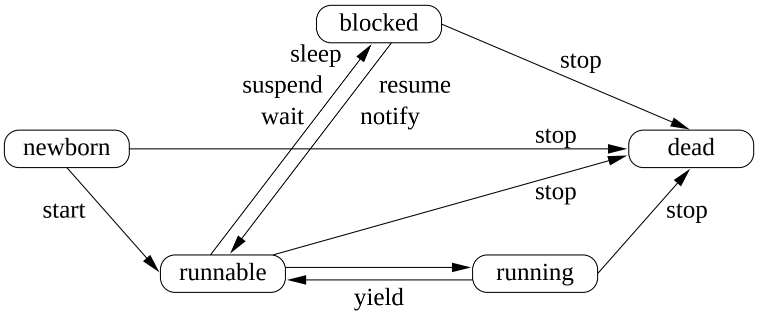
Thread-Erzeugung, -Scheduling und -Terminierung

Mögliche Threadzustände:



Instanziieren von Threads

Durch Ableiten einer Klasse von der Klasse `Thread`:

```
class MyThread extends Thread {
  public void run() { ... }
}
...
Thread thr = new MyThread();
thr.start();
```

Durch Implementieren des Interfaces `Runnable`:

```
class MyRunnable implements Runnable {
  public void run() { ... }
}
...
Runnable r = new MyRunnable();
Thread thr = new Thread(r);
thr.start();
```

Beenden von Threads

Ein Thread wird beendet, wenn die `void run()`-Methode zurückkehrt oder eine Ausnahme geworfen und nicht gefangen wird. Die Methode `void stop()` der Klasse `Thread` wirft eine `ThreadDeath`-, `void stop(Throwable o)` eine beliebige Ausnahme.

Scheduling und Preemption

Sind mehrere Threads *runnable*, aber keiner *running*, so sucht der Scheduler einen Thread aus und versetzt ihn in den Zustand *running*. Bei einer sequentiellen Implementierung ist immer nur ein Thread *running*; um dennoch den Eindruck von Parallelität zu erwecken, kann jedoch eine Technik namens *Preemption* verwendet werden: Nach Ablauf einer sogenannten Zeitscheibe wird der laufende Thread unterbrochen, in den Zustand *runnable* versetzt und ein anderer Thread ausgewählt. Dieses kann der laufende Thread auch explizit durch Aufruf der Methode `void yield()` erbitten. Die Auswahl eines neuen Threads durch den Scheduler kann durch Thread-Prioritäten beeinflußt werden, die durch `void setPriority(int p)` vergeben werden können.

Bemerkung

Die Java-Sprachdefinition legt nicht fest, daß Implementierungen Threads preemptieren müssen (sie dürfen sogar Aufrufe von `yield` ignorieren) oder daß die Prioritäten eingehalten und Threads fair abgearbeitet werden müssen. Aus diesem Grund werden explizite Synchronisationskonstrukte notwendig sein.

Mutual Exclusion, Locks

Mutual Exclusion wird benötigt,

- wenn mehrere nebenläufige Threads auf dieselbe Ressource zugreifen möchten, diese jedoch nur von einen Thread zur Zeit verwendet werden kann;
- wenn die Einhaltung von Invarianten bezüglich des Objektzustands zugesichert werden muß.

In dem folgenden Beispiel kann es ohne Mutual Exclusion eine *Race Condition* geben:

```
class Point {
  private double d, alpha, x, y;
  public void setPolarCoord(double newd, double newalpha) {
    d = newd;
    alpha = newalpha;
    // (mögliche Preemption)
    x = d * Math.sin(alpha);
    y = d * Math.cos(alpha);
  }
}
```

Ein Thread rufe auf:

```
p.setPolarCoord(1.0, 0.0);
```

Ein anderer Thread rufe fast zeitgleich auf:

```
p.setPolarCoord(0.0, 1.0);
```

Die Invariante $x = d * \sin(\alpha) \;\wedge\; y = d * \cos(\alpha)$ kann verletzt werden, wenn an dem markierten Punkt eine Thread-Preemption stattfindet.

Locks

Java bietet zur Realisierung von Mutual Exclusion Locks, die das Konzept der Monitore implementieren, das zuerst von Hoare vorgestellt wurde. Jede Klasse und jedes Objekt besitzt genau ein Lock.

Bemerkung: Alle Zugriffe auf Variablen (mit Ausnahme solcher vom Typ `double` oder `long`) sind garantiert atomar, das heißt, daß keine Thread-Preemption zwischen Beginn und Ende der Ausführung der Operation stattfinden kann. Hier kann gegebenenfalls auf Locks verzichtet werden.

Synchronized-Methoden und -Blöcke

Syntax eines synchronized-Blocks:

`synchronized (`⟨*expr*⟩`) {` ⟨*statement*⟩ `}`

Vor Betreten eines synchronized-Blocks wird der Ausdruck ausgewertet (dieser muß eine Klasse oder ein Objekt ergeben) und dessen Lock überprüft. Ist es frei oder von dem aktuellen Thread gesperrt, so wird es für andere Threads gesperrt und der Block betreten. Nach Beenden des Blocks wird das Lock freigegeben (dies geschieht auch, wenn eine Ausnahme geworfen wird). Überprüfen und Sperren eines Locks stellen zusammen eine atomare Operation dar.

Eine synchronized-Methode ist nur syntaktischer Zucker:

```
synchronized int m() {          int m() {
  ⟨statement⟩            ⟹        synchronized (this) { ⟨statement⟩ }
}                               }
```

Um die Thread-Reentranz realisieren zu können, muß jedes Lock sich den sperrenden Thread merken sowie einen Zähler, der besagt, wie oft das Lock von diesem Thread gesperrt wurde. Bei jedem Verlassen eines synchronized-Blocks wird der Zähler dekrementiert; erst, wenn er Null erreicht, wird das Lock wieder für andere Threads freigegeben.

Beispiel für Thread-Reentranz:

```
class ThreadReentrantLocks {
  public synchronized void print(int x) {
    if (x != 0)
      System.out.println(x);
    else
      printDefault();
  }
  private void printDefault() {
    print(17);
  }
  public static void main(String[] args) {
    new ThreadReentrantLocks().print(Integer.parseInt(args[0]));
  }
}
```

Klassen-Locks können wie folgt gesperrt werden:

- durch statische synchronized-Methoden
- `synchronized (o.getClass()) { ... }`
- `synchronized (String.class) { ... }   // nur in JDK 1.1`

Deadlocks

Wenn auf mehrere Objekte synchronisiert wird, muß darauf geachtet werden, daß keine Deadlocks auftreten. Beispiel hierfür:

```
class Deadlock extends Thread {
  private Object o1, o2;
  public Deadlock(Object p, Object q) {
    o1 = p;
    o2 = q;
  }
  public void run() {
    synchronized (o1) {
      try {
        Thread.sleep(100);
      } catch (InterruptedException e) {
        return;
      }
      synchronized (o2) {
        System.out.println(this + ": " + o1 + ", " + o2);
```

```
      }
    }
  }
  public static void main(String[] args) {
    Object o1 = new Integer(7);
    Object o2 = new Integer(17);
    new Deadlock(o1, o2).start();
    new Deadlock(o2, o1).start();
  }
}
```

Bemerkung: Die Sprachdefinition legt nicht fest, welcher Thread ausgewählt wird, wenn mehrere Threads auf ein Lock warten, während dieses freigegeben wird.

Synchronisation

Jedes Objekt speichert ein *wait set*, das eine Menge von Objekten enthält.

Die Operation wait

Ablauf der Methoden `void wait()` und `void wait(long timeout)` der Klasse `Object`:

1. `o.wait(...)` wird aus einem Thread t_1 aufgerufen. (Verstreichen bei Angabe eines Timeouts soviele Millisekunden wie angegeben, bevor t_1 wieder *runnable* wird, so wird in t_1 eine `InterruptedException` geworfen.)

2. Das Lock von o muß gerade von t_1 gesperrt sein, sonst wird eine Ausnahme der Klasse `IllegalMonitorStateException` geworfen.

3. Das Lock von o wird freigegeben (genauso oft, wie es von t_1 gesperrt wurde) und t_1 wird in das *wait set* von o aufgenommen.

4. t_1 betritt den Zustand *blocked*.

Die Operation notify

Ablauf der Methoden `void notify()` und `void notifyAll()` der Klasse `Object`:

1. `o.notify()` oder `o.notifyAll()` wird aus einem Thread t_2 aufgerufen.

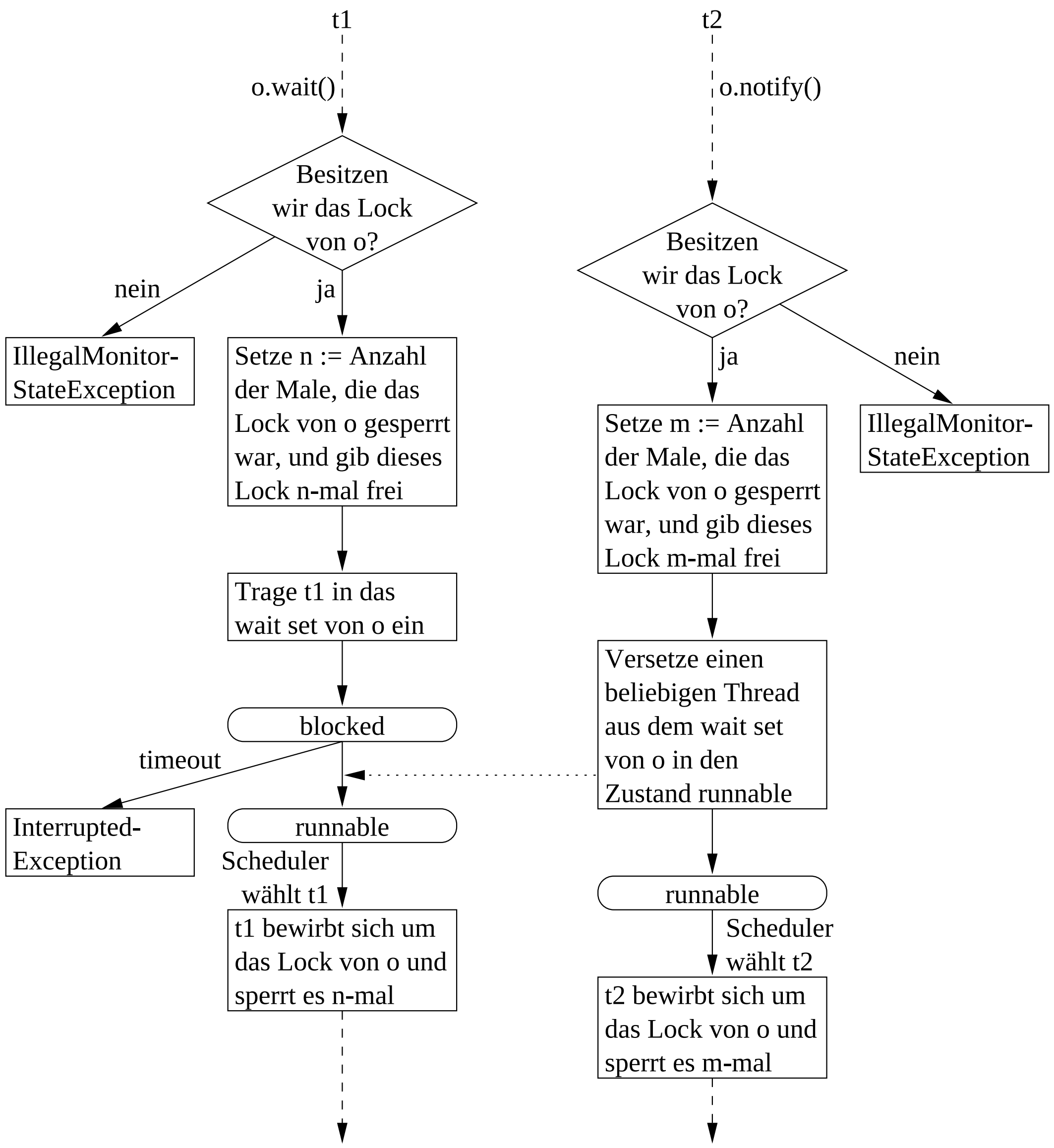
2. Das Lock von o muß gerade von t_2 gesperrt sein, sonst wird eine Ausnahme der Klasse `IllegalMonitorStateException` geworfen.

3. Das Lock von o wird freigegeben (genauso oft, wie es von t_2 gesperrt wurde) und t_2 preemptiert.

4. Ein Thread aus dem *wait set* wird in den Zustand *runnable* versetzt (oder alle bei `notifyAll`). Dessen erste Aktion wird es sein, sich um das Lock von o zu bewerben und genauso oft zu sperren, wie es vor Aufruf von `wait` aus diesem Thread der Fall war.

   Bemerkung: Die Sprachdefinition legt nicht fest, welcher Thread hier ausgewählt wird.

Das folgende Diagramm verdeutlicht das Zusammenspiel der beiden Operationen.



Beispiel für `wait` und `notify`:

```
class BoundedBuffer {
  private int maxSize, size = 0, readPtr = 0, writePtr = 0;
  private Object[] buffer;
  public BoundedBuffer(int n) {
    maxSize = n;
    buffer = new Object[maxSize];
  }
  public synchronized void put(Object item) {
    while (size == maxSize) {
      try {
        wait();
      } catch (InterruptedException e) { }
    }
    buffer[writePtr] = item;
    writePtr = (writePtr + 1) % maxSize;
    size++;
    if (size == 1)
      notifyAll();   // there may be a blocked get()
  }
  public synchronized Object get() {
    while (size == 0) {
      try {
        wait();
      } catch (InterruptedException e) { }
    }
    Object item = buffer[readPtr];
    readPtr = (readPtr + 1) % maxSize;
    size--;
    if (size == maxSize - 1)
      notifyAll();   // there may be a blocked put();
    return item;
  }
}
```