Operating Systems And Concurrency, Lecture 6: Monitors and Concurrency in Java

Dr Amanda Coles

Chapter 6 of "Operating Systems Concepts" Silberschatz, Galvin, Gagne

Chapter 7 of "Principles of Concurrent and Distributed Programming" M. Ben-Ari

Plus Additional Java Examples and extracts from the Java API.

Overview

- Motivation for Monitors;
- Monitor Concept;
  - Monitors using semaphores;
  - Semaphores using monitors;
- Canonical problems:
  - The Dining Philosophers Problem;
  - The readers and writers problem
- Concurrency in Java.
  - Train Bridge Example;
  - Swing Threads.

Issues with Semaphores

- Whilst they represent an effective way to manage the critical section problem, which is more user-friendly than hardware provided systems semaphores are still prone to errors.

- All programmers of a system must correctly implement the pattern:
  - wait(S), Critical Section, signal(S);
  - If one gets it wrong (deliberately, to avoid being blocked by other processes, or accidentally) the whole system could suffer violation of mutual exclusion.
- If a system is being contributed to by a large team this is more than likely to happen…

- Common errors:
  - One programmer forgets to signal -> deadlock;
  - One programmer forgets to wait -> mutual exclusion violated;
  - Wrong order of signal/wait:
    - signal(S), Critical Section, wait(S);
  - Use of wait instead of signal:
    - wait(S), Critical Section, wait(S);

Solving the Problem: Monitors

- Invented by Hoare and Hansen.
- Idea:
  - In OO programming objects are encapsulated in classes.
  - A monitor is similar to a class.
  - The shared resource is represented by a monitor which has methods through which other threads/processes can access that resource.
  - The only way to access the resource is through these methods.
    - Think of it as a private instance variable with public accessor methods.
  - Now we simply impose the requirement that none of these methods can execute concurrently.
  - Now only one process can modify/use the resource at once.

Advantages of Monitors

- Responsibility for mutual exclusion is centralised with the creator of the (monitor for) resource.
  - There is therefore, less scope for individual programmers to create synchronisation bugs.
  - Critical section is within the monitor rather than duplicated in processes.
  - Still not perfect though…
- Distributed management of resources:
  - Each resource (or related group of resources) can be handled by a separate monitor.
  - If processes are requesting to run methods on the same monitor mutual exclusion can be enforced;
  - If processes are requesting to run processes of two separate monitors they can be executed concurrently.
- Fits well with OO design principles.

Overview

- Motivation for Monitors;
- Monitor Concept;
  - Monitors using semaphores;
  - Semaphores using monitors;
  - Producer consumer problem using monitors.
- Canonical problems:
  - The Dining Philosophers Problem;
  - The readers and writers problem
- Concurrency in Java.
  - Train Bridge Example;
  - Swing Threads.

Simple Example of a Monitor

```
monitor Integer
  Integer n = 0;
  method increment()
    n = n + 1;
  method decrement()
    n = n - 1;
```

p	q
p1: Integer.increment()	q1: Integer.increment()
p2: Integer.decrement()	q2: Integer.decrement()

- All methods in the monitor are executed in mutual exclusion;
  - A process must acquire the *lock* for the monitor to execute a method.
- increment (decrement) cannot be executed by both processes at the same time;
- P cannot execute decrement whilst q executes increment and vice versa.

Implementing a Monitor Using Semaphores

```
class Monitor
  semaphore s = 1;
  //any required variables

  method method1()
    s.wait()
    //actual method implementation
    s.signal()

  method method 2()
    s.wait()
    //actual method implementation
    s.signal()


etc.
```

Those feeling pedantic might find issue with this, but it suffices to illustrate the point (read 6.7.3 of “Operating Systems Concepts if interested).

Simulating Semaphores using Monitors

```
monitor Semaphore
  Integer s = k

  method semaphoreWait()
    while(s = 0)
      wait()
    s = s - 1

  method semaphoreSignal()
    s = s + 1
    notifyAll()
```

P	q
loop forever:	loop forever:
p0: non-critical section	q0: non-critical section
p1: Semaphore.semaphoreWait()	q1: Semaphore.semaphoreWait()
p2: critical-section	q2: critical-section
p3: Semaphore.semaphoreSignal()	q3: Semaphore.semaphoreSignal()

Monitor wait() and notifyAll()

- Maintain a set *blocked* of threads that are waiting.
- wait():
  - adds the current thread to the set *blocked.*
  - sets its status to blocked.
  - releases the monitor lock.
- notifyAll():
  - Removes all process from the set blocked (if blocked is non empty);
  - Sets their states to ready (whichever thread the scheduler selects to run first will acquire the lock, the other processes will be blocked on entry).
- Threads can only call wait/notifyAll/notify if they hold the monitor lock (i.e. inside the monitor).

```
wait(){
 blocked = blocked U this;
 this.status = blocked
 lock.release()
}
```

```
notifyAll(){
  while(blocked not empty){
    p = blocked.remove();
    p.status = ready;
  }}
```

notifyAll() versus notify()

- notifyAll():
  - Removes all process from the set blocked (if blocked is non empty);
  - Sets their states to ready (whichever thread the scheduler selects to run first will acquire the lock, the other processes will be blocked on entry).

- notify():
  - selects and removes a process from the set blocked (if blocked is non empty);
  - Sets the state of that thread to ready (i.e. it is the next one allowed to execute).

```
notify(){
  if(blocked not empty){
    p = blocked.remove();
    p.status = ready;
  }}
```

```
notifyAll(){
  while(blocked not empty){
    p = blocked.remove();
    p.status = ready;
  }}
```

Producer Consumer Problem using Monitors

```
monitor Buffer
  int[5] Buffer
  int spaceUsed;

method addItem(int i)
    while(spaceUsed == 5)
      wait()
    buffer[spaceUsed] = i
    ++spaceUsed
    notifyAll()

method removeItem()
    while(spaceUsed == 0)
      wait()
    i = buffer[spaceUsed]
    --spaceUsed
    notifyAll()
    return i
```

p	q
loop forever: p1: i <- produce p2: Buffer.addItem(i)	loop forever: q1: i <-Buffer.removeItem() q2: consume(i)

notifyAll() versus notify()

- Suppose that a producer calls notify() but it is another producer that is woken up
  - The consumers remain blocked because they have not been awoken.

- Suppose, in a more general scenario, the thread selected to run by notify() then waits on some other condition.
  - The other threads cannot then run and acquire the monitor lock because they are in the blocked state.

- notifyAll() gets around this by waking up multiple threads, then if one thread were to wait on another condition, another thread will be in the ready queue, and can acquire the monitor lock and run;
  - However, in waking all threads and allowing the scheduler to select which one is running notifyAll() gives rise to more context switching.

- If you are certain that an arbitrary thread that is woken up will not become blocked then you can use notify(). In general though notifyAll() is safer (what if your code is extended later?), even though it gives rise to more overhead.

We have to Notify Producers of Empty and Consumers of Full

- Actually it would be useful if we could only wake up the processes that are actually waiting for the condition that is true.
- The classical monitor concept does allow for this, but Java doesn’t.
- In classical monitors this is achieved by adding conditions to the monitor on which processes can wait:
  - Wait takes as an argument the condition for which the thread is waiting.
  - Signal signals a specific condition.

```
monitor Buffer
  int[5] Buffer
  int spaceUsed;
  condition notEmpty
  condition notFull

method addItem(int i)
    while(spaceUsed = 5)
     wait(notFull)
    buffer[spaceUsed] = i
    ++spaceUsed
    notifyAll(notEmpty)

method removeItem()
    while(spaceUsed == 0)
      wait(notEmpty)
    i = buffer[spaceUsed]
    --spaceUsed
    notifyAll(notFull)
    return i
```

p	q
loop forever: p1: i <- produce p2: Buffer.addItem(i)	loop forever: q1: i <-Buffer.removeItem() q2: consume(i)

Now maintain a separate queue for each condition.

notifyAll(cond) versus notify(cond)

- notifyAll():
  - Removes all process from the set *cond* (if cond is non empty);
  - Sets their states to ready (whichever thread the scheduler selects to run first will acquire the lock, the other processes will be blocked on entry).

- notify():
  - selects and removes a process from the set *cond* (if cond is non empty);
  - Sets the state of that thread to ready (i.e. it is the next one allowed to execute).

```
notify(cond){
  if(cond not empty){
    p = cond.remove();
    p.status = ready;
  }}
```

```
notifyAll(cond){
  while(cond not empty){
    p = cond.remove();
    p.status = ready;
  }}
```

Can we use a Queue instead of a Set?

- It would only be worth doing this if we can call notify() and wake only the head of the queue:
  - If we are going to unblock all processes the order they are stored is irrelevant.

- In the buffer consumer case if all processes waiting on the same condition are in the same queue, then yes we could use a queue and wake up the head of the queue:
  - Avoids the potential starvation caused by simply allowing the scheduler to decide which process to run.

- In general though need to be careful: make absolutely sure that threads are waiting on the same condition: in Java we can't do this because we don't have conditions (unless there's only one condition in the program) therefore must use notifyAll() most of the time.

Can Monitors Deadlock?

```
monitor Example
  //some variables
  condition cond1 cond2

method method1()                    method method1()
    while(!cond1)                       while(!cond2)
      wait()                              wait()
   //optionally do something       //optionally do something
    while(!cond2)                       while(!cond1)
      wait()                              wait()
    //something using both              //something using both
    notify(cond1)                       notify(cond2)
    notify(cond2)                       notify(cond1)
```

- Yes, still need to be careful;
- But at least in this case the code that deadlocks is in the same class and written by the person who created the object: easier to track down.

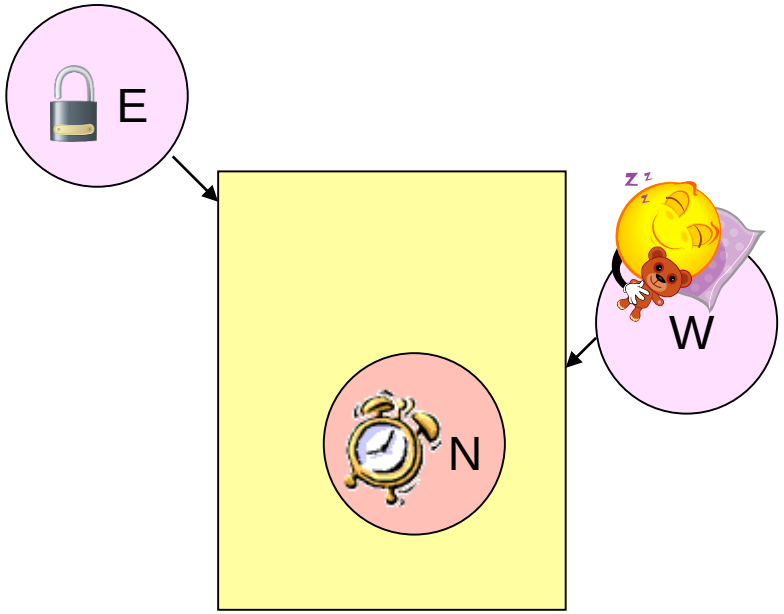
Notifying Before Returning

```
method removeItem()
     while(spaceUsed == 0)
       wait()
     i = buffer[spaceUsed]
     --spaceUsed
     notifyAll()
     return i
```

- Can't always make notifyAll() the last thing we do, what if we have to return something?
- If a thread is in the monitor and it notifies other threads then what happens?
  - This thread gets to keep the lock and carry on executing until it completes?
  - The new threads that have been signalled get the opportunity to run?
  - Any threads that were waiting for the lock to come into the monitor get the opportunity to run?

Thread States w.r.t. Monitors

- Waiting to Enter the monitor (E);
- Waiting to be notified (W);
- Executing notify (N)
  - N.B. in this case the process N currently holds the lock.
- In Java priority is given to thread N, with threads E and W having equal priority:
  - E = W < N
- In the original definition of monitors waiting processes have priority.
  - E < N < W



Implications of Relative Priority

- Take the classical case:
  - Entering < Notifying < Waiting
- If a single process is notified then it knows that the condition holds because it has just been signalled: no other thread has executed between the one that signalled, which has now stopped in favour of the one waiting, so it can continue without rechecking the condition.
- This is the ***Immediate resumption requirement.***
- What about the Java case where E = W < N?

Java Case: E = W < N

- Entering = Waiting < Notifying
  - Good if we want to notify then return.
  - That is the signalling process will carry on until it has left the monitor.
- Now there is no guarantee that the signalling process hasn't changed the condition in the code it continued to execute (which could be arbitrary) before the waiting process was awoken.
- Further, if we are not notifying a single process, but using notifyAll() then a different waiting process might have got to run first and broken the condition: regardless of E, W, N ordering.
- In this case each process must recheck the condition before it can continue executing:
  - We can achieve this by using while(!condition) instead of if(!condition);
  - while cannot exit until the condition has been checked and seen to be false

```
monitor Buffer
  int[5] Buffer
  int spaceUsed;
  condition notEmpty
  condition notFull

method addItem(int i)
    if(spaceUsed = 5)
     wait(notFull)
    buffer[spaceUsed] = i
    ++spaceUsed
    notifyAll(notEmpty)

method removeItem()
    if(spaceUsed = 0)
      wait(notEmpty)
    i = buffer[spaceUsed]
    --spaceUsed
    notifyAll(notFull)
    return i
```

- A process unblocked from the wait here will go straight on to continue without checking that the condition still holds.
- If we used while, the condition must be rechecked:
  - The process will be unblocked, it will then go on to check the condition to see whether it can exit the while loop.
  - If the condition has been changed since notification then the thread will wait again.

b. This part of the question is about monitors.

i. Explain why calling notify() rather than notifyAll() in a monitor might lead to deadlock.

[5 marks]

ii. When considering which thread can access a monitor Java uses the priority order Waiting = Entering < Notifying, whereas the classical monitor uses Entering < Notifying < Waiting. Explain what the states (Waiting, Notifying and Entering) in these priority schedules mean, and what advantage the classical monitor gains by using its strategy in conjunction with notify().

[5 marks]

QUESTION 1 CONTINUES ON NEXT PAGE

May 2015 Q1b (50/150 marks)

Overview

- Motivation for Monitors;
- Monitor Concept;
  - Monitors using semaphores;
  - Semaphores using monitors;
- Canonical problems:
  - The readers and writers problem.
  - The Dining Philosophers Problem (exercise).
- Concurrency in Java.
  - Train Bridge Example;
  - Swing Threads.

Readers and Writers Problem

- Related to the critical section problem:
- Abstraction of reading from and writing to databases.
- There are two types of thread: reader and writer.
  - Many readers can access the database at once.
  - Only one writer can access the database at once.
  - No reader can access the database at the same time as a writer.
- Next slide: a solution using monitors.

```
monitor Buffer
  int readerCount = 0
  boolean writing = false

method startRead()
    while(writing)
     wait()
    ++readerCount;

method endRead()
    --readerCount;
    notifyAll()
```

```
method startWrite()
  while(writing ||
          readerCount != 0)
    wait()
  writing = true;

method endWrite()
    writing = false;
    notifyAll()
```

P	q
loop forever: p1: startRead() p2: read from database p3: endRead()	loop forever: q1: startWrite() q2: write to database q3: endWrite()

Breaking up Conditions

```
method startWrite()
  while(writing ||
        readerCount != 0)
    wait()
  writing = true;
```

```
method startWrite()
  while(writing)
    wait()
  while(readerCount != 0)
    wait()
  writing = true;
```

- On the left both conditions are checked together so we know both hold when we set writing = true;
- On the right we could have violation of our required properties:
  - Wait for writing to be false;
  - Continue to wait for reader count to be zero;
  - It's possible that some other writer, also waiting for reader count to be zero, is scheduled before us when reader count becomes zero;
  - It starts writing but has not yet finished.
  - This writer is awoken, checks reader count is still zero and continues to write.
  - Two writers are now writing simultaneously.
- Lesson: we must check all conditions at once.

Writer Starvation

```
monitor Buffer
  int readerCount = 0
  boolean writing

method startRead()
    while(writing)
     wait()
    ++readerCount;

method endRead()
    --readerCount;
    notifyAll()

method startWrite()
  while(writing ||
          readerCount != 0)
    wait()
  writing = true;

method endWrite()
    writing = false;
    notifyAll()
```

- There is a real risk here that writers will be starved:
  - so long as one reader is reading other readers will be able to enter while writers will be blocked.

Writer Preference

```
monitor Buffer
  int readerCount, waitingReaders = 0
  boolean writing, waitingToWrite = false

method startRead()
    while(writing ||
        waitingToWrite)
     wait()
    ++readerCount;

method endRead()
    --readerCount;
    notifyAll()

method startWrite()
  while(writing ||
          readerCount != 0)
    waitingToWrite = true
    wait()
  writing = true;
  waitingToWrite = false

method endWrite()
    writing = false;
    notifyAll()
```

But what about the readers?

```
monitor Buffer
 int readerCount, waitingReaders = 0
  boolean writing, waitingToWrite, readerTurn = false

method startRead()
    while(writing || (waitingToWrite && !readerTurn))
        ++waitingReaders
        wait()
        --waitingReaders
    ++readerCount;
    readerTurn = false

method startWrite()
  while(writing || readerCount != 0 ||
           (waitingReaders > 0 && readerTurn))
      waitingToWrite = true
      wait()
  writing = true
  waitingToWrite = false
  readerTurn = true;
//end read and end write as before
```

Monitor Solution to the Dining Philosophers Problem

- In the exercises.

Overview

- Motivation for Monitors;
- Monitor Concept;
  - Monitors using semaphores;
  - Semaphores using monitors;
- Canonical problems:
  - The readers and writers problem.
  - The Dining Philosophers Problem (exercise).
- Concurrency in Java.
  - Train Bridge Example;
  - Swing Threads.

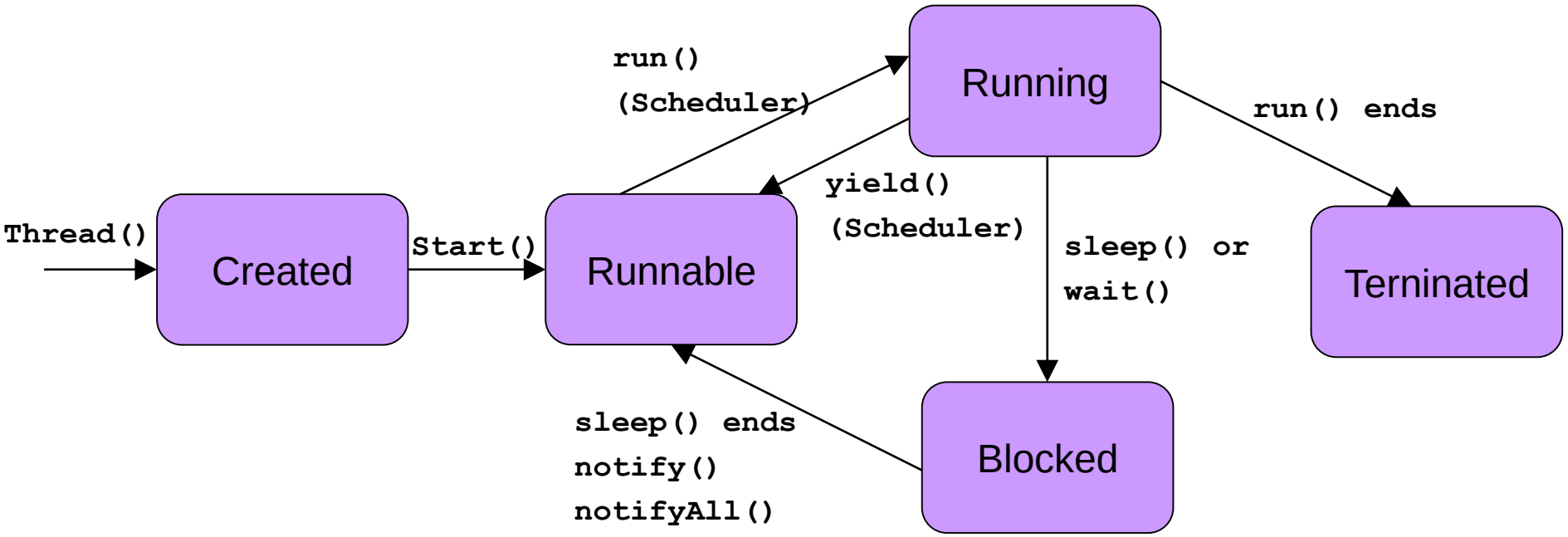
Monitors in Java

- Java's primary means of providing solutions to mutual exclusion is via monitors.
- Monitors are not explicitly represented, rather every class can be thought of as a monitor.
- Unlike traditional monitors not all methods are required to be mutually exclusive.
- Instead methods that the programmer requires to be executed in mutual exclusion are marked using the keyword **`synchronized`**.
- Why not just make everything automatically synchronized?
  - Then no two methods could ever run at the same time on any object even if mutual exclusion is not required.

Object: Java API

protected Object **clone**()	Creates and returns a copy of this object.
boolean **equals**(Object obj)	Indicates whether some other object is "equal to" this one.
protected void **finalize**()	Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class<?> **getClass**()	Returns the runtime class of this Object.
int **hashCode**()	Returns a hash code value for the object.
void **notify**()	**Wakes up a single thread that is waiting on this object's monitor.**
void **notifyAll**()	**Wakes up all threads that are waiting on this object's monitor.**
String **toString**()	Returns a string representation of the object.
void **wait**()	**Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.**
void **wait**(long timeout)	**Causes the current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.**
wait(long timeout, int nanos)	**Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.**

Java Thread Lifecycle



Single-Track Bridge Example

- A bridge running east to west, has only room for a single lane of traffic.
- Traffic must therefore not enter the bridge going in opposite directions.
- The roads on either side are two-way so traffic can pass on those.
- What happens in the code on the next slide?

```
public class Vehicle extends
   Thread(){
  Bridge bridge; String name;
  boolean westbound;
  public Vehicle(Bridge toCross,
   boolean west, String n){
    bridge = toCross;
    name = n; westbound = west;
  }
  public void run(){
    bridge.cross(this);
  }
  public String toString(){
    String dir = "west ";
    if(!westbound) dir = "east ";
    return name + " going " + dir;
  }}
```

```
public class Bridge(){
  boolean westbound = false;

  public void cross(Vehicle v){
    System.out.println(v +
                  "entering bridge");
  try{
     Thread.sleep(100);
   } catch (InterruptedException e){
      e.printStackTrace();
   }
   System.out.println(v +
                  "leaving bridge");
}}
```

```
public static void main (String[] args){
  Bridge b = new Bridge(); boolean dir = false;
  for(int i = 0; i < 5; ++i){
    Thread t = new Vehicle(b, dir, "car" + i);
    t.start();
    dir = !dir;}}
```

What Happens?

```
car0 going east entering bridge
car2 going east entering bridge
car1 going west entering bridge
car3 going west entering bridge
car4 going east entering bridge
car0 going east leaving bridge
car2 going east leaving bridge
car1 going west leaving bridge
car3 going west leaving bridge
car4 going east leaving bridge
```

- Nothing prevents the cars from entering the bridge simultaneously in opposite directions.
- Can we fix it first so that only one car can use the bridge at once?
- Yes, easily!

```
public class Vehicle extends
   Thread(){
  Bridge bridge; String name;
  boolean westbound;
  public Vehicle(Bridge toCross,
   boolean west, String n){
    bridge = toCross;
    name = n; westbound = west;
  }
  public void run(){
    bridge.cross(this);
  }
  public String toString(){
    String dir = "west ";
    if(!westbound) dir = "east ";
    return name + " going " + dir;
  }}
```

```
public class Bridge(){
  boolean westbound = false;

  public synchronized void
   cross(Vehicle v){
    System.out.println(v +
                  "entering bridge");
  try{
     Thread.sleep(100);
   } catch (InterruptedException e){
      e.printStackTrace();
   }
   System.out.println(v +
                  "leaving bridge");
}}
```

```
public static void main (String[] args){
  Bridge b = new Bridge(); boolean dir = false;
  for(int i = 0; i < 5; ++i){
    Thread t = new Vehicle(b, dir, "car" + i);
    t.start();
    dir = !dir;}}
```

Fixed!

```
car1 going west entering bridge
car1 going west leaving bridge
car4 going east entering bridge
car4 going east leaving bridge
car2 going east entering bridge
car2 going east leaving bridge
car0 going east entering bridge
car0 going east leaving bridge
car3 going west entering bridge
car3 going west leaving bridge
```

- Done.
- But… there are unnecessary delays:
  - actually as many cars as we want could enter the bridge (length dependent) so long as they are going in the same direction;
  - Let's say 1 car can go westbound and up to 3 cars can go eastbound at any given time, but still cars cannot cross simultaneously in opposite directions.

Allowing Multiple Cars

```
public class Bridge(){
  int carCount = 0;
  boolean westbound = 0;
  public synchronized void enterBridge(Vehicle v){
    //enter if it is okay to do so, if not wait
  }
  public void cross(Vehicle v){
    //cross the bridge
  }
  public synchronized void exitBridge(Vehicle v){
    //leave and notify all waiting threads
  }
}
```

- We cannot synchronize cross() since more than one car can now use the bridge at once.
- Instead we need to split into, enter, cross and exit, so we can check on entrance and exit.
- This is a general idiom for multiple threads using a resource at once.

Allowing Multiple Cars

```
public class Bridge{
  int carCount = 0;  boolean westbound = false;
  public synchronized void enterBridge(Vehicle v){
    while((v.getWestbound() != westbound && carCount > 0) ||
           (!v.getWestbound() && carCount > 2) ||
           (v.getWestbound() && carCount > 0)) {
      try{
        wait();
      } catch (InterruptedException e){ e.printStackTrace();}
    }
    westbound = v.getWestbound(); //set direction to your direction
    ++carCount; //increment cars using bridge
    System.out.println(v + " entering the bridge");
  } ...
```

- Car must wait if:
  - It is not going in the same direction the bridge is set to and there are cars on the bridge (i.e. a car is on the bridge going the other way); **OR**
  - It is going eastbound and there are more than 2 cars on the bridge; **OR**
  - It is going westbound and there is a car on the bridge.

Allowing Multiple Cars

```
//int carCount = 0;  boolean westbound = false;
  public void crossBridge(){
    try{
      Thread.sleep(100);
    } catch (InterruptedException e){
       e.printStackTrace();
    }
  }
  public synchronized void exitBridge(Vehicle v){
    --carCount;
    System.out.println(v + " exiting the bridge");
    notifyAll();
  }
}
```

- Cross just sleeps to simulate taking some time to cross.
- Exit decrements carCount and must then call notifyAll() to see if anyone who is waiting can now enter.

Modifying Run

```
public void run(){
      bridge.enterBridge(this);
      bridge.crossBridge();
      bridge.exitBridge(this);
  }
```

- Now all cars must call enterBridge(this), then crossBridge() then exitBridge(this).
- Also added the getWestbound() method.

Main Method

```
public static void main (String[] args){
        Bridge b = new Bridge(); boolean dir = false;
        for(int i = 0; i < 5; ++i){
            Thread t = new Vehicle(b, dir, "car" + i);
            t.start();
            dir = !dir;
            try{
                Thread.sleep(70);
            } catch (InterruptedException e){
                e.printStackTrace();
            }
        }
    }
```

- Note I made the main method sleep between creating threads:
  - Otherwise all threads are created during the first bridge crossing;
  - Therefore all the eastbound traffic goes across followed by all westbound and the output doesn't demonstrate the intended function.
  - We could fix this another way similar to how we did for readers/writers.

Output

```
car0 going east entering the bridge
car0 going east exiting the bridge
car1 going west entering the bridge
car1 going west exiting the bridge
car2 going east entering the bridge
car4 going east entering the bridge
car2 going east exiting the bridge
car4 going east exiting the bridge
car3 going west entering the bridge
car3 going west exiting the bridge
```

- Interestingly here there is nothing to stop the cars exiting the bridge in a different order to which they entered.
- We could prevent this but it doesn't really matter we can just look at the entrance order and derive the exit order from that.
- Really we only need exit to show the code works as desired, and so that we can call notifyAll().

```
public class Factory {
 boolean processingMeat = false;
 int linesBeingProduced = 0;
 public synchronized void
  startProduction(Product p){
  //code to be added here by you (1)
  processingMeat = p.isMeat();
  ++linesBeingProduced;
  System.out.println("Producing
          Product " + p);
}
 public void produce(){
  try{
    Thread.sleep(100);
  } catch (Exception e){
    e.printStackTrace();
  }
 }
 public synchronized void
  endProduction(Product p){
  --linesBeingProduced;
  System.out.println("Done
          Product " + p)
  //code to be added here by you (2)
 }
}
```

i. The factory owner must ensure that no meat products are made at the same time as vegetarian products, and that the production limits of a maximum of 2 meat products and 3 vegetarian products being produced at any one time are strictly adhered to.
Complete the `Factory` code, by stating what you would write at the points labelled:

- `//code to be added here by you (1)`; and,
- `//code to be added here by you (2)`

You must ensure the production restrictions are enforced; whilst ensuring the factory can still complete production of all orders.
(Label your answers (1) and (2), to be clear what code goes where.)
[12 marks]

ii. What would be the impact of making the `produce()` method `synchronized`? Explain your answer.
[3 marks]

iii. Assuming a continuous input of products to be produced would your answer above prevent starvation of either meat products, vegetarian products, neither or both? Explain how your answer to part 2.c.i could be modified to give a version that prevents the starvation of meat products but not vegetarian products, and different version that prevents the starvation of both meat and vegetarian products (if your original answer satisfies either of these then state that it does so here). You may add any additional variables you wish.
[10 marks]

May 2015 Q2 (50/150 marks)

Overview

- Motivation for Monitors;
- Monitor Concept;
  - Monitors using semaphores;
  - Semaphores using monitors;
- Canonical problems:
  - The readers and writers problem.
  - The Dining Philosophers Problem (exercise).
- Concurrency in Java.
  - Train Bridge Example;
  - Swing Threads.

Inbuilt Concurrency in Swing

- When working with Swing Java automatically uses threads to allow parallelism to ensure that GUIs are responsive
- This means you can get concurrency bugs without even having explicitly created a thread;
- Swing has a special thread: the *event dispatch thread.*
  - All event-handling code is executed in this thread (this prevents threading issues between event handlers).
  - Most code that interacts with the Swing framework must also execute on this thread.
- This automatically runs in parallel with your main thread (and any others that you create from it) without you doing anything.
- Can also implement swing worker threads to perform time-consuming tasks (e.g. to load background images).

Surprising Problems

```
public class BallPanel extends JPanel {
    ArrayList<Ball> balls;
    Random r;

    public BallPanel(){
        balls = new ArrayList();
        setPreferredSize(new Dimension(600,600));
        r = new Random();
        for(int i = 0; i < 50; ++i){
            balls.add(new Ball(r));
        }
    } ...
```

Here is a JPanel that we have extended and on which we will draw balls, the ball constructor creates a ball object with random x,y,width, height.

Painting

```
@Override
  public void paintComponent(Graphics g){
    System.out.println("paint " + Thread.currentThread().getName());
    super.paintComponent(g);
    g.setColor(Color.black);
    g.drawRect(0, 0, 600, 600);
    for(Ball b : balls){
      g.setColor(b.getColour());
      g.fillOval(b.getX(), b.getY(), b.getWidth(), b.getHeight());
    }
  }
```

- Overriding paint replaces what would normally be drawn on the JPanel with what we want to draw, in this case our balls.
- When paint is called it will be run in the event dispatch thread, in parallel with our code.
- From the println we see: paint AWT-EventQueue-0

Moving Balls Around

```
public void animate(){
    System.out.println("animate " + Thread.currentThread().getName());
    Ball fiveBounce = null;
    for(Ball b : balls){
      b.move();
      if(b.getBounces() >= 5) fiveBounce = b;
    }
    Thread.sleep(10); //surrounded by try catch
    repaint();  //what happens if we call paint(getGraphics());?
    if(fiveBounce != null){
      balls.remove(fiveBounce);
    }
}
```

- This is non-swing code so will run in the main thread.
- It moves each ball (by 1 to 3 pixels horizontally and vertically bouncing if the side is hit, see ball.move()) then calls repaint() to draw the new positions.
- Each ball is removed from the list once it has bounced 5 times.

What Happens?

Exception in thread "AWT-EventQueue-0" java.util.ConcurrentModificationException

at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:819)
at java.util.ArrayList$Itr.next(ArrayList.java:791)
at bouncingballs.BallPanel.paintComponent(BallPanel.java:37)
at javax.swing.JComponent.paint(JComponent.java:1054)
at javax.swing.JComponent.paintToOffscreen(JComponent.java:5221)
at javax.swing.RepaintManager$PaintManager.paintDoubleBuffered(RepaintManager.java:1512)
at javax.swing.RepaintManager$PaintManager.paint(RepaintManager.java:1443)
at javax.swing.RepaintManager.paint(RepaintManager.java:1236)
at javax.swing.JComponent._paintImmediately(JComponent.java:5169)
at javax.swing.JComponent.paintImmediately(JComponent.java:4980)
at javax.swing.RepaintManager$3.run(RepaintManager.java:796)
at javax.swing.RepaintManager$3.run(RepaintManager.java:784)
at java.security.AccessController.doPrivileged(Native Method)
at java.security.ProtectionDomain$1.doIntersectionPrivilege(ProtectionDomain.java:76)
at javax.swing.RepaintManager.paintDirtyRegions(RepaintManager.java:784)
at javax.swing.RepaintManager.paintDirtyRegions(RepaintManager.java:757)
at javax.swing.RepaintManager.prePaintDirtyRegions(RepaintManager.java:706)
at javax.swing.RepaintManager.access$1000(RepaintManager.java:62)
at javax.swing.RepaintManager$ProcessingRunnable.run(RepaintManager.java:1651)
at java.awt.event.InvocationEvent.dispatch(InvocationEvent.java:251)
at java.awt.EventQueue.dispatchEventImpl(EventQueue.java:727)
at java.awt.EventQueue.access$200(EventQueue.java:103)
at java.awt.EventQueue$3.run(EventQueue.java:688)
at java.awt.EventQueue$3.run(EventQueue.java:686)
at java.security.AccessController.doPrivileged(Native Method)
at java.security.ProtectionDomain$1.doIntersectionPrivilege(ProtectionDomain.java:76)
at java.awt.EventQueue.dispatchEvent(EventQueue.java:697)
at java.awt.EventDispatchThread.pumpOneEventForFilters(EventDispatchThread.java:242)
at java.awt.EventDispatchThread.pumpEventsForFilter(EventDispatchThread.java:161)
at java.awt.EventDispatchThread.pumpEventsForHierarchy(EventDispatchThread.java:150)
at java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:146)
at java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:138)
at java.awt.EventDispatchThread.run(EventDispatchThread.java:91)

What Happened?

Paint was doing this in the event dispatch thread:

```
for(Ball b : balls){
  g.setColor(b.getColour());
  g.fillOval(b.getX(), b.getY(), b.getWidth(), b.getHeight());
}
```

… and in the middle of that animate did this in the main thread

```
balls.remove(fiveBounce);
```

If a list is modified during iteration a **`ConcurrentModificationException`** is thrown, which is exactly what happened.

Fixing it

```
@Override
  public synchronized void paintComponent(Graphics g){
    //etc.
  }
```

Paint will still run in a separate thread but now must not execute in parallel with removeBall.

```
  public synchronized void removeBall(Ball toRemove){
    balls.remove(toRemove);
  }
Make animate call removeBall(fiveBounces) instead of removing the ball.
```

Don’t just synchronize the whole of animate because then paint can’t run even when it is sleeping.

An Event Handler

```
//window constructor
  startAgain = new JButton("Start Again");
  add(startAgain,BorderLayout.SOUTH);
  startAgain.addActionListener(new Restarter());

class Restarter implements ActionListener{
  @Override
  public void actionPerformed(ActionEvent e) {
    System.out.println("Action Listener " +
                                    Thread.currentThread().getName());
  }
}
```

- When we invoke the ActionListener by clicking the button we see: Action Listener AWT-EventQueue-0
- It is running in the event dispatch thread, the same one as paint.

Overview

- Motivation for Monitors;
- Monitor Concept;
  - Monitors using semaphores;
  - Semaphores using monitors;
- Canonical problems:
  - The readers and writers problem.
  - The Dining Philosophers Problem (exercise).
- Concurrency in Java.
  - Train Bridge Example;
  - Swing Threads.