Synchronisation von Variablen und Operationen (gemeinsamer Speicher)

- Vorbemerkung
- Beispiel für Wettlaufbedingungen
- Threadsicherheit
- Threadlokale Objekte
- Unveränderliche Objekte
- Threadsicherheit durch volatile-Deklaration
- Threadsicherheit durch atomare Operationen
- Threadsicherheit durch Lock-Mechanismen
- Deadlock
- Passives Warten

Vorbemerkungen generell

Die Folien sind etwas veraltet:

- Man sollte betonen, dass es mehrere Aufgaben gibt: a) sichere Kommunikation, b) Vermeidung von Störungen, c) Warten auf Ereignisse.
- Es sind so viele (zu viele?) Mechanismen besprochen, weil es (noch) keine gute Lösung gibt.
- Moderne Lösungen kommen etwas zu kurz, da sie in Java noch wenig verbreitet sind.
- Das größte Problem ist die Vermeidung von Störungen (Wettlaufbedingungen): untestbare Fehler, Performancenachteile, Deadlocks.
- Ein weiteres Problem ist die Unklarheit worauf es ankommt: Korrekte Programmierung oder maximale Effizienz?

Vorbemerkungen zum Schutz kritischer Abschnitte

Zunächst werden nur relativ einfache Konstrukte besprochen!

Probleme entstehen, sobald mehrerer Threads auf **gemeinsame Variable** zugreifen. Dies kommt daher, dass sowohl die Ausführungsumgebung (vom Compiler bis zur CPU) als auch die logische Sicht der Objektorientierung zunächst nur für den sequentiellen Ablauf in einem Thread ausgelegt sind.

Es entstehen eine Reihe von Problemen im Zusammenhang von:

- **Reordering** von Operationen
- **Sichtbarkeit** der Änderung von Variablen
- **Atomizität** von Operationen
- Beibehaltung von **Klasseninvarianten**

Notation: Threadsichere Klassen kennzeichne ich durch die selbstdefinierten Annotationen `@ThreadSafe` (*threadsicher*) und `@Immutable` (*unveränderlich*).

Gelegentlich hebe ich die Unsicherheit einer Klasse durch die Annotation `@NotThreadSafe` hervor. Später kommt noch `@GuardedBy` hinzu.

Welche Variablen sind lokal zu einem Thread, welche gemeinsam?

In Java haben wir:

Lokale Variable und Funktionsparameter: Diese Variablen liegen auf dem Stack. Sie sind nur innerhalb der laufenden Methode und (damit) auch immer nur in einem Thread sichtbar.

Threadlokale Variable (Klasse `java.lang.ThreadLocal`) sind nur in einem Thread sichtbar.

Objekte und ihre Inhalte sind in dem Heap gespeichert. Sie können von Verschiedenen Threads angesprochen werden. Es können zusätzlich threadlokale Kopien existieren, die nicht zwingend identisch mit dem Original sind (caching).

Klassenvariable verhalten sich wie die Inhalte von Objekten.

Die Probleme der Objektorientierung haben nur mit dem Zugriff auf gemeinsame Variable zu tun!

Fehler sind durch Testen kaum zu erkennen. Daher kommt es darauf an, einfach und sicher zu programmieren!

Sichtbarkeit und Umordnung von Anweisungen

Initialisierung: `a = b = x = y = 0;`

In Thread 1

```
1.1 b = 1;
1.2 x = a;
```

In Thread 2

```
2.1 a = 1;
2.2 y = b;
```

Welche Ergebnisse sind für x und y möglich?

0,1: Reihenfolge 1.1, 1.2, 2.1, 2.2
1,0: Reihenfolge 2.1, 2.2, 1.1, 1.2
1,1: Reihenfolge 1.1, 2.1, 2.2, 1.2

Das Ergebnis hängt vom Zufall ab! (race condition, Wettlauf)

Ist auch das Ergebnis 0,0 möglich?

Sichtbarkeit und Umordnung von Anweisungen

Initialisierung: `a = b = x = y = 0;`

In Thread 1

```
1.1 b = 1;
1.2 x = a;
```

In Thread 2

```
2.1 a = 1;
2.2 y = b;
```

Welche Ergebnisse sind für x und y möglich?

0,1: Reihenfolge 1.1, 1.2, 2.1, 2.2
1,0: Reihenfolge 2.1, 2.2, 1.1, 1.2
1,1: Reihenfolge 1.1, 2.1, 2.2, 1.2

Das Ergebnis hängt vom Zufall ab! (race condition, Wettlauf)

Ist auch das Ergebnis 0,0 möglich?

0,0: Reihenfolge 1.2, 2.1, 2.2, 1.1 -- Umordnung der sequentiellen Anweisungen!

Es ist auch denkbar, dass in dem beobachtenden Thread die aktuellen Werte von
x und y nicht bekannt sind.

Race Conditions und Invarianten

Durch die zusätzliche Freiheit der Nebenläufigkeit entstehen neuartige Probleme (gilt auch bei prozeduraler Programmierung, aber im Detail etwas anders).

In Bezug auf Korrektheit wurden im 2. Semester wichtige Grundregeln formuliert:

1. *Man kann die Korrektheit einer Methode überprüfen, indem man nachvollzieht, welche Anweisungen in der Methode ausgeführt werden und welche Auswirkungen die aufgerufenen Methoden haben.*
2. *Zu einer Klasse gehören Invarianten, d.h. Aussagen über die erlaubten Werte der Instanzvariablen, die beim Eintritt und beim Verlassen einer Methode erfüllt sein müssen.*

Die Verwendung dieser Regeln ermöglicht es, Methoden weitgehend unabhängig voneinander zu entwickeln.

Wenn es möglich ist, dass *während* der Ausführung einer Methode (innerhalb eines Thread) ein anderer Thread eine Methode desselben Objekts aufruft, ist die Korrektheit nicht mehr garantiert.

1. Beispiel für Interferenz

```
class Counter {
    private int count = 0;

    public void inc() {
      // Q: COUNT = count
1     int a = count + 1;
      // count = COUNT, a = COUNT + 1
2     count = a;
      // R: count = COUNT + 1
    }
    ...
}
```

Thread A und Thread B rufen `c.inc()` für <u>ein Counter-Objekt</u> auf (count=0).
Mögliche Reihenfolgen und Ergebnisse:

A1, A2, B1, B2: count = 2
A1, B1, A2, B2: count = 1
A1, B1, B2, A2: count = 1

B1, B2, A1, A2: count = 2
B1, A1, B2, A2: count = 1
B1, A1, A2, B2: count = 1

Das Ergebnis hängt wieder vom Zufall ab!

2. Beispiel für Interferenz

```
class Circle {
    private double x, y;           // Position auf der Kreislinie
    private final double c, s;     // cos(delta-phi), sin(delta-phi)

    public Circle(double radius) {
        c = Math.cos(0.001); s = Math.sin(0.001);
        x = radius; y = 0.0;
    }
                                   // Problem bei Unterbrechung:
    public void nextPoint()   {              // kein Problem
        double newX = x * c - y * s;         // newX ist falsch
        double newY = x * s + y * c;         // läuft wieder zurück
        x = newX;                            // x passt nicht zu y
        y = newY;                            // kein Problem
    }
```

Probleme:

- Zählprobleme (wie vorher)
- Verlassen der Kreislinie, wenn bei der Veränderung x und y nicht zusammenpassende Werte kriegen (Typinvariante verletzt)

Vorsicht: es sind nur Unterbrechungen *zwischen* den Anweisungen diskutiert !!

Es gibt zwei Arten von Zugriffskonflikten

Write/Write-Konflikte entstehen, wenn mehrere Threads gleichzeitig in einem kritischen Bereich operieren. Keiner der Threads kann bei diesem Konflikt, ihre Aufgabe richtig zu Ende führen, da andere Threads dazwischenfunken und die gerade geschriebenen Werte teilweise wieder zerstören.

Read/Write-Konflikte entstehen, wenn während der Zeit in der ein schreibender Thread Teile der Daten (inkonsistent) verändert hat, ein anderer Thread lesend zugreift (oder wenn während dem Lesevorgang die Daten verändert werden).

Es gibt keine Read/Read-Konflikte.

Kritische Abschnitte sind die Programmteile, in denen der Zustand verändert wird und die Programmteile der lesenden Zugriffe auf veränderliche Daten.

Die Aufgabe besteht darin (bei Bedarf) die kritischen Abschnitte eines Objekts geeignet zu schützen.

Thread-Sicherheit (thread safety)

Definition: *Ein Objekt ist **threadsicher**, wenn es von anderen Objekten, selbst in einer multithreading Umgebung, immer in einem gültigen Zustand gesehen wird.*

Der Zustand eines Objekts ist gegeben durch die Werte seiner Instanzvariablen! Klassenvariable gehören zum Klassen"objekt". Lokale Variable, gehören zu einem Thread (und sind damit unproblematisch).

Jedes Objekt sollte von außen stets in einem gültigen Zustand angetroffen werden. Aber nicht jedes Objekt sollte threadsicher programmiert sein, da damit beträchtlicher Laufzeitoverhead verbunden ist.

Für jedes Objekt sollte bekannt sein, ob es threadsicher ist oder nicht. Unveränderliche Objekte (z.B. Strings) sind automatisch threadsicher!

Es lässt sich nicht vermeiden, dass während der Ausführungszeit einer öffentlichen Methode ein Objekt zeitweise den gültigen Zustand verlässt. Threadsicherheit bedeutet, dass während dieser „kritischen" Zeit der Zugriff durch andere Threads verhindert wird.

Es ist nicht schlimm, wenn das Objekt komisch aussieht, solange das keiner sieht!

Maßnahmen zum Gewährleisten der Threadsicherheit

- **Objekte nur in einem Thread benutzen (confinement)**
- Threadlokale Variable nutzen
- Objekte dürfen erst nach der Konstruktion bekannt werden
- **Unveränderliche Objekte verwenden.**
- **Atomare Operationen** nutzen (Bibliothek)
- Gemeinsame Variable als **volatile** deklarieren
- Zugriff zu unsicheren Objekten durch **Objektsperren** sichern
- **Kommunikation** über sichere Bibliotheksklassen
- Minimieren der Interaktion von Threads durch **einfache Architektur**!

Zunächst bespreche ich die einfacheren Mechanismen. Die Objektsperre ist ineffizient und gefährdet die Lebendigkeit eines Programms.

Kennzeichnung von Objektsicherheit

In den Beispielen kennzeichne ich die Threadsicherheit durch Annotationen

- `@NotThreadSafe`
- `@ThreadSafe`
- `@Immutable` (unveränderlich = automatisch threadsicher)
- `@GuardedBy("…")` (wird durch ein angegebenes Monitor-Objekt geschützt)

Unsichere Objektkonstruktion

```
@NotThreadSafe
public class LeakingConstructor extends Thread {
    public final String name;

    public LeakingConstructor(String name) {
        Registry.register(this);   // Fehler
        start();                   // Fehler
        this.name = name;
    }
    …
}
```

`register()` macht das Objekt vor der Fertigstellung bekannt. Damit kann z.B. ein falscher Wert für die Konstante gefunden werden.

Ebenso wird durch das Starten des Threads bewirkt, dass mit einem unvollständig initialisierten Objekt gearbeitet werden kann.

Fazit: **Niemals einen Thread im Konstruktor starten.** Die this-Referenz nicht anderen Objekten vom Konstruktor aus bekannt machen.

Warnung: Die Reihenfolge ist nicht allein schuld! Konstanten gelten erst nach Ablauf des Konstruktors als „eingefroren“.

Problematisches „lazy“ Singleton-Muster

```
@NotThreadSafe
public class BadSingleton {
    private static BadSingleton instance = null;

    public static BadSingleton getInstance() {
        if (instance == null)
            instance = new BadSingleton();
        return instance;
    }
    ...
}
```

Dieses Muster ist sehr populär aber trotzdem schlecht! Besser ist meist die frühe Objektkonstruktion (final ist hier wichtig):

```
@ThreadSafe
public class EarlySingleton {
    private static final EarlySingleton instance = new EarlySingleton();

    public static EarlySingleton getInstance() {
        return instance;
    }
    ...
}
```

Unveränderliche Objekte

Vgl. auch Wertobjekte. Unveränderliche Objekte sind absolut unkritische Behälter für Werte.

Ein Objekt ist unveränderlich, wenn alle seine Komponenten unveränderlich sind und wenn alle Variablen final sind. Unveränderliche Objekte sind threadsicher.

```
@Immutable
public class Point{
    public final double x;
    public final double y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
}
```

Es ist bei solch einfachen Objekten auch nicht zwingend nötig, dass die Variablen gekapselt sind (solange keine Implementierungsgeheimnisse im Spiel sind).

Selbstverständlich darf bei unveränderlichen Objekten der Konstruktor die this-Referenz nicht vorzeitig nach außen geben.

Volatile

Änderungen an volatile-Variablen sind immer in der richtigen Reihenfolge sichtbar. Zusätzlich wird ab Java 5 auch garantiert, nach dem Zugriff auf eine volatile-Variable auch davor stattgefunden Veränderungen durch den ändernden Thread sichtbar sind.
Volatile garantiert auch, dass 64 bit Werte (`double` und `long`) atomar verändert werden.

```java
@ThreadSafe
public class FastCounter implements Runnable {
    private int count = 0;
    private volatile boolean stop = false;

    // Thread terminiert garantiert nur wegen volatile!
    public void run() { while (! stop) count++; }

    public static void main(String[] a) throws Exception {
        FastCounter c = new FastCounter();
        new Thread(c).start();
        Thread.sleep(1000);
        c.stop = true;
        System.out.println(c.count);
    }
}
```

Atomares Test & Set

Atomare Test & Set – Operationen bilden den Kern für die Implementierung aller effizienten Mechanismen der Nebenläufigkeit. Auch der Zähler lässt sich so realisieren. Es bietet aber auch die Möglichkeit für die Implementierung anderer Operationen

```
@ThreadSafe
public class Counter {
    private AtomicInteger counter = new AtomicInteger(0);

    public void increment() {
        for(;;) {
            // Vorbedingung: oldCount = X
            int oldCount = counter.get();
            int newCount = oldCount + 1;
            // Nachbedindung: oldCount = X
            if (counter.compareAndSet(oldCount, newCount)) return;
        }
    }
    public int getCount() {
        return counter.get();
    }
}
```

`compareAndSet` prüft, ob der aktuelle Wert gleich `oldCount` ist und setzt ihn genau dann auf den neuen Wert. Die Rückgabe informiert über den Erfolg der Aktion, die atomar abläuft.

Verwendung von atomaren Referenzen für veränderliche Objekte

```
@ThreadSafe
class Circle {
    private final AtomicReference<Point> Point p =
        new AtomicReference<Point>();
    private final double c, s;

    public Circle(double radius) {
        c = Math.cos(0.001); s = Math.sin(0.001);
        p.set(new Point(radius, 0.0));
    }

    public void nextPoint() {
        Point oldPoint, newPoint;
        do {
            oldPoint = p.get();
            newPoint = new Point(
                oldPoint.x * c - oldPoint.y * s,
                oldPoint.x * s + oldPoint.y * c);
        } while (! p.compareAndSet(oldPoint, newPoint));
    }
```

Atomare Referenzen sind eine einfache Variante des populären Musters STM = *Software Transactional Memory*

Kritischer Abschnitt und Objektsperre

Definition: *Unter einem **kritischen Abschnitt** versteht man eine Folge von Anweisungen, während der kein anderer Thread auf die verwendeten Variablen zugreifen darf. Der Zugriff auf einen kritischen Abschnitt wird durch ein Verfahren des **gegenseitigen Ausschluss** geregelt werden.*

Java:

- Kritische Abschnitte können über Bibliotheksmechanismen geschützt werden. Dabei wird zu Beginn des Bereichs die Methode **`lock()`** und am Ende **`unlock()`** aufgerufen.
- Alternativ kann ein Block (oder eine Methode) gemäß dem Monitorkonzept von Java durch **`synchronized`** `{ ... }` gesichert werden.
- In beiden Fällen verfügt ein Objekt über eine Sperre (lock) (beim Monitorkonzept kann das jedes beliebige Objekt sein), die den Zugang zu dem kritschen Bereich steuert.
- Ein Thread, der einen ungesperrten Bereich betritt, erhält die Sperre für das zugehörige Objekt. Er darf damit alle gesamten geschützten Bereiche des Objekts betreten (auch wenn diese über mehrere Methoden verteilt sind).
  Wenn er alle geschützten Bereiche verlassen hat, gibt er die Sperre wieder frei.
  Diese Eigenschaft heißt auch Wiedereintrittsfähigkeit (*reentrancy*).
- Ein Thread, der versucht, einen gesperrten Bereich zu betreten, wartet bis der Bereich frei wird und er die Sperre erhält.
- Die Sperre verwaltet eine Liste von wartenden Threads (entry-set). Beim Freiwerden der Sperre wird einer der wartenden Threads ausführungsbereit.

Schutz des kritischen Bereichs (Zähler-Beispiel)

```
import java.util.concurrent.lock.*;

@ThreadSafe
public class Counter {
    private Lock lock = new ReentrantLock();

    @GuardedBy("lock")      // die Sperre schützt folgende Variable:
    private int count = 0;

    public void increment() {
        lock.lock();        // Beginn des kritischen Bereichs
        try {
            count++;
        }
        finally {
            lock.unlock();  // Ende des kritischen Bereichs
        }
    }

    public int get() {
        lock.lock();        // garantiert Sichtbarkeit
        try { return count; }
        finally { lock.unlock(); }
    }
}
```

Besonderheiten der Objektsperre

- Eine Sperre schützt kritische Bereiche und dazugehörende Variable (Invariante!).
- Zusammengehörende Bereiche und Variablen müssen mit der gleichen Sperre geschützt werden.
- Fremde Bereiche/Variable werden mit einer anderen Sperre geschützt.
- `lock()` und `unlock()` bewirken die Sichtbarkeit der geschützten Variablen.
- `lock()` und `unlock()` müssen paarweise zusammen aufgerufen werden (try…finally).
- `Lock` schützt kritische Bereiche dynamisch.
- `synchronized` schützt kritische Bereiche statisch.

Monitorkonzept -- Prinzip

Der Ausgangspunkt für das **Monitorkonzept** ist die Erkenntnis, dass die Steuerung von Nebenläufigkeit einerseits einen Mechanismus für **gegenseitigen Ausschluss** und andererseits über einen Mechanismus für das **Warten auf das Eintreten einer Bedingung*** haben muss. Entwickelt von *Brinch-Hansen* und *Hoare* soll es einen sicheren Mechanismus für nebenläufige Programme bieten.

** Das Warten wird weiter unten besprochen, zunächst der gegenseitige Ausschluss.*

A monitor is essentially a shared class with explicit queues. (Brinch Hansen)

In Java ist das Monitorkonzept einfach aber auch unsicher implementiert.

Java's most serious mistake was the decision to use the sequential part of the language to implement the run-time support for the parallel features.

In 1975, Concurrent Pascal demonstrated that platform-independent parallel programs (even small operating systems) can be written as a secure programming language with monitors. It is astounding to me that Java's insecure parallelism is taken seriously by the programming language community a quarter of a century after the invention of monitors and Concurrent Pascal. It has no merit. (Brinch Hansen)

Mal ehrlich: Bisher hat niemand die perfekte Lösung für Nebenläufigkeit !

Monitorkonzept (Pseudocode angelehnt an Concurrent Pascal)

```
@NotJava
shared class Counter {                // alle Methoden geschützt
    // automatisch private und geschützt
    int count = 0;

    public void increment() {         // automatisch geschützt
        count++;
    }

    public T get(){                   // automatisch public
        return count;
    }
}
```

In reinen Monitorkonzept sind gemeinsame Objekte klar gekennzeichnet und garantiert sicher.

Übertriebene Sicherheit kann aber leicht zu gravierenden Fehlern führen (s.u.).

Monitorkonzept in Java (Variante 1)

```
@ThreadSafe
public class Counter {
    @GuardedBy("this")
    private int count = 0;

    public synchronized void increment() {
        count++;
    }

    public synchronized T get(){
        return count;
    }
}
```

Das This-Objekt verwaltet die Objektsperre.
Synchronized erklärt den Körper einer Methode zum kritischen Bereich.
Synchronized funktioniert reentrant und garantiert aktuelle Werte der Variablen.
Synchronized sorgt automatisch für die Freigabe der Sperre.
Synchronized ist nicht Teil der Signatur!

Monitorkonzept in Java (Variante 2)

```
@ThreadSafe
public class Counter {
    private Object lock = "ich bin die Sperre";

    @GuardedBy("lock")
    private int count = 0;

    public void increment() {
        synchronized (lock) {
            count++;
        }
    }

    public T get(){
        synchronized (lock) {
            return count;
        }
    }
}
```

Das angegebene Objekt verwaltet die Objektsperre. Es ist ein beliebiges Objekt.
Synchronized erklärt den Körper einer Blocks zum kritischen Bereich.
Die beiden Varianten können kombiniert werden, sofern sie sich auf dasselbe Objekt beziehen.
Diese Variante bietet die Möglichkeit, nur Teile einer Methode zu sichern.

Beispiel für Ablauf mit `synchronized`

```
class X {
    void a() {}
    sychronized void b(X other) {
        a(); c();           // kein Problem
        other.a();          // kein Problem
        other.b(..);        // muss evtl. auf lock von other warten
    }
    synchronized void c() {}

X xObj = new X();
T1: xObj.a();    T2: xObj.a();          // beliebige Reihenfolge
T1: xObj.b();
                 T2: xObj.c();          // T2 wartet bis xObj frei ist.
T1:   weiter in b()                     // T2 wartet
T1:   verlässt b()                      // T2 hat jetzt eine Chance
                 T2: weiter in c()      // und erhält das Lock
T1: xObj.b();                           // jetzt muss T1 warten
```

Probleme der Objektsperre

Idee des Sicherheitsfanatikers: „grundsätzlich sollten alle Methoden gesichert sein, damit keine Interferenz auftritt“!?

Diese Idee ist schlecht:

- **`Sperren`** erfordert *Aufwand* und senkt dadurch die Effizienz.
- **`Sperren`** erhöht bei paralleler Hardware die Zeiten bloßen *Wartens*, also senkt es nochmals die Effizienz.
- **`Sperren`** bedroht die *Lebendigkeit* eines Programms.

Lebendigkeit: „ein Programm ist lebendig, wenn alle geplanten Aktivitäten irgendwann ausgeführt werden.“

Deadlock: Mehrere Threads warten auf die Freigabe von Ressourcen und blockieren sich dabei gegenseitig. Hier geht es um die Freigabe von Sperren.

Bei einem Deadlock sind mindestens zwei Threads und mindestens zwei Objektsperren beteiligt.

Deadlock-Beispiel (nested monitor)

```
@NotThreadSafe
class NestedMonitor extends Thread {
    private Deadlock p;

    public synchronized void setPartner(NestedMonitor p) {this.p = p;}

    synchronized int tueWas() { ... }

    public synchronized void run() { ... p.tueWas(); ... }
}
NestedMonitor a = new NestedMonitor();
NestedMonitor b = new NestedMonitor();
a.setPartner(b);
b.setPartner(a);
a.start(); b.start();
```

Möglicher Ablauf:

1. *a* startet `run()` und erhält die Sperre von *a*.
2. *b* startet `run()` und erhält die Sperre von *b*.
3. *a* versucht `tueWas()` zu rufen und wartet auf die Sperre von *b*.
4. *b* versucht `tueWas()` zu rufen und wartet auf die Sperre von *a*.
5. Die beiden Threads warten gegenseitig aufeinander (deadlock)

Einfache Faustregeln zum Vermeiden von Deadlocks:

Die folgenden Regeln gelten für Klassen, deren Objekte gleichzeitig von mehreren Threads angesprochen werden können:

Sperre immer fremden Zugriff, wenn Instanzvariablen eines Objekts verändert werden.
Andernfalls können unstimmige Veränderungen vorgenommen werden.

Sperre immer fremden Zugriff beim Lesen von Instanzvariablen, die von anderen Threads verändert werden könnten.
Anderfalls können unstimmige Werte zurückgegeben werden (Alternative: volatile).

Sperre fremden Zugriff nach Möglichkeit nicht, wenn Methoden auf anderen Objekten aufgerufen werden.
Andernfalls können Verklemmungen (deadlock) entstehen.

Die Objektsperre ist ein mächtiger und gefährlicher Mechanismus.
Gemeinsame Objekte sollten sich möglichst einfach verhalten!

Verbessertes Beispiel (kein guter Stil, da nicht einfach!)

```
@ThreadSafe
class NoDeadlock extends Thread {
    private NoDeadlock p;
    public synchronized void setPartner(NoDeadlock p) {
        this.p = p;}

    public void run() {
        synchronized(this) { ... } // Block 1
        // gültiger Zustand !!!
        int x = partner.tueWas();  // x = lokale Variable !!
        // gültiger Zustand !!!
        synchronized(this) { ... } // Block 2
    }
    synchronized int tueWas() { ... }
}
```

Hier kann kein Deadlock entstehen, da das Objekt seine Sperre freigibt, ehe es ein anderes Objekt aufruft. Also kann a auf b warten. Aber nicht gleichzeitig b auf a.

Nach Block1 muss das Objekt in einem erlaubten Zustand sein (Klasseninvariante).

Warten auf Bedingungen

Zur Entkopplung von Threads werden häufig Warteschlangen verwendet. Der „gerufene" Thread holt sich die Daten, die ein anderer Thread in die Schlange gesteckt hat (Erzeuger-Verbraucher-Muster).
Das Herausholen mit `get()` muss warten, wenn keine Daten vorliegen.
Natürlich gibt es bereits passende Bibliotheksklassen. Es geht hier ums Prinzip.

```
@NotThreadSafe
class BlockingQueue<T> {
    private List<T> q;    // q speichert die Daten

    public void put(T data) {
        q.add(data);
        melde, dass Daten vorliegen
    }

    public T get() {
        warte so lange wie q.isEmpty();
        return q.remove(0);
    }
}
```

Beim Warten muss die Objektsperre freigegeben werden! Der Mechanismus des Wartens ist deshalb mit der Sperre gekoppelt.

Fehlerhaftes aktives Warten (busy waiting – schlecht!!)

Die einfachste Idee ist, in einer while-Schleife immer wieder die Bedingung abzufragen:

Das vergeudet Prozessorzeit und ist falsch:

~~`while (q.isEmpty()) /* nichts */ ;`~~

Das ist falsch:

~~`while (q.isEmpty()) Thread.yield() ;`~~

Auch das ist falsch:

~~`while (q.isEmpty()) Thread.sleep(delay) ;`~~

Die meisten Anwendungen des aktiven Wartens vergeuden Prozessorzeit. Sie sind zudem falsch, wenn nicht auf die Sichtbarkeit der Bedingungen geachtet wird.

(Polling geht, wenn man nicht Warten muss, z.B. wenn man arbeitet solange eine Bedingung (nicht) erfüllt ist. Dann muss aber auch die Sichtbarkeit gewährleistet sein!)

Monitorkonzept in Java

Das wesentliche Merkmal ist, dass der Programmierer in Java selbst auf die Einhaltung einiger Regeln zu achten hat.

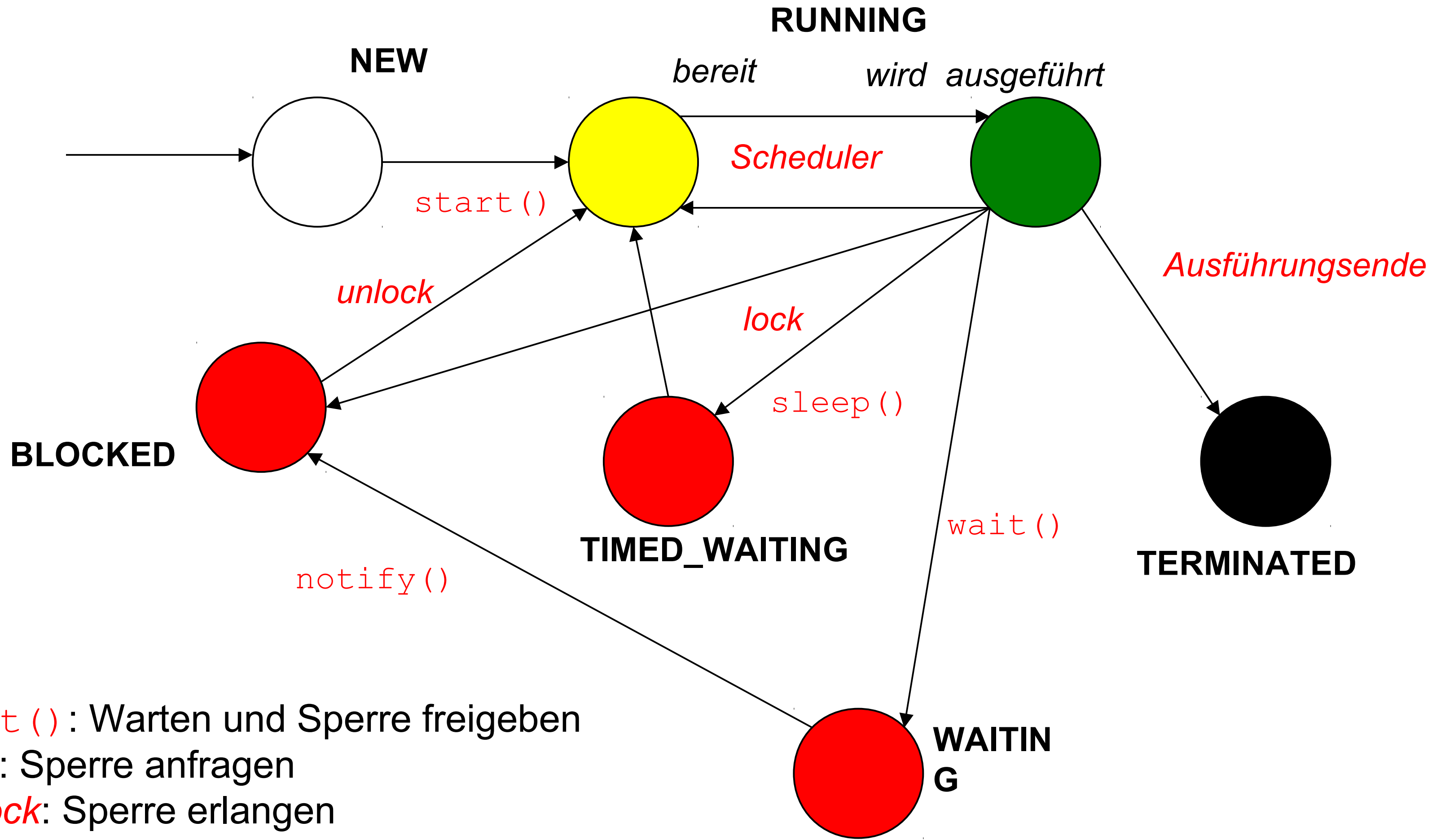
Auch in Java ist das Monitorkonzept so realisiert, dass sowohl die Objektsperre als auch das passive Warten mit einem Objekt verbunden ist. Allerdings enthält ein Java-Objekt nur eine Warteliste. Die einzelnen Mechanismen sind in Java nicht so weitgehend verknüpft wie in dem reinen Monitorkonzept.

In Java wird passives Warten durch den Aufruf `obj.wait()` veranlasst, der den entsprechenden Thread deaktiviert, bis ein anderer Thread für das Objekt `obj`, mittels `obj.notify()` oder `obj.notifyAll()` ein Aufwecksignal erzeugt hat.

In Java sind Ausschluss und Warten auch dadurch verknüpft, dass sowohl `wait()` als auch `notify()`/`notifyAll()` den Besitz der Objektsperre voraussetzen und dass während dem Warten auf das Aufwecksignal eines Objekts dessen Objektsperre zurückgegeben wird.

Java ermöglicht es a) Probleme des Monitorkonzepts zu vermeiden, und macht es b) möglich, Bibliothekslösungen anzubieten. Dafür sind aber die Basismechanismen sehr primitiv. Man braucht in Java-Anwendungen höhere Bibliothekslösungen.

Genaueres Zustandsdiagramm der Threadausführung in Java



Funktionsweise

In beiden Varianten ist die Funktionsweise des Wartens gleich:

- `wait`/`await` gibt die Sperre zurück und legt den Thread in den Wait-Set
- `notify`/`signal` bringt einen Thread aus dem Wait-Set in den Entry-Set
- `notifyAll`/`signalAll` bringt alle Threads aus dem Wait-Set in den Entry-Set
- Das Warten kann mit einem Timeout versehen werden.

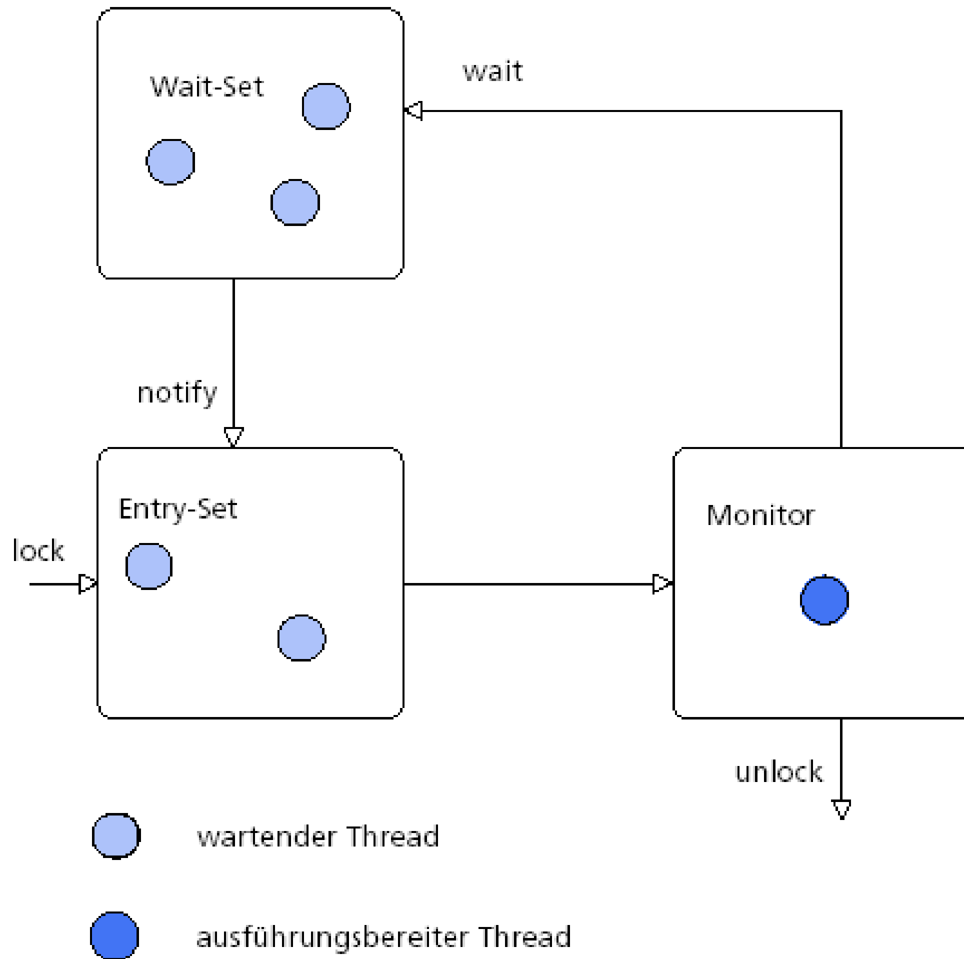
Es gibt aber auch Unterschiede:

- Mit einem Lock können mehrere Bedingungen und damit mehrere Wait-Sets verbunden sein.
- Bei Lock gibt es auch `waitNonInterruptible`, das keine `InterruptedException` wirft.

Zu beachten (vor allem beim Monitor): Sicherer (und lesbarer) Stil verlangt, dass das Warten mit einer Bedingung verknüpft ist, die in einer Schleife! abgefragt wird. Ein bloßes if ist falsch!

Beachten Sie den Merksatz: Threadsicher kann man nur dann programmieren, wenn man sich an klare Regeln hält!

Funktion eines Monitor-Objekts (bzw. Lock + Condition)



Monitor-Mechanismen für passives Warten

Alle nötige Funktionalität wird von der Klasse `Object` geerbt. Daher kann jedes Objekt für eine Wartebedingung stehen.
Allerdings muss die ausführende Thread zum Zeitpunkt des Aufrufs dieser Methoden über die Sperre des Objekts verfügen (wird zur Laufzeit überprüft).

`obj.wait();`

Warte und gib die Sperre frei.
Wenn der Thread ein notify()-Signal erhält, versucht er die Sperre wieder zu bekommen (kann was dauern) und macht dann weiter.

`obj.wait(long msecs);`

Warte maximal msecs Millisekunden.

`wait()` kann von außen unterbrochen werden: `InterruptedException`.

`obj.notify();`

Sende einem auf `obj` wartenden Thread ein notify()-Signal.

`obj.notifyAll();`

Sende allen auf `obj` wartenden Threads ein notify()-Signal.

Korrektes Warten auf Bedingungen (Monitor)

Warten und Aufheben von Warten geschieht über zwei Methoden der Klasse `Object`: `wait()` und `notifyAll()` (evtl. als Optimierung `notify()`). Sie werden aufgerufen mit dem Objekt der Sperre in deren Besitz der Thread sein muss. Wenn dies nicht erfüllt ist, erfolgt eine `IllegalMonitorStateException`.

```
@ThreadSafe
class BlockingQueue<T> {
    @GuardedBy("this")
    private List<T> q;    // q speichert die Daten

    public synchronized void put(T data) {
        q.add(data);
        this.notifyAll();
    }

    public synchronized T get() throws InterruptedException {
        while (q.isEmpty() wait();
        return q.remove(0);
    }
}
```

Der Sicherheit und der Lesbarkeit halber, sollte man immer dieses Ideom verwenden!
Das Thema `InterruptedException` wird noch besprochen.

Locking Mechanismen

Zum Einschließen des kritischen Bereichs :

```
void lock()
void lockInterruptibly()   // Spezialfall
void unlock()
```

Zur Abfrage:

```
boolean tryLock()
boolean tryLock(long time, TimeUnit unit)
```

Das Erzeugen von Bedingungen:

```
Condition newCondition()
```

Warten (einschließlich Freigabe der Sperre und evtl. InterruptedException)

```
void await()
void awaitUninteruptibly()  // ohne InterruptedException
```

Signalisieren, dass es weitergeht::

```
void signal()
void signalAll()
```

Warten auf Bedingungen (Lock)

```
@ThreadSafe
class BlockingQueue<T> {
    private Lock lock = new ReentrantLock();
    private Condition queueNotEmpty = lock.condition();

    @GuardedBy("lock")
    private List<T> q;    // q speichert die Daten

    public void put(T data) {
        lock.lock();
        try {
            q.add(data); queueNotEmpty.signalAll();
        } finally { lock.unlock(); }
    }
    public T get() throws InterruptedException {
        lock.lock();
        try {
            while (q.isEmpty()) queueNotEmpty.await();
            return q.remove(0);
        } finally { (lock.unlock(); }
    }
}
```

notify() / notifyAll() und signal() / signalAll()

Die dargestellte Methode des Wartens ist robust gegen überflüssige (oder irreführende) Signale: sie führt aber manchmal zu unnötigem Test und erneutem Warten.

Die dargestelle Methode ist aber *zwingend* darauf angewiesen, das richtige Signal auch wirklich zu erhalten (andernfalls wartet sie vielleicht ewig).

Da `notifyAll()` mehr Signale sendet als `notify()`, ist es evtl. ineffizienter aber (fast) nie falscher als `notify()`.

Regel: wenn du nicht weißt, welches Konstrukt du verwenden sollst, dann nimm `notifyAll()` !

Bei `signal()` und `signalAll()` sind die Probleme ähnlich. Allerdings gibt es hier die Möglichkeit, für jedes Ereignis ein eigenes Bedindungsobjekt (`Condition`) zu verwenden. Damit ist die effiziente Verwendung von `signal()` viel einfacher möglich. Umgekehrt ist der Effizienznachteil von `signalAll()` dann aber auch geringer.

InterruptedException

`InterruptedException` ist in Java dazu gedacht, einen Thread vorzeitig von außen zu beenden (Cancellation). Darauf wird später noch mal kurz eingegangen.

Sie wird nur erzeugt durch die (Thread-Objekt)-Methode `interrupt()`.

`InterruptedException` ist eine checked exception, d.h. sie muss aufgefangen oder explizit weitergerreicht werden.

In den wenigsten Programmen wird `InterruptException` wirklich erzeugt. Daher macht es auch keinen großen Sinn, sich über ihre Behandlung Gedanken zu machen.

Es gibt keine wirklich gute Lösung.

Aber für den Fall, dass es sich nur um Klassen der eigenen Anwendung handelt (nicht um generelle Bibliotheksklassen) ist es am einfachsten, sie direkt zu ignorieren, indem man leere catch-Klauseln verwendet (ja, das ist nicht schön – der Fehler liegt aber bei den Java-Architekten).

Bei wiederverwendbaren Klassen (wie blockierenden Puffern) sollte man sie aber unbedingt weiterreichen!

Beispiel auf nächster Folie, Genaueres später.

Mögliche Lösungen zu InterruptedException

```
try {
    while (!Bedingung) wait();
    ...
    Thread.sleep(1000);
    ...
catch (InterruptedException neverHappens) { /* nichts */ }
```

Alternative zu /* nichts */:

```
try {
    ...
catch (InterruptedExeption noLongerChecked) {
    throw new RuntimeException(noLongerChecked);
}
```