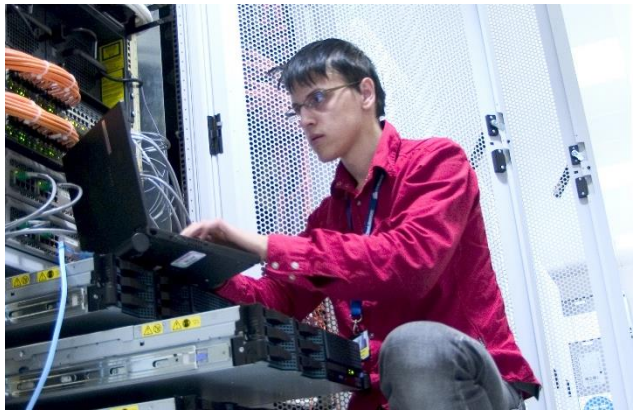
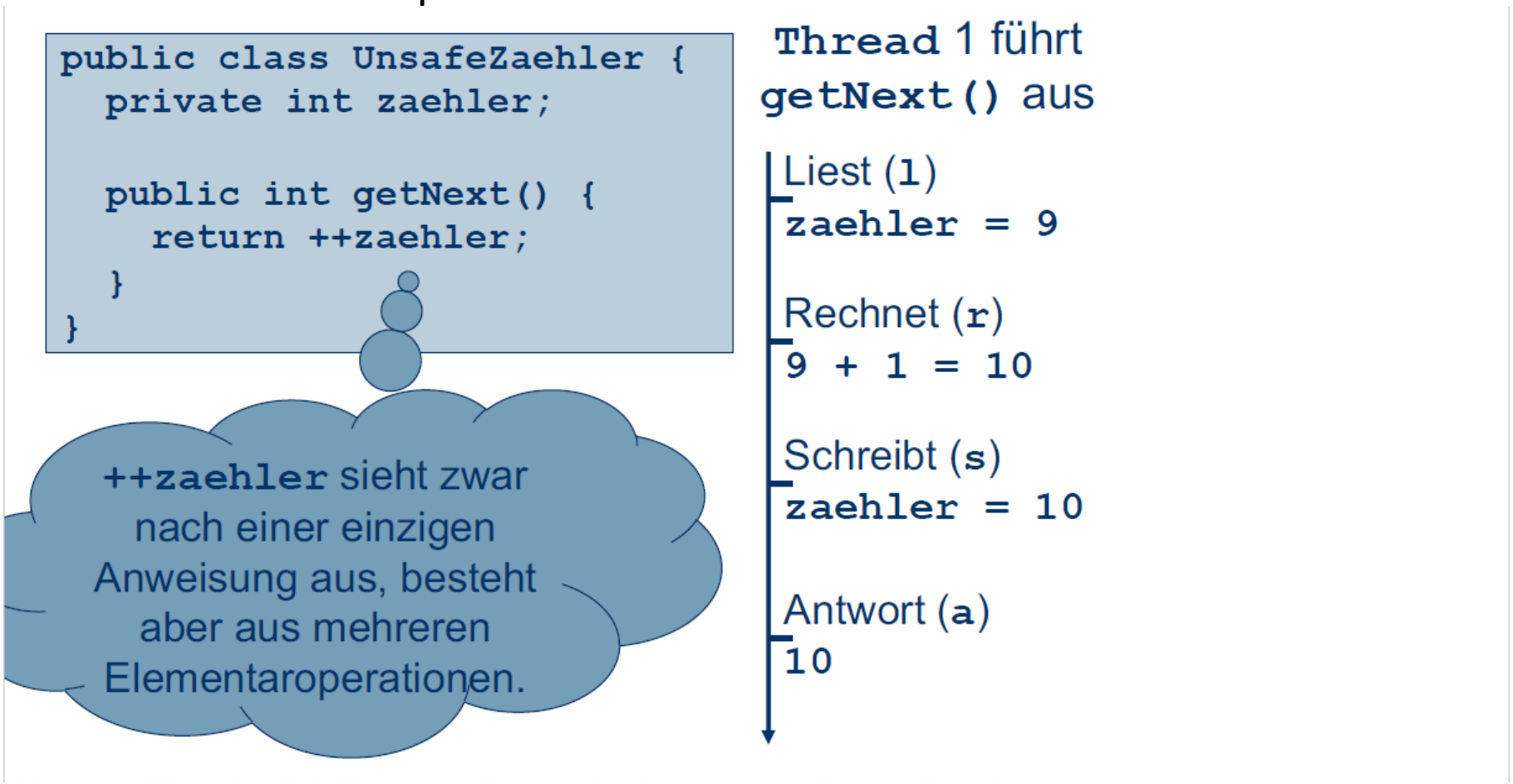
EMIL-VON-BEHRING GYMNASIUM

Wettlaufsituation (Race condition)



Wettlaufsituation (Race condition)

- Der folgende Code funktioniert richtig, solange nur ein Aktivitätsfaden im Spiel ist:



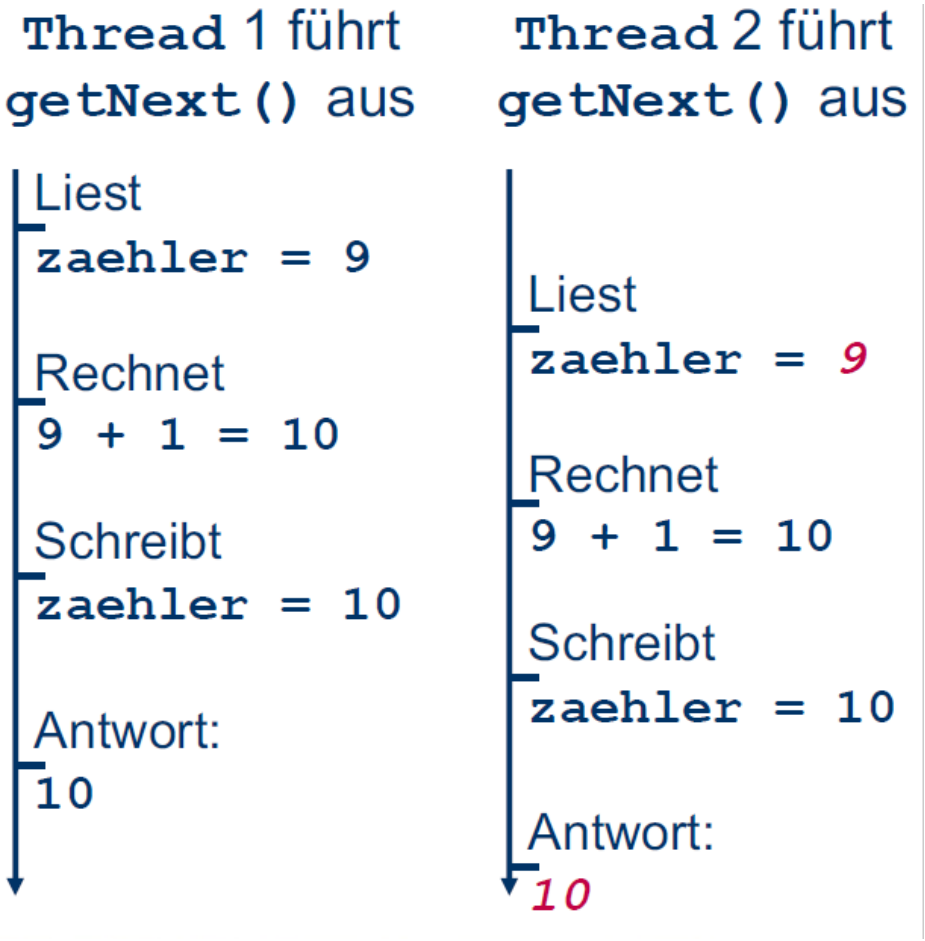
Wettlaufsituation (Race condition)

- Der Code ist aber fehlerhaft, wenn mehr als ein Aktivitätsfaden im Spiel ist:

```
public class UnsafeZaehler {
  private int zaehler;

  public int getNext() {
    return ++zaehler;
  }
}
```

Diese Verschränkung der Ausführung der elementaren Operationen zeigt den Fehler.



(Achtung: Problem vereinfacht dargestellt.)

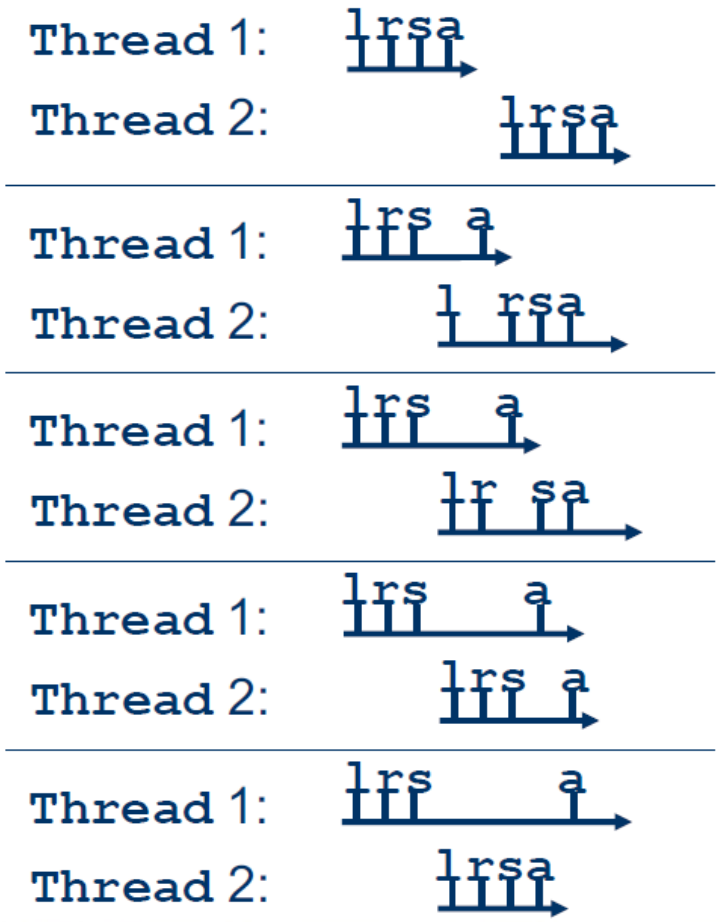
Wettlaufsituation (Race condition)

- Der Fehler wird nur in bestimmten zeitlichen Situationen sichtbar. Meistens geht's gut:

```
public class UnsafeZaehler {
  private int zaehler;

  public int getNext() {
    return ++zaehler;
  }
}
```

OK:



Einsicht: Das Lesen `l` von `Thread 2` muss nach dem Schreiben `s` durch `Thread 1` erfolgen. Sonst beliebig.

Wettlaufsituation (Race condition)

- Das zaehler-Beispiel zeigt die klassische „Lesen-Ändern-Schreiben-Wettlaufsituation“. Sie kann immer dann auftreten (= notwendige Bedingung),
  - wenn eine Variable von mehr als einer Aktivität gelesen, abhängig vom Wert geändert und dann das Ergebnis zurück in die Variable geschrieben wird, also
  - wenn Lesen, Ändern und Schreiben nicht als ununterbrechbare Einheit (atomare Aktion) geschehen.

Wettlaufsituationen sind extrem problematisch

- Auch wenn der Ablaufplaner die zum Fehler führende zeitliche Verschränkung nie verwendet
  - auf Ihrem Rechner
  - auf Ihrer JVM
  - bei der aktuellen Lastsituation des Betriebssystems
  - …

  ist das Programm dennoch fehlerhaft.

Ärger nach Auslieferung, nach Portierung, …

- Tritt die Fehlersituation auf, dann kann durch
  - das Einführen weiterer Anweisungen zur Fehlersuche,
  - das Aktivieren eines „Debuggers“
  - …

  das Auftreten des Fehlerzustands verhindert werden.

„Heisenbug“

Wettlaufsituationen sind extrem problematisch

- Therac-25 (Gerät zur Strahlentherapie):
  - Fehler im Bereich des nebenläufigen Programms haben zu schweren Verletzungen und Todesfällen geführt.
- Mars Rover:
  - Interaktion zwischen nebenläufigen Aktivitäten war fehlerhaft. Deshalb musste das System regelmäßig ganz neu gestartet werden. Dadurch konnte nur ein Bruchteil der geplanten Mission ausgeführt werden.
- Heute „beliebt“:
  - Handy-Prozess kann auf „no-sleep“ schalten, damit irgend eine Aktion sicher ausgeführt wird (ehe der Standby-Modus einsetzt).
  - Mehrere Prozesse lesen und schreiben diesen Wert -> Wettlaufsituationen (doch zu früh Standby, nie Standby dann Akku leer).

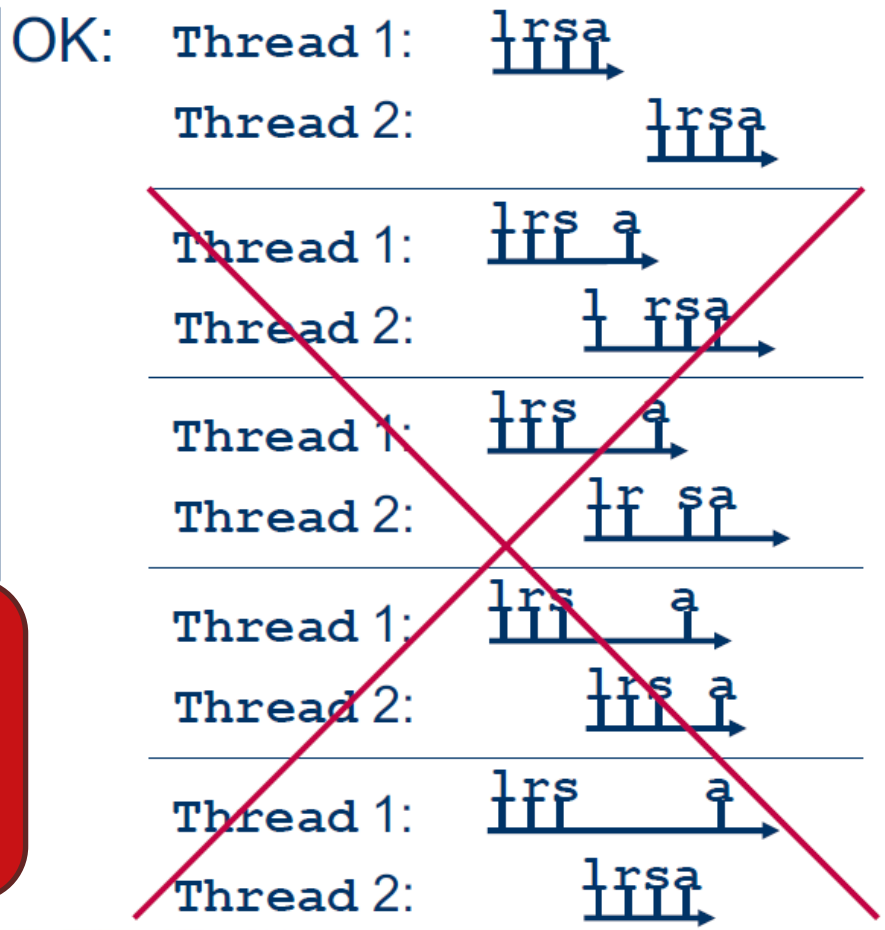
Schlüsselwort synchronized

- Mit getNext() als kritischem Abschnitt werden die fehlerhaften Verschränkungen verhindert (und einige unproblematische).

```
public class UnsafeZaehler {
    private int zaehler;

    public int getZaehler(){
        synchronized (this){
            return ++zaehler;
        }
    }
}
```

Da zu jedem Zeitpunkt nur ein Thread den kritischen Abschnitt betreten darf, treten keine problematischen Verschränkungen mehr auf.



Schlüsselwort synchronized

- Jedes Objekt in Java besitzt genau eine Marke (Token), die sich ein Thread mittels synchronized exklusiv für sich beanspruchen kann. Nur der eine Thread der diese Marke besitzt darf den kritischen Abschnitt „betreten“ und den Code ausführen. Die Threads, die die Marke nicht bekommen haben werden am synchronized-Block blockiert und warten bis sie die Marke bekommen. Am Ende des Codes wird dann diese Marke wieder freigegeben, sodass der nächste Thread den kritischen Abschnitt ausführen kann. Somit wird der kritische Abschnitt sequentiell ausgeführt.

```
synchronized(Referenz_auf_ein_Objekt){
            //kritischer Abschnitt
}
```

Übungen zu synchronized-Beispielen

- Was kann hier schief gehen?

```
public class R1 implements Runnable{
    private static ArrayList<Runnable> runnableList;

    @Override
    public void run() {
        if(runnableList == null){
            runnableList = new ArrayList<Runnable>();
        }
        runnableList.add(this);
    }
}
```

Übungen zu synchronized-Beispielen

- Was kann hier schief gehen?

```
public class R1 implements Runnable{
    private static ArrayList<Runnable> runnableList;

    @Override
    public void run() {
        if(runnableList == null){
            runnableList = new ArrayList<Runnable>();
        }
        runnableList.add(this);
    }
}
```

- Bei einer laufenden Instanz von R1: Nichts
- Bei mehreren, parallel laufenden Instanzen:
  - Ein Thread kann von anderem Thread angelegte Liste überschreiben.
  - Wettlaufsituation

Übungen zu synchronized-Beispielen

- Was kann hier schief gehen?

```
public class R2 implements Runnable{
    private static int counter = 0;

    @Override
    public void run() {
        counter++;
        if(counter == 3){
            System.out.println("3!");
        }
    }
}
```

Übungen zu synchronized-Beispielen

- Was kann hier schief gehen?

```
public class R2 implements Runnable{
    private static int counter = 0;

    @Override
    public void run() {
        counter++;
        if(counter == 3){
            System.out.println("3!");
        }
    }
}
```

- Bei einer laufenden Instanz von T2: Nichts
- Bei mehreren, parallel laufenden Instanzen: Wert in counter kann nach der Ausführung aller n Threads kleiner als n sein.
- Bei > 3 parallel laufenden Instanzen: 3! kann einmal, mehrmals oder gar nicht ausgegeben werden. ->Lesen-Ändern-Schreiben-Wettlaufsituation

Übungen zu synchronized-Beispielen

- Welche möglichen Ausgaben hat das folgende parallele Programm?

```
public class NoSync {
    public static void main(String[] args) {
        System.out.println("Start");
        for (int i = 0; i <5; i++) {
            Thread t = new Thread(new R3());
            t.start();
        }
        System.out.println("End");
    }
}
class R3 implements Runnable{
private static int a = 0;
    @Override
    public void run() {
        a++;
        System.out.println("Ich bin Thread "+a);

    }
}
```

Übungen zu synchronized-Beispielen

- Welche möglichen Ausgaben hat das folgende parallele Programm?

```
public class NoSync {
    public static void main(String[] args) {
        System.out.println("Start");
        for (int i = 0; i <5; i++) {
            Thread t = new Thread(new R3());
            t.start();
        }
        System.out.println("End");
    }
}
class R3 implements Runnable{
private static int a = 0;
    @Override
    public void run() {
        a++;
        System.out.println("Ich bin Thread "+a);

    }
}
```

- Zeile 1: Start.
- Zeile 2: * Ich bin Thread [1-5]
- Zeile 3: * Ich bin Thread [1-5]
- Zeile 4: * Ich bin Thread [1-5]
- Zeile 5: * Ich bin Thread [1-5]
- Zeile 6: * Ich bin Thread [1-5]
- Zeile 7: *

- *: Mögliche Stelle für „End.“
- Doppelte und fehlende Zahlen möglich!

Übungen zu synchronized-Beispielen

- Welche möglichen Ausgaben hat das folgende parallele Programm?
- Hinweis: Nicht nur Objekte einer Klasse sondern auch die Klasse selbst besitzt eine Marke (Token). Mit Klassenname.class kann auf diese zugegriffen werden.

```
public class Sync {
    static int a = 0;

    public static int getNextNumber() {
        synchronized (Sync.class) {
            a++;
            return a;
        }
    }

    public static void main(String[] args) {
        System.out.println("Start");
        for (int i = 0; i <5; i++) {
            Thread t = new Thread(new R4());
            t.start();
        }
        System.out.println("End");
    }
}
class R4 implements Runnable{
    @Override
    public void run() {
        System.out.println("Ich bin Thread "+ Sync.getNextNumber());

    }
}
```

Übungen zu synchronized-Beispielen

- Welche möglichen Ausgaben hat das folgende parallele Programm?
- Zeile 1: Start.
- Zeile 2: * Ich bin Thread [1-5]
- Zeile 3: * Ich bin Thread [1-5]
- Zeile 4: * Ich bin Thread [1-5]
- Zeile 5: * Ich bin Thread [1-5]
- Zeile 6: * Ich bin Thread [1-5]
- Zeile 7: *

- Jede Zahl genau ein Mal!
- *: Mögliche Stelle für „End.“

```
public class Sync {
    static int a = 0;

    public static int getNextNumber() {
        synchronized (Sync.class) {
            a++;
            return a;
        }
    }

    public static void main(String[] args) {
        System.out.println("Start");
        for (int i = 0; i <5; i++) {
            Thread t = new Thread(new R4());
            t.start();
        }
        System.out.println("End");
    }
}
class R4 implements Runnable{
    @Override
    public void run() {
        System.out.println("Ich bin Thread "+ Sync.getNextNumber());

    }
}
```

Übungen zu synchronized-Beispielen

- Was ist hier falsch? Was ist die Ausgabe?

```
public class SyncBlock {
    public static void main(String[] args) {
        System.out.println("Start");
        for (int i = 0; i <5; i++) {
            Thread t = new Thread(new R5());
            t.start();
        }
        System.out.println("End");
    }
}
class R5 implements Runnable{
    private static int a = 0;
    @Override
    public void run() {
        synchronized (this) {
            a++;
            System.out.println("Ich bin Thread " + a);
        }
    }
}
```

Übungen zu synchronized-Beispielen

- Was ist hier falsch? Was ist die Ausgabe?

- this synchronisiert am eigenen Runnable-Objekt. Das heißt, jedes Runnable synchronisiert nur sich selbst -> nutzlos

```
public class SyncBlock {
    public static void main(String[] args) {
        System.out.println("Start");
        for (int i = 0; i <5; i++) {
            Thread t = new Thread(new R5());
            t.start();
        }
        System.out.println("End");
    }
}
class R5 implements Runnable{
    private static int a = 0;
    @Override
    public void run() {
        synchronized (this) {
            a++;
            System.out.println("Ich bin Thread " + a);
        }
    }
}
```

Übungen zu synchronized-Beispielen

- Besser:

```
public class SyncBlock2 {
    public static void main(String[] args) {
        System.out.println("Start");
        for (int i = 0; i <5; i++) {
            Thread t = new Thread(new R6());
            t.start();
        }
        System.out.println("End");
    }
}
class R6 implements Runnable{
    private static int a = 0;

    @Override
    public void run() {
        synchronized (SyncBlock2.class) {
            a++;
            System.out.println("Ich bin Thread " + a);
        }
    }
}
```

```
public class SyncBlock {
    public static void main(String[] args) {
        System.out.println("Start");
        //Da kein gemeinsames Objekt bei den Runnables vorhanden ist,
        //wird einfach ein Objekt nur zum Sychronisieren erzeugt
        //Man kann jedes beliebige Objekt nehmen.
        Object o = new Object();
        for (int i = 0; i <5; i++) {
            Thread t = new Thread(new R5(o)); //Objekt wird übergeben
            t.start();
        }
        System.out.println("End");
    }
}
class R5 implements Runnable{
    private static int a = 0;
    Object o;
    public R5 (Object o){
        this.o = o; //Jedes Runnable speichert DASSELBE Objekt ab
    }
    @Override
    public void run() {
        synchronized (o) { //Alle Runnable synchronisieren sich am selben Objekt
            a++;
            System.out.println("Ich bin Thread " + a);
        }
    }
}
```

„Monitor“ = totale Serialisierung aller Objektzugriffe

```
public class Monitor {

   // no public fields

   private final Object myLock
      = new Object();

   ... someMethod(...) {
      synchronized(myLock) {
         // do something
      }
   }
}
```

- Muster für einen Extremfall der Synchronisierung
  - Kein öffentlich sichtbaren Attribute.
  - Weil alle Methoden an **myLock** synchronisiert sind, ist immer nur höchstens ein **Thread** im **Monitor**-Objekt aktiv.
  - Wegen privatem Objekt **myLock** kann niemand von außen mit der Synchronisierung interferieren.

- Allerdings können trotzdem Wettlaufsituationen in der Klasse auftreten, die den Monitor benutzen! Bsp.: Counter-Klasse als Monitor -> zwei Threads if(monitor.getValue() == 5) -> beide könnten hier das if() mit true auswerten