Monitors

CS511

Review

- We've seen that semaphores are an efficient tool to solve synchronization problems
- However, they have some drawbacks
  1. They are low-level constructs
     - It is easy to forget an acquire or release
  2. They are not related to the data
     - They can appear in any part of the code

Monitors

- Combines ADTs and mutual exclusion
  - Proposed by Tony Hoare:

  Monitors: An Operating System Structuring Concept (Communications of the ACM, 17(10), 549-557, 1974).

- Adopted in many modern PLs
  - Java
  - C#
  - Python
  - Ruby

Main Ingredients

- A set of operations encapsulated in modules
- A unique lock that ensures mutual exclusion to all operations in the monitor
- Special variables called condition variables, that are used to program conditional synchronization

Counter Example

- Construct a counter with two operations:
  - inc()
  - dec()
- No two threads should be able to simultaneously modify the value of the counter
  - Think of a solution using semaphores
  - A solution using monitors

Counter using Semaphores

```
class Counter {
  private int c = 0;
  private Semaphore mutex = new Semaphore(1);

  public void inc() {
      mutex.acquire();
      c++;
      mutex.release();
  }
  public void dec() {
      mutex.acquire();
      c--;
      mutex.release();
  }
}
```

Note: The shared variable `c` need not be declared `volatile` when using `synchronized`[1]

[1] https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html

Counter using Monitors

```
class Counter {

  private int counter = 0;

  public synchronized void inc() {
    counter++;
  }

  public synchronized void dec() {
    counter--;
  }

}
```

- Each object has its own lock called intrinsic or monitor lock
- It also has its own wait-set for this lock (more on this later)
- This code is both Groovy and Java code

Counter as Monitor in Groovy – Continued

```
Counter c = new Counter()

P = Thread.start {
    10.times {
        c.inc()
    }
}

Q = Thread.start {
    10.times {
        c.inc()
    }
}

P.join()
Q.join()
println c.counter
```

Condition Variables

- Apart from the lock, there are condition variables associated to the monitor
  - Built-in: called a wait set and associated to the intrinsic lock
  - User-declared
- They have
  1. Three operations:
     - `Cond.wait()/await()`
     - `Cond.notify()/signal()`
     - `Cond.notifyAll()/signalAll()`
  2. A set of blocked processes.

Condition Variables

`Cond.wait()/await()`

- Always blocks the process and places it in the waiting set of the variable `Cond`.
- When it blocks, it releases the mutex on the monitor.

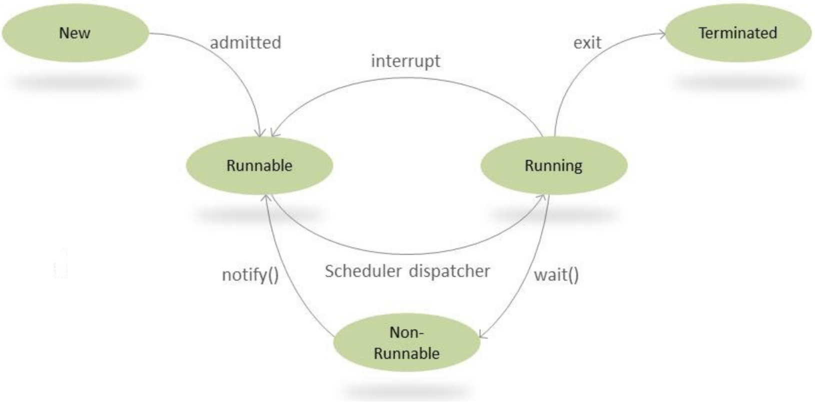
`Cond.notify()/signal()`

- Unblocks the first process in the waiting set of the variable `Cond` and sets it to the RUNNABLE state
- If there are no processes in the waiting set, it has no effect.

`Cond.notifyAll()/signalAll()`

- Unblocks all the processes in the waiting set of the variable `Cond` and sets them to the RUNNABLE state
- If there are no processes in the waiting set, it has no effect.

Condition Variables and Process States[2]



[2]Source: https://www.baeldung.com/java-wait-notify

Example: Buffer of Size 1 in Groovy

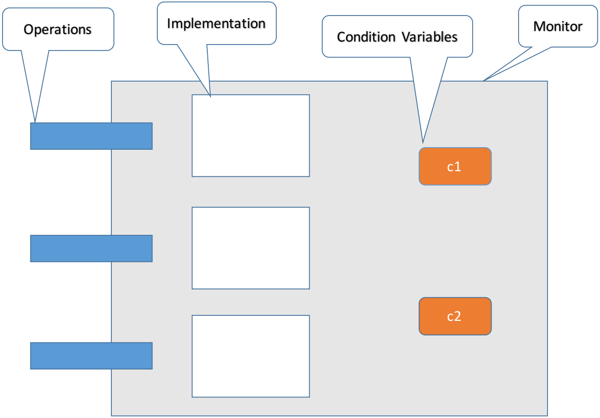
```
class Buffer {
  private Object buffer = null; // shared buffer

  synchronized Object consume() {
    while (buffer == null)
      wait(); // wait on object's wait-set
    Object aux = buffer;
    buffer = null;
    notifyAll(); // signal on object's wait-set
    return aux;
  }

  synchronized void produce(Object o) {
   while (buffer != null)
      wait(); // wait on object's wait-set
    buffer = o;
    notifyAll(); // signal on object's wait-set
  }
}
```

- `wait`, `notify` and `notifyAll` must be called from synchronized methods or else an `IllegalMonitorStateException` is raised

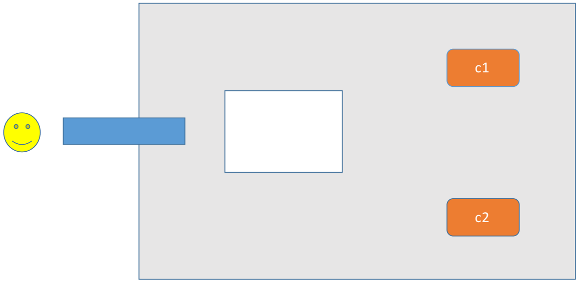
Example: Buffer of Size 1 in Groovy (cont)

```
Buffer b =new Buffer()

20.times {
   int id=it
   Thread.start {
     println (id+": consumer "+ b.consume())
   }
}

20.times {
    int id=it
    Thread.start {
      b.produce(id)
      println (id+": producer ")
  }
}
```

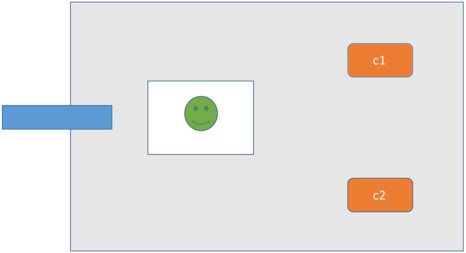
Explaining Monitors Graphically



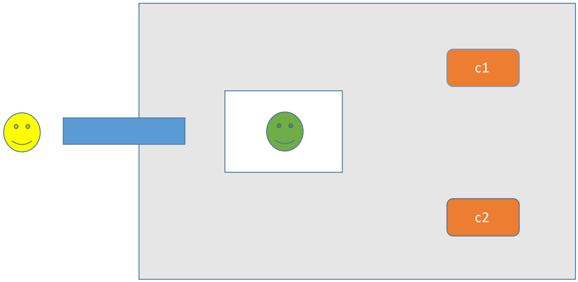
Typical Behavior



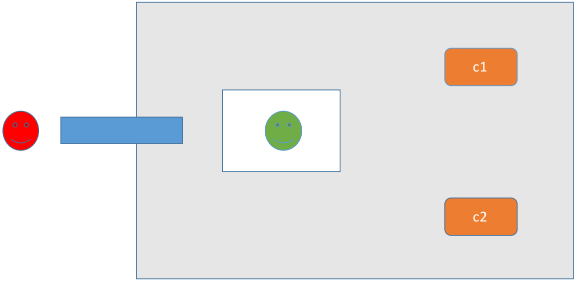
Typical Behavior



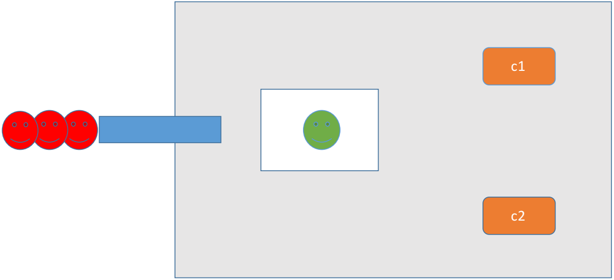
Typical Behavior



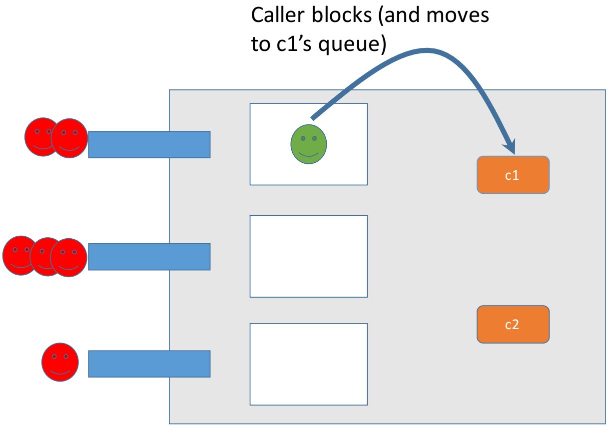
Typical Behavior



Typical Behavior

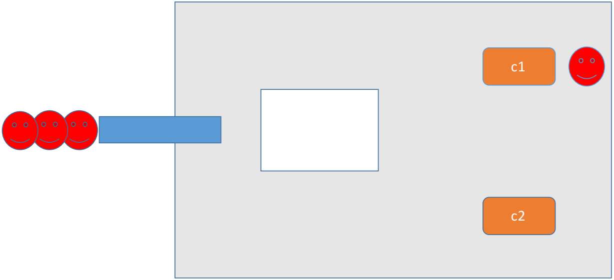


Wait



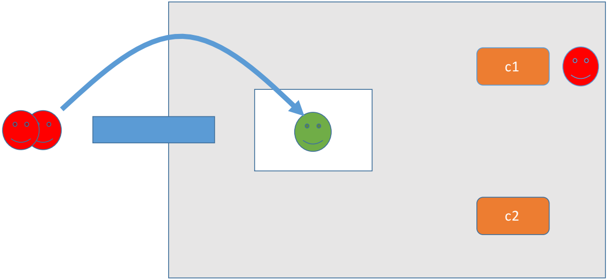
- Blocks process currently executing and associates it to variable's waiting set
- Upon blocking frees the lock allowing the entry of other processes

Wait



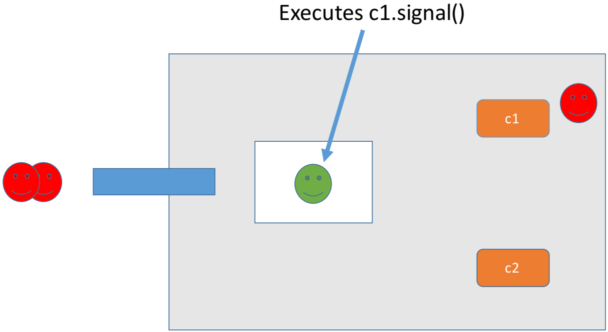
- Blocks process currently executing and associates it to variable's waiting set
- Upon blocking frees the lock allowing the entry of other processes

Wait



- Blocks process currently executing and associates it to variable's waiting set
- Upon blocking frees the lock allowing the entry of other processes

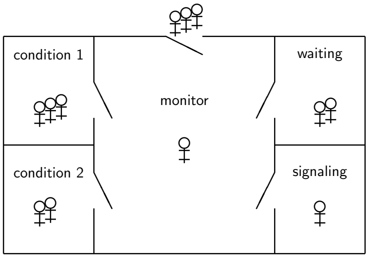
Signal



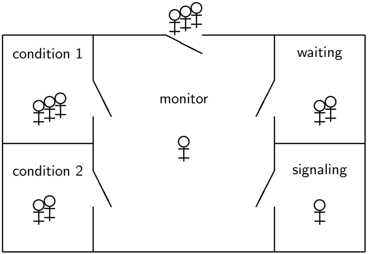
- Signalling process continues to execute after notifying on `c1`?
- Processes waiting in `c1`'s waiting-set start immediately running inside the monitor?
- What about the processes blocked on entry to the monitor?

Signal – States That a Process Can Be In

- Waiting to enter the monitor
- Executing within the monitor (only one)
- Blocked on condition variables
- Set of processes just released from waiting on a condition variable
- Set of processes that have just completed a `signal` operation



Notify



Two strategies:

- Signal and Urgent Wait: $E < S < W$ (classical monitors)
- Signal and Continue: $E = W < S$ (Java $\Leftarrow$ We adopt this)

where the letters denote the precedence of

- S: signalling processes
- W: waiting processes
- E: processes blocked on entry

Buffer of Size 1 in Groovy - Discussion

```
class Buffer {
  private Object buffer = null; // shared buffer

  synchronized Object consume() {
    while (buffer == null)
      wait(); // wait on object's wait-set
    Object aux = buffer;
    buffer = null;
    notifyAll(); // signal on object's wait-set
    return aux;
  }

  synchronized void produce(Object o) {
   while (buffer != null)
      wait(); // wait on object's wait-set
    buffer = o;
    notifyAll(); // signal on object's wait-set
  }
}
```

Buffer of Size 1 in Groovy - Discussion

```
class Buffer {
  private Object buffer = null; // shared buffer

  synchronized Object consume() {
    while (buffer == null)
      wait(); // wait on object's wait-set
    Object aux = buffer;
    buffer = null;
    notify(); // signal on object's wait-set
    return aux;
  }

  synchronized void produce(Object o) {
   while (buffer != null)
      wait(); // wait on object's wait-set
    buffer = o;
    notify(); // signal on object's wait-set
  }
}
```

- What goes wrong and why?

Monitor that Defines a Semaphore

```
class Semaphore {

  private int permissions;

  Semaphore(int n) {
    this.permissions = n;
  }

  synchronized void acquire() {
    while (permissions == 0)
      wait();
    permissions--;
  }

  synchronized void release() {
    permissions++;
    notify();
  }

}
```

Monitor that Defines a Semaphore

```
c=0;
Semaphore mutex = new Semaphore(1);

P = Thread.start {
            20.times{
                mutex.acquire();
                c++;
                mutex.release();
            }
        }

Q = Thread.start {
        20.times {
            mutex.acquire();
            c++;
            mutex.release();
        }
    }

P.join();
Q.join();
println c;
```

Signal and Continue

- Must re-check the condition since may have gained entry long after it was notified

```
synchronized void acquire() {
    while (permissions == 0)
        nonZero.wait();
    permissions--;
}
```

Another reason for re-checking the condition:

- Spurious wakeups: “Implementations are permitted, although not encouraged, to perform "spurious wake-ups", that is, to remove threads from wait sets and thus enable resumption without explicit instructions to do so.”[3]

[3] JLS for Java SE 17 (page 740).
https://docs.oracle.com/javase/specs/jls/se17/jls17.pdf

Signal and Continue

```java
class Semaphore {
  private int permissions;

  Semaphore(int n) {
    this.permissions = n;
  }

  synchronized void acquire() {
    while (permissions == 0)
      wait();
    permissions--;
  }

  synchronized void release() {
    permissions++;
    notifyAll();
  }

}
```

- Is starvation possible?
- See *Specific Notification for Java Thread Synchronization* by Tom Cargill, 1996.[4]

[4] www.dre.vanderbilt.edu/%7Eschmidt/PDF/specific-notification.pdf

Buffer of Size 1 in Groovy - Discussion

```
class Buffer {
  Object buffer = null; // shared buffer

  synchronized Object consume() {
    while (buffer == null)
      wait(); // wait on object's wait-set
    Object aux = buffer;
    buffer = null;
    notifyAll(); // signal on object's wait-set
    return aux;
  }

  synchronized void produce(Object o) {
   while (buffer != null)
      wait(); // wait on object's wait-set
    buffer = o;
    notifyAll(); // signal on object's wait-set
  }
}
```

- Inefficient
- Much more efficient (and clearer) to have multiple wait-sets

Explicit Locks

- Alternative to using intrinsic lock of an object
- Convenient for modeling condition variables
- Example:
  - Declare lock and associate condition variables to it

```java
final Lock lock = new ReentrantLock();
final Condition empty = lock.newCondition();
final Condition full = lock.newCondition();
```

  - Replace `synchronized` with

```java
Lock l = new ReentrantLock();
l.lock();
try {
    // access the resource protected by this lock
} finally {
    l.unlock();
}
```

Condition Variables

```java
import java.util.concurrent.locks.*;

class Buffer {
    Object buffer = null; // shared buffer
    final Lock lock = new ReentrantLock();
    final Condition empty = lock.newCondition();
    final Condition full = lock.newCondition();

    Object consume() {
        lock.lock();
        try {
            while (buffer == null)
                full.await();
            Object aux = buffer;
            buffer = null;
            empty.signal();
            return aux;
        } finally {
            lock.unlock();
        }
    }

    // continues in next slide
}
```

Condition Variables

```
    void produce(Object o) {
        lock.lock();
        try {
            while (buffer != null)
                empty.await();
            buffer = o;
            full.signal();
        } finally {
            lock.unlock();
        }
    }
}
```

Condition Variables

```
Buffer b =new Buffer()

20.times {
    int id=it
    Thread.start {
        println "consumer "+ b.consume();
    }
}

20.times {
    int id=it
    Thread.start {
        b.produce(id);
    }
}
```

Buffer of Size *n*

```
import java.util.concurrent.locks.*;
class PC {
    private Integer[] data;
    private int begin = 0;
    private int end = 0;
    private final int N;
    final Lock lock = new ReentrantLock();
    final Condition notEmpty = lock.newCondition();
    final Condition notFull = lock.newCondition();

    public PC(int size) {
        this.N = size;
        data = new Integer[N];
    }

    public void produce(Integer o) {
        lock.lock();
        try {
            while (isFull()) {
                notFull.await();
            }
            data[begin] = o;
            begin = (begin+1) % N;
            notEmpty.signal()
        } finally {
            lock.unlock();
```

Buffer of Size *n*

```java

    public Integer consume() {
        lock.lock();
        try {
            while (isEmpty()) {
                notEmpty.await();
            }
            Integer result = data[end];
            end = (end+1) % N
            notFull.signal();
            return result;
        } finally {
            lock.unlock();
        }
    }

    private boolean isEmpty() { return begin == end; }
    private boolean isFull()  { return ((begin+1)%N) == end; }
}
```

Readers/Writers

```
class ReadersWriters {
  ...

  public synchronized void read() {
    ...
  }

  public synchronized void write() {
    ...
  }

}
```

What is the problem with this setting?

Readers/Writers

```java
import java.util.concurrent.locks.*;

class ReadersWriters {
    Integer readers = 0;
    Integer writers = 0;
    final Lock lock = new ReentrantLock();
    final Condition okToRead = lock.newCondition();
    final Condition okToWrite = lock.newCondition();

    public void startRead() {
        lock.lock();
        try {
            while (writers != 0) {
                okToRead.await();
            };
            readers = readers + 1;
        } finally {
            lock.unlock();
        }
    }
  // continues
```

Readers/Writers

```java
    public  void endRead() {
        lock.lock();
        try {
            readers = readers - 1;
            if (readers==0) {
                okToWrite.signal();
            }
        } finally {
            lock.unlock();
        }
    }
  // continues
```

Readers/Writers

```
    public void startWrite() {
        lock.lock();
        try {
            while (writers != 0 || readers != 0) {
                okToWrite.await();
            }
            writers = writers + 1;
        } finally {
            lock.unlock();
        }
      }

    public  void endWrite() {
        lock.lock();
        try {
            writers = writers - 1;
            okToWrite.signal();
            okToRead.signalAll();
        } finally {
            lock.unlock();
        }
    }
}
```

Readers/Writers

```
ReadersWriters r = new ReadersWriters()

10.times {
    int id = it;
    Thread.start {
        r.startRead();
        println "$id: start reading...";
        r.endRead();
        println "$id: done reading...";
    }
}

10.times {
    int id = it;
    Thread.start {
        r.startWrite();
        println "$id: start writing..."
        r.endWrite();
        println "$id: done writing..."
    }
  }
```

Assessment

- Upholds the readers-writers invariant but gives priority to readers over writers:
  - new readers can enter the monitor without waiting as long as a reader is active
  - waiting writers have to wait until the last reader calls endRead and signals OKtoWrite
  - as long as readers keep arriving and waiting to enter the monitor, the waiting writers will never execute

RW - Fair on Writers

```
class ReadersWriters {
      ...
      Integer waitingWriters = 0;
      ...
    public void startRead() {
        lock.lock();
        try {
            while (writers != 0 || waitingWriters>0) {
                oKtoRead.await();
            };
            readers = readers + 1;
        } finally {
            lock.unlock();
        }
    }

    public void startWrite() {
    lock.lock();
    try {
        while (writers != 0 || readers != 0) {
            waitingWriters++;
            oKtoWrite.await();
            waitingWriters--;
        }
        writers = writers + 1;
    } finally {
```

Dining Philosophers

```
class ForkMonitor {
    final int N
    List<Integer> forks = []  // forks available to each phil
    final Lock lock = new ReentrantLock()
    final List<Condition> okToEat = []

    ForkMonitor(int n) {
        this.N = n;
        N.times { forks.add(2) }
        N.times { okToEat.add(lock.newCondition()) }
    }
```

`forks[i]` is number of forks available to philosopher *i*

Dining Philosophers

```java
private void updateForkCount(Integer i,Integer delta) {
    lock.lock();
    try {
        forks[(i+1) % N] = forks[(i+1) % N] + delta;
        if (i-1>0) {
            forks[i-1] = forks[i-1] + delta
        } else { // i-1==0
            forks[N-1] = forks[N-1] + delta
        }
    } finally {
        lock.unlock();
    }
}

public void takeForks(int i) {
    lock.lock();
    try {
        while (forks[i] != 2) {
            okToEat[i].await();
        }
        updateForkCount(i,-1);
    } finally {
        lock.unlock();
    }
}
```

Dining Philosophers

```
    public void releaseForks(int i) {
        lock.lock();
        try {
            updateForkCount(i,1);
            if (forks[i+1] == 2) {
                okToEat[i+1].signal();
            }
            if (forks[i-1] == 2) {
                okToEat[i-1].signal();
            }
        } finally {
            lock.unlock();
        }
    }
}
```

Dining Philosophers

```
ForkMonitor dp = new ForkMonitor(5)

5.times {
    int id = it
    Thread.start {
        while (true) {
            dp.takeForks(id)
            println "$id grabbed forks"
            dp.releaseForks(id)
            println "$id released forks"
        }
    }
}
```

Monitors

More Examples

Condition Variables

Visibility

Visibility

- Whether a thread can see the modifications of other threads
- Visibility is subtle because the compiler may
  - Reorder operations
  - Cache values in registers
- `synchronization`, used for atomicity, helps with visibility too:
  - All changes made in one synchronized method or block are visible with respect to other synchronized methods and blocks employing the same lock.

Volatile Variables

```
int n=0;
```

```
Thread.start { // P
  int local1, local2;
  n = some expression;
  computation not using n;
  local1 = (n+5)*7;
  local2 = n+5;
  n = local1 *  local2;
}
```

```
Thread.start { // Q
  int local;
  local = n+6;
}
```

The instruction in Q may be interleaved at any place during the execution of the instructions in P

Volatile Variables

An optimizing compiler could translate statements in thread P as:

```
tempReg1 = some expression
computation not using n
tempReg2 = tempReg1 + 5
local2 = tempReg2
local1 = tempReg2 * 7
n = local1 * local2
```

```
n = some expression;
computation not using n;
local1 = (n+5)*7;
local2 = n+5;
n = local1 *  local2;
.
```

- No assignment to n in the first statement. Original statements p3 and p4 are executed out of order
- Without concurrency, the translated code would be correct
- With concurrency and interleaving, the translated code may no longer be correct
- Specifying a variable as volatile instructs the compiler to load and store the value of the variable at each use, rather than to optimize away these loads and stores.

Example 1[5]

```java
class SharedVariable {
   private static int  sharedVariable = 0;
   public static void main(String[] args) throws InterruptedExcept

   new Thread(new Runnable() {
      @Override
      public void run() {
         try { Thread.sleep(100); }
         catch (InterruptedException e) {
                 e.printStackTrace();
         }
         sharedVariable = 1;
      }}).start();

   for(int i=0;i<1000;i++) {
      for(;;) {
         if(sharedVariable == 1) { break; }
      }
   }
   System.out.println("SharedVariable : " + sharedVariable);
   }
}
```

Try this code as is (loops due to compiler optimization), then add the qualified volatile to the declaration of `sharedVariable` and run

Example 2

```java
class NoVisibility {
    private static boolean ready;
    private static int number;

    private static class ReaderThread extends Thread {
        public void run() {
            while (!ready)
                Thread.yield();
            System.out.println(number);
        }
    }

    public static void main(String[] args) {
        new ReaderThread().start();
        number = 42;
        ready = true;
    }
}
```

- `java.lang.Thread.yield()` causes the currently executing thread object to temporarily pause and allow other threads to execute
- What is the output?

Example 2

```java
class NoVisibility {
    private static boolean ready;
    private static int number;

    private static class ReaderThread extends Thread {
        public void run() {
            while (!ready)
                Thread.yield();
            System.out.println(number);
        }
    }

    public static void main(String[] args) {
        new ReaderThread().start();
        number = 42;
        ready = true;
    }
}
```

- `java.lang.Thread.yield()` causes the currently executing thread object to temporarily pause and allow other threads to execute
- What is the output? Could loop forever or print 0!