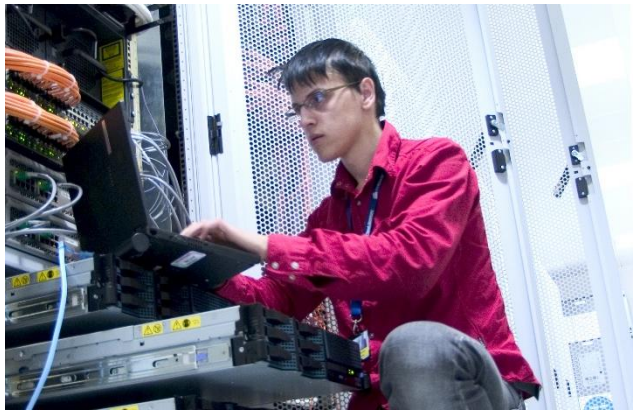
EMIL-VON-BEHRING GYMNASIUM

Wettlaufsituation (Race condition)



Wettlaufsituationen – warum tritt sie auf?

```
if(IstRoboter()) {
    LinksDrehen();
} else {
    Schritt();
}
```

```
if (!canvas.hasColor(x, y)){
    canvas.colorize(x, y, color);
}
```

```
if(!IstWand() && !IstRoboter()) {
    Schritt();
}
```

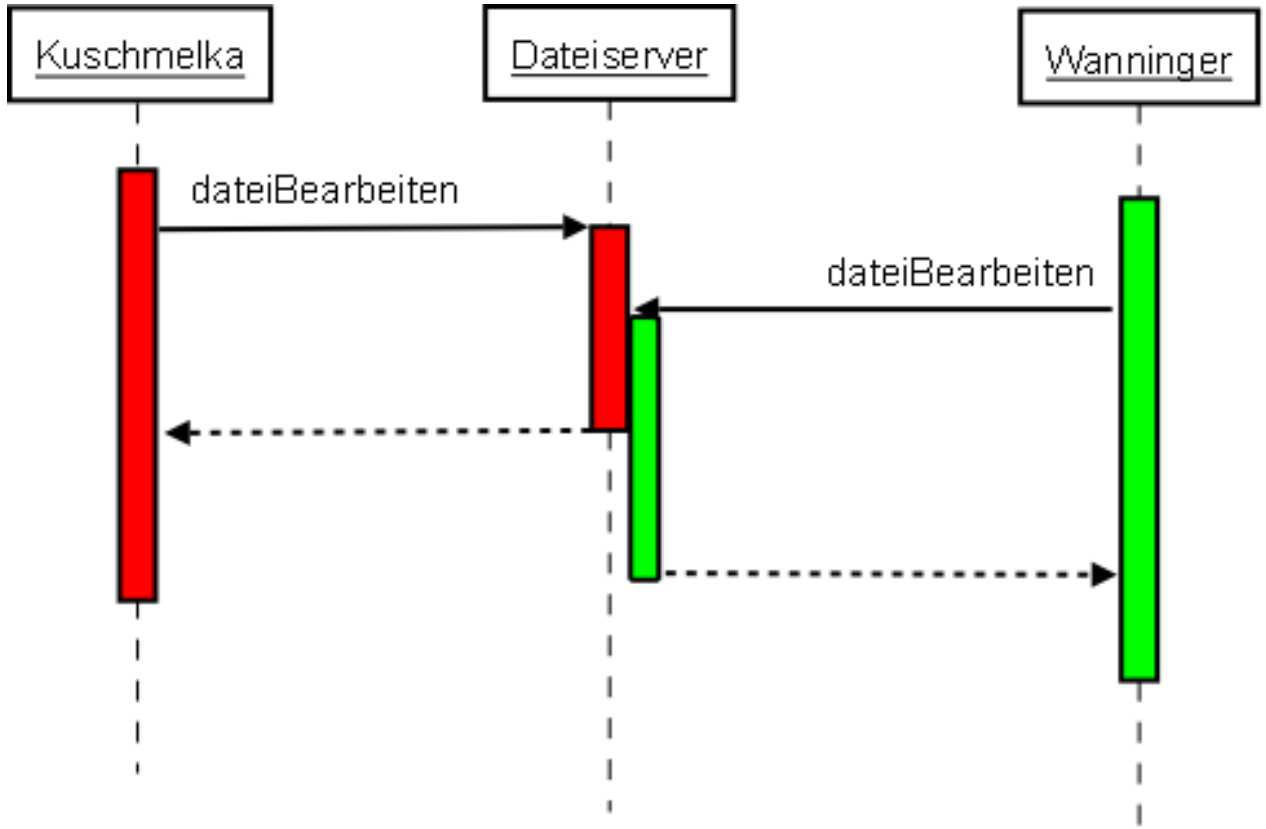
- Gemeinsamkeiten?
- Verzahnung zwischen Prüfen und Handeln
  → Prüfe-und-Handle-Wettlaufsituation
- Auf einem gemeinsamen Objekt → Leinwand bzw. Tanzfläche

Fehlerquellen aufgrund von Parallelität

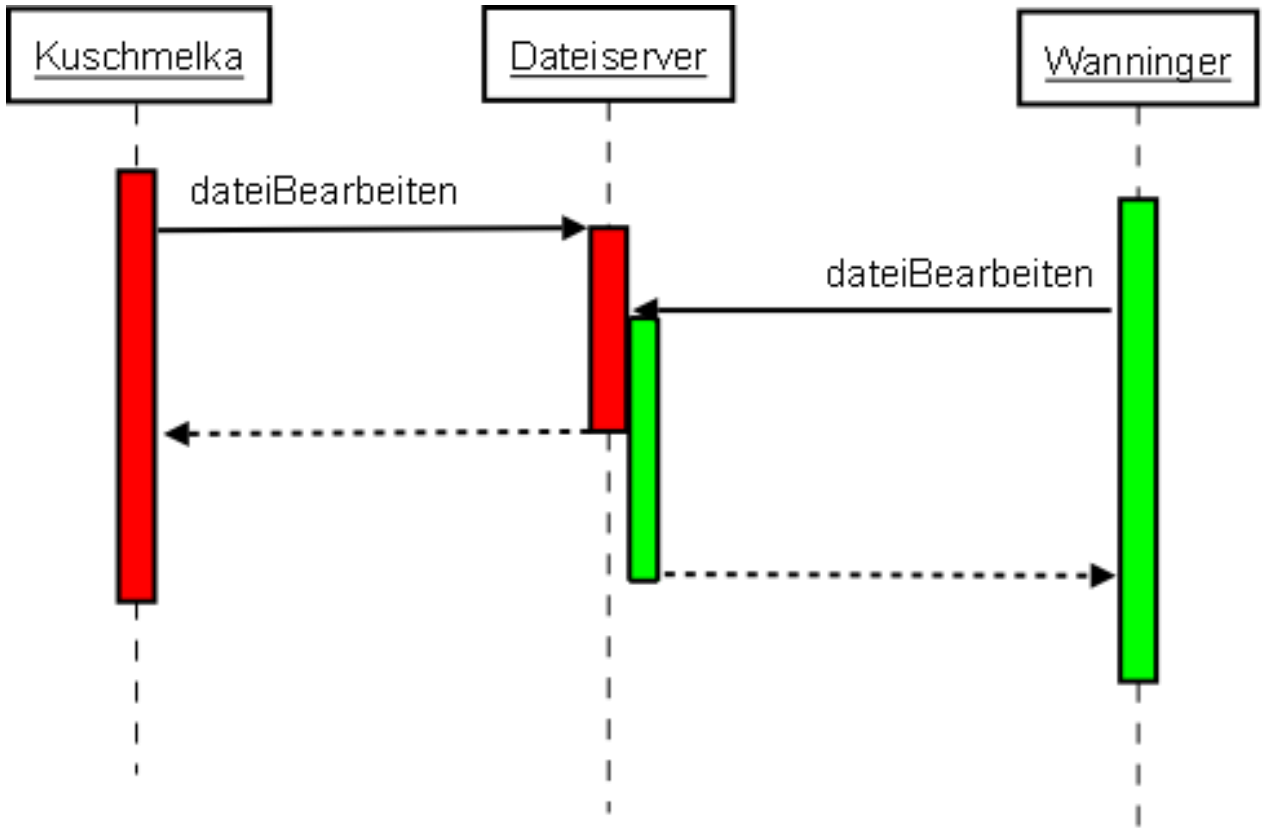
- Syntaxfehler → Programmcode nicht korrekt und kann nicht kompiliert werden.
- Laufzeitfehler führen zu Exceptions (welche ohne catch zum Programmabsturz führen), weil nicht mögliche Operationen versucht werden.
  - Methodenaufruf auf einem nicht vorhandenen Objekt
  - Zugriff auf einen Index im Array welcher nicht existiert usw.
  - …
- Bei parallelen Programmen kommen weitere Fehlerquellen hinzu, die aufgrund ungünstiger Verzahnung von Codeabschnitten verschiedener Threads entstehen können. → liefert manchmal falsche Ergebnisse, führt aber nicht zwangläufig zum Absturz

Beispiel: Dokumentbearbeitung

- Zwei Personen sollen gemeinsam an einem Dokument (= Ressource) arbeiten, das auf einem Server liegt.



Beispiel: Dokumentbearbeitung



- Wanninger überschreibt hier (ungewollt) die Änderungen von Kuschmelka.

Wettlaufsituation

- Eine Wettlaufsituation kann nicht bei einem gemeinsamen Objekt auftreten, sondern auch bei einer gemeinsamen (primitiven) Variable!
- Wir kennen bereits verschieden Typen von Variablen. Welche davon teilen sich die Objekte der Klasse MYTHREAD?
- → Keine
- Wie kann man dann eine solche Variable definieren?
- Öffne das BlueJ-Projekt Schaf Vorlage!

```
class MYTHREAD extends Thread {
    int attribut;
    public MYTHREAD(int parameter){
        int lokaleVariable;
        //...
    }
}
```

Schlüsselwort static

Beim Klassendiagramm werden statische Attribute und Methoden unterstrichen!

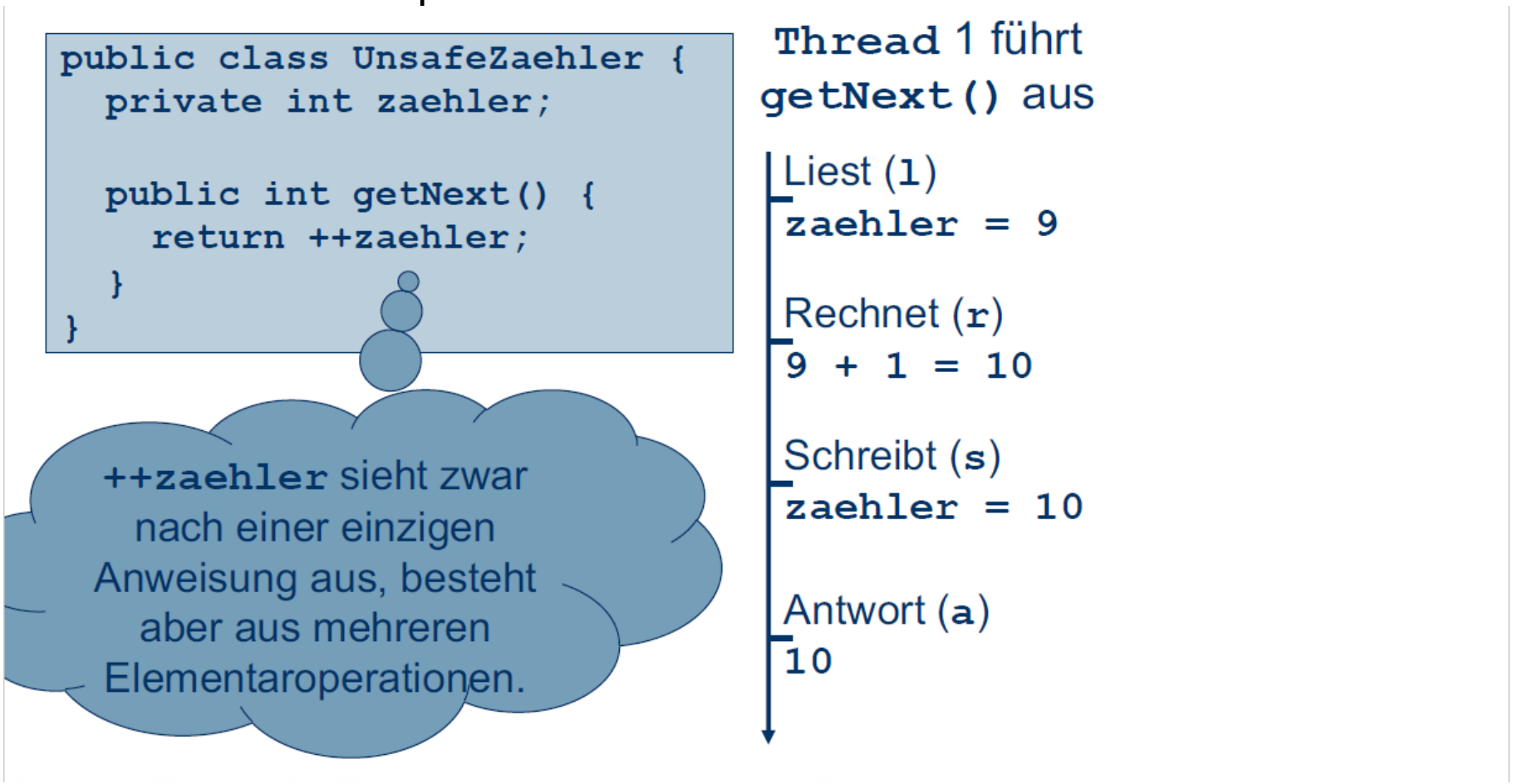
- Das Schlüsselwort **static** in Java bindet eine Variable oder eine Methode an die Klasse anstatt - wie üblich - ans Objekt.
- Das bedeutet, dass Methode ohne ein existierendes Objekt aufgerufen werden können. Bei der Punktnotation wird hier anstatt des Objektnamens der Klassenname hingeschrieben: Klassenname.statischeMethode()
- Statische Attribute existieren nur genau ein Mal als Klassenvariable; auch ohne Objekte dieser Klasse. Falls Objekte dieser Klasse existieren, teilen sich **alle Objekte** dieser Klasse die statische Variable. Objekte anderer Klassen müssen über den Klassennamen darauf zugreifen: Klassenname.Variable

Aufgabe: BrokenCounter

- Schreibe in einem neuen Projekt **BrokenCounter** die Klasse BrokenCounter. Diese hat ein öffentliche und statische Variable counter, welche parallel von mehreren Threads gleichzeitig hochgezählt werden soll.
- So sollen zum Beispiel 10 Threads gestartet werden, die den counter jeweils 10-Mal um 1 erhöhen. Nachdem alle Threads die Variable hochgezählt haben, soll der Main-Thread den counter ausgeben.
- Führe dein Programm mehrfach aus und erhöhe dabei die Anzahl wie oft die Threads den counter erhöhen (ca. bis 100000 pro Thread). Was fällt dir auf?

Wettlaufsituation (Race condition)

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



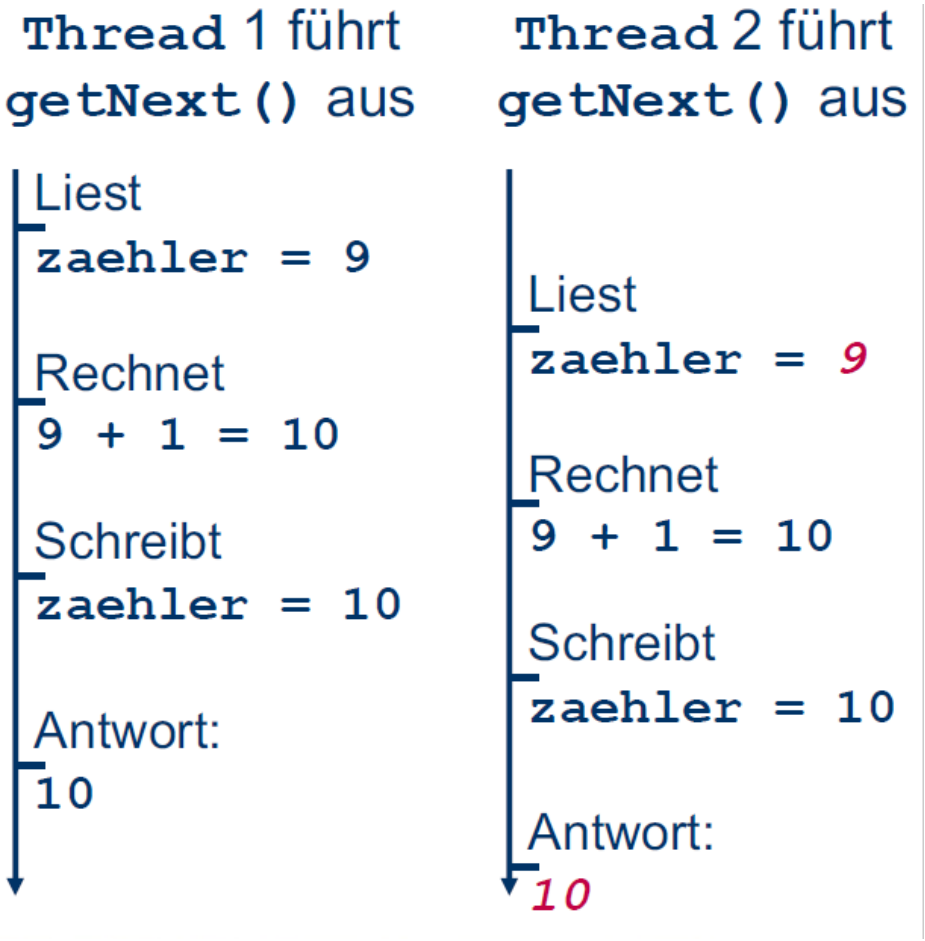
Wettlaufsituation (Race condition)

- 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.)

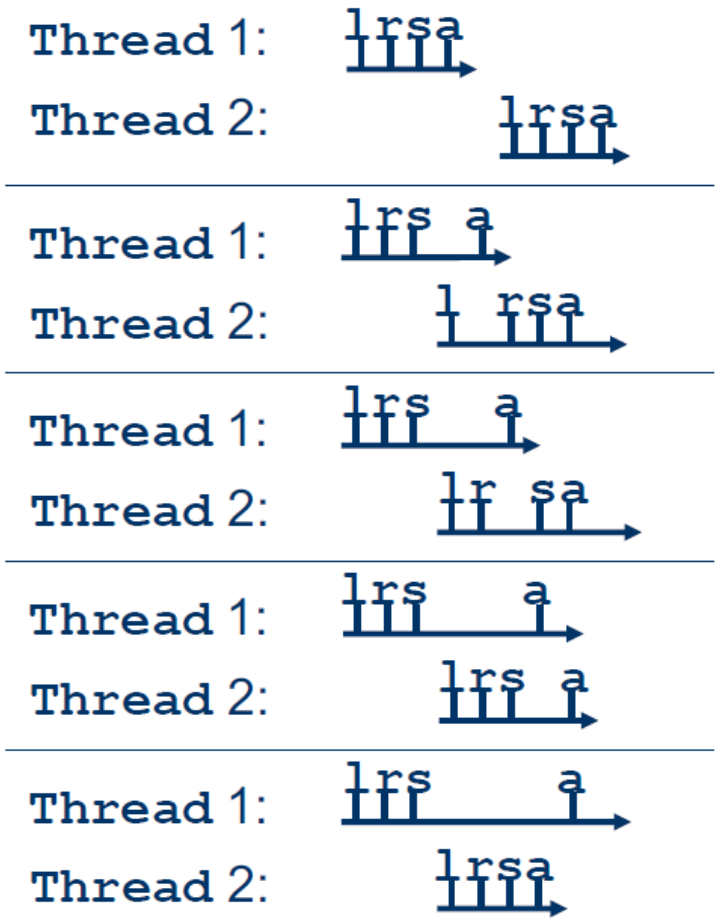
Wettlaufsituation (Race condition)

- 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.

Definition – Wettlaufsituation (Race condition)

- Eine Wettlaufsituation kann immer dann auftreten,
  - wenn eine Variable (primitiv oder ein Objekt) von mehr als einer Aktivität (Thread, Person, Prozess, …)
    - gelesen,
    - abhängig vom Wert geändert und
    - dann das Ergebnis zurück in die Variable geschrieben wird,
  - Das bedeutet, wenn das Lesen, Ändern und Schreiben **nicht** als ununterbrechbare Einheit (atomare Aktion) geschieht.

Beispiele zur Wettlaufsituation

- Das zaehler-Beispiel auf den vorherigen Folien zeigt die klassische „Lesen-Ändern-Schreiben-Wettlaufsituation“.
  - Siehe Inf 13.2 Maschinennahe Programmierung

zaehler++ →	LOAD zaehler	→ lesen
	ADDI 1	→ anhängig vom Wert verändern
	STORE zaehler	→ schreiben

- Das Picasso-Beispiel zeigt eine „Prüfe-Handle-Wettlaufsituation“. Auch hier wird die Variable Canvas von mehreren Threads gleichzeitig
  - gelesen, → hasColor(…)
  - abhängig vom Wert verändert, → if(hasColor(…)) {…}
  - und zurückgeschrieben. → colorize(…)

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“

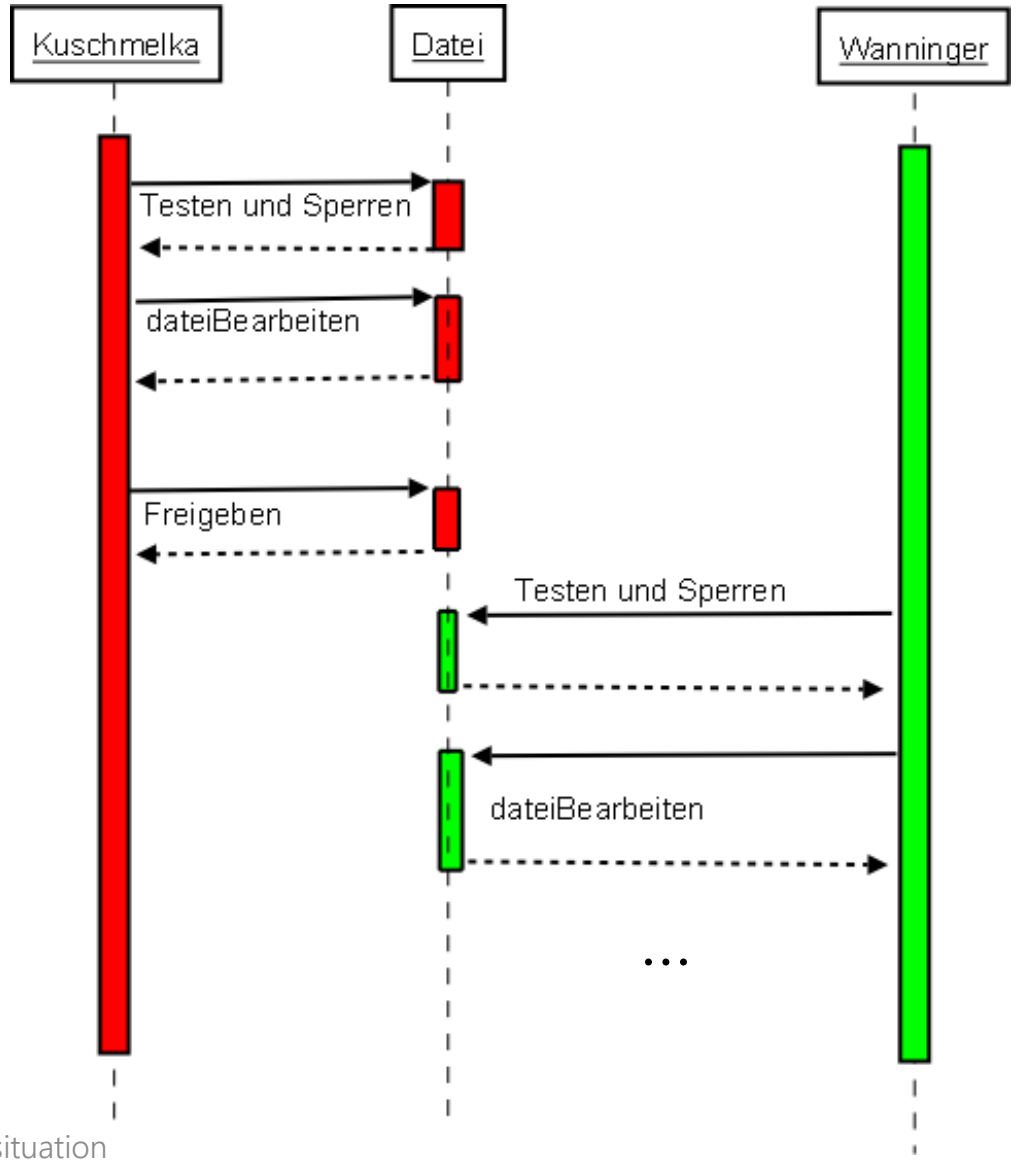
Wettlaufsituationen sind extrem problematisch

- 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 irgendeine 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).

Lösung: Dokumentbearbeitung

- Wanninger kann die Datei nur dann ändern, wenn die Bearbeitung durch Kuschmelka abgeschlossen ist.

- Wir fassen das Lesen, Verändern und Schreiben in eine ununterbrechbare Einheit (atomare Aktion) zusammen.



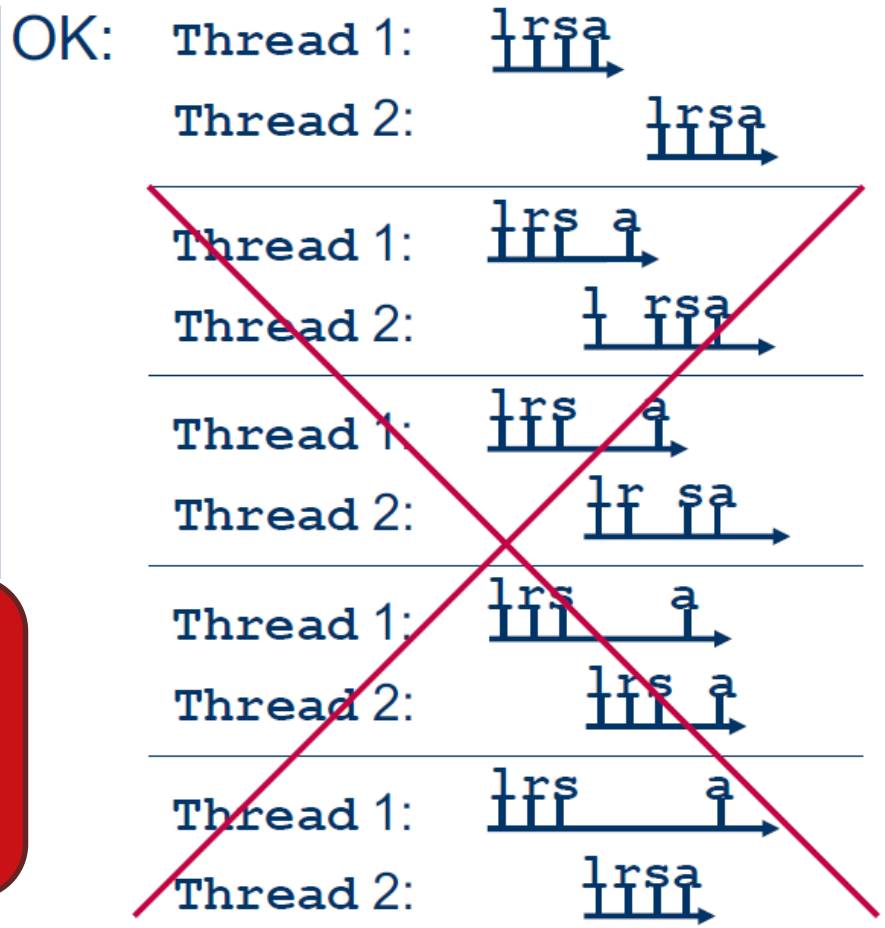
Schlüsselwort synchronized

- 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 sequenziell ausgeführt.

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

Übungen zu synchronized-Beispielen

- Was kann hier schief gehen?

```
public class T1 extends Thread{

    private static ArrayList<Thread> threadList;

    public void run() {
        if(threadList == null){
            threadList = new ArrayList<Thread>();
        }
        threadList.add(this);
    }
}
```

Übungen zu synchronized-Beispielen

- Was kann hier schief gehen?

```
public class T1 extends Thread{

    private static ArrayList<Thread> threadList;

    public void run() {
        if(threadList == null){
            threadList = new ArrayList<Thread>();
        }
        threadList.add(this);
    }
}
```

- Bei einer laufenden Instanz von T1: 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 T2 {

    private static int counter = 0;

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

Übungen zu synchronized-Beispielen

- Was kann hier schief gehen?

```
public class T2 {

    private static int counter = 0;

    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 NoSync(){
        System.out.println("Start");
        for (int i = 0; i <5; i++) {
            T3 t = new T3();
            t.start();
        }
        System.out.println("End");
    }
}
class T3 extends Thread{
private static int a = 0;

    public void run() {
        a++;
        System.out.println("Ich bin Thread " + a);

    }
}
```

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: *

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

```
public class NoSync {
    public NoSync(){
        System.out.println("Start");
        for (int i = 0; i <5; i++) {
            T3 t = new T3();
            t.start();
        }
        System.out.println("End");
    }
}
class T3 extends Thread{
private static int a = 0;

    public void run() {
        a++;
        System.out.println("Ich bin Thread " + a);

    }
}
```

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 Sync() {
        System.out.println("Start");
        for (int i = 0; i <5; i++) {
            T4 t = new T4();
            t.start();
        }
        System.out.println("End");
    }
}
class T4 extends Thread{

    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 Sync() {
        System.out.println("Start");
        for (int i = 0; i <5; i++) {
            T4 t = new T4();
            t.start();
        }
        System.out.println("End");
    }
}
class T4 extends Thread{

    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 SynchronizedBlock {
    public SynchronizedBlock() {
        System.out.println("Start");
        for (int i = 0; i <5; i++) {
            T5 t = new T5();
            t.start();
        }
        System.out.println("End");
    }
}
class T5 extends Thread{
    private static int a = 0;

    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 Thread-Objekt. Das heißt, jeder Thread synchronisiert nur sich selbst -> nutzlos

```
public class SynchronizedBlock {
    public SynchronizedBlock() {
        System.out.println("Start");
        for (int i = 0; i <5; i++) {
            T5 t = new T5();
            t.start();
        }
        System.out.println("End");
    }
}
class T5 extends Thread{
    private static int a = 0;

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

Übungen zu synchronized-Beispielen

- Besser:

```
public class SyncBlock2 {
    public SyncBlock2() {
        System.out.println("Start");
        for (int i = 0; i <5; i++) {
            T6 t = new T6();
            t.start();
        }
        System.out.println("End");
    }
}
class T6 extends Thread{
    private static int a = 0;

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

```
public class SyncBlock {
    public SyncBlock() {
        System.out.println("Start");
        //Da kein gemeinsames Objekt bei den Threads 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++) {
            T5 t = new T5(o); //Objekt wird übergeben
            t.start();
        }
        System.out.println("End");
    }
}
class T5 extends Thread{
    private static int a = 0;
    Object o;
    public T5 (Object o){ this.o = o; } //Jeder Thread speichert DASSELBE Objekt ab

    public void run() {
        synchronized (o) { //Alle Threads 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
  - Keine 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

Allgemeine Regel

- Den synchronized-Block möchte man möglichst „klein“ halten, da sonst der „Speed Up“ durch die Parallelisierung verloren geht.

- Falls nur eine einzige Variable synchronisiert werden muss, bietet Java die Klassen aus java.util.concurrent.atomic an. Die angebotenen Methoden der atomic-Klassen (get(), set(…), getAndIncrement(), getAndAdd(…), compareAndSet(…)) sind werden atomar ausgeführt und sind somit thread-sicher!
- Vorteil: Man kommt ohne aufwändiges Blockieren von Aktivitätsfäden aus, da die Methoden auf ununterbrechbare Hardware-Instruktionen abgebildet werden können.

Aufgabe

- Verhindere durch ein synchronized-Block die Wettlautsituationen bei den Projekten
  - Tanzparty
  - Picasso
  - BrokenCounter

- Achte darauf, dass
  - eine geeignete, gemeinsame Marke/Token verwendet wird
  - und der synchronized-Block möglichst klein ist

- Optional:
  - Löse die Wettlaufsituation bei BrokenCounter mithilfe einer Atomic-Klasse