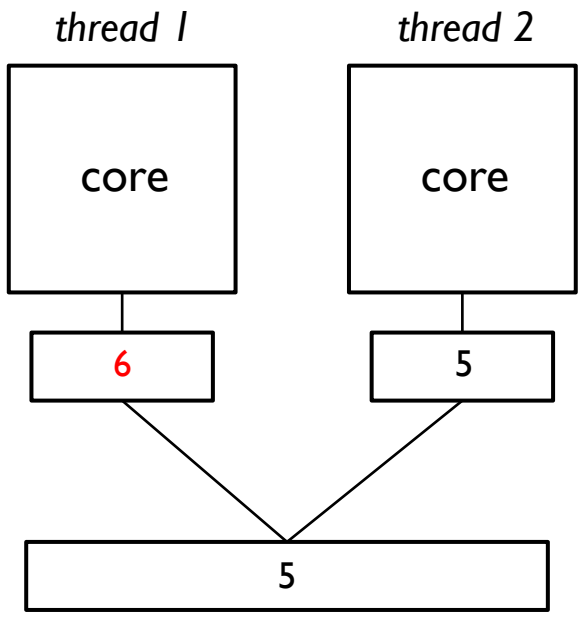
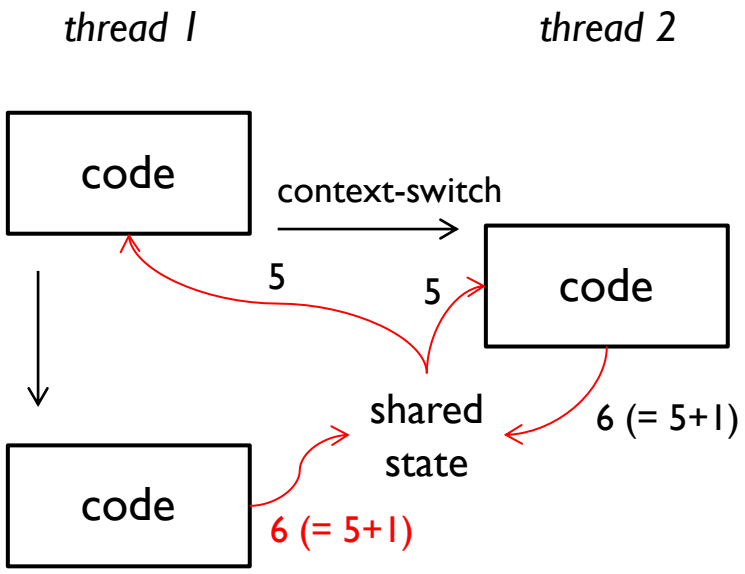
Thread Safety (Part Three)

Concurrent and Parallel Programming

Visibility vs. atomicity



Visibility problem



Atomicity problem

Memory model

- In Java, everything that concerns memory management and synchronization is described in the memory model: an abstraction that allows to provide thread-safety tools, that are independent from the peculiarities of processors and operating systems.
- The Java memory model is described in the Java language specification.

Memory model

- Given a program and a possible execution of that program, the memory model defines whether the execution is a legal one.
- In addition, the memory model defines which are the values that can be read from memory in any instant during program execution.
- Given some Java source code, the compiler is free to generate the sequence of operations it prefers, provided that all its executions are legal and predictable from the memory model's point of view.

Out-of-thin-air safety

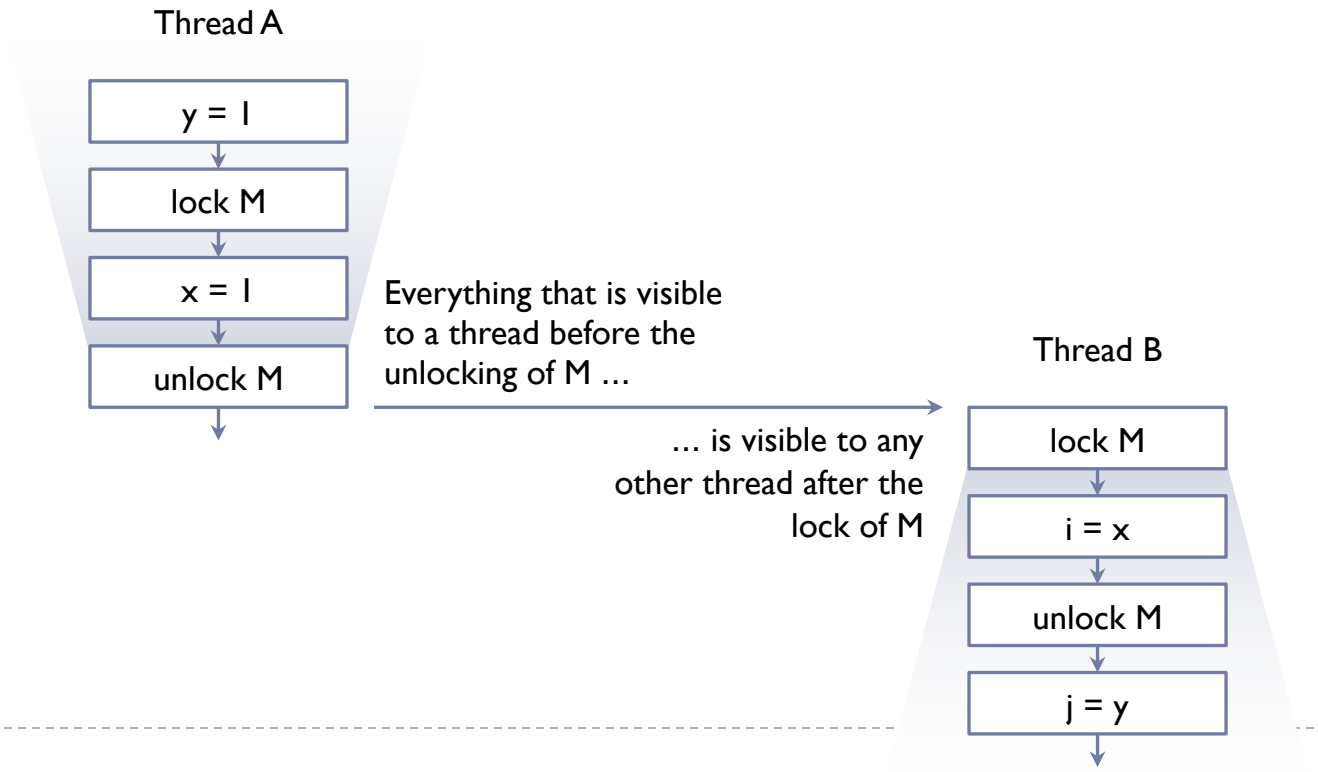
- Java's memory model guarantees the atomicity of fetch and store operations, except for long and double variables:
  - if a thread accesses a variable without synchronization, it is guaranteed that at least one old value is seen. This guarantee is called out-of-thin-air safety.
- But this is definitely not sufficient! Out-of-thin-air safety neither solves the visibility problem, nor the atomicity problem for long and double variables and for all compound actions.

Mutexes and visibility

- As previously introduced, mutexes (both implicit and explicit) can be used to obtain correct memory visibility.
- To ensure that all threads see the latest modifications of all other threads on shared and mutable variables, all threads (either performing read or write operations) must synchronize on a common lock.

Mutexes and visibility

- The visibility effect of the lock works in the following way: anything visible to a thread before the lock is released, will be correctly visible to any thread using the same lock, from the moment the lock is acquired.



Example

Visibility?!

```
class FibonacciNumber {
    private int previousValue = 1;
    private int currentValue = 1;

    public int getPreviousValue() {
        return previousValue;
    }

    public int getCurrentValue() {
        return currentValue;
    }

    public void setNewValue(final int value) {
        previousValue = currentValue;
        currentValue = value;
    }
}
```

Example

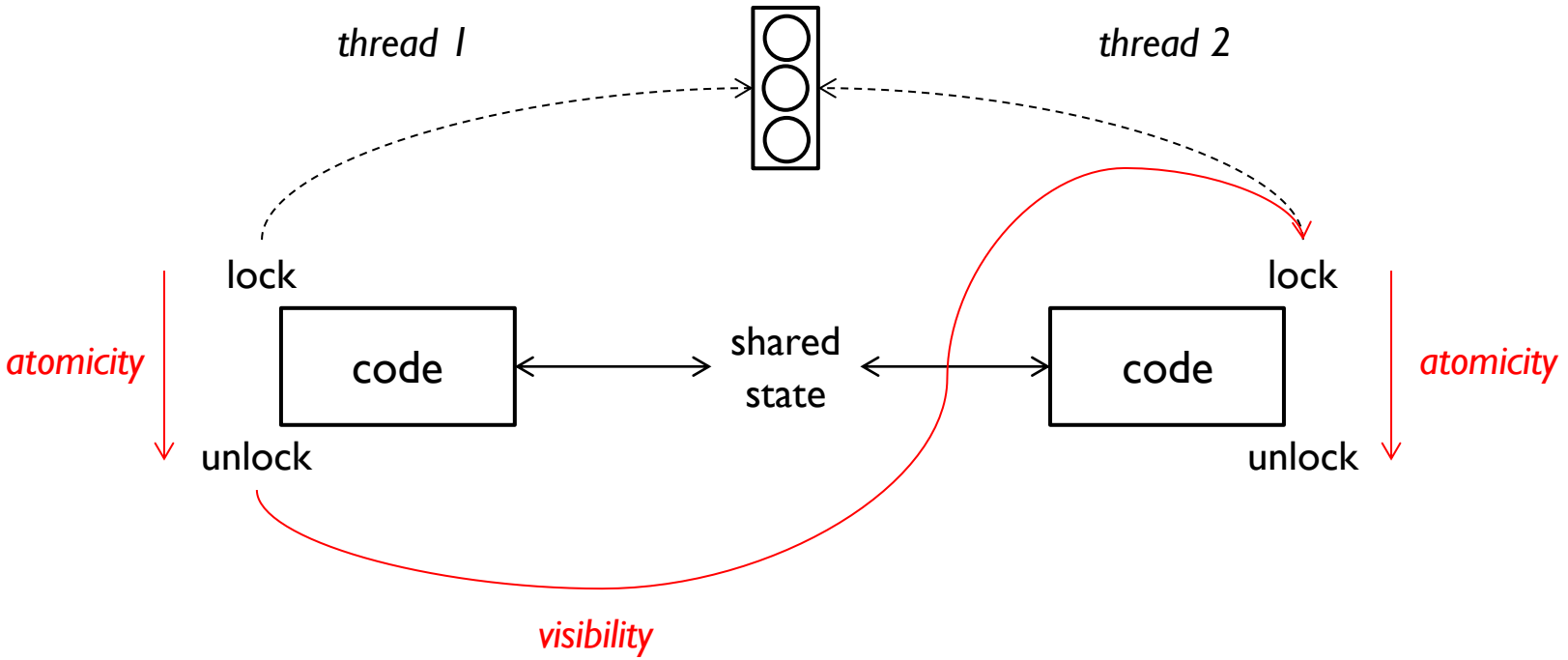
```
class FibonacciCalculator implements Runnable {
    private static int count = 2;
    private FibonacciNumber number = null;

    public FibonacciCalculator(FibonacciNumber number) {
        this.number = number;
    }

    @Override
    public void run() {
        synchronized (number) {
            count++;
            number.setNewValue(number.getPreviousValue()
                               + number.getCurrentValue());
            System.out.println("The " + count + " Fibonacci number is "
                               + number.getCurrentValue());
        }
    }
}
```

Visibility?!

All visibility problems on all variables are solved by using the same lock by all threads!

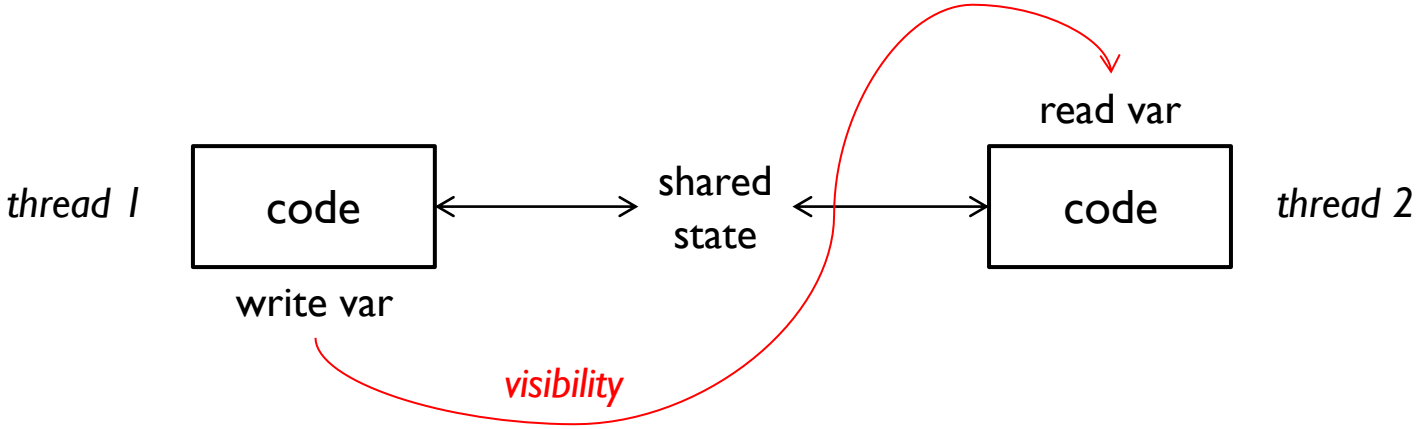
Visibility vs. atomicity

- *Important: the visibility effect of the lock applies differently that the atomicity effect!*



Volatile/atomic vars and visibility

- The visibility effect of volatile and atomic variables extends beyond the volatile/atomic variable itself. If a thread A writes a volatile/atomic variable, which is then read from thread B, the value of all variables visible to A before writing the volatile/atomic variable, becomes correctly visible to B after reading the volatile/atomic variable.
- This behaviour is similar as the one of the lock:



Example

```
public class TestVolatileVisibility extends Thread {
    private int a;
    private int b;
    private volatile int c;

    @Override
    public void run() {
        while (c == 0); // Do nothing
        System.out.println("Thread ended. " + a + ", " + b + ", " + c);
    }

    public static void main(String[] args) throws InterruptedException {
        System.out.println("Program started.");
        TestVolatileVisibility t = new TestVolatileVisibility();
        t.start();
        Thread.sleep(1000);
        t.a = 10;
        t.b = 20;
        t.c = 30;
        System.out.println("Program ended.");
    }
}
```

Output:Program started.
Program ended.
Thread ended. 10, 20, 30

Object construction and visibility

- Important: Objects are built by threads. As a consequence, the values assigned during the construction phase of an object can be subject to visibility problems too.
- Consequently, synchronization must be used also when sharing newly created objects.
- We will discuss this issue later in the semester, when the technique of safe publication of objects will be introduced.

Lock = pessimistic approach

- One of the weaknesses of locks (both implicit and explicit) is that they are based on a pessimistic approach:
  - Threads that fail to acquire a lock are suspended (except for spin-locks, which are not the subject of this lesson). Therefore the assigned scheduling period is not fully consumed.
  - All waiting threads have to wait for the thread owning the lock to complete, regardless of the operations it is performing (including the context-switches it might be subject of).
  - At unlock time, pending threads must frequently wait for rescheduling (with consequent context-switch).

Optimistic locking

- It is possible to profit from an alternative approach: optimistic locking based on Compare and Set (CAS) instructions.
- With CAS instructions it is possible to perform conditional updates in an atomic way.
- A possible pseudo-code of what a CAS instruction does can be sketched as follows:

```
boolean compareAndSet(int expect, int update) {
    if (curval == expect) {
        curval = update;
        return true;
    }
    return false;
}
```

In reality, is one atomic operation!

CAS and Atomic variables

- CAS instructions are provided by Java's atomic variables.
- For example, the AtomicInteger class provides:

Modifier and Type	Method and Description
int	**accumulateAndGet**(int x, **IntBinaryOperator** accumulatorFunction) Atomically updates the current value with the results of applying the given function to the current and given values, returning the updated value.
int	**addAndGet**(int delta) Atomically adds the given value to the current value.
boolean	**compareAndSet**(int expect, int update) Atomically sets the value to the given updated value if the current value == the expected value.
int	**decrementAndGet**() Atomically decrements by one the current value.
double	**doubleValue**() Returns the value of this AtomicInteger as a double after a widening primitive conversion.

Idiom for CAS synchronization

- By taking advantage of CAS operations it is possible to implement a specific idiom for synchronization that results in the optimistic locking approach.
- All threads have to work on the shared data as follows:

Read the shared data

Perform any modification

```
// ...

do {
    oldValue = atomicVar.get();
    newValue = computeNewValue(oldValue);
} while(!atomicVar.compareAndSet(oldValue, newValue));

// ...
```

If there is no contention, then write back

Example

```
public class RandomGeneratorCAS {
    private static final int BASE_RND_SEED = 1;
    private static final int BASE_RND_CONST = 32767;
    private static final int BASE_RND_BASE = 1664525;

    private AtomicInteger uiRndSeed = new AtomicInteger(BASE_RND_SEED);

    public int generate() {
        int oldValue, newValue;
        do {
            oldValue = uiRndSeed.get();
            newValue = oldValue * BASE_RND_BASE;
            newValue = newValue + BASE_RND_CONST;
        } while (!uiRndSeed.compareAndSet(oldValue, newValue));
        return newValue;
    }
}
```

Optimistic Locking

- Optimistic locking focus on the notion of transaction (similar to what is done by DBs): operations that are not able to commit, have to be repeated.

- With optimistic locking:
  - Threads are never suspended (there's no lock).
  - Threads run simultaneously in the protected code region. Threads that are not able to arrive at the synchronization point for first, are forced to repeat the operation.
  - Compared to traditional locks, overheads might be lower (context-switches are not forced), concurrency might be higher, but starvation problems might appear (we'll talk about them ...).

Optimistic Locking in Java 8

- The atomic variables of Java 8 have been extended with methods that support optimistic locking by means of lambda expressions.

```
public long getAndUpdate(LongUnaryOperator updateFunction) {
  long prev, next;
  do {
    prev = atomicLong.get();
    next = updateFunction.applyAsLong(prev);
  } while (!atomicLong.compareAndSet(prev, next));
  return prev;
}
```

```
public long getAndAccumulate(long x, LongBinaryOperator accumulatorFunction) {
  long prev, next;
  do {
    prev = atomicLong.get();
    next = accumulatorFunction.applyAsLong(prev, x);
  } while (!atomicLong.compareAndSet(prev, next));
  return prev;
}
```

Synchronization mechanisms

- To summarize, the main synchronization mechanisms in Java include:

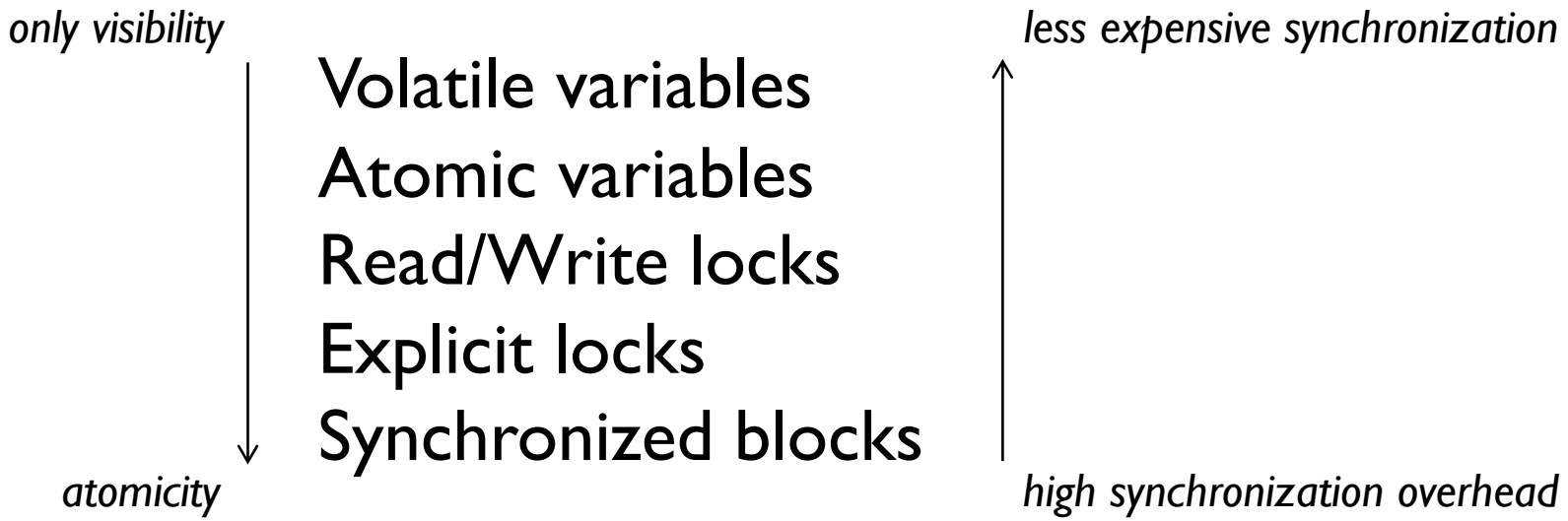
  Already discussed:

  - volatile variables
  - atomic variables (including the optimistic locking approach)
  - explicit locks (Reentrant-Lock and RW-Lock)
  - "synchronized" keyword

  To be discussed in future lectures:

  - concurrent collections
  - executors

Protection and overhead



- Important: even if volatile/atomic variables have a lower synchronization cost, it does not necessarily imply better overall performances.
- For example: an explicit lock might be more effective than a volatile variable, if the lock is rarely contended and the data frequently found in registers and caches!

ThreadLocalRandom

- Java 7 has seen the introduction of a new random number generator for multi-threaded programs.
- Traditional approaches for random number generation (Math class or Random class) are thread-safe, but prove inefficient when used by more than one thread simultaneously.
- The ThreadLocalRandom class features better multi-thread performances.

Example

```
import java.util.Random;
import java.util.concurrent.ThreadLocalRandom;

public class ThreadLocalRandomExample {
    public static void main(String[] args) {
        // Generate a random number b/w 0 and 10. 0 <= R < 10
        // Using Math.random()
        int r1 = (int) (Math.random() * 10);
        // Using Random
        Random rand = new Random();
        int r2 = rand.nextInt(10);
        // Using ThreadLocalRandom
        int r3 = ThreadLocalRandom.current().nextInt(10);
    }
}
```

Race-condition details

- Now that we know about visibility problems, we can now complete our discussion on race conditions with some additional information.
- Let's analyze the standard definition:

> A race-condition is present when two threads are allowed to access (read or write) a variable simultaneously and at least one of the accesses is a write.

- ... this definition is somehow unclear (and in my opinion is the definition of a visibility problem) ...

Race-condition details

- A better definition:

  A race-condition is present when a program reads the value of a variable/field and needs then to perform an operation that depends on that value (e.g. check-then-act or read-modify-write).

- If another thread in the meantime is able to modify the value of the variable in a way that the operation would result incorrect, then the program presents a race-condition.

Example

```
class Worker1 implements Runnable {
    public void run() {
        System.out.println("Checks that the element has been shared");
        if (Worker2.element != null) {
            // simulates other potential operations
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
            }
            Worker2.element.performOperation();
        }
    }
}
```

```
class Worker2 implements Runnable {
    public static volatile Element element = null;

    public void run() {
        System.out.println("Shares the element");
        element = new Element();
        // simulates other potential operations
        try {
            Thread.sleep(1);
        } catch (InterruptedException e) {
        }
        element = null;
    }
}
```

Race-condition and errors

- Race conditions are non-deterministic defects.
- If a race-condition really causes an execution error, depends on the interleaving of the instructions executed by the threads during a given program run.
- The consequences of a race condition can be unpredictable. Frequently the program does not crash immediately. Failure might occur later, for example when the wrong value is used.

They're not all race-condition

- Important: not all asynchronous executions on shared data are race-conditions, ...
- Asynchronous executions are a good thing in concurrent programs!
- A correctly programmed asynchronous execution is able to tolerate any form of interleaving in which the operations might execute.
- For example, it could be required to tolerate an extra loop iteration, if the timing between the threads is not ideal.

Example

```
class WorkerA implements Runnable {
    public void run() {
        System.out.println("Thread ready to start");
        int count = 0;
        boolean complete = WorkerB.finish;
        while (!complete) {
            System.out.println("Waiting for the other thread");
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            count++;
            if (count >= 5)
                break;
            else
                complete = WorkerB.finish;
        }
    }
}
```

```
class WorkerB implements Runnable {
    public static volatile boolean finish = false;

    public void run() {
        System.out.println("Thread ready to start");
        try {
            Thread.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        finish = true;
    }
}
```

Benign race-conditions

- In addition, there are race-conditions that don't have any real consequence, so-called benign race-conditions, that might be tolerated, in particular for performance improvement.
- In a benign race-condition, even if the code is written the wrong way, the program doesn't crash.
- Even if these types of race conditions are benign, it is difficult to control them. Therefore should be avoided!

Example

Loop that iterates on a list of elements and adds them to a thread-safe Set, if the `GROUP_ATTRIBUTE` attribute correctly matches. The `collectFlag` (initialized to false) is used as an optimization to avoid performing the match operation more than once.

```
for (Element element : elements) {
    if (element.collectFlag == true)
        continue;
    if (element.matchAttribute(GROUP_ATTRIBUTE)) {
        element.collectFlag = true;
        group.add(element);
    }
}
```

The Set filters out any double addition

benign race-condition, it's tolerable, but beware of the risk of visibility problems

But so what?

- So when is it required to use synchronization?

  If there is a shared and mutable state, ALWAYS!

  For EVERY READ and EVERY WRITE operation!

- To protect from visibility problems, at least volatile has to be used.
- Depending on the type of compound action and the related potential race-condition, atomic variables, mutexes (implicit or explicit) or other types of synchronization tools should be used.

Summary of topics

- Visibility problems vs. atomicity problems
- Java memory model
- Visibility effect of mutexes and volatile/atomic variables
- CAS and optimistic locking
- General considerations on synchronization
- Additional details about race-conditions