Practical Concurrent and Parallel Programming XIV
End of course / Exam

Raúl Pardo and
Jørgen Staunstrup

Agenda

- Examination
- Important concepts you need to know well (no matter the question)
- Race conditions vs data races, and thread-safety
- Monitors
- Fairness
- Happens-before

Examination

Examination – Dates & guidelines (UPDATED)

- Exam hand-out: Dec 19th at 8.00
- Exam hand-in deadline: Dec 20th at 14.00
- There will be ***<u>no</u>*** fraud control
  - Please ignore the info in the slides for week 13 regarding this issue

Collaboration during the exam

- Collaboration is not allowed
- The exam must be solved individually
- You can think of this exam as regular on-premises exam at ITU
  - But with more time and access to the course literature
- The objective of the exam is to measure your individual knowledge on the subject
  - Not the collective knowledge of a group

References in the exam

- No need to provide references to mandatory readings or lecture slides (although it is not forbidden)
- There is no specific format for providing references
- For instance,
  - Lecture slides: title and course week for the slide deck
  - Book: Authors, title (and chapter) of specific material
  - Research Paper: Authors, title (may be year)
  - Internet resource: URL

Course Code

- You can find the course code in the course description page in learnIT
  - https://learnit.itu.dk/local/coursebase/view.php?ciid=1006
  - KSPRCPP2KU

COURSE INFO

Language:
English
ECTS Points:
7.5
Course Code:
KSPRCPP2KU
Participants Max:
160
Offered To Guest Students:
Yes
Offered To Exchange Students:
Yes
Offered As A Single Subject:
Yes
Price For EU/EEA Citizens (Single Subject):
10625 DKK

On clear answers

- There is no *a* way to provide clear/acceptable answers
- The main reason for insisting on explaining answers was the increase in answers of the type: See file XXX.java
  - Answers of this kind will likely result in no points
- We are simply advising to be as clear as possible in your written answers
  - Specially we insist on avoiding simply referring to the code
- This is not the “PCPP answer style” or anything like that

No-examples or anti-examples and their risks

- When a being asked about thread-safety
  - *Answer*: XXX is thread-safe because I use an AtomicInteger/lock/semaphore/…
  - *Risk*: This implies that any program using AtomicInteger/lock/semaphore/… is thread-safe

Is this the case? Why?

- When being asked to implement something:
  - *Answer*: See file XXX.java
  - *Risk*: Here you are relying on the ability of the examiner to understand your code
- When begin asked to explain the output of a performance measurement
  - *Answer*: When I run the program I observe that X is faster than Y
  - *Risk*: This only describes the output, but does not explain it. An explanation should include the reason why X is faster than Y

Executable code in the exam

- Recall that code must be executable using the steps in the [mini guide on using Gradle]
  - This also requires your code to be compatible with JDK 8
- We will provide a Gradle project with all the dependences we used in the course (and skeleton code related to the exam questions)
  - You only need to focus on writing Java code related to the questions

Important concepts you need to know well (no matter the question)

Threads in Java – Example II

- What was is the problem in the previous program?

- To answer this question we need to understand
  - Atomicity
  - States of a thread
  - Non-determinism
  - Interleavings

Atomicity

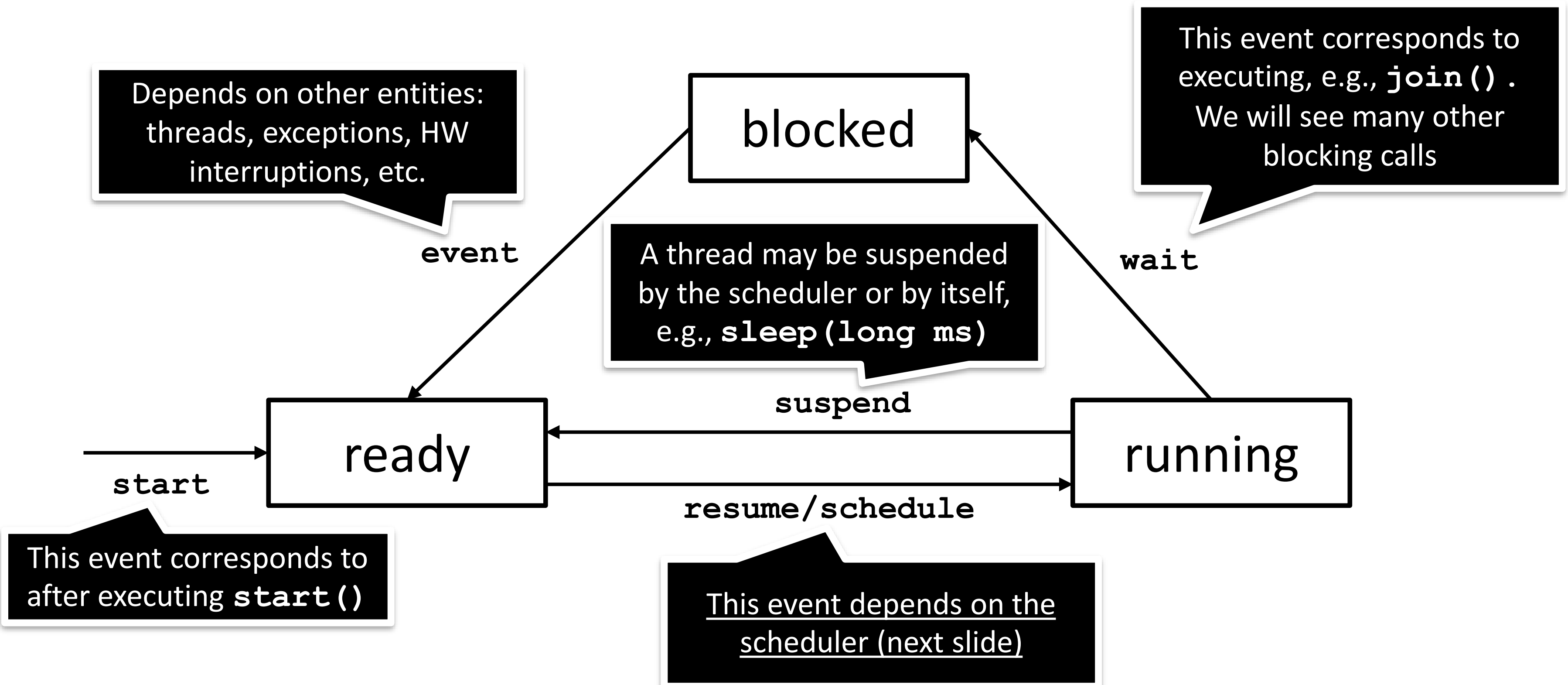
- The program statement **`counter++`** is not *atomic*
- *<u>Atomic statements are executed as a single (indivisible) operation</u>*

```
public class Turnstile extends Thread {
   public void run() {
        for (int i = 0; i < PEOPLE; i++) {
            counter++;
        }
   }
}
```

```
int temp = counter;
counter = temp + 1;
```

<u>Watchout</u>: Just because a program statement is a one-liner, it doesn´t mean that it is atomic

States of a thread (simplified)



Non-determinism | Scheduler

- In all operating systems/executing environments a *scheduler* selects the processes/threads under execution
  - Threads are selected *non-deterministically*, i.e., no assumptions can be made about what thread will be executed next

- Consider two threads t1 and t2 in the ready state; *t1(ready)* and *t2(ready)*
  1. *t1(running) -> t1(ready) -> t1(running) -> t1(ready) -> …*
  2. *t2(running) -> t2(ready) -> t2(running) -> t2(ready) -> …*
  3. *t1(running) -> t1(ready) -> t2(running) -> t2(ready) -> …*
  4. Infinitely many different executions!

Interleaving

- The statements in a thread are executed when the thread is in its “running” state

- An *interleaving* is a possible sequence of operations for a concurrent program
  - Note this: a sequence of operations for a concurrent program, not for a thread. Concurrent programs are composed by 2 or more threads.

Interleaving syntax

- The drawings above are not suitable for thinking about possible interleavings

- When asked to provide an interleaving, use the following syntax

```
<thread>(<step>), <thread>(<step>), …
```

Interleaving – Example II (textual)

*Given the initial memory state on the right, provide an interleaving such that after two threads t1, t2 execute the program on the right the value of **counter==1***

Memory

counter=0	
i=0	i=0

Program

```
public void run() {
  int temp = counter;   // (1)
  counter = temp + 1;   // (2)
}
```

counter=0	
temp=0	temp=0

t1(1),

counter=0	
temp=0	**temp=0**

t2(1),

counter=1	
temp=0	temp=0

t1(2),

counter=1	
temp=0	temp=0

t2(2)

Race conditions, data races & Thread-safety (also very important no matter the question)

Race Conditions

- *A **race condition** occurs when the result of the computation depends on the interleavings of the operations*

Data Races

- *A **data race** occurs when two concurrent threads:*
  - *Access a shared memory location*
  - *At least one access is a write*

Race Conditions vs Data Races

Not all <u>race conditions</u> are <u>data races</u>

- Threads may not access shared memory
- Threads may not write on shared memory

```
public void p() {
  print("P") //P1
}
```

```
public void q() {
  print("Q") //Q1
}
```

```
Interleaving 1: T1(P1)T2(Q1)
Output: P Q
Interleaving 2: T1(Q1)T2(P1)
Output: Q P
```

Not all <u>data races</u> result in <u>race conditions</u>

- The result of the program may not change based on the writes of threads

```
public void p() {
  x=1;   //P1
}
```

```
public void q() {
  x=1;   //Q1
}
```

```
Interleaving 1: T1(P1)T2(Q1)
Final state: x==1
Interleaving 2: T1(Q1)T2(P1)
Final state: x==1
```

Thread-safe program

*A concurrent **program** is said to be **thread-safe** if and only if it is race condition free*

Do not confuse thread-safe classes with thread-safe programs. Thread-safe programs are not defined in Goetz. But it is aligned with the definition of correctly synchronized programs in JLS

PCPP teaching team

Thread-safe classes

WARNING: Note that, in this course, *thread-safety* is not an umbrella term for code that seem to behave correctly in concurrent environments.

*A **class** is said to be **thread-safe** if and only if*
no concurrent execution of
method calls or field accesses (read/write)
result in race conditions

Note that this definition is independent of class invariants as opposed to Goetz Chapter 4. This definition is more similar to Goetz Chapter 2, page 18.

PCPP teaching team

Thread-safe classes

- To analyse whether a class is thread-safe, we must identify/consider:
  - Class state
  - Escaping
  - (Safe) publication
  - Immutability
  - **Mutual exclusion**

Very important slide

Use as a reference when answering questions about thread-safety

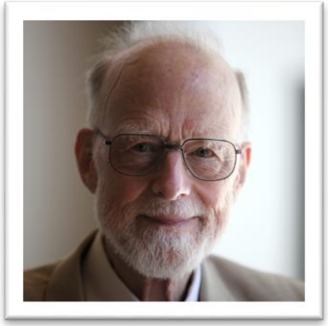
Mutual Exclusion

- An ideal solution to the mutual exclusion problem must ensure the following properties:
  - Mutual exclusion: at most one thread executing the critical section at the same time
  - Absence of *deadlock*: threads eventually exit the critical section allowing other threads to enter
  - Absence of *starvation*: if a thread is ready to enter the critical section, it must eventually do so

Monitors

Monitors

- A *monitor* is a structured way of encapsulating data, methods and synchronization in a single modular package
  - First introduced by Tony Hoare (right photo, see optional readings) and the Danish computer scientist Per Brinch Hansen (left photo)



- *A monitor consists of:*
  - Internal state (data)
  - Methods (procedures)
    - All methods in a monitor are mutually exclusive (ensured via locks)
    - Methods can only access internal state
  - Condition variables (or simply conditions)
    - Queues where the monitor can put threads to wait
- In Java (and generally in OO), monitors are conveniently implemented as classes

Conditions

- Conditions are used when a thread must wait for something to happen, e.g.,
  - A writer thread waiting for all readers and/or writer to finish
  - A reader waiting for the writer to finish

- Queues in condition variables provide the following interface:
  - **`await()`** – releases the lock, and blocks the thread (on the queue)
  - **`signal()`** – wakes up a threads blocked on the queue, if any
  - **`signalAll()`** – wakes up all threads blocked on the queue, if any

- When threads wake up the acquire the lock immediately (before the execute anything else)

Monitors | Readers-Writers Problem

- We define four methods to lock and unlock read and write access to the shared resource

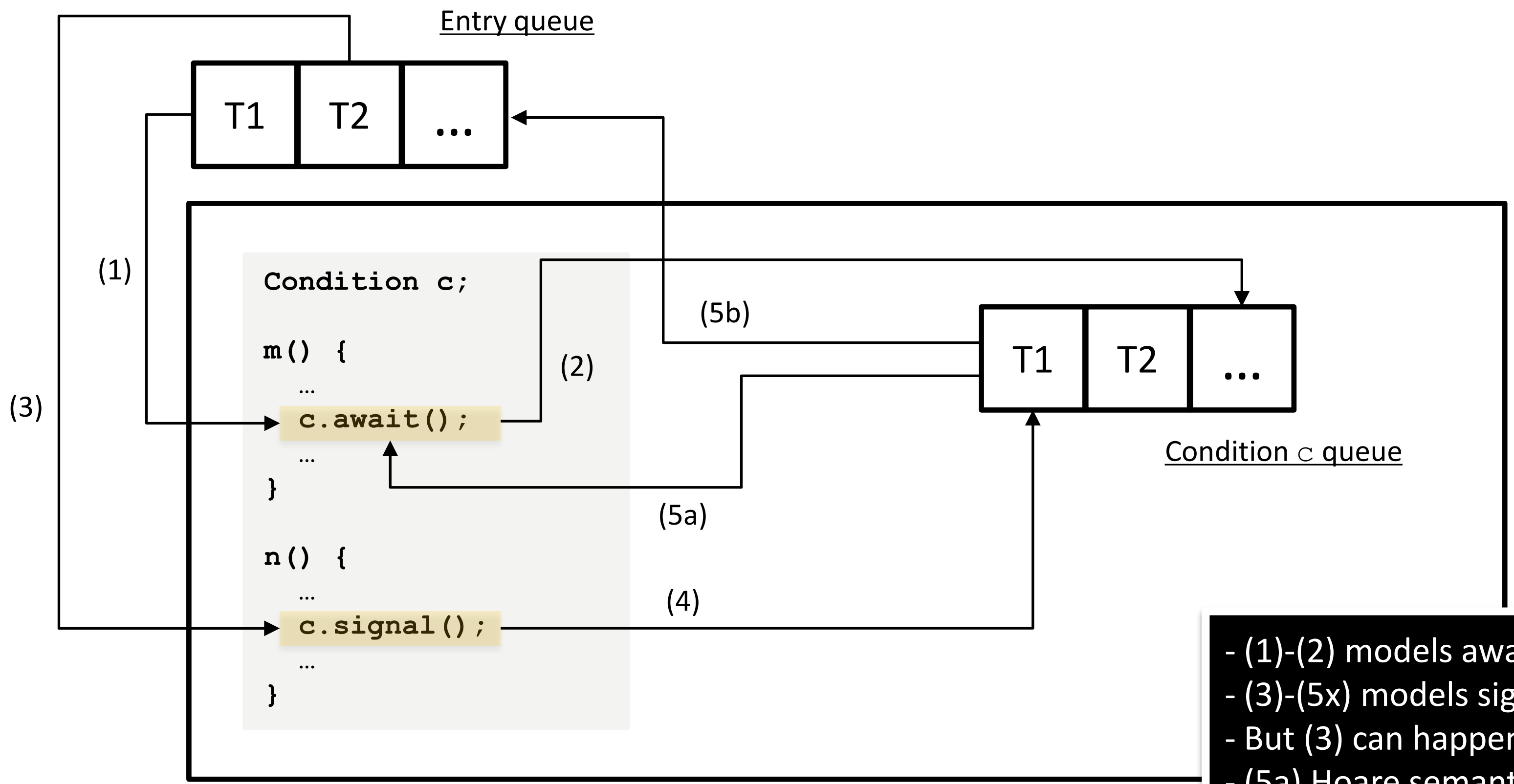
```
public void readLock() {
      lock.lock();
      try {
          while(writer)
            condition.await();
          readers++;
      }
      catch (InterruptedException e) {…}
      finally {lock.unlock();}
}

public void readUnlock() {
      lock.lock();
      try {
          readers--;
          if(readers==0)
            condition.signalAll();
      }
      finally {lock.unlock();}
}
```

```
public void writeLock() {
      lock.lock();
      try {
          while(readers > 0 || writer)
            condition.await();
          writer=true;
      }
      catch (InterruptedException e) {…}
      finally {lock.unlock();}
}

public void writeUnlock() {
      lock.lock();
      try {
          writer=false;
          condition.signalAll();
      }
      finally {lock.unlock();}
}
```

Monitor behaviour - schematic representation



- (1)-(2) models await scenario
- (3)-(5x) models signal scenario
- But (3) can happen before (1)
- (5a) Hoare semantics
- (5b) MESA semantics (and Java)

Fairness

Fairness

- There exist two notions of fairness
  - Weak fairness: A thread that is continuously active will eventually make progress
  - Strong fairness: A thread that is infinitely often active will eventually make progress
- Note that fairness is not always achievable programmatically
  - The scheduler must ensure it
  - This is specially important for strong fairness

Fairness (a bit more formally)

Weak fairness

```
m(Semaphore s) {
  while(true) {
    s.acquire();   // (1)
    // c.s.        // (2)
    s.release();   // (3)
  }
}
```

```
main() {
  Semaphore s = new Semaphore(2)
  Thread t1 = new Thread(() -> m(s));
  Thread t2 = new Thread(() -> m(s));
  t1.start();t2.start();
  …
}
```

- Let *I* denote the set of all possible interleavings of this program
- If [T1(1)T1(2)T1(3)]* is in *I*, then *weak fairness* is not enforced
  - In other words, if there is an interleaving where T2 never executes, then weak fairness is violated
  - This is because the semaphore has enough capacity for two threads, so T2 is *continuously active*
  - Is the interleaving above possible (i.e., included in the set of all interleavings)?

Fairness (a bit more formally)

Strong fairness

```
m(Semaphore s) {
  while(true) {
    s.acquire();  // (1)
    // c.s.        // (2)
    s.release();  // (3)
  }
}
```

```
main() {
  Semaphore s = new Semaphore(1)
  Thread t1 = new Thread(() -> m(s));
  Thread t2 = new Thread(() -> m(s));
  t1.start();t2.start();
  …
}
```

- If there exists an interleaving *i* in *I* such that [*T1(2)*T2(2)*]* is not in *i*, then strong fairness is not ensured
  - In other words, there must be an interleaving where T1(2) and T2(2) are executed infinitely often

Fairness (a bit more formally)

Strong fairness

```
m(Semaphore s) {
  while(true) {
    s.acquire();   // (1)
    // c.s.        // (2)
    s.release();   // (3)
  }
}
```

```
main() {
  Semaphore s = new Semaphore(1)
  Thread t1 = new Thread(() -> m(s));
  Thread t2 = new Thread(() -> m(s));
  t1.start();t2.start();
  …
}
```

- If there exists an interleaving *i* in *I* such that [*T1(2)*T2(2)*]* is not in *i*, then strong fairness is not ensured
  - In other words, there must be an interleaving where T1(2) and T2(2) are executed infinitely often
- The condition above is sufficient because after T1(3) the operation T2(1) is enabled (and vice versa), and this happens infinitely often

Fairness (a bit more formally)

Strong fairness

```
m(Semaphore s) {
  while(true) {
    s.acquire();  // (1)
    // c.s.       // (2)
    s.release();  // (3)
  }
}
```

```
main() {
  Semaphore s = new Semaphore(1)
  Thread t1 = new Thread(() -> m(s));
  Thread t2 = new Thread(() -> m(s));
  t1.start();t2.start();
  …
}
```

- If there exists an interleaving *i* in *I* such that [*T1(2)*T2(2)*]* is not in *i*, then strong fairness is not ensured
  - In other words, there must be an interleaving where T1(2) and T2(2) are executed infinitely often
- The condition above is sufficient because after T1(3) the operation T2(1) is enabled (and vice versa), and this happens infinitely often
- Because in this case T2 is not continuously active (only initially and after T1(3)), the property above is not enough to show that weak fairness is not enforced

Fairness

- In the exam we will not go into details regarding weak vs strong fairness
- For the exam it is more important to consider fairness as absence of starvation
  - Is it possible that a thread that is ready to enter the critical section never makes progress?
- Absence of starvation can be achieved programmatically in some cases
  - Recall the fair readers-writers program
  - Fair flags for Semaphores, see above

Absence of starvation

```
m(Semaphore x, y) {
  while(true) {
    x.acquire();   // (1)
    // c.s.        // (2)
    y.release();   // (3)
  }
}
```

```
main() {
  Semaphore s1 = new Semaphore(1)
  Semaphore s2 = new Semaphore(0)
  Thread t1 = new Thread(() -> m(s1,s2));
  Thread t2 = new Thread(() -> m(s2,s1));
  t1.start();t2.start();
  …
}
```

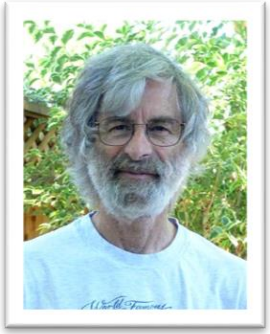
- The program above ensures that both threads run infinitely often
- In fact, allows a single interleaving
  [T1(1)T1(2)T1(3)T2(1)T2(2)T2(3)]*
- Note that this is not always possible; the problem specification may require interleavings which lead to possible starvation problems
  - Recall that for the fair readers-writers, writers may starve (if they keep jumping from the condition queue to the entry queue)
  - This can only be prevented by an scheduler that ensures strong fairness

Happens-before

Happens-before

We will focus on the Java happens-before relation (page 341 Goetz and JLS documentation)

- In fact, we can now characterize an order of execution between some of the operations of a program
- We say that an operation a *happens-before* than operation b, denoted as $a \rightarrow b$, iff
  - a and b belong to the same thread and a appears before b in the thread definition
  - a is an **`unlock()`** and b is a **`lock()`** on the same lock
- In the absence of *happens-before* relation between operations, the JVM is free to choose any execution order
  - In that case we say that operations are executed *concurrently*
  - Sometimes denoted as $a \mid\mid b$
- *Happens-before* is a *partial order* over operations of concurrent programs
  - Reflexive, transitive, antisymmetric
- "Happened-before" was first introduced by Leslie Lamport for distributed systems
  - See optional readings



volatile

In this program the output (0,0) is not possible (explanation based on happens-before)

- Because of volatile we have (the following premises)
  $a \coloneqq 1 \rightarrow x \coloneqq b$ and
  $b \coloneqq 1 \rightarrow y \coloneqq a$
- Assume, by contradiction, that the output of the program is (0,0). This can only happen as a result of any of the following interleavings
  $one(x \coloneqq b), other(y \coloneqq a), one(a \coloneqq 1), other(b \coloneqq 1)$ (1) or
  $one(x \coloneqq b), other(y \coloneqq a), other(b \coloneqq 1), one(a \coloneqq 1)$ (2) or
  $other(y \coloneqq a), one(x \coloneqq b), one(a \coloneqq 1), other(b \coloneqq 1)$ (3) or
  $other(y \coloneqq a), one(x \coloneqq b), other(b \coloneqq 1), one(a \coloneqq 1)$ (4)
- In (1) and (2), it holds $x \coloneqq b \rightarrow a \coloneqq 1$ which contradicts the premise $a \coloneqq 1 \rightarrow x \coloneqq b$
- In (3) and (4), it holds $y \coloneqq a \rightarrow b \coloneqq 1$ which contradicts the premise $b \coloneqq 1 \rightarrow y \coloneqq a$
- Therefore, the output (0,0) is not possible

```
// shared variables
x=0;
y=0;
volatile a=0;
volatile b=0;

// Threads definition
Thread one = new Thread(() -> {
    a=1;
    x=b;
});
Thread other = new Thread(() -> {
    b=1;
    y=a;
});
one.start();other.start();
one.join();other.join();
System.out.println("("+x+","+y+")");
```

volatile

In this program the output (0,0) is not possible (explanation based on happens-before)

- Because of volatile we have (the following premises)
  $a \coloneqq 1 \rightarrow x \coloneqq b$ and
  $b \coloneqq 1 \rightarrow y \coloneqq a$

We also have other premises coming from volatile, depending on the scheduler:
$a \coloneqq 1 \rightarrow y \coloneqq a$ or $y \coloneqq a \rightarrow a \coloneqq 1$ and
$b \coloneqq 1 \rightarrow x \coloneqq b$ or $x \coloneqq b \rightarrow b \coloneqq 1$

These premises come from our earlier definition of happens-before for volatile variables:
A write to a volatile variable happens-before any subsequent read to the volatile variable

Note that
$(a \coloneqq 1 \rightarrow y \coloneqq a$ or $y \coloneqq a \rightarrow a \coloneqq 1) \quad \neq \quad (a \coloneqq 1 \nrightarrow y \coloneqq a$ and $y \coloneqq a \nrightarrow a \coloneqq 1)$

- Therefore, the output (0,0) is not possible

```
// shared variables
x=0;
y=0;
volatile a=0;

inition
ew Thread(() -> {

 new Thread(() -> {

her.start();
er.join();
System.out.println("("+x+","+y+")");
```

volatile

In this program the output (0,0) is not possible (explanation based on happens-before)

- Because of volatile we have (the following premises)
  $a := 1 \rightarrow x := b$ and
  $b := 1 \rightarrow y := a$

We also have other premises coming from volatile, depending on the scheduler:
$a := 1 \rightarrow y := a$ or $y := a \rightarrow a := 1$ and
$b := 1 \rightarrow x := b$ or $x := b \rightarrow b := 1$

These premises come from our earlier definition of happens-before for volatile variables:
A write to a volatile variable happens-before any subsequent read to the volatile variable

Note that
$(a := 1 \rightarrow y := a$ or $y := a \rightarrow a := 1) \quad \neq \quad (a := 1 \nrightarrow y := a$ and $y := a \nrightarrow a := 1)$

```
// shared variables
x=0;
y=0;
volatile a=0;

inition
ew Thread(() -> {

new Thread(() -> {

her.start();
er.join();
x+","+y+")");
```

$a \nrightarrow b$ actually means $a \,||\, b$. In other words, absence of happens before relation between the actions

- Therefore, the output (0,0) is no

Java happens-before rules, Goetz Chp. 16

The rules for *happens-before* are:

Program order rule. Each action in a thread *happens-before* every action in that thread that comes later in the program order.

Monitor lock rule. An unlock on a monitor lock happens-before every subsequent lock on that same monitor lock.[3]

[3]. Locks and unlocks on explicit Lock objects have the same memory semantics as intrinsic locks.

Volatile variable rule. A write to a volatile field happens-before every subsequent read of that same field.[4]

[4]. Reads and writes of atomic variables have the same memory semantics as volatile variables.

Thread start rule. A call to Thread.start on a thread *happens-before* every action in the started thread.

Thread termination rule. Any action in a thread *happens-before* any other thread detects that thread has terminated, either by successfully return from Thread.join or by Thread.isAlive returning false.

Interruption rule. A thread calling interrupt on another thread happens-before the interrupted thread detects the interrupt (either by having InterruptedException thrown, or invoking isInterrupted or interrupted).

Finalizer rule. The end of a constructor for an object *happens-before* the start of the finalizer for that object.

Transitivity. If A *happens-before* B, and B *happens-before* C, then A *happens-before* C.

Mandatory assignments

- To be eligible for the exam, 5 (or more) mandatory assignments must be approved

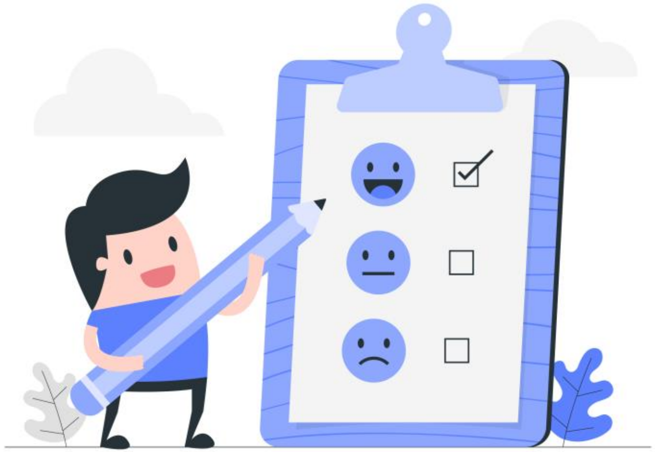
- You will get confirmation in the feedback for assignment 6
  - *"Your assignments have been approved and you may take the exam"*

- *<u>It is your responsibility to let us know if there are any errors in grading</u>*
  - For instance, missing grades, ungraded assignment, etc.

- There will be a final extra deadline in Dec 14^{th} to hand-in assignments that have not yet been approved
  - With no possibility of re-submission and written feedback

Course Evaluation Survey

Please participate in the course evaluation



Questions ?