Karlsruhe Institute of Technology

Programmierparadigmen

Prof. Dr. Ralf Reussner

Topic 5.4

Parallel Programming with Java – Thread Programming Basics

DEPENDABILITY OF SOFTWARE-INTENSIVE SYSTEMS
INSTITUTE OF INFORMATION SECURITY AND DEPENDABILITY, FACULTY OF INFORMATICS

dsis.kastel.kit.edu

Lecture Sessions Overview

Today

Next session

17.12.2025	Lecture
19.12.2025	Exercise
	Christmas Break
07.01.2026	Lecture
09.01.2026	Exercise
14.01.2026	Lecture
16.01.2026	Exercise
21.01.2026	*Lecture by Professor Sebastian Erdweg*

ILIAS course always contains the latest updates.

Overview on Today's Lecture

- Content
  - Creating control threads
  - Race conditions, mutual exclusion, synchronization
  - Lambdas, functional interfaces and method references
  - Basic thread lifecycle
  - Memory Consistency Errors
- Learning goals: participants –
  - are able to implement and start concurrently executed threads
  - understand the problems induced by concurrent access on shared memory
  - know common synchronization mechanisms
  - know about problems due to missing or false synchronization
  - understand the impact of a missing happens-before relationship
  - know statements and keywords to establish a happens-before relationship

Foundation: Lambda Expressions

- Introduced in Java 8
- Bring functional programming capabilities into Java
- Allow to define anonymous functions
- *Example*: Instead of defining an addition function …

```
public int add(int i, int j) {
        return i + j;
}
```

… it is possible to define a lambda expression

```
(int i, int j) -> i + j
```

- The parameter types of lambda expressions can usually be inferred from the context and can thus be omitted:

```
(i, j) -> i + j
```

Foundation: Functional Interfaces

- Introduced in Java 8
- A functional interface …
  - is an interface that declares a single method
  - serves as a target for lambda expressions
- It allows to define anonymous functions with lambda expressions instead of anonymous classes, which is useful to:
  - Reduce code lines
  - Allow sequential / parallel execution of stream operations *(discussed later)*
  - Pass behavior into methods
  - Increase efficiency with laziness

Foundation: Functional Interfaces

- Example for a functional interface realizing a predicate

```
@FunctionalInterface
interface Predicate {
        boolean check(int value);
}
```

- … expected by a function calculating a conditional sum

Could be expressed even more compact with streams

```
public int sum(List<Integer> values, Predicate predicate) {
        int result = 0;
        for (int value : values) {
                if (predicate.check(value)) result += value;
        }
        return result;
}
```

- … used with a given `values` list, summing up values greater than 5

```
sum(values, i -> i > 5);
```

- *Usage here*: Pass behavior to a method

Foundation: Method References

- An alternative for lambda expressions are method references
  - Allow to specify a reference to a method, for example as an argument for an expected functional interface
  - Syntax: *`object::method`*
  - Object can be replaced by a class name if the method is static
- The lambda expression from the previous example can be expressed in an explicit method

```
class SimpleCheckers {
        public static boolean checkGreaterThanFive(Integer value) {
                return value > 5;
        }
}
```

… and then passed to the `sum` method by reference

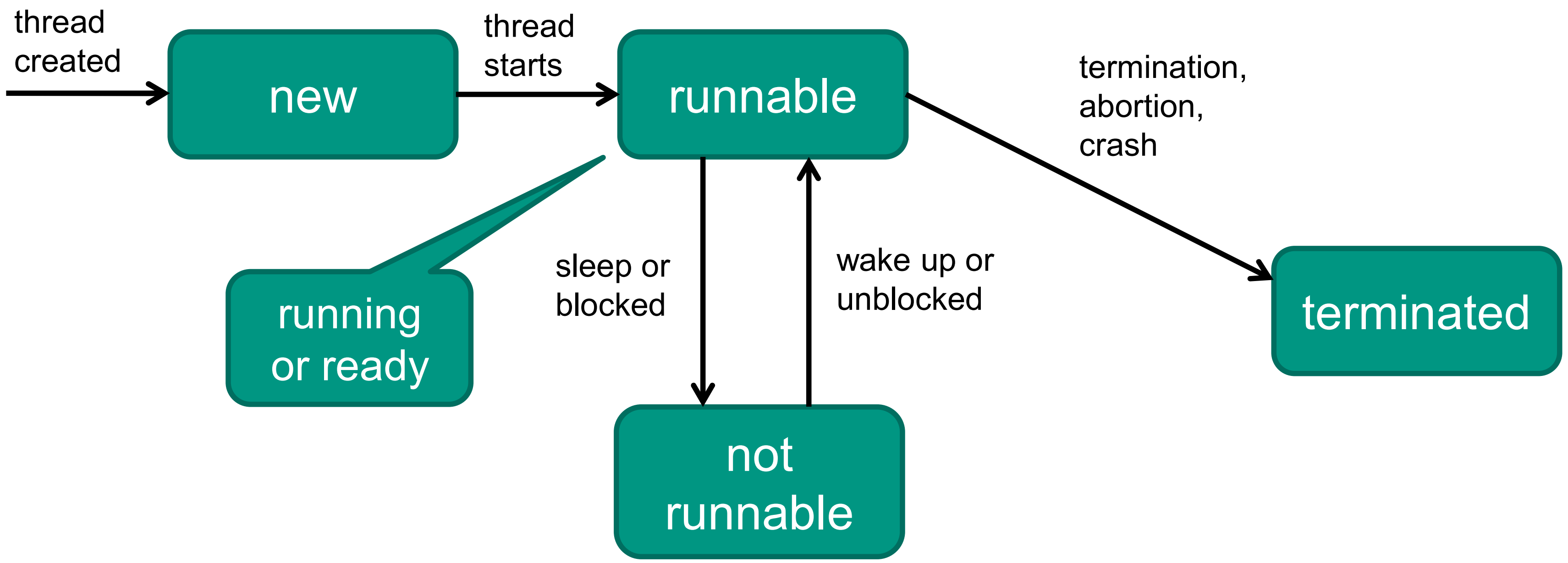
```
sum(values, SimpleCheckers::checkGreaterThanFive);
```

Thread Programming

- Within one process, several threads can be executed concurrently
- A thread …
  - encapsulates „independently“ executable code
  - has its own stack
  - shares the heap with other threads of the same process
- Achieving concurrency using threads was rather difficult before Java was released
- Java supports concurrency using threads since Java 1.0
  - Based on a shared memory model
  - Integrated language constructs

Thread Programming in Java

- Each Java application has at least one thread: the *Main* thread
- Threads can be in different states, which are in Java (simplified):



Constructs for creating parallelism

- Creating and starting a new, concurrent control path cannot be realized by a Java library, but requires the support of the VM.
- The new **control thread** has
  - its own stack and program counter.
  - Access to the shared main memory (heap).
  - possibly local copies of main memory data (due to the caches)

Constructs for creating parallelism (1)

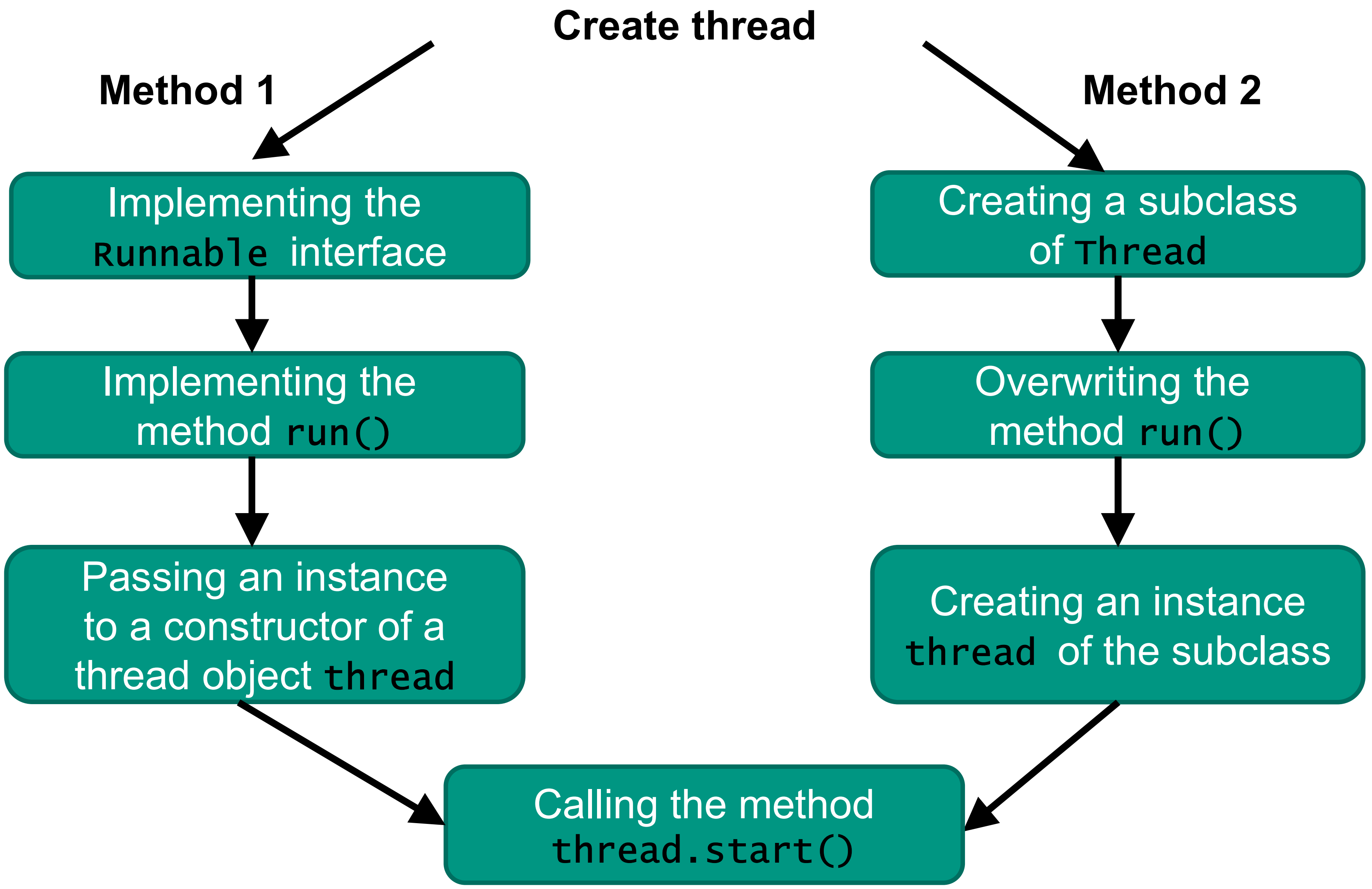
- Since Java 1.0, there have been built-in classes and interfaces for parallel programming:
  - Interface `java.lang.Runnable`

```
public interface Runnable {
  public abstract void run();
}
```

  - Class `java.lang.Thread`

```
public class Thread implements Runnable {
  public Thread(String name);
  public Thread(Runnable target)
  public void start();
  public void run();
  ...
}
```

Constructs for creating parallelism (2)



Create thread
Method 1
Method 2
Implementing the `Runnable` interface
Implementing the method `run()`
Passing an instance to a constructor of a thread object `thread`
Creating a subclass of `Thread`
Overwriting the method `run()`
Creating an instance `thread` of the subclass
Calling the method `thread.start()`

Constructs for creating parallelism: Example (1)

- Class that implements `Runnable`:

```
class ComputeRun implements Runnable {
  long min, max;

  ComputeRun (long min, long max) {
    this.min = min; this.max = max;
  }

  public void run () {
    // Parallel task
  }
}
```

- Create and start control thread:

```
ComputeRun c = new ComputeRun(1, 20);
new Thread(c).start();
```

- Start the new control thread. Only this creates the new activity
- The `start()` method returns immediately, the new control thread continues to work in parallel.
- No restart: `start()` may only be called once.
- do not call `run()` directly

Constructs for creating parallelism: Example (2)

- Class that inherits from `Thread`:

```
 class ComputeRun extends Thread {
  long min, max;

  ComputeRun (long min, long max) {
    this.min = min; this.max = max;
  }

  public void run () {
    // Parallel task
  }
}
```

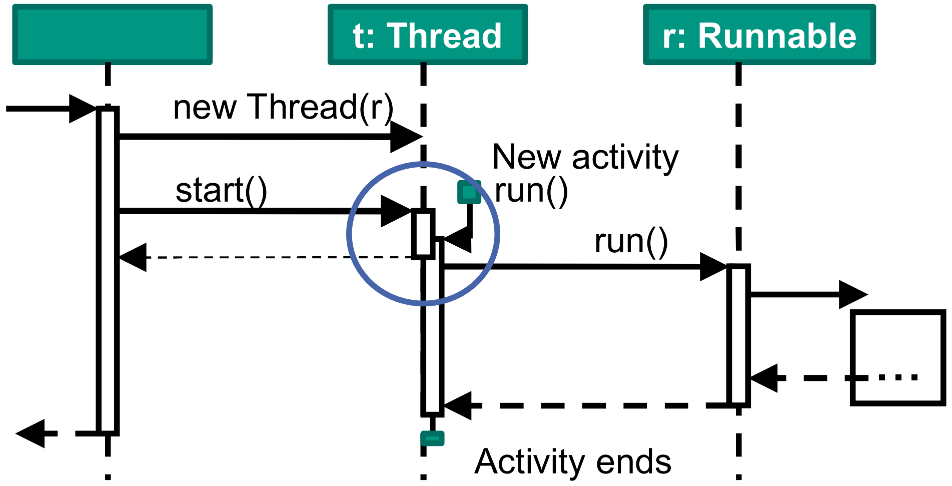
- Create and start control thread:

```
ComputeRun t = new ComputeRun(1, 20);
t.start();
```

Constructs for creating parallelism: Example (3)

- Construct with anonymous inner class also possible:

```
Thread t = new Thread (
  new Runnable() {
    public void run() {
      //Parallel task
    }
  }
);
t.start();
```



- `Thread` itself implements the `Runnable` interface.
- When the new activity is started (`start()`), it first executes the `run()` method of its own thread object.
- The implementation of `Thread.run()` in turn calls `run()` on the `Runnable` `object` passed to the constructor.

Runnable vs. thread

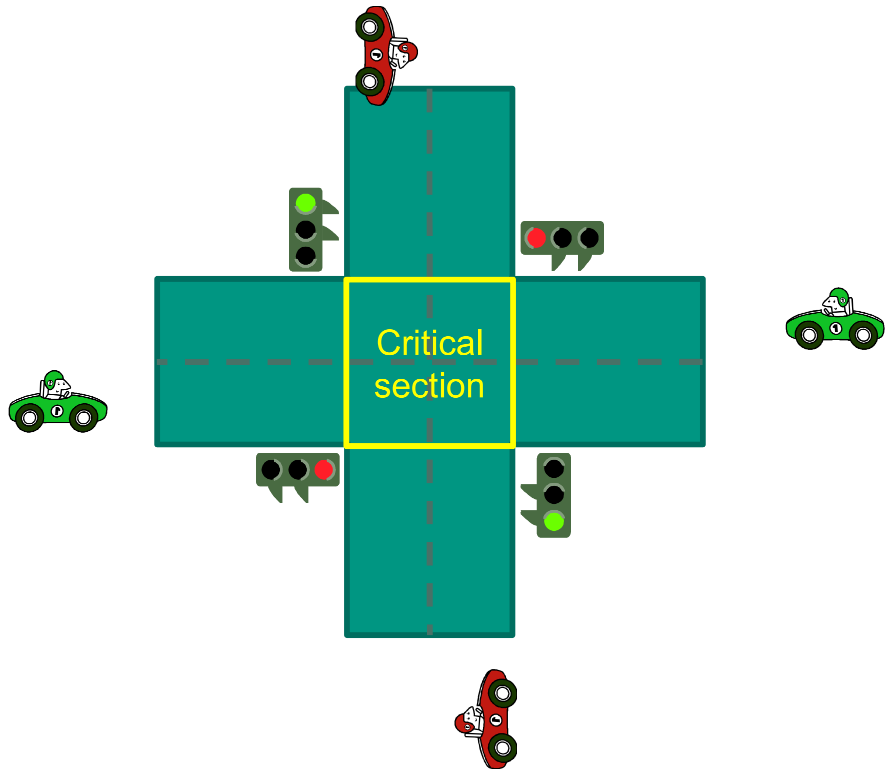
- Why a separate class that implements `Runnable?`
  - Overriding `run()` in a subclass of `Thread` would have the same effect.
  - No „inheritance pollution“ (implementing instead of extending).
- But: Better modularization when using the `Runnable` interface:
  - Encapsulating the task in its own object (or class) makes it available in a sequential context with less overhead.
  - With distribution: The task can be sent via the network (`thread` cannot be serialized).

Coordination

- Basic coordination mechanisms are built into the language.
  - Mutual exclusion:
    - Marking of critical sections that may only be entered by one activity at a time.
  - Waiting for "Events" and notification
    - Activities can wait for state changes caused by other activities.
    - Activities inform other, waiting activities about signals.
  - Interruptions:
    - An activity that is waiting for an event that does not (or no longer) occur can be canceled via an exception condition.

Coordination: Example

- A road junction with traffic lights:



Coordination: For what?

- Two activities execute the following code in parallel. Both have access to the same "global" variable `globalVar`.

```
int globalVar = 1;

...

if (globalVar > 0) {
  globalVar--;
}
```

- Can `globalVar` become negative?

Coordination: For what?

- Answer: Yes, a *race condition* can occur. A race condition can occur if at least 2 threads access the same data, with at least one thread writing.
- The following race situation with thread entanglement could occur:

Thread 1 **Thread 2**

```
                // initial globalVar == 1
if (globalVar > 0) {
                                         if (globalVar > 0) {
    // Break                               globalVar--;
    // Break                             }

  globalVar--;
}
```

Coordination: Critical sections (1)

- A *critical section* is a code segment that may only be executed by one thread at a time
- The exclusive execution of certain code segments may be necessary for many reasons, e.g.
  - Access to shared data, with potentially simultaneous write operations on this data
  - Control of shared resources (e.g. hard disks) that only allow exclusive access
- To avoid race situations, such critical sections must be protected
  - Only one activity may execute a critical section at a time
  - Before entering a critical section, it must be ensured that no other activity is carrying it out

Coordination: Critical sections (2)

- **Atomicity**: Java guarantees that access to individual variables is atomic.
  - Variables of type `double` and `long` are excluded (due to 64 bit instead of 32 bit)
- Accesses to several variables or successive read and write operations (as in the example) are not executed atomically.
  - Also be careful with "**i++**" . Looks atomic, but consists of several instructions that can form races with others!
- Even if access is atomic, synchronization is generally still required in order to securely synchronize the data that may have been cached locally in the cache by an activity with the main memory (see later slides).

Coordination: Monitor (1)

- In Java, monitors can be used to protect critical sections.
- Monitor has data and operations.
  - A monitor offers these two operations, among others:
    - `enter()` "enters" the monitor.
    - `exit()` "exits" the monitor.

Coordination: Monitor (2)

- Principle:
  - When `enter()` is called, an activity occupies a free monitor.
  - If an activity attempts to enter a monitor that is already occupied, it is blocked until the monitor is released again.
  - The monitor remains occupied until the occupying activity finally calls `exit()`.
  - The same activity can enter a monitor several times.

Coordination: Monitor (3)

- Monitors can be used to avoid race situations in critical sections:
  - enter the monitor: `enter()`.
  - execute the critical section.
  - exit the monitor: `exit()`.

```
// monitor.enter()

if (globalVar > 0) {
  globalVar--;
}

// monitor.exit()
```

Coordination: Monitors & Synchronization (1)

- In Java, any object can be used as a monitor.
  - but not primitive types such as int, float, ...
- The virtual machine knows two commands
  - 0xC2 monitorenter <object-ref>
  - 0xC3 monitorexit <object-ref>
- These two commands may only occur in pairs in a block.
  - This prevents "forgetting" to enable the monitor.
  - Before the bytecode is executed, the bytecode verifier (among other things) checks the nesting of the monitor requests and releases.

Coordination: Monitors & Synchronization (2)

- In the language definition, the paired use of monitor request and release is enforced by a block syntax:

```
/*synchronized block*/
synchronized (obj) {
  // critical
  // section
}
```

```
/*synchronized method*/
synchronized void foo(){
  // whole method body is
  // critical section
}
```

- A synchronized method is equivalent to a method whose body is enclosed in a block synchronized to `this` (or the class object in the case of a static method).

Coordination: Monitors & Synchronization (3)

- When executing a synchronized block...
  - the monitor of the associated object is entered: `enter()`
  - the instructions of the block are executed
  - the monitor is exited: `exit()`
- Synchronization is always linked to an object
  - for a `synchronized(x)` statement:
    the specified object x
  - for a synchronized instance method `a.foo()`:
    the current object `this`
  - for a synchronized static method `A.sfoo()`:
    the class object: `A.class`

Coordination: Monitors & Synchronization (4)

- What happens when an activity attempts to enter a busy monitor?
  - The activity is blocked uninterruptibly.
  - There is no escape except by releasing the monitor.
  - There is no test like this: "`wouldBlock(obj)`".
    - Why not?
  - (But: there is `Thread.holdsLock(Object)`
    - Static method with which an activity can find out for itself whether it is holding a monitor.)
- The same activity can enter a monitor any number of times (useful for recursion, for example).

Coordination: Monitors & Synchronization (5)

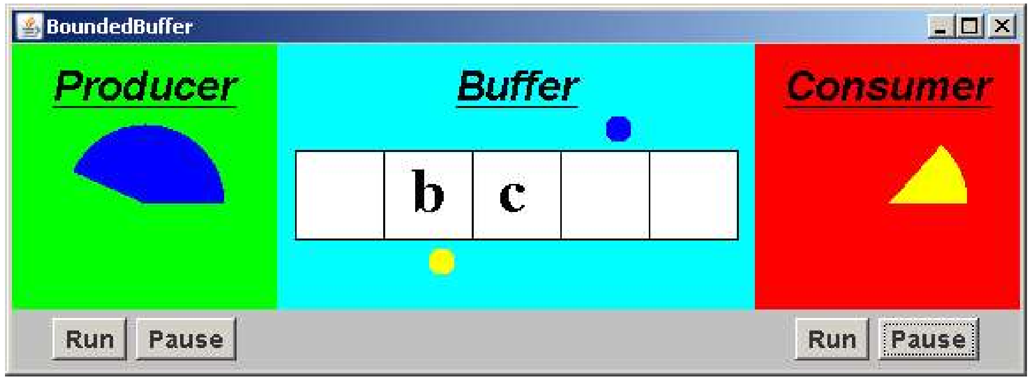
- In is the way we can repair the race situation from the example?

```
synchronized (someObject) {
  if (globalVar > 0) {
     globalVar--;
  }
}
```

- The possible execution sequences of two activities are restricted.
  - What else could be useful?

Coordination: Monitors & signaling (1)

- Mutual exclusion is not enough:
  - The completion of a task can depend on the progress of another activity.
  - Example: Sample "Producer - Consumer"
    - The "manufacturers" place orders in a queue.
    - Several "consumers" receive the orders and execute them.
  - A consumer can only continue if the queue is not empty.
  - The manufacturer must stop when the queue is full.



Coordination: Monitors & signaling (2)

- Producer/consumer - what is wrong here?
- Within the **same** class:

```
// Manufacturer:
synchronized void put(Work w) {
  while (queue.isFull()) {/*NOP*/}
  queue.add(w);
}
```

```
// Consumer:
synchronized Work take() {
  while (queue.isEmpty()) {/*NOP*/}
  return queue.remove();
}
```

Wrong!

Coordination: Monitors & signaling (3)

- **Good idea**: Access to the queue is synchronized.
- **Bad idea**: The manufacturer and consumer wait "actively" if they cannot continue working.
  - Waste of computing time!
- **Bad idea**: The consumer keeps the monitor occupied while waiting.
  - The manufacturer can never add new work to the queue because it blocks when trying to enter the monitor.
  - Conversely, the same applies to the consumer.
  - This can be avoided by making two separate classes, one for producer one for consumer, however, than reading queue state and changing queue state is not atomar anymore (as `put`-operations can interrupt `take`-operations and vice versa.)

Coordination: Monitors & signaling (4)

- Repairing the example
  - If a guard condition fails, an activity must give up its calculation time.
  - If possible, the control should be passed on directly to an activity that can continue the calculation.
  - While an activity is waiting on a guard condition, it must release the monitor.

Coordination: Constructs for waiting and notification (1)

- Reminder: Any object can be a monitor...
- Methods for this in `java.lang.Object`

```
public final void wait()
                  throws InterruptedException;
public final void wait(long timeout)
                  throws InterruptedException;
public final void wait(long timeout, int nanos)
                  throws InterruptedException;
public final void notify();
public final void notifyAll();
```

Coordination: Constructs for waiting and notification (2)

- To be able to call `wait` and `notify`, the current activity with `synchronized` must have already entered the associated monitor.
  - This cannot (always) be checked by the translator. If this rule is violated, an `IllegalMonitorStateException is` thrown at runtime.
- In synchronized methods, the monitor is `this`, and instead of `this.{wait|notify|notifyAll}()` can be called without `this.`

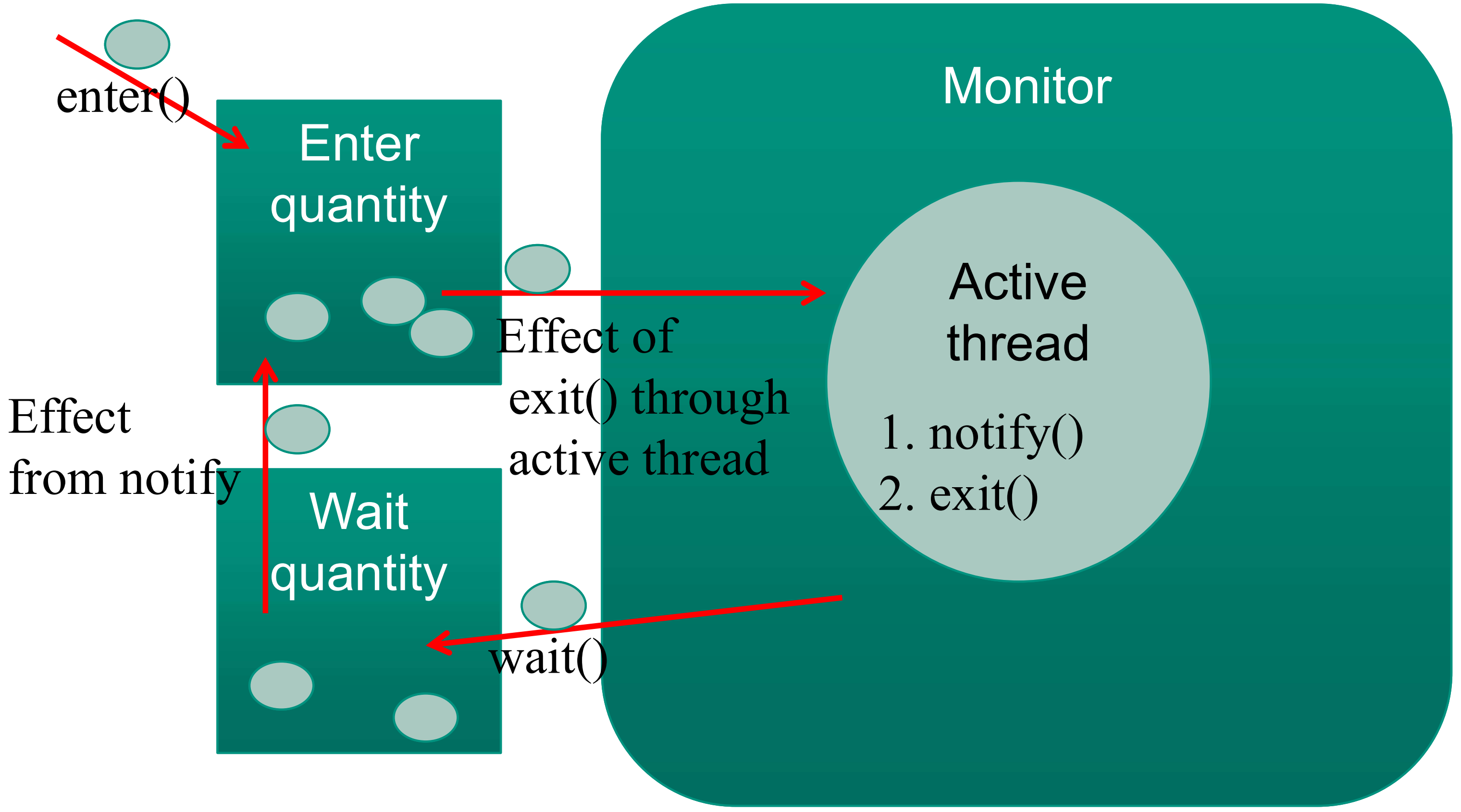
Coordination: Constructs for waiting and notification (3)

- `wait()` puts the current thread in a wait state until a signal arrives at this object. The following happens:
  - The current activity is "put to sleep", i.e. it releases the processor.
  - The activity is inserted into a (VM-internal) wait set at the monitor of the object at which `wait()` is called.
  - The monitor in question is released while waiting.
    - All other monitors remain occupied
  - Other activities can now compete for the monitor.
- Variant: `wait(long timeout, [int nanos])` limits the wait time to the specified value and then returns.
  - However, the waiting time may be longer because you may have to wait for the monitor to be released.

Coordination: Constructs for waiting and notification (4)

- `notify()` and `notifyAll()` send signals to waiting activities of the object in question. The following happens:
  - "Sleeping" activities (which have called wait()) are in a set associated with the monitor (the wait set).
    - `notify()` sends a signal to any (but at least one) activity from the Wait set.
    - `notifyAll()` sends a signal to all activities from the Wait set.
  - The signaled ("woken up") activities are removed from the wait set and compete for the monitor again (possibly with other existing activities) as soon as the monitor is released.

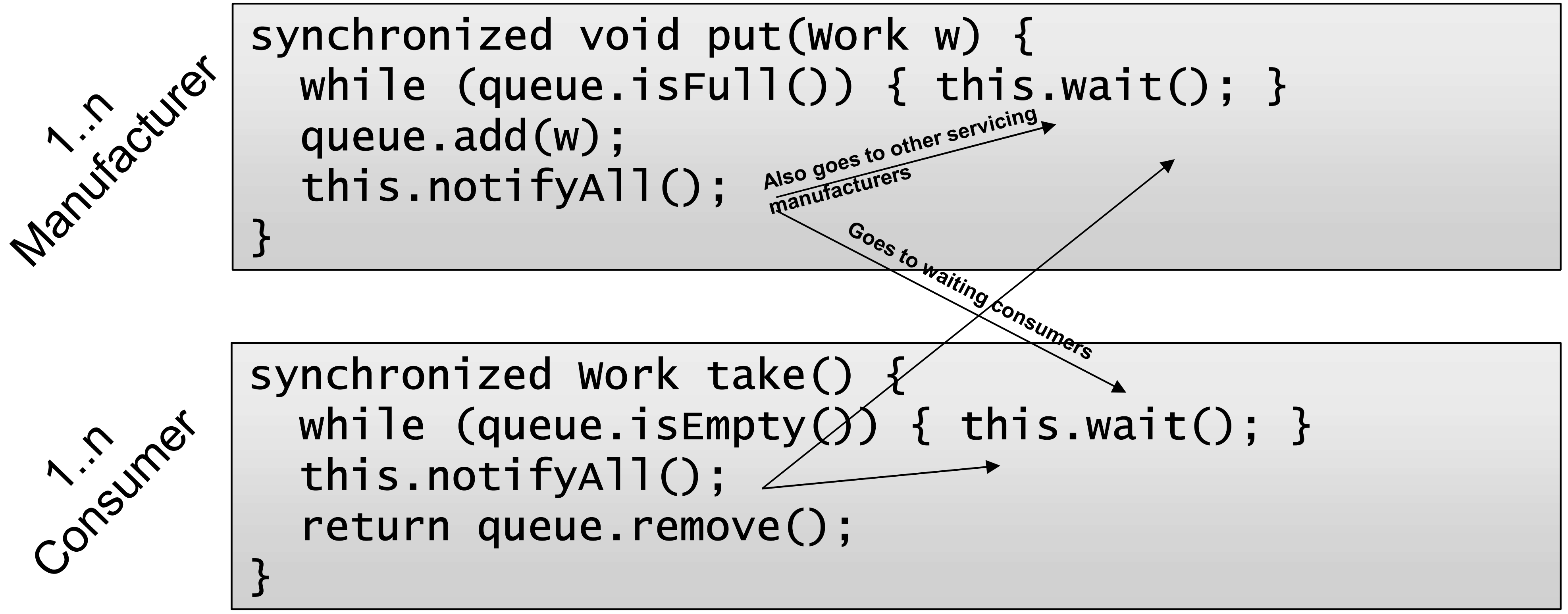
How Wait-Notify works



With notify(), a thread is moved from the wait set to the enter set and competes for the monitor again as soon as the active thread releases the monitor with exit(). The selection of the thread is not specified for neither the wait set or the enter set.
notifyAll() moves all existing threads from the Wait set to the Enter set.

Coordination: Constructs for waiting and notification (5)

- Repair of the example:
  Waiting at guard conditions, signals for condition changes.


1..n Manufacturer
Also goes to other servicing manufacturers
Goes to waiting consumers
1..n Consumer

```
synchronized void put(Work w) {
  while (queue.isFull()) { this.wait(); }
  queue.add(w);
  this.notifyAll();
}
```

```
synchronized Work take() {
  while (queue.isEmpty()) { this.wait(); }
  this.notifyAll();
  return queue.remove();
}
```

Coordination: Safety instructions & rules of thumb (1)

- A signal sent with `notify()/notifyAll()` only reaches an activity that is already waiting when it is sent!
    - The signal is not saved on the monitor.
    - The signal has no effect if no one is waiting.
- Waiting for signals alone can lead to errors, e.g.
    - `wait()` without guard condition
        - FALSE: ... `synchronized(obj) {wait(); }` ...
    - `notify()` without change of state
        - FALSE: ... `synchronized(obj) {notify(); }` ...
    - Why?

Coordination: Safety instructions & rules of thumb (2)

- Extract from the Java language specification (p. 581)
  - "Implementations are permitted, although not encouraged, to perform **"spurious wake-ups"** -
    - to remove threads from waitsets and
    - Thus enable resumption without explicit instructions to do so.
    - Notice that this provision necessitates the Java coding practice of using wait only within loops that terminate only when some logical condition that the thread is waiting for holds."
- Spurious: fake, false, supernumerary.
- This means that activities can be woken up randomly without changing the status of an object.

Coordination:
Safety instructions & rules of thumb (3)

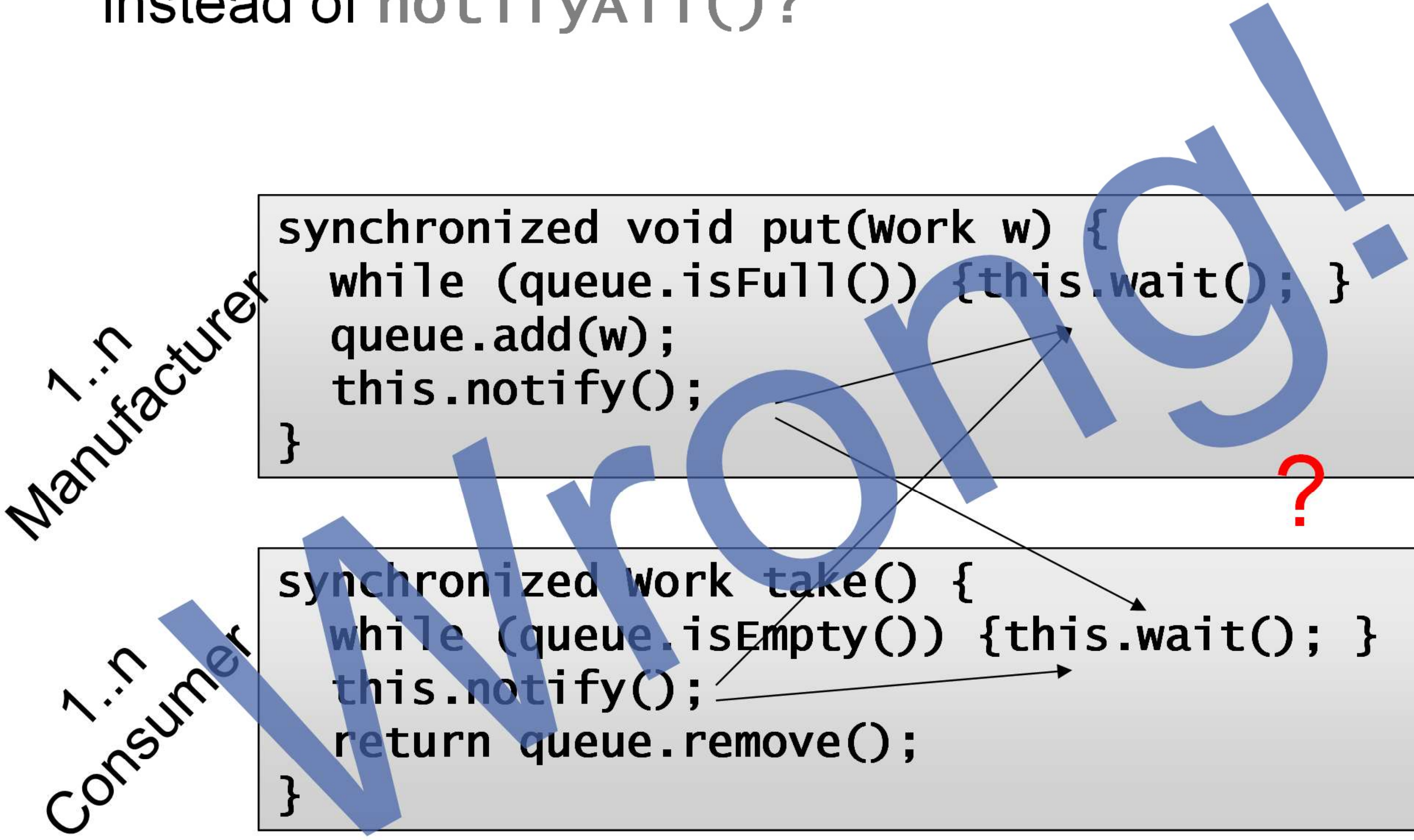
- Always check before and after waiting for the condition!
  - Always use `wait()` in a loop!
    - FALSE: ... `if (! isCondition) {wait(); }` ...
  - Reasons
    - Before: Do not put the control thread "to sleep" unnecessarily
    - After: Not visible why a signal was sent. "Wake up" could have been caused by a "wrong" signal. For example, manufacturers send signals to manufacturers **AND** consumers with `notifyAll()`. However, manufacturers should only ever escape from `wait()` if queue offers space.

Coordination: Safety instructions & rules of thumb (4)

- In a robust program, all `notify()` calls can be replaced by `notifyAll()`.
- A thread selected by `notify()` may not be able to make process. Than signal is “lost”. This problem is avoided by using `notifyAll()`.
- If the monitor is accessible from the outside (`public`), someone else could ...
  - ... steal signals (by keeping other activities waiting).
  - ... unexpectedly send more signals.
- Bulletproof: Always use `notifyAll()`.
  - Some VMs make no distinction between `notify()` and `notifyAll()`!
    - This is explicitely permitted by the JVM specification.

Coordination: Safety instructions & rules of thumb (5)

- Example: What happens if you only use `notify()` instead of `notifyAll()`?



```
synchronized void put(Work w) {
  while (queue.isFull()) {this.wait(); }
  queue.add(w);
  this.notify();
}
```

```
synchronized Work take() {
  while (queue.isEmpty()) {this.wait(); }
  this.notify();
  return queue.remove();
}
```

Coordination: Safety instructions & rules of thumb (6)

- If using "notify()“ only, the program can block or work suboptimally.
- Example: The queue with the work packs is empty.
  - A signal sent by a consumer can affect another waiting consumer instead of a manufacturer.
  - This consumer "swallows" this signal and the manufacturer continues to "sleep".
  - The same applies to signals from manufacturers.

Interruption (1)

- **Problem**: How do you end an activity that is waiting for signals that will no longer arrive?
- Illustration using the last example:
  - Consumers demand work "for all eternity".
  - However, no new work arrives after the manufacturers have finished.
  - How do you solve this problem? Naive: Introduce a special case, for example, set a zero reference in the list of work packages.
    - But what if zero is already "occupied" or simply a valid value?
- Easier with Java language constructs: An interrupt is sent to the relevant activities (`InterruptedException`).

Interruption (2)

- If there is no exception handling, the compiler reports this as fault
- Exception handling is directed at the controlled thread, not to the monitor.

```
synchronized void put(Work w) {
  while (queue.isFull()) {
    try {
      this.wait();
    } catch (InterruptedException ex) {
      // ??
    }
  }
}
```

Interruption (3)

- An interruption is sent directly to an activity:

```
    Thread t;
    t.interrupt();
```

- If t is not currently waiting, the interruption is signaled by the activity t the next time a `wait()` method is called.
  - This means that the interruption request is not lost
- If an activity is interrupted, `wait()` throws an `InterruptedException.`
  - The interruption itself cannot be interrupted
  - The `InterruptedException` must be declared and forwarded or intercepted and handled (`catch`).

Interruption (4) - Rules of thumb

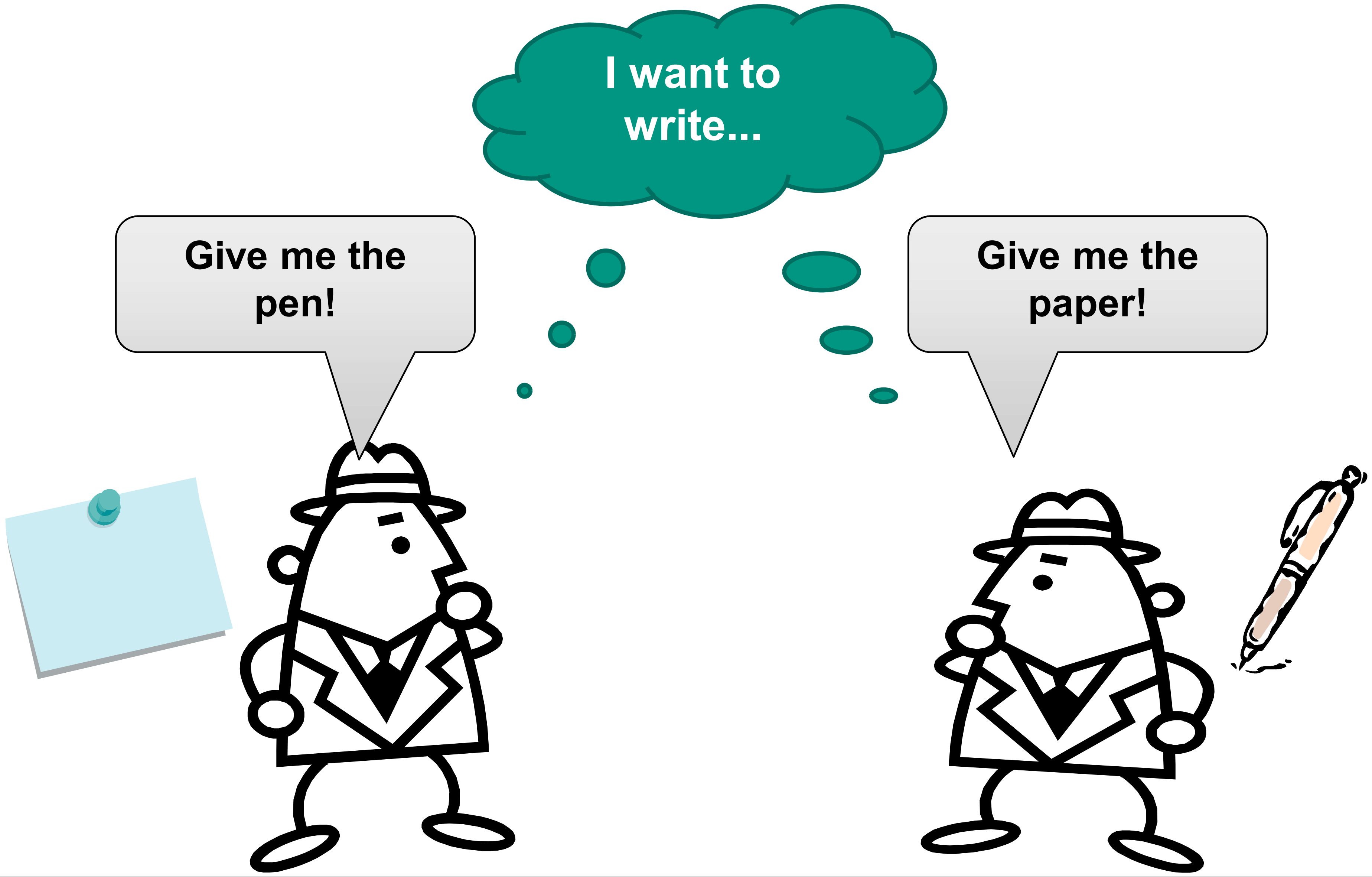
- If no reasonable handling can be specified for an `InterruptedException`, it should be forwarded with `throw`
  - This also applies to any other exception condition...
  - The surrounding code or caller should be able to handle this.
- Use interruptions to end activities cleanly (instead of constructing special cases).
  - This is only possible if these interruptions are not "caught" in trivial treatment routines without effect.
    So **not like this**:

```
try {
  wait();
} catch (InterruptedException ex)
{/*nothing*/}
```

Deadlock (1)

- Despite `synchronized,` other problems can still occur.
- deadlock, blockages
  (deadlock, *deadly embrace)*
  - Blockade caused by a cyclical dependency.
  - Causes all threads involved to remain in the waiting state.

Deadlock (2)



Deadlocks (3)

If multiple threads permanently wait for each other, a deadlock occurred: none of the threads can ever make progress

Threads can reach a deadlock if they compete for multiple monitors, locks or the like

Trivial example: Can a deadlock occur here?

```java
synchronized(m1) {
    synchronized(m2) {
        // do something
    }
}
```

```java
synchronized(m2) {
    synchronized(m1) {
        // do something
    }
}
```

If both threads reach this place simultaneously, a deadlock occurs.

How does a trivial avoidance strategy look like?

Deadlocks (4)

A deadlock *can* only occur iff all four Coffman conditions hold:

Mutual exclusion: Only one thread can use an unshareable resource at one point of time. Further threads that need the resource have to wait.

Hold and wait: A thread that already holds a resource requests access to additional resources (and must potentially wait for them).

No preemption: A resource can only be released by the thread that holds it. Releasing a resource cannot be enforced from outside.

Circular wait: A circular dependency between threads holding and requesting resources must exist. For threads $t_1, \ldots, t_n$, each thread t_i waits for a resource $t_{(i+1) \bmod n}$ is holding.

A deadlock can be avoided by not fulling at least *one* of the conditions. When using the introduced synchronization mechanisms it is necessary to avoid *hold and wait* or *circular wait*

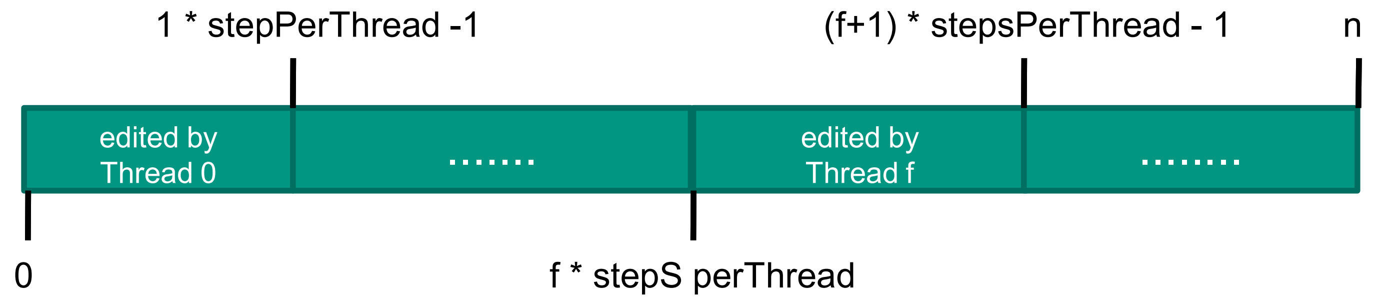
Example for a parallel programm with theads: Vector addition (1)

- We want to perform a parallel vector addition.
- The most important component is the worker class, which adds two (partial) vectors element by element and saves the result in a third vector.

```
class Worker implements Runnable {
  private int[] a, b, c;
  private int left, right;

  public worker (int[] a, int[] b, int[] c, int left, int right) {
    this.a = a; this.b = b; this.c = c;
    this.left = left; this.right = right;
  }

  public void run() { // adds subvectors in the segment [left..right)
    for (int i = left; i < right; i++) {
      c[i] = a[i] + b[i];
    }
  }
}
```

Example: Vector addition (2)

- **Illustration:** How parallel vector addition works
- Each thread `f` processes a segment of the length `stepsPerThread`.



Example: Vector addition (2)

- The main method first defines
  - The vectors,
  - constants to divide the index range of the vectors.

```
public static void main(String[] args) {

  int[] a = ... // fill
  int[] b = ... // fill

  assert a.length == b.length;
  int[] c = new int[a.length];

  final int numberOfThreads = 10;
  final int n = a.length;
  final int stepsPerThread = (int) Math.ceil((double) n / numberOfThreads);
```

Example: Vector addition (3)

- A field is created that contains references to the threads to be started.
- In the loop, the threads are each created and started with the index range that they are to process.
  - Explain what the minimum function is used for when calculating from the `right`.
  - Can `left >= right`? Is that a problem? What do the threads do for which this is the case?

```
Thread[] team = new Thread[numberOfThreads];

for (int f = 0; f < numberOfThreads; ++f) {
  int left = f * stepsPerThread;
  int right = Math.min((f + 1) * stepsPerThread, n);

  team[f] = new Thread(new worker(a, b, c, left,right));
  team[f].start(); //thread f is now running
}
```

Example: Vector addition (4)

- Now you have to wait for the threads to finish. This is done with `join()`
  - Note: `join()` can also be `interrupted` via `InterruptedException.`

```
for (Thread f : team) {
  try {
    f.join(); /* waits until thread ends */
  } catch (InterruptedException ex) {
    System.err.println("Unexpected interruption" +
                       "while waiting for workers");
  }
}

//Now use the result in c[]
```

Memory Consistency Errors

```
public class Main {
        static int flag = 0;
        public static void main(String[] args) {
                Thread t1 = new Thread(() -> flag = 1);
                Thread t2 = new Thread(() ->
                                System.out.println(flag));
                t1.start();
                t2.start();
        }
}
```

- In the above example, 0 can be printed for two reasons:
  - The statement of `t2` is simply executed before `t1`
  - The statement of `t2` is executed after `t1`, but sees an old state of `flag`
- The second reason is a memory consistency error, which can occur if different threads see different states of the same data
  - The reasons are complex and do not have to be fully understood
  - The most relevant reason is caching and instruction reordering
  - It is important to know the basic problems and avoidance strategies

Memory Consistency Error: Caching (1)

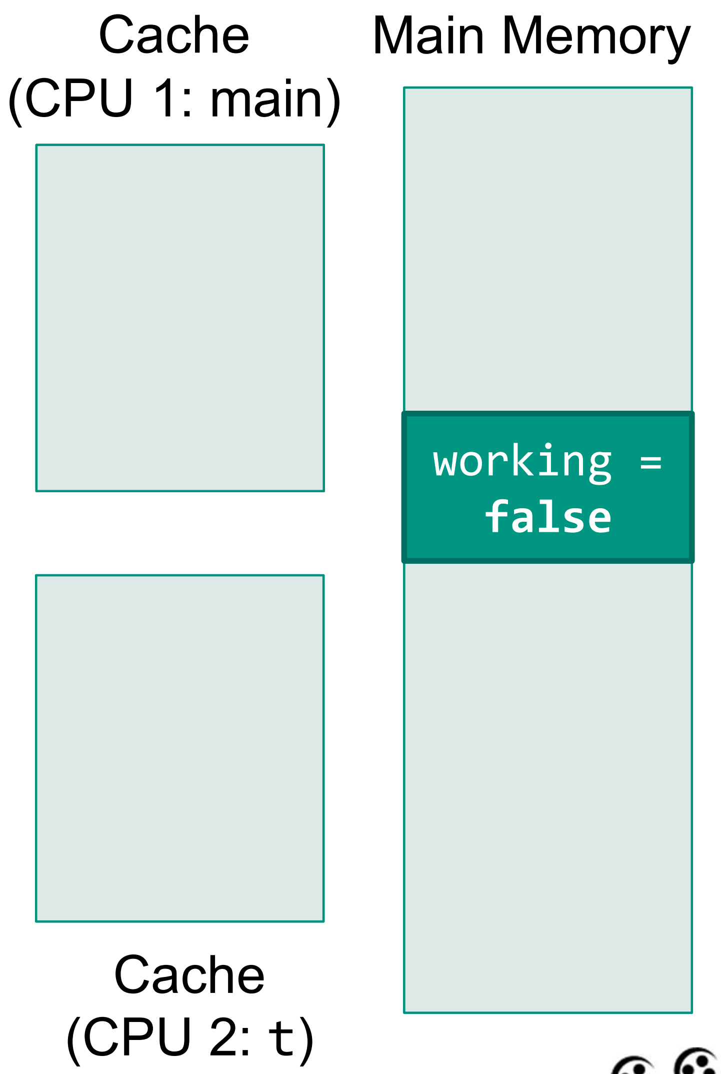
- Expected output:
  ```
  Work!
  Working!
  Stop working!
  Stopped
  ```
- Potential Output:
  ```
  Work!
  Stop working!
  ```
  Execution does not terminate
- Problem: Caching
  - `working` is cached in CPU running `t`
  - `working` is modified by main thread
  - `t` never sees the update of `working`

```java
public class Main {
  static boolean working = false;
  public static void main(String[] args) {
    Thread t = new Thread(() -> {
      while (!working) {}
      System.out.println("Working!");
      while (working) {/*do something*/}
      System.out.println("Stopped");
        });
    t.start();
    Thread.sleep(1000);
    System.out.println("Work!");
    working = true;

    Thread.sleep(1000);
    System.out.println("Stop working");
    working = false;
  }
}
```

Adapted from: https://www.callicoder.com/java-concurrency-issues-and-thread-synchronization/

Memory Consistency Error: Caching (2)

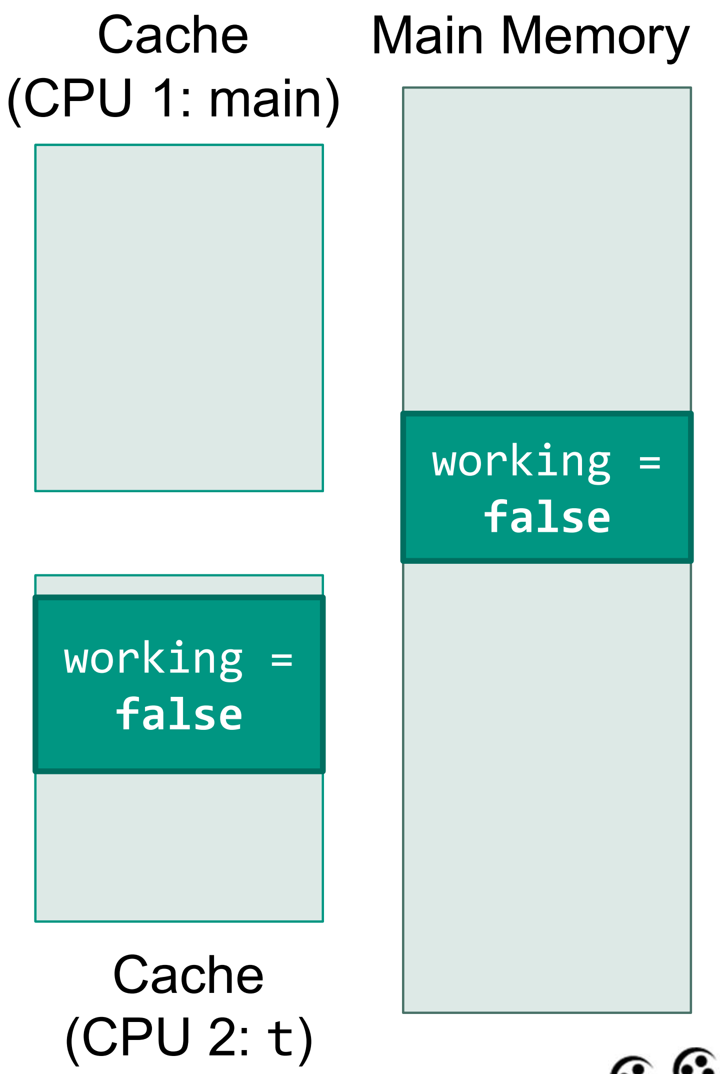
```
public class Main {
  static boolean working = false;
  public static void main(String[] args) {
    Thread t = new Thread(() -> {
      while (!working) {}
      System.out.println("Working!");
      while (working) {/*do something*/}
      System.out.println("Stopped");
        });
    t.start();
    Thread.sleep(1000);
    System.out.println("Work!");
    working = true;

    Thread.sleep(1000);
    System.out.println("Stop working");
    working = false;
  }
}
```



Adapted from: https://www.callicoder.com/java-concurrency-issues-and-thread-synchronization/

Memory Consistency Error: Caching (3)

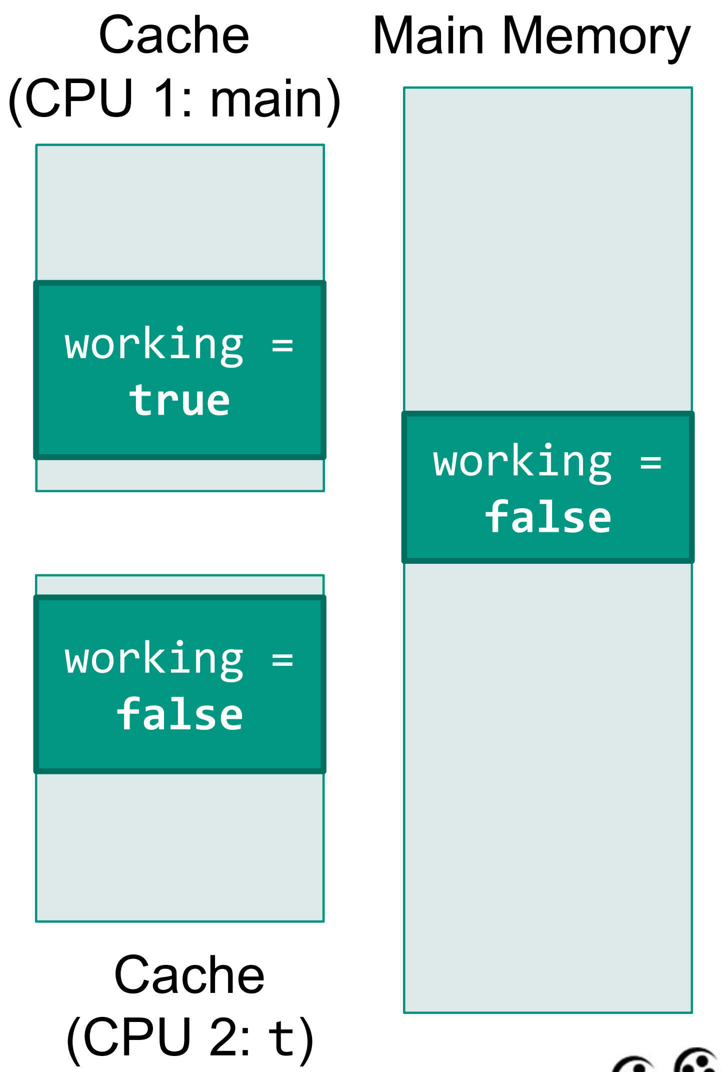
```
public class Main {
  static boolean working = false;
  public static void main(String[] args) {
    Thread t = new Thread(() -> {
      while (!working) {}
      System.out.println("Working!");
      while (working) {/*do something*/}
      System.out.println("Stopped");
        });
    t.start();
    Thread.sleep(1000);
    System.out.println("Work!");
    working = true;

    Thread.sleep(1000);
    System.out.println("Stop working");
    working = false;
  }
}
```



Adapted from: https://www.callicoder.com/java-concurrency-issues-and-thread-synchronization/

Memory Consistency Error: Caching (4)

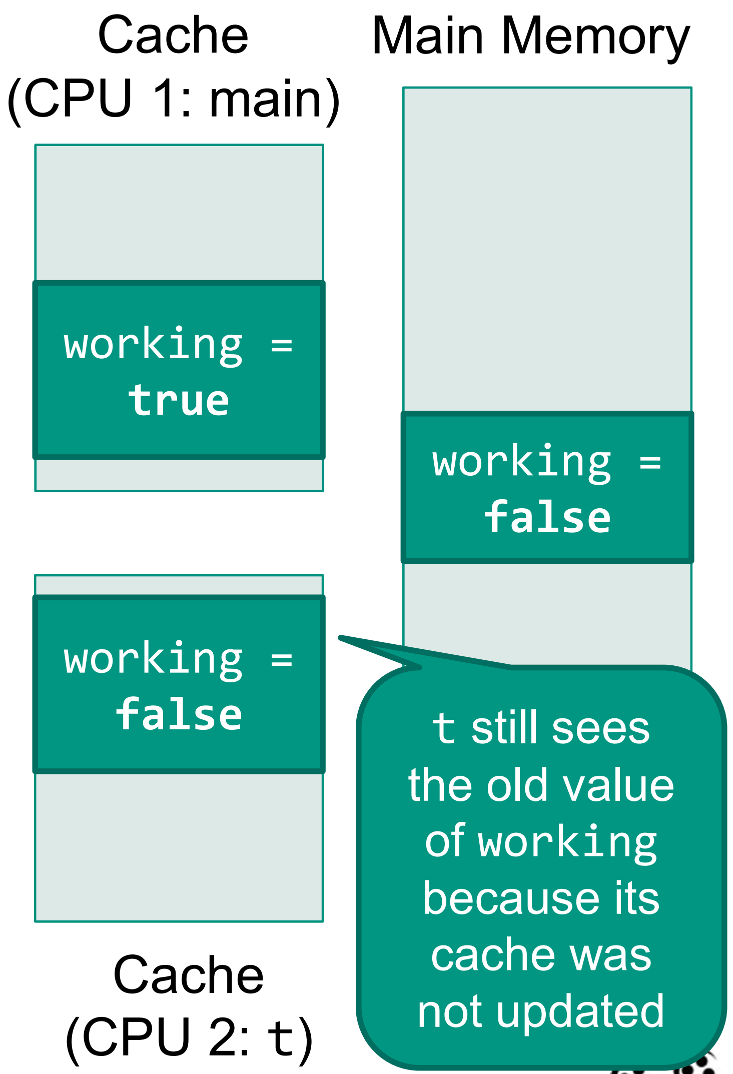
```
public class Main {
  static boolean working = false;
  public static void main(String[] args) {
    Thread t = new Thread(() -> {
      while (!working) {}
      System.out.println("Working!");
      while (working) {/*do something*/}
      System.out.println("Stopped");
        });
    t.start();
    Thread.sleep(1000);
    System.out.println("Work!");
    working = true;

    Thread.sleep(1000);
    System.out.println("Stop working");
    working = false;
  }
}
```



Adapted from: https://www.callicoder.com/java-concurrency-issues-and-thread-synchronization/

Memory Consistency Error: Caching (5)

```java
public class Main {
  static boolean working = false;
  public static void main(String[] args) {
    Thread t = new Thread(() -> {
      while (!working) {}
      System.out.println("Working!");
      while (working) {/*do something*/}
      System.out.println("Stopped");
        });
    t.start();
    Thread.sleep(1000);
    System.out.println("Work!");
    working = true;

    Thread.sleep(1000);
    System.out.println("Stop working");
    working = false;
  }
}
```

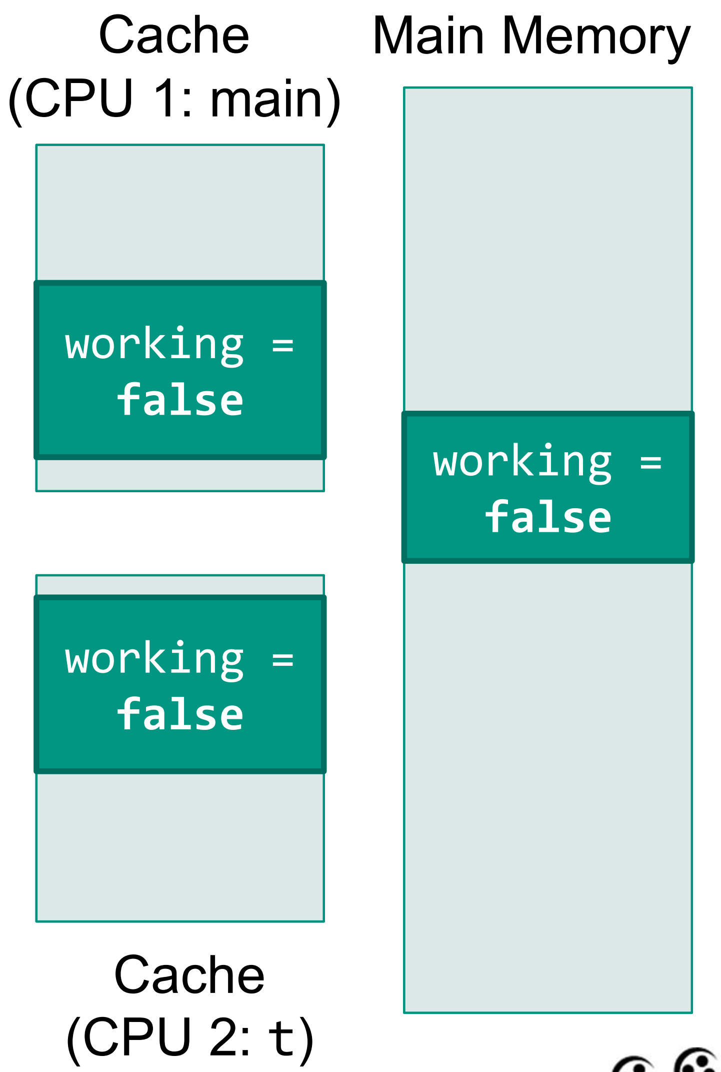


Adapted from: https://www.callicoder.com/java-concurrency-issues-and-thread-synchronization/

Memory Consistency Error: Caching (6)

```
public class Main {
  static boolean working = false;
  public static void main(String[] args) {
    Thread t = new Thread(() -> {
      while (!working) {}
       ystem.out.println("Working!");
         ile (working) {/*do something*/}
         stem.out.println("Stopped");
        });
    t.start();
    Thread.sleep(1000);
    System.out.println("Work!");
    working = true;

    Thread.sleep(1000);
    System.out.println("Stop working");
    working = false;
  }
}
```

t never terminates



Adapted from: https://www.callicoder.com/java-concurrency-issues-and-thread-synchronization/

Memory Consistency Error: Reordering (1)

```
public class Main {
public static void main(String[] args) {
  State state = new State();
  new Thread(() -> {
    state.a = 1;
    state.b = 1;
    state.c = state.a + 1;
  }).start();

  new Thread(() -> {
    if (state.c == 2 && state.b == 0)
{System.out.println("Wrong");}
  }).start();
}

public class State {
  int a = 0; int b = 0; int c = 0;
}
```

- Reading the code sequentially, `state.c` is written after `state.b`
- *Compiler Optimization:* Reordering of statements when the result of sequential intra-thread execution is not changed

Adapted from: https://stackoverflow.com/questions/52648800/how-to-demonstrate-java-instruction-reordering-problems

Memory Consistency Error: Reordering (2)

```java
public class Main {
public static void main(String[] args) {
  State state = new State();
  new Thread(() -> {
    state.a = 1;
    state.b = 1;
    state.c = state.a + 1;
  }).start();

  new Thread(() -> {
    if (state.c == 2 && state.b == 0)
{System.out.println("Wrong");}
  }).start();
}

public class State {
  int a = 0; int b = 0; int c = 0;
}
```

- Possible reordering (no change in local execution result):

```java
state.a = 1;
state.c = state.a + 1;
state.b = 1;
```

Consequence:
Second thread can print "Wrong" if it is executed before `state.b = 1;`

Adapted from: https://stackoverflow.com/questions/52648800/how-to-demonstrate-java-instruction-reordering-problems

Happens-before Relationship (1)

```
public class Main {
        static int flag = 0;
        public static void main(String[] args) {
                Thread t1 = new Thread(() -> flag = 1);
                Thread t2 = new Thread(() -> System.out.println(flag));
                t1.start();
                t2.start();
        }
}
```

- Java defines the happens-before relationship to avoid such errors
  - If two statements *s1* and *s2* have a happens-before relationship, Java guarantees that a potential write in *s1* is visible to *s2*
  - The happens-before relation is transitive, so if *s1* happens before *s2* and *s2* before *s3*, there is also a happens-before relation between *s1* and *s3*
  - In the above example, there is no happens-before relationship between the statements in `t1` and `t2`

Happens-before Relationship (2)

- Each statement has a happens-before relationship to every statement in the thread that comes later in the program's order.
- There are several statements that introduce additional happens-before relationship:
  - `Thread.start`: all statements executed before starting the thread (within the starting thread) have a happens-before relationship to statements in the new thread
  - `Thread.join`: all statements in the terminated thread have a happens-before relationship to the statements following the join
  - `synchronized`: all statements in a synchronized block have a happens-before relationship to all statements in a subsequent execution of a synchronized block using the same monitor
  - We will see further statements in the following
- **Again**: If there is no happens-before relationship between two statements, the second one may not see the result of the first one!

The volatile-Keyword

- `volatile` ensures that changes to variables are immediately visible to all threads / processors
  - `volatile` establishes a happens-before relationship: a write to a volatile variable happens-before every subsequent read to that variable
  - **This means that all writes to (potentially different) variables before writing a volatile variable are visible to all reads of that variables after reading the volatile variable, because statements within a thread have a happens-before relationship in their program order**
  - Values are not locally cached in a CPU cache (every read/write directly back to main memory)
  - Compiler/Processor optimizations are disabled: instruction reordering is not possible for the volatile variable

volatile Example: Caching

- Avoid memory inconsistency due to caching:
  - Declare variable `working` as `volatile`
  - `working` is always written to and read from main memory

  →Output as expected

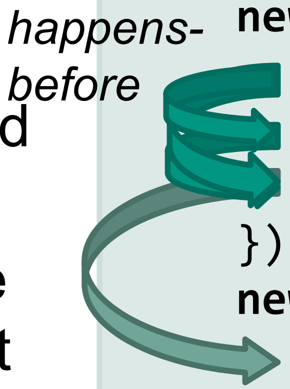
  →Program terminates

```
public class Main {
  static volatile boolean working = false;

  public static void main(String[] args) {
    Thread t = new Thread(() -> {
      while (!working) {}
      System.out.println("Working!");
      while (working) {/*do something*/}
      System.out.println("Stopped");
    });
    t.start();
    Thread.sleep(1000);
    System.out.println("Work!")
    working = true;

    Thread.sleep(1000);
    System.out.println("Stop Working")
    working = false;
  }
}
```

volatile Example: Reordering

- Declaring variable `state.c` as volatile induces a happens-before relation between its write and its read
- The happens-before relation ensures that all writes before writing `state.c` are seen by all reads after reading `state.c` (transitive)

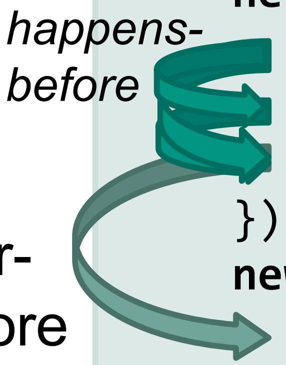
  → Printing “Wrong” impossible



```
public class Main {
  public static void main(String[] args) {
    State state = new State();
    new Thread(() -> {
      state.a = 1;
      state.b = 1;
      state.c = state.a + 1;
    }).start();
    new Thread(() -> {
      if(state.c == 2 && state.b == 0) {
        System.out.println("Wrong");
      }
    }).start();
  }
}

public class state {
        int a = 0; int b = 0;
        volatile int c = 0;
}
```

Happens-before $\not\Rightarrow$ happens before

- A happens-before relation does *not* mean that one statement is actually executed before another
- No reordering for statements with inter-thread happens-before relations (e.g. writes to `volatile` variables), but possible reordering for other statements **if behavior is not changed**
- E.g.: Writes to `state.a` and `state.b` can still be reordered

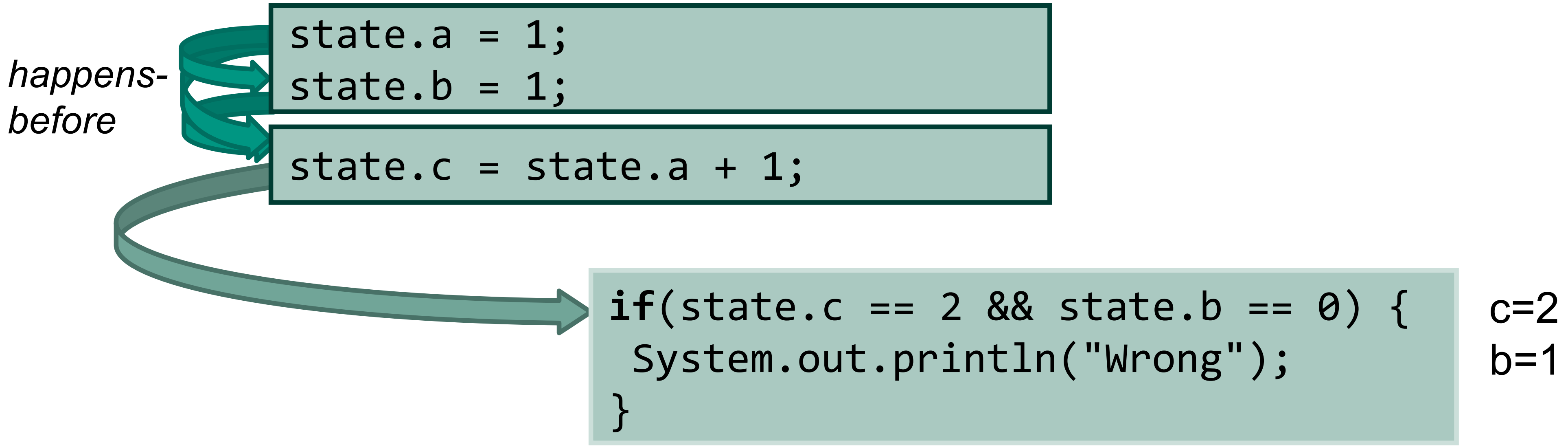


```java
public class Main {
  public static void main(String[] args) {
    State state = new State();
    new Thread(() -> {
      state.a = 1;
      state.b = 1;
      state.c = state.a + 1;
    }).start();
    new Thread(() -> {
      if(state.c == 2 && state.b == 0) {
        System.out.println("Wrong");
      }
    }).start();
  }
}

public class state {
        int a = 0; int b = 0;
        volatile int c = 0;
}
```

can be reordered

`volatile` Example: Reordering: Possible Execution Orders

Thread 1 before thread 2:



```
state.a = 1;
state.b = 1;
```

```
state.c = state.a + 1;
```

```
if(state.c == 2 && state.b == 0) {
 System.out.println("Wrong");
}
```

c=2
b=1

`volatile` Example: Reordering: Possible Execution Orders

Thread 2 before thread 1:

```java
if(state.c == 2 && state.b == 0) {
 System.out.println("Wrong");
}
```

c=0
b=0

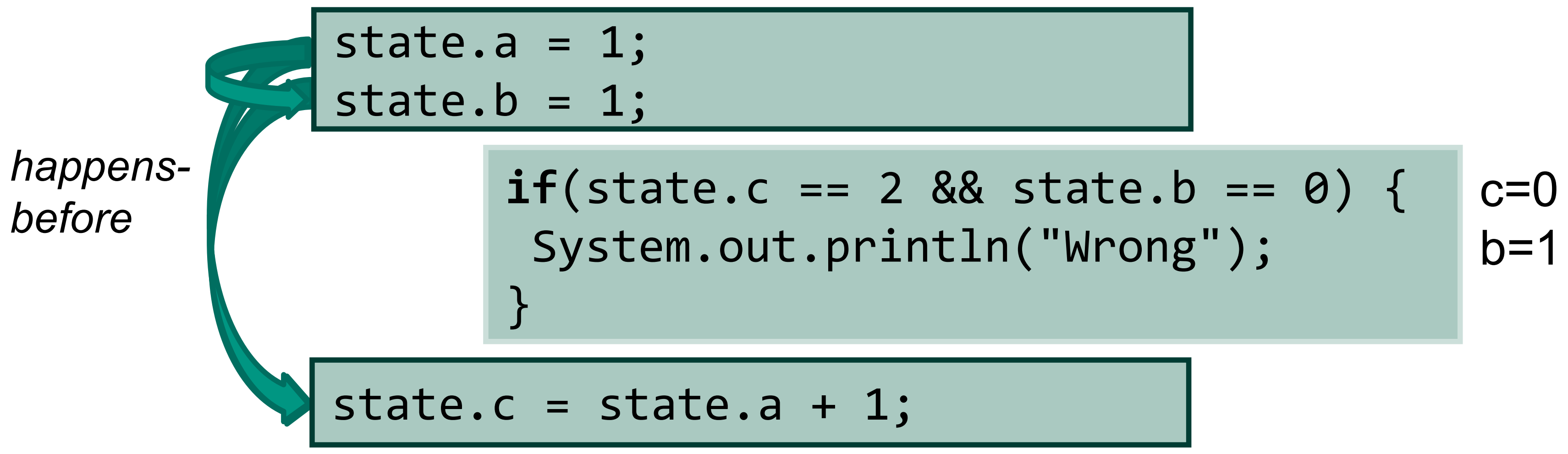
happens-before

```java
state.a = 1;
state.b = 1;
```

```java
state.c = state.a + 1;
```

`volatile` Example: Reordering: Possible Execution Orders

Thread 1 and thread 2 interleaved:



```
state.a = 1;
state.b = 1;
```

```
if(state.c == 2 && state.b == 0) {
 System.out.println("Wrong");
}
```

c=0
b=1

```
state.c = state.a + 1;
```

Volatile: Any write to c can only be read when the writes that happened before are also executed

Summary: Happens-before Relations

- Visibility of variable modifications only guaranteed within one thread
- When using multiple threads, memory consistency errors can arise
  - Caching of values in CPU caches (invisibility of changes for other threads)
  - Optimization techniques reorder statements in the execution of one thread
- Problems can be avoided by considering happens-before relationships
  - Only statements having a happens-before relation ensure that one statement sees the result of the other
  - When no happens-before relation exists, old variable values may be used
  - Missing happens-before relations are often *not* obvious
- The program order within a thread induces a happens-before relation
- Several statements introduce happens-before relationship between threads, some are:
  - `Thread.start()`, `Thread.join()`
  - `synchronized`
  - `volatile`

Conclusion

- Basic thread programming in Java
  - `Threads` and `Runnables` for simple threading
  - Threads can be started, stopped, joined, interrupted
- Synchronization
  - Critical sections to avoid race conditions
  - Monitors with reentrance using the `synchronized` keyword
  - Happens-before relationship and keyword volatile

Evaluation