Digitally signed by Mario Jeckle
Reason: I am the author of this document
Location: www.jeckle.de
Date: 2004.01.12 09:35:16 +01'00'

jeckle.de

Vorlesung Java Threads

- Vorlesungstermine
- Gliederung
- Netzressourcen zu in der Vorlesung behandelten Themen

- Einführung
- Java API-Unterstützung zur Threadprogrammierung
- Synchronisationskonzepte und -primitive
- Synchronisationsprobleme und ihre Lösung
- Verklemmungen
- Weiterführende Konzepte
- Anwendungsbeispiele

- Definitionsverzeichnis
- Schlagwortverzeichnis
- Abbildungsverzeichnis
- Verzeichnis der Beispiele

Vorbemerkung

Dieses Scriptum enthält vorlesungsbegleitende Informationen zur Vorlesung *Java Threads* am Fachbereich Informatik der Fachhochschule Augsburg. Es stellt keinen Lehrbuchersatz dar, und ist daher nur beschränkt zum Selbststudium geeignet. Seine Aufgabe ist es vielmehr die Schlüsselbegriffe und -Aussagen der Vorlesung festzuhalten und um die dort diskutierten Beispiele zu ergänzen.
Vertiefende Information zum Thema kann der empfohlenen Literatur entnommen werden.
Mit hoher Wahrscheinlichkeit enthält dieses Scriptum (noch) den ein oder anderen Fehler. Für Hinweise aller Art ist der Autor jederzeit dankbar!
Bei der betrachteten Java-Version handelt es sich durchgängig, soweit nicht anders vermerkt, um JDK 1.4, welches kostenfrei von der Web-Seite der Firma Sunsoft bezogen werden kann.

Einführung

Ziel der Vorlesung: Verständnis Thread-gestützter (Parallel-)Programmierung mit der Programmiersprache Java und ihrer Ausführungsumgebung.

Was vermittelt die Vorlesung?

- Grundlagen nebenläufiger Verarbeitung
- Java-API zur Realisierung Thread-gestützter Programme
- Bekannte Problemstellungen nebenläufiger Verarbeitung
- ... und ihre Lösungsmöglichkeiten und -Ansätze
- ... an Beispielen

Was vermittelt die Vorlesung nicht?

- Java-Grundlagen
- Betriebssystem-Grundlagen
- Vertiefung der Architektur und Umsetzung verteilter Systeme

Definition 1: *Parallelität*
Im Wortsinne: *gleichartige Beschaffenheit*.
Rechner ermittelt automatisch, ob einzelne Abarbeitungselemente (etwa: Prozeduren, Befehle,

Maschinen-Instruktionen) „gleichzeitig“ ausgeführt werden können und führt dies, falls möglich, für den Anwender und Programmierer transparent durch. Voraussetzung hierfür ist die Existenz logisch trennbarer Aktivitäten.
Die tatsächliche Abarbeitungsreihenfolge kann daher von der codierten abweichen!
Auf Maschinen mit nur einer einzelnen CPU lassen sich im allgemeinen nur **quasi-parallele** Abläufe realisieren. Bekanntestes Beispiel hierfür sind Pipline- oder Superskalararchitekturen.
Echter Parallelismus erfordert üblicherweise mehr als eine CPU, bzw. geeignet gestaltete Recheneinheiten (d.h. mehrere Befehlszähler und Statusregister, etwa bei Intels *Hyper-Threading*-Architektur für Pentium 4)

Parallelverarbeitung kann die Verarbeitung auf mehreren physisch getrennten vollständigen Maschinen (d.h. mit eigenen CPU, Speicher, E/A-Einheiten umfassen.

Definition 2: *Nebenläufigkeit (Concurrency)*
Parallele Ausführung von Anweisungen auf einem oder mehreren Prozessoren. Die Organisation der Parallelität erfolgt hierbei explizit durch den Programmierer in einer geeigneten Hochsprache.

Nebenläufige Ausführung ist auf eine einzige physische Maschine beschränkt. Diese kann jedoch mehrere vollständige CPUs enthalten.
Bei Java-Threads handelt es sich daher um eine Möglichkeit der nebenläufigen Ablaufsteuerung, da diese explizit (konkret: in Form von API-Aufrufen) durch den Applikationprogrammierer festgelegt wird.
Durch die Nebenläufigkeit tritt der codierte Kontrollfluß in den Hintergrund, zugunsten eines durch den Programmierer nicht beeinflußbaren *Nichtdeterminismus*. Hierbei obliegt es ausschließlich dem prozessorzeitzuteilenden Betriebsystem in welcher Reihenfolge die einzelnen Anweisungen ausgeführt werden. Dabei muß eine einmal eingetretene Ausführungsreihenfolge Die Literatur trifft die Unterscheidung zwischen *Parallelität* und *Nebenläufigkeit* nicht immer trennscharf und gleichermaßen eindeutig; für die Vorlesung sind die in den Definitionen 1 und 2 gegebenen bindend.

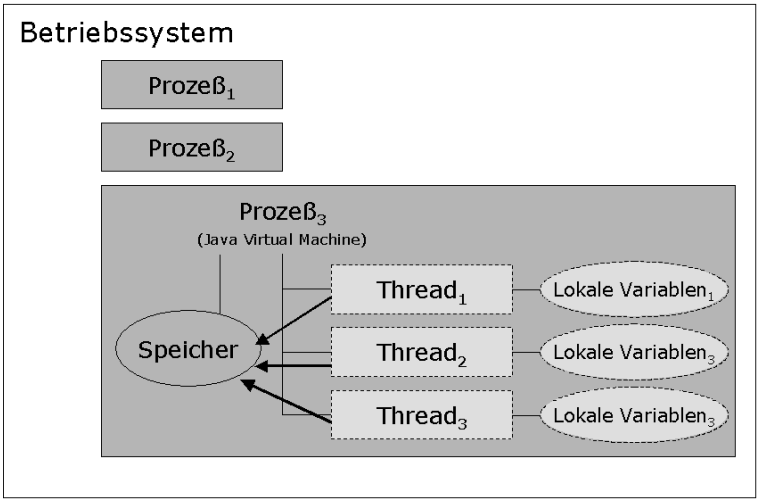
Definition 3: *Prozeß*
Bezeichnet ein im Speicherzugriff befindliches ablauffähiges Programm mit seinen dafür notwendigen betriebsystemseitigen Datenstrukturen wie zugeordneten Eigenschaften (z.B. Stack- und Programmzähler, Prozeßzustand, sowie Eigenschaften der Speicher- und Dateiverwaltung).
siehe auch: BS-Skript

Ein Prozeß wird durch das Betriebsystem als eigenständige Instanz -- in der Regel unabhängig und geschützt von anderen -- ausgeführt. Multitaskingsysteme (wie UNIX und neuere Generationen der Windowsfamilie) gestatten die parallele Prozessausführung.

Definition 4: *Thread*
Ein Thread (engl. für *Faden*) stellt eine nebenläufige Ausführungseinheit innerhalb genau eines Prozesses dar.

Aufgrund dieser Definition erben Threads viele der prozeßtypischen Eigenschaften, wie Zustand, Programmzähler etc. Den Hauptunterschied zu voll entwickelten Prozessen bildet der zwischen allen Threads eines Prozesses geteilte Speicher. Gleichzeitig besitzt jeder Thread seinen eigenen lokalen Speicherbereich in dem u.a. die lokalen Variablen verwaltet werden. Aus diesen Gründen werden Threads oft auch als leichtgewichtige Prozesse bezeichnet.
Alle Threads bewegen sich im Ausführungskontext des erzeugenden Prozesses d.h. „externe Operationen“ wie Plattenzugriffe etc. wirken sich über Threadgrenzen hinaus aus.
In Multithretaing-Systemen besitzt jeder ablaufende Prozeß mindestens einen Thread, der den Kontrollfluß realisiert.
Multitasking und Multithreating bedingen sich daher gegenseitig. Ohne die systemseitige Unterstützung der parallelen Taskausführung kann keine prozeßinterne Threadabarbeitung erfolgen.

Abbildung 1: *Multithreaded-Prozesse in einer Multitasking-Umgebung*



(click on image to enlarge!)

Warum Thread-basierte Programmierung?

Ein wenig Geschichte ...

- 1940er Jahre: Stapelbetrieb.
  Aufträge (engl. *Jobs*)werden sequentiell an wenige existierende Rechner übermittelt.
  Die Eingabe erfolgt zumeist über Lochstreifen, später Lochkarten. Die Auftragserfüllung ist nicht-interaktiv und kann während ihrer Durchführung nicht beeinflußt werden.
- ab 1965: Mehrprogrammbetrieb.
  Die knappe und teure Rechenzeit genutzt um mehrere Aufträge zeitlich verzahnt abzuarbeiten. So kann eine Aufgabe rechnend ausgeführt werden, während andere beispielsweise auf Ein-/Ausgaben warten.
- ab 1970er Jahren: Mehrbenutzerbetrieb.
  Abweichung von der aufgabenzentrierten Betrachtung, hin zu einer dialogorientierten Sichtweise. Mehrere Benutzer können simultan (d.h. quasi Parallel) Prozesse auf derselben Maschine ausführen. Die Illusion der gleichzeitigen Ausführung wird durch das Betriebssystem erzeugt, welches in rascher Folge zwischen den ablaufenden Prozessen umschaltet (sog. Time-Sharing-Betrieb).
- in den 1980er Jahren: Thread-Konzept:
  Die schwergewichtigen Prozesse erwiesen sich als zunehmend inadäquat für komplexe Problemstellungen, insbesondere Fragestellungen, die eine inhärente Parallelität aufweisen wie bestimmte numerische Fragestellungen (Paradebeispiel: Matrizenmultiplikation). Ihr betriebssystemseitiger Verwaltungsaufwand verhindert die schnelle Erzeugung und der (im allgemeinen gewünschte) Schutz der Prozesse untereinander erschwert den Datenaustausch bei der Operation auf gemeinsamen Daten.

Warum Threads in Java?

Eigentlich eine zweiteilige Frage ...

1. Warum sollte sich der Programmierer um das Thema kümmern?
2. Warum unterstützt die Java-API und die virtuelle Maschine Threads?

Skizze einer Antwort: zu 1.)

- Threads nutzen die vorhandenen Ressourcen moderner Betriebssysteme und Hardware effizienter aus.
  Beispielsweise durch Weiterführung der Berechnungspperationen, während andere Threads auf Ein-/Ausgaben warten.
- Threads werden durch ein geeingetes Betriebssystem für den Programmierer und Applikationsanwender transparent auf mehrere Prozessoren (fall vorhanden) verteilt.
- Threads erhöhen den Interaktionskomfort für den Applikationsanwender (Beispiel: GUI-intensive Anwendungen).
  Berechnungsaufwendige Arbeiten können „in den Hintergrund“ verlagert werden.
- Threads sind in Erzeugung, Handhabung und Ressourcenbedarf leichtgewichtiger als vollentwickelte Prozesse und erzeugen daher weniger Systemlast.

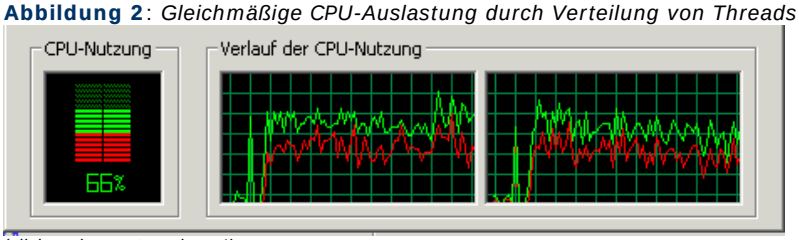
zu 2.)

- ... Hochsprachenschnittstelle, da in Java kein direkter Hardware- und Betriebssystemzugriff möglich.
- ... Abstraktionsschicht von Thread-Implementierung in Java Virtual Machine.
- ... Erleichterung und dadurch Forcierung nebenläufiger Programmierung.

- ... Portabilität Thread-basierter Anwendungen auf andere Hardwareplattformen, die Java unterstützen.

Die Verwendung von Threads führt im technischen Sinne nicht zu einer beschleunigten Ausführung, im Gegenteil, durch Verwaltungsaufwände bei der Erzeugung und Koordination ergibt sich im allgemeinen sogar eine insgesamt vergrößerte Ausführungszeit, jedoch bedingt die effizientere Ressorucennutzung einerseits die bessere Auslastung der vorhanden Hardware und gleichzeitig entsteht für den Anwender der Eindruck einer flüssigeren Verarbeitung.
Erst beim Einsatz mehrere Prozessoren in einer Maschine ergibt sich ein echter positiver Laufzeiteffekt durch die Möglichkeit Threads auf verschiedenen CPUs zur Ausführung zu bringen.
Die Verteilung und Koordination obliegt herbei dem Betriebssystem und erfolgt für den Hochsprachenprogrammierer transparent.

Abbildung 2: *Gleichmäßige CPU-Auslastung durch Verteilung von Threads auf zwei Prozessoren*



(click on image to enlarge!)

Java API-Unterstützung zur Threadprogrammierung

Zwei generelle Ansätze:

1. Durch (spezialisierende) Ableitung von der Klasse `Thread`
2. Durch Implementierung der Schnittstelle `Runnable`

Der erste Fall läßt sich auf den zweiten zurückführen, da die Klasse `Thread` selbst die Schnittstelle `Runnable` implementiert.
Klasse und Schnittstelle sind im Standardpaket `java.lang` organisiert, das im Rahmen des Compilierungsvorganges automatisch importiert wird.

Definition 5: *Implementierung nebenläufiger Abläufe in Java*
Jeder nebenläufige Programmfaden wird generell durch eine eigenständige Klasse repräsentiert, welche die Schnittstelle `Runnable` implementiert.

Die Schnittstelle `Runnable` definiert als einzige Methode `run`, welche durch das Laufzeitsystem automatisch zu Beginn der Thread-gestützten Verarbeitung zur Ausführung gebracht wird.
Daher sollte diese Methode niemals direkt auf einem Thread-Objekt aufgerufen werden, sondern jeder Thread ausschließlich mit der dafür vorgesehenen Methode `start` dem Laufzeitsystem als rechenwillig gemeldet werden.
Threadzustände

Definition 6: *run-Methode*
Die `run`-Methode wird nach der Threaderzeugung automatisch durch das Laufzeitsystem asynchron ausgeführt.

Ihre Rolle entspricht daher konzeptionell einer `main`-Methode für Threads.

Beispiel 1: *Nebenläufige Implementierung auf Basis der Ableitung von Thread*
Quellcode Ausgabe der Ausführung

```
(1)public class ThreadBased1
(2){
(3)     public static void main(String[] argv)
(4)     {
(5)             HelloThread northGerman = new HelloThread( "Moin Moin" );
(6)             HelloThread southGerman = new HelloThread( "Gruess Gott" );
(7)
(8)             northGerman.start();
(9)             southGerman.start();
(10)
(11)            System.out.println( "threads started ..." );
(12)    } //end main()
```

```
(13)} //end class ThreadBased1
(14)// ************************************************************
(15)class HelloThread extends Thread
(16){
(17)    protected String greetingText;
(18)
(19)    public HelloThread (String greetingText)
(20)    {
(21)            this.greetingText = greetingText;
(22)    } //standard constructor
(23)
(24)    public void run()
(25)    {
(26)            while (true)
(27)            {
(28)                    try
(29)                    {
(30)                            Thread.sleep(500);
(31)                    } //try
(32)                    catch (InterruptedException ie)
(33)                    {
(34)                            System.out.println("an InterruptedException occurred\n"+ie.toString()
+"\n"+ie.getMessage() );
(35)                    } //catch
(36)
(37)                    System.out.println( greetingText);
(38)            } //while
(39)    } //run()
(40)}//class HelloThread
```

Das Beispiel erzeugt zwei Threads (Zeile 5 und 6), die beide alle 500 Millisekunden (siehe Ausführungssuspendierung in Zeile 30) einen fixen Text am Bildschirm ausgeben.
Die beiden Threads sind beide Objekte der Klasse `HelloThread` welche von `Thread` erbt (Zeile 15).

Das 2. Beispiel setzt die zuvor auf Basis der Ableitung von der Klasse `Thread` gezeigte Implementierung durch Realisierung der Schnittstelle `Runnable` um.

Beispiel 2: *Nebenläufige Implementierung auf Basis der Realisierung der Schnittstelle Runnable*
Quellcode Ausgabe der Ausführung

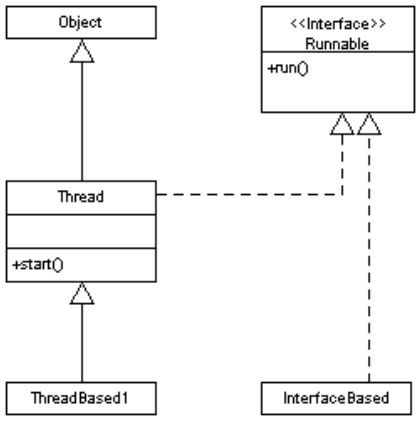
```
(1)public class InterfaceBased1
(2){
(3)     public static void main(String[] argv)
(4)     {
(5)             HelloThread northGerman = new HelloThread( "Moin Moin" );
(6)
(7)             Thread northGermanThread = new Thread( northGerman );
(8)             Thread southGermanThread = new Thread( new HelloThread( "Gruess Gott" ) );
(9)
(10)            northGermanThread.start();
(11)            southGermanThread.start();
(12)
(13)            System.out.println( "threads started ..." );
(14)    } //end main()
(15)} //end class InterfaceBased1
(16)// ************************************************************
(17)class HelloThread implements Runnable
(18){
(19)    protected String greetingText;
(20)
(21)    public HelloThread (String greetingText)
(22)    {
(23)            this.greetingText = greetingText;
(24)    } //standard constructor
(25)
(26)    public void run()
(27)    {
(28)            while (true)
(29)            {
```

```
(30)                    try
(31)                    {
(32)                            Thread.sleep(500);
(33)                    } //try
(34)                    catch (InterruptedException ie)
(35)                    {
(36)                            System.out.println("an InterruptedException occurred\n"+ie.toString()
+"\n"+ie.getMessage() );
(37)                    } //catch
(38)
(39)                    System.out.println( greetingText);
(40)            } //while
(41)    } //run()
(42)}//class HelloThread
```

Der Code gleicht dem vorangegangenen Beispiel, lediglich, daß die Klasse (`HelloThread`) deren Objekte nebenläufig ausgeführt werden hier die Schnittstelle `Runnable` implementiert (Zeile 17). Zusätzlich kann die Methode `start` in dieser Umsetzungsvariante nicht mehr direkt auf Objekten der Klasse `HelloThread` ausgeführt werden (Zeile 10 und 11). Stattdessen müssen Ausprägungen der Klasse als Ausprägungen der Klasse `Thread` aufgefaßt werden (Zeile 7 und 8).

Die statische Struktur der beiden Beispiele ergibt sich daher als UML-Klassendiagramm:

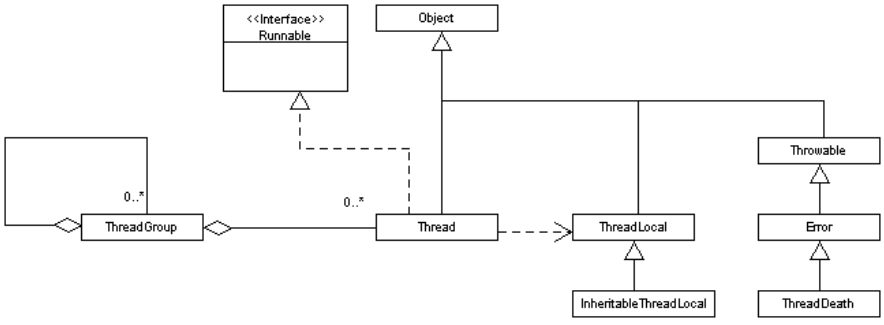
Abbildung 3: *UML-Klassendiagramm der ersten beiden Code-Beispiele*



(click on image to enlarge!)

Nachdem an den Beispielen bereits mit der Klasse `Thread` und der Schnittstelle `Runnable` die beiden wesentlichen Elemente der threadgestützten Programmierung in Java angerissen wurden, nachfolgend eine Übersicht der Funktionsangebots der kompletten Java API für diesen Problemkreis. Das UML-Klassendiagramm der Abbildung 4 zeigt die fundamentalen Klassen, sowie die Schnittstelle `Runnable`.

Abbildung 4: *Thread-API in Java v1.4*



(click on image to enlarge!)

Die Klasse `Thread`

Die Klasse `Thread` findet sich im Paket `java.lang`.

Quellcode der Klasse `Thread`

Die nachfolgende Zusammenstellung basiert auf der tatsächlichen Implementierung der Klasse `Thread` und enthält daher auch für den Programmierer im allgemeinen nicht zugängliche Attribute und Operationen. Aus Gründen der Förderung des Verständnis der internen Abläufe und Vollständigkeit der Charakteristika der einzelnen API-Primitive erfolgt daher auch ihre Darstellung.

Attributdefiniton	Semantik
`private volatile Interruptible blocker`	Verweis auf das Objekt welches die Ausführung des Threads durch ein (unterbrechbare) E/A-Operation unterbrochen hat.
`private ClassLoader contextClassLoader`	ClassLoader für diesen Thread.
`private boolean daemon = false`	Gibt an ob ein Thread als *Daemonthread* ausgeführt wird. Ist dies der Fall, so kann eine Applikation terminieren, die ausschließlich arbeitende Daemonthreads enthält.
`private long eetop`	Keine Dokumentation verfügbar. Dieses Attribut wird überdies weder in der definieren Klasse, noch an einer anderen Stelle der Java-API verwendet.
`private ThreadGroup group`	Gruppe der ein Thread zugeordnet ist.
`ThreadLocal.ThreadLocalMap inheritableThreadLocals`	Diejenigen threadspezifischen Variablen die an Kindthreads weitergegeben werden können.
`private AccessControlContext inheritedAccessControlContext`	Ererbter Zugriffskontrollkontext zur Ermittlung der Berechtigungen für den Zugriff auf Systemressourcen.
`private char[] name`	Name des Threads. Jeder Thread eines Prozesses trägt einen Namen. Weist der Programmierer keinen solchen zu, so wählt das Laufzeitsystem einen eigenen. Diese Namen gehorchen der Form `Thread-`*`laufendeNummer`*
`public final static int MIN_PRIORITY = 1`	Minimalpriorität eines Threads. siehe Java API Specification
`public final static int NORM_PRIORITY = 5`	Standardpriorität eines Threads. siehe Java API Specification
`public final static int MAX_PRIORITY = 10`	Maximalpriorität eines Threads. siehe Java API Specification
`private int priority`	Priorität des Threads. Die numerische Priorität eines Threads wird zur Zuteilung der Rechenzeit herangezogen. Sie kann sich in einem durch die Konstanten `MIN_PRIORITY` und `MAX_PRIORITY` begrenzten Intervall bewegen. Standardmäßig wird jeder Programmfaden mit der durch die Konstante `NORM_PRIORITY` festgelegten Priorität ausgeführt.
`private boolean single_step`	Gibt an ob ein Thread im *single-step-Modus* ausgeführt wird. Vermutlich für Debuggingzwecke verwendet.
`private long stackSize`	Angefragter Stackbereich für einen Thread. Ist dieser Wert 0, so wurde durch den Erzeuger keine Festlegung getroffen. Virtuelle Maschinen sind in der Behandlung dieses Wertes vollkommen frei; sie können ihn sogar gänzlich ignorieren.
`private boolean stillborn = false`	Gibt an ob die Terminierung des Threads noch vor Beginn seiner Ausführung angeordnet wurde.
`private static RuntimePermission stopThreadPermission`	Enthält die Rechte welche einem Thread zur Laufzeit eingeräumt werden.
`private Runnable target`	Verweis auf den ausführbaren Kern (d.h. die Implementierung) eines Threads.
`private static int threadInitNumber`	Automatisch geführter Zähler zur eineindeutigen Identifikation.

ThreadLocal.ThreadLocalMap threadLocals	Verweis auf diejenigen lokalen Variablen eines Threads die namens- und strukturgleich in allen Threads existieren, aber in jedem Programmfaden mit einem anderen Wert belegt sein können.
private Thread threadQ	Keine Dokumentation verfügbar. Dieses Attribut wird überdies weder in der definieren Klasse, noch an einer anderen Stelle der Java-API verwendet.

Operation	Semantik
public Thread()	Erzeugt eine neuen Thread. Die Wirkung entspricht Thread(null,null,*gname*), wobei *gname* den Namen des neuen Threads festlegt. siehe Java API Specification
public Thread(Runnable target)	Erzeugt einen neuen Thread. Die Wirkung entspricht Thread(null,*target*,*gname*), wobei *target* die nebenläufig auszuführende Klasse bezeichnet und *gname* den Namen des neuen Threads festlegt. siehe Java API Specification
public Thread(ThreadGroup group, Runnable target)	Erzeugt eine neue Thread. Die Wirkung entspricht Thread(group,target,gname), wobei *group* die Threadgruppe bezeichnet, innerhalb der der neu erzeugte Thread ablaufen soll. *target* bezeichnet die nebenläufig auszuführende Klasse und *gname* den Namen des neuen Threads, der automatisch generiert wird. siehe Java API Specification
public Thread(String name)	Erzeugt einen neuen Thread des durch name bezeichneten Namens. Die Wirkung entspricht Thread(null, null, name). siehe Java API Specification
public Thread(ThreadGroup group, String name)	Erzeugt einen neuen Thread. Die Wirkung entspricht Thread(group, null, name), wobei *group* die Threadgruppe bezeichnet, innerhalb der der neu erzeugte Thread ablaufen soll. name bezeichnet den Namen es neuen Threads. siehe Java API Specification
public Thread(Runnable target, String name)	Erzeugt einen neuen Thread. Die Wirkung entspricht Thread(null, target, name), wobei *target* die nebenläufig auszuführende Klasse bezeichnet und *name* den Namen des neuen Threads festlegt. siehe Java API Specification
public Thread(ThreadGroup group, Runnable, String name)	Erzeugt einen neuen Thread. *target* bezeichnet die nebenläufig auszuführende Klasse, welche zwingend die Schnittstelle Runnable implementiert. *name* bezeichnet den Namen des Threads. *group* bezeichnet im Falle der Existenz die Threadgruppe der der neue Thread zugeordnet werden soll. Ist diese null und ein SecurityManager existiert, so wird dessen Methode getThreadGroup zur Ermittlung herangezogen. Existiert kein SecurityManager oder getThreadGroup liefert null, so wird der neu erzeugte Thread derselben Gruppe zugeordnet wie der erzeugende Thread. Existiert ein SecurityManager so wird seine checkAccess-Methode aufgerufen um zu ermitteln, ob der Thread zur Erzeugungsoperation berechtigt ist. Ist er dies nicht, so wird ein Ausnahmeereignis des Typs SecurityException erzeugt. Ist das *target*-Argument ungleich null, so wird die Methode run-Methode des referenzierten Objektes zum Startzeitpunkt des Threads aufgerufen. Die Priorität eines neue erzeugten Threads entspricht der des erzeugenden Threads. Handelt es sich beim erzeugenden Thread um einen Daemon-Thread, so wird der neu erzeugte auch als Daemon-Thread markiert.

	siehe Java API Specification
`public Thread(ThreadGroup group, Runnable target, String name, long stackSize)`	Erzeugt einen neuen Thread. Das Verhalten ist identisch zu `Thread(ThreadGroup,Runnable,String)`, mit Ausnahme der Möglichkeit die für den Thread zu reservierende Stackgröße durch den Parameter *stackSize* vorgeben zu können. Dieser Parameter kann genutzt werden um durch die Erhöhung des für den Programmfaden reservierten Speichers eine höhere Rekursionstiefe zu erreichen. Jedoch ist die Behandlung dieser Angabe im Konstruktor stark plattformabhängig und kann in verschiedenen Java-Implementierungen differieren! Die restlichen Parameter verhalten sich identisch zu `Thread(ThreadGroup,Runnable,String)`. Diese Methode erzeugt innerhalb des Threadsystems eine Referenz auf jedes nebenläufig auszuführende Objekt (d.h. das durch den Parameter *target* referenzierte). Wird ein erzeugter Thread nicht durch `start` gestartet und geeignet terminiert, so verbleibt das referenzierte Objekt dauerhaft im Speicher. siehe Java API Specification
`public static int activeCount()`	Liefert die Anzahl der aktiven Threads innerhalb der Threadgruppe zurück, der der aktuelle Thread angehört. siehe Java API Specification
`private void blockedOn(Interruptible b)`	Setzt `blocker`-Attribut durch Reflexionsmechanismen des NIO-Paketes.
`public final void checkAccess()`	Prüft ob der aktuelle Thread zur Änderung der Threadeigenschaften berechtigt ist. Zur Durchführung der Prüfung muß ein `SecurityManager` im System existieren. Liegt die Berechtigung zur Modifikation für den anfragenden Thread nicht vor, so wird ein `SecurityException`-Ausnahmeereignis erzeugt. siehe Java API Specification
`public native int countStackFrames()`	Gibt die Anzahl der Stackframes eines zuvor unterbrochenen Threads zurück. Diese Methode ist als *deprecated* gekennzeichnet und sollte daher nicht mehr verwendet werden! siehe Java API Specification
`public static native Thread currentThread()`	Liefert eine Referenz auf den aktuellen Thread zurück. siehe Java API Specification Codebeispiel
`public void destroy()`	Zerstört einen Thread augenblicklich, ohne durch ihn gesetzte Sperren freizugeben. Diese Methode ist nicht implementiert und führt beim Aufruf zu einem Ausnahmeereignis des Typs `NoSuchMethodError`. siehe Java API Specification
`public static void dumpStack()`	Gibt den Stacktrace des aktuellen Threads aus. Diese Methode ist lediglich zur Fehlersuche sinnvoll einsetzbar. Intern wird zur Ausgabe des aktuellen Stackinhaltes ein Ausnahmeereignisobjekt des Typs `Exception` erzeugt. siehe Java API Specification
`public static int enumerate(Thread[] tarray)`	Kopiert Verweise auf alle Programmfäden die sich in derselben Gruppe, oder einer ihrer Untergruppen, wie der aktuelle Thread befinden in den Array *tarray*. Zuvor wird durch Aufruf der Methode `checkAccess` geprüft, ob er anfragende Thread zu dieser Operation berechtigt ist. Ist dies nicht der Fall, so wird ein `SecurityException`-Ausnahmeereignis erzeugt. siehe Java API Specification

`private void exit()`	Wird durch das Laufzeitsystem automatisch bei Terminierung eines Threads aufgerufen. Konzeptionell bildet diese Methode das Analogon des Destruktors für Klassen. Seit JDK 1.1 wird innerhalb dieser Methode dem durch den Thread ausgeführten Objekt explizit `null` zugewiesen um seine Entfernung durch den Garbage Collector sicherzustellen. (siehe Java Bug 4006245)
`public final String getName`	Liefert den Namen des Threads zurück. siehe Java API Specification Codebeispiel
`public final int intPriority()`	Liefert die Priorität eines Threads. siehe Java API Specification Codebeispiel
`public final ThreadGroup getThreadGroup()`	Liefert die Threadgruppe zurück, welcher der Thread zugeordnet ist. siehe Java API Specification Codebeispiel
`public static native boolean holdsLock (Object obj)`	Liefert *true* falls der Thread eine Sperre auf das Objekt *obj* hält. siehe Java API Specification
`private void init(ThreadGroup g, Runnable target, String name, long stackSize)`	Initialisiert einen Thread. Parameter: •`g` bezeichnet die Threadgruppe, der der neue Thread zugeordnet werden soll. •`target` bezeichnet die nebenläufig auszuführende Klasse. •`name` legt den Namen des Threads fest. •`stackSize` schlägt der virtuellen Maschine die für den Thread zu reservierende Stackgröße vor.
`public void interrupt()`	Unterbricht den laufenden Thread und setzt sein Unterbrechungsflag. Zunächst wird durch Aufruf der Methode `checkAccess` geprüft ob der Thread zur Unterbrechung anderen Programmfadens berechtigt ist. Ist dies nicht der Fall, so wird ein `SecurityException`-Ausnahmeereignis erzeugt. Fall der zu unterbrechende Thread sich im Zustand *blockiert* befindet, wird ein `InterruptedException`-Ausnahmeereignis erzeugt. Wurde der zu unterbrechende Thread durch eine E/A-Operation auf einem *unterbrechbaren Kanal* blockiert, so verliert er den Kanal und sein Unterbrechungsflag wird gesetzt. Zusätzlich empfängt er ein `ClosedByInterruptException`-Ausnahmeereignis. Sollte sich der zu unterbrechende Thread innerhalb eines `Selector`-Objekts blockiert finden, so kehrt er unverzüglich (evtl. auch mit leerer Rückgabe) zurück. (Dieses Verhalten ist Äquivalent zum Aufruf der Methode `wakeup` auf einem `Selector`-Objekt.) siehe Java API Specification Codebeispiel
`public static boolean interrupted()`	Liest das Unterbrechungsflag aus und setzt es zurück. siehe Java API Specification
`public final native boolean isAlive()`	Testet ob sich ein Thread im Zustand *alive* befindet, d.h. ob er gestartet und noch nicht beendet wurde. siehe Java API Specification
`public boolean isInterrupted()`	Liest das Unterbrechungsflag aus. Diese Methode setzt das Flag jedoch nach dem Lesevorgang -- im Gegensatz zu `interrupted` -- nicht zurück. siehe Java API Specification
`private native boolean isInterrupted (boolean ClearInterrupted)`	Liest das Unterbrechungsflag eines Threads aus und setzt es in Abhängigkeit von *ClearInterrupted* zurück. Diese Methode wird durch `interrupted` und `isInterrupted` intern aufgerufen.

`public final synchronized void join(long millis)`	Wartet höchstens *millis* Millisekunden auf die Beendigung eines anderen Threads. Ein Wert von 0 erlaubt unbegrenztes Warten. Wartezeiten <0 ziehen eine `IllegalArgumentException` nach sich. Während der Wartezeit wird der wartende Thread blockiert. Wird er während diesem Zeitraum unterbrochen, so wird ein `InterruptedException`-Ausnahmeereignis erzeugt. siehe Java API Specification
`public final synchronized void join(long millis, int nanos)`	Wartet die in Millisekunden (*millis*) und Nannosekungen (*nanos*) festgelegte Zeitspanne auf das Beenden eines Threads. De facto wird das *nanos*-Argument bei *millis*>0 für Werte kleiner 500000 ignoriert. Andernfalls (d.h. *millis*=0) wird genau eine Millisekunde gewartet. In beiden Fällen wird der Aufruf auf `join` abgebildet. siehe Java API Specification
`public final void join()`	Wartet beliebig lange auf das Beenden einen Threads. Der Aufruf ist äquivalent zur `join(0)`. siehe Java API Specification Codebeispiel
`private static synchronized int nextThreadNum()`	Vergibt automatisch die Threadnummer.
`private static native void registerNatives()`	Native Methode, welche direkt nach Erzeugung eines Thread-Objekts ausgeführt wird.
`public final void resume()`	Überführt einen zuvor durch `suspend` unterbrochenen Thread wieder in den Zustand *alive*. Diese Methode ist als *deprecated* gekennzeichnet und sollte daher nicht mehr verwendet werden, da sie kann im Zusammenhang mit Sperren zu unauflösbaren Verklemmungen führen kann. (Weitere Informationen dazu) siehe Java API Specification
`public void run()`	Wurde der Thread durch ein eigenständiges Objekt des Typs `Runnable` erzeugt, so kann er durch expliziten Aufruf dieser Methode gestartet werden. Konzeptionell bildetet die Methode `run` die main-Methode eines Threads. Üblicherweise überschreiben von `Thread` abgeleitete Klassen diese Methode. siehe Java API Specification
`public final void setName(String name)`	Setzt den Namen eines Threads. Die Berechtigung zur Namensänderung wird durch Aufruf der Methode `checkAccess` geprüft. Liegt sie nicht vor, so wird ein `SecurityException`-Ausnahmeereignis erzeugt. siehe Java API Specification
`public final void setPriority(int newPriority)`	Legt die Priorität eines Threads fest. siehe Java API Specification Codebeispiel
`public static native void sleep(long millis)` `throws InterruptedException`	Hält die Ausführung des aktuellen Threads für die durch `millis` spezifizierte Zeitspanne in Millisekunde an. Der Aufruf `sleep(0)` ist äquivalent zu `yield()`. Während der Suspendierung werden durch den Thread gehaltene Sperren (Monitore) nicht freigegeben. Wird der Thread in diesem Zeitraum durch einen anderen Programmfaden unterbrochen, so wird ein Ausnahmeereignis erzeugt. siehe Java API Specification Codebeispiel

`public static void sleep(long millis, int nanos) throws InterruptedException`	Hält die Ausführung des aktuellen Threads für die durch `millis` spezifizierte Zeitspanne in Millisekunde zuzüglich der in nanos angegebenen Zeitspanne in Nanosekunden an. Während der Suspendierung werden durch den Thread gehaltene Sperren (Monitore) nicht freigegeben. Wird der Thread in diesem Zeitraum durch einen anderen Programmfaden unterbrochen, so wird ein Ausnahmeereignis erzeugt. siehe Java API Specification
`public final void stop()`	Beendet einen in Ausführung befindlichen Thread unverzüglich unter Freigabe der durch ihn gehaltenen Sperren. Diese Methode ist durch SUN als *deprecated* gekennzeichnet und sollte nicht mehr verwendet werden. Sie gibt gesperrte Objekte auch dann frei, wenn sie sich dadurch in einem inkonsistenten Zustand befinden. (Weitere Informationen dazu) siehe Java API Specification
`public final synchronized void stop (Throwable obj)`	Beendet einen in Ausführung befindlichen Thread unverzüglich unter Freigabe der durch ihn gehaltenen Sperren. Auch diese Methode ist (genauso wie `stop`) als deprecated markiert und sollte nicht mehr verwendet werden. Zusätzlich zu den bei `stop` gegebenen Anmerkungen tritt bei ihrer Verwendung die Gefahr auf, daß durch *obj* ein Ausnahmeereignisobjekt im Stack plaziert wird, für dessen Verarbeitung der empfangende Thread nicht präpariert wurde. siehe Java API Specification
`public final void suspend()`	Unterbricht die Ausführung eines Threads. Diese Methode ist als *deprecated* gekennzeichnet und sollte daher nicht mehr verwendet werden, da sie kann im Zusammenhang mit Sperren zu unauflösbaren Verklemmungen führen kann. (Weitere Informationen dazu) siehe Java API Specification
`public String toString()`	Liefert eine Stringrepräsentation des Threads zurück. Die erzeugte Zeichenkettendarstellung ist der Form: **Thread[*Threadname*,*Priorität*,*Name der Threadgruppe*]**, wobei die Gruppenangabe fehlen kann. siehe Java API Specification
`public static native void yield()`	Veranlaßt den aktuellen Thread sein Zeitscheibe vorzeitig abzugeben, so daß ein anderer Programmfaden zur Ausführung gebracht werden kann. siehe Java API Specification Codebeispiel

Die Schnittstelle `Runnable`

Die Schnittstelle `Runnable` befindet sich im Paket `java.lang`.
Quellcode der Schnittstelle `Runnable`
Die Schnittstelle wird durch die Klasse `Thread` standardmäßig implementiert, weshalb zur nebenläufigen Ausführung einer beliebigen Klasse das Erben von `Thread` genügt.
In vielen praktischen Fällen ist dies jedoch nicht gewünscht, da hierdurch eine mit unter ungewollte Typisierung entsteht, oder nicht möglich, etwa weil bereits eine Superklasse existiert, in diesen Fällen wird auf die Implementierung der Schnittstelle `Runnable` zurückgegriffen.

Operation	Semantik
`public abstract void run()`	Diese Methode wird durch Aufruf der `start`-Methode auf einem `Thread`-Objekt (oder einer Subklasse davon) ausgeführt. Konzeptionell entspricht sie einer main-Methode für einen Thread. siehe Java API Specification Codebeispiel

Die Klasse ThreadGroup

Die Klasse `ThreadGroup` befindet sich im Paket `java.lang`.
Sie gestattet es Threads wahlfrei zu gruppieren und zu strukturieren.
Überdies erlaubt sie es threadspezifische Operationen gesammelt auf einer Gruppe von Threads auszuführen.
Quellcode der Klasse `ThreadGroup`

Attribut	Semantik
`boolean daemon`	`true` falls es sich bei der Threadgruppe um eine Daemonthreadgruppe handelt.
`ThreadGroup groups[]`	Liste mit Verweisen auf die Threadgruppen innerhalb einer Threadgruppe.
`int maxPriority`	Enthält die höchstmögliche Priorität eines Threads innerhalb der Threadgruppe. Standardmäßig wird die Variable während des Initialisierungsvorganges mit dem Wert der Konstanten `MAX_PRIORITY` belegt.
`String name`	Name der Threadgruppe.
`int ngroups`	Anzahl der Threadgruppen innerhalb einer Threadgruppe.
`int nthreads`	Anzahl der Threads einer Threadgruppe.
`ThreadGroup parent`	Verweis auf die Elternthreadgruppe, d.h. die Threadgruppe welche die aktuelle beinhaltet.
`Thread threads[]`	Liste mit Verweisen auf die Threads innerhalb der Threadgruppe.
`boolean vmAllowSuspension`	Boole'scher Wert, der angab (inzwischen wird er nicht mehr verwendet) ob im Falle knappen Speichers Threads angehalten werden dürfen.

Operation	Semantik
`private ThreadGroup()`	Erzeugt eine Threadgruppe. Dieser Konstruktor wird durch eine native C-Methode aufgerufen um die system-Threadgruppe zu erzeugen. Auch die Initialisierung der Variable `maxPriority` erfolgt in dieser Methode.
`public ThreadGroup(String name)`	Erzeugt eine neue Threadgruppe als Kindgruppe der Gruppe welcher der aktuell ausgeführte Thread zugeordnet ist. Durch Aufruf der Methode `checkAccess` auf der Threadgruppe des aktuellen Threads wird geprüft ob dieser zur Gruppenerzeugung berechtigt ist. Ist dies nicht der Fall, so wird ein `SecurityException`-Ausnahmeereignis erzeugt. siehe Java API Specification
`public ThreadGroup(ThreadGroup parent, String name)`	Erzeugt eine neue Threadgruppe unter dem Namen *name* und ordnet diese der Gruppe *parent* als Kindgruppe zu. Durch Aufruf der Methode `checkAccess` auf der Threadgruppe des aktuellen Threads wird geprüft ob dieser zur Gruppenerzeugung berechtigt ist. Ist dies nicht der Fall, so wird ein `SecurityException`-Ausnahmeereignis erzeugt. Existiert die Elternthreadgruppe nicht (oder die Referenz hat den Wert `null`), so wird ein `NullPointerException`-Ausnahmeereignis erzeugt. siehe Java API Specification
`public int activeCount()`	Liefert einen Schätzwert der aktuell aktiven Threads innerhalb einer Threadgruppe zurück. Der Zählvorgang beeinflußt die Lebenszyklen der Threads im System nicht, daher kann das Ergebnis von der Anzahl der dann tatsächlich existierenden Programmfäden abweichen. siehe Java API Specification
`public int activeGroupCount()`	Liefert einen Schätzwert der aktuell aktiven Threadgruppen innerhalb eines Threads. Der Zählvorgang beeinflußt die Lebenszyklen der Threadgruppen im System nicht, daher kann das Ergebnis von der Anzahl der dann tatsächlich existierenden Gruppen abweichen. siehe Java API Specification

`private final void add(ThreadGroup g)`	Fügt der aktuellen Threadgruppe die Threadgruppe *g* hinzu.
`void add(Thread t)`	Fügt den Thread *t* zur aktuellen Threadgruppe hinzu.
`public boolean allowThreadSuspension (boolean b)`	Durch die virtuelle Maschine benutzt um im Falle niedrigen verfügbaren Speichers Threads gruppenweise anzuhalten. Diese Methode ist als *deprecated* gekennzeichnet und sollte nicht mehr verwendet werden! siehe Java API Specification
`public final void checkAccess()`	Prüft ob der aktuell ausgeführte Thread die Berechtigung zur Veränderung der Threadgruppencharakteristika besitzt. Zur Ermittlung der Berechtigung wird die checkAccess-Methode des SecurityManagers aufgerufen. Liegt keine Berechtigung vor, so wird ein SecurityException-Ausnahmeereignis erzeugt. siehe Java API Specification
`public final void destroy()`	Zerstört die Threadgruppe einschließlich der darin enthaltenen Kindgruppen. Die Gruppen dürfen keine Threads mehr enthalten, andernfalls wird ein IllegalThreadStateException-Ausnahmeereignis erzeugt. Ist der ausführende Thread nicht zur Operation berechtigt, so wird ein SecurityException-Ausnahmeereignis erzeugt. siehe Java API Specification
`public int enumerate(Thread list[])`	Kopiert rekursiv alle aktiven Threads der aktuellen Gruppe und deren Kindgruppen in den übergebenen Array. Vor Ausführung der Kopieroperation wird die Berechtigung mittels des Aufrufs von checkAccess auf der Threadgruppe geprüft. Liegt keine hinreichende Berechtigung vor, so wird ein SecurityException-Ausnahmeereignis erzeugt. Zur Ermittlung der notwendigen Arraygröße sollte die Methode activeCount genutzt werden. Andernfalls werden diejenigen Threads, welche aufgrund der Großenbeschränkung nicht mehr in den Array aufgenommen werden können stillschweigend ignoriert. Der Aufruf entspricht enumerate(list, 0, true). siehe Java API Specification
`public int enumerate(Thread list[], boolean recurse)`	Kopiert alle aktiven Threads der aktuellen Gruppe in den übergebenen Array. Ist Parameter *recurse* mit `true` belegt, so wird dieser Vorgang rekursiv auch für die Kindthreadgruppen durchgeführt. Vor Ausführung der Kopieroperation wird die Berechtigung mittels des Aufrufs von checkAccess auf der Threadgruppe geprüft. Liegt keine hinreichende Berechtigung vor, so wird ein SecurityException-Ausnahmeereignis erzeugt. Zur Ermittlung der notwendigen Arraygröße sollte die Methode activeCount genutzt werden. Andernfalls werden diejenigen Threads, welche aufgrund der Großenbeschränkung nicht mehr in den Array aufgenommen werden können stillschweigend ignoriert. Der Aufruf entspricht enumerate(list, 0, recurse). siehe Java API Specification
`private int enumerate(Thread list[], int n, boolean recurse)`	Kopiert alle, bis auf *n* Threads der aktuellen Gruppe in den übergebenen Array. Ist Parameter *recurse* mit `true` belegt, so wird dieser Vorgang rekursiv auch für die Kindthreadgruppen durchgeführt. Vor Ausführung der Kopieroperation wird die Berechtigung mittels des Aufrufs von checkAccess auf der Threadgruppe geprüft. Liegt keine hinreichende Berechtigung vor, so wird ein SecurityException-Ausnahmeereignis erzeugt. Zur Ermittlung der notwendigen Arraygröße sollte die Methode activeCount genutzt werden.

`public int enumerate(ThreadGroup list[])`	Kopiert rekursiv alle aktiven Threadgruppen und deren Kindgruppen in den übergebenen Array. Vor Ausführung der Kopieroperation wird die Berechtigung mittels des Aufrufs von `checkAccess` auf der Threadgruppe geprüft. Liegt keine hinreichende Berechtigung vor, so wird ein `SecurityException`-Ausnahmeereignis erzeugt. Zur Ermittlung der notwendigen Arraygröße sollte die Methode `activeGroupCount` genutzt werden. Andernfalls werden diejenigen Threadgruppen, welche aufgrund der Großenbeschränkung nicht mehr in den Array aufgenommen werden können stillschweigend ignoriert. Der Aufruf entspricht `enumerate(list, 0, true)`. siehe Java API Specification
`public int enumerate(ThreadGroup list[], boolean recurse)`	Kopiert alle aktiven Threadgruppen in den übergebenen Array. Ist Parameter *recurse* mit `true` belegt, so wird dieser Vorgang rekursiv auch für die Kindthreadgruppen durchgeführt. Vor Ausführung der Kopieroperation wird die Berechtigung mittels des Aufrufs von `checkAccess` auf der Threadgruppe geprüft. Liegt keine hinreichende Berechtigung vor, so wird ein `SecurityException`-Ausnahmeereignis erzeugt. Zur Ermittlung der notwendigen Arraygröße sollte die Methode `activeGroupCount` genutzt werden. Andernfalls werden diejenigen Threadgruppen, welche aufgrund der Großenbeschränkung nicht mehr in den Array aufgenommen werden können stillschweigend ignoriert. Der Aufruf entspricht `enumerate(list, 0, recurse)`. siehe Java API Specification
`private int enumerate(ThreadGroup list[], int n, boolean recurse)`	Kopiert alle, bis auf *n* Threadgruppen in den übergebenen Array. Ist Parameter *recurse* mit `true` belegt, so wird dieser Vorgang rekursiv auch für die Kindthreadgruppen durchgeführt. Vor Ausführung der Kopieroperation wird die Berechtigung mittels des Aufrufs von `checkAccess` auf der Threadgruppe geprüft. Liegt keine hinreichende Berechtigung vor, so wird ein `SecurityException`-Ausnahmeereignis erzeugt. Zur Ermittlung der notwendigen Arraygröße sollte die Methode `activeGroupCount` genutzt werden.
`public final int getMaxPriority()`	Liefert die maximal für Threads der Gruppe festlegbare Priorität zurück. siehe Java API Specification
`public final String getName()`	Liefert den Namen der Threadgruppe. Zumindest die vordefinierten Threadgruppen `system` und `main` existieren immer im System, wobei `main` als Subgruppe von `system` organisiert ist. siehe Java API Specification
`public final ThreadGroup getParent()`	Liefert im Falle der Existenz einen Verweis auf die Elternthreadgruppe. Zumindest die vordefinierten Threadgruppen `system` und `main` existieren immer im System, wobei `main` als Subgruppe von `system` organisiert ist. siehe Java API Specification
`public final void interrupt()`	Unterbricht alle Threads der Gruppe. Diese Methode ist als *deprecated* gekennzeichnet und sollte nicht mehr verwendet werden! Da die Implementierung den Aufruf elementweise auf die `interrupt`-Methode der Klasse Thread abbildet, wirken die dort skizzierten Probleme hier fort. siehe Java API Specification
`public final boolean isDaemon()`	Testet ob es sich um eine Daemon-Threadgruppe handelt. Eine solche Gruppe wird automatisch nach Terminierung des letzten enthaltenen Threads oder der letzten enthaltenen Threadgruppe zerstört. siehe Java API Specification

`public synchronized boolean isDestroyed()`	Testet ob die Threadgruppe zerstört wurde. siehe Java API Specification
`public void list()`	Stellt Informationen über die aktuelle Threadgruppe an der Standardausgabe zur Verfügung. Diese Methode ist lediglich zur Fehlersuche sinnvoll einsetzbar. siehe Java API Specification
`public final boolean parentOf(ThreadGroup g)`	Testet ob die als *g* übergebene Threadgruppe gleich dem Threadgruppen-Objekt ist auf dem die Methode ausgeführt wird, oder eine der Vorfahren-Threadgruppen. siehe Java API Specification
`private void remove(ThreadGroup g)`	Entfernt die Threadgruppe *g* aus der aktuellen Threadgruppe.
`public final void resume()`	Nimmt die Verarbeitung für alle zuvor blockierten oder unterbrochenen Threads der Gruppe wieder auf. Diese Methode wird ausschließlich im Zusammenspiel mit `suspend` und `interrupt` verwendet, die beide als *deprecated* markiert sind und daher nicht mehr verwendet werden sollten. siehe Java API Specification
`public final void setDaemon(boolean daemon)`	Setzt den Daemonzustand einer Threadgruppe. Die Berechtigung zu dieser Operation wird durch Aufruf der `checkAccess`-Methode auf der zu modifizierenden Threadgruppe geprüft. Liegt diese nicht vor, so wird ein `SecurityException`-Ausnahmeereignis erzeugt. Wird diese Methode während der Ausführungszeit eines Threads aufgerufen, so wird einej `IllegalThreadStateException`-Ausnahme erzeugt. siehe Java API Specification Codebeispiel
`public final void setMaxPriority(int pri)`	Setzt die Prioritätsoberschranke einer Threadgruppe. Der Wer von *pri* muß sich zwischen `MIN_PRIORITY` und `MAX_PRIORITY` bewegen, andernfalls wird der Aufruf folgenlos ignoriert. Zusätzlich muß übergebene Wert unterhalb der Prioritätsschranke des Elternthreadgruppe liegen. Um die rekursive Anwendbarkeit dieser Festlegung sicherzustellen, wird die `system`-Threadgruppe mit der Priorität `MAX_PRIORITY` belegt. siehe Java API Specification
`public final void stop()`	Beendet alle Threads der Gruppe. Diese Methode ist als *deprecated* gekennzeichnet und sollte nicht mehr verwendet werden! Da die Implementierung den Aufruf elementweise auf die `stop`-Methode der Klasse `Thread` abbildet, wirken die dort skizzierten Probleme hier fort. siehe Java API Specification
`private boolean stopOrSuspend(boolean suspend)`	Hilfsmethode die durch `stop` und `suspend` verwendet wird.
`public final void suspend()`	Brockiert alle Threads der Gruppe. Diese Methode ist als *deprecated* gekennzeichnet und sollte nicht mehr verwendet werden! Da die Implementierung den Aufruf elementweise auf die `suspend`-Methode der Klasse `Thread` abbildet, wirken die dort skizzierten Probleme hier fort. siehe Java API Specification
`public String toString()`	Liefert eine Stringrepräsentation der Threadgruppe zurück. Die erzeugte Zeichenkettendarstellung ist der Form: ***Klassenname*[name=*Threadname*, maxpri=*Prioritätsschranke*]**, wobei der Klassenname üblicherweise (sofern keine eigene Ableitung von `ThreadGroup` vorgenommen wurde) *java.lang.ThreadGroup* lautet. siehe Java API Specification

`public void uncaughtException(Thread t, Throwable e)`	Wird durch die virtuelle Maschine aufgerufen wenn ein Thread der Gruppe aufgrund einer nicht abgefangenen Ausnahme angehalten wird. Existiert eine Elternthreadgrupe, so wird der Aufruf unverändert an dieselbe Methode dort weitergeleitet. Zusätzlich wird, durch `printStackTrace` der Stackinhalt auf die Standardfehlerausgabe geleitet falls es sich um ein `ThreadDeath`-Ausnahmeereignis handelt. siehe Java API Specification

Die Klasse `ThreadLocal`

Die Klasse `ThreadLocal` befindet sich im Paket `java.lang`.
Quellcode der Klasse `ThreadLocal`
Die Klasse schafft die Möglichkeit threadspezifische Variablen zu definieren, die außerhalb des Threads verwaltet werden.
Implementiert ist die Verwaltung durch eine `HashMap` pro Thread.

Operation	Semantik
`ThreadLocal()`	Erzeugt ein `ThreadLocal`-Objekt. siehe Java API Specification
`public Object get()`	Liefert die threadspezifische Variablenbelegung zurück. Der notwendige Wert innerhalb der `HashMap` wird beim ersten Aufruf durch den Thread transparent erzeugt. siehe Java API Specification
`protected Object initialValue()`	Liefert den threadspezifischen Vorgabewert der Variable zurück. Ist durch den Programmierer eine andere Initialisierung als `null` gewünscht, so muß dieses Verhalten durch Ableitung von der Klasse `ThreadLocal` selbst implementiert werden. Hierfür bietet sich die Verwendung anonymer innerer Klassen an. siehe Java API Specification
`public void set(Object value)`	Setzt den threadspezifischen Vorgabewert der Variable auf einen anderen Wert. siehe Java API Specification

Die Klasse `InheritableThreadLocal`

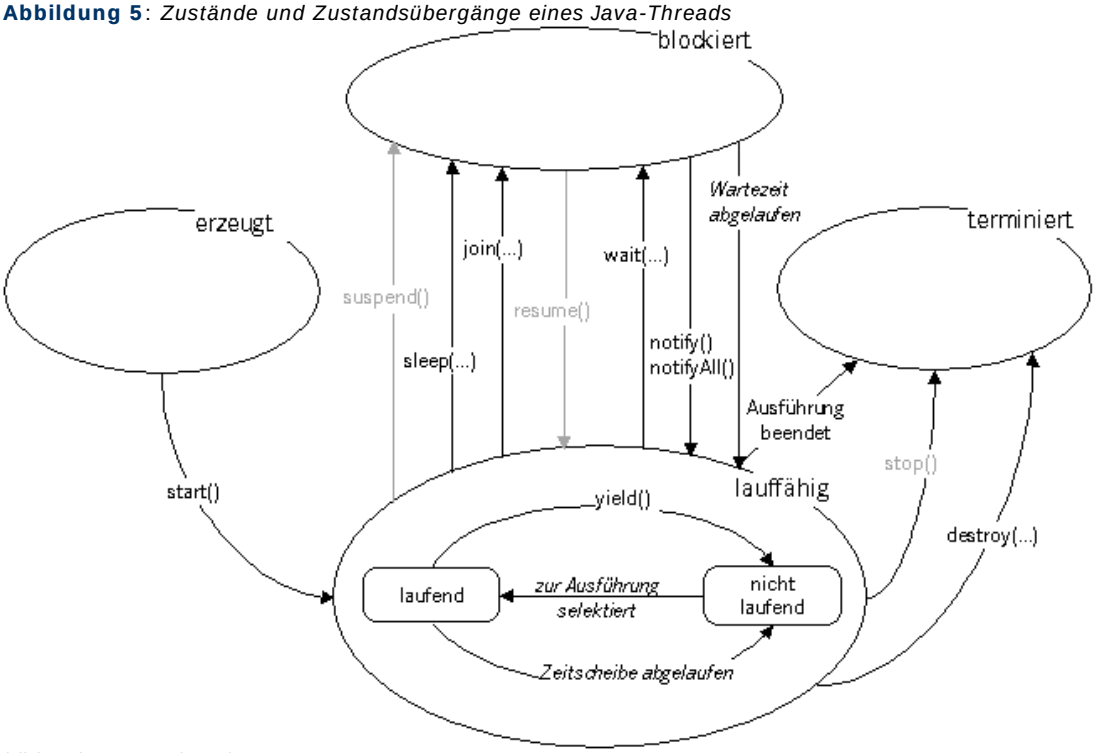
Die Klasse `InheritableThreadLocal` befindet sich im Paket `java.lang`.
Quellcode der Klasse `InheritableThreadLocal`
Diese Klasse erweitert die zuvor diskutierte Klasse `ThreadLocal` um die Möglichkeit den aktuellen Zustand der threadspezifischen Variablen an einen Kindthread zu übergeben.

Operation	Semantik
`public InheritableThreadLocal()`	Erzeugt ein `InheritableThreadLocal`-Objekt. siehe Java API Specification
`protected Object childValue(Object parentValue)`	Berechnet die initialen Belegungen der threadspezifischen Variablen. Dies geschieht durch Abfrage der threadspezifischen Variablen des Elternthreads zum Erzeugungszeitpunkt des Kindthreads. Die Methode wird durch den Elternthread vor Erzeugung des Kindthreads aufgerufen. siehe Java API Specification

Thread-Zustände

Zustandsübergänge eines Threads

Abbildung 5: *Zustände und Zustandsübergänge eines Java-Threads*



(click on image to enlarge!)

Synchronisationskonzepte und -primitive

Zur Motivation ...

Beispiel 3: *Nebenläufiges Inkrementieren eines Zählers in einer Datei*
Quellcode Ausgabe der Ausführung

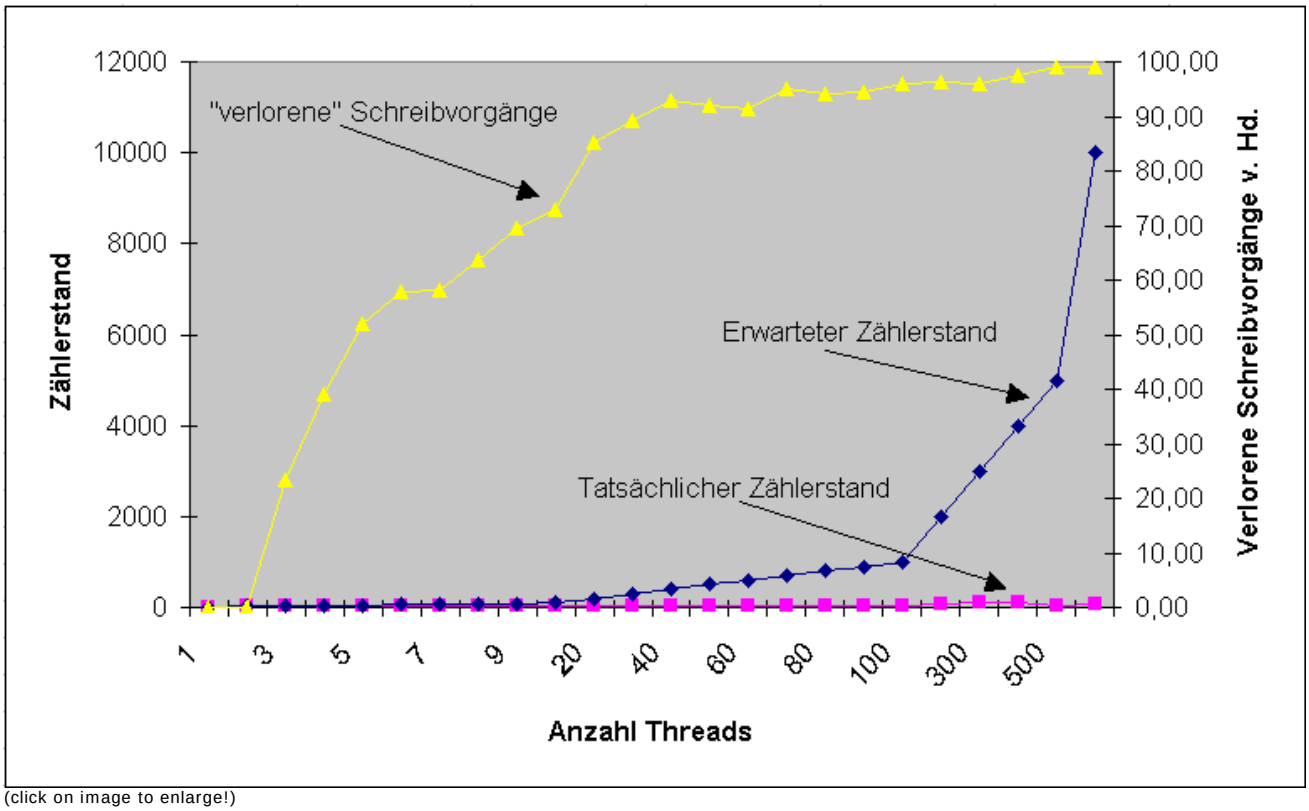
```
(1)import java.io.*;
(2)
(3)public class ConcurrentIncrement
(4){
(5)     public ConcurrentIncrement(int noThreads) throws Exception
(6)     {
(7)             Increment[] threadObjects = new Increment[noThreads];
(8)             Thread[] threads = new Thread[noThreads];
(9)
(10)            for (int i=0; i<noThreads; i++)
(11)            {
(12)                    threadObjects[i] = new Increment(this);
(13)            } //for
(14)
(15)            DataOutputStream dos = new DataOutputStream( new FileOutputStream( "testfile") );
(16)            dos.writeInt( 0 );
(17)
(18)            for (int i=0; i<noThreads; i++)
(19)            {
(20)                    threads[i] = new Thread( threadObjects[i] );
(21)                    threads[i].start();
(22)            } //for
(23)
(24)            for (int i=0; i<noThreads; i++)
(25)            {
(26)                    threads[i].join();
(27)            } //for
(28)
(29)            DataInputStream dis = new DataInputStream( new FileInputStream( "testfile") );
(30)            System.out.println("counter value: "+dis.readInt() );
```

```
(31)    } //constructor
(32)
(33)    public static void main(String argv[]) throws Exception
(34)    {
(35)            ConcurrentIncrement ci = new ConcurrentIncrement(Integer.parseInt(argv[0]));
(36)    } //main()
(37)    public void addOne()
(38)    {
(39)            try
(40)            {
(41)                    DataInputStream dis = new DataInputStream( new FileInputStream( "testfile") );
(42)                    int value = dis.readInt();
(43)
(44)                    System.out.println("thread named "+Thread.currentThread().getName()+" read:
"+value);
(45)                    value++;
(46)
(47)                    DataOutputStream dos = new DataOutputStream( new FileOutputStream
( "testfile") );
(48)                    dos.writeInt( value );
(49)                    System.out.println("thread named "+Thread.currentThread().getName()+" wrote:
"+value);
(50)
(51)                    dis.close();
(52)                    dos.close();
(53)            } //try
(54)            catch (IOException ioe)
(55)            {
(56)                    System.out.println("An IOException occured\n"+ioe.getMessage());
(57)                    ioe.printStackTrace();
(58)                    System.exit(1);
(59)            } //catch()
(60)    } //addOne()
(61)} //class ConcurrentIncrement
(62)// ------------------------------------------------------
(63)class Increment implements Runnable
(64){
(65)    ConcurrentIncrement ci;
(66)
(67)    public Increment(ConcurrentIncrement ci)
(68)    {
(69)            this.ci = ci;
(70)    } //constructor
(71)
(72)    public void run()
(73)    {
(74)            for (int i=0; i<10; i++)
(75)            {
(76)                    ci.addOne();
(77)            } //for
(78)    } //run()
(79)} //class Increment
```

Verhalten: Das Beispiel definiert eine nebenläufige Operation, durch welche ein int-Zählerstand aus einer Datei gelesen wird und um eins erhöht wieder in dieselbe Datei zurückgegschrieben wird. Die Anzahl der nebenläufig auszuführenden Threads kann über einen Komandozeilenparameter festgelegt werden.

Beobachtung: Entgegen der intuitiven Erwartung weißt der Zählerstand in der Datei nach erfolgreichem Ende der Ausführung aller erzeugten Threads nicht den vermuteten Wert von 10*Anzahl Threads auf.

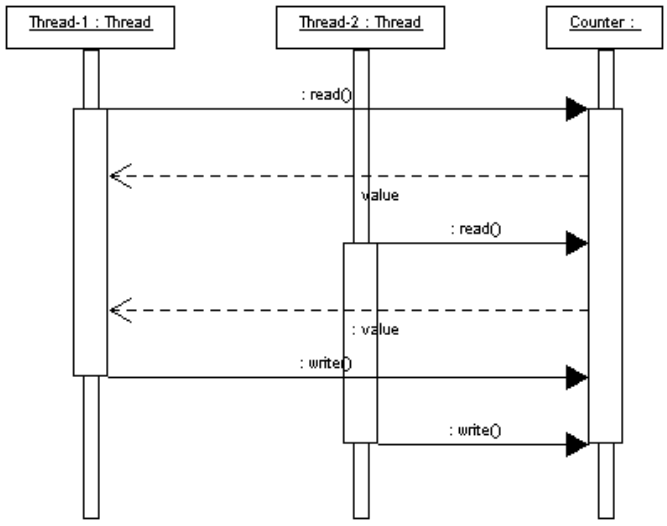
Abbildung 6: *Statistische Auswertung der unberücksichtigten Schreibvorgänge*



(click on image to enlarge!)

Analyse: Durch Unterbrechung eines Threads nach dem Einlesen des Wertes -- noch vor dem Rückschreiben es inkrementierten Zählerstandes -- kann ein anderer Thread zur Ausführung gelangen durch den nochmals der bereits gelesene Zählerstand aus der Datei verarbeitet wird. Später wird durch beide Threads derselbe erhöhte Variablenwert rückgeschrieben; ein Erhöhunsvorgang ist daher verloren, da er auf veralteten (da bereits (teil-)verarbeiteten) Daten beruht.
Die sich zeitlich überscheidenden Threads haben in diese Fall auf inkonsistente Datenstände Zugriff erlangt.
Dies muß jedoch in der Ausführung nicht immer der Fall sein. Das Ergebnis der gesamten Programmausführung hängt entscheidend von der Abarbeitungsreihenfolge der Einzelthreads ab. Es handelt sich dabei um eine sog. *race condition*.

Abbildung 7: *Lost-Update-Problem durch unsynchronisierte Zugriffe*



(click on image to enlarge!)

Definition 7: *Race Condition*
Fehlerquelle eines nebenläufigen Programms, die durch unsynchronisierte Abhängigkeiten zwischen Threads entsteht.
Abhängig von der tatsächliche Ausführungsreihenfolge der Threads auf dem Prozessor können Fehler in der gesamten Applikationsausführung entstehen.

Definition 8: *Atomar*
Eine Routine heißt atomar, wenn sie nicht in separate kleinere Einheiten unterteilt werden kann, die während ihrer Ausführung unterbrochen werden können.

Definition 9: *kritischer Abschnitt*
Eine Folge von Anweisungen, deren Ausführung nicht nebenläufig erfolgen kann oder sollte.

Definition 10: *Ressource*
Eine Ressource (in der Literatur auch als Betriebsmittel geführt) bezeichnet eine zur Programmausführung notwendige Einheit.
Hierbei kann es sich um CPU-Zeit, Variablen- oder Speicherzugriff aber auch um die Verfügbarkeit externer Peripheriegeräte wie Drucker handeln.
Knappe Ressourcen, um deren Verfübarkeit Konkurrenzsituationen entstehenden werden als kritische Ressourcen bezeichnet.

Eine erste (naive) Lösung

Idee: Konzentration der kritischen Anweisungen in eine einzige Programmzeile.
etwa: `dos.writeInt( dis.readInt()+1 )`.

Kritik: Zwar eine naheliegende Idee, aber keine Problemlösung.
Selbst diese vermeintlich atomare Anweisung zerfällt zunächst in eine Fülle von Java-Anweisungen, welche die API-Methoden `writeInt`(Quellcode) bzw. `readInt`(Quellcode) bilden.
Darüberhinaus sind selbst diese konstituierenden Java-Anweisungen der API keineswegs atomar, sondern werden ihrerseits die Bytecodeanweisungen der JVM gebildet (Bytecode: `readInt`, `writeInt`).
Schlußendlich werden die Bytecode-Instruktionen selbst nicht direkt durch die physische Hardware ausgeführt, sondern durch die virtuelle Maschine in native Anweisungen übersetzt.

Dieser (naive) Ansatz kann daher nicht zur Synchronisation eingesetzt werden!

Eine zweite Lösung

Idee: Sicherung des kritischen Bereichs durch Sperroperation.
(Vorgriff: Die Implementierung orientiert sich an der Idee der *Semaphore*.)

Beispiel 4: *Naive Sperroperation*
Quellcode Ausgabe der Ausführung ... auf einer Multiprozessormaschine

```
(1)import java.io.*;
(2)
(3)public class UsrLockConcurrentIncrement
(4){
(5)     boolean locked = false;
(6)
(7)     public UsrLockConcurrentIncrement(int noThreads) throws Exception
(8)     {
(9)             Increment[] threadObjects = new Increment[noThreads];
(10)            Thread[] threads = new Thread[noThreads];
(11)
(12)            for (int i=0; i<noThreads; i++)
(13)            {
(14)                    threadObjects[i] = new Increment(this);
(15)            } //for
(16)
(17)            DataOutputStream dos = new DataOutputStream( new FileOutputStream( "testfile") );
(18)            dos.writeInt( 0 );
(19)
(20)            for (int i=0; i<noThreads; i++)
(21)            {
(22)                    threads[i] = new Thread( threadObjects[i] );
(23)                    threads[i].start();
(24)            } //for
(25)
(26)            for (int i=0; i<noThreads; i++)
(27)            {
(28)                    threads[i].join();
(29)            } //for
(30)
(31)            DataInputStream dis = new DataInputStream( new FileInputStream( "testfile") );
(32)            System.out.println("counter value: "+dis.readInt() );
```

```
(33)    } //constructor
(34)
(35)    public boolean obtainLock()
(36)    {
(37)            if (locked)
(38)            {
(39)                    System.out.println("thread named "+Thread.currentThread().getName()+" cannot
obtain lock");
(40)                    return false;
(41)            } //if
(42)            else
(43)            {
(44)                    System.out.println("thread named "+Thread.currentThread().getName()+" obtained
lock");
(45)                    locked = true;
(46)                    return true;
(47)            } //else
(48)    } //obtainLock()
(49)
(50)    public void releaseLock()
(51)    {
(52)            System.out.println("thread named "+Thread.currentThread().getName()+" released
lock");
(53)            locked = false;
(54)    } //releaseLock()
(55)
(56)    public static void main(String argv[]) throws Exception
(57)    {
(58)            UsrLockConcurrentIncrement ci = new UsrLockConcurrentIncrement(Integer.parseInt(argv
[0]));
(59)    } //main()
(60)
(61)    public void addOne()
(62)    {
(63)            try
(64)            {
(65)                    while ( !this.obtainLock() )
(66)                    {
(67)                            System.out.println("thread named "+Thread.currentThread().getName()+"
waiting for lock");
(68)                    } //while
(69)
(70)                            DataInputStream dis = new DataInputStream( new FileInputStream
( "testfile") );
(71)                            int value = dis.readInt();
(72)
(73)                            System.out.println("thread named "+Thread.currentThread().getName()+"
read: "+value);
(74)                            value++;
(75)
(76)                            DataOutputStream dos = new DataOutputStream( new FileOutputStream
( "testfile") );
(77)                            dos.writeInt( value );
(78)                            System.out.println("thread named "+Thread.currentThread().getName()+"
wrote: "+value);
(79)
(80)                            dis.close();
(81)                            dos.close();
(82)                    this.releaseLock();
(83)            } //try
(84)            catch (IOException ioe)
(85)            {
(86)                    System.out.println("An IOException occured\n"+ioe.getMessage());
(87)                    ioe.printStackTrace();
(88)                    System.exit(1);
(89)            } //catch()
(90)    } //addOne()
(91)} //class UsrLockConcurrentIncrement
(92)// -------------------------------------------------------
```

```
(93)class Increment implements Runnable
(94){
(95)    UsrLockConcurrentIncrement ci;
(96)
(97)    public Increment(UsrLockConcurrentIncrement ci)
(98)    {
(99)            this.ci = ci;
(100)   } //constructor
(101)
(102)   public void run()
(103)   {
(104)           for (int i=0; i<10; i++)
(105)           {
(106)                   ci.addOne();
(107)           } //for
(108)   } //run()
(109)} //class Increment
```

Analyse: Trotz der guten Absicht bleibt das Problem bestehen.
Genaugenommen wurde es lediglich verlagert ...
Konnte zuvor der Thread zwischen Lese- und Schreibzugriff auf die Datei unterbrochen werden, so kann dies nun zwischen Anforderung und Erteilung der Sperre geschehen.
Überdies verbraucht die gewählte Implementierung ab Zeile 65 unnötig Rechenzeit durch den aktiven Wartevorgang (*busy waiting*).

Zur Implementierung ist eine unteilbare Hardwareoperation notwendig.
Kap. 7.2.1.2 des IA-32 Intel® Architecture Software Developer's Manuals empfiehlt zur Implementierung auf einem IA-32 Prozessor hierfür die Instruktion `CMPXCHG`.

Das Schlüsselwort synchronized

Eigenschaften:

- Das Schlüsselwort synchronized gestattet es kritische Abschnitte auf Hochsprachenebene deklarativ zu kennzeichnen.
- Es können sowohl Programmblöcke als auch komplette Methoden als synchronized gekennzeichnet werden.
- Die gesamte Synchronisationlogik wird durch die Java Virtual Machine bereitgestellt.
- Die Durchführung der Synchronisation hat negative Auswirkungen auf die Laufzeit.

Synchronisation vollständiger Methoden

Idee: Durch Anbringung des Schlüsselwortes in der Signatur einer Methode wird die virtuelle Maschine veranlaßt ein so gekennzeichnete Methode nicht nebenläufig auszuführen. Alle Aufrufe werden strikt serialisiert und ggf. verzögert bis die Methode zum alleinigen Zugriff zur Verfügung steht.

Anwendungsbeispiel: Die Veränderung im Quellcode betrifft ausschließlich Zeile 37. Dort wird die Methode addOne zusätzlich mit dem Schlüsselwort synchronized versehen.

Beispiel 5: *Synchronisation der nebenläufigen Zähler-Inkrementierung durch das Schlüsselwort synchronized*
Quellcode Ausgabe der Ausführung ... auf einer Multiprozessormaschine

```
(1)import java.io.*;
(2)
(3)public class SyncConcurrentIncrement
(4){
(5)     public SyncConcurrentIncrement(int noThreads) throws Exception
(6)     {
(7)             Increment[] threadObjects = new Increment[noThreads];
(8)             Thread[] threads = new Thread[noThreads];
(9)
(10)            for (int i=0; i<noThreads; i++)
(11)            {
(12)                    threadObjects[i] = new Increment(this);
(13)            } //for
(14)
```

```
(15)            DataOutputStream dos = new DataOutputStream( new FileOutputStream( "testfile") );
(16)            dos.writeInt( 0 );
(17)
(18)            for (int i=0; i<noThreads; i++)
(19)            {
(20)                    threads[i] = new Thread( threadObjects[i] );
(21)                    threads[i].start();
(22)            } //for
(23)
(24)            for (int i=0; i<noThreads; i++)
(25)            {
(26)                    threads[i].join();
(27)            } //for
(28)
(29)            DataInputStream dis = new DataInputStream( new FileInputStream( "testfile") );
(30)            System.out.println("counter value: "+dis.readInt() );
(31)    } //constructor
(32)
(33)    public static void main(String argv[]) throws Exception
(34)    {
(35)            SyncConcurrentIncrement sci = new SyncConcurrentIncrement(Integer.parseInt(argv[0]));
(36)    } //main()
(37)    public synchronized void addOne()
(38)    {
(39)            try
(40)            {
(41)                    DataInputStream dis = new DataInputStream( new FileInputStream( "testfile") );
(42)                    int value = dis.readInt();
(43)
(44)                    System.out.println("thread named "+Thread.currentThread().getName()+" read:
"+value);
(45)                    value++;
(46)
(47)                    DataOutputStream dos = new DataOutputStream( new FileOutputStream
( "testfile") );
(48)                    dos.writeInt( value );
(49)                    System.out.println("thread named "+Thread.currentThread().getName()+" wrote:
"+value);
(50)
(51)                    dis.close();
(52)                    dos.close();
(53)            } //try
(54)            catch (IOException ioe)
(55)            {
(56)                    System.out.println("An IOException occured\n"+ioe.getMessage());
(57)                    ioe.printStackTrace();
(58)                    System.exit(1);
(59)            } //catch()
(60)    } //addOne()
(61)} //class SyncConcurrentIncrement
(62)// ------------------------------------------------------
(63)class Increment implements Runnable
(64){
(65)    SyncConcurrentIncrement sci;
(66)
(67)    public Increment(SyncConcurrentIncrement sci)
(68)    {
(69)            this.sci = sci;
(70)    } //constructor
(71)
(72)    public void run()
(73)    {
(74)            for (int i=0; i<10; i++)
(75)            {
(76)                    sci.addOne();
(77)            } //for
(78)    } //run()
(79)} //class Increment
```

Weiteres Anwendungsbeispiel: Im vorherigen Beispiel war die Synchronisation auf eine Methode angewandt worden, die auf einem (existierenden) Objekt ausgeführt wurde.
Prinzipiell kann derselbe Mechanismus auch für statische Methoden zur Anwendung gebracht werden.
Hierbei wird die Sperrinformation nicht durch das Objekt, welches die zu sperrende Methode enthält verwaltet, sondern durch die beherbergende Klasse selbst.

Zum Beispiel: Der Code simuliert einen Hotelmanager bei der Arbeit. Eine über Kommandozeile übergebene Anzahl gleichzeitig eintreffender Gäste (Threads) möchte das letzte verfügbare Einzelzimmer (statische Methode `visit`) besuchen.

Beispiel 6: *Sperrung einer statischen Methode durch das Schlüsselwort synchronized*
Quellcode Ausgabe der Ausführung

```
(1)import java.util.*;
(2)
(3)public class RoomManager
(4){
(5)     static Random r;
(6)
(7)     public static void main(String argv[])
(8)     {
(9)             int noThreads = Integer.parseInt(argv[0]);
(10)
(11)            r = new Random();
(12)            Person[] threadObjects = new Person[noThreads];
(13)            Thread[] threads = new Thread[noThreads];
(14)
(15)            for (int i=0; i<noThreads; i++)
(16)            {
(17)                    threadObjects[i] = new Person();
(18)            } //for
(19)
(20)            for (int i=0; i<noThreads; i++)
(21)            {
(22)                    threads[i] = new Thread( threadObjects[i] );
(23)                    threads[i].start();
(24)            } //for
(25)    } //main()
(26)
(27)    public synchronized static void visit()
(28)    {
(29)            System.out.println("thread named "+Thread.currentThread().getName()+" entered");
(30)            try
(31)            {
(32)                    Thread.sleep(r.nextInt(1000));
(33)            } //try
(34)            catch (InterruptedException ie)
(35)            {
(36)                    System.out.println("An InterruptedException occured\n"+ie.getMessage());
(37)                    ie.printStackTrace();
(38)                    System.exit(1);
(39)            } //catch()
(40)            System.out.println("thread named "+Thread.currentThread().getName()+" left");
(41)    } //visit()
(42)} //class RoomManager
(43)// ----------------------------------------------------
(44)class Person implements Runnable
(45){
(46)    public void run()
(47)    {
(48)            for(int i=0; i<10; i++)
(49)            {
(50)                    RoomManager.visit();
(51)            } //for
(52)    } //run()
(53)} //class Person
(54)
(55)
```

(56)

Kritik am synchronized-Ansatz:

positiv

1. Einfache Durchführbarkeit
2. Deklarativ

negativ

1. Sperrgranularität fixiert auf Methodengröße
2. Auswirkungen auf Laufzeit
3. Inflexibilität hinsichtlich Anzahl der nebenläufigen Zugriffe (immer auf 1 fixiert)

Synchronization einzelner Anweisungen

Idee: Behebung des negativen Aspekts (1) unter Beibehaltung der Vorteile (1) und (2).

Lösung: Das bereits bekannte Schlüsselwort synchronized kann innerhalb von Methodenrümpfen zur Bildung synchronisierter Anweisungsblöcke eingesetzt werden.
Hierbei können nebenläufige Zugriffe auf den Block bezüglich eines beliebigen Objekts serialisiert werden.

Anwendungsbeispiel: In Zeile 46 wird durch das Schlüsselwort synchronized ein entsprechender Block geöffnet.
Aufgrund der Restriktion Zugriffe nur hinsichtlich von Objekten serialisieren zu können muß die bisherige Implementierung modifiziert werden.
So muß die als kritische Ressource anzusehende Variable value zwingend als Objekt repräsentiert werden.
Hierzu wird ab Zeile 89 die Klasse myInteger definiert, die als Prototyp einen int-Wert kapselt. Die Zählerzugriffe werden daher entsprechend umgesetzt.

Beispiel 7: *Synchronisation der nebenläufigen Zähler-Inkrementierung durch einen synchronized-Block*
Quellcode Ausgabe der Ausführung ... auf einer Multiprozessormaschine

```
(1)import java.io.*;
(2)
(3)public class SyncConcurrentIncrement2
(4){
(5)     myInteger value = new myInteger(0);
(6)
(7)     public SyncConcurrentIncrement2(int noThreads) throws Exception
(8)     {
(9)             Increment[] threadObjects = new Increment[noThreads];
(10)            Thread[] threads = new Thread[noThreads];
(11)
(12)            for (int i=0; i<noThreads; i++)
(13)            {
(14)                    threadObjects[i] = new Increment(this);
(15)            } //for
(16)
(17)            DataOutputStream dos = new DataOutputStream( new FileOutputStream( "testfile") );
(18)            dos.writeInt( 0 );
(19)
(20)            for (int i=0; i<noThreads; i++)
(21)            {
(22)                    threads[i] = new Thread( threadObjects[i] );
(23)                    threads[i].start();
(24)            } //for
(25)
(26)            for (int i=0; i<noThreads; i++)
(27)            {
(28)                    threads[i].join();
(29)            } //for
(30)
(31)            DataInputStream dis = new DataInputStream( new FileInputStream( "testfile") );
(32)            System.out.println("counter value: "+dis.readInt() );
(33)    } //constructor
(34)
```

```
(35)    public static void main(String argv[]) throws Exception
(36)    {
(37)            SyncConcurrentIncrement2 sci = new SyncConcurrentIncrement2(Integer.parseInt(argv
[0]));
(38)    } //main()
(39)    public void addOne()
(40)    {
(41)            try
(42)            {
(43)                    DataInputStream dis;
(44)                    DataOutputStream dos;
(45)
(46)                    synchronized(value)
(47)                    {
(48)                            dis = new DataInputStream( new FileInputStream( "testfile" ) );
(49)                            value.setValue( dis.readInt() );
(50)
(51)                            System.out.println("thread named "+Thread.currentThread().getName()+"
read: "+value.getValue());
(52)                            value.setValue( value.getValue()+1 );
(53)
(54)                            dos = new DataOutputStream( new FileOutputStream( "testfile" ) );
(55)                            dos.writeInt( value.getValue() );
(56)                            System.out.println("thread named "+Thread.currentThread().getName()+"
wrote: "+value.getValue());
(57)                    } //synchronized
(58)
(59)                    dis.close();
(60)                    dos.close();
(61)            } //try
(62)            catch (IOException ioe)
(63)            {
(64)                    System.out.println("An IOException occured\n"+ioe.getMessage());
(65)                    ioe.printStackTrace();
(66)                    System.exit(1);
(67)            } //catch()
(68)    } //addOne()
(69)} //class SyncConcurrentIncrement2
(70)// ------------------------------------------------------
(71)class Increment implements Runnable
(72){
(73)    SyncConcurrentIncrement2 sci;
(74)
(75)    public Increment(SyncConcurrentIncrement2 sci)
(76)    {
(77)            this.sci = sci;
(78)    } //constructor
(79)
(80)    public void run()
(81)    {
(82)            for (int i=0; i<10; i++)
(83)            {
(84)                    sci.addOne();
(85)            } //for
(86)    } //run()
(87)} //class Increment
(88)// ------------------------------------------------------
(89)class myInteger
(90){
(91)    private int i;
(92)    public myInteger(int i)
(93)    {
(94)            this.i = i;
(95)    } //constructor()
(96)
(97)    public void setValue(int i)
(98)    {
(99)            this.i = i;
(100)   } //setValue()
```

```
(101)
(102)   public int getValue()
(103)   {
(104)           return this.i;
(105)   } //getValue()
(106)} //class myInteger
```

Äquivalenz der beiden Varianten: Die Wirkung einer synchronized-Methode entspricht einem synchronized-Block, der den gesamten Methodenrumpf umfaßt und bezüglich dem this-Objekt zugriffsserialisiert wird.

Die Formulierungen:

```
public synchronized void foo()
    {
        //...
    } //foo()
```

und

```
public void foo()
    {
        synchronized(this)
        {
            //...
        } //synchronized
    } //foo()
```

sind daher gleichwertig.

Wirkung der Synchronisationsprimitive synchronized: Die Zugriffsserialisierung mit synchronized erfolgt immer auf Klassen- (für statische Methoden) bzw. auf Objektebene (für Instanzmethoden), trotz der methoden- oder blockspezifischen Schlüsselwortverwendung. Insbesondere bei Klassen mit umfangreichem Methodenangebot kann dies signifikant negative Auswirkungen auf die Laufzeit haben.

Anwendungsbeispiel: Das Beispiel definiert zwei statische Methoden (visitRoom in Zeile 27 und eat in Zeile 43).
Beide werden durch nebenläufig ausgeführte Threads unabhängig voneinander aufgerufen.

Beispiel 8: *Klasse mit zwei synchronisierten Methoden*
Quellcode Ausgabe der Ausführung

```
(1)import java.util.*;
(2)
(3)public class Hotel
(4){
(5)     static Random r;
(6)
(7)     public static void main(String argv[])
(8)     {
(9)             int noThreads = Integer.parseInt(argv[0]);
(10)
(11)            r = new Random();
(12)            Person[] threadObjects = new Person[noThreads];
(13)            Thread[] threads = new Thread[noThreads];
(14)
(15)            for (int i=0; i<noThreads; i++)
(16)            {
(17)                    threadObjects[i] = new Person();
(18)            } //for
(19)
(20)            for (int i=0; i<noThreads; i++)
(21)            {
(22)                    threads[i] = new Thread( threadObjects[i] );
(23)                    threads[i].start();
(24)            } //for
(25)    } //main()
```

```
(26)
(27)    public static synchronized void visitRoom()
(28)    {
(29)            System.out.println("thread named "+Thread.currentThread().getName()+" entered room");
(30)            try
(31)            {
(32)                    Thread.sleep(r.nextInt(1000));
(33)            } //try
(34)            catch (InterruptedException ie)
(35)            {
(36)                    System.out.println("An InterruptedException occured\n"+ie.getMessage());
(37)                    ie.printStackTrace();
(38)                    System.exit(1);
(39)            } //catch()
(40)            System.out.println("thread named "+Thread.currentThread().getName()+" left room");
(41)    } //visitRoom()
(42)
(43)    public synchronized static void eat()
(44)    {
(45)            System.out.println("thread named "+Thread.currentThread().getName()+" is eating");
(46)            try
(47)            {
(48)                    Thread.sleep(r.nextInt(1000));
(49)            } //try
(50)            catch (InterruptedException ie)
(51)            {
(52)                    System.out.println("An InterruptedException occured\n"+ie.getMessage());
(53)                    ie.printStackTrace();
(54)                    System.exit(1);
(55)            } //catch()
(56)            System.out.println("thread named "+Thread.currentThread().getName()+" meal ended");
(57)    } //eat()
(58)} //class Hotel
(59)// ------------------------------------------------------
(60)class Person implements Runnable
(61){
(62)    public void run()
(63)    {
(64)            for(int i=0; i<10; i++)
(65)            {
(66)                    Hotel.visitRoom();
(67)                    Hotel.eat();
(68)            } //for
(69)    } //run()
(70)} //class Person
```

Beobachtung: Trotz des unabhängigen nebenläufigen Aufrufs der beide Methoden werden zu keinem Beobachtungszeitpunkt die Rümpfe beider Methoden ausgeführt.

Folgerungen:

1. Als synchronized deklarierte Methoden werden immer klassenbasiert sychronisiert, auch wenn sie auf einem Objekt aufgerufen werden.
2. Die mit synchronized einhergehenden Sperren werden klassenbasiert (für statische Methoden) und objektbasiert (für Instanzmethoden) verwaltet.
   So können lediglich je eine statische und ein Instanzmethode einer Klasse nebenläufig ausgeführt werden.

Aussage (2) wird durch das folgende Beispiel illustriert.
Nur weil visitRoom und eat in unterschiedlichen Kontexten synchronisiert werden ist es möglich diese nebenläufig auszuführen.

Beispiel 9: *Klasse mit einer statischen und einer Instanzmethode, die beide als synchronized deklariert sind*
Quellcode Ausgabe der Ausführung ... auf einer Multiprozessormaschine

```
(1)import java.util.*;
(2)
(3)public class Hotel2
(4){
(5)     static Random r;
```

```
(6)
(7)     public static void main(String argv[])
(8)     {
(9)             int noThreads = Integer.parseInt(argv[0]);
(10)            Hotel2 h = new Hotel2();
(11)
(12)            r = new Random();
(13)            Person[] threadObjects = new Person[noThreads];
(14)            Thread[] threads = new Thread[noThreads];
(15)
(16)            for (int i=0; i<noThreads; i++)
(17)            {
(18)                    threadObjects[i] = new Person(h);
(19)            } //for
(20)
(21)            for (int i=0; i<noThreads; i++)
(22)            {
(23)                    threads[i] = new Thread( threadObjects[i] );
(24)                    threads[i].start();
(25)            } //for
(26)    } //main()
(27)
(28)    public synchronized static void visitRoom()
(29)    {
(30)            System.out.println("thread named "+Thread.currentThread().getName()+" entered room");
(31)            try
(32)            {
(33)                    Thread.sleep(r.nextInt(1000));
(34)            } //try
(35)            catch (InterruptedException ie)
(36)            {
(37)                    System.out.println("An InterruptedException occured\n"+ie.getMessage());
(38)                    ie.printStackTrace();
(39)                    System.exit(1);
(40)            } //catch()
(41)            System.out.println("thread named "+Thread.currentThread().getName()+" left room");
(42)    } //visitRoom()
(43)
(44)    public synchronized void eat()
(45)    {
(46)            System.out.println("thread named "+Thread.currentThread().getName()+" is eating");
(47)            try
(48)            {
(49)                    Thread.sleep(r.nextInt(1000));
(50)            } //try
(51)            catch (InterruptedException ie)
(52)            {
(53)                    System.out.println("An InterruptedException occured\n"+ie.getMessage());
(54)                    ie.printStackTrace();
(55)                    System.exit(1);
(56)            } //catch()
(57)            System.out.println("thread named "+Thread.currentThread().getName()+" meal ended");
(58)    } //visit()
(59)} //class Hotel2
(60)// ----------------------------------------------------
(61)class Person implements Runnable
(62){
(63)    Hotel2 h;
(64)
(65)    public Person(Hotel2 h)
(66)    {
(67)            this.h = h;
(68)    } //constructor
(69)
(70)    public void run()
(71)    {
(72)            for(int i=0; i<10; i++)
(73)            {
(74)                    Hotel2.visitRoom();
```

```
(75)                    h.eat();
(76)            } //for
(77)    } //run()
(78)} //class Person
```

Explizites Setzen einer klassenbasierten Sperre: durch die Angabe des Klassenobjektes (*Klassenname*.class) als Argument des synchronized-Blocks kann expliziter Zugriff auf die klassenbasierte Sperre erlangt werden.
Das nachfolgende Beispiel zeigt diesen Zugriff in Zeile Zeile 49. Die gewählte Konstellation gestattet die nebenläufige Ausführung der beiden Methoden visitRoom und eat, was an der Ausgabekonstellation thread named Thread-X entered room -- thread named Thread-1 is eating abzulesen ist.

Beispiel 10: *Explizites Setzen einer klassenbasierten Sperre*
Quellcode Ausgabe der Ausführung

```
(1)import java.util.*;
(2)
(3)public class Hotel4
(4){
(5)     static Random r;
(6)
(7)     public static void main(String argv[])
(8)     {
(9)             int noThreads = Integer.parseInt(argv[0]);
(10)            Hotel4 h = new Hotel4();
(11)
(12)            r = new Random();
(13)            Person[] threadObjects = new Person[noThreads];
(14)            Thread[] threads = new Thread[noThreads];
(15)
(16)            for (int i=0; i<noThreads; i++)
(17)            {
(18)                    threadObjects[i] = new Person(h);
(19)            } //for
(20)
(21)            for (int i=0; i<noThreads; i++)
(22)            {
(23)                    threads[i] = new Thread( threadObjects[i] );
(24)                    threads[i].start();
(25)            } //for
(26)    } //main()
(27)
(28)    public void visitRoom()
(29)    {
(30)            synchronized (this)
(31)            {
(32)            System.out.println("thread named "+Thread.currentThread().getName()+" entered room");
(33)            try
(34)            {
(35)                    Thread.sleep(r.nextInt(1000));
(36)            } //try
(37)            catch (InterruptedException ie)
(38)            {
(39)                    System.out.println("An InterruptedException occured\n"+ie.getMessage());
(40)                    ie.printStackTrace();
(41)                    System.exit(1);
(42)            } //catch()
(43)            System.out.println("thread named "+Thread.currentThread().getName()+" left room");
(44)    } //synchronized
(45)    } //visitRoom()
(46)
(47)    public void eat()
(48)    {
(49)            synchronized(Hotel4.class)
(50)            {
(51)                    System.out.println("thread named "+Thread.currentThread().getName()+" is
eating");
(52)            try
```

```
(53)            {
(54)                    Thread.sleep(r.nextInt(1000));
(55)            } //try
(56)            catch (InterruptedException ie)
(57)            {
(58)                    System.out.println("An InterruptedException occured\n"+ie.getMessage());
(59)                    ie.printStackTrace();
(60)                    System.exit(1);
(61)            } //catch()
(62)            System.out.println("thread named "+Thread.currentThread().getName()+" meal ended");
(63)    } //synchonized
(64)    } //visit()
(65)} //class Hotel4
(66)// ------------------------------------------------------
(67)class Person implements Runnable
(68){
(69)    Hotel4 h;
(70)    public Person(Hotel4 h)
(71)    {
(72)            this.h = h;
(73)    } //constructor
(74)    public void run()
(75)    {
(76)            for(int i=0; i<10; i++)
(77)            {
(78)                    h.visitRoom();
(79)                    h.eat();
(80)            } //for
(81)    } //run()
(82)} //class Person
```

Folge in der Praxis: Oftmals wird entweder die Klassensperre zur einfachen Gewinnung einer zweiten Sperre für ein Objekt „mißbraucht“ oder künstliche `Serviceobjekte` erzeugt, die später als Argumente für `synchronized`-Blöcke dienen, nur um Zugriff auf ihre Klassensperren zu erlangen.

Anwndungsbeispiel aus der Java-API:

1. Seit JDK v1.2 steht die Klasse `Vector` (Quellcode ansehen)(JavaDoc) in einer hinsichtlich nebenläufiger Zugriffe abgesicherten Implementierung zur Verfügung.
2. Für die Klasse `HashMap` wird hingegen, wie für alle Klassen des *Collection Frameworks*, die Zugriffsserialisierung dem Anwender überlassen (Quellcode ansehen)(JavaDoc).

Zusammenfassung: Die verschiedenen Ausprägungen des Schlüsselwortes `synchronized` können eingesetzt werden um wirkungsvoll die gleichzeitige Abarbeitung von Anweisungsfolgen durch mehr als genau einen Thread zu verhindern.
Auf diesem Wege läßt sich die konsistente Datenbereitstellung auf dem Wege strikter Serialisierung der Aufrufer eines kritischen Abschnitts realisieren.
Die Synchronisation ist dabei als *wechselseitiger Ausschluß* realisiert.

Definition 11: *Wechselseitiger Ausschluß*
Wechselseitiger Ausschluß stellt sicher, daß sich zu jedem Zeitpunkt höchstens eine Programmeinheit (in unserem Falle: Thread) in einem kritischen Abschnitt befindet.
Hierfür wird der Abschnitt durch einen Sperrmechanismus gesichert, der überwunden werden muß (d.h. die Sperre wird gesetzt) bevor die Ausführung der kritischen Anweisungen gestattet wird.
Nach dem Ausführungsende (d.h. Verlassen des kritischen Abschnitts) wird die Sperre wieder freigegeben.
Während der Ausführungszeit erfolgende weitere Zugriffsversuche werden an der Sperre blockiert und verzögert bis die in Ausführung befindliche Programmeinheit den kritischen Abschnitt erlassen hat.
Durch den wechselseitigen Ausschluß der nebenläufigen Ausführung von Programmteilen entsteht daher die Illusion einer atomaren Anweisung.

Das Methodenpaar `wait` und `notify`

Der Einsatz des Schlüsselwortes `synchronized` bietet zwar einen leistungsfähigen Mechanismus zur Realisierung des wechselseitigen Ausschlusses an kritischen Abschnitten an. Jedoch treten bei der

Realisierung praktischer Probleme zwei entscheidende Einschränkungen zu Tage.

1. Der wechselseitige Ausschluß läßt den Eintritt nur genau eines Threads in einen kritischen Bereich zu.
2. Die Realisierung der Warteschlangenverwaltung muß durch den Programmierer erfolgen.
3. Der Einsatz von synchronized bedingt häufig aktives Warten und damit keinen effizienten Umgang mit Ressource CPU.

Anschauungsbeispiel: Das Hotel der vorangegangenen Beispiele wird auf reale Gegebenheiten adaptiert. Nun sollten zehn gleichzeitig bewohnbare Zimmer sowie ein Speisesaal mit 20 Plätzen zur Verfügung stehen.
Gäste denen kein Zimmer oder Sitzplatz zur Verfügung steht, warten geduldig bis dies der Fall ist.

Mit den bishererfügbaren Synchronisationsprimitiven ist dieses Ziel kaum sinnvoll zu erreichen ...

- Einführung einer zusätzlichen Methode zur Anfrage nach knappen Ressourcen.
- Die Implementierung der Anfrage nach den knappen Ressourcen ist strukturell identisch.
- Mehrere synchronized-Routinen behindern sich gegenseitig.
- Die Umsetzung der Warteschlange ist aufwendig.
- Mehrere kritische Abschnitte existieren verteilt im Programm.

Die Primitive *Semaphore*

Das bekannte Konzept der Semaphore räumt mit zwei Restriktionen des wechselseitigen Ausschlusses durch synchronized aus:

1. Der Limitierung der Sperren auf jeweils eine Klassen- und eine Objektsperre.
2. Der Limitierung der gleichzeitigen Zugriffe auf genau einen, sofern dies gewünscht ist.

Definition 12: *Semaphor*
Das Semaphor (Wortbedeutung: Singalmast, optischer Telegraph) ist ein Variablentyp zur sicheren Zählung von Wartesignalen.
Ein Semaphor verfügt über die beiden atomaren Operationen *p* (nach dem holländischen *passeeren*) und *v* (*vrijgeven*) zur Belegung einer durch durch das Semaphor verwalteten Ressource bzw. deren Freigabe.
Zur korrekten funktionsfähigen Implementierung eines Semaphor ist Unterstützung durch eine unteilbare Hardwareoperation notwendig.

Implementierung durch eine eigenständige Java-Klasse:

Beispiel 11: *Eine Semaphore*
Quellcode

```
(1)public class Semaphore
(2){
(3)     int s;
(4)
(5)     public Semaphore(int s)
(6)     {
(7)             assert(s > 0);
(8)             this.s = s;
(9)     } //constructor
(10)
(11)    public synchronized void p()
(12)    {
(13)            while (s == 0)
(14)            {
(15)                    try
(16)                    {
(17)                            wait();
(18)                    } //try
(19)                    catch (InterruptedException ie)
(20)                    {
(21)                    System.out.println("An InterruptedException caught\n"+ie.getMessage());
(22)                    ie.printStackTrace();
(23)                    } //catch
(24)            } //while
(25)            s--;
(26)    } //p()
(27)
```

```
(28)    public synchronized void v()
(29)    {
(30)            s++;
(31)            notify();
(32)    } //v()
(33)} //class Semaphore
```

Eine einfache Erweiterung des klassischen Semaphormechanismus stellt der Übergang von der skalaren Sperrvariablen s zu einem strukturierten Sperrobjekt dar:

Beispiel 12: *Eine Semaphorenumsetzung, die ein Feld von Sperrvariablen kontrolliert*
Quellcode

```
(1)public class SemaphoreGroup
(2){
(3)     int[]   values;
(4)
(5)     public SemaphoreGroup(int noElements)
(6)     {
(7)             values = new int[noElements];
(8)     } //constructor
(9)
(10)    public synchronized void changeValues(int[] deltas)
(11)    {
(12)            while (!canChange(deltas))
(13)            {
(14)                    try
(15)                    {
(16)                            wait();
(17)                    } //try
(18)                    catch (InterruptedException ie)
(19)                    {
(20)                            System.out.println("An InterruptedException caught\n"+ie.getMessage());
(21)                            ie.printStackTrace();
(22)                            System.exit(1);
(23)                    } //catch()
(24)            } //while
(25)            doChange(deltas);
(26)            notifyAll();
(27)    } //changeValues()
(28)
(29)    private boolean canChange(int[] deltas)
(30)    {
(31)            for(int i=0; i<values.length; i++)
(32)            {
(33)                    if(values[i] + deltas[i] < 0)
(34)                            return false;
(35)            } //for
(36)            return true;
(37)    } //canChange()
(38)
(39)    private void doChange(int[] deltas)
(40)    {
(41)            for (int i=0; i<values.length; i++)
(42)            {
(43)                    values[i] += deltas[i];
(44)            } //for
(45)    } //doChange()
(46)
(47)    public int getNumberOfMembers()
(48)    {
(49)            return values.length;
(50)    } //getNumberOfMembers()
(51)} //class SemaphoreGroup
```

Das Beispiel verwaltet den wechselseitigen Ausschluß beim Zugriff auf ein int-Feld. Die Methode changeValues vereinigt die Eigenschaften der klassischen p-Operation und v-Operation in sich. Hierdurch wird jedoch die korrekte Behandlung der Sperrvariablenstände zurück zum

Anwendungsprogrammierer verlagert!

Anwendungsbeispiel:

Beispiel 13: *Synchronisation unter Einsatz einer Semaphore*
Quellcode Ausgabe der Ausführung ... auf einer Multiprozessormaschine

```
(1)import java.util.*;
(2)
(3)public class Hotel5
(4){
(5)     static Random r;
(6)     static Semaphore roomsSem;
(7)     static Semaphore mealsSem;
(8)
(9)     public static void main(String argv[])
(10)    {
(11)            int noThreads = Integer.parseInt(argv[0]);
(12)            Hotel5 h = new Hotel5();
(13)            r = new Random();
(14)            roomsSem = new Semaphore( 10 );
(15)            mealsSem = new Semaphore( 20 );
(16)
(17)            Person[] threadObjects = new Person[noThreads];
(18)            Thread[] threads = new Thread[noThreads];
(19)
(20)            for (int i=0; i<noThreads; i++)
(21)            {
(22)                    threadObjects[i] = new Person(h);
(23)            } //for
(24)
(25)            for (int i=0; i<noThreads; i++)
(26)            {
(27)                    threads[i] = new Thread( threadObjects[i] );
(28)                    threads[i].start();
(29)            } //for
(30)    } //main()
(31)
(32)    public void visitRoom()
(33)    {
(34)            System.out.println("thread named "+Thread.currentThread().getName()+" entered room");
(35)            try
(36)            {
(37)                    Thread.sleep(r.nextInt(1000));
(38)            } //try
(39)            catch (InterruptedException ie)
(40)            {
(41)                    System.out.println("An InterruptedException caught\n"+ie.getMessage());
(42)                    ie.printStackTrace();
(43)                    System.exit(1);
(44)            } //catch()
(45)            System.out.println("thread named "+Thread.currentThread().getName()+" left room");
(46)    } //visitRoom()
(47)
(48)    public void eat()
(49)    {
(50)            System.out.println("thread named "+Thread.currentThread().getName()+" is eating");
(51)            try
(52)            {
(53)                    Thread.sleep(r.nextInt(1000));
(54)            } //try
(55)            catch (InterruptedException ie)
(56)            {
(57)                    System.out.println("An InterruptedException caught\n"+ie.getMessage());
(58)                    ie.printStackTrace();
(59)                    System.exit(1);
(60)            } //catch()
(61)            System.out.println("thread named "+Thread.currentThread().getName()+" finished meal");
(62)    } //visitRoom()
(63)
```

```
(64)} //class Hotel5
(65)// ------------------------------------------------------
(66)class Person implements Runnable
(67){
(68)    Hotel5 h;
(69)    public Person(Hotel5 h)
(70)    {
(71)            this.h = h;
(72)    } //constructor
(73)    public void run()
(74)    {
(75)            h.roomsSem.p();
(76)            h.visitRoom();
(77)            h.roomsSem.v();
(78)
(79)            h.mealsSem.p();
(80)            h.eat();
(81)            h.mealsSem.v();
(82)
(83)    } //run()
(84)} //class Person
```

Im Beispiel werden zur Synchronisation die bekannten Primitive verwendet. Zur Realisierung der Warteschlangen werden zwei neue Methoden eingesetzt `wait()` und `notify`.

Die Methode `wait()` veranlaßt einen Thread auf ein Signal einer anderen Ausführungseinheit zu warten.
Signalisiert ein Thread durch `notify()` das Eintreten eines gewissen Zustandes, so wird der durch Aufruf von `wait` blockierte Thread geweckt und fährt in seiner Berechnung fort.
`notify` stellt somit eine einfache Möglichkeit der Inter-Threadkommunikation dar.

`wait` und `notify` sind selbst kritische Operationen und müssen daher in `synchronized`-Abschnitte oder -Methoden eingebettet werde.

Operation	**Semantik**
`public notify()`	Weckt genau einen durch `wait(...)` blockierten Thread.
`public notifyAll()`	Weckt alle durch `wait(...)`blockierten Threads.
`public wait()`	Wartet auf den Aufruf von `notify` durch einen anderen Thread.
`public wait(long l)`	Wartet höchstes die durch `l` vorgegebene Zeitspanne in Millisekunden auf den Aufruf von `notify` durch einen anderen Thread.
`public wait(long l, int n )`	Wartet höchstens die durch `l` vorgegebene Zeitspanne in Millisekunden zuzüglich des durch `n` spezifizierten Zeitraumes in Nanosekunden auf den Aufruf von `notify` durch einen anderen Thread.

Es ist leicht einzusehen, daß der Zugriff auf die Sperrvariable `s` in den Zeilen 13, 25 und 25 selbst einen kritischen Abschnitt darstellt und daher synchronisiert werden muß.
Der Rückgriff auf das bekannte Schlüsselwort `synchronized` löst das sich ergebende Synchronisationsproblem jedoch strenggenommen nicht, sondern verlagert es in die Hochsprachenebene und delegiert somit die korrekte Durchführung der Synchronisation an Übersetzer und Laufzeitsystem.

Probleme beim Einsatz von `wait` und `notify`: Der Aufruf des Methodenpaars `wait`-`notify` ist selbst laufzeitkritisch hinsichtlich der Reihenfolge der Aufrufe. In nebenläufigen Ausführungsumgebungen können Bedingungen eintreten, in denen ein Thread sein `notify` *vor* dem (eigentlich) zugehörigen `wait` des zu verzögernden Threads aufruft.

Beispiel 14: *Probleme beim Einsatz von Wait und Notify*
Quellcode

```
(1)public class MissedNotify {
(2)     public static void main(String argv[]) {
(3)             LockObj lo = new LockObj();
(4)             Thread ts = new Thread(new Sleeper("sleep",lo,Integer.parseInt(argv[0])));
(5)             ts.start();
```

```
(6)             new Thread(new Sleeper("wakeUp",lo,Integer.parseInt(argv[1]))).start();
(7)
(8)             try {
(9)                     Thread.sleep(2500);
(10)            }  catch (InterruptedException ie) {
(11)                    System.out.println("An InterruptedException caught\n"+ie.getMessage());
(12)                    ie.printStackTrace();
(13)                    System.exit(1);
(14)            } //catch()
(15)
(16)            ts.interrupt();
(17)    } //main()
(18)} //class MissedNotify
(19)
(20)class Sleeper implements Runnable {
(21)    LockObj lo;
(22)    int time;
(23)
(24)    String command;
(25)    public Sleeper(String command, LockObj lo, int time) {
(26)            this.command = command;
(27)            this.lo = lo;
(28)            this.time = time;
(29)    } //constructor
(30)
(31)    public void run() {
(32)            if( command.compareTo("sleep") == 0 )
(33)                    goToSleep();
(34)            else
(35)                    wakeUp();
(36)    } //run()
(37)
(38)    public  void goToSleep() {
(39)            try {
(40)                    Thread.sleep(time);
(41)            } catch (InterruptedException ie) {
(42)                    System.out.println("An InterruptedException caught\n"+ie.getMessage());
(43)                    ie.printStackTrace();
(44)                    System.exit(1);
(45)            } //catch()
(46)
(47)            synchronized(lo) {
(48)                    System.out.println("sleeping 'til notification ...");
(49)                    try {
(50)                            lo.wait();
(51)                    } catch (InterruptedException ie) {
(52)                            System.out.println("An InterruptedException caught\n"+ie.getMessage());
(53)                            ie.printStackTrace();
(54)                            System.exit(1);
(55)                    } //catch()
(56)                    System.out.println("notified and awaked");
(57)            } //synchronize
(58)    } //goToSleep()
(59)
(60)    public  void wakeUp() {
(61)            try {
(62)                    Thread.sleep(time);
(63)            } catch (InterruptedException ie) {
(64)                    System.out.println("An InterruptedException caught\n"+ie.getMessage());
(65)                    ie.printStackTrace();
(66)                    System.exit(1);
(67)            } //catch()
(68)
(69)            synchronized(lo) {
(70)                    System.out.println("Trying to awake ...");
(71)                    lo.notify();
(72)            } //synchronize
(73)    } //wakeUp()
(74)} //class Sleeper
```

```
(75)
(76)class LockObj
(77){ } //class LockObj
```

Der Aufruf `java MissedNotify 0 50` liefert:

```
sleeping 'til notification ...
Trying to awake ...
notified and awaked
```

`java MissedNotify 100 50` hingegen sorgt dafür, daß `notify` *vor* `wait` erreicht wird:

```
Trying to awake ...
sleeping 'til notification ...
An InterruptedException caught
null
java.lang.InterruptedException
        at java.lang.Object.wait(Native Method)
        at java.lang.Object.wait(Object.java:426)
        at Sleeper.goToSleep(MissedNotify.java:64)
        at Sleeper.run(MissedNotify.java:41)
        at java.lang.Thread.run(Thread.java:536)
```

Als Lösung bietet es sich an, den Aufruf von `wait` in eine `while`-Schleife zu kleiden, die prüft, ob die Bedingung für den Eintritt in den Wartezustand vorliegt.
Im Beispiel ist dies durch die Variable `sleeping` in `LockObj` umgesetzt.
Der bedingte Aufruf von `while` findet sich ab Zeile 64.

Beispiel 15: *Lösung des Problems beim Einsatz von Wait und Notify*
Quellcode

```
(1)public class MissedNotify2
(2){
(3)     public static void main(String argv[])
(4)     {
(5)             LockObj lo = new LockObj();
(6)             Thread ts = new Thread(new Sleeper("sleep",lo,Integer.parseInt(argv[0])));
(7)             ts.start();
(8)             new Thread(new Sleeper("wakeUp",lo,Integer.parseInt(argv[1]))).start();
(9)
(10)            try
(11)            {
(12)                    Thread.sleep(2500);
(13)            } //try
(14)            catch (InterruptedException ie)
(15)            {
(16)                    System.out.println("An InterruptedException caught\n"+ie.getMessage());
(17)                    ie.printStackTrace();
(18)                    System.exit(1);
(19)            } //catch()
(20)
(21)            ts.interrupt();
(22)    } //main()
(23)} //class MissedNotify2
(24)
(25)class Sleeper implements Runnable
(26){
(27)    LockObj lo;
(28)    int time;
(29)
(30)    String command;
(31)    public Sleeper(String command, LockObj lo, int time)
(32)    {
(33)            this.command = command;
(34)            this.lo = lo;
(35)            this.time = time;
(36)    } //constructor
(37)
(38)    public void run()
```

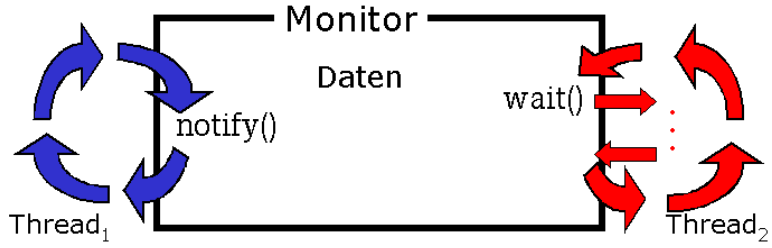
```
(39)    {
(40)            if( command.compareTo("sleep") == 0 )
(41)                    goToSleep();
(42)            else
(43)                    wakeUp();
(44)    } //run()
(45)
(46)    public  void goToSleep()
(47)    {
(48)            try
(49)            {
(50)                    Thread.sleep(time);
(51)            } //try
(52)            catch (InterruptedException ie)
(53)            {
(54)                    System.out.println("An InterruptedException caught\n"+ie.getMessage());
(55)                    ie.printStackTrace();
(56)                    System.exit(1);
(57)            } //catch()
(58)
(59)            synchronized(lo)
(60)            {
(61)                    System.out.println("sleeping 'til notification ..."+lo.sleeping);
(62)                    try
(63)                    {
(64)                            while (lo.sleeping == true)
(65)                            {
(66)                                    lo.sleeping = true;
(67)                                    lo.wait();
(68)                            }
(69)                    } //try
(70)                    catch (InterruptedException ie)
(71)                    {
(72)                            System.out.println("An InterruptedException caught\n"+ie.getMessage());
(73)                            ie.printStackTrace();
(74)                            System.exit(1);
(75)                    } //catch()
(76)                    System.out.println("notified and awaked");
(77)            } //synchronize
(78)    } //goToSleep()
(79)
(80)    public  void wakeUp()
(81)    {
(82)            synchronized(lo)
(83)            {
(84)                    try
(85)                    {
(86)                            Thread.sleep(time);
(87)                    } //try
(88)                    catch (InterruptedException ie)
(89)                    {
(90)                            System.out.println("An InterruptedException caught\n"+ie.getMessage());
(91)                            ie.printStackTrace();
(92)                            System.exit(1);
(93)                    } //catch()
(94)
(95)                    System.out.println("Trying to awake ...");
(96)                    lo.sleeping = false;
(97)                    lo.notify();
(98)            } //synchronize
(99)    } //wakeUp()
(100)} //class Sleeper
(101)
(102)class LockObj
(103){
(104)   public boolean sleeping;
(105)} //class LockObj
```

Monitore

Innerhalb der Java-Laufzeitumgebung werden *Monitore* als Hochsprachensynchronisationsprimitive eingesetzt. (siehe C. A. R. Hoare: *Monitors: An Operating System Structuring Concept*, Communications of the ACM, Vol. 17, No. 10, Oct. 1974, pp. 549--557).
Diese werden durch die beiden Bytecodeinstruktionen `monitorenter` und `monitorexit` direkt von der virtuellen Maschine unterstützt.

Abbildung 8 skizziert das Verhalten der virtuellen Maschine bei einer monitorbasierten Synchronisation mittels `wait` und `notify`.

Abbildung 8: *Verhalten eines Monitors zur Synchronisation*



(click on image to enlarge!)

Aus dieser Mimik erklärt sich auch die Verwendung von `while` statt `if` in Zeile 13 der Semaphorenimplementierung, da andernfalls die Bedingung beim Wiederbetreten des Monitors nach `wait()` nicht erfüllt sein könnte.
Zusätzlich läßt die Abbildung deutlich werden, weshalb es während des Wartens innerhalb von `wait()` (das sich seinerseits in einem `synchronized`-Block befindet, nicht zu Verklemmungen kommt: `wait` gibt intern währen des Wartevorganges die gehaltene Sperre frei, so daß ein anderer Programmfaden in einen synchronisierten Abschnitt eintreten und `notify` aufrufen kann.

Anmerkung:

- `notifyAll` deblockiert zwar alle wartenden Threads, da sich deren `wait`-Aufrufe jedoch in `synchronized`-Methoden oder -Abschnitten befinden serialisieren sich diese in der Abarbeitung hinsichtlich des Erhalts der Objektsperre.
- Die Java-Spezifikation trifft keinerlei Aussagen darüber in welcher Reihenfolge die blockierten Threads geweckt werden.
- Probleme falls `wait`- und `notify`-Aufrufe während der Ausführungsphase nicht korrekt verzahnt abgearbeitet werden.
- Nachteile des Monitoransatzes:
  - Monitore bleiben entlang von geschachtelten Aufrufen gesperrt, nur der letzte, `wait` aufrufende, wird explizit deblockiert.
  - Im klassischen Konzept führen externe Einflüsse auf den Monitor (externes Entnehmen aus der Warteschlange) zu Schäden am Monitor.
  - Während des Wartens kann ein blockierter Prozeß nicht auf externe Ereignisse reagieren.
  - Das Ursprungskonzept sieht kein zeitlich begrenztes Warten vor.

Beispiel 16 zeigt die Implementierung eines Monitors mit Hilfe der vorgestellten Semaphoreumsetzung.
Der Monitor muß vor der Ausführung eines kritischen Abschnitts durch den Programmierer (Aufruf der Methode `enter`) betreten werden. Entsprechend wird ein Monitor explizit durch `leave`, sofern keine Threads auf den Eintritt warten, oder `notify`, falls Threads durch Aufruf von `wait` auf den Eintritt in den kritischen Abschnitt wartend blockiert wurden, verlassen werden.

Anmerkung:

- Keine Methode innerhalb des Monitors ist als synchronized deklariert!
  Die Synchronisation durch wechselseitigen Ausschluß erfolgt konzentriert in den Objekten der Klasse `Semaphore.`
- Die Methoden `wait` und `notify` greifen auf eine durch den Aufrufer zu erzeugende Semaphore zurück.
  Sie dient der Steuerung der gleichzeitig im Rumpf eines Monitors zugelassenen aktiven Einheiten.

Beispiel 16: *Implementierung eines Monitors mit Hilfe von Semaphoren*
Quellcode

```
(1)public class Monitor {
(2)     private Semaphore mutex;
(3)
(4)     Monitor() {
(5)             mutex = new Semaphore(1);
```

```
(6)     } //constructor
(7)
(8)     public void enter() {
(9)             mutex.p();
(10)    } //enterMonitor()
(11)
(12)    public void leave() {
(13)            mutex.v();
(14)    } //leaveNormally()
(15)
(16)    public void notify(Semaphore s) {
(17)            s.v();
(18)    } //notify()
(19)
(20)    public void wait(Semaphore s) {
(21)            mutex.v();
(22)            s.p();
(23)            mutex.p();
(24)    } //wait()
(25)} //class Monitor
```

Synchronisationsprobleme und ihre Lösung

Erzeuger-Verbraucher Probleme

Auch bekannt als: *Leser-Schreiber-Problem*

Motivation: Typische Beziehung: Genau ein oder eine Menge von Produzenten stellen eine Ware oder Dienstleistung zur Verfügung die von genau einem oder einer Menge von Konsumenten entgegengenommen wird.

Die ausschließliche Verwendung von Semaphoren führt direkten Synchronisation von Erzeuger und Verbraucher, da der Erzeuger nicht „auf Vorrat“ produzieren kann, sondern den Verbraucher immer direkt beliefern muß.

Das Beispiel simuliert eine Backerei. Der Bäcker (=Produzent) ist in der Lage eine gewisse Anzahl von Kunden (übergebener Komandozeilenparameter als Initialisierung der Semaphore) gleichzeitig zu bedienen. Alle weiteren eintreffenden Kunden müssen waren, bis der Bäcker wieder verfügbar ist. Neue Kunden können durch Tastendruck dynamisch erzeugt werden.

Anmerkung: Die Subtraktion vom Schätzwert der aktuell in Ausführung befindlichen Threads in Zeile 79 liefert ausschließlich für die SUN Java-Referenzimplementierung korrekte Werte, die Anzahl der durch die virtuelle Maschine definierten Threads für Systemaufgaben kann auf anderen Plattformen abweichen.

Beispiel 17: *Erzeuger-Verbraucherproblem am Beispiel einer Bäckerei*
Quellcode Ausgabe der Ausführung ... auf einer Multiprozessormaschine

```
(1)import java.io.*;
(2)
(3)public class BakerySimulation
(4){
(5)     public static void main(String argv[])
(6)     {
(7)             Baker b = new Baker(Integer.parseInt(argv[0]));
(8)             (new Thread(new BakeryDesk(b))).start();
(9)     } //end main()
(10)} //class BakerySimulation
(11)
(12)class BakeryDesk implements Runnable
(13){
(14)    Baker b;
(15)    public BakeryDesk(Baker b)
(16)    {
(17)            this.b = b;
(18)    } //constructor
```

```
(19)
(20)    public void run()
(21)    {
(22)            FileReader fr = new FileReader( FileDescriptor.in );
(23)            try
(24)            {
(25)                    while (fr.read() != 'x')
(26)                    {
(27)                            (new Thread(new Customer(b))).start();
(28)                    } //while
(29)            } //try
(30)            catch (IOException ioe)
(31)            {
(32)                    System.out.println("An IOException caught\n"+ioe.getMessage());
(33)                    ioe.printStackTrace();
(34)                    System.exit(1);
(35)            } //catch()
(36)    } //run()
(37)} //class BakeryDesk
(38)
(39)class Baker
(40){
(41)    Semaphore bakerSem;
(42)    public Baker(int noBakers)
(43)    {
(44)            bakerSem = new Semaphore(noBakers);
(45)    } //constructor
(46)
(47)    public void bake()
(48)    {
(49)            try
(50)            {
(51)                    Thread.sleep(500);
(52)            } //try
(53)            catch (InterruptedException ie)
(54)            {
(55)                    System.out.println("An InterruptedException caught\n"+ie.getMessage());
(56)                    ie.printStackTrace();
(57)                    System.exit(1);
(58)            } //catch()
(59)    } //bake()
(60)} //class Baker
(61)
(62)class Customer implements Runnable
(63){
(64)    Baker myBaker;
(65)
(66)    public Customer(Baker b)
(67)    {
(68)            myBaker = b;
(69)    } //constructor
(70)
(71)    public void run()
(72)    {
(73)            //System.out.println("new customer "+Thread.currentThread().getName()+" shows up");
(74)            long start = System.currentTimeMillis();
(75)            myBaker.bakerSem.p();
(76)            myBaker.bake();
(77)            myBaker.bakerSem.v();
(78)            System.out.println("waiting time for "+Thread.currentThread().getName()+": "+ (System.
currentTimeMillis()-start));
(79)            System.out.println("queue lenght: "+ (Thread.currentThread().activeCount()-4) );
(80)    } //requestBread();
(81)} //class Customer
```

Beobachtung: Bei zunehmender Kundenzahl steigen die Wartezeiten ab einer gewissen Grenze stark an. Trotz der „Entschärfung“ durch die Anzahl der parallel bedienbaren Kunden offenbart sich daher die direkte Kopplung zwischen Erzeuger und Verbraucher als potentieller Engpaß.

Gleichzeitig fällt auf, daß die freie Kapazität des Bäckers zwischen Kundenbesuchen nicht genutzt wird.

Abhilfe: Einführung eines Puffers (im Sinne des Beispiels: Regallagers) zur Entkopplung von Produzent und Konsument.

Beispiel 18: *Datenpuffer zur Entkopplung von Sender und Empfänger*
Quellcode

```
(1)import java.util.LinkedList;
(2)
(3)public class Buffer
(4){
(5)     int maxElements;
(6)     LinkedList buffer;
(7)
(8)     public Buffer(int size)
(9)     {
(10)            buffer = new LinkedList();
(11)            maxElements = size;
(12)    } //constructor
(13)
(14)    public synchronized void put(Object o)
(15)    {
(16)            while(maxElements == buffer.size())
(17)            {
(18)                    try
(19)                    {
(20)                            wait();
(21)                    } //try
(22)                    catch (InterruptedException ie)
(23)                    {
(24)                            System.out.println("An InterruptedException caught\n"+ie.getMessage());
(25)                            ie.printStackTrace();
(26)                    } //catch
(27)            } //while
(28)
(29)            buffer.add(o);
(30)            notifyAll();
(31)    } //put()
(32)
(33)    public synchronized Object get()
(34)    {
(35)            while ( buffer.isEmpty() )
(36)            {
(37)                    try
(38)                    {
(39)                            wait();
(40)                    } //try
(41)                    catch (InterruptedException ie)
(42)                    {
(43)                            System.out.println("An InterruptedException caught\n"+ie.getMessage());
(44)                            ie.printStackTrace();
(45)                    } //catch
(46)            } //while
(47)
(48)            notifyAll();
(49)            return( buffer.removeFirst() );
(50)    } //get()
(51)
(52)    public synchronized int getSize()
(53)    {
(54)            return ( buffer.size() );
(55)    } //getSize()
(56)} //class Buffer
```

Anmerkungen zum Code:

- Die Verwaltung strukturierter Nachrichten, bis hin zu Objektgraphen ist durch die ausschließliche

Verwaltung von Daten des Typs `Object` innerhalb der Standard-API-Klasse `LinkedList` (siehe Zeile 6) realisiert.

- Seit Java v1.4 stehen komfortable Methoden zur Umsetzung typspezifischer Puffer für alle numerischen Primitivtypen (z.B. `IntBuffer`) zur Verfügung.

Beispiel 19: *Erzeuger-Verbraucherproblem am Beispiel einer Bäckerei unter Verwendung eines Puffers*
Quellcode Ausgabe der Ausführung ... auf einer Multiprozessormaschine

```
(1)import java.io.*;
(2)
(3)public class BakerySimulation2
(4){
(5)     public static void main(String argv[])
(6)     {
(7)             Buffer shelf = new Buffer(Integer.parseInt(argv[1]));
(8)             Baker b = new Baker(Integer.parseInt(argv[0]), shelf);
(9)             Thread bakerThread = new Thread(b);
(10)            bakerThread.setDaemon(true);
(11)            bakerThread.start();
(12)            (new Thread(new BakeryDesk(b))).start();
(13)    } //end main()
(14)} //class BakerySimulation2
(15)
(16)class BakeryDesk implements Runnable
(17){
(18)    Baker b;
(19)    public BakeryDesk(Baker b)
(20)    {
(21)            this.b = b;
(22)    } //constructor
(23)
(24)    public void run()
(25)    {
(26)            FileReader fr = new FileReader( FileDescriptor.in );
(27)            try
(28)            {
(29)                    while (fr.read() != 'x')
(30)                    {
(31)                            (new Thread(new Customer(b))).start();
(32)                    } //while
(33)            } //try
(34)            catch (IOException ioe)
(35)            {
(36)                    System.out.println("An IOException caught\n"+ioe.getMessage());
(37)                    ioe.printStackTrace();
(38)                    System.exit(1);
(39)            } //catch()
(40)    } //run()
(41)} //class BakeryDesk
(42)
(43)class Baker implements Runnable
(44){
(45)    Semaphore bakerSem;
(46)    Buffer shelf;
(47)    public Baker(int noBakers, Buffer shelf)
(48)    {
(49)            bakerSem = new Semaphore(noBakers);
(50)            this.shelf = shelf;
(51)    } //constructor
(52)
(53)    public void run()
(54)    {
(55)            for (;;)
(56)            {
(57)                    try
(58)                    {
(59)                            Thread.sleep(500);
(60)                    } //try
(61)                    catch (InterruptedException ie)
(62)                    {
```

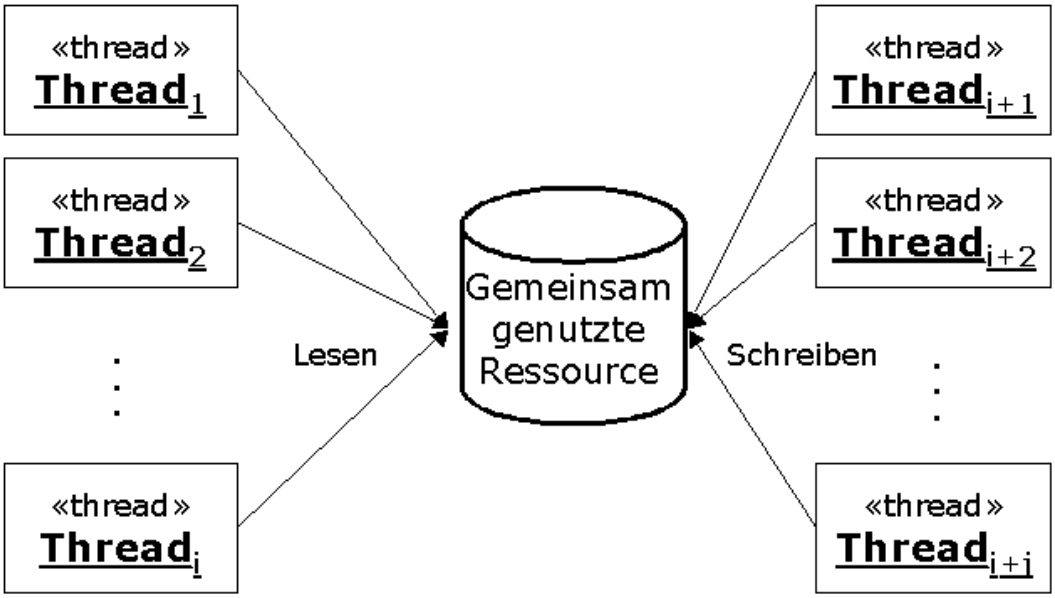
```
(63)                            System.out.println("An InterruptedException caught\n"+ie.getMessage());
(64)                            ie.printStackTrace();
(65)                            System.exit(1);
(66)                    } //catch()
(67)                    shelf.put("bread");
(68)                    System.out.println("refilling shelf ...");
(69)            } //do forever
(70)    } //run()
(71)
(72)    public void bake()
(73)    {
(74)            shelf.get(); //return type is ignored here
(75)            System.out.println("stock size: "+shelf.getSize() );
(76)    } //bake()
(77)} //class Baker
(78)
(79)class Customer implements Runnable
(80){
(81)    Baker myBaker;
(82)
(83)    public Customer(Baker b)
(84)    {
(85)            myBaker = b;
(86)    } //constructor
(87)
(88)    public void run()
(89)    {
(90)            //System.out.println("new customer "+Thread.currentThread().getName()+" shows up");
(91)            long start = System.currentTimeMillis();
(92)            myBaker.bakerSem.p();
(93)            myBaker.bake();
(94)            myBaker.bakerSem.v();
(95)            System.out.println("waiting time for "+Thread.currentThread().getName()+": "+ (System.
currentTimeMillis()-start));
(96)            System.out.println("queue lenght: "+ (Thread.currentThread().activeCount()-4) );
(97)    } //requestBread();
(98)} //class Customer
```

Anmerkungen zum Code:

- Die Klasse Baker ist ebenfalls nebenläufig realisiert, um die nebenläufige Befüllung des Lagers zu gestatten.
- Der Thread welcher die nebenläufige Klasse Baker kapselt ist als *Daemon*-Thread definiert, um die Endlosschleife beim Programmende verlassen zu können.

Neben der Synchronisation der Einzelthreads *durch* gemeinsame Resourcennutzung kann aber auch die Synchronisation *zur* gemeinsamen Resourcennutzung in den Vordergrund treten.
Das bekannteste Beispiel hier dürften gemeinsam genutzte Ressourcen sein, die sowohl von mehreren Lesern als auch Schreibern nebenläufig zugegriffen werden.

Abbildung 9: *Nebenläufige Lese- und Schreibvorgänge*



(click on image to enlarge!)

Beispiel 20: *Implementierung der Synchronisation nebenläufiger Lese- und Schreibvorgänge mit wait und notify*
Quellcode Ausgabe der Ausführung ... auf einer Multiprozessormaschine JavaDoc

```
(1)public class ReaderWriter
(2){
(3)     /**
(4)     * argv[0]: number of concurrently executed readers<br/>
(5)     * argv[1]: number of concurrently executed writers<br/>
(6)     * argv[2]: number of reads per executed reader<br/>
(7)     * argv[3]: number of reads per executed writer<br/>
(8)     */
(9)     public static void main(String argv[])
(10){
(11)            IntCRW data = new IntCRW();
(12)            for (int i=0; i<Integer.parseInt(argv[0]); i++)
(13)            {
(14)                    new Thread(new Writer(data, Integer.parseInt(argv[2]))).start();
(15)            } //for
(16)            for (int i=0; i<Integer.parseInt(argv[1]); i++)
(17)            {
(18)                    new Thread(new Reader(data, Integer.parseInt(argv[3]))).start();
(19)            } //for
(20)    } //main()
(21)} //class ReaderWriter
(22)//---------------------------------------------------------
(23)class IntCRW extends AccessControl
(24){
(25)    int data;
(26)
(27)    protected Object reallyRead()
(28)    {
(29)            return new Integer(data);
(30)    } //reallyRead()
(31)
(32)    protected void reallyWrite(Object obj)
(33)    {
(34)            data = ((Integer) obj).intValue();
(35)    } //reallyWrite()
(36)} //class IntCRW
(37)//---------------------------------------------------------
(38)class Reader implements Runnable
(39){
(40)    private IntCRW data;
(41)    private int noOfReads;
(42)
(43)    public Reader(IntCRW data, int noOfReads)
(44)    {
(45)            this.data = data;
(46)            this.noOfReads = noOfReads;
```

```
(47)    } //constructor
(48)
(49)    public void run()
(50)    {
(51)            for (int i=0; i<noOfReads; i++)
(52)            {
(53)                    Integer myInt = (Integer) data.read();
(54)                    System.out.println(Thread.currentThread().getName()+" read value "+myInt );
(55)            } //for
(56)    } //run()
(57)} //class Reader()
(58)//-------------------------------------------------------
(59)class Writer implements Runnable
(60){
(61)    private IntCRW data;
(62)    private int noOfWrites;
(63)
(64)    public Writer(IntCRW data, int noOfWrites)
(65)    {
(66)            this.data = data;
(67)            this.noOfWrites = noOfWrites;
(68)    } //constructor
(69)
(70)    public void run()
(71)    {
(72)            int value;
(73)            for (int i=0; i<noOfWrites; i++)
(74)            {
(75)                    value = (int) (Math.random()*Integer.MAX_VALUE);
(76)                    data.write(new Integer( value ));
(77)                    System.out.println(Thread.currentThread().getName()+" wrote value "+value );
(78)            } //for
(79)    } //run()
(80)} //class Writer
(81)//-------------------------------------------------------
(82)abstract class AccessControl
(83){
(84)    private int activeReaders = 0;
(85)    private int activeWriters = 0;
(86)    private int waitingReaders = 0;
(87)    private int waitingWriters = 0;
(88)
(89)    protected abstract Object reallyRead();
(90)    protected abstract void reallyWrite(Object obj);
(91)
(92)    public Object read()
(93)    {
(94)            beforeRead();
(95)            Object obj = reallyRead();
(96)            afterRead();
(97)
(98)            return obj;
(99)    } //read()
(100)
(101)   public void write(Object obj)
(102)   {
(103)           beforeWrite();
(104)           reallyWrite(obj);
(105)           afterWrite();
(106)   } //write()
(107)
(108)   private synchronized void beforeRead()
(109)   {
(110)           waitingReaders++;
(111)           while( waitingWriters != 0 || activeWriters != 0 )
(112)           {
(113)                   try
(114)                   {
(115)                           wait();
```

```
(116)                   } //try
(117)                   catch (InterruptedException ie)
(118)                   {
(119)                           System.out.println("An InterruptedException caught\n"+ie.getMessage());
(120)                           ie.printStackTrace();
(121)                           System.exit(1);
(122)                   } //catch()
(123)           } //while
(124)           waitingReaders--;
(125)           activeReaders++;
(126)   } //beforeRead()
(127)
(128)   private synchronized void afterRead()
(129)   {
(130)           activeReaders--;
(131)           notifyAll();
(132)   } //afterRead
(133)
(134)   private synchronized void beforeWrite()
(135)   {
(136)           waitingWriters++;
(137)           while( activeReaders != 0 || activeWriters != 0 )
(138)           {
(139)                   try
(140)                   {
(141)                           wait();
(142)                   } //try
(143)                   catch (InterruptedException ie)
(144)                   {
(145)                           System.out.println("An InterruptedException caught\n"+ie.getMessage());
(146)                           ie.printStackTrace();
(147)                           System.exit(1);
(148)                   } //catch()
(149)           } //while
(150)           waitingWriters--;
(151)           activeWriters++;
(152)   } //beforeWrite()
(153)
(154)   private synchronized void afterWrite()
(155)   {
(156)           activeWriters--;
(157)           notifyAll();
(158)   } //afterWrite()
(159)} //class AccessControl
```

Anmerkungen:

- Die Implementierung nimmt keine Rücksicht auf doppelt konsumierte Daten. Durch Kombination mit einem Puffer kann dies leicht durch Schreiben aller Produzenten in den Buffer und Entnahme der dort verwalteten Daten durch die Konsumenten realisiert werden.
- Die Lösung bevorzugt beim Lesen die schreibenden Threads (siehe Zeile)
- ... beim Schreiben wird hingegen nur auf aktive (lese- und schreib-)Threads Rücksicht genommen, wartende Lesethreads bleiben unberücksichtigt (siehe Zeile)

Beispiel 21: *Implementierung der Synchronisation nebenläufiger Lese- und Schreibvorgänge mit Semaphoren*
Quellcode Ausgabe der Ausführung ... auf einer Multiprozessormaschine JavaDoc

```
(1)abstract class AccessControl
(2){
(3)     private Semaphore2 sem = new Semaphore2(100);
(4)
(5)     protected abstract Object reallyRead();
(6)     protected abstract void reallyWrite(Object obj);
(7)
(8)     public Object read()
(9)     {
(10)            beforeRead();
(11)            Object obj = reallyRead();
(12)            afterRead();
(13)
```

```
(14)            return obj;
(15)    } //read()
(16)
(17)    public void write(Object obj)
(18)    {
(19)            beforeWrite();
(20)            reallyWrite(obj);
(21)            afterWrite();
(22)    } //write()
(23)
(24)    private synchronized void beforeRead()
(25)    {
(26)            sem.p();
(27)    } //beforeRead()
(28)
(29)    private synchronized void afterRead()
(30)    {
(31)            sem.v();
(32)    } //afterRead
(33)
(34)    private synchronized void beforeWrite()
(35)    {
(36)            sem.pAll();
(37)    } //beforeWrite()
(38)
(39)    private synchronized void afterWrite()
(40)    {
(41)            sem.vAll();
(42)    } //afterWrite()
(43)} //class AccessControl
```

Anmerkungen:

- Das Beispiel verwendet eine gegenüber der ursprünglichen Semaphoren-Implementierung erweitere Umsetzung, die es gestattet die Sperrvariable vollständig zu belegen, d.h. exklusiven Zugriff zu erlangen.
- Durch die Initialisierung des Semaphor kann, über das bisherige Steuerungsinstrument der Threadanzahl hinaus, die Menge der nebenläufigen Lesevorgänge gesteuert werden.

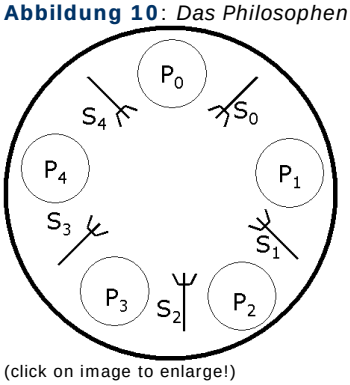
Hungrige Philosophen

Das sog. *Problem der hungrigen Philosophen* dürfte das bekannteste Synchronisationsproblem überhaupt sein ...

Die (bekannten) Grundvoraussetzungen:

- *n* hungrige Philosophen um einen runden Tisch
- *n* bereitliegende Gabeln (je nach Phantasie auch Ess-Stäbchen oder ähnliches)
- Zur Nahrungsaufnahme benötigt jeder Philosoph zwei Gabeln
- Nach der Nahrungsaufnahme werden beide Gabeln wieder zurück auf den Tisch gelegt

Abbildung 10: *Das Philosophen-Problem*



(click on image to enlarge!)

Beispiel 22: *Implementierung der hungrigen Philosophen mit wait und notify*
Quellcode Ausgabe der Ausführung ... auf einer Multiprozessormaschine

```
(1)class Table
(2){
```

```
(3)     boolean forkInUse[];
(4)
(5)     public Table(int seats)
(6)     {
(7)             forkInUse = new boolean[seats];
(8)             for (int i=0; i<forkInUse.length; i++)
(9)                     forkInUse[i] = false;
(10)    } //constructor
(11)
(12)    private int left(int i)
(13)    {
(14)            return i;
(15)    } //left()
(16)
(17)    private int right(int i)
(18)    {
(19)            if (i+1 < forkInUse.length){
(20)                    return (i+1);
(21)            } //if
(22)            else {
(23)                    return 0;
(24)            } //else
(25)    } //right()
(26)
(27)    public synchronized void useFork(int seat)
(28)    {
(29)            while( forkInUse[left(seat)]  || forkInUse[right(seat)] )
(30)            {
(31)                    System.out.println("Philosopher #"+Thread.currentThread().getName()+" is
waiting for forks");
(32)                    try
(33)                    {
(34)                            wait();
(35)                    } //try
(36)                    catch (InterruptedException ie)
(37)                    {
(38)                            System.out.println("An InterruptedException caught\n"+ie.getMessage());
(39)                            ie.printStackTrace();
(40)                            System.exit(1);
(41)                    } //catch()
(42)            } //while
(43)            forkInUse[left(seat)] = true;
(44)            forkInUse[right(seat)] = true;
(45)    } //useFork()
(46)
(47)    public synchronized void releaseFork(int seat)
(48)    {
(49)            forkInUse[left(seat)] = false;
(50)            forkInUse[right(seat)] = false;
(51)            notifyAll();
(52)    } //relaeaseFork()
(53)} //class Table
(54)
(55)class Philosopher implements Runnable
(56){
(57)    Table myTable;
(58)    int seat;
(59)
(60)    public Philosopher(Table table, int seat)
(61)    {
(62)            myTable = table;
(63)            this.seat = seat;
(64)    } //constructor
(65)
(66)    public void run()
(67)    {
(68)            while(true)
(69)            {
(70)                    think(seat);
```

```
(71)                    myTable.useFork(seat);
(72)                    eat(seat);
(73)                    myTable.releaseFork(seat);
(74)            } //loop endlessly
(75)    } //run()
(76)
(77)    void think(int seat)
(78)    {
(79)            System.out.println("Philosopher #"+Thread.currentThread().getName()+" is thinking");
(80)            try
(81)            {
(82)                    Thread.sleep( (int) (Math.random() * 200) );
(83)            } //try
(84)            catch (InterruptedException ie)
(85)            {
(86)                    System.out.println("An InterruptedException caught\n"+ie.getMessage());
(87)                    ie.printStackTrace();
(88)                    System.exit(1);
(89)            } //catch()
(90)    } //think()
(91)
(92)    void eat(int seat)
(93)    {
(94)            System.out.println("Philosopher #"+Thread.currentThread().getName()+" is eating");
(95)            try
(96)            {
(97)                    Thread.sleep( (int) (Math.random() * 200) );
(98)            } //try
(99)            catch (InterruptedException ie)
(100)           {
(101)                   System.out.println("An InterruptedException caught\n"+ie.getMessage());
(102)                   ie.printStackTrace();
(103)                   System.exit(1);
(104)           } //catch()
(105)           System.out.println("Philosopher #"+Thread.currentThread().getName()+" finished
eating");
(106)   } //eat()
(107)} //class Philosopher
(108)
(109)public class HungryPhilosophers
(110){
(111)   public static void main(String argv[])
(112)   {
(113)           int hungryPhilosophers = Integer.parseInt(argv[0]);
(114)           Table table = new Table( hungryPhilosophers );
(115)           for (int i=0; i<hungryPhilosophers; i++)
(116)           {
(117)                   new Thread(new Philosopher(table, i),""+(i+1)).start();
(118)           } //for
(119)   } //main()
(120)} //class HungryPhilosophers
```

Beispiel 23: *Implementierung der hungrigen Philosophen mit einer Semaphoregruppe*
Quellcode Ausgabe der Ausführung ... auf einer Multiprozessormaschine

```
(1)public class HungryPhilosophers2 {
(2)     public static void main(String argv[]) {
(3)             int hungryPhilosophers = Integer.parseInt(argv[0]);
(4)
(5)             SemaphoreGroup accessForks = new SemaphoreGroup( hungryPhilosophers );
(6)             int[] init = new int[ hungryPhilosophers ];
(7)             for (int i=0; i< init.length; i++)
(8)                     init[i] = 1;
(9)
(10)            accessForks.changeValues(init);
(11)
(12)            Table table = new Table( hungryPhilosophers );
(13)            for (int i=0; i<hungryPhilosophers; i++) {
(14)                    new Thread(new Philosopher(table, accessForks, i),""+(i+1)).start();
```

```
(15)            } //for
(16)    } //main()
(17)} //class HungryPhilosophers2
(18)
(19)class Table {
(20)    boolean forkInUse[];
(21)
(22)    public Table(int seats)
(23)    {
(24)            forkInUse = new boolean[seats];
(25)            for (int i=0; i<forkInUse.length; i++)
(26)                    forkInUse[i] = false;
(27)    } //constructor
(28)
(29)    public int left(int i) {
(30)            return i;
(31)    } //left()
(32)
(33)    public int right(int i) {
(34)            if (i+1 < forkInUse.length)
(35)                    return (i+1);
(36)            else
(37)                    return 0;
(38)    } //right()
(39)} //class Table
(40)
(41)class Philosopher implements Runnable {
(42)    SemaphoreGroup accessForks;
(43)    Table table;
(44)    int seat;
(45)
(46)    public Philosopher(Table table, SemaphoreGroup accessForks, int seat) {
(47)            this.accessForks = accessForks;
(48)            this.table = table;
(49)            this.seat = seat;
(50)    } //constructor
(51)
(52)    public void run() {
(53)            int deltas[] = new int[accessForks.getNumberOfMembers()];
(54)
(55)            while (true) {
(56)                    think(seat);
(57)                    deltas[table.left(seat)] = -1;
(58)                    deltas[table.right(seat)] = -1;
(59)                    accessForks.changeValues(deltas);
(60)
(61)                    eat(seat);
(62)                    deltas[table.left(seat)] = 1;
(63)                    deltas[table.right(seat)] = 1;
(64)                    accessForks.changeValues(deltas);
(65)            } //loop endlessly
(66)    } //constructor
(67)
(68)    void think(int seat)    {
(69)            System.out.println("Philosopher #"+Thread.currentThread().getName()+" is thinking");
(70)            try {
(71)                    Thread.sleep( (int) (Math.random() * 20000) );
(72)            } //try
(73)            catch (InterruptedException ie) {
(74)                    System.out.println("An InterruptedException caught\n"+ie.getMessage());
(75)                    ie.printStackTrace();
(76)                    System.exit(1);
(77)            } //catch()
(78)    } //think()
(79)
(80)    void eat(int seat){
(81)            try {
(82)                    System.out.println("Philosopher #"+Thread.currentThread().getName()+" is
eating");
```

```
(83)                    Thread.sleep( (int) (Math.random() * 20000) );
(84)            } //try
(85)            catch (InterruptedException ie) {
(86)                    System.out.println("An InterruptedException caught\n"+ie.getMessage());
(87)                    ie.printStackTrace();
(88)                    System.exit(1);
(89)            } //catch()
(90)            System.out.println("Philosopher #"+Thread.currentThread().getName()+" finished
eating");
(91)    } //eat()
(92)} //class Philosopher
```

Als abschließendes Beispiel 24 ist die Umsetzung des Philosophenproblems mit Hilfe von Monitoren realisiert.
Auffallend hierbei ist insbesondere, daß die Methoden useFork und releaseFork nicht mehr als synchronized deklariert werden müssen, da die Realisierung des wechselseitigen Ausschlusses durch den Monitor -- und hierbei durch das dieser Umsetzung zugrundeliegende Semaphor -- erfolgt.
Ferner ist die Äquivalenz der Java-API-Methode wait und der gleichnamigen Implementierung des Monitors für diesen Anwendungsfall offensichtlich. Vielmehr noch, läßt sich sogar unter Kenntnis der Implementierung des Semaphor zeigen, daß diese auf die Nutzung des genannten API-Aufrufes zurückzuführen ist.
Ähnliches gilt im analogen Falle, der Substitution des Aufrufes von notifyAll durch die Methode notify (Zeile 40).

Beispiel 24: *Implementierung der hungrigen Philosophen mit Hilfe eines Monitors*
Quellcode Ausgabe der Ausführung ... auf einer Multiprozessormaschine

```
(1)class Table {
(2)     boolean forkInUse[];
(3)     Monitor forkMon = new Monitor();
(4)     Semaphore forkSem = new Semaphore(1);
(5)
(6)     public Table(int seats) {
(7)             forkInUse = new boolean[seats];
(8)             for (int i=0; i<forkInUse.length; i++)
(9)                     forkInUse[i] = false;
(10)    } //constructor
(11)
(12)    private int left(int i) {
(13)            return i;
(14)    } //left()
(15)
(16)    private int right(int i) {
(17)            if (i+1 < forkInUse.length){
(18)                    return (i+1);
(19)            } //if
(20)            else {
(21)                    return 0;
(22)            } //else
(23)    } //right()
(24)
(25)    public void useFork(int seat) {
(26)            forkMon.enter();
(27)            while( forkInUse[left(seat)]  || forkInUse[right(seat)] ) {
(28)                    System.out.println("Philosopher #"+Thread.currentThread().getName()+" is
waiting for forks");
(29)                    forkMon.wait(forkSem); //wait();
(30)            } //while
(31)                    forkInUse[left(seat)] = true;
(32)                    forkInUse[right(seat)] = true;
(33)            forkMon.leave();
(34)    } //useFork()
(35)
(36)    public void releaseFork(int seat) {
(37)            forkMon.enter();
(38)            forkInUse[left(seat)] = false;
(39)            forkInUse[right(seat)] = false;
(40)            forkMon.notify(forkSem);  //notifyAll();
```

```
(41)            forkMon.leave();
(42)    } //relaeaseFork()
(43)} //class Table
(44)
(45)class Philosopher implements Runnable {
(46)    Table myTable;
(47)    int seat;
(48)
(49)    public Philosopher(Table table, int seat) {
(50)            myTable = table;
(51)            this.seat = seat;
(52)    } //constructor
(53)
(54)    public void run() {
(55)            while(true) {
(56)                    think(seat);
(57)                    myTable.useFork(seat);
(58)                    eat(seat);
(59)                    myTable.releaseFork(seat);
(60)            } //loop endlessly
(61)    } //run()
(62)
(63)    void think(int seat) {
(64)            System.out.println("Philosopher #"+Thread.currentThread().getName()+" is thinking");
(65)            try {
(66)                    Thread.sleep( (int) (Math.random() * 20000) );
(67)            } //try
(68)            catch (InterruptedException ie) {
(69)                    System.out.println("An InterruptedException caught\n"+ie.getMessage());
(70)                    ie.printStackTrace();
(71)                    System.exit(1);
(72)            } //catch()
(73)    } //think()
(74)
(75)    void eat(int seat) {
(76)            System.out.println("Philosopher #"+Thread.currentThread().getName()+" is eating");
(77)            try {
(78)                    Thread.sleep( (int) (Math.random() * 20000) );
(79)            } //try
(80)            catch (InterruptedException ie) {
(81)                    System.out.println("An InterruptedException caught\n"+ie.getMessage());
(82)                    ie.printStackTrace();
(83)                    System.exit(1);
(84)            } //catch()
(85)            System.out.println("Philosopher #"+Thread.currentThread().getName()+" finished
eating");
(86)    } //eat()
(87)} //class Philosopher
(88)
(89)public class HungryPhilosophers3 {
(90)    public static void main(String argv[]) {
(91)            int hungryPhilosophers = Integer.parseInt(argv[0]);
(92)            Table table = new Table( hungryPhilosophers );
(93)            for (int i=0; i<hungryPhilosophers; i++)        {
(94)                    new Thread(new Philosopher(table, i),""+(i+1)).start();
(95)            } //for
(96)    } //main()
(97)} //class HungryPhilosophers
```

Verklemmungen

Verklemmungen (engl. *Deadlocks*) sind eine kennzeichnende Fehlersituation die als Folge fehlerhafter oder unzureichender Synchronisation in verteilten und nebenläufigen Abläufen auftreten können.

Definition 13: *Verklemmung*
Eine Menge von Threads befindet sich in einem Deadlock-Zustand, falls jeder Thread der Menge auf ein Ereignis wartet, daß nur ein anderer Thread der Menge auslösen kann.

Ein solches Ereignis kann die Freigabe einer Sperre oder die Verfügbarkeit einer Ressource sein.

Bedingungen für das Auftreten einer Verklemmung:

- Seriell nutzbare Ressourcen.
  Der Zugriff wird zumeist durch wechselseitigen Ausschluß realisiert.
- Inkrementelle Ressourcenaquise.
  Threads beanspruchen bereits Ressourcen während sie jederzeit weitere nachfordern können.
- Kein Ressourcenentzug.
  Zugeteilte Ressourcen können nicht extern entzogen werden.
- Zyklische Wartebedingungen.

Bemerkung: Aus Sicht des Programmflusses betrachtet stellt ein Deadlock einen Zustand in einem endlichen Automaten dar, zu ausschließlich eingehende Transitionen existieren.

Beispiel 25 zeigt eine Implementierung die zu Verklemmungen führen kann, aber nicht muß.
Das Eintreten einer Verklemmungssituation hängt von der tatsächlichen Reservierungsreihenfolge der ab, die über eine Zufallsvariable in Zeile 35 gesteuert wird.

Beispiel 25: *Implementierung einer Konkurrenzsituation um eine Ressource die zu Verklemmungen führen kann*
Quellcode

```
(1)public class SimpleDeadlock {
(2)/**
(3) * argv[0]: number of concurrently executed threads<br/>
(4)*/
(5)     public static void main(String argv[]) {
(6)             Semaphore resources[] = new Semaphore[2];
(7)
(8)             for (int i=0; i<resources.length; i++)
(9)                     resources[i] = new Semaphore(1);
(10)
(11)            for (int i=0; i<Integer.parseInt(argv[0]); i++)
(12)                    new Thread( new DeadLockingThread(resources) ).start();
(13)    } //main()
(14)} //class SimpleDeadlock
(15)
(16)class DeadLockingThread implements Runnable {
(17)    Semaphore resources[];
(18)
(19)    public DeadLockingThread(Semaphore resources[]) {
(20)            this.resources = new Semaphore[resources.length];
(21)            for (int i=0; i<resources.length; i++)
(22)                    this.resources[i] = resources[i];
(23)    } //constructor
(24)
(25)    public void run() {
(26)            System.out.println(Thread.currentThread().getName()+" is sleeping");
(27)            try {
(28)                    Thread.sleep( (int) (Math.random() * 20000) );
(29)            } //try
(30)            catch (InterruptedException ie) {
(31)                    System.out.println("An InterruptedException caught\n"+ie.getMessage());
(32)                    ie.printStackTrace();
(33)                    System.exit(1);
(34)            } //catch()
(35)            int requestedResource=(int) (Math.random()*1.1);
(36)
(37)            System.out.println(Thread.currentThread().getName()+" requests resource
#"+requestedResource);
(38)            resources[requestedResource].p();
(39)            System.out.println(Thread.currentThread().getName()+" locks resource
#"+requestedResource);
(40)            try {
(41)                    Thread.sleep( (int) (Math.random() * 20000) );
(42)            } //try
```

```
(43)            catch (InterruptedException ie) {
(44)                    System.out.println("An InterruptedException caught\n"+ie.getMessage());
(45)                    ie.printStackTrace();
(46)                    System.exit(1);
(47)            } //catch()
(48)
(49)            //deadlock prone section begins
(50)            int otherResource=Math.abs(requestedResource-1);
(51)            System.out.println(Thread.currentThread().getName()+" requests resource
#"+otherResource);
(52)            resources[otherResource].p();
(53)            System.out.println(Thread.currentThread().getName()+" locks resource #"+otherResource);
(54)            try {
(55)                    Thread.sleep( (int) (Math.random() * 20000) );
(56)            } //try
(57)            catch (InterruptedException ie) {
(58)                    System.out.println("An InterruptedException caught\n"+ie.getMessage());
(59)                    ie.printStackTrace();
(60)                    System.exit(1);
(61)            } //catch()
(62)            resources[otherResource].v();
(63)            System.out.println(Thread.currentThread().getName()+" released resource
#"+otherResource);
(64)            //deadlock prone section ends
(65)            resources[requestedResource].v();
(66)            System.out.println(Thread.currentThread().getName()+" released resource
#"+requestedResource);
(67)    } //run()
(68)} //class DeadLockingThread
```

Ausgabe eines verklemmungssfreien Ablaufs

```
Thread-2 is sleeping
Thread-1 is sleeping
Thread-1 requests resource #0
Thread-2 requests resource #0
Thread-1 locks resource #0
Thread-1 requests resource #1
Thread-1 locks resource #1
Thread-1 released resource #1
Thread-1 released resource #0
Thread-2 locks resource #0
Thread-2 requests resource #1
Thread-2 locks resource #1
Thread-2 released resource #1
Thread-2 released resource #0
[Programmende]
```

Ausgabe Ablaufs der zu einer Verklemmung führt

```
Thread-1 is sleeping
Thread-2 is sleeping
Thread-2 requests resource #0
Thread-1 requests resource #1
Thread-2 locks resource #0
Thread-1 locks resource #1
Thread-2 requests resource #1
Thread-1 requests resource #0
[Programmstillstand, keine weiteren Ausgaben]
```

Visualisierung von Verklemmungsszenarien

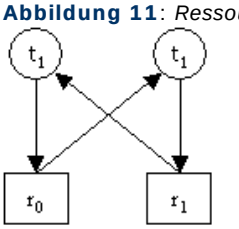
Zur Analyse und graphischen Veranschaulichung von Verklemmungsszenarien hat sich die Darstellung als gerichteter Graph eingebürgert.
Im folgenden ist hierfür folgende symbolische Notation gewählt:

- Threads werden durch beschriftete Knoten repräsentiert.
- Ressourcen werden durch beschriftete Rechtecke repräsentiert.

- Gerichtete Kanten ausgehend von einem Tread t_i zu einer Ressource r_j werden als *Thread t_i fordert Ressource r_j an* interpretiert.
- Gerichtete Kanten ausgehend von einer Ressource r_k zu einem Thread t_l werden als *Ressource r_k ist von Thread t_l belegt* interpretiert.
- Üblicherweise, d.h. sofern nicht anders angegeben kann eine Ressource nur exklusiv belegt werden.

Treten in einem solchen Ressourcengraphen zu einem Zeitpunkt Zyklen auf, so liegt ein Verklemmungszustand vor.
Abbildung 11 zeigt dies für den oben dargestellten Fall.

Abbildung 11: *Ressourcengraph einer Verklemmungssituation*



(click on image to enlarge!)

Verklemmt trotzt synchronized ...

Ähnlich der Verwendung von Semaphoren im vorangegangenen Beispiel, kann der Einsatz von durch die Laufzeitumgebung synchronisierte Abschnitte oder Methoden nicht die Verklemmungsfreiheit eines Ablaufes garantieren. In manchen Fällen kann sie sogar ursächlich für das Eintreten einer Verklemmungssituation sein.

Beispiel 1 zeigt eine Verklemmung die bei der Verwendung von `synchronized` geschützten Abschnitten eintritt.
Der Code simuliert auf einfache Weise Überweisungen die zwischen Konten (modelliert durch Objekte der Klasse `Account`) vorgenommen werden. Die Ermittlung der beiden beteiligten Konten, sowie des zu transferierenden Betrages erfolgt zufallsgesteuert.
Vor der Durchführung der Überweisung (Zeile) wird das Quell- und Zielkonto durch wechselseitigen Ausschluß gesperrt. Hierzu werden nacheinander die entsprechenden Objektsperren gesetzt (Zeile und).

Beispiel 26: *Verklemmung bei der Verwendung des Schlüsselwortes synchronized*
Quellcode

```
(1)public class Bank {
(2)/**
(3) * argv[0]: number of concurrently executed threads
(4) * argv[1]: number of managed accounts
(5)*/
(6)
(7)     public static void main(String argv[]) {
(8)             int noAccounts = Integer.parseInt(argv[1]);
(9)             Account accounts[] = new Account[noAccounts];
(10)            for (int i=0; i<noAccounts; i++)
(11)                    accounts[i] = new Account(Math.random()*1000);
(12)
(13)            int fromAccount;
(14)            int toAccount;
(15)
(16)            for (int i=0; i<Integer.parseInt(argv[0]); i++) {
(17)                    fromAccount = (int) (Math.random()*noAccounts);
(18)                    do {
(19)                            toAccount = (int) (Math.random()*noAccounts);
(20)                    } while (toAccount == fromAccount);
(21)
(22)                    new Thread( new Transferal(accounts, fromAccount, toAccount, Math.random()
*1000)).start();
(23)            } //for
(24)    } //main()
(25)} //class Bank
(26)
(27)class Account {
(28)    double balance;
(29)
(30)    public Account(double init) {
```

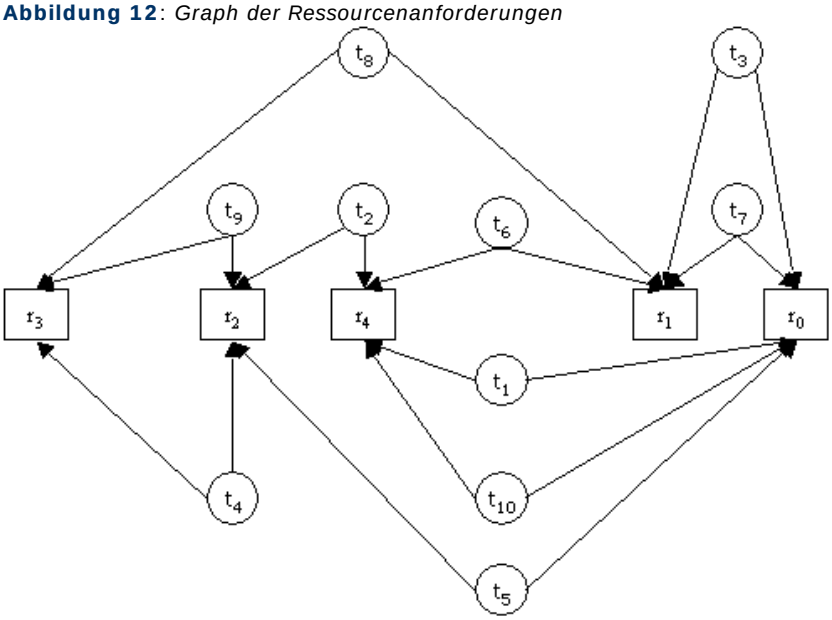
```
(31)            balance = init;
(32)    } //constructor
(33)
(34)    public void change(double amount) {
(35)            balance += amount;
(36)    } //change()
(37)} //class Account
(38)
(39)class Transferal implements Runnable {
(40)    Account accounts[];
(41)    int fromAccount;
(42)    int toAccount;
(43)    double amount;
(44)
(45)    public Transferal(Account accounts[], int fromAccount, int toAccount, double amount) {
(46)            this.accounts = accounts;
(47)            this.fromAccount = fromAccount;
(48)            this.toAccount = toAccount;
(49)            this.amount = amount;
(50)    } //constructor
(51)
(52)    public void run() {
(53)            System.out.println("("+Thread.currentThread().getName()+") accounts["+fromAccount+"]
--"+amount+"--> accounts["+toAccount+"] started");
(54)            synchronized( accounts[fromAccount] ) {
(55)                    System.out.println("("+Thread.currentThread().getName()+") account
["+fromAccount+"] locked (fromAccount)");
(56)                    synchronized( accounts[toAccount] ) {
(57)                            System.out.println("("+Thread.currentThread().getName()+") account
["+toAccount+"] locked (toAccount");
(58)                            try {
(59)                                    Thread.sleep( (int) (Math.random() * 200) );
(60)                            } //try
(61)                            catch (InterruptedException ie) {
(62)                                    System.out.println("An InterruptedException caught\n"+ie.
getMessage());
(63)                                    ie.printStackTrace();
(64)                                    System.exit(1);
(65)                            } //catch()
(66)                            accounts[toAccount].change(amount);
(67)                            accounts[fromAccount].change(-amount);
(68)                    } //synchronized
(69)                    System.out.println("("+Thread.currentThread().getName()+") account["+toAccount
+"] released");
(70)            } //synchronized
(71)            System.out.println("("+Thread.currentThread().getName()+") account["+fromAccount+"]
released");
(72)            System.out.println("("+Thread.currentThread().getName()+") accounts["+fromAccount+"]
--"+amount+"--> accounts["+toAccount+"] finished");
(73)    } //run()
(74)} //class Transferal
```

Vereinfachte **Ausgabe Ablaufs der zu einer Verklemmung führt**

```
(Thread-2) accounts[4] --40.66333304967762--> accounts[2] started
(Thread-8) accounts[3] --366.4855767109112--> accounts[1] started
(Thread-6) accounts[1] --502.98046127613713--> accounts[4] started
(Thread-10) accounts[4] --184.08552879039075--> accounts[0] started
(Thread-4) accounts[2] --292.5794829489161--> accounts[3] started
(Thread-1) accounts[0] --714.5208007641102--> accounts[4] started
(Thread-7) accounts[0] --170.7740899847421--> accounts[1] started
(Thread-9) accounts[2] --720.4974239388226--> accounts[3] started
(Thread-5) accounts[0] --44.318640244877415--> accounts[2] started
(Thread-3) accounts[1] --390.3961760904666--> accounts[0] started
(Thread-2) account[4] locked
(Thread-8) account[3] locked
(Thread-6) account[1] locked
(Thread-4) account[2] locked
```

```
(Thread-1) account[0] locked
[Programmstillstand, keine weiteren Ausgaben]
```

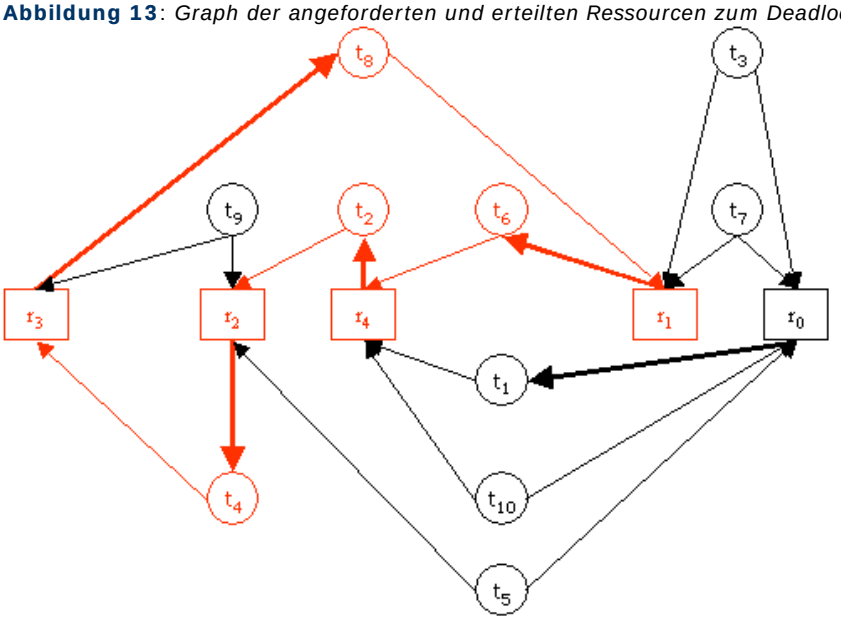
Abbildung 12: *Graph der Ressourcenanforderungen*



(click on image to enlarge!)

Der Ressourcengraph aus Abbildung 13 ordnet die gewährten Ressourcen den sie belegenden Threads zu (dicke Pfeile) und hebt die an der Verklemmung beteiligten Threads und Ressourcen rot hervor.

Abbildung 13: *Graph der angeforderten und erteilten Ressourcen zum Deadlockzeitpunkt*



(click on image to enlarge!)

Analyse: Ursächlich für die Verklemmung ist das unkoordinierte Anfordern der Objektsperren durch `synchronized`. Die durch jeden Thread implementierte Strategie belegt zunächst (in Zeile 54) das Quellkonto und hält dieses solange belegt bis das Zielkonto (in Zeile 56) ebenfalls gesperrt werden kann.
Eine Verklemmungssituation tritt immer dann auf, wenn sich zur Laufzeit Sperrzyklen wie in Abbildung 13 dargestellt ergeben, die Anzahl der beteiligten Threads und Ressourcen kann hierbei durchaus größer zwei sein.
Im Beispiel wurde gezeigt, daß der Einsatz des Schlüsselwortes `synchronized` keineswegs a priori alle Synchronisationsprobleme zur Lösung an die virtuelle Maschine delegiert, sondern im Gegenteil -- im Falle des undurchdachten Einsatzes -- selbst zu Verklemmungssituationen führt.
Konsequenterweise wirkt sich dies auch auf die höheren Synchronisationsprimitive wie Semaphoren oder Monitore aus, die direkt oder indirekt auf der Anwendung dieses Schlüsselwortes basieren.

Abschließend sei gezeigt, daß sich die gezeigte Fehlersituation keineswegs auf den Einsatz von `synchronized` Realisierung des wechselseitigen Ausschlusses auf Blöcke beschränkt, sondern verhaltensgleich auch für die Anwendung auf Methoden übertragen läßt.

Der Beispielcode 27 skizziert eine (naive) objektorietierte Implementierung einer Hüllklasse (engl.

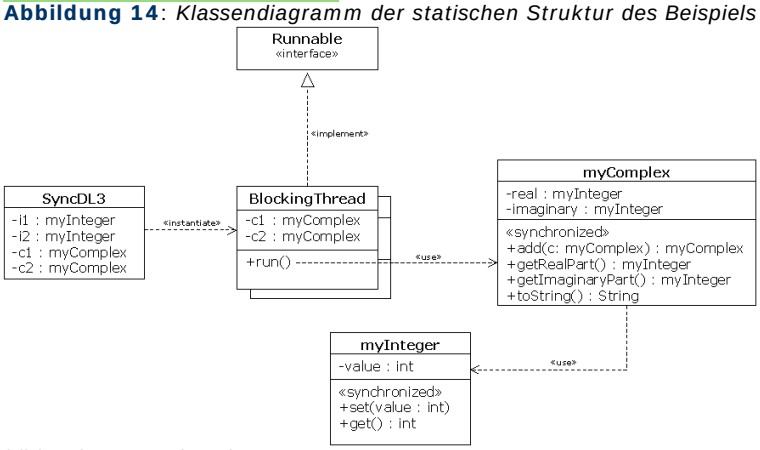
wrapper class) (myComplex) für komplexe Zahlen. Diese Klasse bedient sich ihrerseits einer Hüllklasse für ganze Zahlen (myInteger).
Die komplexen Zahlen 1+i2 und 2+i1 werden auf ausgehend von denselben Objekten -- eines für die Zahl *1* und eines für *2* -- der Integer-Klasse erzeugt.
Durch dieses Vorgehen entsteht eine „Überkreuzabhängigkeit“, dergestalt, daß beide erzeugten komplexen Zahlen speicherintern auf dieselben myInteger-Objekte verweisen.
Die Fehlersituation, welche dann zur Verklemmung führt, entsteht durch die nebenläufige Operation auf verschiedenen synchronisierten Methoden der Klasse myInteger. Konkret tritt der Stillstand in den Fällen ein, in denen eine Threadumschaltung nach einem Teilzugriff (entweder beim Auslesen des auf Real- oder Imaginärteils) erfolgt. Die Verklemmung tritt dann beim Ausleserversuch des Real- oder Imaginärteils der „anderen“ Zahl auf, die auf dieselben Bestandteile (dieselben Objekte) zurückzuführen ist, auf denen bereits synchronized-Methoden ausgeführt werden.

Beispiel 27: *Verklemmung bei der Verwendung synchronisierter Methoden*
Quellcode

```
(1)public class SyncDL3 {
(2)     private myInteger i1;
(3)     private myInteger i2;
(4)
(5)     private myComplex c1;
(6)     private myComplex c2;
(7)
(8)     public SyncDL3(int noThreads) {
(9)             i1 = new myInteger(1);
(10)            i2 = new myInteger(2);
(11)
(12)            c1 = new myComplex( i1, i2, "c1" );
(13)            c2 = new myComplex( i2, i1, "c2" );
(14)
(15)            for (int i=0; i<noThreads; i++) {
(16)                    new Thread( new BlockingThread(c1,c2) ).start();
(17)                    new Thread( new BlockingThread(c2,c1) ).start();
(18)            } //for
(19)    } //constructor
(20)
(21)    public static void main(String argv[]) {
(22)            new SyncDL3(Integer.parseInt(argv[0]));
(23)    } //main()
(24)} //class SyncDL3
(25)
(26)class BlockingThread implements Runnable {
(27)    myComplex c1;
(28)    myComplex c2;
(29)
(30)    public BlockingThread(myComplex c1, myComplex c2) {
(31)            this.c1 = c1;
(32)            this.c2 = c2;
(33)    } //constructor
(34)
(35)    public void run() {
(36)            System.out.println("("+Thread.currentThread().getName()+") "+c1.toString() +" + "+ c2.
toString() +" = "+ (c1.add(c2)).toString() );
(37)    } //main()
(38)
(39)} //class BlockingThread
(40)
(41)class myComplex {
(42)    private myInteger real;
(43)    private myInteger imaginary;
(44)    private String name;
(45)
(46)    public myComplex (myInteger re, myInteger im, String name) {
(47)            real = re;
(48)            imaginary = im;
(49)            this.name = name;
(50)    } //constructor
(51)
(52)    public synchronized myComplex add (myComplex c) {
(53)            System.out.println(Thread.currentThread().getName()+" entered add of number "+name);
```

```
(54)            myInteger resultRealPart = new myInteger( this.getRealPart()+c.getRealPart() );
(55)            System.out.println(Thread.currentThread().getName()+" calculated real part of number
"+name);
(56)            myInteger resultImaginaryPart = new myInteger( this.getImaginaryPart()+c.
getImaginaryPart() );
(57)            System.out.println(Thread.currentThread().getName()+" left add of number "+name);
(58)            return ( new myComplex(resultRealPart, resultImaginaryPart, "" ) );
(59)    } //add()
(60)
(61)    public synchronized int getRealPart() {
(62)            System.out.println(Thread.currentThread().getName()+" entered getRealPart of number
"+name);
(63)            return real.get();
(64)    } //getRealPart()
(65)
(66)    public synchronized int getImaginaryPart() {
(67)            System.out.println(Thread.currentThread().getName()+" entered getImaginaryPartof
number "+name);
(68)            return imaginary.get();
(69)    } //getImaginaryPart()
(70)
(71)    public synchronized String toString() {
(72)            StringBuffer result = new StringBuffer( ""+real.get() );
(73)            int img = imaginary.get();
(74)
(75)            if (img >= 0)
(76)                    result.append("+i"+img);
(77)            else
(78)                    result.append("i"+img);
(79)            return ( result.toString() );
(80)            } //printValue()
(81)} //class myComplex
(82)
(83)class myInteger {
(84)    private int value;
(85)
(86)    myInteger (int init) {
(87)            value = init;
(88)    } //constructor
(89)
(90)    public synchronized void set(int value) {
(91)            this.value = value;
(92)    } //set()
(93)
(94)    public synchronized int get() {
(95)            return value;
(96)    } //get()
(97)} //class myInteger
```

Abbildung 14: *Klassendiagramm der statischen Struktur des Beispiels*



(click on image to enlarge!)

Vereinfachte **Ausgabe eines Ablaufes der zu einer Verklemmung führt**

```
...
Thread-41 entered add of number c1
```

```
Thread-41 entered getRealPart of number c1
Thread-41 entered getRealPart of number c2
Thread-41 calculated real part of number c1
Thread-41 entered getImaginaryPartof number c1
Thread-41 entered getImaginaryPartof number c2
Thread-41 left add of number c1
(Thread-41) 1+i2 + 2+i1 = 3+i3
Thread-144 entered add of number c2
Thread-144 entered getRealPart of number c2
Thread-144 entered getRealPart of number c1
Thread-144 calculated real part of number c2
Thread-144 entered getImaginaryPartof number c2
Thread-144 entered getImaginaryPartof number c1
Thread-144 left add of number c2
(Thread-144) 2+i1 + 1+i2 = 3+i3
Thread-132 entered add of number c2
Thread-132 entered getRealPart of number c2
Thread-129 entered add of number c1
Thread-129 entered getRealPart of number c1
[Programmstillstand, keine weiteren Ausgaben]
```

Verklemmt trotz Semaphore ...

Semaphoren bilden -- bei korrekter Anwendung -- einen gleichermaßen leistungsfähigen wie einfach zu handhabenden Mechanismus zur Synchronisation durch den Applikationsprogrammierer. Werden sie jedoch fehlerhaft eingesetzt, so führt dies unweigerlich zum Auftreten von Verklemmungen.

Dies läßt sich leicht an einer naheliegenden aber fehlerhaften Umsetzung des Philosophenproblems verdeutlichen:

Beispiel 28: *Fehlerhafte Implementierung des Philosophenproblems mit Semaphoren*
Quellcode

```
(1)class Table {
(2)     Semaphore forkInUse[];
(3)
(4)     public Table(int seats) {
(5)             int forks = (seats==1?2:seats);
(6)
(7)             forkInUse = new Semaphore[forks];
(8)             for (int i=0; i<forks; i++)
(9)                     forkInUse[i] = new Semaphore(1);
(10)    } //constructor
(11)
(12)    private int left(int i) {
(13)            return i;
(14)    } //left()
(15)
(16)    private int right(int i) {
(17)            if (i+1 < forkInUse.length) {
(18)                    return (i+1);
(19)            } //if
(20)            else {
(21)                    return 0;
(22)            } //else
(23)    } //right()
(24)
(25)    public void useFork(int seat)   {
(26)            forkInUse[left(seat)].p();
(27)            System.out.println("Philosopher #"+Thread.currentThread().getName()+" received left
fork");
(28)            forkInUse[right(seat)].p();
(29)            System.out.println("Philosopher #"+Thread.currentThread().getName()+" received right
fork");
(30)    } //useFork()
(31)
(32)    public void releaseFork(int seat) {
(33)            forkInUse[left(seat)].v();
```

```
(34)            System.out.println("Philosopher #"+Thread.currentThread().getName()+" released left
fork");
(35)            forkInUse[right(seat)].v();
(36)            System.out.println("Philosopher #"+Thread.currentThread().getName()+" released right
fork");
(37)    } //relaeaseFork()
(38)} //class Table
(39)
(40)class Philosopher implements Runnable
(41){
(42)    Table myTable;
(43)    int seat;
(44)
(45)    public Philosopher(Table table, int seat)
(46)    {
(47)            myTable = table;
(48)            this.seat = seat;
(49)    } //constructor
(50)
(51)    public void run()
(52)    {
(53)            while(true)
(54)            {
(55)                    think(seat);
(56)                    myTable.useFork(seat);
(57)                    eat(seat);
(58)                    myTable.releaseFork(seat);
(59)            } //loop endlessly
(60)    } //run()
(61)
(62)    void think(int seat)    {
(63)            System.out.println("Philosopher #"+Thread.currentThread().getName()+" is thinking");
(64)            try
(65)            {
(66)                    Thread.sleep( (int) (Math.random() * 0) );
(67)            } //try
(68)            catch (InterruptedException ie)
(69)            {
(70)                    System.out.println("An InterruptedException caught\n"+ie.getMessage());
(71)                    ie.printStackTrace();
(72)                    System.exit(1);
(73)            } //catch()
(74)    } //think()
(75)
(76)    void eat(int seat)
(77)    {
(78)            System.out.println("Philosopher #"+Thread.currentThread().getName()+" is eating");
(79)            try
(80)            {
(81)                    Thread.sleep( (int) (Math.random() * 0) );
(82)            } //try
(83)            catch (InterruptedException ie)
(84)            {
(85)                    System.out.println("An InterruptedException caught\n"+ie.getMessage());
(86)                    ie.printStackTrace();
(87)                    System.exit(1);
(88)            } //catch()
(89)            System.out.println("Philosopher #"+Thread.currentThread().getName()+" finished
eating");
(90)    } //eat()
(91)} //class Philosopher
(92)
(93)public class HungryPhilosophers4
(94){
(95)    public static void main(String argv[])
(96)    {
(97)            int hungryPhilosophers = Integer.parseInt(argv[0]);
(98)            Table table = new Table( hungryPhilosophers );
(99)            for (int i=0; i<hungryPhilosophers; i++)
```

```
(100)           {
(101)                   new Thread(new Philosopher(table, i),""+(i+1)).start();
(102)           } //for
(103)   } //main()
(104)} //class HungryPhilosophers4
```

Das Beispiel realisiert eine naive Ressourcenbelegungsstrategie, da es zunächst einen Teil der benötigten Ressourcen (im Beispiel die linke Gabel) reserviert und dann -- unter Erhalt der Sperre -- auf die Verfügbarkeit der weiteren Ressourcen (im Beispiel die rechte Gabel) wartet.
Wurde diese Ressource bereits belegt und tritt zusätzlich eine Anforderung an die gesperrte Ressource des bereits wartenden Threads auf, so tritt der Verklemmungsfall ein.

Eine typische **Programmausgabe**:

```
Philosopher #1 is thinking
Philosopher #3 is thinking
Philosopher #2 is thinking
Philosopher #2 received left fork
Philosopher #1 received left fork
Philosopher #3 received left fork
[Programmstillstand, keine weiteren Ausgaben]
```

Bis zum Eintritt der Verklemmung kann unterschiedlich viel Zeit vergehen und zuvor durchaus eine Reihe korrekter Ressourcenverteilungsvorgänge abgewickelt werden. (Beispiel)

Auch Semaphoren stellen sich damit keineswegegs als Verklemmungsvermeidungsmechanismus dar, sondern zeigen, daß die unsachgemäße Handhabung dieser höheren Synchronisationsprimitive durchaus zu Deadlocksituationen führen kann.

Strategien zur Vermeidung von Verklemmungen

Durch die aufgezeigten Beispiele wird deutlich, daß sich Verklemmungen nicht a priori durch Hochsprachen oder -Übersetzermechanismen erkennen und vermeiden lassen, sondern das sie zumeist aus der fehlerhaften Verwendung der Synchronisationsmechanismen resultieren.
Aus diesem Grunde lassen sich auch keine algorithmischen Strategien zur Aufdeckung von potentiell verklemmungsgefährlichen Situationen formulieren, sondern lediglich bewährte Verhaltensweisen formulieren um Verklemmungen vorzubeugen.

- Anforderung von kritischen Ressourcen sollte ausschließlich dann geschehen, wenn zum Anforderungszeitpunkt keine keine anderen Ressourcensperren gehalten werden.
- Zur Vermeidung zyklischer Wartebedingungen sollten Ressourcen immer in derselben Reihenfolge angefordert werden.
- Erfüllung einer Sperranforderung nur, wenn durch sie keine Verklemmung entstehen kann.
- Einführung eines separaten *Ressourcenverwalters*.

Realisierungen

Das Beispiel 29 zeigt die Umsetzung einer Resourcenverwalterklasse.

Beispiel 29: *Erste Implementierung eines Resourcenverwalters*
Quellcode

```
(1)public class ResourceMgr {
(2)     private SemaphoreGroup resources;
(3)
(4)     public ResourceMgr(int initV[]) {
(5)             resources = new SemaphoreGroup(initV.length);
(6)             resources.changeValues(initV);
(7)     } //constructor
(8)
(9)     public void request(int[] requestedAmount) {
(10)            //add dispatching strategy here!
(11)            int tmp[] = new int[requestedAmount.length];
(12)            for (int i=0; i<requestedAmount.length; i++)
(13)                    tmp[i] = -requestedAmount[i];
(14)            resources.changeValues(tmp);
(15)    } //request()
(16)
```

```
(17)    public void release(int[] releasedAmount) {
(18)            resources.changeValues(releasedAmount);
(19)    } //release()
(20)} //class ResourceMgr
```

Die Implementierung verwendet die Klasse SemaphoreGroup, die das gleichzeitige setzen mehrerer Semaphor-Sperren gestattet.
Jedoch garantiert der Einsatz eines Resourcenverwalters, wie dem des Beispiels, keine Verklemmungsfreiheit. Vielmehr bleibt das ursächliche Problem unverändert bestehen, wie die nachfolgende Umsetzung zeigt:

Beispiel 30: *Verklemmung trotz Resourcenmanager*
Quellcode

```
(1)public class DeadLockWithResourceMgr {
(2)     public static void main(String argv[]) {
(3)             int resources[] = {1,1};
(4)             ResourceMgr rMgr = new ResourceMgr(resources);
(5)             for (int i=0; i<Integer.parseInt(argv[0]); i++)
(6)                     new Thread(new BlockingThreadResourceMgr(rMgr) ).start();
(7)     } //main()
(8)} //class DeadLockWithResourceMgr
(9)
(10)class BlockingThreadResourceMgr implements Runnable {
(11)    ResourceMgr rMgr;
(12)
(13)    public BlockingThreadResourceMgr(ResourceMgr rMgr) {
(14)            this.rMgr = rMgr;
(15)    } //constructor
(16)
(17)    public void run() {
(18)            System.out.println(Thread.currentThread().getName()+" is sleeping");
(19)            try {
(20)                    Thread.sleep( (int) (Math.random() * 0) );
(21)            } //try
(22)            catch (InterruptedException ie) {
(23)                    System.out.println("An InterruptedException caught\n"+ie.getMessage());
(24)                    ie.printStackTrace();
(25)                    System.exit(1);
(26)            } //catch()
(27)            int requestedResource1=(int) (Math.random()*1.1);
(28)            int requestedResource2=Math.abs(requestedResource1-1);
(29)            int resourceV[] = {requestedResource1, requestedResource2};
(30)
(31)            System.out.println(Thread.currentThread().getName()+" requests resources ["+resourceV
[0]+","+resourceV[1]+"]");
(32)            rMgr.request(resourceV);
(33)            System.out.println(Thread.currentThread().getName()+" locks resources ["+resourceV[0]
+","+resourceV[1]+"]");
(34)            try {
(35)                    Thread.sleep( (int) (Math.random() * 0) );
(36)            } //try
(37)            catch (InterruptedException ie) {
(38)                    System.out.println("An InterruptedException caught\n"+ie.getMessage());
(39)                    ie.printStackTrace();
(40)                    System.exit(1);
(41)            } //catch()
(42)
(43)            //deadlock prone section begins
(44)            int otherResourceV[]={requestedResource2, requestedResource1};
(45)            System.out.println(Thread.currentThread().getName()+" requests resources
["+otherResourceV[0]+","+otherResourceV[1]+"]");
(46)            rMgr.request(otherResourceV);
(47)            System.out.println(Thread.currentThread().getName()+" locks resources ["+otherResourceV
[0]+","+otherResourceV[1]+"]");
(48)            try {
(49)                    Thread.sleep( (int) (Math.random() * 0) );
(50)            } //try
(51)            catch (InterruptedException ie) {
```

```
(52)                    System.out.println("An InterruptedException caught\n"+ie.getMessage());
(53)                    ie.printStackTrace();
(54)                    System.exit(1);
(55)            } //catch()
(56)            rMgr.release(otherResourceV);
(57)            System.out.println(Thread.currentThread().getName()+" released resources
["+otherResourceV[0]+","+otherResourceV[1]+"]");
(58)            //deadlock prone section ends
(59)            rMgr.release(resourceV);
(60)            System.out.println(Thread.currentThread().getName()+" released resources ["+resourceV
[0]+","+resourceV[1]+"]");
(61)    } //run()
(62)} //class BlockingThreadResourceMgr
(63)
```

Ausgabe des Programms:

```
Thread-1 is sleeping
Thread-2 is sleeping
Thread-1 requests resources [0,1]
Thread-2 requests resources [1,0]
Thread-1 locks resources [0,1]
Thread-2 locks resources [1,0]
Thread-1 requests resources [1,0]
Thread-2 requests resources [0,1]
[Programmstillstand, keine weiteren Ausgaben]
```

Als Lösung bietet sich die Implementierung einer Resourcenverteilungsstrategie an.
Hierbei können je nach Bedarf die eingangs erwähnten Strategien verwirklicht werden.

Weiterführende Konzepte

Prioritäten

Die Java-Laufzeitumgebung weist jedem Thread bei seiner Erzeugung eine Priorität zu, die während der Laufzeit zur Ermittlung der für zuzuteilenden CPU-Zeit herangezogen werden. Sofern durch den Programmierer nicht anders festgelegt oder beeinflußt erhält jeder neu erzeugte Thread die Priorität seines Elternthreads dessen Priorität sich wiederum nach der Prozeßpriorität der virtuellen Maschine im Betriebsystem richtet.

Einführendes Beispiel :

Beispiel 31: *Threads mit verschiedener Priorität*
Quellcode

```
(1)import java.util.*;
(2)
(3)public class PriorityExample {
(4)/**
(5) * argv[0]: number of increment operations per thread
(6)*/
(7)     public static void main(String argv[]) {
(8)             int maxPri = Thread.currentThread().getThreadGroup().getMaxPriority();
(9)             Thread threads[] = new Thread[maxPri - Thread.MIN_PRIORITY+2];
(10)
(11)            for (int i=Thread.MIN_PRIORITY; i<=maxPri; i++) {
(12)                    threads[i] = new Thread ( new CounterThread( new Counter(Long.parseLong(argv
[0]))));
(13)                    threads[i].setPriority(i);
(14)                    threads[i].setName("pri"+i);
(15)                    } //for
(16)
(17)            for (int i=Thread.MIN_PRIORITY; i<=maxPri; i++)
(18)                    threads[i].start();
(19)    } //main()
(20)} //class PriorityExample
```

```
(21)
(22)class CounterThread implements Runnable {
(23)    private Counter counter;
(24)    private Counter max;
(25)
(26)    public CounterThread(Counter max) {
(27)            this.max = max;
(28)            this.counter = new Counter();
(29)    } //constructor
(30)
(31)    public void run() {
(32)            long start = System.currentTimeMillis();
(33)            while (counter.get() < max.get())
(34)                    counter.increment();
(35)            long end = System.currentTimeMillis() - start;
(36)            System.out.println("Thread with priority "+Thread.currentThread().getPriority()+" took
"+end+"ms");
(37)    } //run
(38)} //class CounterThread
(39)
(40)class Counter {
(41)    private long counter;
(42)
(43)    public void increment() {
(44)            System.out.println("INCREMENT by "+Thread.currentThread().getName());
(45)            ++counter;
(46)    } //incremenet
(47)
(48)    public long get() {
(49)            return counter;
(50)    } //get()
(51)
(52)    public Counter() {
(53)            this(0);
(54)    } //constructor
(55)
(56)    public Counter(long init) {
(57)            counter = init;
(58)    } //constructor
(59)} //class Counter
(60)
```

Das Beispiel erzeugt eine durch den Anwender festlegbare Anzahl nebenläufiger Programmfäden, die anwendergesteuert mit verschiedenen Prioritäten versehen werden können. Jeder Thread inkrementiert einen als Klasse realisierten Zähler bis zur selben (über Kommandozeilenparameter festgelegten) Obergrenze.
Nach Erreichen der Zählgrenze gibt jeder Thread den benötigten Zeitraum und die ihm zugewiesene Priorität aus.

Beobachtungen:

```
$java PriorityExample 10000000
Thread with priority 10 took 110ms
Thread with priority 8 took 109ms
Thread with priority 5 took 313ms
Thread with priority 9 took 344ms
Thread with priority 6 took 110ms
Thread with priority 7 took 109ms
Thread with priority 3 took 500ms
Thread with priority 4 took 188ms
Thread with priority 1 took 141ms
Thread with priority 2 took 140ms

$java PriorityExample 5000000000
Thread with priority 10 took 56219ms
Thread with priority 9 took 86578ms
Thread with priority 8 took 117844ms
Thread with priority 6 took 162438ms
```

```
Thread with priority 7 took 177562ms
Thread with priority 5 took 220375ms
Thread with priority 4 took 245391ms
Thread with priority 3 took 274703ms
Thread with priority 1 took 317985ms
Thread with priority 2 took 340781ms
```

- Die Prioritäten haben (erwartungsgemäß) Einfluß auf die Zuteilung der CPU-Zeit an einen Thread.
- Eine höhere Priorität bedeutet im absoluten Sinne nicht unbedingt eine schnellere Ausführung als eine niedere.
  So benötigt der Thread mit Priorität sechs genausoviel Rechenzeit (110ms) wie der höchstpriore (Priorität 10).
  Bei hinreichend langer Ausführungszeit (zweite Ausgabe) tritt jedoch der erwartete Effekt deutlicher, nicht jedoch in der intuitiv erwarteten Klarheit, zu Tage.

Folgerungen:
Der Zusammenhang zwischen Prioritätsangaben und CPU-Zuteilung ist nicht unbedingt gegeben. Letztlich hängt die Ressourcenzuteilung an einen Thread von Faktoren wie der eingesetzten virtuellen Maschine, aber auch dem ausführenden Betriebsystem sowie der Lastsituation zum Ausführungszeitpunkt ab.
Einsatzempfehlungen sind daher für Prioritäten nur schwer zu geben, eine denkbare Möglichkeit wäre die Klassifizierung von Threads in Hintergrundthreads, die ausschließlich mit Rechenaufgaben beschäftigt sind und im Vordergrund mit dem Benutzer interagierende Programmfäden (z.B. GUI-Elemente). Hier böte sich die Bevorzugung der interaktiven Ausführungseinheiten an.
Keinesfalls sollten Prioritäten zur Umsetzung oder Unterstützung von Wartestrategien oder Synchronisationsbedarfen eingesetzt werden!

Vorgabebemäß bildet die Java-Laufzeitumgebung jeden Java-Thread in einen Thread des ausführenden Betriebsystems ab, sofern diese Primitive dort zur Verfügung steht.
Auf einzelnen Plattformen (o.a. Linux und Solaris) kann die Threadverwaltung vollständig in die virtuelle Maschine verlagert werden. Beim Einsatz dieser -- *green threads* benannten Mimik -- werden die Threads durch die virtuelle Maschine simuliert und auf genau einen singlethreaded Prozeß des Betriebssystems abgebildet.

Im Falle der Transparenz der Threads für das zugrundeliegende Betriebssystem werden auch die javaseitigen Threadprioritäten auf die durch das Betriebssystem angebotenen abgebildet.
Hierbei nehmen auf die tatsächliche Priorität eines betriebssystemseitigen Threads sowohl die Priorität des Prozesses der virtuellen Maschine als auch die Festlegung der Threadpriorität in Java Einfluß.
Die nachfolgende Tabelle stellt die möglichen Kominationen für die SUN Referenzimplementierung J2SE v1.4 (build 92) auf der Win32-Plattform zusammen.

Abbildung 15: *Abbildung der Java Threadprioritäten auf die der Win32 Betriebssystemplattform*

Win32 Prozeßpriorität

Java Threadpriorität	Idle (4)	Below Normal (6)	Normal (8)	Above Normal (10)	High (13)	Realtime (24)
1	2	4	6	8	11	22
2	2	2	6	8	11	22
3	3	5	7	9	12	23
4	3	5	7	9	12	23
5	4	6	8	10	13	24
6	5	7	9	11	14	25
7	5	7	9	11	14	25
8	6	8	10	12	15	26
9	6	8	10	12	15	26
10	15	15	15	15	15	31

(click on image to enlarge!)

Neben der Priorität beeinflußt zusätzlich der *Dämon-Status* das Ausführungsverhalten eines Threads. Zwar wird sich die Charakterisierung als Dämon-Thread nicht zur Laufzeit auf die CPU-Nutzung eines Threads aus, sie ist aber dafür verantwortlich wann ob das Laufzeitende eines Threads vorzeitig (d.h. vor Ende seiner Aufgabe) extern impliziert werden kann. Damit unterscheidet sich der endgültige Entzug der CPU und das Entfernen der Threadstrukturen aus dem Hauptspeicher von der Terminierung mittels `interrupt`, welches den Thread explizit in seiner Ausführung unterbricht.

Definition 14: *Dämon-Thread*
Ein Dämon-Thread ist ein Thread der durch die Java-API-Funktion `setDaemon` vor dessen Ausführungsbeginn als Dämon-Thread ausgezeichnet wurde.
Zum Zeitpunkt der Terminierung eines Threads prüft die virtuelle Maschine ob sich nach dem Ende des aktuellen Threads ausschließlich lauffähige Dämon-Threads im System befinden. Ist dies der Fall, so wird die applikationsausführung beendet, obwohl sich noch (Dämon-)Threads im Zustand lauffähig befinden.

In Abgrenzung zu den Dämon-Threads klassifiziert Java die nicht-Dämon-Threads als *Anwender-Threads*.
Grundidee dieser Unterscheidung ist ein Vorgriff auf die Anwendungsmöglichkeiten dieses Konzeptes. So sollen Dämon-Threads, wegen ihrer unvorhersehbaren Eigenschaften nicht zur Resultatberechnung eingesetzt werden, sondern ausschließlich zur Umsetzung unterstützender Dienstleistungen an andere Threads währen deren Laufzeit.
Aus diesem Grunde setzen einige Implementierung, die den Garbage Collector als eigenständige Thread realisieren diesen als Dämon-Thread um, da seine Dienste nach Ausführungsende der resultatberechnenden Threads nicht mehr benötigt werden.

Die Applikation *ThreadInfo* ermittelt Informationen zu allen für den Anwender sichtbaren Threads innerhalb der Laufzeitumgebung. Wird sie mit dem Parameter `-gui` gestartet, so zeigt sie auch ein AWT-basiertes Fenster um die Erzeugung der Threads für die Verwaltung der interaktiven Abläufe zu initiieren.

Beispiel 32: *Ermittlung threadspezifischer Information*
Quellcode

```
(1)import java.awt.*;
(2)
(3)public class ThreadInfo {
(4)     public static void main(String[] argv) {
(5)             Frame wnd;
(6)             if (argv.length == 1) {
(7)                     if (argv[0].compareTo("-gui") == 0) {
(8)                             wnd = new Frame("Einfaches Fenster");
(9)                             wnd.setSize(400,300);
(10)                    wnd.setVisible(true);
(11)            } //if
(12)      } //if
(13)            ThreadGroup tg = null;
(14)            ThreadGroup tgTmp = Thread.currentThread().getThreadGroup();
(15)
(16)            do {
(17)                    tg = tgTmp;
(18)                    tgTmp = tgTmp.getParent();
(19)            } while (tgTmp != null);
(20)
(21)            int noActiveThreads = tg.activeCount();
(22)
(23)            Thread[] tList = new Thread[noActiveThreads];
(24)
(25)            noActiveThreads = tg.enumerate(tList,true);
(26)            for (int i=0; i<noActiveThreads; i++) {
(27)                    System.out.println("Thread Group: "   +tList[i].getThreadGroup().getName()
+"\n"+
(28)                                               "Thread Name: "    +tList[i].getName()+"\n"+
(29)                                               "Thread Priority: "+tList[i].getPriority()+"\n"+
(30)                                               "isDaemon: "       +tList[i].isDaemon()+"\n");
(31)            } //for
(32)            System.exit(0);
(33)    } // main()
(34)} //class ThreadInfo
```

"

Name	Thread Priorität	Threadtyp	Bemerkung
Reference Handler	10	Dämon	
Finalizer	8	Dämon	Aufgrund der Organisation als Dämon-Thread ist der Aufruf der Methode `finalize` vor dem Applikationsende nicht gewährleistet.
Signal Dispatcher	10	Dämon	
CompileThread0	10	Dämon	
main	5	User	Die standardmäßig zugewiesene Priorität entspricht `NORM_PRIORITY`.

AWT-Shutdown	8	User	Thread wird nur bei Verwendung der AWT-/Swing-Bibliothek erzeugt.
AWT-Windows	6	Dämon	Thread wird nur bei Verwendung der AWT-/Swing-Bibliothek erzeugt.
AWT-EventQueue-0	6	Dämon	Thread wird nur bei Verwendung der AWT-/Swing-Bibliothek für sichtbare Interaktionselemente erzeugt.

Beispiel 33 startet zwei Threads, die beide jeweils einen threadlokalen Zähler inkrementieren. Der als *Daemon* benannte Thread ist als Dämon-Thread deklariert, der andere als Anwenderthread. Durch Kommandozeilenparamter kann der jeweils zu erreichende Zählerhöchststad pro Thread eingestellt werden.

Beispiel 33: *Verhalten eines Dämon-Threads*
Quellcode Ausgabe der Ausführung

```
(1)public class DaemonDemo {
(2)     public static void main(String[] argv) {
(3)             Thread t = new Thread( new CounterThread(new Counter(Long.parseLong(argv
[0]))),"Daemon");
(4)             t.setDaemon(true);
(5)             t.start();
(6)
(7)             new Thread( new CounterThread(new Counter(Long.parseLong(argv[1]))),"Normal").start();
(8)     } //main()
(9)} //class DaemonDemo
(10)
(11)class CounterThread implements Runnable {
(12)    private Counter counter;
(13)    private Counter max;
(14)
(15)    public CounterThread(Counter max) {
(16)            this.max = max;
(17)            this.counter = new Counter();
(18)    } //constructor
(19)
(20)    public void run() {
(21)            long start = System.currentTimeMillis();
(22)            while (counter.get() < max.get()) {
(23)                    counter.increment();
(24)                    System.out.println(Thread.currentThread().getName()+"'s counter state:
"+counter.get() );
(25)            } //while
(26)    } //run
(27)} //class CounterThread
(28)
(29)class Counter {
(30)    private long counter;
(31)
(32)    public void increment() {
(33)            ++counter;
(34)    } //incremenet
(35)
(36)    public long get() {
(37)            return counter;
(38)    } //get()
(39)
(40)    public Counter() {
(41)            this(0);
(42)    } //constructor
(43)
(44)    public Counter(long init) {
(45)            counter = init;
(46)    } //constructor
(47)} //class Counter
```

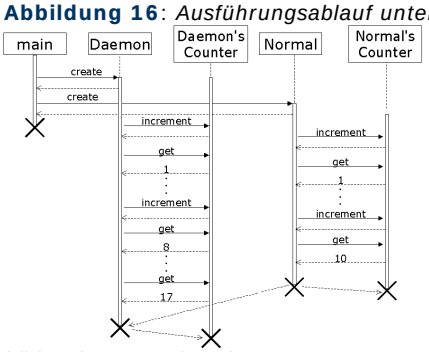
Eine mögliche **Ausgabe**:

```
$java DaemonDemo 100 10
Daemon's counter state: 1
```

```
Daemon's counter state: 2
Daemon's counter state: 3
Daemon's counter state: 4
Daemon's counter state: 5
Daemon's counter state: 6
Daemon's counter state: 7
Daemon's counter state: 8
Normal's counter state: 1
Normal's counter state: 2
Normal's counter state: 3
Normal's counter state: 4
Normal's counter state: 5
Normal's counter state: 6
Normal's counter state: 7
Normal's counter state: 8
Normal's counter state: 9
Normal's counter state: 10
Daemon's counter state: 9
Daemon's counter state: 10
Daemon's counter state: 11
Daemon's counter state: 12
Daemon's counter state: 13
Daemon's counter state: 14
Daemon's counter state: 15
Daemon's counter state: 16
Daemon's counter state: 17
```

Beobachtung: Beide Zählthreads inkrementieren ihre Zähler nebenläufig, solange bis der Anwenderthread seinen Höchststand erreicht und terminiert. Nachdem der aus der main-Methode resultierende Hauptausführungsthread (ebenfalls ein Anwenderthread) bereits nach Abarbeitung aller seiner Anweisungen (konkret: der Erzeugung der beiden Threads) terminierte existiert zu diesem Zeitpunkt kein Anwender-Thread mehr im System und die Applikation wird bei der nächsten Prozessorzuteilung an den Thread *Normal* beendet unabhängig davon in welchem Ausführungszustand sich der Dämon-Thread befindet.

Abbildung 16: *Ausführungsablauf unter Einsatz eines Dämon-Threads*



(click on image to enlarge!)

Abbildung 16 zeigt verkürzt einen dynamischen Ausführungsablauf des Beispielprogramms. Auffallend ist insbesondere, daß die Applikation nicht unverzüglich nach dem Erreichen des Zählerhöchststandes durch den Anwenderthread *Normal* terminiert, sondern dies erst nach der der Neuzuteilung der CPU erfolgt. Während dieses Zeitraumes wird der Dämon-Thread weiter ausgeführt.

Scheduling

Ausgehend von den zugeordneten Prioritäten teilt die Java-Laufzeitumgebung jedem Thread im sog. *Schedulingvorgang* Rechenzeit zu.
Die interne Realisierung dieses Zuteilungsschemas ist nicht einheitlich spezifiziert und kann daher zwischen verschiedenen virtuellen Maschinen variieren.

Üblicherweise wird ein prioritätsgesteuertes Round-Robin-Verfahren verwirklicht, das zumeist durch einen preemptiven Scheduler ausgeführt wird. Dies läßt sich an den Ausgaben der bisher angegebenen Beispiele ablesen, da in ihnen der Entzug der CPU auch im Zustand *laufend* eintritt. Es sind jedoch auch virtuelle Maschinen verfügbar, die kooperatives Scheduling zugrunde legen. In diesen Umgebungen muß durch den Programmierer sorge für die Abgabe der Zeitscheibe während rechenintesiver Vorgänge getragen werden. Tritt während des Programmablaufes keine Wartebedingung (etwa auch E/A-Operationen, `sleep`-Aufrufe oder `wait`-Aufrufe) ein, so muß zur Threadumschaltung in regelmäßigen Abständen die API-Methode `yield` aufgerufen werden.

Ihr Aufruf in preemptiven Umgebungen wirkt sich ebenfalls auf das Umschaltverhalten aus und läßt im allgemeinenenen flüssigeren Eindruck der Nebenläufigkeit entstehen.

Beispiel 34: *Verwendung der Methode yield*
Quellcode Ausgabe der Ausführung

```
(1)public class DaemonDemo {
(2)     public static void main(String[] argv) {
(3)             Thread t = new Thread( new CounterThread(new Counter(Long.parseLong(argv
[0]))),"Daemon");
(4)             t.setDaemon(true);
(5)             t.start();
(6)
(7)             new Thread( new CounterThread(new Counter(Long.parseLong(argv[1]))),"Normal").start();
(8)     } //main()
(9)} //class DaemonDemo
(10)
(11)class CounterThread implements Runnable {
(12)    private Counter counter;
(13)    private Counter max;
(14)
(15)    public CounterThread(Counter max) {
(16)            this.max = max;
(17)            this.counter = new Counter();
(18)    } //constructor
(19)
(20)    public void run() {
(21)            long start = System.currentTimeMillis();
(22)            while (counter.get() < max.get()) {
(23)                    counter.increment();
(24)                    System.out.println(Thread.currentThread().getName()+"'s counter state:
"+counter.get() );
(25)                    Thread.yield();
(26)            } //while
(27)    } //run
(28)} //class CounterThread
(29)
(30)class Counter {
(31)    private long counter;
(32)
(33)    public void increment() {
(34)            ++counter;
(35)    } //incremenet
(36)
(37)    public long get() {
(38)            return counter;
(39)    } //get()
(40)
(41)    public Counter() {
(42)            this(0);
(43)    } //constructor
(44)
(45)    public Counter(long init) {
(46)            counter = init;
(47)    } //constructor
(48)} //class Counter
```

Beispiel 34 illustriert Verwendung und Auswirkung der `yield`-Methode. Durch den expliziten Aufruf nach Ausgabe des Zählerstandes wird durch den Thread selbst der Übergang in den Zustand *nicht laufend* herbeigeführt und so in anderer rechenbereiter Thread zur Ausführung gebracht.

Thread Pooling

Grundidee

- Anwendergesteurte Ressourcenkontrolle und -zuteilung.
  Aus Gründen der Lastverteilung kann es sinnvoll sein die Anzahl der zu einem Zeitpunkt der zugelassenen nebenläufigen Programmfäden zu begrenzen.

- Erreichung positiver Laufzeitaspekte.
  Aus der Vorhaltung von Threads kann sich, insbesondere bei einer großen Anzahl kurzlaufender Threads ein Geschwindigkeisvorteil ergeben.

Funktionelle Charakteristika der Lösung:

- Erzeugung einer statischen Anzahl von Threads, die auf Aufgaben warten.
- Verwaltung dieser Threads in einem Ressourcen-Pool.
- Bei Eintreffen einer neuen Aufgabe erfolgt die Entnahme eines vorbereiteten Threads aus diesem Pool und Zuweisung der Aufgabe.
- Ist kein Thread im Pool mehr verfügbar, so wird die anstehende Aufgabe solange verzögert, bis dies wieder der Fall ist.
- Nach Abarbeitung der Aufgabe wird der Thread zurück in den Pool gestellt.

Technische Charakteristika der Lösung:

- Realisierung mit Java-Threads.
- Verwaltung des Thread-Pools mit den Möglichkeiten der Java Collection API.
- Alle Aufgaben realisieren die Schnittstelle `Runnable`.
- Die Threadverwaltung ist selbst als nebenläufiger Thread realisiert.

Implementierung:

... siehe Vorlesungsergebnisse

Anwendungsbeispiele

Algorithmische Aspekte

Dieses Teilkapitel stellt exemplarisch an der Modifikation des bekannten Sortierverfahren *Quick Sort* einige Schritte und Überlegungen dar, die bei der Umsetzung nicht-nebenläufiger Algorithmen in nebenläufige Analoga eine Rolle spielen.

Um den folgenden Bemühungen eines gleich vorauszuschicken: Der zu erwartende Geschindigkeitsgewinn hält sich trotz der hervorragenden Nebenläufigkeitseigenschaften von Quick Sort in sehr engen Grenzen.
Tests aus der Praxis zeigen, daß er in vielen Fällen sogar zu vernachlässigen ist, ja mehr noch -- sogar negative Effekte zeigt --, und daher zumeist unterbleiben kann.
Dennoch eignet sich Quick Sort wegen seiner einfachen Struktur und der bekannten und gut erforschten Eigenschaften als Studienobjekt.

Ausgangssituation:

Beispiel 35: *Sequentielle Quick Sort-Implementierung*
Quellcode

```
(1)public class QuickSort {
(2)     public static void main(String argv[]) {
(3)             int limit = Integer.parseInt(argv[0]);
(4)             int i[] = new int[limit];
(5)             for (int c=0; c<limit; c++)
(6)                     i[c]=(int) (Math.random()*limit);
(7)
(8)             displayArray(i,0,i.length-1);
(9)             quicksort(i,0,i.length-1);
(10)            displayArray(i,0,i.length-1);
(11)    } //main()
(12)
(13)    public static void quicksort(int[] a, int l, int r) {
(14)            int i=l;
(15)    int j=r;
(16)    int pivot = a[(int) ((l+r)/2)];
(17)       do {
(18)            while(a[i] < pivot)
(19)                    i++;
(20)          while(pivot < a[j])
(21)            j--;
(22)          if(i<j)
```

```
(23)            swap(a,i,j);
(24)            if (i<=j){
(25)                                    i++;
(26)            j--;
(27)                    } //if
(28)            } while (i<j);
(29)       if (l<j)
(30)            quicksort(a,l,j);
(31)       if (i<r)
(32)            quicksort(a,i,r);
(33)    } //quicksort()
(34)
(35)    private static void swap(int[] a, int s, int d) {
(36)    int temp = a[s];
(37)       a[d] = a[s];
(38)       a[s] = temp;
(39)    } //swap()
(40)
(41)    private static void displayArray(int[] a, int l, int r) {
(42)    for(int i=l;i<r-l+1;i++)
(43)            System.out.print(a[i]+" ");
(44)            System.out.println();
(45)    } //displayArray()
(46)} //class QuickSort
```

Beispiel 35 zeigt eine sequentielle (d.h. nicht nebenläufige) rekursive Implementierungsvariante von Quick Sort.
Als Pivot-Element, dasjenige bezüglich dem die übrigen Elemente einer Sortierparition umzugruppieren sind ist statisch auf die rechnerische Mitte der Partition festgelegt.
Ferner vermeidet die angegebene Implementierung unnötige Verauschungsoperationen und Rekursionsschritte.

Einführung von Nebenläufigkeit:

Im allgemeinen kommt drei Einzelaspekten eine zentrale Rolle bei der Einführung von Nebenläufigkeit zu:

- Ermöglichung des Zugriff auf gemeinsame Datenbereiche
- Synchronisation
- Organisation der Nebenläufigkeit

Beispiel 36: *Nebenläufige Quick Sort-Implementierung*
Quellcode ... auf einer Multiprozessormaschine

```
(1)public class cQuickSort {
(2)     public static void main(String argv[]) {
(3)             int limit = Integer.parseInt(argv[0]);
(4)             int i[] = new int[limit];
(5)
(6)             int noThreadsStartup = Thread.currentThread().getThreadGroup().activeCount();
(7)
(8)             for (int c=0; c<limit; c++)
(9)                     i[c]=(int) (Math.random()*limit);
(10)
(11)            displayArray(i,0,i.length-1);
(12)
(13)            new Thread(new QuickSort(i,0,i.length-1)).start();
(14)
(15)            while (Thread.currentThread().getThreadGroup().activeCount() > noThreadsStartup);
(16)
(17)            displayArray(i,0,i.length-1);
(18)    } //main()
(19)
(20)    private static void displayArray(int[] a, int l, int r) {
(21)            for(int i=l;i<r-l+1;i++)
(22)            System.out.print(a[i]+" ");
(23)            System.out.println();
(24)    } //displayArray()
(25)} //class cQuickSort
(26)
```

```
(27)class QuickSort implements Runnable {
(28)    int a[];
(29)    int l;
(30)    int r;
(31)    public QuickSort(int[] a, int l, int r) {
(32)            this.a = a;
(33)            this.l = l;
(34)            this.r = r;
(35)    } //constructor
(36)
(37)    private static void swap(int[] a, int s, int d) {
(38)            int tmp = a[s];
(39)            a[s] = a[d];
(40)            a[d] = tmp;
(41)    } //swap();
(42)
(43)    public void run() {
(44)            System.out.println(Thread.currentThread().getName()+" l="+l+", r="+r);
(45)      int i=l;
(46)      int j=r;
(47)      int k = (int) ((l+r)/2);
(48)      System.out.println(Thread.currentThread().getName()+" pivot="+k+" (value="+a[k]+")");
(49)      int pivot = a[k];
(50)      do {
(51)        while(a[i] < pivot)
(52)          i++;
(53)        while(pivot < a[j])
(54)          j--;
(55)        if(i<j && a[i] != a[j]) {
(56)          System.out.println(Thread.currentThread().getName()+" exchanging a["+i+"] (="+a[i]+")
and a["+j+"] (="+a[j]+")");
(57)          swap(a,i,j);
(58)        } //if
(59)        if (i<=j) {
(60)                      i++;
(61)          j--;
(62)              } //if
(63)      } while (i<j);
(64)
(65)      if (l<j) {
(66)            System.out.println(Thread.currentThread().getName()+" child: l="+l+", r="+j);
(67)            new Thread(new QuickSort(a,l,j)).start();
(68)      } //if
(69)      if (i<r) {
(70)            System.out.println(Thread.currentThread().getName()+" child: l="+i+", r="+r);
(71)            new Thread(new QuickSort(a,i,r)).start();
(72)      } //if
(73)   } //run
(74)} //class QuickSort
```

Die threadbasierte nebenläufige Implementierung ersetzt zunächst die beiden Rekursionschritte der Zeilen 30 und 32 durch Threaderzeugungen. Ferner wird auch der erste Aufruf des Algorithmus als Ausführung eines Threads umgesetzt.

Neben dem javaspezifischen Problem der Organisation der Parameterübergabe durch die Konstruktormethode statt einer direkten Möglichkeit gewinnt der Aspekt der evtl. zu leistenden Synchronisation zwischen den Einzelthreads besonderes Gewicht.
Hierbei sind zunächst durch Analyse der durch jeden Thread zugegriffenen Datenbereiche die potentiell gleichzeitig im konkurrierenden Zugriff befindlichen Ressourcen zu ermitteln.

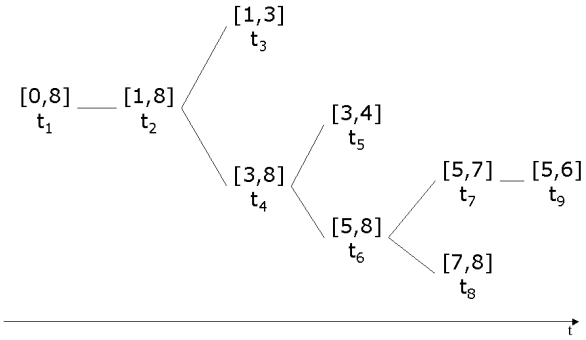
Abbildung 17 illustriert die durch die Einzelthreads verarbeiteten Feldelemente und stellt zusätzlich die Erzeugungsstruktur der Threads im zeitlichen Verlauf dar.

Auffallend hierbei ist, daß konkurrierende Datenzugriffe -- bedingt durch die Struktur des Quick Sort Algorithmus -- ausschließlich durch Kindthreads eines gegebenen Threads erfolgen.

Eine Synchronisation müßte daher lediglich zwischen Threads und ihren Abkömmlingen durchgeführt werden.

Durch die Struktur der Algorithmus wird jedoch klar, daß auch dies nicht zu erfolgen braucht, da die gesamten modifizierenden Datenzugriffe in Zeile 57 erfolgen und demnach *vor* der Erzeugung der Kindthreads (in Zeile 67 bzw. 71) abgeschlossen sind.
Eine explizite Synchronisation muß daher im vorliegenden Falle nicht erfolgen.

Abbildung 17: *Erzeugungsgraph der einzelnen Threads*



(click on image to enlarge!)

Graphische Benutzeroberflächen

Wie bereits in der Threadübersichtstabelle gezeigt verwendet das *Abstract Windowing Toolkit* und die Oberflächenbibliothek *Swing* bereits intern eigene Threads.
Nachfolgend werden die Möglichkeiten des Thread-Einsatzes im Umfeld graphischer Benutzeroberflächen eingeführt.

Beispiel 37: *Einfach Swing-Applikation*
Quellcode

```
(1)import javax.swing.*;
(2)
(3)public class HelloSwing {
(4)     public static void main(String argv[]) {
(5)             JFrame f = new JFrame ("Simple Swing Window");
(6)             f.getContentPane().add(new JLabel("Hello Swing"));
(7)             f.setSize(400,100);
(8)             f.setLocation(30,50);
(9)             f.setVisible(true);
(10)            System.out.print("main() finished");
(11)    } //main()
(12)} //class HelloSwing
```

Das Beispiel 37 verwendet im angegebenen Code keine (anwenderdefinierten) Threads.
Dennoch deutet das Verhalten der Applikation, insbesondere die fortgesetzte Ausführung obwohl die `main`-Methode abgearbeitet wurde, auf die Existenz von (nicht-Daemon) Threads hin.

Das folgende Beispiel ermittelt durch welchen Thread die Abarbeitung eines Ereignisses (z.B. Mausklick auf Button) erfolgt:

Beispiel 38: *Ermittlung des Ereignisbehandlungssthreads*
Quellcode

```
(1)import java.awt.*;
(2)import java.awt.event.*;
(3)import javax.swing.*;
(4)
(5)public class sSwing2 {
(6)     public static void main(String argv[]) {
(7)             JFrame f = new JFrame("simple Swing");
(8)             f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
(9)             Container container = f.getContentPane();
(10)            container.setLayout(new GridLayout(0, 1));
(11)            JButton b1 = new JButton("click me!");
(12)            container.add(b1);
(13)            JButton b2 = new JButton(".. or me!");
(14)            container.add(b2);
(15)
(16)            myListener myL = new myListener();
```

```
(17)
(18)            b1.addActionListener( myL );
(19)            b2.addActionListener( myL );
(20)
(21)            f.setLocation(300, 50);
(22)            f.setSize(150, 200);
(23)            f.setVisible(true);
(24)    } //main()
(25)} //class sSwing2
(26)
(27)class myListener implements ActionListener {
(28)    public void actionPerformed(ActionEvent e) {
(29)            System.out.println("Action handled by thread "+Thread.currentThread().getName() );
(30)    } //actionPerformed()
(31)} //class myListener
(32)
```

Die Anwendung liefert bei jedem Drücken einer der beiden Schaltflächen dieselbe Ausgabe `Action handled by thread AWT-EventQueue-0`.
Bei der *Event Queue* handelt es sich um einen Thread der nebenläufig die Verarbeitung der eingetretenen Ereignisse einer GUI-basierten Swing-Applikation übernimmt. Zur Verwaltung ausgelöster, jedoch noch nicht abgearbeiteter Ereignisse wird intern eine FIFO-Warteschlange herangezogen.

Am Beispiel fällt auf, daß alle Ereignisse offensichtlich durch denselben Thread verarbeitet werden. Dies läßt sich zusätzlich leicht prüfen indem künstlich Ausführungszeit der Behandlungsroutine eines Schaltflächenereignisses, etwa durch den Einbau einer Wartebedingung, erhöht wird.
In diesem Falle werden die gesamten Interaktionsmöglichkeiten der Applikation bis zum Ende der Ereignisbehandlungsroutine blockiert.
Beispiel 39 welches das vorangegangene geeignet modifiziert zeigt dies:

Beispiel 39: *Zeitintensive Ereignisbehandlung*
Quellcode

```
(1)import java.awt.*;
(2)import java.awt.event.*;
(3)import javax.swing.*;
(4)
(5)public class sSwing21 {
(6)     public static void main(String argv[]) {
(7)             JFrame f = new JFrame("simple Swing");
(8)             f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
(9)             Container container = f.getContentPane();
(10)            container.setLayout(new GridLayout(0, 1));
(11)            JButton b1 = new JButton("click me!");
(12)            container.add(b1);
(13)            JButton b2 = new JButton(".. or me!");
(14)            container.add(b2);
(15)
(16)            myListener myL = new myListener();
(17)
(18)            b1.addActionListener( myL );
(19)            b2.addActionListener( myL );
(20)
(21)            f.setLocation(300, 50);
(22)            f.setSize(150, 200);
(23)            f.setVisible(true);
(24)    } //main()
(25)} //class sSwing21
(26)
(27)class myListener implements ActionListener {
(28)    public void actionPerformed(ActionEvent e) {
(29)            System.out.println("Action handling started");
(30)            try {
(31)                    Thread.sleep(5000);
(32)            } catch (InterruptedException ie) {
(33)                    System.out.println("an InterruptedException occurred\n"+ie.toString()+"\n"+ie.
getMessage() );
(34)            } //catch
```

```
(35)            System.out.println("Action handling ended");
(36)    } //actionPerformed()
(37)} //class myListener
(38)
```

Aus diesem Verhalten folgt, daß zeitintensive Behandlungen -- wie etwa langwierige Initialisierungsphasen -- nicht direkt in der Ereignisbehandlungsroutine erfolgen sollten, da hierdurch die gesamte Applikation in ihrem Interaktionsverhalten negativ beeinflußt wird. Sinnvollerweise sollten hierzu anwenderdefinierte und -kontrollierte Threads zum Einsatz kommen.

Einsatz von anwenderdefinierten Threads in Swing:

Definition 15: *Single Thread Rule*
Alle Operationen, die vom Status einer visuellen Swingkomponente abhängen oder diesen verändern sollten ausschließlich innerhalb des Ereignisbearbeitungs-Threads (*event-dispatching thread*) ausgeführt werden.

Die Definition 15 liefert die Begründung weshalb die Threasicherheit nur für die wenigsten Methoden der Swing API gewährleistet ist.
Konzeptionell werden Threads innerhalb der Swing-basierten Oberflächenprogrammierung ausschließlich durch Methoden der API erzeugt und gesteuert. Dem Programmierer stehen die Mechanismen der Nebenläufigkeit lediglich in Form von anwendungsspezifisch abstrahierten Klassen und Methoden (z.B. `Timer`) zur Verfügung.
Gleichzeitig liefert die *Single Thread Rule* aber auch eine Handreichung für den Einsatz anwenderdefinierter Threads in Swing-Applikationen. Da alle durch Benutzerinteraktion implizierten Ereignisbehandlungen ausschließlich durch den Ereignisbehandlungs-Thread verarbeitet werden sollen liefert dieser auch einen natürlichen Ansatzpunkt zur Erzeugung eigener Threads.

Beispiel 40: *Ereignisbehandlung durch Threads*
Quellcode

```
(1)import java.awt.*;
(2)import java.awt.event.*;
(3)import javax.swing.*;
(4)
(5)public class sSwing3 {
(6)     public static void main(String argv[]) {
(7)             JFrame f = new JFrame("simple Swing");
(8)             f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
(9)             Container container = f.getContentPane();
(10)            container.setLayout(new GridLayout(0, 1));
(11)            JButton b1 = new JButton("click me!");
(12)            container.add(b1);
(13)            JButton b2 = new JButton(".. or me!");
(14)            container.add(b2);
(15)
(16)            myListener myL = new myListener();
(17)
(18)            b1.addActionListener( myL );
(19)            b2.addActionListener( myL );
(20)
(21)            f.setLocation(300, 50);
(22)            f.setSize(150, 200);
(23)            f.setVisible(true);
(24)    } //main()
(25)} //class sSwing3
(26)
(27)class myListener implements ActionListener {
(28)    public void actionPerformed(ActionEvent e) {
(29)            System.out.println("Command="+e.getActionCommand());
(30)            new Thread(new myHandlingThread()).start();
(31)    } //actionPerformed()
(32)} //class myListener
(33)
(34)class myHandlingThread implements Runnable {
(35)    public void run() {
(36)            System.out.println("Action handling done by thread "+Thread.currentThread().getName()
+" started ..." );
```

```
(37)            try {
(38)                    Thread.sleep(5000);
(39)            } catch (Exception e) {}
(40)            System.out.println("Action handling done by thread "+Thread.currentThread().getName()
+" ... ended" );
(41)    } //run()
(42)} //class myHandlingThread
```

Beispiel 40 erweitert das vorangegangene Beispiel in Zeile 30 geringfügig um die Ereignisbehandlung durch einen anwenderdefinierten Thread erfolgen zu lassen.
Hierdurch wird die tatsächliche Realisierung der Behandlung vom Aufruf der Behandlungsroutine durch den Ereingisbehandlungs-Thread entkoppelt.
Dies zeigt sich im Falle des Beispiels durch ein deutlich reaktiveres Verhalten des Gesamtapplikation obwohl immer noch unverändert fünf Sekunden für die Ereignisverarbeitung benötigt werden.
Der Preis dieses Verhaltens liegt in den zu veranschlagenden Ressourcen in Laufzeit (Threaderzeugung) und Speicher (Threadstrukturen).

Gleichzeitig tritt jedoch mit der aus Anwendersicht wünschenswerten Entkopplung auch ein zusätzliches Problem zu Tage: Die realisierbare Nebenläufigkeit in der Ereignisbehandlung kann sich bei schreibenden Zugriffen auf visuelle Elemente der Oberfläche negativ auswirken, da diese Schreiboperationen generell unsynchronisiert erfolgen.

Das Beispiel 41 zeigt eine fehlerhafte Ereignisbehandlung in deren Verlauf einzelne Ereignisse „verlorengehen“, d.h. ihre Veränderungen an der GUI durch zeitlich nachgelagerte Ereignisse überschrieben werden.

Beispiel 41: *Fehlerhafte Ereignisbehandlung durch Threads*
Quellcode

```
(1)import java.awt.*;
(2)import java.awt.event.*;
(3)import javax.swing.*;
(4)
(5)public class sSwing4 {
(6)     public static void main(String argv[]) {
(7)             JFrame frm = new JFrame("simple Swing");
(8)             frm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
(9)             Container container = frm.getContentPane();
(10)            container.setLayout(new GridLayout(0, 1));
(11)            JButton b1 = new JButton("add one now!");
(12)            container.add(b1);
(13)            JButton b2 = new JButton("add ten every second");
(14)            container.add(b2);
(15)            JTextField fld = new JTextField("0", 1);
(16)            fld.setEditable(false);
(17)            container.add(fld);
(18)
(19)            b1.addActionListener( new addOne(fld) );
(20)            b2.addActionListener( new addTenPerSecond(fld) );
(21)
(22)            frm.setLocation(300, 50);
(23)            frm.setSize(150, 200);
(24)            frm.setVisible(true);
(25)    } //main()
(26)} //class sSwing4
(27)
(28)class addOne implements ActionListener {
(29)    private JTextField fld;
(30)    public addOne(JTextField fld) {
(31)            this.fld = fld;
(32)    } //constructor
(33)
(34)    public void actionPerformed(ActionEvent e) {
(35)            int value = Integer.parseInt(fld.getText());
(36)            //simulating some complex operation here
(37)            try {
(38)                    Thread.sleep(500);
(39)            } catch (InterruptedException ie) {
(40)                    System.out.println("an InterruptedException occurred\n"+ie.toString()+"\n"+ie.
```

```
getMessage() );
(41)            } //catch
(42)            value++;
(43)            fld.setText(""+value);
(44)    } //actionPerformed()
(45)} //class addOne
(46)
(47)class addTenPerSecond implements ActionListener {
(48)    private JTextField fld;
(49)    public addTenPerSecond(JTextField fld) {
(50)            this.fld = fld;
(51)    } //constructor
(52)    public void actionPerformed(ActionEvent e) {
(53)            new Thread(new addTen(fld) ).start();
(54)    } //actionPerformed()
(55)} //class addOne
(56)
(57)class addTen implements Runnable {
(58)    private JTextField fld;
(59)    public addTen(JTextField fld) {
(60)            this.fld = fld;
(61)    } //constructor
(62)
(63)    public void run() {
(64)            while (true) {
(65)                    int value = Integer.parseInt(fld.getText());
(66)                    //simulating some complex operation here
(67)                    try {
(68)                            Thread.sleep(500);
(69)                    } catch (InterruptedException ie) {
(70)                            System.out.println("an InterruptedException occurred\n"+ie.toString()
+"\n"+ie.getMessage() );
(71)                    } //catch
(72)                    value+=10;
(73)                    fld.setText(""+value);
(74)                    try {
(75)                            Thread.sleep(1000);
(76)                    } catch (InterruptedException ie) {
(77)                            System.out.println("an InterruptedException occurred\n"+ie.toString()
+"\n"+ie.getMessage() );
(78)                    } //catch
(79)            } //while
(80)    } //actionPerformed()
(81)} //class addTen
```

Definition 16: *UI Element Access Rule*
Operationen die den Zustand visueller Elemente ändern werden ausschließlich aus dem Ereignisbehandlungs-Thread heraus ausgeführt und verwenden hierzu ausschließlich die dafür vorgesehen Methoden `invokeLater` oder `invokeAndWait`.

Diese beiden Methoden plazieren ein Ereignisobjekt in der Behandlungswarteschlage Wartschlange, wobei `invokeLater` als asynchroner Aufruf sofort nach dem Erzeugen und Einstellen des Objekts zurückkehrt, und `invokeAndWait` synchron auf die Ausführung des erzeugten Ereignisses wartet.

Die beiden Methoden erwarten beide als Eingabeparameter ein Objekt welches konform zur Schnittstelle `Runnable` implementiert ist.
Die `run`-Methode dieses Objekts wird im Zuge der Swing-internen Ereignisbehandlung durch den Ereignisbehandlungs-Thread ausgeführt, sobald das Objekt aus der Ereigniswarteschlange entnommen wurde.

Beispiel 42 zeigt die korrekte Ereignisbehandlung durch die Definition eines eigenen Ereignisobjektes ab Zeile 79.

Beispiel 42: *Korrekte nebenläufige Ereignisbehandlung*
Quellcode

```
(1)import java.awt.*;
(2)import java.awt.event.*;
```

```
(3)import javax.swing.*;
(4)
(5)public class sSwing5 {
(6)     public static void main(String argv[]) {
(7)             JFrame frm = new JFrame("simple Swing");
(8)             frm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
(9)             Container container = frm.getContentPane();
(10)            container.setLayout(new GridLayout(0, 1));
(11)            JButton b1 = new JButton("add one now!");
(12)            container.add(b1);
(13)            JButton b2 = new JButton("add ten every second");
(14)            container.add(b2);
(15)            JTextField fld = new JTextField("0", 1);
(16)            fld.setEditable(false);
(17)            container.add(fld);
(18)
(19)            b1.addActionListener( new addOne(fld) );
(20)            b2.addActionListener( new addTenPerSecond(fld) );
(21)
(22)            frm.setLocation(300, 50);
(23)            frm.setSize(150, 200);
(24)            frm.setVisible(true);
(25)    } //main()
(26)} //class sSwing5
(27)
(28)class addOne implements ActionListener {
(29)    private JTextField fld;
(30)    public addOne(JTextField fld) {
(31)            this.fld = fld;
(32)    } //constructor
(33)
(34)    public void actionPerformed(ActionEvent e) {
(35)            //simulating some complex operation here
(36)            try {
(37)                    Thread.sleep(500);
(38)            } catch (InterruptedException ie) {
(39)                    System.out.println("an InterruptedException occurred\n"+ie.toString()+"\n"+ie.
getMessage() );
(40)            } //catch
(41)            SwingUtilities.invokeLater( new AddEvent(fld, 1) );
(42)    } //actionPerformed()
(43)} //class addOne
(44)
(45)class addTenPerSecond implements ActionListener {
(46)    private JTextField fld;
(47)    public addTenPerSecond(JTextField fld) {
(48)            this.fld = fld;
(49)    } //constructor
(50)    public void actionPerformed(ActionEvent e) {
(51)            new Thread(new addTen(fld) ).start();
(52)    } //actionPerformed()
(53)} //class addOne
(54)
(55)class addTen implements Runnable {
(56)    private JTextField fld;
(57)    public addTen(JTextField fld) {
(58)            this.fld = fld;
(59)    } //constructor
(60)
(61)    public void run() {
(62)            while (true) {
(63)                    //simulating some complex operation here
(64)                    try {
(65)                            Thread.sleep(500);
(66)                    } catch (InterruptedException ie) {
(67)                            System.out.println("an InterruptedException occurred\n"+ie.toString()
+"\n"+ie.getMessage() );
(68)                    } //catch
(69)                    SwingUtilities.invokeLater( new AddEvent(fld, 10) );
```

```
(70)                    try {
(71)                            Thread.sleep(1000);
(72)                    } catch (InterruptedException ie) {
(73)                            System.out.println("an InterruptedException occurred\n"+ie.toString()
+"\n"+ie.getMessage() );
(74)                    } //catch
(75)            } //while
(76)    } //actionPerformed()
(77)} //class addTen
(78)
(79)class AddEvent implements Runnable {
(80)    private JTextField fld;
(81)    private int increment;
(82)
(83)    public AddEvent(JTextField fld, int inc) {
(84)            this.fld = fld;
(85)            increment = inc;
(86)    } //constructor
(87)
(88)    public void run() {
(89)            fld.setText( ""+(Integer.parseInt(fld.getText())+increment) );
(90)    } //run()
(91)} //class AddEvent
```

Netz-spezifisches ...

... Hier nur zur Ergänzung zusammengestellt, da seit Java2 Version 1.4 nicht mehr originäres Thread-Thema, sondern durch die *New IO* anderweitig lösbar.

Beispiel 43: *Nicht nebenläufige Verarbeitung empfangenen Daten*
Quellcode

```
(1)import java.net.*;
(2)import java.io.*;
(3)
(4)public class Net1 {
(5)     public static void main(String argv[]) {
(6)             ServerSocket servSock1 = null;
(7)             Socket sock1 = null;
(8)             InputStream in = null;
(9)             InputStreamReader isr = null;
(10)
(11)            try {
(12)                    servSock1 = new ServerSocket(80);
(13)                    sock1 = servSock1.accept();
(14)                    in = sock1.getInputStream();
(15)                    isr = new InputStreamReader(in);
(16)
(17)                    int c;
(18)                    while ( (c = isr.read()) != -1 ) {
(19)                            System.out.print((char) c);
(20)                    } //while
(21)            } catch (IOException io) {
(22)                    System.out.println("an IOException occurred\n"+io.toString()+"\n"+io.getMessage
() );
(23)            } //catch
(24)    } //main()
(25)} //class Net1
```

Beispiel 44: *Einfache nebenläufige Verarbeitung von empfangenen Daten*
Quellcode

```
(1)import java.net.*;
(2)import java.io.*;
(3)
(4)public class Net2 {
(5)     public static void main(String argv[]) {
(6)             new Thread( new PortEcho(80) ).start();
```

```
(7)             new Thread( new PortEcho(81) ).start();
(8)     } //main()
(9)} //class Net2
(10)
(11)class PortEcho implements Runnable {
(12)    private int port;
(13)    private ServerSocket servSock1 = null;
(14)    private Socket sock1 = null;
(15)    private InputStream in = null;
(16)    private InputStreamReader isr = null;
(17)
(18)    public PortEcho (int port) {
(19)            this.port = port;
(20)            try {
(21)                    servSock1 = new ServerSocket(port);
(22)                    sock1 = servSock1.accept();
(23)                    System.out.println("listening at port "+port);
(24)            } catch (IOException io) {
(25)                    System.out.println("an IOException occurred\n"+io.toString()+"\n"+io.getMessage
() );
(26)            } //catch
(27)    } //constructor
(28)    public void run() {
(29)            try {
(30)                    in = sock1.getInputStream();
(31)                    isr = new InputStreamReader(in);
(32)
(33)                    int c;
(34)                    while ( (c = isr.read()) != -1 ) {
(35)                            System.out.println("("+port+") " + (char) c);
(36)                    } //while
(37)            } catch (IOException io) {
(38)                    System.out.println("an IOException occurred\n"+io.toString()+"\n"+io.getMessage
() );
(39)            } //catch
(40)    } //run()
(41)} //class PortEcho
(42)
```

Beispiel 45: *Ein einfacher TCP-Server der pro eingehender Verbindung einen Thread startet*
Quellcode

```
(1)import java.net.*;
(2)import java.io.*;
(3)
(4)public class TCPServer implements Cloneable, Runnable {
(5)     Thread runner = null;
(6)     ServerSocket server = null;
(7)     Socket data = null;
(8)     volatile boolean shouldStop = false;
(9)
(10)    public synchronized void startServer(int port) throws IOException {
(11)            if (runner == null) {
(12)                    server = new ServerSocket(port);
(13)                    runner = new Thread(this);
(14)                    runner.start();
(15)            } //if
(16)    } //startServer()
(17)
(18)    public synchronized void stopServer() {
(19)            if (server != null) {
(20)                    shouldStop = true;
(21)                    runner.interrupt();
(22)                    runner = null;
(23)                    try {
(24)                            server.close();
(25)                    } catch (IOException ioe) {}
(26)                    server = null;
(27)            } //if
```

```
(28)    } //stopServer()
(29)
(30)    public void run() {
(31)            if (server != null) {
(32)                    while (!shouldStop) {
(33)                            try {
(34)                                    Socket datasocket = server.accept();
(35)                                    TCPServer newSocket = (TCPServer) clone();
(36)
(37)                                    newSocket.server = null;
(38)                                    newSocket.data = datasocket;
(39)                                    newSocket.runner = new Thread(newSocket);
(40)                                    newSocket.runner.start();
(41)                            } catch (Exception e) {}
(42)                    } //while
(43)            } else {
(44)                    run(data);
(45)            } //if
(46)    } //run()
(47)
(48)    public void run(Socket data) {
(49)            InputStream in = null;
(50)            InputStreamReader isr = null;
(51)            try {
(52)                    in = data.getInputStream();
(53)                    isr = new InputStreamReader(in);
(54)
(55)                    int c;
(56)                    while ( (c = isr.read()) != -1 ) {
(57)                            System.out.println("("+data.getLocalPort()+"/"+Thread.currentThread().
getName()+") " + (char) c);
(58)                    } //while
(59)            } catch (IOException io) {
(60)                    System.out.println("an IOException occurred\n"+io.toString()+"\n"+io.getMessage
() );
(61)            } //catch
(62)    } //run()
(63)} //class TCPServer
```

Beispiel 46: *Einfaches Treiberprogramm*
Quellcode

```
(1)public class TestTCPServer {
(2)     public static void main(String argv[]) {
(3)             TCPServer servers[] = new TCPServer[argv.length];
(4)
(5)             for (int i=0; i<argv.length; i++) {
(6)                     servers[i] = new TCPServer();
(7)             } //for
(8)
(9)             try {
(10)                    for (int i=0; i<argv.length; i++) {
(11)                            servers[i].startServer(Integer.parseInt(argv[i]));
(12)                    } //for
(13)            } catch (Exception e) {}
(14)    } //main()
(15)} //class TestTCPServer
```

Definitionsverzeichnis

Atomar
Dämon-Thread
Implementierung nebenläufiger Abläufe in Java
kritischer Abschnitt
Nebenläufigkeit (Concurrency)
Parallelität

Prozeß
Race Condition
Ressource
run-Methode
Semaphor
Single Thread Rule
Thread
UI Element Access Rule
Verklemmung
Wechselseitiger Ausschluß

Schlagwortverzeichnis

Anwender-Thread
Atomar
Batchbetrieb
Beispiele zu synchronized aus der Java-API
Betriebsmittel
busy waiting
Concurrency
Dämon-Thread
Deadlock
Dialogbetrieb
Erzeuger-Verbraucher Problem
Explizites Setzen einer klassenbasierten Sperre
green threads
Hyper-Threading
Implementierung nebenläufiger Abläufe in Java
Klasse InheritableThreadLocal
Klasse Thread
Klasse ThreadGroup
Klasse ThreadLocal
kooperatives Scheduling
Kritik am synchronized-Ansatz
kritischer Abschnitt
kritische Ressource
Leser-Schreiber-Problem
Mehrbenutzerbetrieb
Mehrprogrammbetrieb
Monitor
Multitasking
Multithretaing
Nebenläufigkeit (Concurrency)
Nichtdeterminismus
Parallelismus, echter
Parallelismus, quasi
Parallelität
Philosophenproblem
Pipelinearchitektur
Priorität
Prozeß, leichtgewichtiger
Prozeß
Puffers
Quick Sort
Race Condition
Ressource
Ressourcengraph
Round-Robin-Verfahren
run-Methode
Schnittstelle Runnable
Semaphor
Single Thread Rule
Stapelbetrieb
Superskalararchitektur
Synchronisation, objektbasierte
Synchronisation vollständiger Methoden

Synchronisaton, klassenbasierte
Synchronization einzelner Anweisungen
Thread
Thread-Konzept
threadspezifische extern verwaltete Variable
Threadumschaltung
Thread-Zustände
Time-Sharing-Betrieb
UI Element Access Rule
Umsetzung nicht-nebenläufiger Algorithmen in nebenläufige Analoga
unteilbare Hardwareoperation
Verklemmung
Wechselseitiger Ausschluß
Wirkung der Synchronisationsprimitive synchronized
Zeitscheibe
Zustandsübergänge eines Threads

Abbildungsverzeichnis

Multithreaded-Prozesse in einer Multitasking-Umgebung
Gleichmäßige CPU-Auslastung durch Verteilung von Threads auf zwei Prozessoren
UML-Klassendiagramm der ersten beiden Code-Beispiele
Thread-API in Java v1.4
Zustände und Zustandsübergänge eines Java-Threads
Statistische Auswertung der unberücksichtigten Schreibvorgänge
Lost-Update-Problem durch unsynchronisierte Zugriffe
Verhalten eines Monitors zur Synchronisation
Nebenläufige Lese- und Schreibvorgänge
Das Philosophen-Problem
Ressourcengraph einer Verklemmungssituation
Graph der Ressourcenanforderungen
Graph der angeforderten und erteilten Ressourcen zum Deadlockzeitpunkt
Klassendiagramm der statischen Struktur des Beispiels
Abbildung der Java Threadprioritäten auf die der Win32 Betriebssystemplattform
Ausführungsablauf unter Einsatz eines Dämon-Threads
Erzeugungsgraph der einzelnen Threads

Verzeichnis der Beispiele

Nebenläufige Implementierung auf Basis der Ableitung von Thread
Nebenläufige Implementierung auf Basis der Realisierung der Schnittstelle Runnable
Nebenläufiges Inkrementieren eines Zählers in einer Datei
Naive Sperroperation
Synchronisation der nebenläufigen Zähler-Inkrementierung durch das Schlüsselwort synchronized
Sperrung einer statischen Methode durch das Schlüsselwort synchronized
Synchronisation der nebenläufigen Zähler-Inkrementierung durch einen synchronized-Block
Klasse mit zwei synchronisierten Methoden
Klasse mit einer statischen und einer Instanzmethode, die beide als synchronized deklariert sind
Explizites Setzen einer klassenbasierten Sperre
Eine Semaphore
Eine Semaphorenumsetzung, die ein Feld von Sperrvariablen kontrolliert
Synchronisation unter Einsatz einer Semaphore
Probleme beim Einsatz von Wait und Notify
Lösung des Problems beim Einsatz von Wait und Notify
Implementierung eines Monitors mit Hilfe von Semaphoren
Erzeuger-Verbraucherproblem am Beispiel einer Bäckerei
Datenpuffer zur Entkopplung von Sender und Empfänger
Erzeuger-Verbraucherproblem am Beispiel einer Bäckerei unter Verwendung eines Puffers
Implementierung der Synchronisation nebenläufiger Lese- und Schreibvorgänge mit wait und notify
Implementierung der Synchronisation nebenläufiger Lese- und Schreibvorgänge mit Semaphoren
Implementierung der hungrigen Philosophen mit wait und notify
Implementierung der hungrigen Philosophen mit einer Semaphoregruppe
Implementierung der hungrigen Philosophen mit Hilfe eines Monitors

Implementierung einer Konkurrenzsituation um eine Ressource die zu Verklemmungen führen kann
Verklemmung bei der Verwendung des Schlüsselwortes synchronized
Verklemmung bei der Verwendung synchronisierter Methoden
Fehlerhafte Implementierung des Philosophenproblems mit Semaphoren
Erste Implementierung eines Resourcenverwalters
Verklemmung trotz Resourcenmanager
Threads mit verschiedener Priorität
Ermittlung threadspezifischer Information
Verhalten eines Dämon-Threads
Verwendung der Methode yield
Sequentielle Quick Sort-Implementierung
Nebenläufige Quick Sort-Implementierung
Einfach Swing-Applikation
Ermittlung des Ereignisbehandlungssthreads
Zeitintensive Ereignisbehandlung
Ereignisbehandlung durch Threads
Fehlerhafte Ereignisbehandlung durch Threads
Korrekte nebenläufige Ereignisbehandlung
Nicht nebenläufige Verarbeitung empfangenen Daten
Einfache nebenläufige Verarbeitung von empfangenen Daten
Ein einfacher TCP-Server der pro eingehender Verbindung einen Thread startet
Einfaches Treiberprogramm

Service provided by Mario Jeckle

Generated: 2004-01-12T09:07:30+01:00

Feedback SiteMap

This page's original location: http://www.jeckle.de/vorlesung/javaThreads/script.html

RDF description for this page