

# Threads – Einführung

- ▶ Die Ausführung eines **Java**-Programms besteht in Wahrheit nicht aus einem, sondern **mehreren** parallel laufenden **Threads**.
- ▶ Ein Thread ist ein sequentieller Ausführungsstrang.
- ▶ Der Aufruf eines Programms startet einen Thread **main**, der die Methode **main()** des Programms ausführt.
- ▶ Ein weiterer Thread, den das Laufzeitsystem parallel startet, ist die **Garbage Collection**.
- ▶ Die Garbage Collection soll mittlerweile nicht mehr erreichbare Objekte beseitigen und den von ihnen belegten Speicherplatz der weiteren Programmausführung zur Verfügung stellen.

# Threads – Anwendungen

- ▶ Mehrere Threads sind auch nützlich, um
  - ▶ ...mehrere Eingabe-Quellen zu überwachen (z.B. Maus, Tastatur) ↑**Graphik**;
  - ▶ ...während der Blockierung einer Aufgabe etwas anderes Sinnvolles erledigen zu können;
  - ▶ ...die Rechenkraft mehrerer Prozessoren auszunutzen.
- ▶ Neue Threads können deshalb vom Programm selbst erzeugt und gestartet werden.
- ▶ Dazu stellt **Java** die Klasse **Thread** und das Interface **Runnable** bereit.

# Version A

```
1 public class MyThread extends Thread {
2     public void hello(String s) {
3         System.out.println(s);
4     }
5     public void run() {
6         hello("I'm running ...");
7     } // end of run()
8     public static void main(String[] args) {
9         MyThread t = new MyThread();
10        t.start();
11        System.out.println("Thread has been started ...");
12    } // end of main()
13 } // end of class MyThread
```

# Erläuterungen

- ▶ Neue Threads werden für Objekte aus (Unter-) Klassen der Klasse `Thread` angelegt.
- ▶ Jede Unterklasse von `Thread` sollte die Objekt-Methode `public void run();` implementieren.
- ▶ Ist `t` ein `Thread`-Objekt, dann bewirkt der Aufruf `t.start();` das folgende:
  1. ein neuer Thread wird initialisiert;
  2. die (parallele) Ausführung der Objektmethode `run()` für `t` wird angestoßen;
  3. die eigene Programmausführung wird hinter dem Aufruf fortgesetzt.

## Version B

```
1 public class MyRunnable implements Runnable {
2     public void hello(String s) {
3         System.out.println(s);
4     }
5     public void run() {
6         hello("I'm running ...");
7     } // end of run()
8     public static void main(String[] args) {
9         Thread t = new Thread(new MyRunnable());
10        t.start();
11        System.out.println("Thread has been started ...");
12    } // end of main()
13 } // end of class MyRunnable
```

# Erläuterungen

- ▶ Auch das Interface `Runnable` verlangt die Implementierung einer Objektmethode `public void run()`;
- ▶ `public Thread(Runnable obj)`; legt für ein `Runnable`-Objekt `obj` ein `Thread`-Objekt an.
- ▶ Ist `t` das `Thread`-Objekt für das `Runnable obj`, dann bewirkt der Aufruf `t.start()`; das folgende:
  1. ein neuer Thread wird initialisiert;
  2. die (parallele) Ausführung der Objekt-Methode `run()` für `obj` wird angestoßen;
  3. die eigene Programm-Ausführung wird hinter dem Aufruf fortgesetzt.

# Mögliche Ausführungen

## Entweder

```
Thread has been started ...  
I'm running ...
```

## oder

```
I'm running ...  
Thread has been started ...
```

# Scheduling


- ▶ Ein Thread kann nur eine Operation ausführen, wenn ihm ein Prozessor (CPU) zur Ausführung zugeteilt worden ist.
- ▶ Im Allgemeinen gibt es mehr Threads als CPUs.
- ▶ Der **Scheduler** verwaltet die verfügbaren CPUs und teilt sie den Threads zu.
- ▶ Bei verschiedenen Programmläufen kann diese Zuteilung verschieden aussehen!!!
- ▶ Es gibt verschiedene Strategien, nach denen sich Scheduler richten können (↑**Betriebssysteme**). Z.B.:
  - ▶ Zeitscheibenverfahren
  - ▶ Naives Verfahren



## Strategie

- ▶ Ein Thread erhält eine CPU nur für eine bestimmte Zeitspanne (**Time Slice**), in der er rechnen darf.
- ▶ Danach wird er unterbrochen. Dann darf ein anderer.

# Beispiel: Zeitscheibenverfahren

Thread-3: 


Thread-2: 


Thread-1: 

Schedule: 

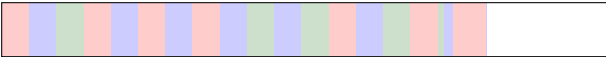
# Beispiel: Zeitscheibenverfahren

Eine andere Ausführung:

Thread-3: 

Thread-2: 

Thread-1: 

Schedule: 


# Erläuterungen – Zeitscheibenverfahren

- ▶ Ein Zeitscheiben-Scheduler versucht, jeden Thread **fair** zu behandeln, d.h. ab und zu Rechenzeit zuzuordnen — egal, welche Threads sonst noch Rechenzeit beanspruchen.
- ▶ Kein Thread hat jedoch Anspruch auf einen bestimmten Time-Slice.
- ▶ Für den Programmierer sieht es so aus, als ob sämtliche Threads „echt“ parallel ausgeführt werden, d.h. jeder über eine eigene CPU verfügt.

## Strategie

- ▶ Erhält ein Thread eine CPU, darf er laufen, so lange er will...
- ▶ Gibt er die CPU wieder frei, darf ein anderer Thread arbeiten...

# Beispiel – Naives Scheduling

Thread-3: 

Thread-2: 

Thread-1: 

Schedule: 

# Beispiel

```
1 public class Start extends Thread {
2     public void run() {
3         System.out.println("I'm running...");
4         while (true);
5     }
6     public static void main(String[] args) {
7         (new Start()).start();
8         (new Start()).start();
9         (new Start()).start();
10        System.out.println("main is running...");
11        while (true);
12    }
13 } // end of class Start
```

# Beispiel

**Ausgabe** (bei naivem Scheduling)

```
main is running...
```

Weil **main** nie fertig wird, erhalten die anderen Threads keine Chance, sie **verhungern**.

Faires Scheduling mit Zeitscheibenverfahren würde z.B. liefern:

```
I'm running...  
main is running...  
I'm running...  
I'm running...
```



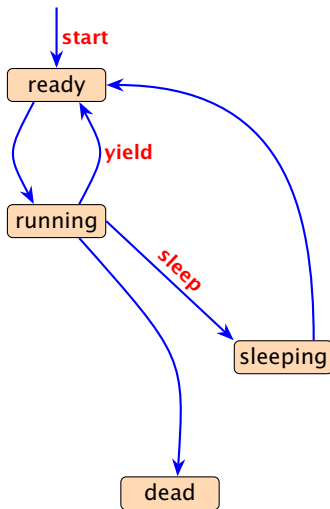
# Implementierung in Java

- ▶ **Java** legt nicht fest, wie intelligent der Scheduler ist.
- ▶ Die aktuelle Implementierung unterstützt **fares** Scheduling.
- ▶ Programme sollten aber für jeden Scheduler das **gleiche Verhalten** zeigen. Das heißt:
- ▶ ... Threads, die aktuell nichts sinnvolles zu tun haben, z.B. weil sie auf Verstreichen der Zeit oder besseres Wetter warten, sollten stets ihre CPU anderen Threads zur Verfügung stellen.
- ▶ ... Selbst wenn Threads etwas Vernünftiges tun, sollten sie ab und zu andere Threads laufen lassen.

**Achtung:** Threadwechsel ist teuer!!!

- ▶ Dazu verfügt jeder Thread über einen **Zustand**, der bei der Vergabe von Rechenzeit berücksichtigt wird.

# Threadzustände

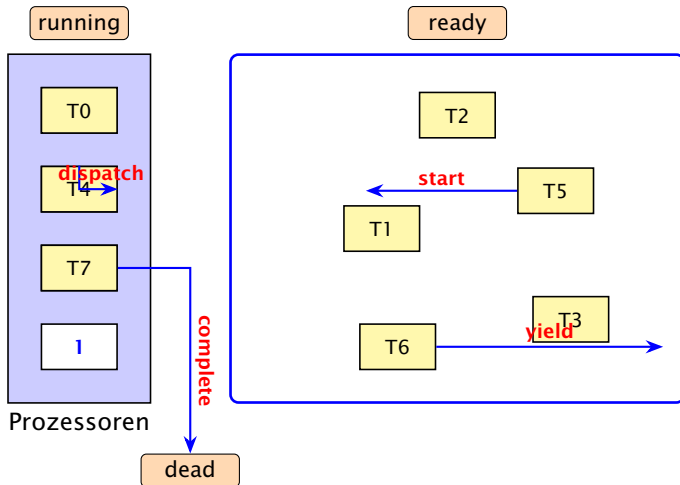


# Threadzustände

- ▶ `public void start();` legt einen neuen Thread an, setzt den Zustand auf `ready` und übergibt damit den Thread dem Scheduler zur Ausführung.
- ▶ Der Scheduler ordnet den Threads, die im Zustand `ready` sind, Prozessoren zu („dispatching“). Aktuell laufende Threads haben den Zustand `running`.
- ▶ `public static void yield();` setzt den aktuellen Zustand zurück auf `ready` und unterbricht damit die aktuelle Programmausführung. Andere ausführbare Threads erhalten die Gelegenheit zur Ausführung.
- ▶ `public static void sleep(int msec) throws InterruptedException;` legt den aktuellen Thread für `msec` Millisekunden schlafen, indem der Thread in den Zustand `sleeping` wechselt.

# Threadzustände

Animation ist nur in der Vorlesungsversion der Folien vorhanden.



## 16.1 Futures

- ▶ Die Berechnung eines Zwischenergebnisses kann lange dauern.
- ▶ Während dieser Berechnung kann möglicherweise etwas anderes Sinnvolles berechnet werden.

### Idee:

- ▶ Berechne das Zwischenergebnis in einem eigenen Thread.
- ▶ Greife auf den Wert erst zu, wenn sich der Thread beendet hat.

## 16.1 Futures

Eine **Future** startet die Berechnung eines Werts, auf den später zugegriffen wird. Das generische Interface

```
public interface Callable<T> {  
    T call () throws Exception;  
}
```

aus `java.util.concurrent` beschreibt Klassen, für deren Objekte ein Wert vom Typ `T` berechnet werden kann.

```
1 public class Future<T> implements Runnable {  
2     private T value = null;  
3     private Exception exc = null;  
4     private Callable<T> work;  
5     private Thread task;  
6 // continued...
```

# Implementierung

```
7     public Future(Callable<T> w) {
8         work = w;
9         task = new Thread (this);
10        task.start();
11    }
12    public void run() {
13        try {value = work.call();}
14        catch (Exception e) { exc = e;}
15    }
16    public T get() throws Exception {
17        task.join();
18        if (exc != null) throw exc;
19        return value;
20    }
```

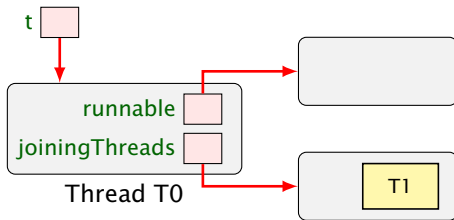
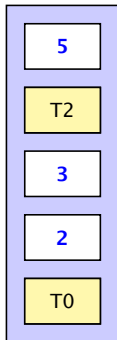
# Erläuterungen

- ▶ Der Konstruktor erhält ein `Callable`-Objekt.
- ▶ Die Methode `run()` ruft für dieses Objekt die Methode `call()` auf und speichert deren Ergebnis in dem Attribut `value` — bzw. eine geworfene Exception in `exc` ab.
- ▶ Der Konstruktor legt ein Thread-Objekt für die Future an und startet diesen Thread, der dann `run()` ausführt.
- ▶ Die Methode `get()` wartet auf Beendigung des Threads. Dazu verwendet sie die Objekt-Methode `public final void join() throws InterruptedException` der Klasse `Thread`...
- ▶ Dann liefert `get()` den berechneten Wert zurück — falls keine Exception geworfen wurde. Andernfalls wird die Exception `exc` erneut geworfen.



# Die Join-Operation

Animation ist nur in der Vorlesungsversion der Folien vorhanden.

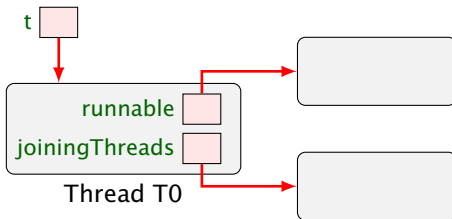
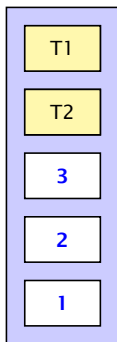


# Erläuterungen

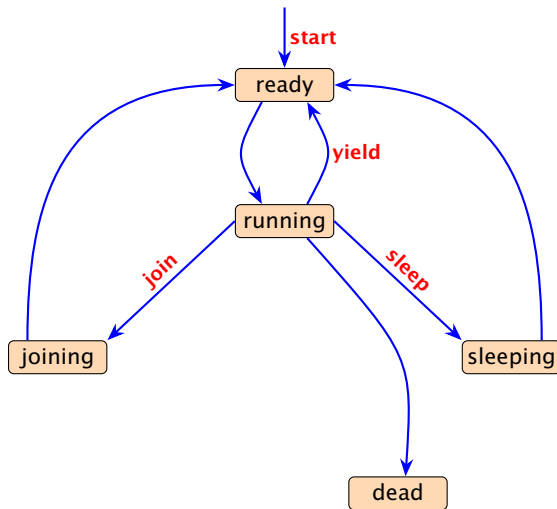
- ▶ Für jedes Threadobjekt `t` gibt es eine Schlange `ThreadQueue joiningThreads`.
- ▶ Threads, die auf Beendigung des Threads `t` warten, werden in diese Schlange eingefügt.
- ▶ Dabei gehen sie konzeptuell in einen Zustand `joining` über und werden aus der Menge der ausführbaren Threads entfernt.
- ▶ Beendet sich ein Thread, werden alle Threads, die auf ihn warteten, wieder aktiviert. . .

# Die Join-Operation

Animation ist nur in der Vorlesungsversion der Folien vorhanden.



# Threadzustände



## Weiteres Beispiel

```
1 public class Join implements Runnable {
2     private static int count = 0;
3     private int n = count++;
4     private static Thread[] task = new Thread[3];
5     public void run() {
6         try {
7             if (n>0) {
8                 task[n-1].join();
9                 System.out.println("Thread-"+n+
10                    " joined Thread-"+(n-1));
11             }
12         } catch (InterruptedException e) {
13             System.err.println(e.toString());
14         }
15     } // continued...
```

## Weiteres Beispiel

```
16     public static void main(String[] args) {
17         for(int i=0; i<3; i++)
18             task[i] = new Thread(new Join());
19         for(int i=0; i<3; i++)
20             task[i].start();
21     }
22 } // end of class Join
```

liefert:

```
> java Join
Thread-1 joined Thread-0
Thread-2 joined Thread-1
```

# Variation

```
1 public class CW implements Runnable {
2     private static int count = 0;
3     private int n = count++;
4     private static Thread[] task = new Thread[3];
5     public void run() {
6         try { task[(n+1) % 3].join(); }
7         catch (InterruptedException e) {
8             System.err.println(e.toString());
9         }
10    }
11    public static void main(String[] args) {
12        for(int i=0; i<3; i++)
13            task[i] = new Thread(new CW());
14        for(int i=0; i<3; i++) task[i].start();
15    }
16 } // end of class CW
```

# Variation

- ▶ Das Programm terminiert möglicherweise nicht...
- ▶ `task[0]` wartet auf `task[1]`,  
`task[1]` wartet auf `task[2]`,  
`task[2]` wartet auf `task[0]`

ist möglich...

`t.join` angewendet auf einen nicht gestarteten Thread, hat keine Auswirkungen. Deshalb kann das Programm je nach Scheduling auch durchlaufen...



# Deadlock

- ▶ Jeder Thread geht in einen Wartezustand (hier: **joining**) über und wartet auf einen anderen Thread.
- ▶ Dieses Phänomen heißt auch **Circular Wait** oder **Deadlock** oder Verklemmung — eine unangenehme Situation, die man in seinen Programmen tunlichst vermeiden sollte.

Die Vermeidung von Deadlocks ist ein sehr schwieriges Problem.

## 16.2 Monitore

- ▶ Damit Threads sinnvoll miteinander kooperieren können, müssen sie miteinander Daten austauschen.
- ▶ Zugriff mehrerer Threads auf eine gemeinsame Variable ist problematisch, weil nicht feststeht, in welcher Reihenfolge die Threads auf die Variable zugreifen.
- ▶ Ein Hilfsmittel, um geordnete Zugriffe zu garantieren, sind **Monitore**.

## Beispiel — Erhöhen einer Variablen

```
1 public class Inc implements Runnable {
2     private static int x = 0;
3     private static void pause(int t) {
4         try {
5             Thread.sleep((int) (Math.random()*t*1000));
6         } catch (InterruptedException e) {
7             System.err.println(e.toString());
8         }
9     }
10    public void run() {
11        String s = Thread.currentThread().getName();
12        pause(3); int y = x;
13        System.out.println(s+ " read "+y);
14        pause(4); x = y+1;
15        System.out.println(s+ " wrote "+(y+1));
16    }
17 // continued...
```

## Beispiel

```
18     public static void main(String[] args) {
19         (new Thread(new Inc())).start();
20         pause(2);
21         (new Thread(new Inc())).start();
22         pause(2);
23         (new Thread(new Inc())).start();
24     }
25 } // end of class Inc
```

- ▶ `public static Thread currentThread();` liefert (eine Referenz auf) das ausführende Thread-Objekt.
- ▶ `public final String getName();` liefert den Namen des Thread-Objekts.
- ▶ Das Programm legt für 3 Objekte der Klasse `Inc` Threads an.
- ▶ Die Methode `run()` inkrementiert die Klassen-Variable `x`.

# Beispiel

## Mögliche Ausführung

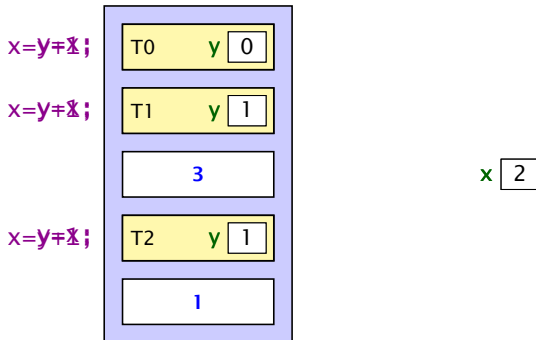
```
> java Inc  
Thread-0 read 0  
Thread-0 wrote 1  
Thread-1 read 1  
Thread-2 read 1  
Thread-1 wrote 2  
Thread-2 wrote 2
```

x wurde dreimal erhöht, hat aber am Ende den Wert 2!!!

Beachte, dass das gleiche auch passieren könnte, wenn wir in der Methode `run()`, direkt `x=x+1` schreiben würden.

# Erklärung

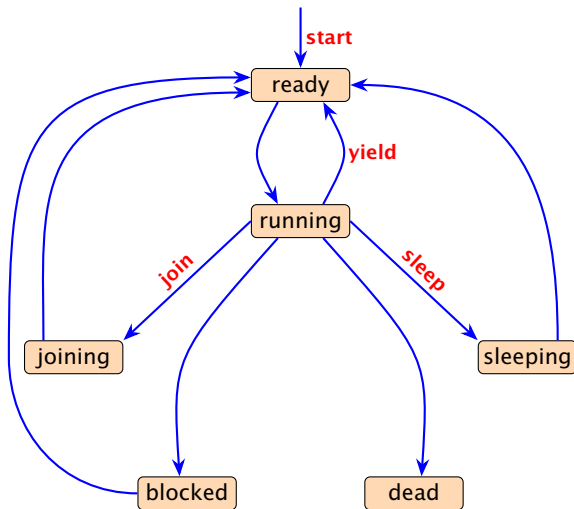
Animation ist nur in der Vorlesungsversion der Folien vorhanden.



## Monitore — Idee

- ▶ Inkrementieren der Variable `x` sollte ein **atomarer Schritt** sein, d.h. nicht von parallel laufenden Threads unterbrochen werden können.
- ▶ Mithilfe des Schlüsselworts **synchronized** kennzeichnen wir Objekt-Methoden einer Klasse `L` als ununterbrechbar.
- ▶ Für jedes Objekt `obj` der Klasse `L` kann zu jedem Zeitpunkt nur ein Aufruf `obj.synchMeth(...)` einer **synchronized**-Methode `synchMeth()` ausgeführt werden. Die Ausführung einer solchen Methode nennt man **kritischen Abschnitt** („critical section“) für die gemeinsame Resource `obj`.
- ▶ Wollen mehrere Threads gleichzeitig in ihren kritischen Abschnitt für das Objekt `obj` eintreten, werden alle bis auf einen **blockiert**.

# Threadzustände





# Locks

Dieses ist nur ein mentales Modell dafür was bei Eintritt/Austritt aus einer synchronized Methode passiert. Man kann diese Attribute nicht direkt zugreifen, und die Implementierung könnten in der Realität ganz anders aussehen.

- ▶ Ein Objekt `obj` mit `synchronized`-Methoden verfügt über:
  1. boolesches Flag `boolean locked`; sowie
  2. eine Warteschlange `ThreadQueue blockedThreads`.
- ▶ Vor Betreten seines kritischen Abschnitts führt ein Thread (**implizit**) die atomare Operation `obj.lock()` aus:

```
private void lock() {  
    if (!locked) locked = true; // betrete krit. Abschnitt  
    else { // Lock bereits vergeben  
        Thread t = Thread.currentThread();  
        blockedThreads.enqueue(t);  
        t.state = blocked; // blockiere  
    }  
} // end of lock()
```

# Locks

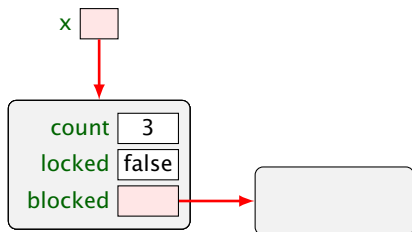
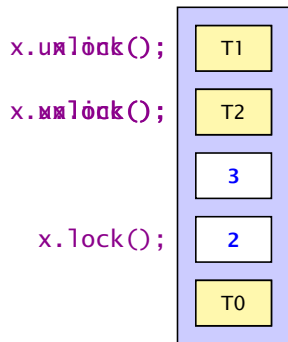
- ▶ Verlässt ein Thread seinen kritischen Abschnitt für `obj` (evt. auch mittels einer Exception), führt er (implizit) die atomare Operation `obj.unlock()` aus:

```
1 private void unlock() {
2     if (blockedThreads.empty())
3         locked = false; // Lock frei geben
4     else { // Lock weiterreichen
5         Thread t = blockedThreads.dequeue();
6         t.state = ready;
7     }
8 } // end of unlock()
```

- ▶ Dieses Konzept nennt man **Monitor**.

# Beispiel

Animation ist nur in der Vorlesungsversion der Folien vorhanden.



# Implementierung

```
class Count {
    private int count = 0;
    public synchronized void inc() {
        String s = Thread.currentThread().getName();
        int y=count; System.out.println(s+" read "+y);
        count=y+1; System.out.println(s+" wrote "+count);
    }
} // end of class Count
public class IncSync implements Runnable {
    private static Count x = new Count();
    public void run() { x.inc(); }
    public static void main(String[] args) {
        (new Thread(new IncSync())).start();
        (new Thread(new IncSync())).start();
        (new Thread(new IncSync())).start();
    }
} // end of class IncSync
```

# Beispiel

**liefert:**

```
> java IncSync  
Thread-0 read 0  
Thread-0 wrote 1  
Thread-1 read 1  
Thread-1 wrote 2  
Thread-2 read 2  
Thread-2 wrote 3
```

## Achtung:

- ▶ Die Operation `lock()` erfolgt nur, wenn der Thread nicht bereits **vorher** das Lock des Objekts erworben hat.
- ▶ Ein Thread, der das Lock eines Objekts `obj` besitzt, kann **weitere** Methoden für `obj` aufrufen, ohne sich selbst zu blockieren.

## Locks — im Detail

Diese Art von Locks heißen auch **Reentrant Locks** oder **Rekursive Locks**. Sie haben teilweise einen schlechten Ruf.

- ▶ Um das zu garantieren, legt ein Thread für jedes Objekt `obj`, dessen Lock er nicht besitzt, aber erwerben will, einen neuen Zähler an:

```
int countLock[obj] = 0;
```

- ▶ Bei jedem Aufruf einer **synchronized**-Methode `m(...)` für `obj` wird der Zähler inkrementiert, für jedes Verlassen (auch mittels Exceptions) dekrementiert:

```
if (0 == countLock[obj]++) lock();  
obj.m(...);  
if (--countLock[obj] == 0) unlock();
```

- ▶ `lock()` und `unlock()` werden nur ausgeführt, wenn  
`(countLock[obj] == 0)`

## Beispiel: synchronized

```
1 public class StopThread extends Thread {
2     private static boolean stopRequested;
3
4     public void run() {
5         int i = 0;
6         while (!stopRequested)
7             i++;
8     }
9     public static void main(String[] args) {
10        Thread background = new StopThread();
11        background.start();
12        try {Thread.sleep(5);}
13        catch (InterruptedException e) {};
14        stopRequested = true;
15    }
16 }
```

## Beispiel: synchronized

Das Programm terminiert (eventuell) nicht.

Die Änderung der Variablen `stopRequested` wird dem anderen Prozess nie mitgeteilt.

Wenn man auf die Variable durch `synchronized`-Methoden zugreift, ist sichergestellt, dass die Kommunikation korrekt durchgeführt wird.



## Beispiel: synchronized

```
1 public class StopThreadCorrect extends Thread {
2     private static boolean stopRequested;
3     private static synchronized void setStop() {
4         stopRequested = true;
5     }
6     private static synchronized boolean getStop() {
7         return stopRequested;
8     }
9     public void run() {
10        int i = 0;
11        while (!getStop()) i++;
12    }
13    public static void main(String[] args) {
14        Thread background = new StopThreadCorrect();
15        background.start();
16        try {Thread.sleep(5);}
17        catch (InterruptedException e) {};
18        setStop();
19    } }
```

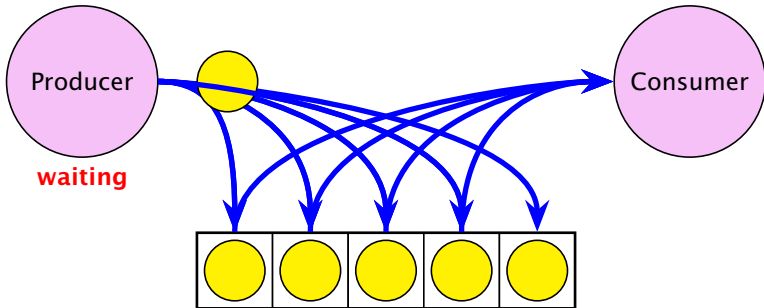
## 16.3 Semaphore

### Aufgabe:

- ▶ Zwei Threads möchten mehrere/viele Daten-Objekte austauschen.
- ▶ Der eine Thread erzeugt die Objekte einer Klasse **Data** (**Producer**).
- ▶ Der andere konsumiert sie (**Consumer**).
- ▶ Zur Übergabe dient ein Puffer, der eine feste Zahl **N** von **Data**-Objekten aufnehmen kann.

# Beispiel

Animation ist nur in der Vorlesungsversion der Folien vorhanden.



# Consumer/Producer

## 1.Idee

- ▶ Wir definieren eine Klasse `Buffer`, die (im wesentlichen) aus einem Feld der richtigen Größe, sowie zwei Verweisen `int first`, `last` zum Einfügen und Entfernen verfügt:

```
1 class Data {}
2 public class Buffer {
3     private int cap, free, first, last;
4     private Data[] a;
5     public Buffer(int n) {
6         free = cap = n; first = last = 0;
7         a = new Data[n];
8     }
9 // continued...
```

- ▶ Einfügen und Entnehmen sollen synchrone Operationen sein...

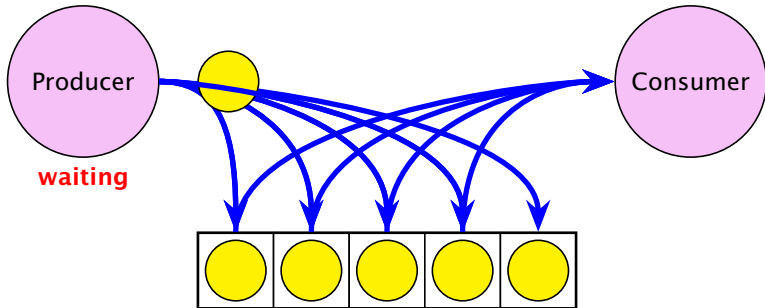
## Probleme

- ▶ Was macht der Consumer, wenn der Producer mit der Produktion nicht nachkommt, d.h. der Puffer leer ist?
- ▶ Was macht der Producer, wenn der Consumer mit der Weiterverarbeitung nicht nach kommt, d.h. der Puffer voll ist?

Lösungsvorschlag: **Warten...**

# Beispiel

Animation ist nur in der Vorlesungsversion der Folien vorhanden.



# Umsetzung

- ▶ Jedes Objekt (mit `synchronized`-Methoden) verfügt über eine weitere Schlange `ThreadQueue` `waitingThreads` am Objekt wartender Threads sowie die Objekt-Methoden:

```
public final void wait() throws InterruptedException;  
public final void notify();  
public final void notifyAll();
```

- ▶ Diese Methoden dürfen nur für Objekte aufgerufen werden, über deren Lock der Thread verfügt!!!

# Umsetzung

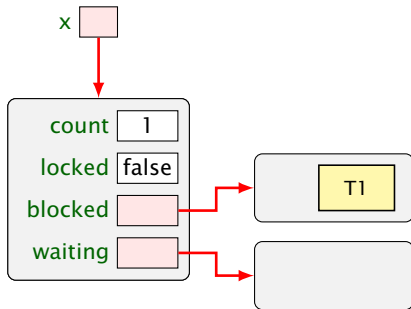
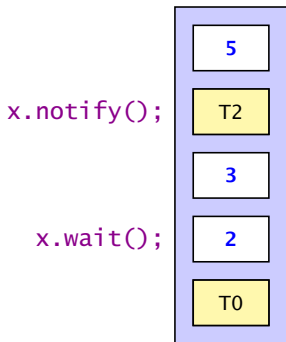
- ▶ Ausführen von `wait()`; setzt den Zustand des Threads auf `waiting`, reiht ihn in eine geeignete Warteschlange ein, und gibt das aktuelle Lock frei:

```
public void wait() throws InterruptedException {  
    Thread t = Thread.currentThread();  
    t.state = waiting;  
    waitingThreads.enqueue(t);  
    unlock();  
}
```



# Beispiel

Animation ist nur in der Vorlesungsversion der Folien vorhanden.



## Umsetzung

- ▶ Ausführen von `notify()`; weckt den ersten Thread in der Warteschlange auf, d.h. versetzt ihn in den Zustand `blocked` und fügt ihn in `blockedThreads` ein:

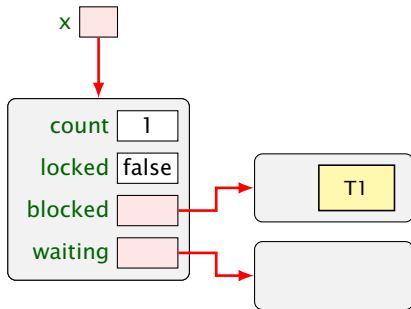
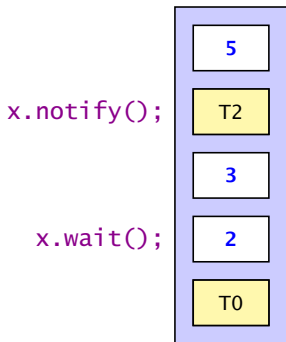
```
public void notify() {
    if (!waitingThreads.isEmpty()) {
        Thread t = waitingThreads.dequeue();
        t.state = blocked;
        blockedThreads.enqueue(t);
    }
}
```

- ▶ `notifyAll()`; weckt alle wartenden Threads auf d.h. fügt alle in `blockedThreads` ein:

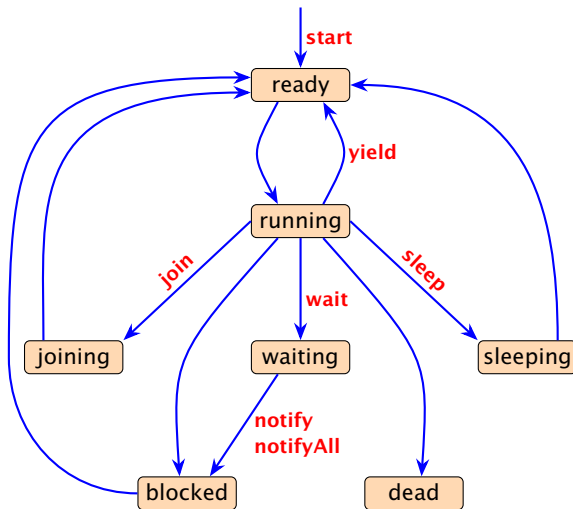
```
public void notifyAll() {
    while (!waitingThreads.isEmpty()) notify();
}
```

# Beispiel

Animation ist nur in der Vorlesungsversion der Folien vorhanden.



# Threadzustände



# Implementierung

```
10     public synchronized void produce(Data d)
11         throws InterruptedException {
12         if (free==0) wait(); free--;
13         a[last] = d;
14         last = (last+1) % cap;
15         notify();
16     }
17     public synchronized Data consume()
18         throws InterruptedException {
19         if (free==cap) wait(); free++;
20         Data result = a[first];
21         first = (first+1) % cap;
22         notify();
23         return result;
24     }
25 } // end of class Buffer
```

# Erläuterungen

- ▶ Ist der Puffer voll, d.h. keine Zelle frei, legt sich der Producer schlafen.
- ▶ Ist der Puffer leer, d.h. alle Zellen frei, legt sich der Consumer schlafen.
- ▶ Gibt es für einen Puffer genau einen Producer und einen Consumer, weckt das `notify()` des Consumers stets den Producer (und umgekehrt).
- ▶ Was aber, wenn es **mehrere** Producers gibt? Oder **mehrere** Consumers?

Puffer der Größe 1, mit einem gespeichertem Element:

```
P1->lock(), P1->wait(), C1->lock(), P2->lock(), C1->storeData(), C1->notify(),  
C1->unlock(), P2->removeData(), P2->unlock(), P1->removeData()⚡
```

P1 versucht ein Element zu entfernen, welches nicht mehr vorhanden ist.

# Consumer/Producer

## 2. Idee: Wiederholung des Tests

- ▶ Teste nach dem Aufwecken erneut, ob Zellen frei sind.
- ▶ Wecke nicht einen, sondern alle wartenden Threads auf...

```
9 public synchronized void produce(Data d)
10     throws InterruptedException {
11     while (free==0) wait(); free--;
12     a[last] = d;
13     last = (last+1) % cap;
14     notifyAll();
15 }
```

Man sollte auch bei einem Producer/Consumer immer `while` benutzen und nie nur `if`. Es kann vorkommen, dass ein Thread aufgeweckt wird, ohne dass `notify` aufgerufen wurde (*spurious wakups*).

Problem ohne `notifyAll()` (wieder Puffer der Größe 1 mit einem Element):

```
P1->lock(), P1->wait(), P2->lock(), P2->wait(), C1->lock(), C2->lock(),
C1->removeData(), C1->notify(), C1->unlock(), C2->wait(), P1->storeData(),
P1->notify(), P1->unlock(), P2->wait()⚡
```

Ein Consumer wartet während der Puffer voll ist.

# Consumer/Producer

```
16 public synchronized Data consume()
17     throws InterruptedException {
18     while (free==cap) wait(); free++;
19     Data result = a[first];
20     first = (first+1) % cap;
21     notifyAll();
22     return result;
23 }
```

- ▶ Wenn ein Platz im Puffer frei wird, werden **sämtliche** Threads aufgeweckt — obwohl evt. nur einer der Producer bzw. nur einer der Consumer aktiv werden kann.



# Consumer/Producer

## 3. Idee: Semaphore

- ▶ Producers und Consumers warten in **verschiedenen** Schlangen.
- ▶ Die Producers warten darauf, dass **free > 0** ist.
- ▶ Die Consumers warten darauf, dass **cap-free > 0** ist.

```
1 public class Sema {
2     private int x;
3     public Sema(int n) { x = n; }
4     public synchronized void up() {
5         x++; if (x <= 0) this.notify();
6     }
7     public synchronized void down()
8         throws InterruptedException {
9         x--; if (x < 0) this.wait();
10    }
11 } // end of class Sema
```

Diese Implementierung ist problematisch, da es in Java zu **spurious wakups** kommen kann.

# Semaphore

- ▶ Ein **Semaphor** enthält eine private **int**-Objekt-Variable und bietet die **synchronized**-Methoden **up()** und **down()** an.
- ▶ **up()** erhöht die Variable, **down()** erniedrigt sie.
- ▶ Ist die Variable positiv, gibt sie die Anzahl der verfügbaren Ressourcen an.  
Ist sie negativ, zählt sie die Anzahl der wartenden Threads.
- ▶ Eine **up()**-Operation weckt genau einen wartenden Thread auf.

Machmal wird die **up()**-Routine mit **V()** bezeichnet, und die **down()**-Routine mit **P()**.  
**V** steht für *verhoog* (erhöhen) und **P** für *prolaag*.

# Semaphore – Alternative Implementierung

```
1 public class SemaCorrected {
2     private int waiting, permits;
3     public Sema(int n) { permits = n; }
4     public synchronized void up() {
5         permits++; if (waiting > 0) notify();
6     }
7     public synchronized void down()
8         throws InterruptedException {
9         waiting++;
10        while (permits <= 0) wait();
11        waiting--;
12        permits--;
13    }
14 } // end of class Sema
```

# Anwendung - 1. Versuch

```
1 public class BufferFaulty {
2     private int cap, first, last;
3     private Sema free, occupied;
4     private Data[] a;
5     public BufferFaulty(int n) {
6         cap = n;
7         first = last = 0;
8         a = new Data[n];
9         free = new Sema(n);
10        occupied = new Sema(0);
11    }
12 // continued...
```

# Anwendung - 1. Versuch

```
13     public synchronized void produce(Data d)
14                                     throws InterruptedException {
15         free.down();
16         a[last] = d;
17         last = (last+1) % cap;
18         occupied.up();
19     }
20     public synchronized Data consume()
21                                     throws InterruptedException {
22         occupied.down();
23         Data result = a[first];
24         first = (first+1) % cap;
25         free.up();
26         return result;
27     }
28 } // end of class BufferFaulty
```

- ▶ Gut gemeint — aber leider **fehlerhaft**...
- ▶ Jeder Producer benötigt **zwei** Locks gleichzeitig, um zu produzieren:
  1. dasjenige für den Puffer;
  2. dasjenige für einen Semaphor.
- ▶ Muss er für den Semaphor ein **wait()** ausführen, gibt er das Lock für den Semaphor wieder zurück... nicht aber dasjenige für den Puffer!!!
- ▶ Die Folge ist, dass niemand mehr eine Puffer-Operation ausführen kann, insbesondere auch kein **up()** mehr für den Semaphor ⇒ **Deadlock**

## 2. Versuch – Entkopplung der Locks

```
12 // Methoden sind nicht synchronized
13 public void produce(Data d) throws
14     InterruptedException {
15     free.down();
16     synchronized (this) {
17         a[last] = d; last = (last+1) % cap;
18     }
19     occupied.up();
20 }
21 public Data consume() throws
22     InterruptedException {
23     Data result; occupied.down();
24     synchronized (this) {
25         result = a[first]; first = (first+1) % cap;
26     }
27     free.up(); return result;
28 }
29 } // end of corrected class Buffer
```

- ▶ Das Statement `synchronized (obj) { stmts }` definiert einen kritischen Bereich für das Objekt `obj`, in dem die Statement-Folge `stmts` ausgeführt werden soll.
- ▶ Threads, die die neuen Objekt-Methoden `void produce(Data d)` bzw. `Data consume()` ausführen, benötigen zu jedem Zeitpunkt nur genau ein Lock.

Man kann sich eine `synchronized`-Methode vorstellen, als hätte sie solch einen `synchronized (this)`-Block um den gesamten Funktionsrumpf.

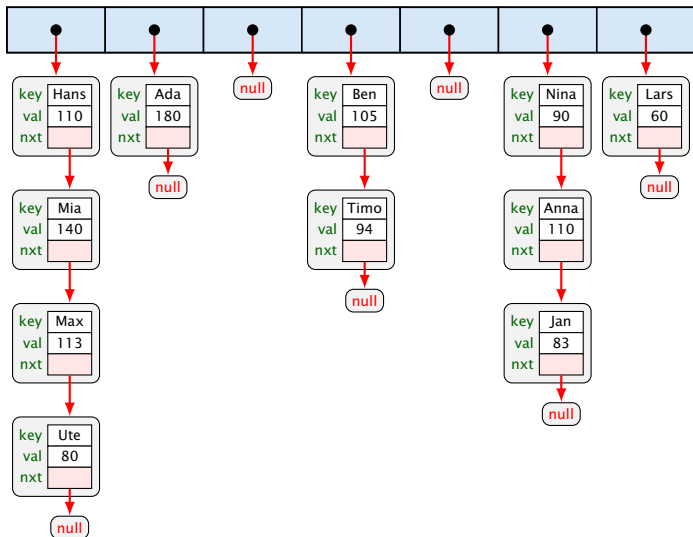


## 16.4 RW-Locks

### Ziel:

- ▶ eine Datenstruktur soll gemeinsam von mehreren Threads benutzt werden.
- ▶ Jeder Thread soll (gefühl) **atomar** auf die Datenstruktur zugreifen.
- ▶ Lesende Zugriffe sollen die Datenstruktur nicht verändern, schreibende Zugriffe dagegen können die Datenstruktur modifizieren.

# Hashtabelle



## 1. Idee: Synchronisiere Methodenaufrufe

```
1 public class HashTable<K,V> {
2     private List2<K,V> [] a;
3     private int n;
4     public HashTable (int n) {
5         a = new List2[n]; // Compilerwarnung
6         this.n = n;
7     }
8     public synchronized V lookup (K k) {...}
9     public synchronized void update (K k, V v) {...}
10 }
```

# Arrays und Generics

Wir sagen `a = new List2[n]` anstatt `a = new List2<K,V>[n]`. Letzteres führt zu einem Compilerfehler.

Wir bekommen hier eine Compilerwarnung, da wir Arrays mit Generics mischen.

Weder der Compiler noch die Laufzeitumgebung kann sicherstellen, dass das Array wirklich nur Elemente vom Typ `List2<K,V>` enthält.

Der Programmierer muss das sicherstellen...

# Diskussion

- ▶ Zu jedem Zeitpunkt darf nur ein Thread auf die HashTable zugreifen.
- ▶ Für schreibende Threads ist das evt. sinnvoll.
- ▶ Threads, die nur lesen, stören sich gegenseitig aber überhaupt nicht!

⇒ **ReaderWriterLock**

- ▶ ist entweder im Lese-Modus, im Schreibmodus oder frei.
- ▶ Im Lese-Modus dürfen beliebig viele Leser eintreten, während sämtliche Schreiber warten müssen.
- ▶ Haben keine Leser mehr Interesse, ist das Lock wieder frei.
- ▶ Ist das Lock frei, darf ein Schreiber eintreten. Das RW-Lock wechselt nun in den Schreib-Modus.
- ▶ Im Schreib-Modus müssen sowohl Leser als auch weitere Schreiber warten.
- ▶ Ist ein Schreiber fertig, wird das Lock wieder frei gegeben. . .

# Implementierung

```
1 public class RW {
2     private int countReaders = 0;
3
4     public synchronized void startRead()
5         throws InterruptedException {
6         while (countReaders < 0) wait();
7         countReaders++;
8     }
9
10    public synchronized void endRead() {
11        countReaders--;
12        if (countReaders == 0) notify();
13    }
14    // continued...
```

# Implementierung

```
15     public synchronized void startWrite()
16         throws InterruptedException {
17         while (countReaders != 0) wait();
18         countReaders = -1;
19     }
20
21     public synchronized void endWrite() {
22         countReaders = 0;
23         notifyAll ();
24     }
25 }
```



# Diskussion

- ▶ Die Methoden `startRead()`, und `endRead()` sollen eine Leseoperation eröffnen bzw. beenden.
- ▶ Die Methoden `startWrite()`, und `endWrite()` sollen eine Schreiboperation eröffnen bzw. beenden.
- ▶ Die Methoden sind `synchronized`, damit sie selbst atomar ausgeführt werden.
- ▶ Die unterschiedlichen Modi eines RW-Locks sind mit Hilfe des Zählers `count` implementiert.
- ▶ Ein negativer Zählerstand entspricht dem Schreib-Modus, während ein positiver Zählerstand die Anzahl der aktiven Leser bezeichnet. . .

# Diskussion

- ▶ `startRead()` führt erst dann kein `wait()` aus, wenn das RW-Lock entweder frei oder im Lese-Modus ist. Dann wird der Zähler inkrementiert.
- ▶ `endRead()` dekrementiert den Zähler wieder. Ist danach das RW-Lock frei, wird ein(!) wartender Thread benachrichtigt; dieses ist automatisch ein schreibender Thread.
- ▶ `startWrite()` führt erst dann kein `wait()` aus, wenn das RW-Lock definitiv frei ist. Dann wird der Zähler auf -1 gesetzt.
- ▶ `endWrite()` setzt den Zähler wieder auf 0 zurück und benachrichtigt dann alle wartenden Threads.

# Die HashTable mit RW-Lock

```
1 public class HashTable<K,V> {
2     private RW rw;
3     private List2<K,V> [] a;
4     private int n;
5     public HashTable (int n) {
6         rw = new RW();
7         a = new List2[n];
8         this.n = n;
9     }
10 // continued...
```

## Die HashTable mit RW-Lock

```
11     public V lookup(K key) throws
12         InterruptedException {
13         rw.startRead();
14         int i = Math.abs(key.hashCode() % n);
15         V result = (a[i] == null) ? null :
16             a[i].lookup(key);
17         rw.endRead();
18         return result;
19     }
```

- ▶ Da `lookup()` nicht weiß, wie mit einem interrupt umzugehen ist, wird die Exception weiter geworfen.

## Die HashTable mit RW-Lock

```
20     public void update(K key, V value) throws
21                          InterruptedException {
22         rw.startWrite();
23         int i = Math.abs(key.hashCode() % n);
24         if (a[i] == null)
25             a[i] = new List2<K,V> (key,value,null);
26         else
27             a[i].update(key,value);
28         rw.endWrite();
29     }
```

- ▶ Die Methode `update(K key, V value)` der Klasse `List2<K,V>` sucht nach Eintrag für `key`. Wird dieser gefunden, wird dort das Wert-Attribut auf `value` gesetzt. Andernfalls wird ein neues Listenobjekt für das Paar `(key,value)` angefügt.

# Diskussion

- ▶ Die neue Implementierung unterstützt nebenläufige Lesezugriffe auf die HashTable.
- ▶ Ein einziger Lesezugriff blockiert aber Schreibzugriffe — selbst, wenn sie sich letztendlich auf **andere Teillisten** beziehen und damit unabhängig sind...
- ▶ Genauso blockiert ein einzelner Schreibzugriff sämtliche Lesezugriffe, selbst wenn sie sich auf **andere Teillisten** beziehen...

⇒ Eingrenzung der kritischen Abschnitte...

Erst suchen der richtigen Liste (**nicht synchronisiert**), dann Lesen/Schreiben dieser Liste (**synchronisiert**).

# Realisierung

```
1 class ListHead<K,V> {
2     private List2<K,V> list = null;
3     private RW rw = new RW();
4     public V lookup (K key) throws InterruptedException {
5         rw.startRead();
6         V result= (list==null) ? null : list.lookup(key);
7         rw.endRead();
8         return result;
9     }
10    public void update (K key, V value) throws
11        InterruptedException {
12        rw.startWrite();
13        if (list == null)
14            list = new List2<K,V>(key,value,null);
15        else list.update(key,value);
16        rw.endWrite();
17 } }
```

# Diskussion

- ▶ Jedes Objekt der Klasse `ListHead` enthält ein eigenes RW-Lock zusammen mit einer Liste (eventuell `null`).
- ▶ Die Methoden `lookup()` und `update` wählen erst (unsynchronisiert) die richtige Liste aus, um dann geordnet auf die ausgewählte Liste zuzugreifen...

```
// in der Klasse HashTable:  
public V lookup (K key) throws InterruptedException {  
    int i = Math.abs(key.hashCode() % n);  
    return a[i].lookup(key);  
}  
public void update (K key, V value)  
                throws InterruptedException {  
    int i = Math.abs(key.hashCode() % n);  
    a[i].update (key, value);  
}
```



# Lock Support in Java

- ▶ **ReentrantLock**

`lock()`, `unlock()`, `tryLock()`, `newCondition()`

`wait()`, `notify()` heißen `await()` und `signal()` und werden auf einem `Condition`-Objekt ausgeführt.

- ▶ Ein `ReentrantLock` kann in einer anderen Funktion/einem anderen Block (anders als `synchronized`) freigegeben werden.
- ▶ **Achtung:** Ein `Reentrant-Lock` kann nicht von einem anderen Thread freigegeben werden.

# Lock Support in Java

```
1 class Test {  
2     synchronized void put() throws  
3         InterruptedException {  
4         ...  
5         while (...) wait();  
6         ...  
7     }  
8 }
```

# Lock Support in Java

```
1 class Test {
2     Lock lock = new ReentrantLock();
3     Condition cond = lock.newCondition();
4     void put() {
5         lock.lock();
6         try {
7             ...
8             while (...) cond.await()
9             ...
10        }
11        finally {
12            lock.unlock();
13        }
14    }
15 }
```

# Semaphore Support in Java

- ▶ Klasse `Semaphore`,
- ▶ Konstruktor `Semaphore(int permits)`
- ▶ Hauptmethoden: `acquire()`, `release()`

Kann von einem anderen Thread `release()`d werden als von dem, der `acquire()` aufgerufen hat.

# RW-Locks in Java

- ▶ Klasse `ReentrantReadWriteLock`
- ▶ `readLock()`, `writeLock` gibt das zugehörige `readLock`, bzw. `writeLock` zurück.

```
1 ...  
2 ReentrantReadWriteLock rw;  
3 rw.readLock().lock();  
4 try {  
5     ...  
6 }  
7 finally { rw.readLock().unlock(); }  
8 ...
```

# Warnung

Threads sind nützlich, sollten aber nur mit Vorsicht eingesetzt werden. Es ist besser,

- ▶ ... **wenige** Threads zu erzeugen als mehr.
- ▶ ... **unabhängige** Threads zu erzeugen als sich wechselseitig beeinflussende.
- ▶ ... kritische Abschnitte zu **schützen**, als nicht synchronisierte Operationen zu erlauben.
- ▶ ... kritische Abschnitte zu **entkoppeln**, als sie zu schachteln.

# Warnung

Finden der Fehler bzw. Überprüfung der Korrektheit ist ungleich schwieriger als für sequentielle Programme:

- ▶ Fehlerhaftes Verhalten tritt eventuell nur gelegentlich auf. . .
- ▶ bzw. nur für bestimmte Scheduler.
- ▶ Die Anzahl möglicher Programm-Ausführungsfolgen mit potentiell unterschiedlichem Verhalten ist gigantisch.
- ▶ Heisenbugs.

Ein Heisenbug ist ein Bug, der verschwindet wenn man das System beobachtet...