuerte Ausführung

Ziel: Einmalige oder wiederholte Ausführung einer Aktion nach einer festen Zeitspanne oder zu einem bestimmten Zeitpunkt.
Beispiele: Regelmäßige Freigabe von nicht mehr referenziertem Speicher (Garbage Collector), regelmäßige Ausgabe von Berichten.

Threads sind geeignet, eine solche zeitlich gesteuerte (Warte- und Überwachungs-)Aktivität auszuführen, da sie Ressourcen-sparend im Hintergrund laufen können.

In Java gibt es hierfür verschiedene Pakete sowie weitere geeignete externe Bibliotheken insbesondere die Klassen `java.util.Timer` und `java.util.TimerTask`.

Ein Client erzeugt eine `Timer`-Instanz und `TimerTask`-Instanzen. Der `Timer`-Instanz werden die `TimerTask`-Instanzen zur zeitgesteuerten Ausführung durch den Aufruf der Instanzmethode `schedule(...)` übergeben. Im Beispiel unten wird die Aktion in `run()` der `MyTimerTask`–Instanz nach einer Wartezeit von 1000ms alle 200ms ausgeführt.
Einer `Timer`-Instanz ist ein Thread zugeordnet, der alle Tasks des Timers sequentiell ausführt.



Bemerkungen:

- Im Standardfall ist der Thread, der die Tasks für den Timer ausführt, kein Daemon-Thread. Da ein Programm erst zu Ende ist, wenn der letzte Thread darin beendet ist, bietet die `Timer`-Klasse die Methode `cancel()`. Mit ihr kann der Thread, der die `TimerTask` ausführt, beendet werden.
- Die `Timer`-Klasse ist Thread-sicher (eine Erklärung des Begriffs folgt unten).
- Die `Timer`-Klasse garantiert keine Echtzeit, d.h. sie gibt keine Garantie, dass bestimmte Zeittoleranzen (mindestens oder maximal) eingehalten werden.
- Die `Timer`-Klasse arbeitet zur Zeitsteuerung intern mit `Object.wait(long)`.
- TimerTasks sollten schnell fertig sein, da sonst die Zeitvorgaben eventuell nicht eingehalten werden können.

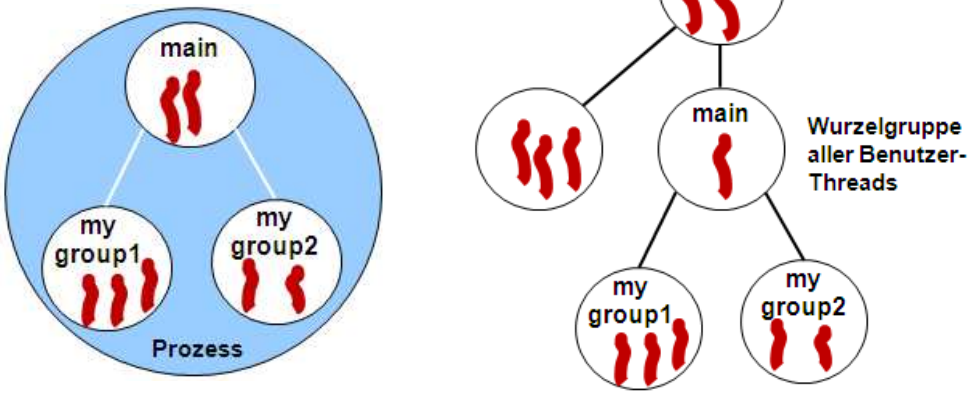
9.3.4 Thread-Gruppen

Threads können zu Gruppen zusammengefasst werden.

Nutzen von Thread-Gruppen:
Thread-Gruppen vereinfachen die Thread-Verwaltung, durch Zusammenfassung zusammengehöriger Threads in Gruppen (Modularisierung).
Threads einer gemeinsamen Gruppe können gemeinsame Eigenschaften bekommen (z.B. Setzen einer maximal möglichen Priorität für alle Threads einer Gruppe, Setzen der Dämon-Eigenschaft aller Threads der Gruppe) und gemeinschaftlich bearbeitet werden (z.B. gleichzeitiges Unterbrechen aller Threads einer Gruppe).

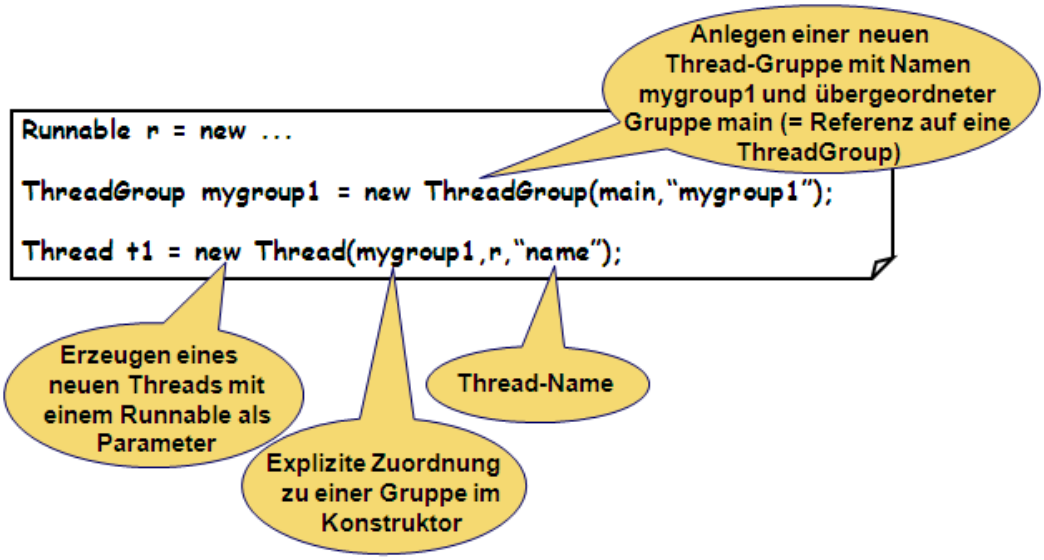
Java verwaltet Gruppen durch die JVM.
Entwickler können mit Hilfe der Klassen java.lang.ThreadGroup und java.lang.Thread mit den Gruppen arbeiten.

Die Anordnung der Gruppen hat eine baumartige Struktur.
Wurzel der Benutzer-Threads: Gruppe main
Wurzel aller Threads: Gruppe system



Zuordnung von Threads zu Gruppen:
Ein erzeugter Thread ist immer einer Gruppe zugeordnet.
Diese Gruppe wird durch ein Objekt der Klasse ThreadGroup repräsentiert.
Eine nachträgliche Änderung der Gruppenzugehörigkeit ist nicht mehr möglich.

- Explizite Zuordnung eines Threads zu einer Gruppe mit folgenden Schritten:

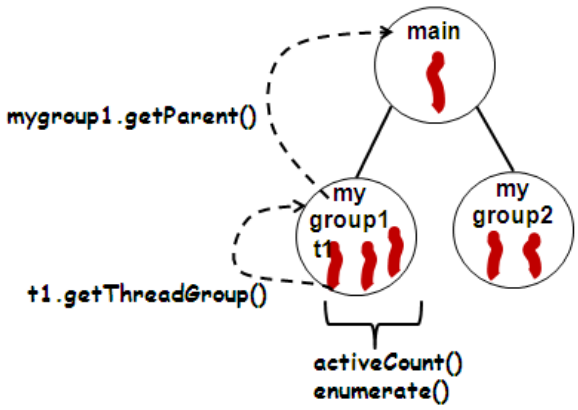


(1) Anlegen einer neuen Gruppe in Form eines neuen ThreadGroup-Objekts.
Falls beim Anlegen nicht explizit eine übergeordnete Gruppe angegeben wird, ist die übergeordnete Gruppe automatisch diejenige, der der aktuelle Thread zugeordnet ist.
(2) Threads bei dessen Erzeugung zu dieser neuen Gruppe hinzufügen.

- Implizit, automatische Zuordnung eines Threads zu einer Gruppe:
Falls bei der Erzeugung nicht explizit eine Gruppe angegeben wird, wird der erzeugte Thread automatisch der Gruppe zugeordnet, der schon der erzeugende Thread angehörte.

Einige Möglichkeiten zum Arbeiten mit der Baumstruktur aus Gruppen:
Der Baum mit den Gruppen(informationen) kann durchlaufen, manipuliert und abgefragt werden.
Sei mygroup1 eine Instanz einer ThreadGroup und t1 eine Instanz von Thread in der Gruppe mygroup1.

- mygroup1.getParent() ⇒ Antwort: Gruppe darüber (im Beispiel: main).
- t1.getThreadGroup() ⇒ Antwort: Gruppe, in der t1 liegt (im Beispiel: mygroup1).
- mygroup1.activeCount() ⇒ Antwort: Anzahl der aktiven (d.h. noch nicht beendeten Threads) in Gruppe mygroup1 inklusive aller Untergruppen.
- mygroup1.enumerate(...) ⇒ Antwort: Kopiert eine Referenz auf jeden aktiven Thread in Gruppe mygroup1 in ein als Parameter angegebenes Feld. Je nach weiterer Parameterangabe werden auch Referenzen auf Threads der Untergruppen kopiert.



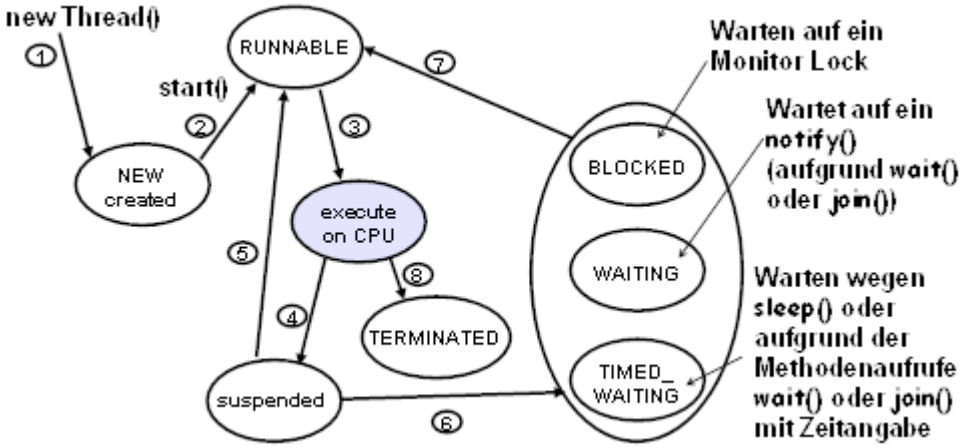
Neben Klasse ThreadGroup ist für viele Aufgaben auch die Klasse Thread verwendbar. Die Verwendung ist für Gruppen teils ähnlich, allerdings auf Klassenebene mit Klassenmethoden. Beispiele:

- Thread.activeCount() ⇒ Antwort: Anzahl der noch nicht beendeten Threads in der Gruppe des gerade aktiven Threads.
- Thread.enumerate(...) ⇒ Antwort: Array mit allen Threads in der gleichen Gruppe sowie Untergruppen dazu, wie der gerade aktive Thread.
- Thread.currentThread() ⇒ Antwort: Thread, der gerade aktiv ist (d.h. der gerade genau diese Anweisung „Thread.currentThread()“ ausführt).

Bemerkung / Achtung:
Manche Funktionalität in Thread sieht nur ähnlich zu der in ThreadGroup aus, ist aber tatsächlich semantisch unterschiedlich!
Beispiel: mygroup1.setMaxPriority(...) setzt nicht die Priorität aller Threads in der Gruppe mygroup1, sondern setzt die theoretisch höchstens einstellbare Priorität für Threads in mygroup1.

9.4 Thread Zustände

Ähnlich zu Prozessen haben auch Threads definierte Zustände.



In Java kann der Thread-Zustand bzw. Status mit `getState()` abgefragt werden.
Java liefert nur die Zustände, die im Bild oben in Grossbuchstaben notiert sind.

Ein Java-Thread zeigt auf Anfrage sechs verschiedene Stadien an:

- NEW: Der Thread wurde erstellt, aber noch nicht gestartet.
- RUNNABLE: Der Thread wurde gestartet.
- BLOCKED: Der Thread läuft momentan nicht, sondern wartet auf einen Monitor.
- WAITING: Der Thread wartet aufgrund eines Methodenaufrufs `wait()` oder `join()`.
- TIMED_WAITING: Der Thread wartet aufgrund der Methode `sleep()` oder aufgrund der Methoden `wait()` und `join()` mit einer Zeitangabe.
- TERMINATED: Der Thread ist beendet. Die Abfrage `isAlive()` liefert die Information, ob ein Thread gestartet, aber noch nicht beendet ist.

Möglichkeiten zur Kontrolle der Zustände von Threads (in Java):

- Thread anlegen, starten und beenden (wie oben gesehen).

- Jeder Thread verfügt über eine Priorität, die aussagt, wie viel Rechenzeit ein Thread relativ zu anderen Threads erhält. Thread-Prioritäten beeinflussen den Scheduler in der Einplanung. Thread-Prioritäten können mit `setPriority()` geändert und `getPriority()` abgefragt werden.
  Bei seiner Initialisierung bekommt jeder Thread die Priorität des erzeugenden Threads.
  Mögliche Werte für die Priorität:
  - eine Zahl zwischen `Thread.MIN_PRIORITY` (1), `Thread.MAX_PRIORITY` (10),
  - Default-Wert: `Thread.NORM_PRIORITY` (5).

  In Java ist die Prioritätenimplementierung nicht spezifiziert und auch nicht zwingend. Man kann Prioritäten also setzen, sich aber nicht auf die Wirkung verlassen.

- Threads für eine bestimmte Anzahl an Millisekunden schlafen schicken: `sleep()`.
  Der Aufruf bezieht sich immer auf den ausführenden Thread. Man kann keinen fremden

Thread, über dessen Referenz man verfügt, schlafen legen. Ein Thread kann so kooperativ die CPU für andere freigeben.
Eine Variante ist die zusätzliche Angabe eines TimeUnit-Objekts als Parameter. Damit kann sich ein Thread für eine bestimmte Zeit Schlafen legen.
Bei Unterbrechung des Schlafs mittels interrupt() (vgl. unten) wird eine InterruptedException geworfen. InterruptedException ist keine RuntimeException und muss daher aufgefangen werden. Eine entsprechende throws-Klausel im Methodenkopf ist aufgrund der Vererbung der Methode run() nicht möglich (Substitutionsprinzip, Kovarianz für Exceptions).

- Signal von außen an den Thread für einen kontrollierten Abbruch: interrupt() setzt ein internes Flag.
  Achtung: Methoden wie sleep(), join() und wait() reagieren mit InterruptedException auf das Flag und löschen (!) gleichzeitig das Flag.
  Bemerkungen:
  - Mit der Methode isInterrupted() kann der Unterbrechungsstatus abgefragt werden. Ist als Thread-Status der Abbruch eingetragen, kann nun innerhalb des Threads der Abbruch vorgenommen werden. In der Regel bedeutet dies, dass eine Schleife beendet wird, die in der run()-Methode implementiert ist. Auf diese Weise kann ein Thread selbst bestimmen, wann er abbricht und kann ggf. noch bestimmte Aufgaben und Aufräumarbeiten erledigen, bevor er anhält.
  - Beenden eines Threads mit interrupt() ist gegenüber stop() zu bevorzugen, da der Thread so selbst entscheidet, wie er sich beendet und auch noch abschliessende Arbeiten durchführen kann.

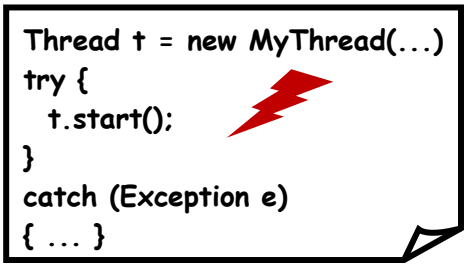
- Thread (unter Berücksichtigung seiner Priorität) in die Thread-Warteschlange des Systems zurücklegen: yield()
  Der Thread teilt der Thread-Verwaltung mit: „Ich will jetzt nicht mehr; ich mache weiter, wenn ich das nächste Mal an der Reihe bin.“

- suspend() und resume() (sind genauso wie stop() *deprecated*).

- Zustandswechsel durch Synchronisation (wie wir später sehen werden).

9.5 Threads und Fehler

Wegen der Nebenläufigkeit ergibt sich ein Problem für Rückgabewerte und Ausnahmen (Exceptions) innerhalb eines Threads, die aber nicht innerhalb des Threads behandelt werden: Wohin sollen sie geschickt werden? Wer soll sie auffangen?
Es gibt keinen wartenden Aufrufer; dieser ist nebenläufig selbst bereits längst an einer anderen Programmstelle aktiv.



try/catch um start() funktioniert nicht! Wegen Nebenläufigkeit läuft der neue Thread und (im Gegensatz zu den kennengelernten try/catch-Blöcken) auch der Programmteil mit dem try/catch nach Aufruf von start() weiter.

Abhilfe in Java: UncaughtExceptionHandler
Dieser wird an einen Thread (individuell) oder an alle Threads (pauschal) gehängt.

Der UncaughtExceptionHandler wird immer benachrichtigt, wenn ein Thread wegen einer nicht behandelten Exception beendet wird. Die JVM ruft dann die Methode uncaughtException(). Auf diese Weise kann innerhalb dieser Methode noch auf die Ausnahme reagiert werden. Die Ausnahme wird wie üblich von der JVM als Parameter vom Typ Throwable übergeben.

Beispiel für die Nutzung von UncaughtExceptionHandler:

```
Thread t1 = new MyThread(...);
Thread t2 = new MyThread(...);
...
t1.setUncaughtExceptionHandler(new MyExceptionHandler(„Msg“));
Thread.setDefaultUncaughtExceptionHandler(new MyExc...
...
```

Voraussetzungen zum Beispiel:

- Implementierung von MyThread bzw. über die Runnable-Alternative.
- Implementierung von MyExceptionHandler (dieser muss das Interface UncaughtExceptionHandler implementieren): Die Klasse enthält in der Methode uncaughtException() die Aktionen, die bei einer Exception im Thread ausgeführt werden sollen.

Wie im Beispiel zu sehen ist, kann eine Default-Exception-Behandlung für alle Threads oder eine individuelle Exception-Behandlung festgelegt werden. Die individuelle Behandlung „überschreibt“ dabei die Default-Behandlung.

9.6 Synchronisation

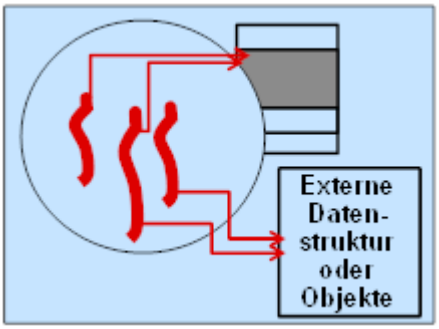
Durch Nebenläufigkeit erreichen wir Vorteile, handeln uns aber auch Schwierigkeiten ein. Die zugrunde liegenden Problematiken sowie die Lösungsprinzipien sind nicht begrenzt auf Threads, sondern gelten gleichermaßen für alle nebenläufigen Aktivitäten. Auch sind die Probleme und Lösungsprinzipien nicht an eine bestimmte Sprache gebunden.

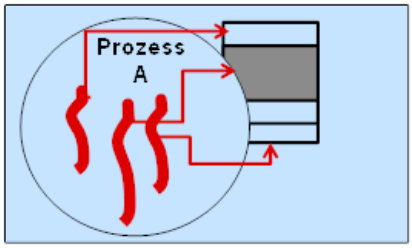
Bemerkung:
Viele Problemstellungen und Lösungen auf diesem Gebiet stammen ursprünglich aus dem Bereich der Betriebssystementwicklung, da Betriebssysteme mit einer Vielzahl nebenläufiger Prozesse umgehen müssen.

Im Weiteren betrachten wir Nebenläufigkeit aus der Sicht nebenläufiger Threads.

Gemeinsame Ressourcen:
- Nebenläufige Threads können Ressourcen gemeinsam (aus)nutzen.
- Verschiedene Exemplare einer Thread-Klasse können durch gemeinsame Ressourcen kommunizieren.

Beispiele für typische gemeinsame Ressourcen im Programm:
- Daten ablegen bzw. entnehmen z.B. in bzw. aus Klassenvariablen.
- Zugriff auf gemeinsam bekannte Datenstrukturen oder Objekte durch gemeinsame Nutzung von Referenzen.





Bei nebenläufigen Aktionen bekommen wir Probleme durch konkurrierende Zugriffe auf gemeinsam genutzte Ressourcen.
- Kritisch: Gemeinsamer Zugriff mit mindestens einem schreibberechtigten Thread.
- Unkritisch sind:
  1. Thread-lokale Variablen,
  2. nur Lesezugriffe oder immutable Objekte.

Synchronisation: Abhilfe bei konkurrierenden Zugriffen und zur Interaktion
Das Problem konkurrierender Zugriffe kann durch Sicherstellung von gegenseitigem Ausschluss durch folgende Maßnahmen erreicht werden:
a) Schutz von Ressourcen (ressourcenorientiert),
b) Synchronisation des Zugriffs (zugriffsorientiert).

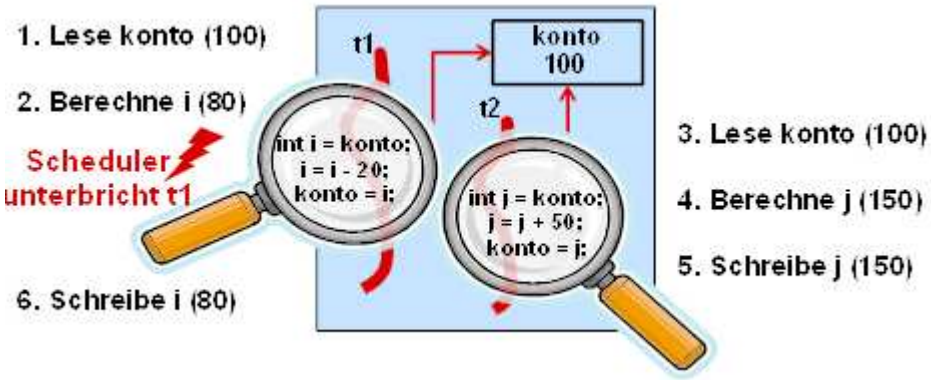
Die Umsetzung der Abhilfen kann wieder neuen Problemen führen, den sogenannten Deadlocks (Verklemmungen). Dazu sehen wir später mehr.

9.6.1 Zugriffsprobleme

Problem: Lost Updates

Ein *Lost Update* (verlorene Aktualisierung) ist ein Zugriffsproblem durch die nichtdeterministische Nebenläufigkeit (in Form echter Parallelität oder Quasi-Parallelität durch den Scheduler).

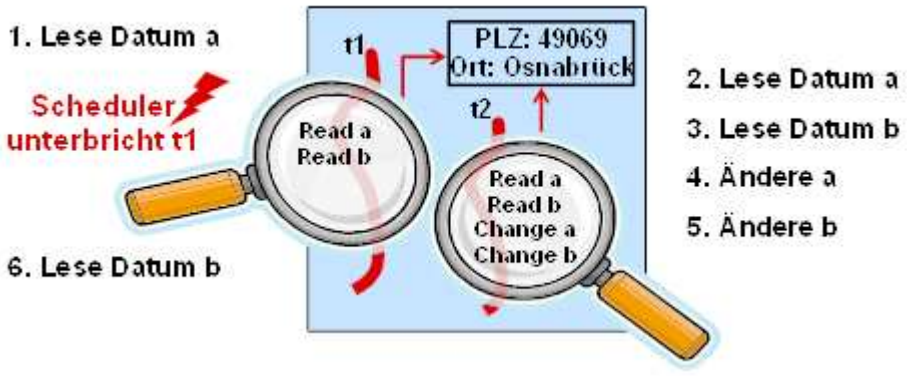
Beispiel: Kontozugriffe
Die Nummern geben die zeitliche Schrittreihenfolge an.



- Lost Update: Update von t2 ist verloren.
- Race Condition bzw. Race Hazard (Wettlaufsituation): Das Ergebnis hängt vom Ausführungszeitpunkt der Threads ab.

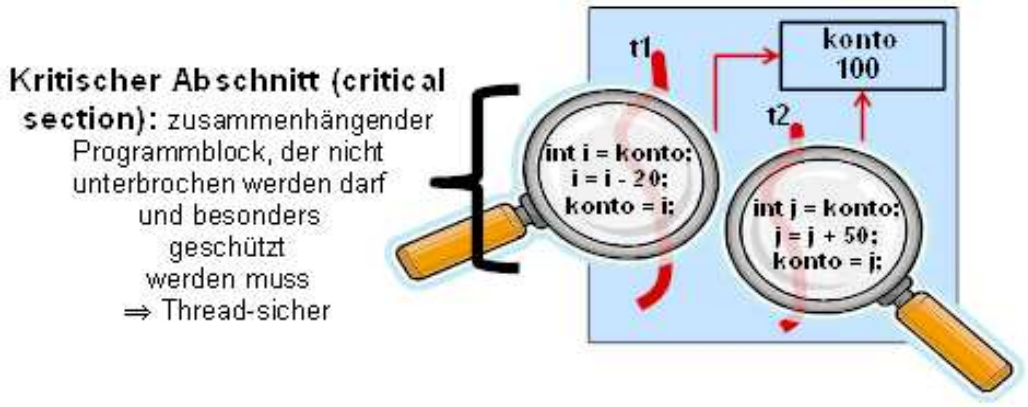
Problem: Dirty Read

Ein *Dirty Read* („Schmutziges“ Lesen) ist eine Inkonsistenz zweier zusammengehöriger Datenelemente.
Beispiel: Auslesen zweier zusammengehöriger Daten



- Dirty Read im Beispiel: t1 liest einen inkonsistenten Datensatz (a passt nicht mit b zusammen, z.B. a = Postleitzahl, b = Wohnort).

9.6.2 Kritischer Abschnitt (engl. Critical Section)



In unserem Beispiel von oben war die kritische Ressource im gemeinsamen Zugriff das Datum *konto* (bzw. die beiden Daten *PLZ* und *Ort*). Der kritische Abschnitt war das jeweilige Programmstück, in dem auf die kritische Ressource zugegriffen wurde und in dem es zu Zugriffskonflikten mit anderen Threads kommen konnte.

Abhilfe:
(1) Bestimmung der kritischen (gemeinsam benutzten) Ressource und Sperrung dieser Ressource für den exklusiven Zugriff (ressourcenorientiert).
(2) Sicherstellung der unterbrechungsfreien Abarbeitung der kritischen Programmzeilen, in denen auf die kritische Ressource zugegriffen wird (zugriffsorientiert).

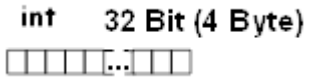
Gegenseitiger Ausschluss (mutual exclusion, auch mutex) / atomar:
Immer nur ein Thread befindet sich (unterbrechungsfrei) im kritischen Programmteil.

Bemerkung: Begriff „Thread-sicher“
Ein Konzept wird *Thread-sicher* genannt, wenn es so gestaltet ist, dass nebenläufige Threads zu keiner Fehlersituation führen können.

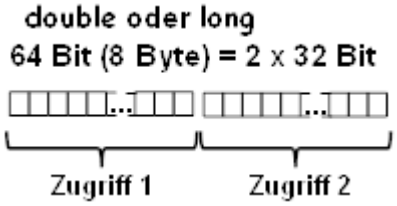
9.6.3 Abhilfe auf Sprachebene

Das Schlüsselwort volatile dient als Abhilfe für das Dirty Read/Lost Update-Problem auf Sprachebene in Java.

- Die JVM arbeitet bei Ganzzahl-Datentypen kleiner gleich int intern nur mit int.
  Zugriff auf die Daten vom Typ int: Mit einem Bytecode-Befehl.



- Zugriff auf 64-Bit-Datentypen: Mit zwei Bytecode-Befehlen!
  Die Unteilbarkeit des Zugriffs ist damit nicht mehr gesichert!
  Greifen zwei Threads auf die gleiche 64-Bit-Variable zu, so könnte möglicherweise der eine Thread eine Hälfte schreiben und der andere Thread die andere.



- Abhilfe in Java: Eine Deklaration mit dem Schlüsselwort volatile für Objekt- und Klassenvariablen sorgt (mit zusätzlichem, internen Aufwand) für einen atomaren Zugriff. Achtung: Das Schlüsselwort funktioniert nicht für lokale Variablen. Allerdings sind lokale Variablen auch nicht für andere Threads zugreifbar.

Das Schlüsselwort volatile ist sprachspezifisch. Andere Programmiersprachen unterstützen einen atomaren Datenzugriff für bestimmte Datentypen gegebenenfalls durch andere Sprachmittel und Schlüsselworte.

Das Sprachmittel hilft auf Sprachebene beim Zugriff auf Datentypen. Es hilft allerdings nicht bei sonstigen Operationen.

Beispiel i++:

```
public class IPlusPlus {
   static int i = 0;
   public static void main(String args[]) {
      i++;
   }
}
```

```
public static void main(java.lang.String[]);
   Code:
   0:   getstatic
   3:   iconst_1
   4:   iadd
   5:   putstatic
   8:   return
```

Ist i++ eine atomare Anweisung?

Die Antwort ist im resultierenden, zugehörigen Byte-Code zu finden.

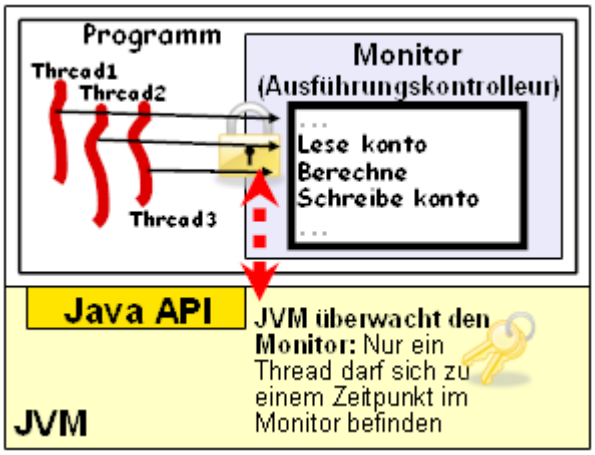
Auch volatile hilft hier nicht, da wir eine ganz andere Problemstellung haben.
Die Operation (und nicht ein Datenzugriff!) besteht im Byte-Code aus mehreren Anweisungen, die prinzipiell in der Mitte unterbrochen werden können.

Wir müssen genau in die Sprache schauen, um zu erkennen, welche Anweisungen tatsächlich atomar sind. Nur gesichert atomare Anweisungen können nicht unterbrochen werden, d.h. sie werden entweder ganz oder gar nicht abgearbeitet.

9.6.4 Schützen kritischer Abschnitte mit Monitoren

[nach A.R. Hoare ´74, P. Brinch Hansen ´75]

- Die Laufzeitumgebung (in Java: JVM) kann den Monitor als belegt kennzeichnen, wenn ein Thread in den vom Monitor überwachten kritischen Abschnitt eintritt.
- Ein zweiter Thread muss beim Zugriffsversuch warten.
- Verlässt der erste Thread den kritischen Abschnitt, wird der Monitor wieder frei und ein anderer Thread kann eintreten.



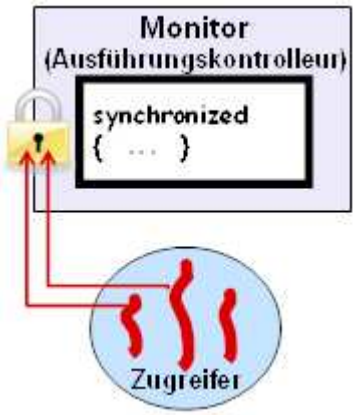
Synchronisation gemeinsamer Zugriffe durch Schützen kritischer Abschnitte mit Monitoren in Java:
In Java gibt es zwei zentrale Sprachkonstrukte zur Umsetzung von Monitoren:

- Java Sprachkonstrukt `synchronized`
- Java Sprachkonstrukt `Lock`

Schützen kritischer Abschnitte in Java: `synchronized`

Zutaten:

- Schlüsselwort `synchronized` zur Definition des Blocks, der geschützt abgearbeitet werden soll.
- Wahl eines geeigneten Monitorobjekts:
  Das Monitorobjekt ist ein gemeinsames Objekt, das mehreren Zugreifern synchronisierten Zugriff ermöglicht. Es wird als Zugriffskontrolleur zum Schützen eines Blocks benötigt. Es kontrolliert, ob ein kritischer Block betreten werden kann bzw. darf.



Typisches Monitorobjekt: Das Objekt, das die kritische Ressource enthält, die zugegriffen werden soll.

Soll die Laufzeitumgebung nur einen Thread pro Zeit in einen Block lassen, wird ein Monitor benötigt. Ein Monitor ist in Java ein Objekt, welches den einzelnen Threads bekannt sein sollte. Tritt ein Thread in einen kritischen Abschnitt ein, kann die JVM den Monitor als belegt kennzeichnen. Kommt ein zweiter Thread zu einem solchen abgeschlossenen kritischen Bereich, muss er warten und wird erst hineingelassen, wenn die Markierung gelöscht ist. Erst wenn der erste Thread den kritischen Bereich beendet hat, gibt die JVM den Monitor wieder frei, und ein anderer Thread kann den kritischen Bereich betreten.
Die Überwachung übernimmt die JVM.

synchronized-Varianten in Java

(1) Synchronisation (d.h. ggf. Sperren) einer ganzen Methode: Für das Objekt (!) kann zu einem Zeitpunkt nur ein Thread die Methode ausführen (alle übrigen Aufrufer müssen warten). Das Schlüsselwort synchronized leitet einen blockweise geschützten Bereich ein. Für den Schutz ist ein Monitor-Objekt notwendig. Wird keines explizit angegeben, ist dies implizit immer das jeweilige Objekt selbst (d.h. das Monitor-Objekt ist dann this).

Übliche Darstellung
(Monitor-Objekt ist implizit this)

```
public synchronized void doSth()
{ // Methodenrumpf }
```

≙

```
public void doSth()  {
  synchronized (this) {
  // Methodenrumpf
  }
}
```

Das Sperren einer ganzen Methode ist oft zu grobgranular.

(2a) Synchronisation (d.h. ggf. Sperren) eines (feingranularen) kritischen Blocks: Für das Objekt (!) kann zu einem Zeitpunkt nur ein Thread den synchronized-Block ausführen (alle übrigen Aufrufer müssen warten). Das Monitor-Objekt ist automatisch und implizit das Objekt, in dem der synchronized-Block enthalten ist, sofern nichts anderes angegeben ist.

```
public void doSth() {
   // nicht kritische Anweisungen
   synchronized (this) {
      // kritische Anweisungen
   }
   // nicht kritische Anweisungen
}
```

Monitor-Objekt ist immer implizit this, wo nichts angegeben ist.

(2b) Synchronisation (d.h. ggf. Sperren) eines (feingranularen) kritischen Blocks mit expliziter Angabe eines Monitor-Objekts:

```
public void doSth() {
   // nicht kritische Anweisungen
   synchronized (monitorObject) {
      // kritische Anweisungen
   }
   // nicht kritische Anweisungen
}
```

Am besten ist das Objekt als Monitor zu verwenden, welches synchronisiert manipuliert wird.

Die Programmiererin gibt im synchronized-Block selbst ein geeignetes Monitor-Objekt an.

(3) Synchronisation auf Klassenebene:

Auf Klassenebene gibt es kein this-Objekt
⇒ Anlegen eines "künstlichen" Object-Objekts zur Verwendung als Monitor-Objekt.

```
private static final Object o = new Object();

static void someStaticMethod() {
  synchronized(o) {
     // kritischer Abschnitt
  }
}
```

Bemerkungen zur Wahl des Monitor-Objekts:
Aufgrund der Vererbungsbeziehung zu `Object` könnte grundsätzlich jedes Objekt in Java als Monitor eingesetzt werden. Die richtige und gute Wahl des Monitor-Objekts ist aber nicht immer einfach.

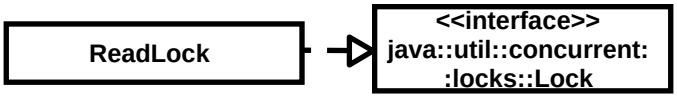
Nur ein Monitor-Objekt pro Synchronisationsaufgabe:
- Man muss außerdem darauf achten, dass der Monitor für alle zu synchronisierenden Threads das gleiche (identische) Objekt ist.
- Beliebter Fehler ist die Synchronisation auf `this` innerhalb der `run()`-Methode eines Threads. Dies kann nicht funktionieren, da jeder Thread auf sich selbst (d.h. sich als Objekt) synchronisieren würde. Statt einem übergreifenden Zugriffskontrolleur hätten wir dadurch viele, unabhängige Kontrolleure als Monitor-Objekte.
  Eine Synchronisation auf `this` kann dann Sinn machen, wenn Variablen im `this`-Objekt die kritische Ressource sind.

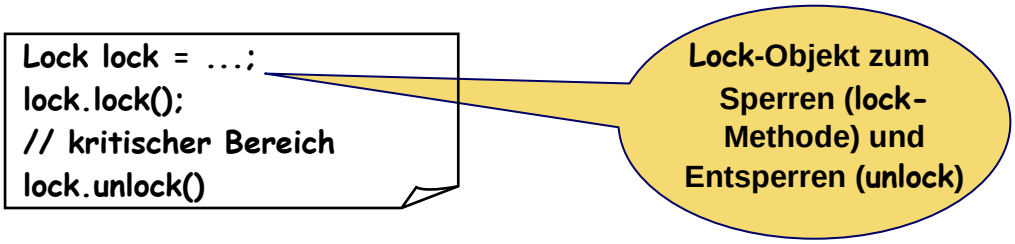
Kritische Ressource selbst als Monitor oder künstliches Monitor-Objekt falls notwendig:
- Am besten ist es, das Objekt als Monitor zu verwenden, welches konkurrierend manipuliert wird. Dazu muss man zunächst genau analysieren, welche Objekte innerhalb eines kritischen Blockes manipuliert werden.
- Werden mehrere Objekte manipuliert, kann man z.B. ein Klassen-Objekt (welches immer nur einmal existiert) als Monitor verwenden. Ein Klassenobjekt ist ein Objekt, das in der Laufzeitumgebung intern als Repräsentation einer Klasse verwendet wird.
  Nachteil dieser Lösung ist, dass auch andere Blöcke ggf. dieses Objekt als Monitor verwenden. Dadurch können Bereiche gesperrt sein, die ohne Probleme bearbeitet werden könnten. Beispiele für Klassenobjekte: Object.class, Person.class
- Eine andere Alternative sind eigens eingeführte Klassenvariablen, die nur als Monitor verwendet werden. Ein solches Monitor-Objekt sollte `final` sein. Dadurch hat der Programmierer ein bestimmtes, selbst erzeugtes Objekt, welches ausschließlich für seine Monitor-Zwecke vorhanden ist und auf das sich die Threads synchronisieren können.

`synchronized`-Blöcke sind reentrant:
- Betritt ein Thread einen `synchronized`-Block, bekommt er den gesetzten Monitor (sofern dieser nicht schon belegt ist).
- Wenn der Thread eine andere Methode aufruft, in der ein Block vorhanden ist, der am gleichen Monitorobjekt synchronisiert ist, kann der Thread sofort eintreten und muss nicht warten.
- Diese Eigenschaft heißt *reentrant* bzw. *Reentranz*.
- Ohne diese Möglichkeit würde z.B. Rekursion nicht funktionieren!

Schützen kritischer Abschnitte in Java mit Locks

- Locks sind ein alternativer Schutz eines kritischen Bereichs vor dem gemeinsamen Zugriff mehrerer Threads (vorhanden seit Java5).
- Locks sind Klassen, die das Interface java.util.concurrent.locks.Lock implementieren.
- Klassen, die das Interface implementieren:
  - ReentrantLock
  - ReadLock
  - WriteLock





Der Programmierer muss sicherstellen, dass die Sperrung aufgehoben wird: Ohne Aufruf von unlock() bleibt der Bereich gesperrt.
Das macht beispielsweise Probleme, wenn Ausnahmen auftreten:

Beispiel: Exception vor unlock()
⇒ unlock() wird nicht mehr ausgeführt
⇒ Sperrung bleibt
Abhilfe: try-finally-Konstruktion

```
Lock lock = ...;
lock.lock();
   // kritischer Bereich
   int i = 10/0;
lock.unlock()
```

Implementierungs- und Entwurfsentscheidungen

- synchronized versus Locks:

synchronized	Lock
Leichter zu handhaben und damit weniger fehleranfällig	Flexibler, weil nicht auf Blöcke begrenzt

⇒ synchronized statt Lock, wo möglich

- Synchronisieren versus nebenläufige Abarbeitung:

Synchronisation	Nebenläufigkeit
- Unnötige Wartezeiten - Aufwendige Synchronisationsverwaltung in der JVM (kostet Ressourcen, v.a. Laufzeit) - Deadlock-Gefahr	Gefahr unerwünschter Nebeneffekte wie z.B. Lost Updates (wegen Nebeneffekten auch schwer zu debuggen)

⇒ Möglichst wenig Sperren und Nebenläufigkeit ausnutzen!

9.6.5 Probleme der Synchronisation: Deadlocks

- Deadlock (Verklemmung): Verklemmung, weil zwei Einheiten (z.B. Threads) gegenseitig auf die Freigabe jeweils einer Ressource warten.
- Deadlocks können durch gegenseitigen Ausschluss (Sperren) auftreten.
- In Java:
  - Die JVM erkennt und meldet die meisten Deadlocks, bricht sie aber nicht selbst ab.
  - *jconsole* ist eine grafische Anwendung zur Anzeige von Informationen (wie z.B. Deadlocks) zur Laufzeit.

Aus: [JavaInsel09]

jconsole kann sich an das Java-Programm „anhängen“, wenn dieses mit einer entsprechenden Property gestartet wird:

```
java –Dcom.sun.management.jmxremote <Programmname>
```

Die System-Property erlaubt die Beobachtung des Programms (z.B. durch *jconsole*) von außen. Nach dem Programmstart mit der Property kann *jconsole* gestartet werden.

9.6.6 Synchronisation durch Kommunikation

- Bisher: Kontrolle des Zugriffs auf eine gemeinsame Ressource.
- Jetzt: Kommunikation zwischen Threads (direkte Abhängigkeiten zwischen Threads) und Synchronisation durch direkte Kommunikation,
  z.B. auch zur Vermeidung von Deadlocks oder
  z.B. zur Realisierung einer Producer-Consumer Synchronisation.

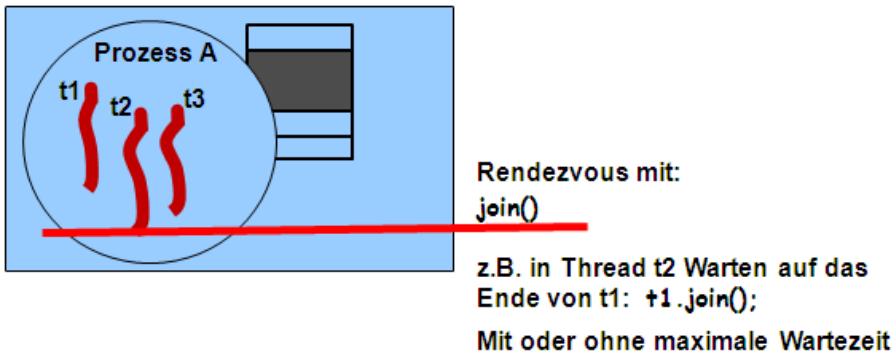
Überblick über die Realisierungsalternativen einer Synchronisation durch Kommunikation in Java:

- Vorherige Vereinbarungen auf Entwicklerebene: Ein Deadlock kann vermieden werden, wenn die Vereinbarung gilt, dass alle Threads immer in der gleichen Reihenfolge sperren. Beispiel: Soll Papier und Stift verwendet werden, so wird ein Deadlock verhindert, wenn alle Threads zuerst versuchen, das Papier zu sperren und erst nach Erfolg auch den Stift sperren.
- Flags und Daten auf einem gemeinsamen Speicher (Rückführung auf gemeinsamen Zugriff auf die Ressource „Speicher“): An einer gemeinsam zugreifbaren Speicherstelle werden Informationen abgelegt, die die Threads zur Synchronisation nutzen.
- `join()`: Zusammenführen von Ergebnissen mehrerer Threads.
- `wait()` und `notify()` bzw. `notifyAll()` (nur in `synchronized`-Blöcken).
- `await()` und `signal()` (nur in Verbindung mit Locks).
- Busy Waiting (als zwar mögliche aber äußerst schlechte Synchronisationsform).
- Semaphoren (binäre Semaphoren in Java bereits durch `wait`/`notify` oder `await`/`signal`, allgemeine Semaphoren in Java mit Hilfe von `java.util.concurrent.Semaphore`).

Die Realisierungsalternativen haben eine teils Java-spezifische Umsetzung. Die dahinter liegenden Prinzipien sind jedoch auch außerhalb von Java weit verbreitet.
Sie sind beispielsweise zentrale Konzepte in der Betriebssystementwicklung.

(1) Zusammenführung bzw. Einsammeln der Ergebnisse mehrerer Threads: join()

Werden Aufgaben auf mehrere Threads verteilt, kommt ein Zeitpunkt, an dem die Ergebnisse eingesammelt werden sollen. Die Resultate können erst dann zusammengebracht werden, wenn alle Threads mit ihrer Ausführung fertig sind. Mit der Instanzmethode join() kann in Java auf einen anderen Thread gewartet werden.



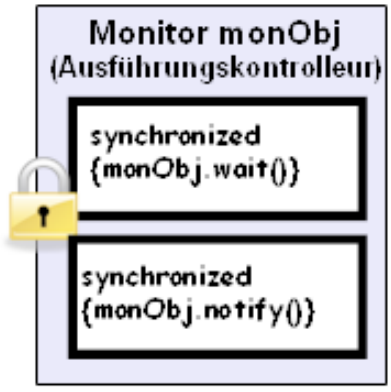
Erreicht eine interrupt()-Nachricht einen Thread, der gerade wegen eines join() wartet, wird eine InterruptedException ausgelöst.

(2) wait() und notify() in synchronized-Blöcken: Aufruf am Monitor-Objekt

Zwei Threads sollen sich am Monitor-Objekt monObj synchronisieren.
Thread t1 soll auf Daten von Thread t2 warten.
Wenn der zweite Thread t2 den Monitor des Objekts monObj bekommt, kann er den wartenden Thread t1 aufwecken (notify()).

```
synchronized(monObj) {      // in Thread t1
  try {
      // Aktivität
      monObj.wait();
      // Fortsetzung nach notify Eingang
  } catch ( InterruptedException e ) { ... }
}
```

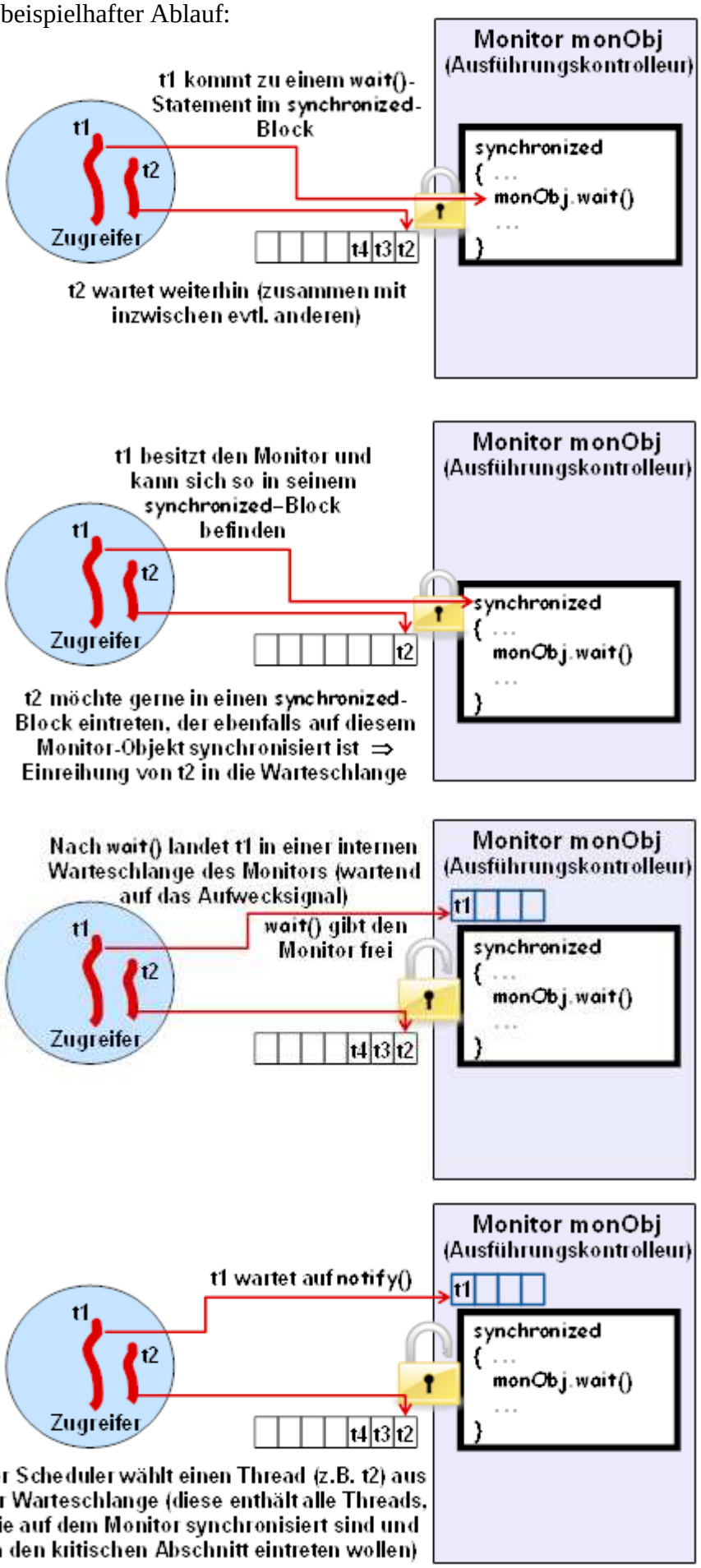
```
synchronized(monObj) {      // in Thread t2
  // Aktivität
  monObj.notify(); // Signal
}
```

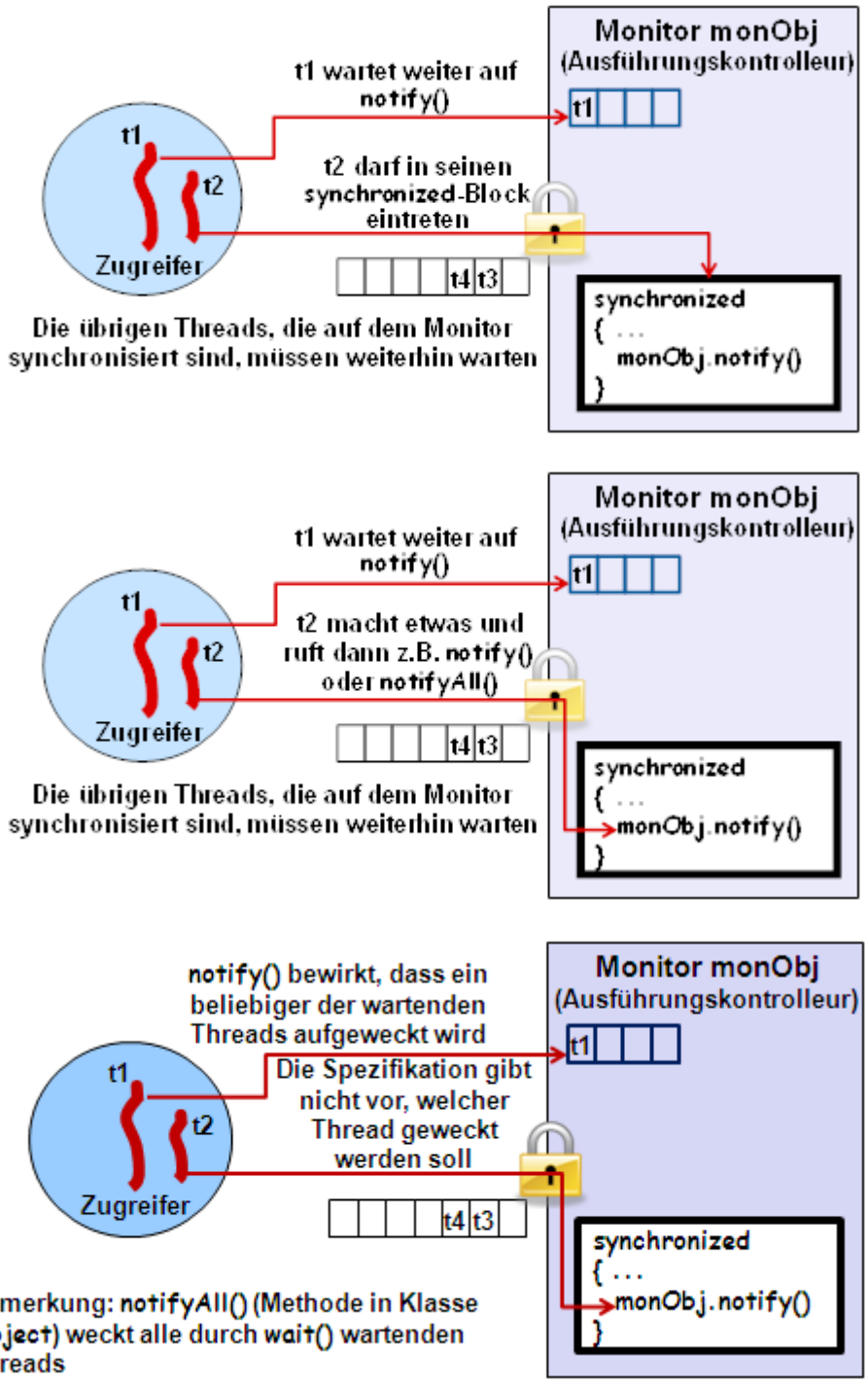


Der zweite Thread t2 kann nur den Monitor bekommen, wenn dieser nicht bereits belegt ist. Damit das überhaupt möglich ist, gibt der Aufruf von wait() in t1 gibt den Monitor wieder frei.

Die Methoden wait() und notify() sind nur mit dem entsprechenden Monitor gültig. Diesen Monitor besitzt das Programmstück, wenn es sich in einem über diesen Monitor synchronisierten Block aufhält.

Schrittweiser, beispielhafter Ablauf:



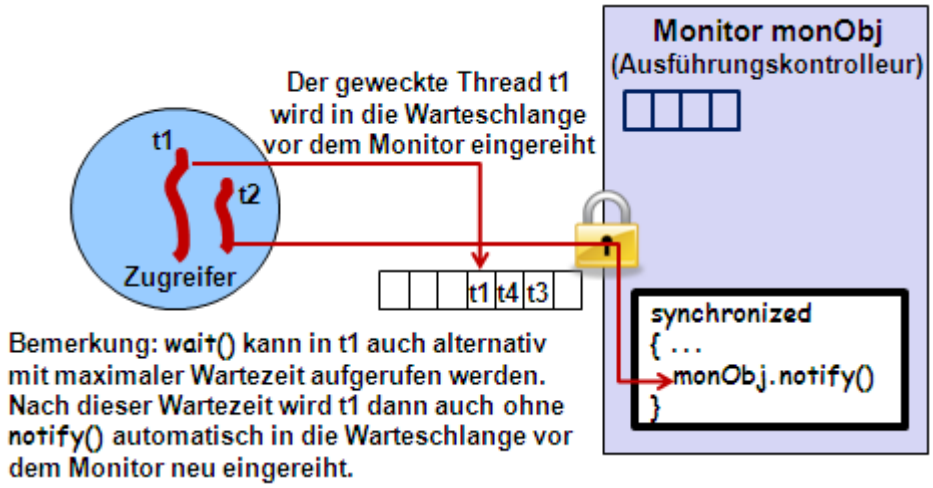


notify():

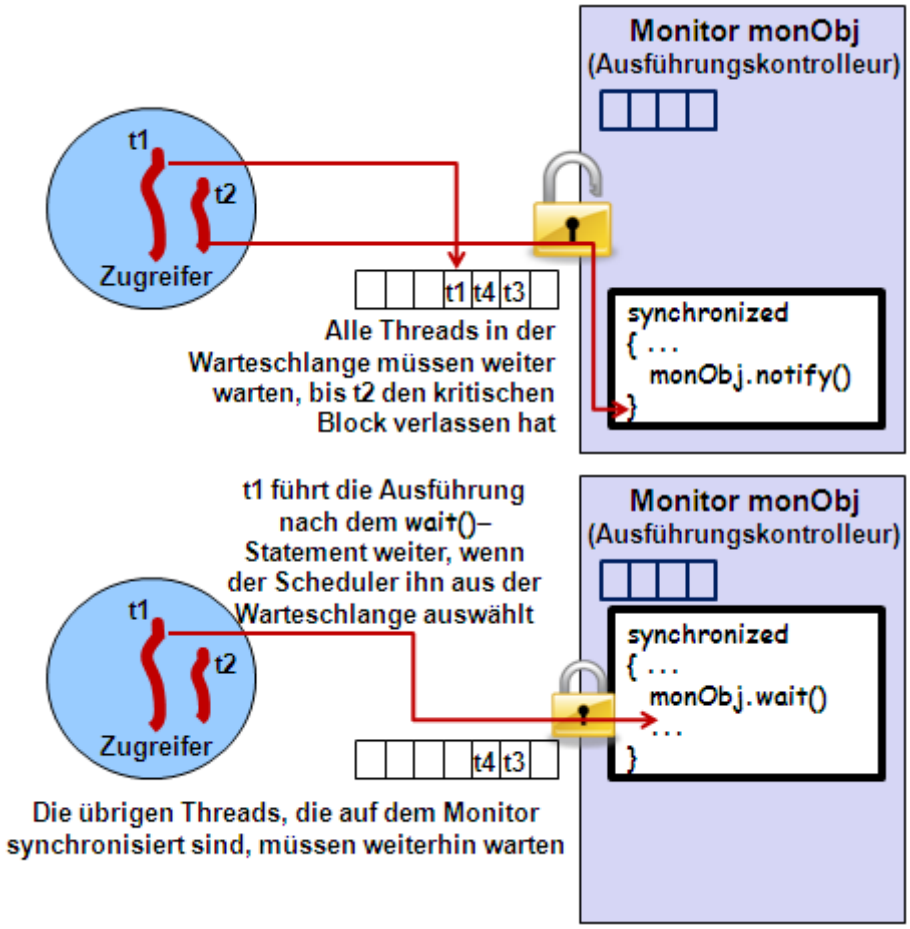
Einer der Threads in der internen Warteschlange wird (nicht-deterministisch) geweckt und in die andere Warteschlange zum Eintritt in den Monitor eingereiht. Dabei und anschließend wird er gegenüber anderen wartenden Threads nicht bevorzugt oder benachteiligt. Der frisch aufgeweckte Thread (oder ein anderer in der Warteschlange vor dem Monitor!) kann als nächstes in den Monitor erst eintreten, wenn der Thread, der `notify()` gesendet hat, den Monitor schließlich verlässt und der Scheduler ihn auswählt. `notify()` ist häufig, muss aber nicht zwingend die letzte Anweisung im `synchronized`-Block sein.

In unserem Beispiel landet t1, nachdem er im Rahmen eines `notify()` „erwählt“ wurde, also in der Warteschlange, in der bereits t4 und t3 warten. Erst wenn t2 den Monitor freigibt, darf

t1 gleichberechtigt mit t4 und t3 um den Monitor „kämpfen“. Der Scheduler entscheidet über die weitere Reihenfolge für t1, t4 und t3.
Bemerkung:
Ein wartender Thread t1 kann nur geweckt werden, wenn das Monitor-Objekt, an das die Benachrichtigung geschickt wurde (monObj.notify()) identisch ist zum Monitor-Objekt, auf das der Thread t1 wartet (monObj.wait()).



Eine zweite Variante von wait() erlaubt die Angabe einer Zeitspanne, die maximal gewartet werden soll. Falls das Ereignis innerhalb dieser Zeit nicht eintritt, setzt der Thread (hier t1) seine Ausführung fort (sobald er den Monitor erhält). Er wird also nach der spezifizierten Zeitspanne auch ohne notify() automatisch in die Warteschlange zum Eintritt in den Monitor eingereiht.



(3) await() und signal() bei Locks

Analog zu wait() und notify() bei synchronized-Blöcken werden in Java bei Locks die Methoden await() und signal() verwendet. Diese Methoden werden an einem java.util.concurrent.lock.Condition-Objekt aufgerufen. Das Condition-Objekt wird von der Methode newCondition() einer Lock-Klasse zurückgeliefert.

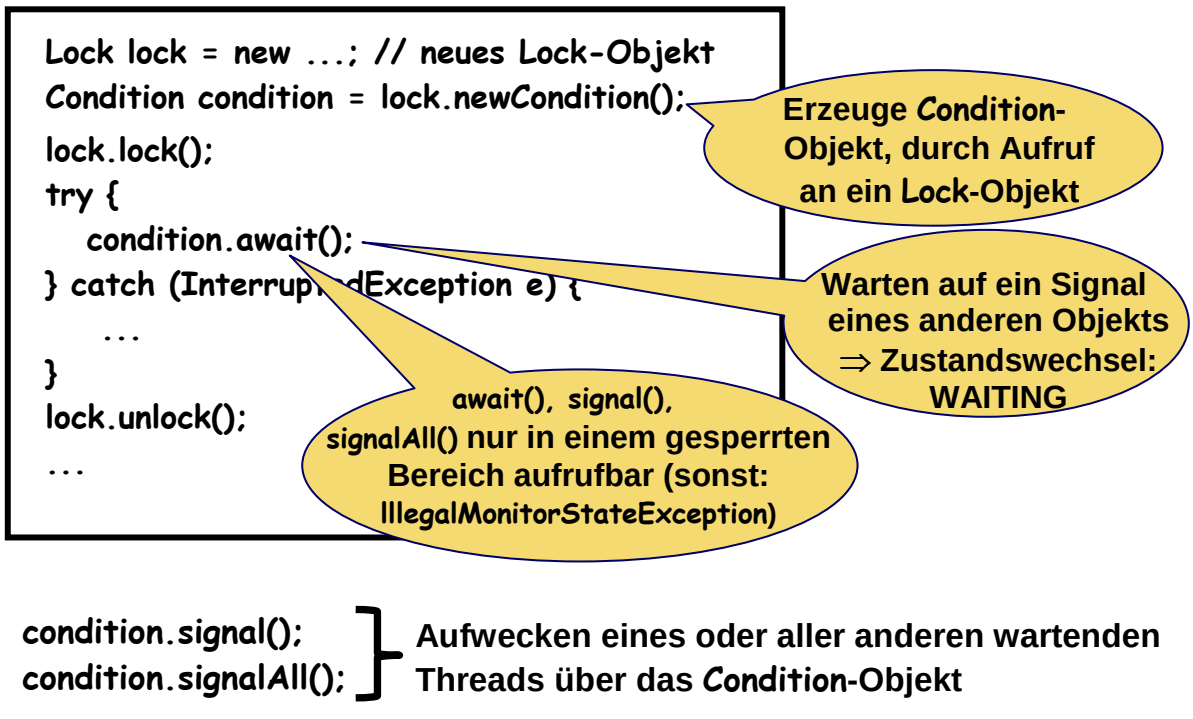
Um einen anderen Thread aufzuwecken, wird die Methode signal() verwendet.

Sollen alle wartenden Threads einen Hinweis bekommen, wird die Methode signalAll() verwendet.

Die Methoden await(), signal() und signalAll() können nur innerhalb eines gesperrten Bereiches aufgerufen werden.

Werden die Methoden außerhalb eines gesperrten Bereiches aufgerufen, wird eine java.lang.IllegalMonitorStateException geworfen

Mit der Methode await() geht der Thread in den Zustand „nicht ausführend“ über.




```
condition.signal();
condition.signalAll();
```
} Aufwecken eines oder aller anderen wartenden Threads über das Condition-Objekt

(4) Busy Wait(ing) statt synchronized-Block/wait()-Aufruf

- Idee: while-Schleife im Thread auf die Freigabe der Ressource warten lassen, z.B.
  `while (isFull());`
- Problem: Busy Wait(ing)
  Es wird immer wieder nachgefragt, ob weitergemacht werden kann oder nicht.
  Im Beispiel wird (z.B. im Rahmen eines Producer-Consumer Programms) immer wieder und solange die Methode isFull() aufgerufen, bis diese irgendwann schließlich false zurückliefert.

⇒ Sehr rechenintensiv, daher zu vermeiden!

⇒ Die Synchronisation ist nicht zuverlässig korrekt, wenn isFull() ungeschickt unterbrochen werden kann!

(5) Semaphoren

Semaphoren wurden 1968 von E.W. Dijkstra eingeführt.
Sie sind - analog zu Monitoren - ein auch außerhalb von Java verbreitetes Konzept.

Ziel und grundsätzliche Funktionsweise:
Semaphoren bilden einen Sperrmechanismus und ermöglichen die Synchronisation einer bestimmten Anzahl von Threads auf ein Programmstück.
Eine Semaphore verwaltet intern eine Zählvariable für die Menge von Erlaubnissen (engl. Permits). Ist das festgelegte Maximum erreicht, müssen weitere Threads warten, bis ein Thread den kritischen Bereich wieder verlässt.

Arten von Semaphoren:

- Binäre Semaphore: Es ist höchstens ein Thread pro kritischem Programmstück möglich. Die oben kennengelernten Konzepte (wait()/notify von Object, await()/signal() mit Condition) sind bereits binäre Semaphoren.
- Allgemeine Semaphoren: Eine bestimmte, begrenzte Menge an Threads können in den kritischen Abschnitt eintreten. Mit allgemeinen Semaphoren können auch binäre implementiert werden.

In Java: java.util.concurrrent.Semaphore

- Semaphore(int permits): Im Konstruktoraufruf wird die maximal mögliche Anzahl an Threads festgelegt, die im zusammen im kritischen Abschnitt sein dürfen.
- Semaphore(int permits, boolean fair): Wie für Semaphore(int permits) aber statt zufällige Auswahl, welcher der wartenden Threads in den kritischen Bereich darf, wird darauf geachtet, dass die Ressource fair verteilt wird (fair = true). Jeder Thread kommt dann im Schnitt gleich oft zum Zug.
- acquire(): Versuch in den kritischen Abschnitt einzutreten, Warten falls belegt, Verminderung des Erlaubniszählers bei Eintritt. Im Gegensatz zu einer isFull()-Methode wie oben in Abschnitt Busy Waiting enthalten, stellt die Semaphore als Sprachkonstrukt intern sicher, dass der Test und das Setzen nicht unterbrochen werden kann.
- tryAcquire(): Wie für acquire() aber kein Warten, sondern Rückkehr mit Rückgabewert false, falls der kritische Abschnitt schon mit der maximal erlaubten Anzahl an Threads belegt ist.
- release(): Verlassen des kritischen Abschnitts und Inkrementieren des Erlaubniszählers. Daraufhin wird von der Semaphore der frei gewordene Permit analog zum schon bekannten notify() an einen beliebigen Thread aus der Menge der wartenden Threads weitergegeben.
  Achtung: Wie bei Locks muss auf die Rückgabe der Erlaubnisse selbst geachtet werden. Mit einem finally-Konstrukt kann z.B. sichergestellt werden, dass die Erlaubnis zurückgegeben wird selbst wenn im kritischen Abschnitt eine Exception geworfen wird.

9.7 Klassische Probleme in der Synchronisation

Im Rahmen der Synchronisation haben sich Problemmuster herausgebildet, die typische Synchronisationsprobleme beschreiben. Einige sehr bekannte Probleme sind:

- Producer-Consumer bzw. Bounded Buffer,
- Dining Philosophers,
- Sleeping Barber.

Die Problembeschreibungen abstrahieren von konkreten Problemstellungen (z.B. die Koordination des Zugriffs mehrerer Threads auf das Netzwerk oder auf gemeinsame Ressourcen im Netzwerk) und reduzieren die Problemgröße auf die wesentlichen Bausteine. Dadurch sind sie als Ausgangsbasis für die Suche nach Lösungen gut geeignet.

Anforderungen an eine gute Synchronisationslösung:
(1) Zwei Prozesse bzw. Threads dürfen nicht gleichzeitig im kritischen Abschnitt sein.
(2) Es darf keine Annahme über Geschwindigkeit oder Anzahl der CPUs gemacht werden.
(3) Kein Prozess bzw. Thread, der nicht im kritischen Abschnitt ist, darf andere Prozesse oder Threads blockieren.
(4) Kein Prozess bzw. Thread muss unendlich lange warten bis er den kritischen Abschnitt betreten darf.

9.7.1 Producer-Consumer bzw. Bounded Buffer Problem

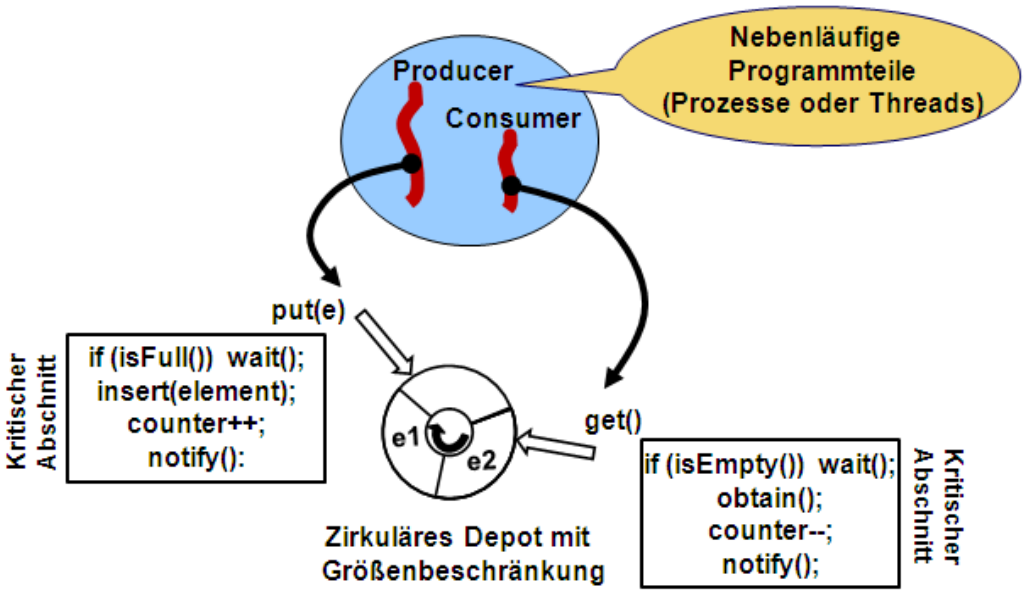
Problembeschreibung:
Zwei nebenläufig agierende Programmteile (z.B. Threads) greifen auf denselben Speicherbereich als Depot (*Buffer*) mit begrenzter Größe zu.
Ein Producer (Erzeuger) erzeugt Einheiten und legt sie ins Depot. Ein Consumer (Konsument) liest die Einheiten aus dem Depot. Ist das Depot voll, muss der Producer warten. Ist das Depot leer, muss der Consumer warten.

Anforderung an die Lösung:
Die Herausforderung an der Problemstellung ist die Gestaltung der Synchronisation derart, dass das Warten keine Ressourcen bindet und das Programm eine insgesamt optimale Laufzeit hat (d.h. z.B. kein unnötiges Warten).
Es sollen generell die oben genannten Anforderungen an eine gute Synchronisation gelten.

Erste Lösungsidee: Fest vorgegebene Reihenfolge (strenge Abwechslung)
Beispielsituation in einer Lösung mit fest vorgegebener Abwechslung:
Ein Depot hat 3 Plätze. Einer der Plätze wurde gerade frisch von einem Producer belegt, zwei weitere sind noch frei. Ein weiterer Producer will nun ins Depot schreiben. Obwohl das Depot noch Platz hat, ist das nicht möglich, da als Vorgabe als nächster ein Consumer an der Reihe ist. Der Producer muss nun solange warten, bis ein Consumer auf das Depot zugegriffen hat.
⇒ Verstoß gegen Anforderung (3):
Die Consumer außerhalb des kritischen Abschnitts blockieren den Producer.

Da wir keine Vorhersage über Producer und Consumer machen können, hilft es auch nicht, den Producer in der fest vorgegebenen Reihenfolge häufiger hintereinander einzuplanen.

Lösungsmöglichkeit für genau einen Producer und genau einen Consumer:



Der kritische Abschnitt muss dabei durch eine Synchronisationsmaßnahme (z.B. Monitore) geschützt werden.

Problemerweiterung: Multi-Producer-Consumer

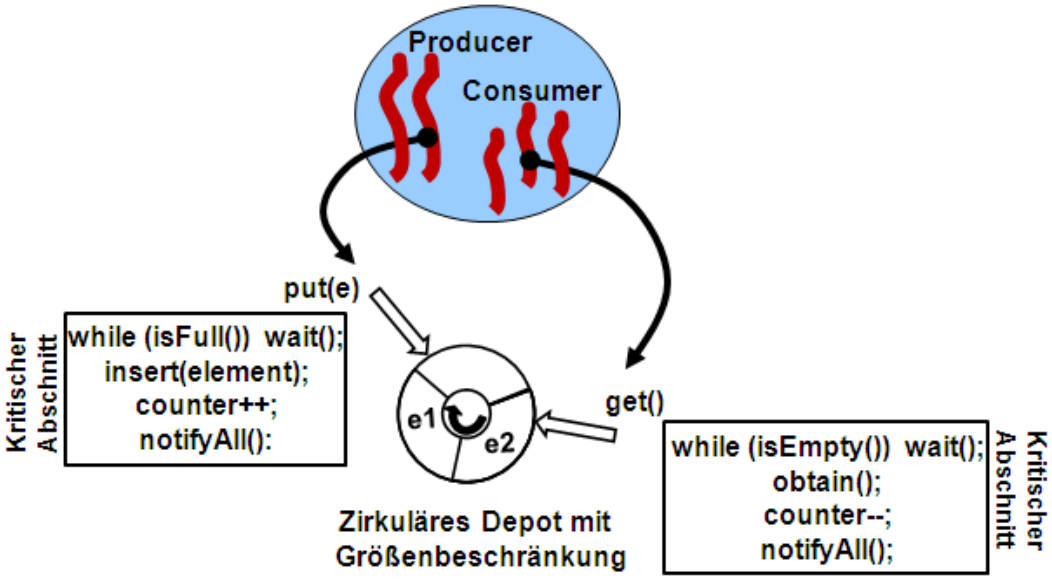
Mehrere Producer und Consumer sollen synchronisiert werden. Nur einer darf zu einem Zeitpunkt ins Depot schreiben oder aus diesem lesen.

Probleme mit der bisherigen Lösung:

- Beispielsituation 1:
  Alle Consumer und Producer warten. Ein Producer hat gerade frisch den letzten freien Platz belegt und sendet `notify()`. Zufällig wird dadurch ein anderer, wartender Producer aktiviert und setzt seine Arbeit nach der Programmstelle fort, an der er bisher wartete. Die `isFull()`-Prüfung liegt dadurch bereits hinter ihm. Der geweckte Producer legt damit ein weiteres Element in das Depot, obwohl kein Platz mehr vorhanden ist (Ausnahmesituation!).
  ⇒ Abhilfe: Erneute Prüfung des Füllungsgrads des Depots durch `while` statt `if`:
  `while(isFull()) wait();`

- Beispielsituation 2 mit der um `while` erweiterten Lösung:
  Die Situation sei zunächst wie in Beispielsituation 1. Ein Producer wird wieder zufällig mit `notify()` benachrichtigt. Er setzt seine Arbeit nun innerhalb der neuen `while`-Schleife fort. Die erneute Prüfung auf `isFull()` ergibt, dass das Depot tatsächlich bereits voll ist. Wir haben erfolgreich verhindert, dass der Producer versucht, das schon volle Depot über die Begrenzung hinaus zu füllen.
  Allerding führt die Prüfung dazu, dass der Producer sich wieder in die Warteposition begibt. Damit warten alle Producer und alle Consumer; keiner kann `notify()` senden; das Programm hängt.
  ⇒ Abhilfe: Statt zufällige Benachrichtigung nur eines Producers oder nur eines Consumers, Aufwecken aller in der internen Warteschlange:
  `notifyAll()`.

Lösungsmöglichkeit:



9.7.2 Dining Philosophers

Das Dining Philosophers Problem ist als klassisches Problem für Deadlocks bekannt.

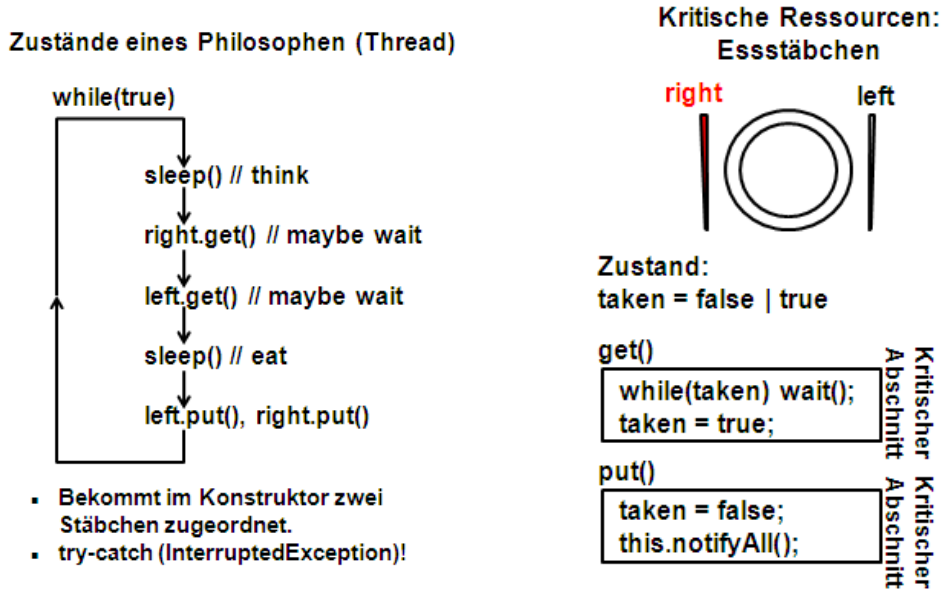
Problembeschreibung:
An einem runden Tisch sitzen n Philosophen (engl. Philosopher), die nur essen und denken, aber nicht sprechen ($n \geq 2$). Zwischen je zwei Philosophen liegen Essstäbchen (engl. Chopstick) auf dem runden Tisch.
Irgendwann nach der Denkarbeit bekommt ein Philosoph Hunger und greift das Stäbchen, das rechts zu ihm auf dem Tisch liegt. Anschließend greift er nach dem linken Stäbchen. Er kann essen, wenn er beide Stäbchen hat. Anschließend legt er die Stäbchen zurück und wartet wieder eine Zeitlang (d.h. er denkt wieder nach). Der Vorgang wiederholt sich unendlich. Kann der Philosoph eines der Stäbchen nicht greifen, weil es gerade von einem anderen Philosophen benutzt wird, wartet er, bis das Stäbchen wieder frei ist.

[Aus: Wikipedia, 2011]

Bemerkung:
Während in der englischen Literatur von *Chopsticks* (gelegentlich *Forks*) gesprochen wird, finden sich in der deutschen Literatur meist Gabeln auf dem runden Tisch.

Wir benötigen ein Objekt je *Philosopher* (angeordnet z.B. in einem Array) und ein Objekt je *Chopstick*. Die Anzahl der Essstäbchen ist gleich der Anzahl der Philosophen (d.h. gleiche Array-Größe).
Jedem Philosophen (mit Nummer PHILOSOPHENNR) werden die für ihn greifbaren Stäbchen links und rechts von ihm zugeordnet:
ChopStick[PHILOSOPHENNR] und
ChopStick[(PHILOSOPHENNR+1) mod ANZAHL_PHILOSOPHEN].

Die Philosophen durchlaufen in einer Endlosschleife folgende Zustände:



Mögliche Probleme der Lösung rechts im Bild:

- Deadlock: Alle Philosophen haben das rechte Stäbchen. Wenn alle Warten, kann keiner mehr `notify()` senden, so dass kein Philosoph den Wartezustand mehr verlassen kann.
- Mangelnde Fairness bis hin zu *Starvation* (Verhungern): Unter Umständen kommt ein Philosoph nie zum Zug.

Verbesserungsideen:

- Verhinderung eines Deadlocks durch Entfernen der `sleep()`-Anweisung zwischen `right.get()` und `left.get()`.
  ⇒ Änderung der originären Problemstellung (unerwünscht).
  ⇒ Deadlocks werden nicht verhindert, da auch ohne `sleep()` ausreichend kleine Zeitverschiebungen und zufällige Unterbrechungen durch den Scheduler auftreten können.
- Ein Philosoph handelt anders als die übrigen: Er greift erst links dann rechts.
  ⇒ Änderung der originären Problemstellung (unerwünscht).
  ⇒ Kein Deadlock, aber auch keine gleichberechtigte Ressourcenverteilung:
  Er selber und sein linker Nachbar bekommen weniger zu essen als die anderen. Der rechte Nachbar zum neuen Außenseiter wird bevorteilt (keine Fairness).
- Nachgeben, wenn ein Philosoph das zweite Stäbchen nicht bekommt, d.h. Zurücklegen des rechten Stäbchens, wenn er das linke Stäbchen nicht ebenfalls bekommt. Belegte Ressourcen werden dadurch immer wieder freigegeben. Es gibt damit kein Warten auf dem zweiten Stäbchen (`get()` implementiert `return false` anstelle des `wait()`).

Zur Verbesserung der Fairness können wir die Priorität für Philosophen erhöhen, wenn sie zurücklegen mussten. Nach dem erfolgreichen Essen wird ihre Priorität wieder auf den Normwert gesetzt.
⇒ Kein Deadlock.
⇒ Änderung der originären Problemstellung (unerwünscht). Eine Ressourcenfreigabe und ein Verhandeln zwischen den Philosophen war nicht gewollt.

- Einsatz von fairen Semaphoren (vgl. Abschnitt zum Thema Semaphoren).

Klassisches Problem für Deadlocks: Ohne Änderung der ursprünglichen Aufgabenstellung gibt es keine Problemlösung, die frei von Deadlocks ist.

9.7.3 Sleeping Barber

Das Sleeping Barber Problem ist ein klassisches Problem für den Einsatz von Semaphoren.

Problembeschreibung:
Ein Frisörsalon hat n Wartestühle (begrenzte Anzahl) und einen Frisierstuhl (engl. *Barber chair*). Ein neuer Kunde betritt den Salon und setzt sich auf einen freien Wartestuhl (engl. *Waiting chair*). Ist keiner frei, geht er wieder und kommt später wieder.
Ein auf einem Warteplatz wartender Kunde schaut, ob der Frisierstuhl frei ist. Falls dieser frei ist, nimmt er dort Platz und weckt den bis dahin im Frisierstuhl schlafenden Frisör (engl. *Barber*). Der Frisör schneidet dem Kunden daraufhin die Haare.
Anschließend gibt der Kunde den Frisierstuhl wieder frei und der Frisör schläft wieder ein.
Jeder Kunde kommt nach einem gewissen Zeitintervall wieder.

Einschränkungen:

- Nur einem Kunden (dem auf dem Frisierstuhl) kann gleichzeitig die Haare geschnitten werden.
- Die Anzahl der wartenden Kunden ist begrenzt (auf die Anzahl verfügbarer Warteplätze).
- Es sollen die üblichen Anforderungen an Synchronisationslösungen gelten und möglichst wenige Ressourcen verbraucht werden.