Coverage of Threads and Concurrency

1. What are Threads?
2. Why are Threads needed?
3. Properties of Concurrent and Parallel Programs
    - Race Conditions
    - Liveness
    - Safety
4. Applications of Threads
    - Applets
5. True Parallelism - Distributed Computing

Threads - Simultaneous Activities

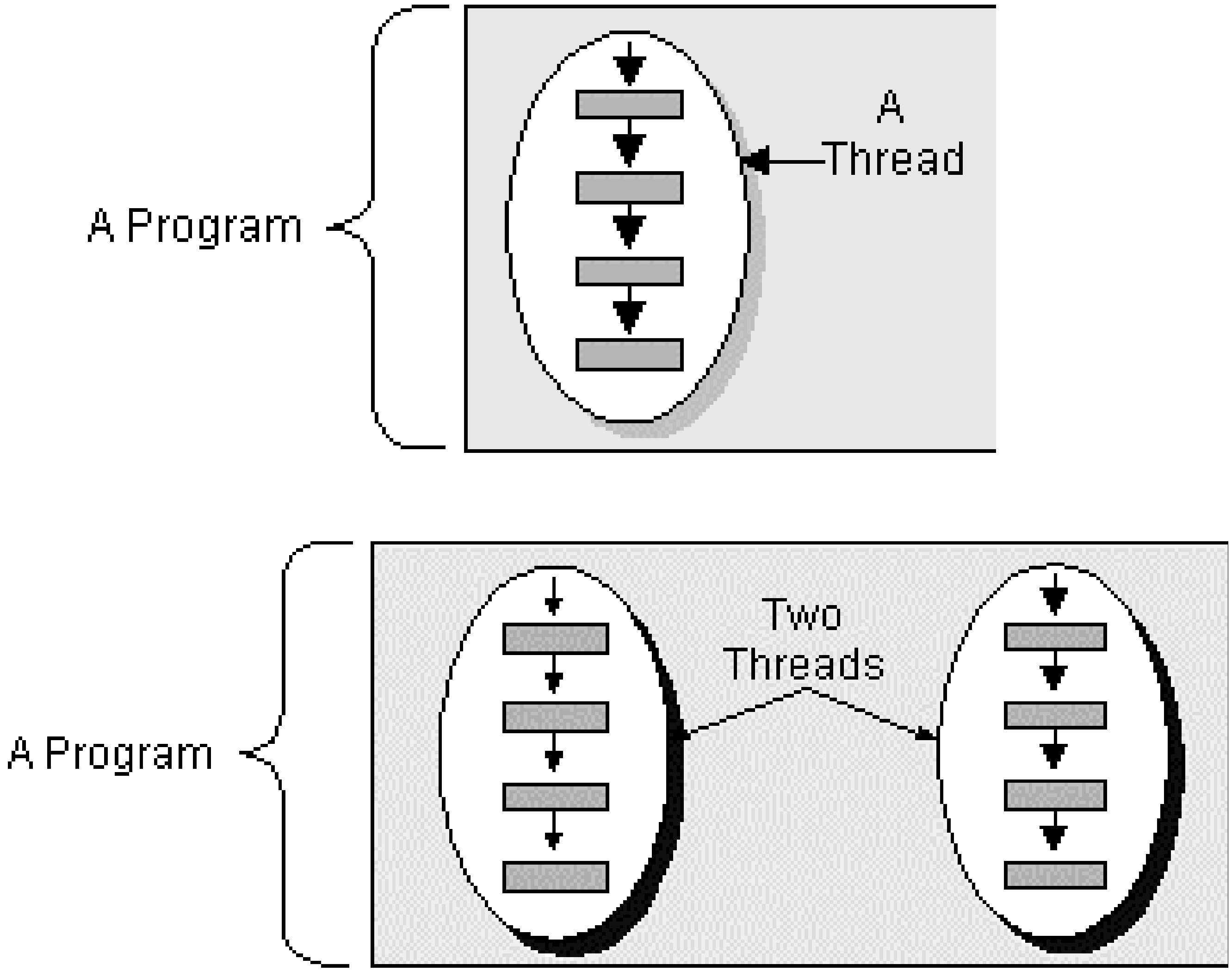
Human activities are intrinsically parallel. When you drive a car you steer, work the accelerator, listen to the radio, talk on your cell phone, etc. Computer programs often need this same form of simultaneous behavior. Threads are the Java mechanism for realizing simultaneous activities.

1. A Thread is a stream of activity.
2. A Java program can have many separate but interacting streams of activity.
3. There are mechanisms for creating threads of activity and mechanisms for assigning units of Java programs to these threads of activity.

The material which follows is taken from the Sun documentation on Threads.

Definition: A thread is a single sequential flow of control within a program.

Threads and Concurrency



Interactions among Threads

Threads are like people. If threads just ran concurrently and did not interact then thread programming would be little different from sequential programming. If people people did not interact there would be no quarrels and wars. But threads, like people, can interact:

by sharing access to data

by sending one another messages (covered later in the lectures.)

by invoking methods of other object instances

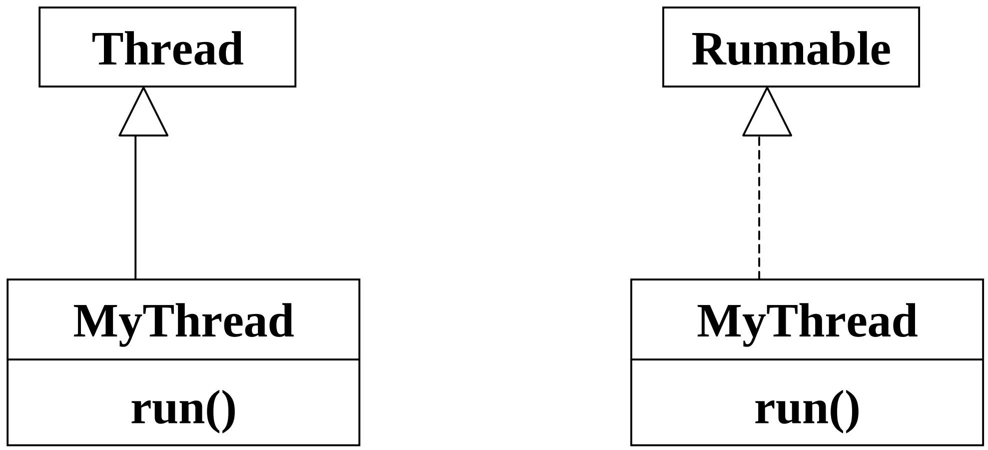
by causing other threads to wait or to execute.

The goal of a set of interacting threads is to accomplish some goal. This usually implies that there is some dependence among the concurrent interacting threads. This dependence may manifest itself as a requirement that threads execute in some sequence.

How do we specify the correct sequences of interaction?

What mechanisms are available to implementing these specifications?

Specification of Threads



Two modes of implementing Threads.
Run() is a “hook” method which must be implemented by any class either implementing Runnable or extending Thread.

public class Thread extends <u>Object</u> implements <u>Runnable</u>

A *thread* is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.

Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority. Each thread may or may not also be marked as a daemon. When code running in some thread creates a new Thread object, the new thread has its priority initially set equal to the priority of the creating thread, and is a daemon thread if and only if the creating thread is a daemon.

When a Java Virtual Machine starts up, there is usually a single non-daemon thread (which typically calls the method named main of some designated class). The Java Virtual Machine continues to execute threads until either of the following occurs:

- **The exit method of class Runtime has been called and the security manager has permitted the exit operation to take place.**
- **All threads that are not daemon threads have died, either by returning from the call to the run method or by throwing an exception that propagates beyond the run method.**

There are two ways to create a new thread of execution. One is to declare a class to be a subclass of Thread. This subclass should override the run method of class Thread. An instance of the subclass can then be allocated and started. For example, a thread that computes primes larger than a stated value could be written as follows:

```
class PrimeThread extends Thread {
   long minPrime;
   PrimeThread(long minPrime) {
      this.minPrime = minPrime;
   }
```

```
        public void run() {
            // compute primes larger than minPrime
             . . .
        }
    }
```

The following code would then create a thread and start it running:

```
    PrimeThread p = new PrimeThread(143);
    p.start();
```

The other way to create a thread is to declare a class that implements the Runnable interface. That class then implements the run method. An instance of the class can then be allocated, passed as an argument when creating Thread, and started. The same example in this other style looks like the following:

```
    class PrimeRun implements Runnable {
       long minPrime;
       PrimeRun(long minPrime) {
          this.minPrime = minPrime;
       }
```

```
public void run() {
        // compute primes larger than minPrime
        . . .
    }
}
```

The following code would then create a thread and start it running:

```
PrimeRun p = new PrimeRun(143);
new Thread(p).start();
```

Every thread has a name for identification purposes. More than one thread may have the same name. If a name is not specified when a thread is created, a new name is generated for it.

Some Methods of class Thread

```
static void  dumpStack()
Prints a stack trace of the current thread.
void  interrupt()
Interrupts this thread.
static boolean      interrupted()
Tests whether the current thread has been interrupted
boolean      isInterrupted()
Tests whether this thread has been interrupted.
void  join()
Waits for this thread to die.
void  join(long millis)
Waits at most millis milliseconds for this thread to die.
void  join(long millis, int nanos)
Waits at most millis milliseconds plus nanos nanoseconds for
this thread to die.
```

void <u>run</u>()
If this thread was constructed using a separate Runnable run object, then that Runnable object's run method is called; otherwise, this method does nothing and return

static void <u>sleep</u>(long millis)
Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
static void <u>sleep</u>(long millis, int nanos)
Causes the currently executing thread to sleep (cease execution) for the specified number of milliseconds plus the specified number of nanoseconds.
void <u>start</u>()
Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.
static void <u>yield</u>()
Causes the currently executing thread object to temporarily pause and allow other threads to execute.

```
public class Counter1 extends Thread {
    protected int count;
    protected int inc;
    protected int delay;
    public Counter1( int  init,  int  inc,  int  delay ) {
        this.count = init;
        this.inc = inc;
        this.delay = delay;}
    public void run() {
        try {
            for (int i = 1; i <= 5; i = i +1) {
                System.out.print(count + " ");
                count = count + inc;
                                System.out.print(count + " ");
                sleep(delay);}
        } catch (InterruptedException e) {}
          System.exit(0);}
    public static void main(String[] args) {
        new Counter1(0, 1, 33).start();
        new Counter1(0, -1, 100).start();
    }
}
```

```
public class Counter1 extends Thread {
   protected static int count;
   protected int inc;
   protected int delay;
   public Counter1( int  init,  int  inc,  int  delay ) {
      this.count = init;
      this.inc = inc;
      this.delay = delay;}
   public void run() {
      try {
         for (int i = 1; i <= 5; i = i +1) {
            System.out.print(count + " ");
            count = count + inc;
                    System.out.print(count + " ");
            sleep(delay);}
      } catch (InterruptedException e) {}
       System.exit(0);}
   public static void main(String[] args) {
      new Counter1(0, 1, 33).start();
      new Counter1(0, -1, 100).start();
   }
}
```

What difference is made in the program by making Count static?

Some Methods that Thread inherits from Object.

void notify()
Wakes up a single thread that is waiting on this object's monitor.

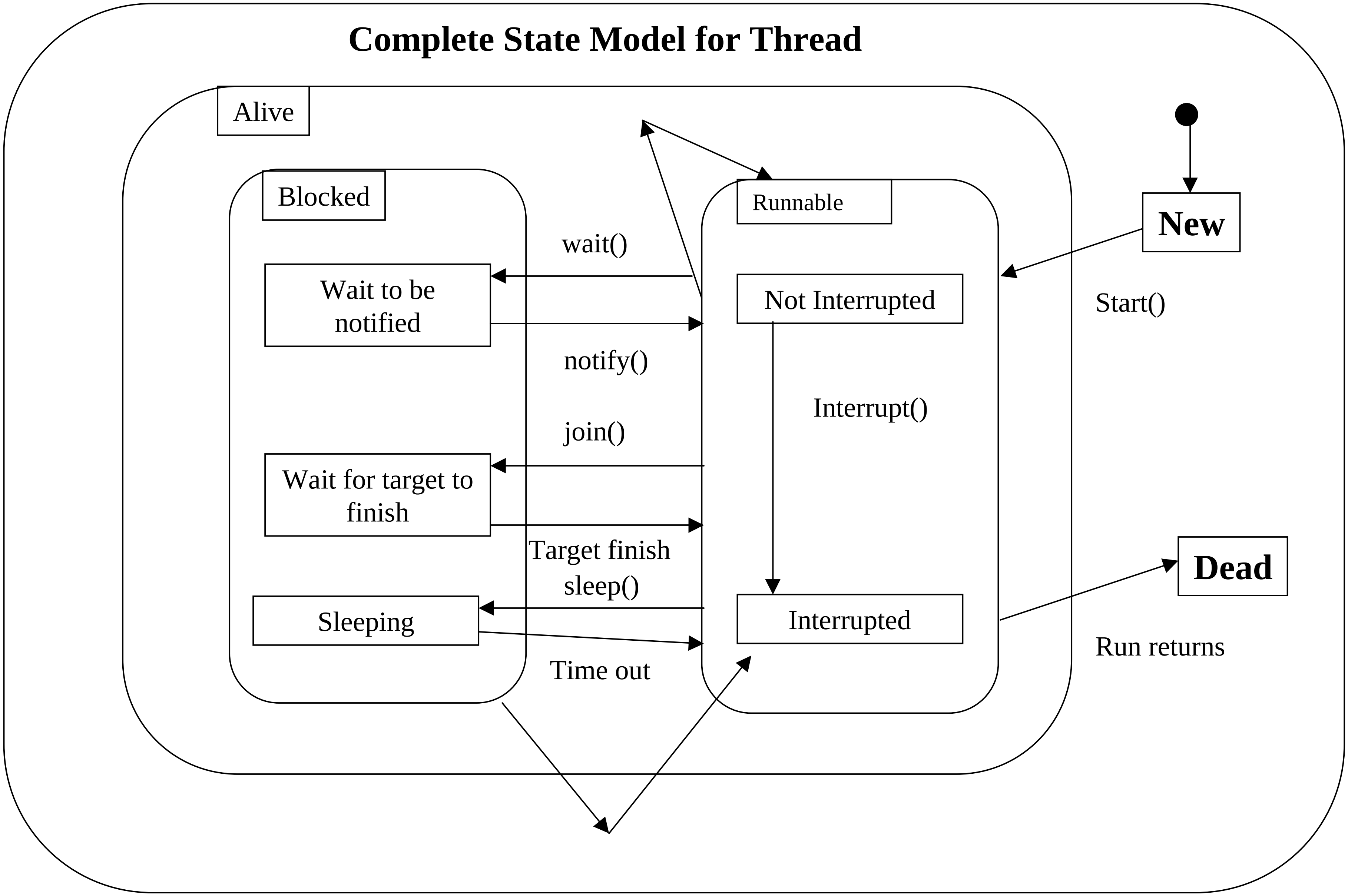
void notifyAll()
Wakes up all threads that are waiting on this object's monitor.

void wait()
Causes current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.

void wait(long timeout)
Causes 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.

Threads and Concurrency

Complete State Model for Thread



Thread Execution Behavior

Threads in one class invoke methods of other classes which may not have been declared as threads. In this case the "thread" of control of the thread which invoked the method is in the invoked method.

1. Class A extends Thread

2. Class B is an ordinary class

3. Method A.m of A invokes method B.m of B.

4. The thread of control of this instance of A is in method B.m of B.

Locks and Wait Queues

1. Lock - The JVM can associate a "hidden" "lock" variable with any method or expression. The "lock" variable has two states, "locked" and "unlocked."

2. Wait Queue:

When a thread executes a "wait()" operation, it is put in the blocked state and is placed on the "wait queue."

When another thread in the same JVM executes a notify() the thread at the top of the wait queue is "awakened" and control is turned over to the awakened thread.

When another thread in the same JVM executes a notifyAll () all of the threads the wait queue are "awakened" and control is given to the thread which was at the top of the queue and it executes. The other awakened threads eventually get to execute.

Race Conditions and Synchronization

```
Public class Account

{……….

………..

Public boolean withdraw(long amount)

        {if (amount <= balance)

        {long newBalance = balance - amount;

        balance = newBalance;

        return true;}

        else return false;

        }

private long balance;

}
```

1. Assume the balance is $10,
2. A thread executing on behalf of one ATM machine starts withdraw($10) and gets as far as executing the statement
long newBalance = balance - amount;
and its quantum of execution expirs.
3. Then another thread executing on behalf of one ATM machine starts withdraw($10) and gets as far as executing the statement
balance = newBalance;
4. Then the first thread resumes and executes
balance = newBalance;
5. Then what is the value of balance?

```
Public class Account
{……….
………..
public synchronized boolean withdraw(long
amount)
            {if (amount <= balance)
            {long newBalance = balance - amount;
            balance = newBalance;
            return true;}
            else return false;
            }
private long balance;
}
```

If we make a method "synchronized" then one invocation of that method can be active at once.

The method has become an atomic action. It is said to be "thread-safe." Specifically the synchronized method is "race-free."

The JVM associates a "lock" with each synchronized method.

But there are still other problems with concurrency.

Interaction Versus Collaboration

Threads can be programmed to interact safely and correctly or they can be programming to collaborate to attain a common goal.

1. Free of Race Conditions - synchronization

2. Safety and Liveness

- **Threads, like people, can interact in constructive or destructive modes.**
- **One thread can ignore the work of another.**
- **Pairs of threads can block one another from proceeding.**
  - **Deadlock**
  - **Flow dependence failures**
  - **Cooperation failures**
- **Some threads can delay the progress of anther thread for an indefinite period of time.**
  - **(Starvation)**

Deadlock - The simplest way for deadlock to occur is for two threads to seek control over two resources.

1. Thread A enters synchronized method p Method p must invoke method q before it completes.

2. Thread B enters synchronized method q. Method q must invoke method p before it completes.

3. Result - Neither A nor B can make progress.

Starvation - The Readers and Writers problem.

1. Let there be 100 reader threads each of which is making requests to read as quickly as possible using a synchronized method but only a single thread trying to write.

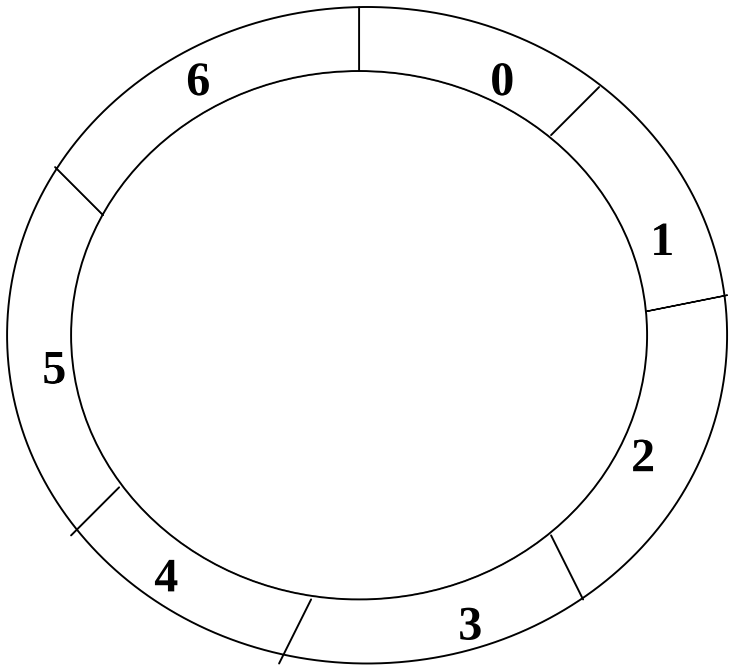
2. Then the synchronized method will be mostly held by the readers and a writer may not be able to gain access for a long period of time.

wait() and notify() give threads the means to collaborate.

The typical pattern is:

1. **Assume threads A and B are cooperating. A creates data needed by B and puts it in a communication object such as a bounder buffer.**
2. **Thread B looks to see if there is any data for him/her. If not the thread B does a wait()**
3. **Thread A creates some data needed by Thread A and puts it in a communication object.**
4. **Thread A then does a notify() followed by a wait().**
5. **When thread B has completed accessing the communication object it may do a notify() to awaken thread A so that thread A can generate additional data.**

Bounded or Circular Buffer



A circular bounded buffer operates like a circular queue. The buffers are filled in sequence and emptied in sequence.

The purpose of this data structure is for several processes or threads to communicate.

That is: the producer fills buffer 0, then buffer 1, etc. and begins with 0 again after filling 6.

Similarly the consumer cycles through the buffers beginning with buffer 0.

If the producer is faster than the consumer then entries may be overwritten without being used.

```
public class BoundedQueue
{  protected Object rep[];
   protected int front = 0;
   protected int back = -1;
   protected int size = 0;
   protected int count = 0;

   public BoundedQueue(int size) {
      if (size > 0) {
         this.size = size;
         rep = new Object[size];
         back = size - 1;
      }
   }
   public boolean isEmpty() {
      return (count == 0);
   }
/* isEmpty() returns zero if there are no entries in the queue. */
```

This implementation of the BoundedQueue will generally work for sequential programs but is not suitable for use with threads..

```
public boolean isFull() {
        return (count == size);
    }
    public int getCount() {
        return count;
    }
    public void put(Object e)
{ /* put checks to see if the object it is to put in the buffer has a value and if
            the buffer is full before executing.  This is a precondition for this
            method. */
        if (e != null && !isFull())
       {
            back++;
            if (back >= size)
                back = 0;
            rep[back] = e;
            count++;
        } return
    }
```

```
public Object get() {
        Object result = null;
        if (!isEmpty()) { // Checks to see if there are entries in the buffer.
            result = rep[front];
            rep[front] = null; // Sets entry which has been accessed to null.
            front++;
            if (front >= size)
                front = 0;
            count--; }
        return result; }
    public static void main(String args[]) {
        BoundedQueue queue = new BoundedQueue(10);
        for (int i = 0; !queue.isFull(); i++) {
            queue.put(new Integer(i)); // Simple producer.
            System.out.println("put: " + i); }
        while (!queue.isEmpty()) {  // Simple consumer.
            System.out.println("get: " + queue.get()); }
    }
}
```

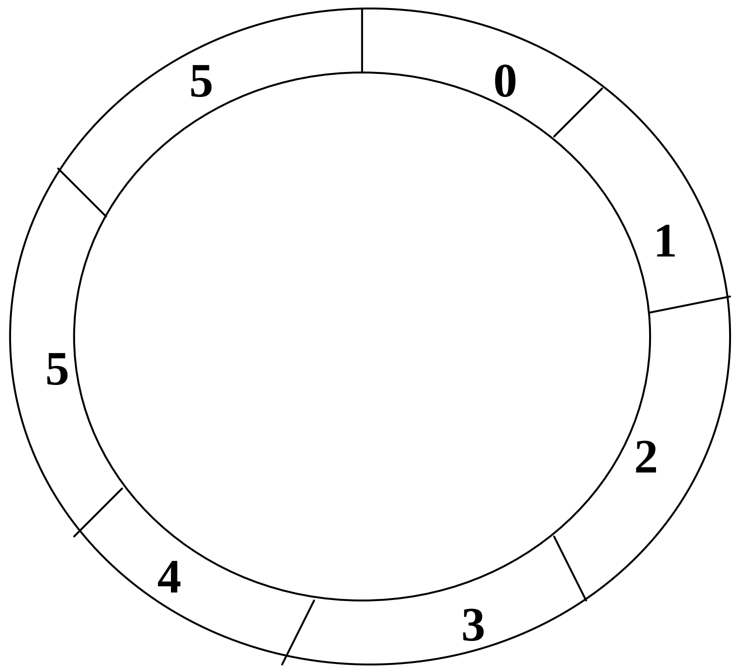
Not we are not using the Producer and Consumer classes as of yet.

This main will fill up the buffer and then empty it.

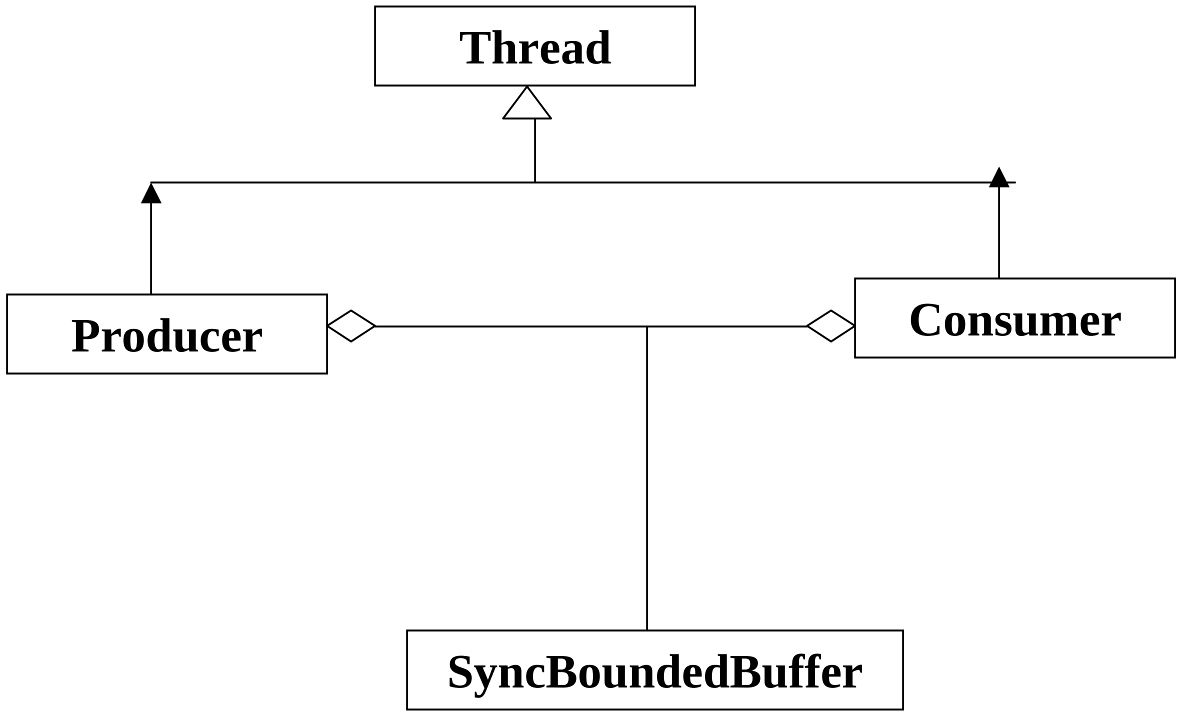
This BoundedBuffer will work fine as long as the producer and consumer execute sequentially.

But what happens if the producer and the consumer are each threads and interact through the buffer? There are two problems. The producer and consumer can both access the same buffer at the same time This is a race condition.

Or suppose the producer is faster than the consumer. Will this code work?



Producer	**consumer**
0	
1	**0**
2	
3	
4	**1**
5	
0	
1	**2**
2	
3	
4	**3**



Let us make the Producer and Consumer classes threads and have an instance of SyncBoundedBuffer as a shared instance variable..

```
public class SyncBoundedQueue extends BoundedQueue
{   public SyncBoundedQueue(int size){
      super(size);}
   synchronized public boolean isEmpty() {
      return super.isEmpty();}
   synchronized public boolean isFull() {
      return  super.isFull();}
   synchronized public int getCount() {
      return super.getCount();}
   synchronized public void put(Object e) {
      super.put(e);}
   synchronized public Object get() {
      return super.get();}
   public static void main(String args[]) {
      SyncBoundedQueue queue = new SyncBoundedQueue(5);
      new Producer(queue, 15).start();
      new Consumer(queue, 10).start(); }
}
```

The producer and consumer can now never access the buffer at the same time so we have no race condition.

```
public class Producer extends Thread {
   protected BoundedQueue queue;
   protected int n;

   public Producer(BoundedQueue queue, int n) {
      this.queue = queue;
      this.n = n;
   }
   public void run() {
      for (int i = 0; i < n; i++) {
         queue.put(new Integer(i));
         System.out.println("produce: " + i);
         try {
           sleep((int)(Math.random() * 100));
         }  catch (InterruptedException e) {}
      }
   }
}
```

When the run() method of Producer executes the put() method of BoundedQueue the thread of control of the run() method is in the put() method of BoundedQueue.

Note that the Producer takes multiples of 100 milliseconds to generate another entry for the buffer.

```
public class Consumer extends Thread {
   protected BoundedQueue queue;
   protected int n;

   public Consumer(BoundedQueue queue, int n) {
      this.queue = queue;
      this.n = n;
   }
   public void run() {
      for (int i = 0; i < n; i++) {
         Object obj = queue.get();
         if (obj != null)
            System.out.println("\tconsume: "+obj);
         try {
            sleep((int)(Math.random() * 400));
         }  catch (InterruptedException e) {}
      }
   }
}
```

If get() finds a null object in a given buffer slot then it waits for it to become full.

But the producer can still overwrite buffers which have not yet been read.

Principle: "Don't use time as a means of flow control. Instead, explicitly specify sequences of actions."

Note that the Consumer takes multiples of 400 milliseconds to consume a buffer entry.

```
public class BoundedQueueWithGuard extends
BoundedQueue {
   public BoundedQueueWithGuard(int size) {
      super(size);
   }
   synchronized public boolean isEmpty() {
      return super.isEmpty();
   }
   synchronized public boolean isFull() {
      return super.isFull();
   }
   synchronized public int getCount() {
      return super.getCount();
   }
   synchronized public void put(Object obj) {
      try {
         while (isFull()) {
            wait();
         }
      } catch (InterruptedException e) {}
      super.put(obj);
      notify();
   }
```

This is an example of a guard in a synchronized method. If there is no empty slot then put() will cause a wait() for the instance of the Producer which executes .

```
synchronized public Object get() {
    try {
      while (isEmpty()) {
        wait();
      }
    } catch (InterruptedException e) {}
    Object result = super.get();
    notify();
    return result;
  }

  public static void main(String args[]) {
    BoundedQueueWithGuard queue =
      new BoundedQueueWithGuard(5);
    new Producer(queue, 15).start();
    new Consumer(queue, 15).start();
  }
}
```

If there is no object to get then get() will execute a wait() which will put the invoking instance of Consumer of the wait queue.

When the Producer instance puts an entry in the buffer it will execute a notify() which will cause the instance of Consumer waiting for a buffer entry to awaken.

Interactions among Producer and Consumer Threads

Producer is Faster than Consumer

1. Producer acquires lock associated with queue and invokes put() but queue is full.

2. The wait method is invoked and the producer thread is suspended. The lock on the queue is released by the producer thread.

3. The consumer thread acquires the look for the queue and invokes the get() method. The guard of the get() method evaluates to true and an item is removed.

4. The notify method is invoked to awaken the suspended producer thread. The consumer thread completes get() and releases the lock.

5. The producer thread is awakened in the put() method of the queue, reacquires the lock and completes the put().

Interactions among Producer and Consumer Threads

Producer is Slower than Consumer

1. Consumer acquires lock associated with queue and invokes get() but queue is full.

2. The wait() method is invoked and the consumer thread is suspended. The lock on the queue is released by the consumer thread.

3. The producer thread acquires the look for the queue and invokes the put() method. The guard of the put() method evaluates to true and an item is added to the queue.

4. The notify method is invoked to awaken the suspended consumer thread. The producer thread completes put() and releases the lock.

5. The consumer thread is awakened in the get() method of the queue, reacquires the lock and completes the get().

Variations on a Theme

The lecture will include execution and display of output for a set of BoundedQueue implementations:

1. BoundedQueueWithGuard with Consumer started first.

2. BoundedQueueWithGuard with Producer started first.

3. BoundedQueueWithGuard with FastConsumer.

(In the base versions the Consumer is slower than the Producer.)

4. Annotated version of these routines which print out the wait and notify invocations.

Threads to Enhance Displays - The Blinking Ball Applet

```
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;
public class BlinkingBalls extends Applet implements
                    ActionListener {
      private Canvas display = new Canvas();
      private Button startStop = new Button("Start");
      private Ball fastBall, slowBall;
      public void init(){
            display.setSize(200,100);
            add(display);
            add(startStop);
            setSize(250,200);
            startStop.addActionListener(this);
      } // init()
```

```
public void actionPerformed(ActionEvent e){
             if (startStop.getLabel() == "Start") {
                    fastBall = new Ball(display, 10,
                    10, 1);
                    fastBall.start();
                    slowBall = new Ball(display, 100,
                    10, 5);
                    slowBall.start();
                    startStop.setLabel("Stop");
             }
             else {
                    fastBall.stop();
                    slowBall.stop();
                    startStop.setLabel("Start");
                    display.repaint();
             }
       } // processAction

} // class BlinkingBalls
```

```
import java.awt.*;
public class Ball extends Thread {
        private Graphics g;
        private int x, y;
        private int speed;
        private Color[] colors  =
                {Color.red, Color.green, Color.blue};
        private int nextColor = 0;
public Ball(Canvas c, int x, int y, int speed) {
                this.x = x;
                this.y = y;
                this.speed = speed;
                g = c.getGraphics();
  }
```

```
public void run() {
      drawNextFrame(); // draw one initially.
       while (true)
       { try { sleep(speed*1000);}
       catch(InterruptedException e) {};
       drawNextFrame();
       }
   }

  public void drawNextFrame(){

    g.setColor(colors[nextColor]);
    g.fillOval(x,y,50,50);
    nextColor = (nextColor + 1) % colors.length;
  }

} // class Ball
```

```
<HTML>
<HEAD>
<TITLE> HTML For Blinking Balls Applet
</TITLE>
</HEAD>
<BODY>
<applet code="BlinkingBalls.class"
width=300 height=200>
</applet>
</BODY>
</HTML>
```