Question

```java
private static int i = 0;
private static int j = 0;

public static void write() {
  i++; j++;
}
public static void read() {
  System.out.println("i=" + i + " j=" + j);
}
```

One thread repeatedly invokes `write` whereas another thread repeatedly invokes `read`. When executing `read`, can it ever be the case that the value of `j` is greater than the value of `i`?

Answer

Yes.

Question

```
private static int i = 0;
private static int j = 0;

public static synchronized void write() {
  i++; j++;
}
public static synchronized void read() {
  System.out.println("i=" + i + " j=" + j);
}
```

One thread repeatedly invokes `write` whereas another thread repeatedly invokes `read`. When executing `read`, can it ever be the case that the value of `j` is greater than the value of `i`?

Answer

No.

Static synchronized methods

With each class `C` is associated an object `C.class`.

To execute a static synchronized method of class `C`, first the lock associated with the object `C.class` has to be obtained.

Question

```
private static volatile int i = 0;
private static volatile int j = 0;

public static void write() {
  i++; j++;
}
public static void read() {
  System.out.println("i=" + i + " j=" + j);
}
```

One thread repeatedly invokes `write` whereas another thread repeatedly invokes `read`. When executing `read`, can it ever be the case that the value of `j` is greater than the value of `i`?

Answer

No.

Volatile

An attribute may be declared volatile, in which case the Java memory model ensures that all threads see a consistent value for the attribute.

We will come back to the Java memory model later in the course.

Final

Question

When should you declare an attribute final?

Answer

Whenever you can.

When the constructor exits, the values of final attributes are guaranteed to be visible to other threads accessing the constructed object.

Readers-writers problem

```
public class Database
{
  private int activeReaders;
  private boolean writing;

  public Database() {
    this.activeReaders = 0;
    this.writing = false;
  }

  public void read() {

  }

  public void write() {

  }
```

Synchronized blocks

```
synchronized(o) {
  ...
}
```

Before executing the block of code, the lock of the object `o` needs to be acquired.

Producer-consumer problem

```
public class BoundedBuffer<T> {
  private final Object[] content;
  private int size;
  private int next;

  public BoundedBuffer(int capacity) {
    this.content = new Object[capacity];
    this.size = 0;
    this.next = 0;
  }
```

Producer-consumer problem

```
public synchronized void put(T value) {
  this.content[this.next] = value;
  this.size++;
  this.next =
    (this.next + 1) % this.content.length;
}

public synchronized T get() {
  int index =
    (this.next - this.size) % this.content.length;
  T value = (T) this.content[index];
  this.size--;
  return value;
}
```

Concurrent Object Oriented Languages

java.util.concurrent.locks

https://wiki.cse.yorku.ca/course/6490A

java.util.concurrent.locks

The package java.util.concurrent.locks contains the iterfaces

- Condition
- Lock
- ReadWriteLock

Lock

The interface Lock is implemented by the classes

- ReentrantLock
- ReentrantReadWriteLock.ReadLock
- ReentrantReadWriteLock.WriteLock

It provides more flexibility than synchronized methods and synchronized blocks.

Lock

The Lock interface contains the methods

- lock(): acquire this lock
- unlock(): release this lock
- newCondition(): returns a condition variable bound this lock

Lock chaining

```
Node parent = null;
Node node = this.getRoot();
node.lock()
while (!node.isLeaf())
{
   parent = node;
   node = node.getLeft();
   node.lock();
   parent.unlock();
}
node.unlock();
```

Locks and Exceptions

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

Condition

The Condition interface contains the methods

- await(): causes the current thread to wait on this condition
- signal(): wakes up one thread waiting on this condition
- signalAll(): wakes up all threads waiting on this condition

Condition

The interface Condition is implemented by the classes

- AbstractQueuedLongSynchronizer.ConditionObject
- AbstractQueuedSynchronizer.ConditionObject

The producer-consumer problem

Problem

Implement the class BoundedBuffer and its methods put and get using Locks and Conditions.

ReadWriteLock

The interface ReadWriteLock contains the methods

- readLock(): the lock used for reading
- writeLock(): the lock used for writing

ReadWriteLock

The interface ReadWriteLock is implemented by the class ReentrantReadWriteLock.

The readers-writers problem

Problem

Implement the class Database and its methods read and write using ReadWriteLocks.