Betriebssysteme

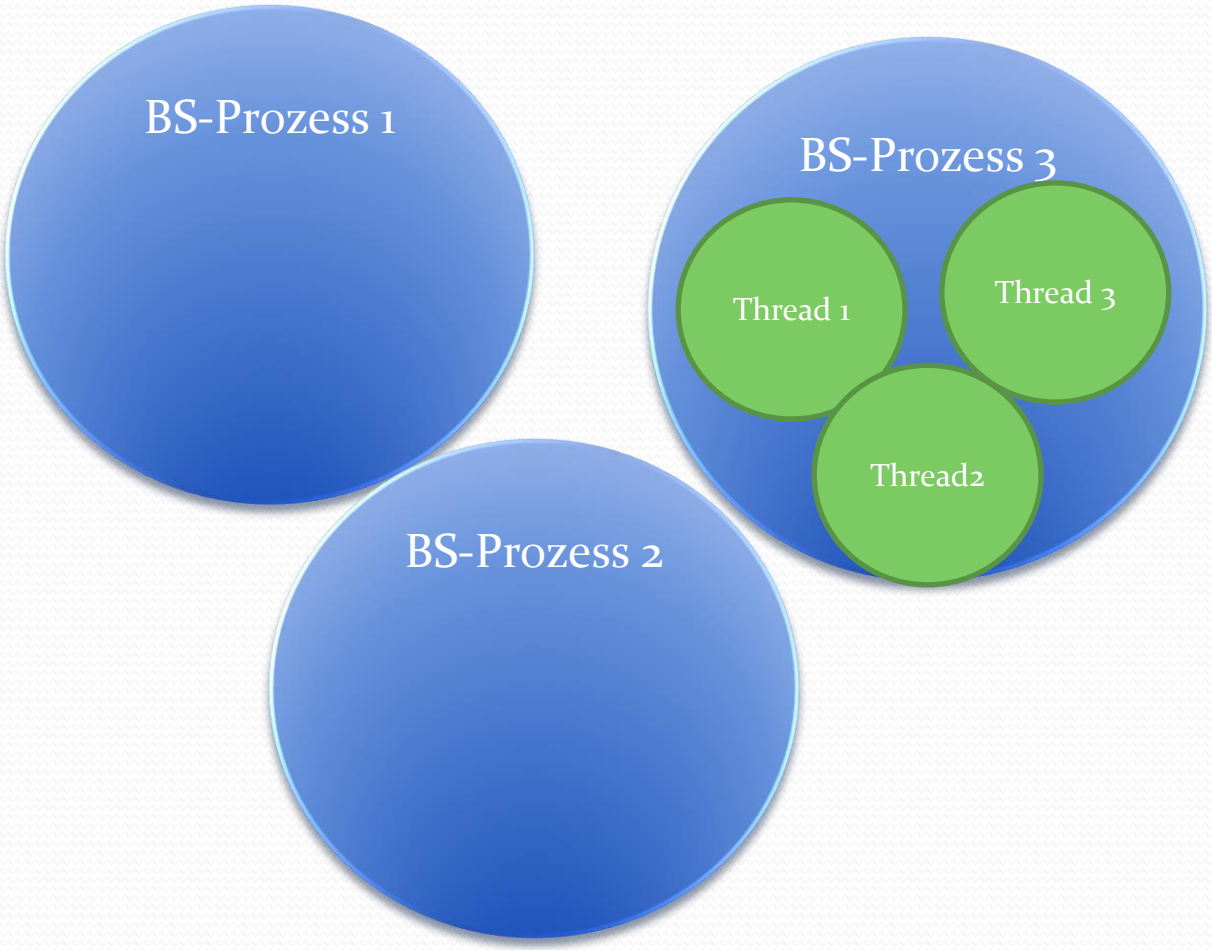
Threads unter Java

Literatur Verzeichnis

- Goll, Joachim; Heinisch Cornelia; Java als erste Programmiersprache; 8.Aufl. 2016; Springer Verlag

Java Virtuelle Maschine (JVM)

- Eine JVM läuft in einen Betriebssystemprozess (BS-Prozess) ab.
- Jeder Betriebssystemprozess hat seine eigen JVM, wenn mehre Java Programme in getrennten BS-Prozesse ablaufen sollen.



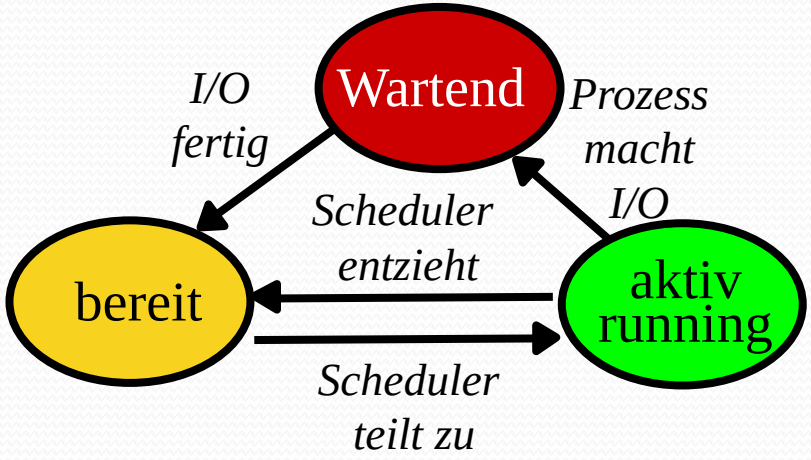
@Goll

Threads in Java

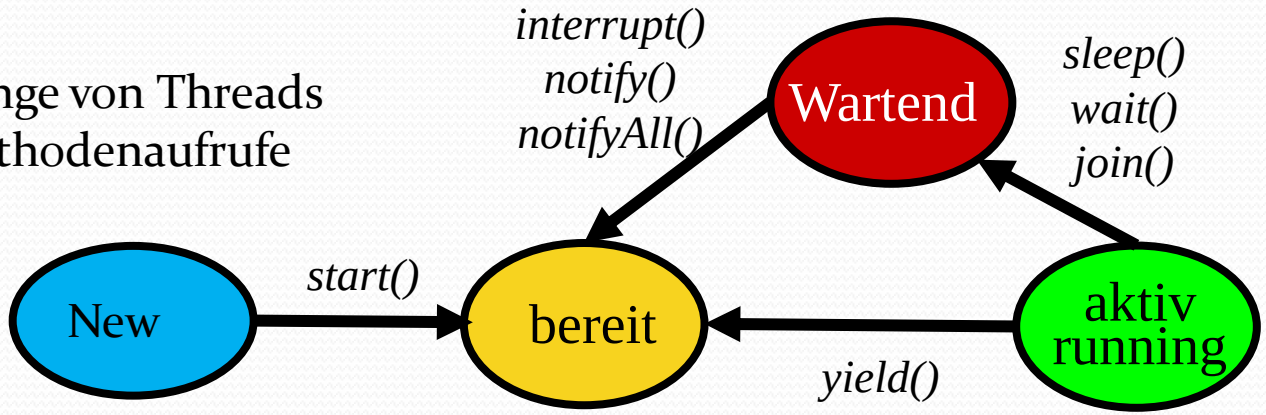
- Threads teilen sich:
  - den Heap
  - Code und Klassenvariablen
  - I/O-Kanäle
- Jeder Thread hat seinen eigenen
  - Befehlszähler
  - Registersatz
  - Stack zur Ablage der lokalen Variablen, Übergabeparameter und Befehlszeiger zum Rücksprung bei Methodenaufrufen

Prozesszustandsmodell

Zustandsübergangsdiagramm

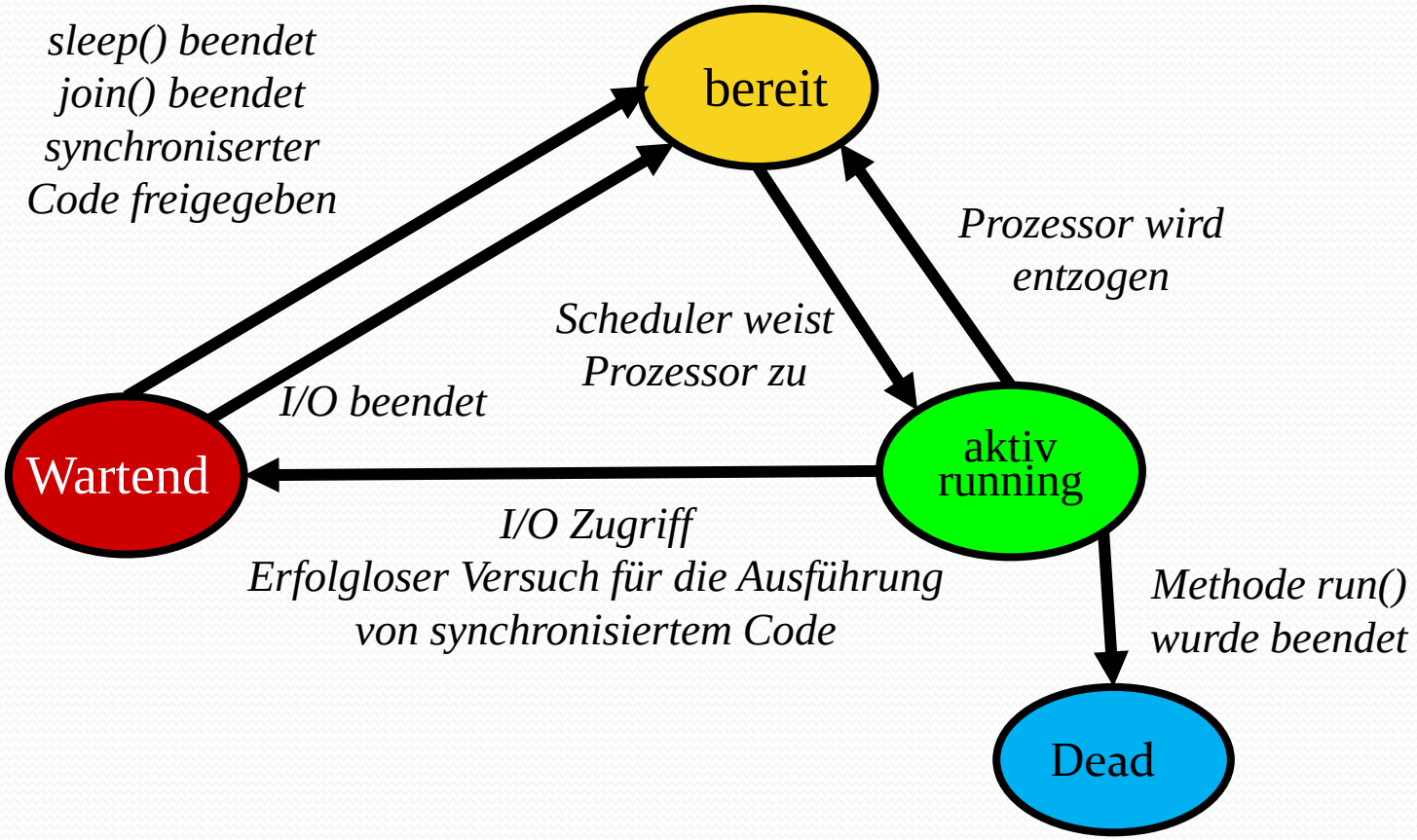


Zustandsübergänge von Threads
als Folge von Methodenaufrufe



@Goll

Die von der JVM verursachten Zustandsübergänge

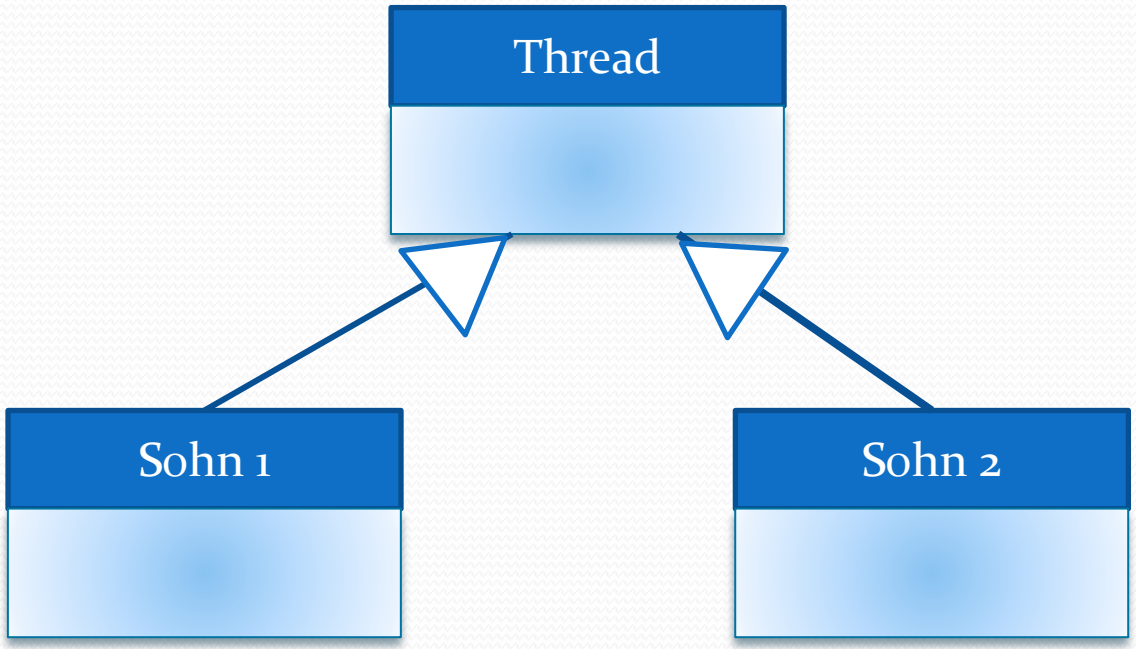


Programmieren von Threads in Java

- Es gibt 2 Möglichkeiten einen Threads zu programmieren
  - Durch eine direkte Ableitung der Klasse dieses Threads von der Klasse Thread
  - Übergabe eines Objektes, dessen Klasse die Schnittstelle Runnable implementiert, an ein Objekt der Klasse Thread.

Ableiten von der Klasse Thread

- Ableiten von der Klasse java.lang.Thread
- Die Methode run() muss überschrieben werden.
- Der Code in run() wird während des „aktiv running“-Zustands des entsprechenden Thread-Objekt ausgeführt.



Programmbeispiel Vererbung

Klasse Time

```
import java.util.*;

public class Time extends Thread{
        public void run() {
                GregorianCalendar d;
                int i = 0;
                while( i < 20) {
                        d = new GregorianCalendar();
                        System.out.println(
                        d.get(Calendar.HOUR_OF_DAY) +":" +
                        d.get(Calendar.MINUTE) + ":" +
                        d.get(Calendar.SECOND));
                        try {
                            Thread.sleep(1000);
                            } catch (InterruptedException e) {
                        } // try-Block
                        i++;
                } // while-Block
                System.out.println("Threadende");
        } // run
} // Klasse
```

Programmbeispiel Vererbung

Klasse Uhr

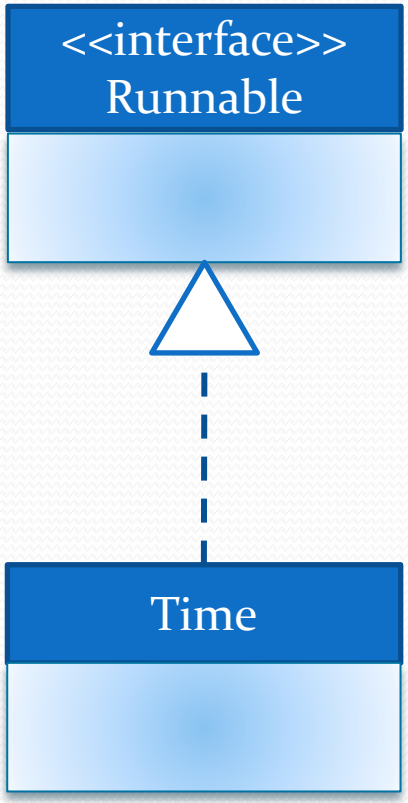
```
public class Uhr {
    public static void main(String[] args)
    {
    Time t = new Time();
    t.start();
    }
}
```

Ausgabe des Codes

17:32:55
17:32:56
17:32:57
17:32:58
17:32:59
17:33:0
17:33:1
17:33:2
17:33:3
17:33:4
17:33:5
17:33:6
17:33:7
17:33:8
17:33:9
17:33:10
17:33:11
17:33:12
17:33:13
17:33:14
Threadende

Implementieren der Schnittstelle

- Implementieren der Schnittstelle Runnable erlaubt, dass diese Klasse von einer anderen Klasse erben kann.
- Die Schnittstelle Runnable deklariert nur eine einzige Methode run().



Programmbeispiel Schnittstelle

```
import java.util.Calendar;
import java.util.GregorianCalendar;

public class Time1 implements Runnable {
	public void run() {
		GregorianCalendar d;
		int i = 0;
		while( i < 20) { // Block wird auf 20 Ausgaben beschränkt
			d = new GregorianCalendar();
			System.out.println(
				d.get(Calendar.HOUR_OF_DAY) +":" +
				d.get(Calendar.MINUTE) + ":" +
				d.get(Calendar.SECOND));
			try {
				Thread.sleep(1000);
			} catch (InterruptedException e) {
				// da in diesem Programm nie ein Exception auftritt, bräuchte man keine
Behandlung
				System.out.println("Oweija ein Interrupt!");
				return;
			} //try-Block
			i++;
		} // While-Block
		System.out.println("Threadende");
	} // run
}
```

Programmbeispiel Schnittstelle

```
public class Uhr1 {
    public static void main(String[] args)
    {
    Thread t = new Thread(new Time1());
    t.start();
    }
}
```

Threads gezielt beenden

- Es gibt mehrere Methoden um Threads zu terminieren
  - Aufruf der interrupt-Methode
    - sleep() erhält ein InterruptException und run() wird beendet.
    - Wird sleep() gerade ausgeführt und ist im Zustand „Wartend“ wird er in den Zustand „bereit“ und dann in den Zustand „aktiv running“ gebracht. Dann wird der interrupt behandelt.
    - Wird der interrupt früher aufgerufen, wird dieser erst ausgeführt wenn die Anweisung sleep() an die Reihe kommt.
  - Ein privates Datenfeld vom Typ boolean wird genutzt

Beenden von Treads (interrupted)

Klasse Time 2

```
import java.util.Calendar;
import java.util.GregorianCalendar;

public class Time2 implements Runnable {
    public void run() {
        GregorianCalendar d;
// Vorsicht Endlosschleife
        while( true) {
            d = new GregorianCalendar();
            System.out.println(
                    d.get(Calendar.HOUR_OF_DAY) +":" +
                    d.get(Calendar.MINUTE) + ":" +
                    d.get(Calendar.SECOND));
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                System.out.println("Interrupted!");
                System.out.println("Threadende");
                return;
            } //try-Block
        } // While-Block
    } // run
}
```

Interrupt Block

Beenden von Treads (interrupted)

Klasse Uhr 2

```
import java.io.*;

public class Uhr2 {
        public static void main(String[] args)
        {
            String kommando;
            Thread t = new Thread(new Time2());
            t.start();
            // Warten auf Benutzereingabe. Terminieren mit exit
            while(true) {
                BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
                try {
                kommando = reader.readLine();
                } catch( IOException e ) {
                    System.out.println("Eingabefehler");
                    break;
                } // try-Block
                if(kommando.equals("exit")) break; // Loop terminieren
            } // while
            t.interrupt(); // Thread ein Interrupt schicken
        } // main
    } // Klasse
```

Threads gezielt beenden

- Ein privates Datenfeld vom Typ boolean wird genutzt, welches man abfragt.
- Mit einer Methode der beenden() der Klasse Time setzt man diese Variable

Beenden von Treads (Variable)

Klasse Time 3

```
import java.util.Calendar;
import java.util.GregorianCalendar;

public class Time3 extends Thread {
        private boolean running = true;
        public void run() {
                GregorianCalendar d;
// Vorsicht Endlosschleife
                while( running) {
                        d = new GregorianCalendar();
                        System.out.println(
                                d.get(Calendar.HOUR_OF_DAY) +":" +
                                d.get(Calendar.MINUTE) + ":" +
                                d.get(Calendar.SECOND));
                        try {
                                Thread.sleep(1000);
                        } catch (InterruptedException e) {
                                System.out.println("Interrupted!");
                                return;
                        }
                } // While-Block
        } //run
        public void beenden() {
                running = false;
                System.out.println("Thread wird beendet");
        }
}
```

Beenden von Treads (Variable)

Klasse Uhr3

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class Uhr3 {
        public static void main(String[] args) throws IOException
        {
            String kommando;
            Time3 t = new Time3();
            t.start();
            // Warten auf Benutzereingabe. Terminieren mit exit
            while(true) {
                BufferedReader reader = new BufferedReader(new InputStreamReader (System.in));
                try {
                kommando = reader.readLine();
                } catch( IOException e ) {
                    System.out.println("Eingabefehler");
                    break;
                }
                if(kommando.equals("exit")) break; // Loop terminieren
            } // while
            t.beenden(); // Aufruf der Methode beenden
        } // main
    } // Klasse
```

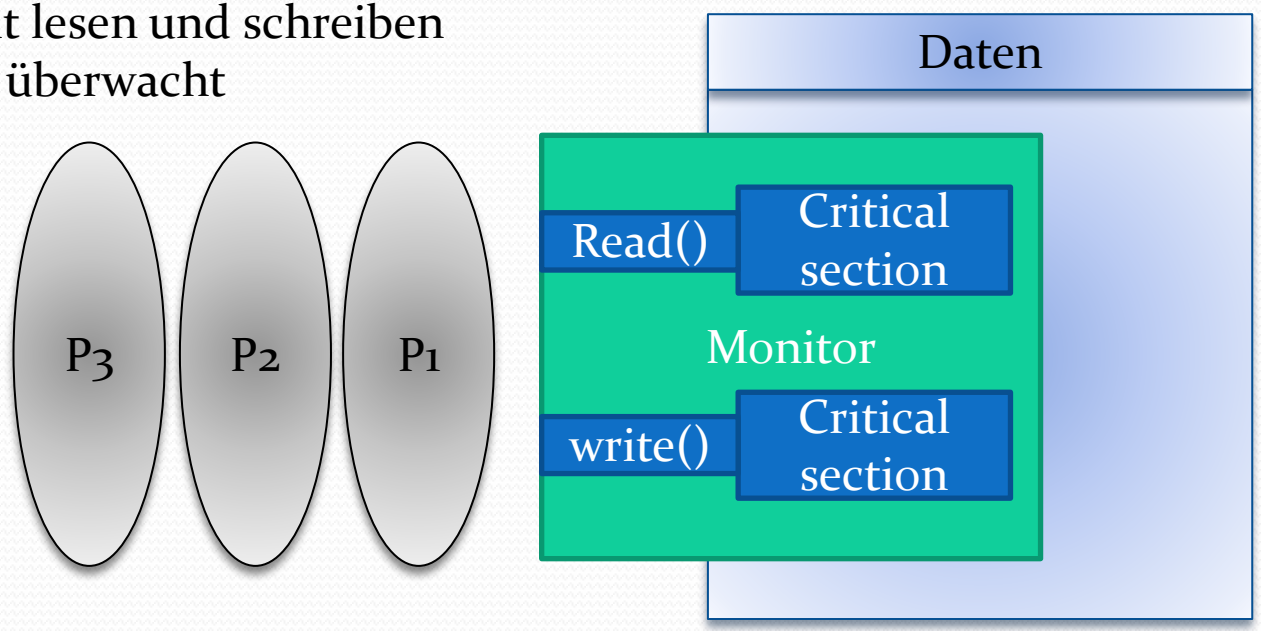
Zugriff auf gemeinsame Ressourcen

- Prinzip des wechselseitigen Ausschlusses
- Realisieren mit
  - Semaphore oder
  - mit dem Monitorkonzept in Java

Monitor

- Das Monitorkonzept von Hoare 1974 vorgeschlagen.
- Die Daten, auf denen im kritischen Abschnitt gearbeitet werden und die kritische Abschnitte selbst in einem zentralen Konstrukt zusammenfassen.
- In objektorientierten Sprachen lässt sich dies als ein Objekt mit speziellen Eigenschaften realisieren.

Kritischer Bereich mit lesen und schreiben durch einen Monitor überwacht



Monitor in Java

Generelle Anforderungen

- Eine spezielle Klasse mit folgenden Eigenschaften
  - Alle Daten der Klasse müssen private deklariert sein
  - Nur ein Thread kann zu jedem Zeitpunkt in einem Monitor aktive sein, d.h. jeder Monitor hat eine Sperre
  - Die Sperre kann mit einer beliebigen Anzahl von Bedingungen verwendet werden
  - Es ist Aufgabe der VM, den wechselseitigen Ausschluss der Monitoreingänge zu garantieren.

Monitor

- Einfacher zu überschauen als Semaphore
- Grundlegende Eigenschaften
  - Kritische Abschnitte, die auf den selben Daten arbeiten, sind Methoden eines Monitors
  - Ein Prozess betritt einen Monitor durch den Aufruf einer Methode des Monitors.
  - Nur ein einziger Prozess kann zur selben Zeit den Monitor betreten. Jeder andere Prozess muss warten, bis der Monitor wieder verfügbar ist.
  - Bedingt kritische Abschnitte:
    - Ein Prozess, der den Monitor betritt, kann, wenn die angeforderten Ressourcen noch nicht da sind, mittels wait() seinen Prozess unterbrechen und den Monitor freigeben. Ein andere Prozess kann die Monitor betreten.
    - Mittels signal() kann ein Prozess, der den Monitor verlässt, den wartenden Prozessen mitteilen, dass der Zugang wieder freigegeben ist.

Monitor in Java

- Wechselseitiger Ausschluss (Mutual Exclusion) wird in Java mittels des Monitor-Konzepts realisiert.
- Das Schlüsselwort in Java, was einen Monitor anzeigt, ist synchronized.
- In Klassen können so die Methoden zu Methoden mit Monitoreigenschaften umgewandelt werden.
- Die anderen Methoden sind, dann aber nicht durch den Monitor geschützt.

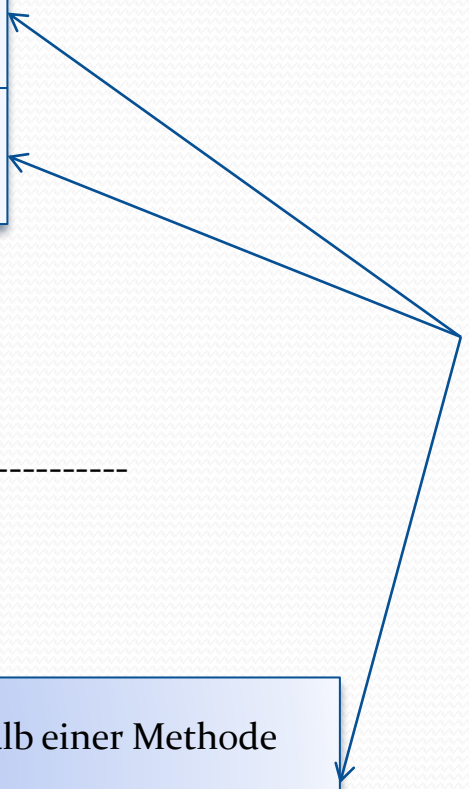
Monitor in der Java Realisierung

- Java setzt diese Konzept nur teilweise um
  - Attribute einer Klasse müssen bei Java nicht private sein
  - Nicht alle Methoden müssen als synchronized deklariert sein
- Das führt zum unsicheren Umgang und wurde kritisiert.

Monitor in Java Beispiel

```
public class Beispiel1
        {
        ...
        public static synchronized void methode1() {
                // .. Kritischer Abschnitt
        }
        public synchronized void methode2() {
                // .. Kritischer Abschnitt
        }
        public void methode3() {
                // .. Kritischer Abschnitt
        }
...
}
```

```
public class Beispiel2
        {
        ...
        public void methode1() {
                Object schluessel = Schluessel.getSchluessel();
                synchronized(schluessel)  // kritischer Bereich innerhalb einer Methode
                        {
                        // .. Kritischer Abschnitt
                        }
                }
        ...
        }
```



Beispiel Monitor in Java

Lesen und Beschreiben einer Pipe

- Es gibt 4 Klassen
  - Pipe, welche die Pipe zur Verfügung stellt mit den beiden Methoden:
    - Write(): zum Reinschreiben
    - Read(): zum Lesen aus der Pipe
  - Reader, welche von der Pipe liest
  - Writer, welcher in die Pipe schreibt
  - Test1 ist eine Klasse, welche eine Pipe anlegt und einen Schreiber und einen Leser initialisiert.

Klasse Pipe

```
public class Pipe {
        private int[] array = new int[3];
        private int index = 0;

// Methoden

        public synchronized void write(int i) {
            if (index == array.length){
                System.out.println(
                    "Schreibender Thread muss warten");
                try {
                    this.wait();
                } catch(InterruptedException e){

                } // try-Block
            } // if-Block
            array[index] = i; // Wert i speichern
            index ++; // index erhöhen
            if(index == 1) this.notify(); // Leser aufwecken
            System.out.println("Wert:" + i + " gechrieben");
        } // write
```

```
public synchronized int read() {
            int value;
            if (index == 0){ // Kein Wert vorhanden
                System.out.println(
                    "Lesender Thread muss warten");
                try {
                    this.wait();
                } catch(InterruptedException e){

                } // try-Block
            } // if-Block
            value = array[0]; // Wert auslesen
            index --; // index erniedrigen
// Werte nach unten kopieren
            for(int i = 0; i < index; i++) array[i]=array[i+1];
// Schreiber aufwecken
            if(index == array.length-1) this.notify();

            System.out.println(
            "Wert:" + value + " ausgegeben");
            return value;
        } // write
}
```

Klasse Reader und Writer

```
// Klasse, die aus der Pipe liest
public class Reader extends Thread {
      private Pipe pipe;
// Methoden

      public Reader(Pipe p){
            pipe = p;
      } // Konstruktor

      public void run() {
            int empfang = 0;
// Eine 0 kennzeichnet das Ende des Empfangs
            while((empfang = pipe.read()) != 0);
      } // run
}
```

```
// Klasse, die in die Pipe beschreibt

public class Writer extends Thread {
      private Pipe pipe;
      private int[] sendeArray =
          {1,2,3,4,5,6,7,8,9,0};
// Mit 0 wird der Block terminiert
      public Writer(Pipe p){
          pipe = p;
      }
      public void run() {
          for (int i = 0; i < sendeArray.length;i++){
              pipe.write(sendeArray[i]);
          }
      }
}
```

Klasse Test1

```
// Klasse zum Testen der Pipe
public class Test1 {
    public static void main(String arg[]){
        Pipe pipe = new Pipe();
        Reader readerThread = new Reader(pipe);
        Writer writerThread = new Writer(pipe);
        readerThread.start();
        writerThread.start();
    }
}
```