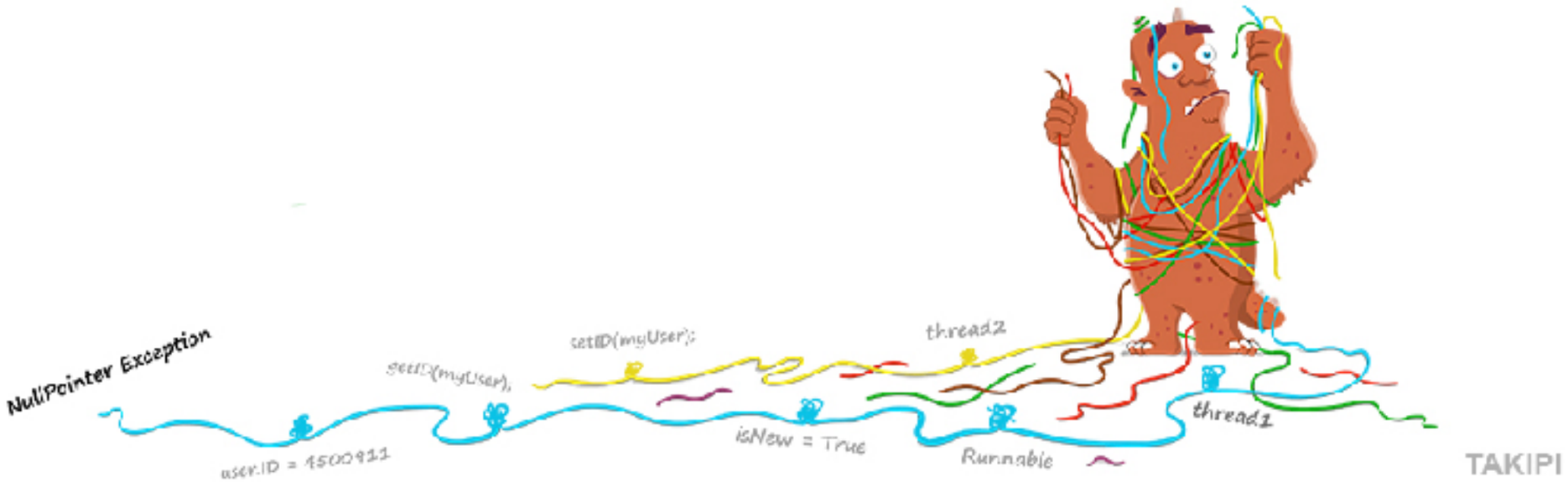


TAKIPI

THREADS UND NEBENLÄUFIGE PROGRAMMIERUNG

SEW 4

DI Thomas Helml

HTL WELS

IT INFORMATIONSTECHNOLOGIE

THREADS – INHALT

- 5.1 Nebenläufigkeit
- 5.2 Threads erzeugen
- 5.3 Thread-Eigenschaften und –Zustände
- 5.4 Der Ausführer (Executor) kommt
- 5.5 Synchronisation über kritische Abschnitte
- 5.6 Synchronisation über Warten und Benachrichtigen
- 5.7 Datensynchronisation durch besondere Concurrency-Klassen
- 5.8 Atomare Operationen und frische Werte mit volatile
- 5.9 Threads in einer Thread-Gruppe
- 5.10 Einen Abbruch der virtuellen Maschine erkennen

THREADS – INHALT

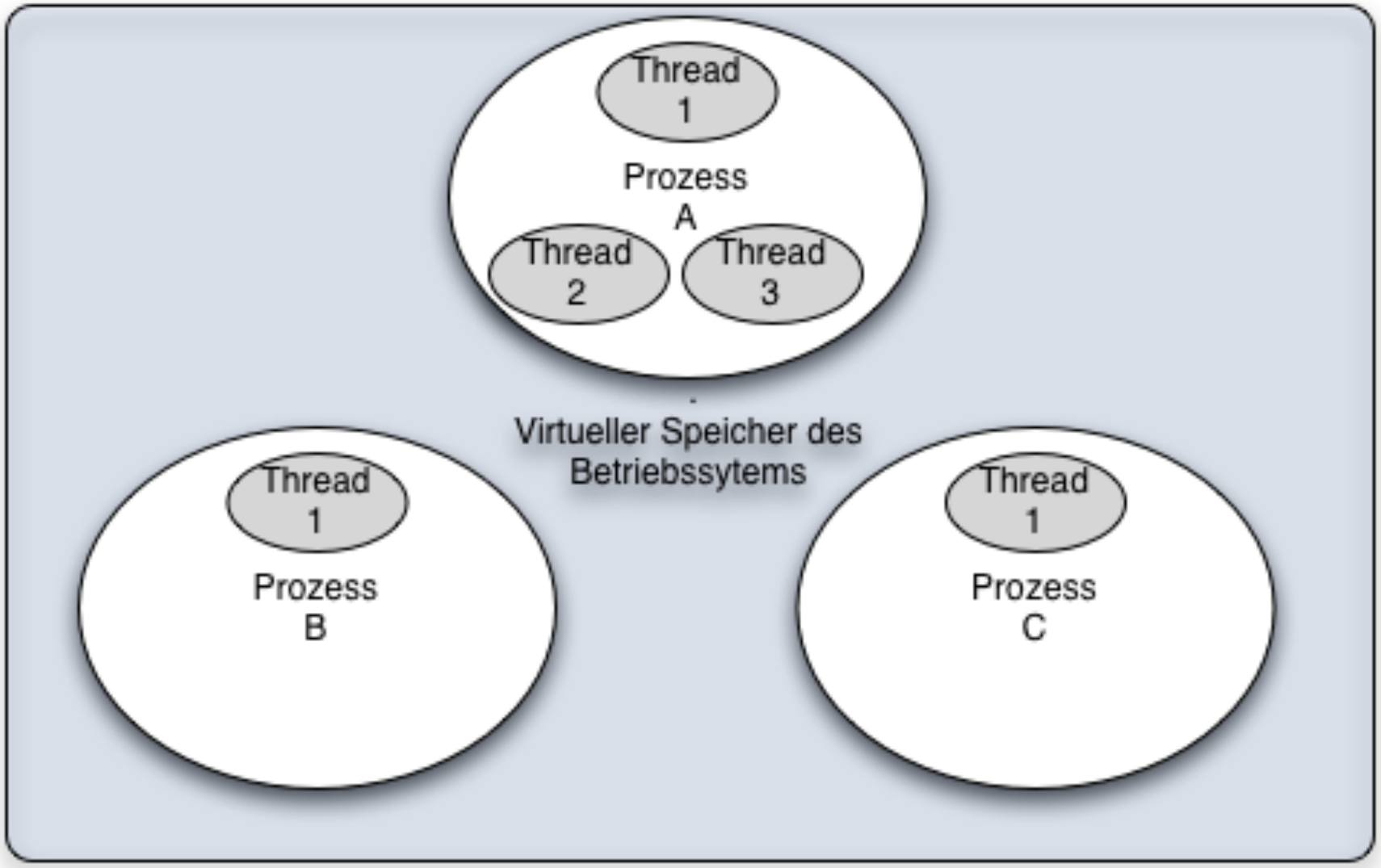
5.1 NEBENLÄUFIGKEIT

- Nebenläufigkeit/Concurrency:
  - mehrere Ereignisse stehen in keiner kausaler Beziehung zueinander (beeinflussen sich nicht)
- Aktionen können parallel ausgeführt werden, wenn keine das Resultat der anderen benötigt
- auf Ein-Prozessorsystemen wird uns Nebenläufigkeit vorgegaukelt
  - Scheduler schaltet zwischen Prozessen um)

PROZESS/THREAD

- Prozess:
  - Verwaltungseinheit für OS
  - kooperieren nicht, müssen voneinander geschützt werden
  - eigener Adressraum
    - jeder Prozess hat mind. einen Thread
- **Threads**:
  - laufen innerhalb eines Prozesses
  - kooperieren
  - teilen sich z.B. Heap, d.h. können über gemeinsamen Speicher kommunizieren

KLASSE THREAD



JAVA UND NEBENLÄUFIGKEIT

- `Thread`
  - jeder laufende Thread ist ein Exemplar dieser Klasse
- `Runnable`
  - beschreibt den Programmcode, den die JVM parallel ausführen soll
- `Lock`
  - dient zum Markieren von kritischen Abschnitten, in denen sich nur ein Thread befinden darf
- `Condition`
  - Threads können auf die Benachrichtigung anderer Threads warten.

THREADS – INHALT

5.2. THREADS ERZEUGEN

- Thread muss wissen, was er ausführen soll
  - -> Anweisungsfolge: wird in Objekt vom Typ `Runnable` verpackt
  - `Thread` bekommt dieses Objekt
    - beim Start arbeitet er Befehlsobjekt parallel zum restlichen Code ab
- Schnittstelle `Runnable` schreibt nur eine `run()`-Methode vor.

RUNNABLE

- jeder Thread in Java muss das Interface `Runnable` implementieren (direkt oder indirekt):
- In der Methode `run()` steht der Code, welcher im Thread ausgeführt werden soll

THREADS MIT RUNNABLE STARTEN

- ruft man `run()` direkt auf => **keine** Nebenläufigkeit
- Threads step-by-step:

1. Objekt der Klasse `Thread` erzeugen: Konstruktor hat Runnable als Parameter
2. Methode `start()` aufrufen. Jeder Thread nur 1x starten!

   sollte Thread schon laufen => `IllegalThreadStateException`

THREADS BEISPIEL

- Aufgabe GitHub:
  - *4101_FirstThread*

LOS!

KLASSE THREAD ERWEITERN

- Klasse Thread implementiert selbst Runnable
  - daher kann auch Thread erweitert werden:

```
public class DateThread extends Thread {
     @Override public void run() {
          for ( int i = 0; i < 20; i++ )
               System.out.println( new Date() );
          }
}
```

Aufruf:

```
Thread t = new DateThread();
t.start();
```

ODER:

```
new DateThread().start();
```

SELBSTSTARTER

- Ein Thread kann sich auch selbst starten:
  - `start()` Methode im Thread-Konstruktor aufrufen

```
class DateThread extends Thread {
 DateThread() {
    start();
 }
 // ... der Rest bleibt ...
}
```

THREADS – INHALT

5.3 THREAD-EIGENSCHAFTEN UND -ZUSTÄNDE

- Eigenschaften von Threads:
  - Zustand
  - Priorität
  - Namen
  - ...
- Zugriff über statische Methode `currentThread()`:

```
System.out.println(Thread.currentThread().getPriority() );
```

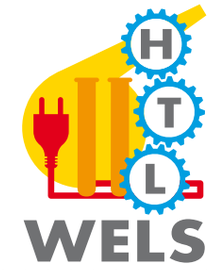
SCHLÄFER GESUCHT

- Aktueller (!!) Thread kann angehalten werden
- Fremde Threads nicht!
  - Statische Methode `Thread.sleep()` bzw.
  - sleep() auf TimeUnit-Objekt

```
try {
 Thread.sleep( 2000 );
} catch ( InterruptedException e ) { }


Dann mit TimeUnit:


try {
TimeUnit.SECONDS.sleep( 2 );
} catch ( InterruptedException e ) { }
```

NANOSECONDS
MICROSECONDS
MILLISECONDS
SECONDS
MINUTES
HOURS
DAYS

MIT `YIELD()` AUF RECHENZEIT VERZICHTEN



- `Thread.yield()`
  - ordnet den Thread bezüglich seiner Priorität wieder in die Thread-Warteschlange des Systems ein

DER THREAD ALS DÄMON

- Server läuft i.d.R. ewig (Endlosschleife)
- In manchen Fällen soll Serverfunktionalität mit Anwendung beendet werden
  - Thread dieser Art nennt man Dämon(en)
  - Abbruch erfolgt (abrupt) sobald JVM erkennt, dass nur mehr Dämonen laufen
  - Vorsicht bei I/O Transaktionen!

WIE EIN THREAD IN JAVA ZUM DÄMON WIRD

```
class DaemonThread extends Thread {
   DaemonThread() {
      setDaemon( true );
   }
   @Override public void run() {
      while ( true )
         System.out.println( "Lauf!" );
   }
   public static void main( String[] args )   {
      new DaemonThread().start();
   }
}
```

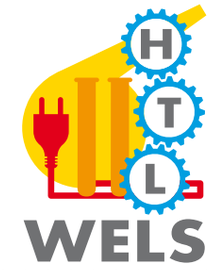
DAS ENDE EINES THREADS

- Thread ist beendet, wenn eine der folgenden Bedingungen zutrifft:
  - run()-Methode wurde ohne Fehler beendet
  - in `run()`-Methode tritt `RuntimeException` auf,
  - Thread wurde von außen abgebrochen
    - Methode `stop()`: von Verwendung wird abgeraten (`deprecated`)
- virtuelle Maschine wird beendet und nimmt alle Threads mit ins Grab

EINEN THREAD HÖFLICH MIT INTERRUPT BEENDEN

- Thread wir gebeten, seine Arbeit aufzugeben
  - interrupt() setzt von außen in einem Thread-Objekt ein internes Flag
- Periodische Überprüfung, ob jemand von außen den Abbruchswunsch geäußert hat
  - in der `run()`-Methode wird mittels `isInterrupted()` periodisch abgefragt, ob Thread enden soll

AUFGABE



- Github:
  - `4102_Interrupt`

EINEN THREAD HÖFLICH MIT INTERRUPT BEENDEN

- `run()`: Schleife wird verlassen, wenn `isInterrupted()` `true` ergibt
  - Passiert durch Aufruf von `interrupt()` in `main()`-Methode
- Was macht `interrupt()` im `catch`-Block?
  - Programm würde sonst nicht funktionieren
  - Ausgabe jede halbe Sekunde, d.h. Thread schläft fast gesamte Zeit
  - somit stört `interrupt()` den Thread beim Schlafen

EINEN THREAD HÖFLICH MIT INTERRUPT BEENDEN

- `interrupt()` löst => `InterruptedException` aus, `catch`-Behandler fängt die Ausnahme ein
- durch Unterbrechung wird das interne Flag zurückgesetzt (!!)
- `isInterrupted()` liefert wieder `false`
- daher `interrupt()` erneut aufrufen!
- Beachte bei der Nutzung von `isInterrupted()`:
  - NEBEN `sleep()` löschen auch `join()` und `wait()` das Flag durch die `InterruptedException`

THREADS – INHALT

5.4 DER AUSFÜHRER (EXECUTOR) KOMMT

- Starke Bindung von Thread und Runnable:
  - beim Erzeugen eines Threads muss `Runnable` übergeben werden (nicht später möglich)
  - 2. Aufruf von `.start()` führt zu Exception, soll Runnable 2x aufgerufen werden -> 2 Thread Objekte
  - Thread beginnt sofort nach Aufruf von .start() – kann nicht (zeit-)gesteuert werden

DIE SCHNITTSTELLE EXECUTOR

- Seit Java 5 – Schnittstelle `Executor`
  - `void execute(Runnable command)`
- 2 konkrete Implementierungen
  - `ThreadPoolExecutor`:
    - Klasse baut Sammlung von Threads auf
      - -> `Thread Pool`
    - Ausführung wird von freien Threads übernommen
  - `ScheduledThreadPoolExecutor`:
    - Ausführung von Threads zu bestimmten Zeiten oder Wiederholungen

DIE SCHNITTSTELLE EXECUTOR

Utility Klasse für die Erzeugung von Executor:

`class java.util.concurrent.Executors`

Diverse Hilfsmethoden:

```
static ExecutorService newCachedThreadPool()
```
Liefert einen Thread-Pool mit wachsender Größe

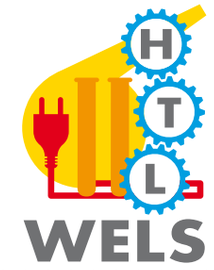
```
static ExecutorService newFixedThreadPool(int nThreads)
```
Liefert einen Thread-Pool mit maximal n Threads

```
static ScheduledExecutorService    newSingleThreadScheduledExecutor()

static ScheduledExecutorService
    newScheduledThreadPool(int corePoolSize)
```
gibt spezielle Executor-Objekte zurück, um Wiederholungen festzulegen

KONKRETE EXECUTOREN

- `ExecutorService`: Schnittstelle, die `Executor` erweitert
  - u.a. sind Operationen zu finden, die die Ausführer herunterfahren
  - bei Thread-Pools nützlich, da die Threads ja sonst nicht beendet würden (weil sie auf neue Aufgaben warten)

AUSGABE



- Aufgabe GitHub
  - `4103_ThreadPools`

4103_THREADPOOLS AUSGABE

```
A1 Thread[pool-1-thread-1,5,main]
A2 Thread[pool-1-thread-1,5,main]
B1 Thread[pool-1-thread-2,5,main]
B2 Thread[pool-1-thread-2,5,main]
```

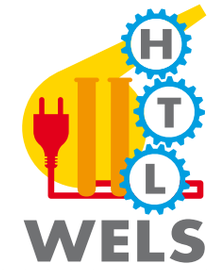
```
B1 Thread[pool-1-thread-1,5,main]
B2 Thread[pool-1-thread-1,5,main]
A1 Thread[pool-1-thread-2,5,main]
A2 Thread[pool-1-thread-2,5,main]
```

DIE THREAD-POOLS

- Methoden zum Steuern des Pool-Endes:
- `void shutdown()`
  - fährt Thread-Pool herunter
  - laufende Threads werden nicht abgebrochen
  - neue Anfragen werden nicht angenommen
- `boolean isShutdown()`
  - wurde der Executor schon heruntergefahren?
- `List<Runnable> shutdownNow()`
  - gerade ausführende Befehle werden zum Stoppen angeregt
  - Rückgabe ist eine Liste der zu beendenden Kommandos

THREADS – INHALT

AUSGABE



- Aufgabe GitHub
  - `4104_Bank`

THREADS BEISPIEL 2

Beispiel Ausgabe:

Vorher:

```
Konto 0: 30
Konto 1: 50
Konto 2: 100
```

Nachher:

```
Konto 0: 10
Konto 1: 70
Konto 2: 100
 ...
```

THREADS BEISPIEL 2

- Prinzipiell kann das Beispiel ohne Probleme funktionieren.
- Falls jedoch von beiden Threads zur gleichen Zeit der Kontostand von Konto A erniedrigt werden soll, so kann es vorkommen, dass beide den gleichen Ausgangskontostand lesen, in ihrer temporären Variablen speichern, davon subtrahieren und dann den neuen Wert schreiben.
- Je nachdem, ob der erste oder der zweite Thread beim Schreiben schneller ist, wird der Kontostand von Konto A um 10 oder um 20 erniedrigt. Der korrekte Wert wäre jedoch die Summe der Einzelabbuchungen, also 30, gewesen. Die Ausgabe des Programms sieht so (oder ähnlich) aus:

THREADS BEISPIEL 2

Vorher:

Konto 0: 30

Konto 1: 50

Konto 2: 100

Nachher:

Konto 0: 10

Konto 1: 70

Konto 2: 100

Nachher:

Konto 0: 10

Konto 1: 30

Konto 2: 120

Nachher:

Konto 0: 30

Konto 1: 30

Konto 2: 80

THREADS BEISPIEL 2

- Die Ursache des Problems liegt darin, dass der Vorgang der Abbuchung, also das Lesen, Subtrahieren und Schreiben in mehreren Schritten abläuft und so ein zweiter Thread mit einer eigentlich ungültigen Zahl arbeitet.
- Dieses Verhalten nennt man **Race-Condition**.
- Generell sind **Race-Conditions** Programmfehler, die nur manchmal auftreten, nämlich genau dann, wenn zufällig zwei parallele Threads zur gleichen Zeit auf bestimmte Objekte zugreifen.
- Derartige Fehler sind in der Praxis schwer zu lokalisieren.

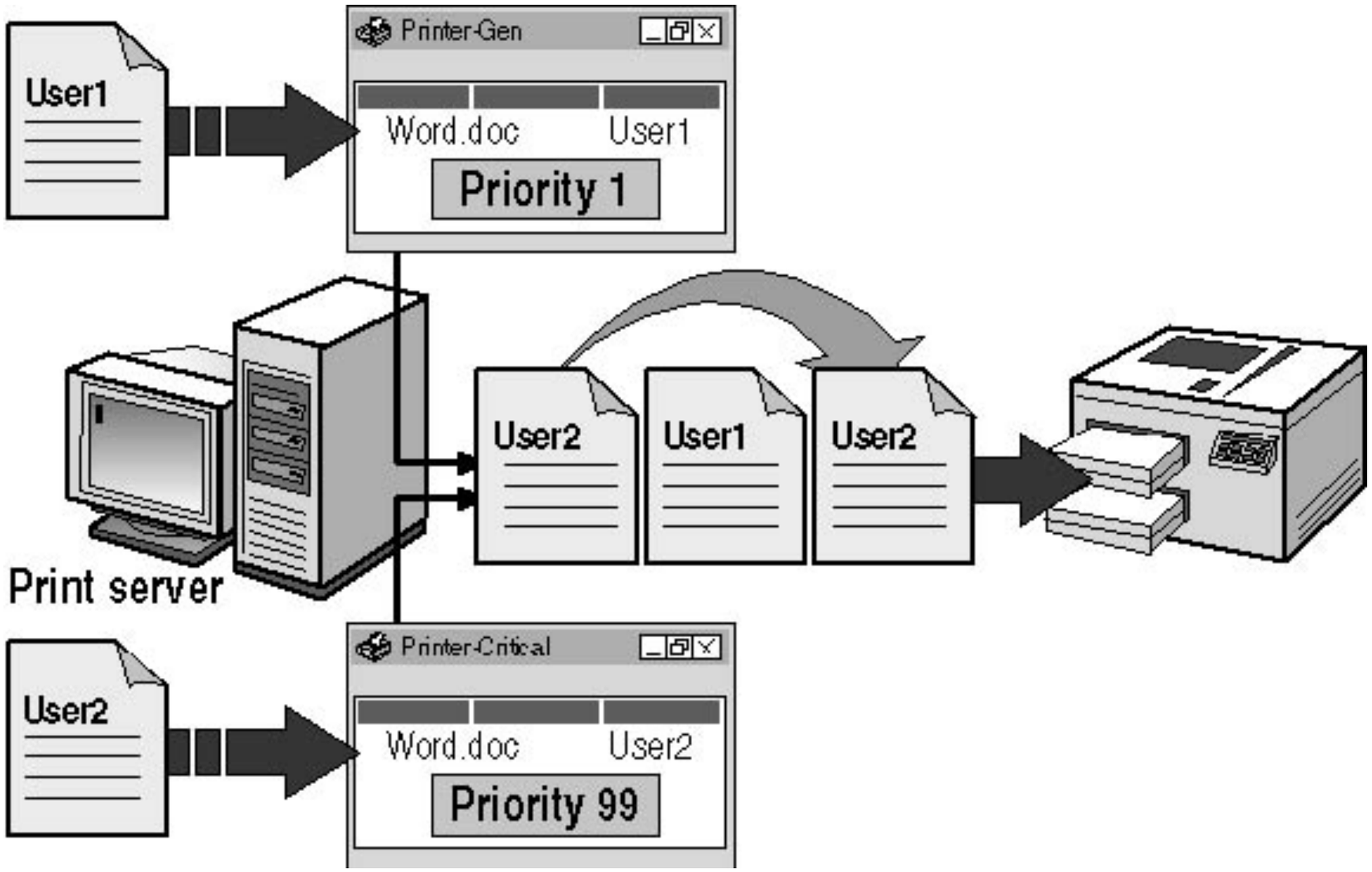
5.5. SYNCHRONISATION ÜBER KRITISCHE ABSCHNITTE

- Gemeinsam genutzte Daten:

```
class T extends Thread {
     static int result;

     public void run() {
          ...
     }
}
```

- Datenaustausch mehrere Objekte vom `Typ T` über result
- Leseoperationen ok, Schreiboperationen kritisch

PROBLEME BEIM GEGENSEITIGEN ZUGRIFF UND KRITISCHE ABSCHNITTE



PROBLEME BEIM GEGENSEITIGEN ZUGRIFF UND KRITISCHE ABSCHNITTE

- Bei bestimmten Aktionen möchte ich der Einzige sein, der eine Ressource nutzt
- Erst, wenn Aktion abgeschlossen, darf der nächste
- Ansonsten: Beispiel Drucker – Seiten abwechselnd gedruckt – im schlimmsten Fall einzelne Zeilen abwechselnd!

KRITISCHE ABSCHNITTE

- **Kritischer Abschnitt**
  - Zusammenhängender Codeblock, der von nur einem Thread gleichzeitig abgearbeitet werden darf und geschützt werden muss
- **Gegenseitiger Ausschluss / mutial exclusion**
  - Nur ein Thread arbeitet Programmteil ab (atomar)
- **Thread-sicher / Thread-safe**
  - Threads arbeiten Programm korrekt ab

NICHT KRITISCHE ABSCHNITTE

- Mehrere Threads verursachen nicht immer Probleme
  - Bsp. ***Immutable*** Objekte sind automatisch thread-safe (keine Schreibzugriffe, nur Lesezugriffe)
- In Java API gibt es viele Klassen, die nicht thread-safe sind
  - im Zweifelsfall in API-Dokumentation nachsehen!
  - `StringBuffer` (thread-safe), `StringBuilder` (nicht thread-safe)

PUNKTE PARALLEL INITIALISIEREN

- Annahme:
  - 2 Threads wollen Punkt initialisieren

Thread T1	Thread T2
`p.x = 1;`	`p.x = 2;`
`p.y = 1;`	`p.y = 2;`

Zwei Threads belegen beide den Punkt p

Thread T1	Thread T2	x/y
`p.x = 1;`		1/0
	`p.x = 2;`	2/0
	`p.y = 2;`	2/2
`p.y = 1;`		2/1

Mögliche sequentielle Abarbeitung der Punktbelegung

PUNKTE PARALLEL INITIALISIEREN

```
final Point p = new Point();


Runnable r = new Runnable() {
     @Override public void run() {
          int x = (int)(Math.random() * 1000), y = x;


          while ( true ) {
               p.x = x; p.y = y; // *
               int xc = p.x, yc = p.y; // *
               if ( xc != yc )
                    System.out.println( "Aha: x=" + xc + ", y=" + yc );
               }
          }
     };

new Thread( r ).start();
new Thread( r ).start();
```

AUSGABE:

Aha: x=58, y=116

Aha: x=116, y=58

Aha: x=58, y=116

Aha: x=58, y=116 ...

PUNKTE PARALLEL INITIALISIEREN

- Ergebnis hängt von Ausführungszeiten der einzelnen Operationen ab
- Abhängig vom Ort der Unterbrechung wird fehlerhaftes Verhalten produziert
- Dieses Szenario nennt man **race condition**
- Problem ist noch tiefgreifender:
  - z.B. i++ ist **nicht atomar**!

KRITISCHE ABSCHNITTE SCHÜTZEN

- Bsp. Toilette -> Tür geschlossen, warten
- In Java: **Monitor** (C. A. R. Hoare, 1978)
  - Realisierung mittels **Locks** (Schloss)
  - **Schloss** (Kritischer Abschnitt)
    - wird von Thread mittels **lock** geschlossen
    - und mittels **unlock** geöffnet

- 2 Konzepte:

Konstrukt	Eingebautes Schlüsselwort	Java-Standardbibliothek
Schlüsselwort/Typen	`synchronized`	`java.util.concurrent.locks.Lock`

Nutzungsschema	`synchronized`
	`{`
	`  Tue1`
	`  Tue2`
	`}`
		`lock.lock();`
		`{`
		`  Tue1`
		`  Tue2`
		`}`
		`lock.unlock();*`

Lock-Konzepte (Vereinfachte Darstellung, später mehr.)*

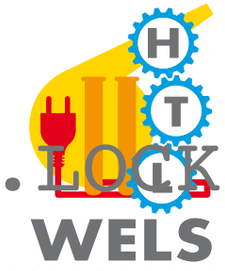
KRITISCHE ABSCHNITTE MIT REENTRANTLOCK SCHÜTZEN

```
final Lock lock = new ReentrantLock();


final Point p = new Point();
Runnable r = new Runnable() {
     @Override public void run() {
          int x = (int)(Math.random() * 1000), y = x;
          while ( true ) {
               lock.lock();
               p.x = x; p.y = y;
               int xc = p.x, yc = p.y;
               lock.unlock();
               if ( xc != yc )
                System.out.println( "Aha: x=" + xc + ", y=" + yc );
          }
     }
};

new Thread( r ).start();
new Thread( r ).start();
```

DIE SCHNITTSTELLE `JAVA.UTIL.CONCURRENT.LOCKS.LOCK`

- Interface `Lock`, mehrere Implementierung
  - `ReentrantLock` wichtigste Implementierung
- `void lock()`
  - wartet, kritischen Abschnitt betreten kann, und markiert ihn
  - bei `ReentrantLock` kann `lock()` wieder aufgerufen ohne sich selbst zu sperren
- `boolean tryLock()`
  - kann kritische Abschnitt nicht sofort betreten werden, Rückgabe `true`,
  - sonst `false` und es wird nicht gewartet
- `void unlock()`
  - verlässt kritischen Block

DIE SCHNITTSTELLE JAVA.UTIL.CONCURRENT.LOCKS.LOCK



- `ReentrantLock`
  - `ReentrantLock` wichtigste Implementierung
- `ReentrantLock()`
  - Erzeugt ein neues Lock-Objekt, das nicht dem am längsten Wartenden den ersten Zugriff gibt.wartet, kritischen Abschnitt betreten kann, und markiert ihn
- `ReentrantLock(boolean fair)`
  - Erzeugt ein neues Lock-Objekt mit fairem Zugriff, gibt also dem am längsten Wartenden den ersten Zugriff.
- `boolean isLocked()`
  - Fragt an, ob der Lock gerade genutzt wird und im Moment kein Betreten möglich ist.
- `final int getQueueLength()`
  - Ermittelt, wie viele auf das Betreten des Blocks warten.
- `int getHoldCount()`
  - Gibt die Anzahl der erfolgreichen lock()-Aufrufe ohne passendes unlock() zurück. Sollte nach Beenden des Vorgangs 0 sein.

- Kann der kritische Abschnitt nicht betreten werden, kann in der Zwischenzeit was anderes gemacht werden:

```
Lock lock = ...;
if ( lock.tryLock() )
{
    try {
        ...
    }
    finally { lock.unlock(); }
}
else
    ...
```

SYNCHRONISIEREN MIT `SYNCHRONIZED`

- Mit dem Schlüsselwort `synchronized` können ganze (kritische) Methoden geschützt werden:

```
synchronized void foo()
{
 ... kritischer Bereich
}
```

MIT SYNCHRONIZED SYNCHRONISIERTE BLÖCKE

- Jedes Objekt kann als Monitor dienen

```
synchronized ( objektMitDemMonitor )
{
    ...
}
```

- Somit kann auch über beliebigen Monitor synchronisiert werden:

```
synchronized( this )
{
   // Code der Methode.
}
```

DANN MACHEN WIR DOCH GLEICH ALLES SYNCHRONISIERT!

- Synchronisierte Methoden werden von der JVM überwacht
  - => **Rechenzeit**!
- Überflüssig synchronisierte Methode führt sehr lange Operationen (im schlimmsten Fall Endlosschleife aus)
  - Freigabe dauert zu lange
  - Kein Vorteil bei Mehrprozessorsystemen -> alles wird sequentiell abgearbeitet
- Gefahr eines **Deadlocks** (gegenseitiges Blockieren von Threads) steigt

FREIGABE IM FALL VON EXCEPTIONS

- Kommt es zu `RuntimeException`:
  - Automatische Freigabe des Locks
  - d.h. kein endloses Blockieren

DEADLOCKS

- Thread A belegt Ressource, die Thread B haben möchte
- Thread B belegt Ressource, die Thread A haben möchte
- Deadlocks können in Java nicht erkannt und verhindert werden
- => verhindern, dass dieser Zustand auftritt

PHILOSOPHENPROBLEM

THREADS – INHALT

SYNCHRONISATION ÜBER WARTEN UND BENACHRICHTIGEN

- wir können jetzt Daten in einer synchronisierten Art austauschen
- ABER: es ist nicht möglich, dass ein Thread das Ankommen von Daten signalisiert, während andere informiert werden wollen, wenn Daten bereitstehen
- Typische Beispiele für ein Problem mit „Warten und Bereitstellen“ sind Produzenten-Konsumenten (Producer-Consumer) Beispiele

DIE SCHNITTSTELLE CONDITION

- Interface Lock stellt Methode `newCondition()` zur Verfügung
  - `Condition condition = lock.newCondition();`
- über Conditions können sich mehrere Threads durch Warten/ Benachrichtigen synchronisieren

DIE SCHNITTSTELLE CONDITION

- **`Interface Condition:`**

`void await() throws InterruptedException`

- Wartet auf ein Signal, oder die Methode wird unterbrochen.

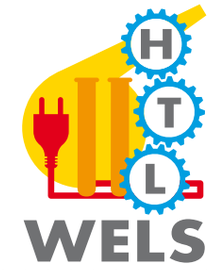
`void signal()`

- Weckt einen wartenden Thread auf.

VOR CONDITION KOMMT EIN LOCK

```
lock.lock();
try {
  condition.await();
} catch ( InterruptedException e ) {
...
} finally {
  lock.unlock();
}
```

MEHRERE WARTENDE

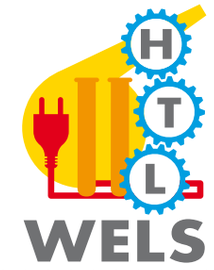


- Was wenn mehrere Threads warten und aufgeweckt werden wollen?
- `signal()` wählt aus der Liste der wartenden Threads einen aus und gibt ihm das (Weck)signal
- `signalAll()` hingegen weckt alle wartenden Threads auf

AWAIT MIT ZEITSPANNE

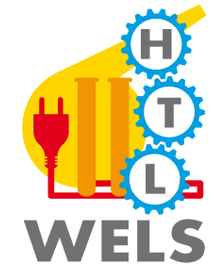
- ein **`await()`** wartet im schlechtesten Fall unendlich, falls kein **`signal()`** kommt, daher gibt es mehrere Varianten:
- **`long awaitNanos(long nanosTimeout) throws InterruptedException`**
  - Wartet eine bestimmte Anzahl Nanosekunden auf ein Signal, oder die Methode wird unterbrochen. Die Rückgabe gibt die Wartezeit an.
- **`boolean await(long time, TimeUnit unit) throws InterruptedException`**
- **`boolean awaitUntil(Date deadline) throws InterruptedException`**
  - Wartet eine bestimmte Zeit lang auf ein Signal. Kommt das Signal in der Zeit nicht, geht die Methode weiter und liefert true. Kam das Signal oder ein interrupt(), liefert die Methode false.
- **`void awaitUninterruptibly()`**
  - Wartet ausschließlich auf ein Signal und lässt sich nicht durch ein interrupt() beenden.

AUSGABE



- Aufgabe GitHub
  - `4105_Synchronisation`

AUSGABE



- Aufgabe GitHub
  - `4106_ProducerConsumer`

THREADS – INHALT

- **Semaphor (Dijkstra, 1968)**
  - Stellt sicher, dass nur eine bestimmte Anzahl an Threads auf ein Programmstück zugreifen
  - Klasse: `java.util. concurrent.Semaphore`
  - `Semaphore(int permits)`
    - Das neue Semaphor, das bestimmt, wie viele Threads in einem Block sein dürfen
- `void acquire()`
  - Versucht, in den kritischen Block einzutreten. Wenn der gerade belegt ist, wird gewartet. Vermindert die Menge der Erlaubnisse um eins.
- `void release()`
  - Verlässt den kritischen Abschnitt und legt eine Erlaubnis zurück.

DIE KLASSE SEMAPHORE

```
import java.util.concurrent.Semaphore;
public class SemaphoreDemo {
  static Semaphore semaphore = new Semaphore( 2 );

  static Runnable r = new Runnable() {
    @Override public void run() {
      while ( true ) {
        try {
          semaphore.acquire();
          try {
            System.out.println( "Thread=" + Thread.currentThread().getName() +
                                ", Available Permits=" + semaphore.availablePermits());
                        TimeUnit.SECONDS.sleep( 2 );
         } finally {semaphore.release();}
      } catch ( InterruptedException e ) {
        e.printStackTrace();
        Thread.currentThread().interrupt();
        break;
       }
    }
  }
};
```

DIE KLASSE SEMAPHORE

- Der kritische Abschnitt beginnt mit dem `acquire()` und endet mit `release()`
- Wichtig ist die richtige Ausnahmebehandlung
- Fünf Dinge müssen beachtet werden:
- `acquire()` kann eine `InterruptedException` auslösen => Behandlung
- `acquire()` kann wie `sleep()` eine `InterruptedException` auslösen können, Ausnahme signalisiert die Bitte, das Warten zu beenden => Schleifenabbruch mit `break`
- Immer, wenn `acquire()` erfolgreich war, muss `release()` folgen => `release()` im `finally`, sollte andere `RuntimeException` auftauchen
- `release()` darf nur erfolgen, wenn es zugehöriges `acquire()` gibt
  - `try { semaphore.acquire(); ... }`
  - `finally { semaphore.release(); }`
- wenn `acquire()` eine Ausnahme erzeugt, wird `release()` ausgelöst.
- `release()` erzeugt keine Ausnahme => keine Behandlung

DIE KLASSE SEMAPHORE

```
import java.util.concurrent.Semaphore;
public class SemaphoreDemo {
  static Semaphore semaphore = new Semaphore( 2 );

  static Runnable r = new Runnable() {
    @Override public void run() {
      while ( true ) {
        try {
          semaphore.acquire();
          try {
            System.out.println( "Thread=" + Thread.currentThread().getName() +
                                ", Available Permits=" + semaphore.availablePermits());
                        TimeUnit.SECONDS.sleep( 2 );
         } finally {semaphore.release();}
      } catch ( InterruptedException e ) {
        e.printStackTrace();
        Thread.currentThread().interrupt();
        break;
       }
    }
  }
};
```

DIE KLASSE SEMAPHORE

Drei Threads sollen sich koordinieren:

```
    public static void main( String[] args )
    {
      new Thread( r ).start();
      new Thread( r ).start();
      new Thread( r ).start();
    }
}
```

Nach dem Starten ist gut zu beobachten, wie jeweils zwei Threads im Abschnitt sind (eine Leerzeile symbolisiert die Wartezeit):

```
Thread=Thread-0, Available Permits=1
Thread=Thread-1, Available Permits=0

Thread=Thread-2, Available Permits=0
Thread=Thread-0, Available Permits=0

Thread=Thread-2, Available Permits=0
Thread=Thread-0, Available Permits=0
```

FAIR UND UNFAIR

- Ausgabe:
  - Thread 0, 1 und 2 können Aufgaben ausführen
  - plötzlich entsteht Sequenz 0, 2, 0
- Was ist mit Thread 1?
  - `acquire()` berücksichtigt nicht, wer am längsten wartet
  - aus der Liste der Wartenden wird beliebiger Thread ausgewählt
- Faires Verhalten: über Konstruktor von Semaphore

FAIR UND UNFAIR

- Programm ändern:

```
static Semaphore semaphore = new Semaphore( 2, true );
```

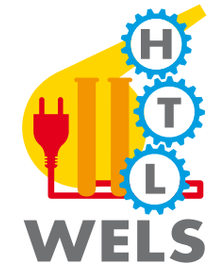
- Nun bekommen wir eine Ausgabe wie die folgende:

```
Thread=Thread-0, Available Permits=1
Thread=Thread-1, Available Permits=0
Thread=Thread-2, Available Permits=0
Thread=Thread-0, Available Permits=0
Thread=Thread-1, Available Permits=0
Thread=Thread-2, Available Permits=0
Thread=Thread-0, Available Permits=0
Thread=Thread-1, Available Permits=0
```

CONCURRENT COLLECTIONS

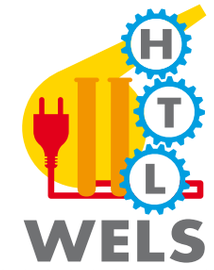
- Collections-API nicht Thread-safe!
- Es gibt aber Thread-safe Klassen im Package `java.util.concurrent`
- Diese Klassen implementieren folgende Interfaces
  - `Vector`: Concurrent ArrayList
  - `BlockingQueue`: FIFO Datenstruktur
  - `ConcurrentMap`: Schlüssel/Wertepaare

AUSGABE



- Aufgabe GitHub
  - `4105_Synchronisation`
  - `Löse Aufgabe 2)`

AUSGABE



- Aufgabe GitHub
  - `4106_ProducerConsumer`
  - Erstelle eine Kopie des Programms und löse es mittels einer Concurrent Collection

QUELLEN

- *„Java 7 – Fortgeschrittene Programmierung“*, Christian Ullenboom, HERDT-Verlag für Bildungsmedien GmbH