Weeks 2/3
Concurrency and Multithreading

Concurrency in Distributed Systems

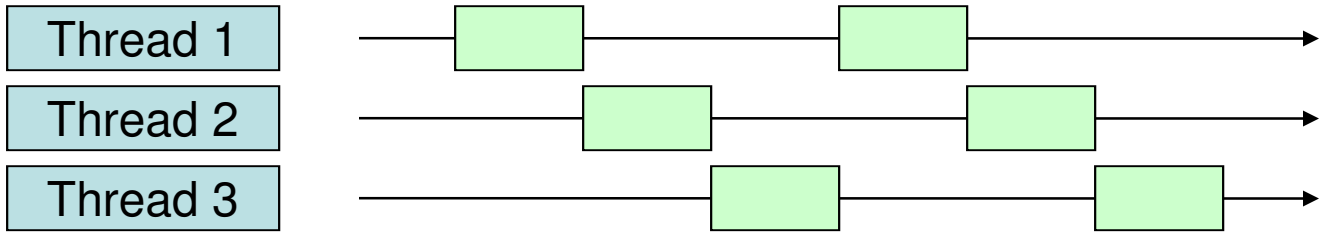
- In a distributed system a number of activities are carried out at the same time.
- A number of clients may request a server for a particular service at the same time.
- Distributed system can be viewed as a set of concurrent processes running on a number of different machines.
- In this section we will study concurrency issues in client server applications and how Java facilities can be used to create and control concurrent operations.

What is a Thread ?

A thread is the flow of execution of a task in a program. Java allows you to launch multiple threads concurrently.

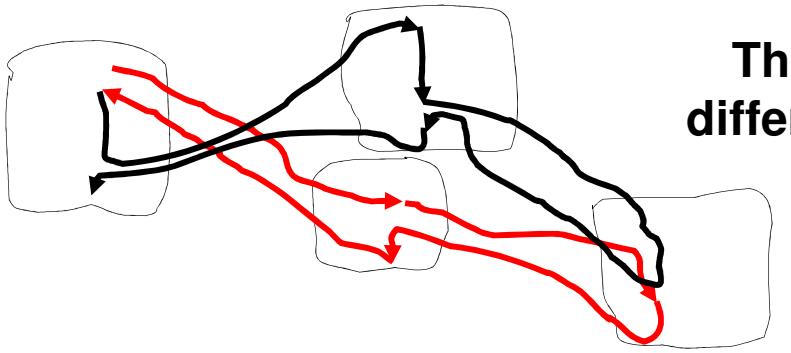
In systems with many processors threads executed simultaneously

In single processor systems threads share CPU time:



What are multiple threads?

- More than one stream of program execution running in parallel or concurrently.
- CPU executes a list of statements in one thread before switching to another and then back again until that thread is complete.
- Along the way threads may call various methods from different objects.
- Two threads may attempt to access the same method at once causing unexpected outcome.

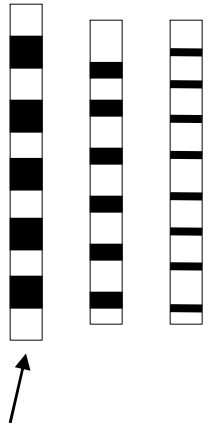


Thread paths through different objects/methods

Why use Threads?

- Exploit all available CPUs. A program with only one thread can run on at most one CPU at a time.
- Avoid idle CPU time, e.g. when waiting for the slower devices (printer) or input from human.
- Many applications (such as web-servers) need to serve more than one client at a time. E.g. Java allows a server program to start a new thread for each new client.
- Some tasks (such as garbage collection of unused memory) should be executed in a low-priority thread. The higher the priority the greater the amount of CPU time allocated.

Multiple threads sharing CPU time doing multiple tasks



High priority thread

Risks of Using Threads

- **Performance Hazard:** program performs slowly due to context switch, poor multithread design etc.
- **Liveness Hazard:** program is unable to make progress due to problems such as deadlock (more later)
- **Safety Hazard:** program behaves incorrectly under multiple threads (more later)

Creating threads extending the Thread class (poor design)

- The Thread class provides constructors and methods for creating and controlling the threads.
- The run() method of Thread class specifies what actions to be taken when a thread is started.
- Hence the user creating a thread must derive a subclass of Thread overriding the run() method.
- This class has a method named start() which creates a ***actual thread*** and calls its run() method
- Better design on next slide.

Creating threads by implementing Runnable

- The Runnable interface declares a run() method.
- To create a thread, you have to:
  - Create a RunnableTask class that implements Runnable, thus provide the implementation for the run() method.
  - Create a Thread object by passing the RunnableTask object into its constructor.
  - Invoke the start() method of the Thread object.

Sample Program using Thread class

- The program below has 2 independent threads. The first thread prints numbers from 10000 to 10100 in steps of 1. The second thread prints numbers from 20000 to 20100.
- The cpu has taken turn to print the two sequences

1st thread printing 10xxx

2nd thread printing 20xxx

```
 10000 10001 10002 10003 10004 10005 10006 10007 10008 10009 10010 10011 10012 10013 10014 10015
10016 10017 10018 10019 10020 10021 10022 10023 10024 10025 10026 10027 10028 10029 10030 20000 10031
20001 10032 20002 10033 20003 10034 20004 10035 20005 10036 20006 10037 20007 10038 20008 10039 20009
10040 20010 10041 20011 10042 20012 10043 20013 10044 20014 10045 20015 10046 20016 10047 20017 10048
20018 10049 20019 10050 20020 10051 20021 10052 20022 10053 20023 10054 20024 10055 20025 10056 20026
10057 20027 10058 20028 10059 20029 10060 20030 10061 20031 10062 20032 10063 20033 10064 20034 10065
20035 10066 20036 10067 20037 20038 20039 20040 20041 20042 20043 20044 20045 10068 20046 10069 20047
10070 20048 10071 20049 10072 20050 10073 20051 10074 20052 10075 20053 10076 20054 10077 20055 10078
20056 10079 20057 10080 20058 10081 20059 10082 20060 10083 20061 10084 20062 10085 20063 10086 20064
10087 20065 10088 20066 10089 20067 10090 20068 10091 20069 10092 20070 10093 20071 10094 20072 10095
20073 10096 20074 10097 20075 10098 20076 10099 20077 10100 20078 20079 20080 20081 20082 20083 20084
20085 20086 20087 20088 20089 20090 20091 20092 20093 20094 20095 20096 20097 20098 20099 20100
```

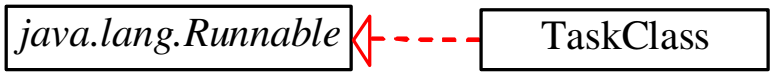
```
public class TaskThreadDemo {
  public static void main(String[] args) {
    // Create tasks
    Runnable print1 = new PrintNum(10000);
    Runnable print2 = new PrintNum(20000);

    // Create threads
    Thread thread1 = new Thread(print1);
    Thread thread2 = new Thread(print2);

    // Start threads
    thread1.start();
    thread2.start();
  }
}
```

```
// The task class for printing 100 numbers, starting from initial
class PrintNum implements Runnable {
  private int initial;


  public PrintNum(int n) {
    initial = n;
  }

  /** Tell the thread how to run */
  public void run() {
    for (int i = initial; i <= initial + 100; i++) {
      System.out.print(" " + i);
    }
  }
}
```

Program 1 Features

- The PrintNum class implements Runnable interface.
- This class has to implement the run() method declared in Runnable interface.
- The run() method is automatically invoked by the JVM. You should not invoke it.
  - What if you still invoke run()?
- The main method has created 2 instances of Thread, each has its own Runnable object.
- Calling the start() method of these instances cause a new thread to be created in JVM which calls the run() method resulting in the 2 sequences being printed.

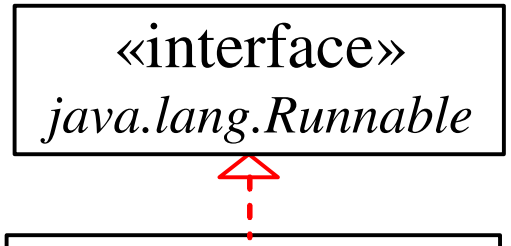
Creating Tasks and Threads




```
// Custom task class
public class TaskClass implements Runnable {
  ...
  public TaskClass(...) {
    ...
  }

  // Implement the run method in Runnable
  public void run() {
    // Tell system how to run custom thread
    ...
  }
  ...
}
```

```
// Client class
public class Client {
  ...
  public void someMethod() {
    ...
    // Create an instance of TaskClass
    TaskClass task = new TaskClass(...);

    // Create a thread
    Thread thread = new Thread(task);

    // Start a thread
    thread.start();
    ...
  }
  ...
}
```

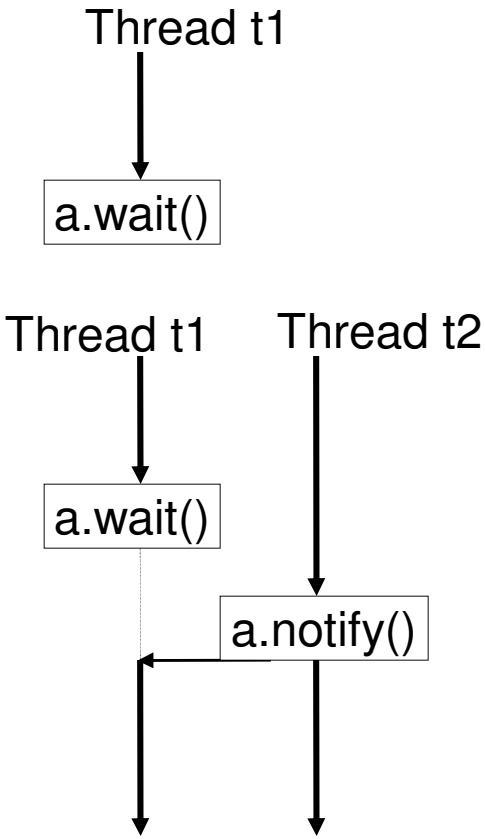
The Thread class



java.lang.Thread	
+Thread()	Creates a default thread.
+Thread(task: Runnable)	Creates a thread for a specified task.
+start(): void	Starts the thread that causes the run() method to be invoked by the JVM.
+isAlive(): boolean	Tests whether the thread is currently running.
+setPriority(p: int): void	Sets priority p (ranging from 1 to 10) for this thread.
+join(): void	Waits for this thread to finish.
+sleep(millis: long): void	Puts the runnable object to sleep for a specified time in milliseconds.
+yield(): void	Causes this thread to temporarily pause and allow other threads to execute.
+interrupt(): void	Interrupts this thread.

Object (class) methods used in thread communications

- public final void wait()
  - Forces thread to wait until notify() or notifyAll() called on the same object
- public final void notify()
  - Awakens one of the waiting threads
- public final void notifyAll()
  - Awakens all waiting threads
- Must be called in a synchronized method or block on the receiving object.
- Methods wait(), notify(), notifyAll() are part of Object class.



Quiz: What will be the output? Why?

```
public class TaskThreadDemo {
  public static void main(String[] args) {
    Runnable print1 = new PrintNum(10000);
    Runnable print2 = new PrintNum(20000);
    Thread thread1 = new Thread(print1);
    Thread thread2 = new Thread(print2);

    // Start threads
    thread1.start();
    thread2.start();
  }
}

// The task class for printing 100 numbers, starting from initial
class PrintNum implements Runnable {
  private int initial;

  public PrintNum(int n) {
    initial = n;
  }

  /** Tell the thread how to run */
  public void run() {
    for (int i = initial; i <= initial + 100; i++) {
      System.out.print(" " + i);
      Thread.yield();
    }
  }
}
```

Quiz: What will be the output? Why ?

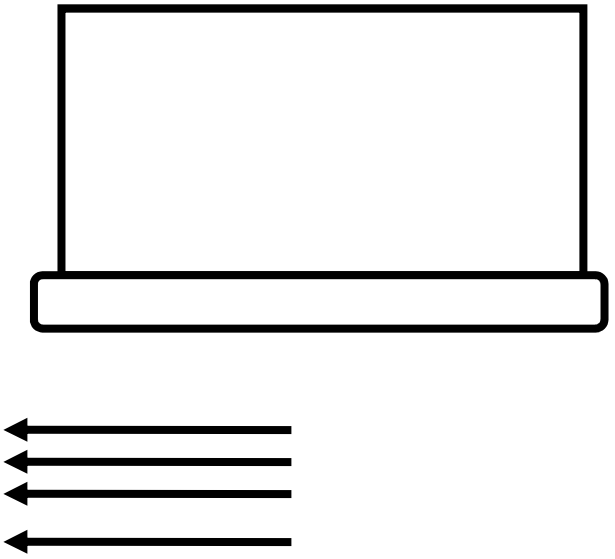
```
public class TaskThreadDemo {
  public static void main(String[] args) {
    Runnable print1 = new PrintNum(10000);
    Runnable print2 = new PrintNum(20000);
    Thread thread1 = new Thread(print1);
    Thread thread2 = new Thread(print2);
    thread1.setPriority(10);

    // Start threads
    thread1.start();
    thread2.start();
  }
}

// The task class for printing 100 numbers, starting from initial
class PrintNum implements Runnable {
  private int initial;

  public PrintNum(int n) {
    initial = n;
  }

  /** Tell the thread how to run */
  public void run() {
    for (int i = initial; i <= initial + 100; i++) {
      System.out.print(" " + i);
    }
  }
}
```

Quiz: What will be the output? Why ?

```
public class TaskThreadDemo {
  public static void main(String[] args) {

    Runnable print1 = new PrintNum(10000, 10050);
    Runnable print2 = new PrintNum(20000);

    Thread thread1 = new Thread(print1);
    Thread thread2 = new Thread(print2);

    thread1.start();
    thread2.start();
  }
}

// The task class for printing 100 numbers, starting from initial
class PrintNum implements Runnable {
  private int initial, num=-1;

  public PrintNum(int init) { initial = init; }
  public PrintNum(int init, int n) { initial = init; num=n; }

  /** Tell the thread how to run */
  public void run() {
    for (int i = initial; i <= initial + 100; i++) {
      System.out.print(" " + i);
      // try - catch Needed when calling sleep
      try {  if ( i == num ) Thread.sleep(10);   }
      catch (InterruptedException ex) {  }
    }
  }
}
```

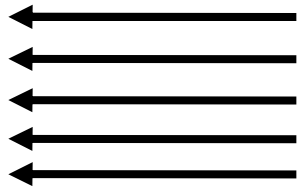
Quiz: What will be the output ? Why ?

```
public class TaskThreadDemo {
  public static void main(String[] args) {
    // Create tasks
    Runnable print1 = new PrintNum(10000);
    Runnable print2 = new PrintNum(20000);

    // Create threads
    Thread thread1 = new Thread(print1);
    Thread thread2 = new Thread(print2);

    thread1.start();
    thread2.start();
    System.out.println("In Main Before loop");
    while (thread1.isAlive() || thread2.isAlive())
        ;
    System.out.println("In main After loop");
  }
}


// The task class for printing 100 numbers, starting from initial
class PrintNum implements Runnable {
  private int initial;

  public PrintNum(int init) { initial = init; }

  /** Tell the thread how to run */
  public void run() {
    for (int i = initial; i <= initial + 100; i++) {
      System.out.print(" " + i);
    }
  }
}
```

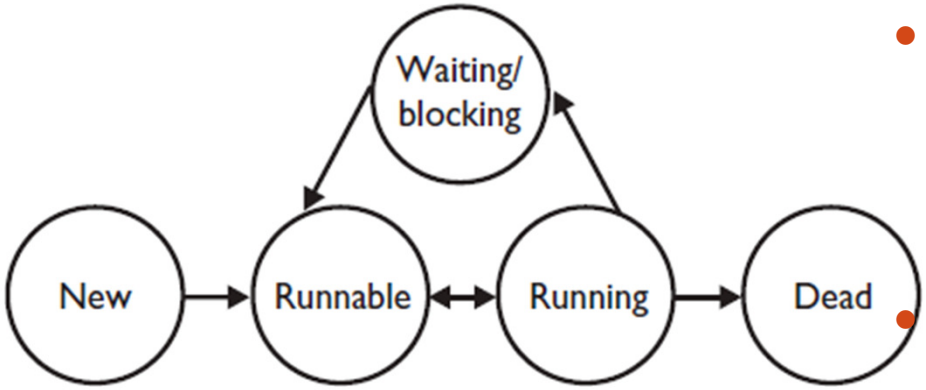


Quiz: What will be the output? Why?

```
public class TaskThreadDemoJoin {
    private Thread t1 = new Thread(new PrintNum(10000));
    private Thread t2 = new Thread(new PrintNum(20000));
    public TaskThreadDemoJoin() {
        t1.start();
        t2.start();
    }

    public static void main(String[] args) { new TaskThreadDemoJoin(); }

    // The task class for printing 100 numbers, starting from initial
    class PrintNum implements Runnable {
        private int initial, num=-1;

        public PrintNum(int init) { initial = init; }
        public PrintNum(int init, int n) { initial = init; num=n; }

        /** Tell the thread how to run */
        public void run() {
            for (int i = initial; i <= initial + 100; i++) {
                System.out.print(" " + i);
                try {
                  if ( i == initial+50 && Thread.currentThread() == t1)
                     t2.join();
                }
                catch ( InterruptedException ex){}
            }
        }
    }
}
```



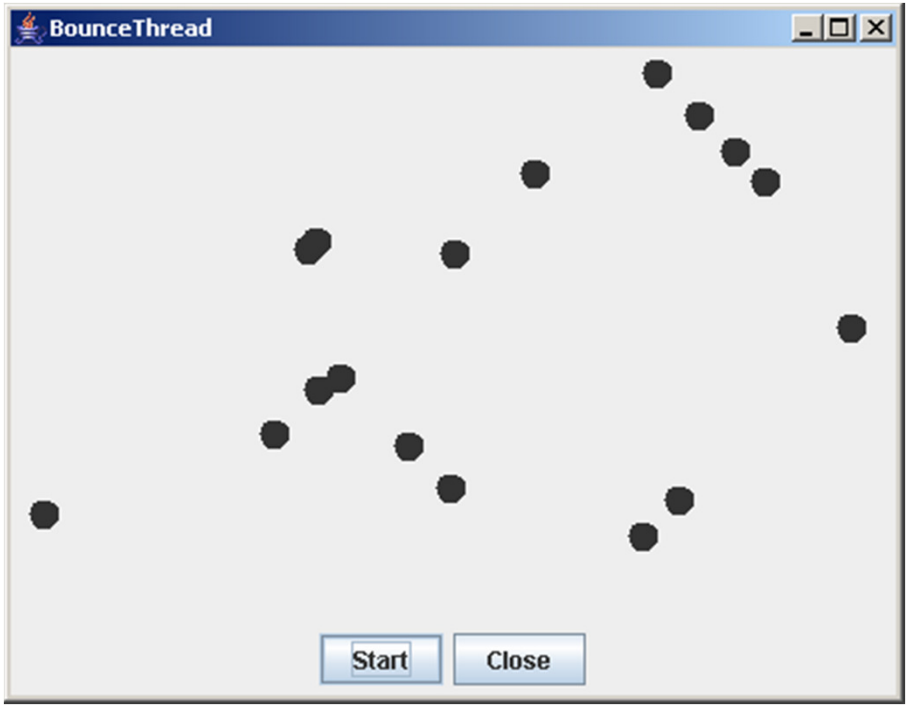
Thread States



- When a thread is created it is placed in the **New** state
- When the thread is started (via start() method) it is placed in the **Runnable** state
- A thread enters the **Running** state when CPU time is allocated to it. It returns to **Runnable** state when CPU time expires or the thread calls the yield().
- The thread enters the **Waiting/Sleeping/Blocking** state when it invokes wait() on a lock, sleeps, or is blocked on IO/lock. It enters the **Runnable** state when notified/timeout/IO-finish respectively.
- When a thread is finished it enters the **Dead** state

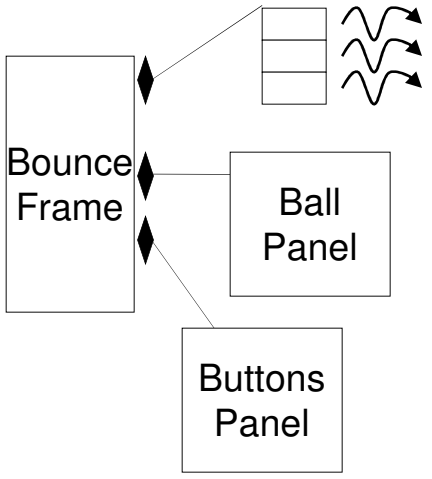
A GUI Based Application using Threads

- In this application each ball bounces within a rectangular area using a separate thread
- The Start button adds a new ball



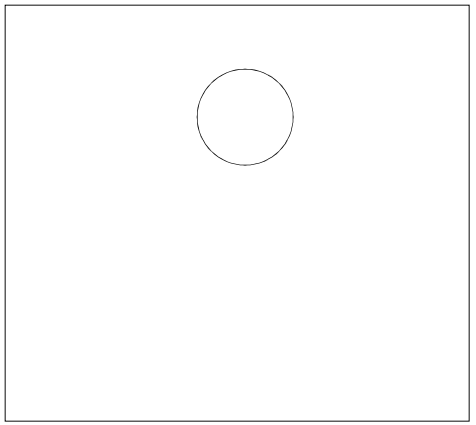
Classes and their functionality

- **BounceThread**
  - This class has the main method which constructs a BounceFrame object
- **BounceFrame**
  - Frame derived class stores a reference to the panel containing balls (BallPanel) and a reference to array of Threads created.
  - also contains a panel containing all the buttons (Start, Close). When the Start button is pressed its event handler constructs a new Ball object and add its reference to BallPanel object.
  - Next it constructs and starts a new Thread using the BallRunnable interface which takes a reference to the BallPanel and the Ball objects.



Classes and their functionality

- **BallRunnable**
  - The run method of the thread continuously updates the position of the Ball object using its move() method and repaints the BallPanel
- **BallPanel**
  - The paintComponent() method of this class draws all the balls
- **Ball**
  - The move() method of this class updates the position of the ball by dx and dy (either 1 or -1) in x and y directions respectively.



```
/* adapted from the book Core Java 2 */
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import java.util.*;
import javax.swing.*;

/** Shows an animated bouncing ball */

public class BounceThread {
    public static void main(String args[]) {
        JFrame frame = new BounceFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}


/** A runnable that animates a bouncing ball */
class BallRunnable implements Runnable {
    public BallRunnable(Ball aBall, Component aComponent) {
        ball = aBall;
        component = aComponent;
    }

    public void run() {
        try {
            while(true) {
                ball.move(component.getBounds());

                component.repaint();
                Thread.sleep(DELAY);
            }
        } catch (InterruptedException e){       }
    }

    private Ball ball;
    private Component component;
    public static final int DELAY = 5;
}
```

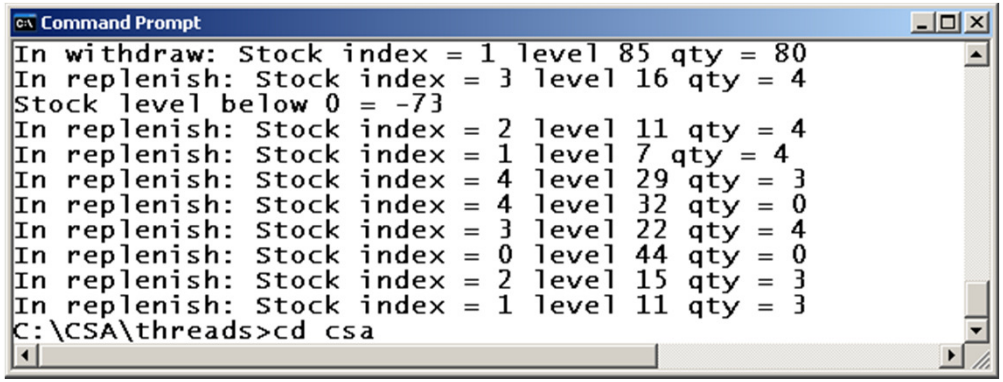
```
/* A ball that moves and bounces off the edges of a rectangle  */
class Ball {
    /* Moves the ball reversing when hitting the edge */
    public void move(Rectangle bounds) {
        x += dx;
        y += dy;
        if ( x < bounds.getMinX()) {
            x = bounds.getMinX();
            dx = -dx;
        }
        if ( x + XSIZE >= bounds.getMaxX()) {
            x = bounds.getMaxX() - XSIZE;
            dx = -dx;
        }
        if ( y < bounds.getMinY()) {
            y = bounds.getMinY();
            dy = -dy;
        }
        if ( y + YSIZE >= bounds.getMaxY()) {
            y = bounds.getMaxY() - YSIZE;
            dy = -dy;
        }
    }

    /* Get shape of the ball at its current position.   */
    public Ellipse2D getShape() {
        return new Ellipse2D.Double(x,y,XSIZE, YSIZE);
    }

    private static final int XSIZE = 15;
    private static final int YSIZE = 15;
    private double x = 0;
    private double y = 0;
    private double dx = 1;
    private double dy = 1;
}
```

```
/* The panel that draws the balls */
class BallPanel extends JPanel {
    /* Add a ball to the panel   */
    public void add(Ball b) {
        balls.add(b);
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D) g;
        for (int i=0; i<balls.size(); i++) {
            g2.fill(((Ball)balls.get(i)).getShape());
        }
    }
    private ArrayList balls = new ArrayList();
}
```

```
/* the frame with panel and buttons   */
class BounceFrame extends JFrame {
    private static int n;
    public BounceFrame() {
        setSize(WIDTH, HEIGHT);
        setTitle("BounceThread");
        panel = new BallPanel();
        add(panel, BorderLayout.CENTER);
        JPanel buttonPanel = new JPanel();
        addButton(buttonPanel, "Start", new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                addBall();
            }
        }
        );
        addButton(buttonPanel, "Close", new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                System.exit(0); }
        }
        );
        add(buttonPanel, BorderLayout.SOUTH);
    }

    public void addButton(Container c, String title, ActionListener listener) {
        JButton button = new JButton(title);
        c.add(button);
        button.addActionListener(listener);
    }

    /* Adds a bouncing ball to a canvas and starts a thread to make it bounce */
    public void addBall() {
        Ball b = new Ball();
        panel.add(b);
        Runnable r = new BallRunnable(b,panel);
        Thread t = new Thread(r);
        n++;
        t.setPriority(n);
        t.start();
        threads.add(t);
    }

    private ArrayList threads = new ArrayList();
    private BallPanel panel;
    public static final int WIDTH = 500;
    public static final int HEIGHT = 400;
}
```

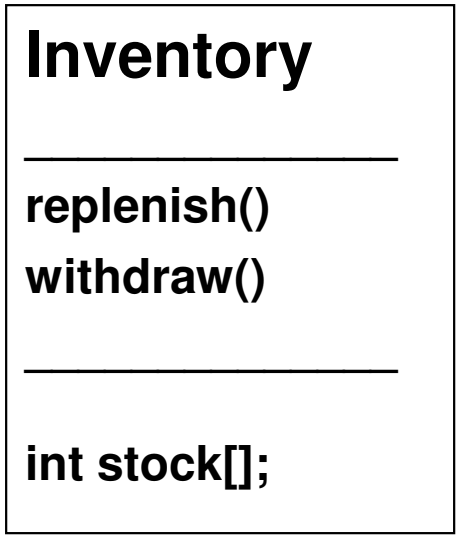
Thread Synchronization & Race Condition

- Race condition: the situation when the correct behavior of the application depends on the relative timing or interleaving of multiple threads by the runtime.
- A class is thread-safe if it can be accessed by multiple threads without resulting any race condition while requiring no external synchronization from client-code.
- In the following program an Inventory object storing the stock levels of 5 different items is accessed by 20 different threads. Out of these 10 threads are used for replenishing and 10 for withdrawing. Withdrawing is allowed only if sufficient parts are available.
- However, the next program aborts after running for a short time with stock level falling below 0, caused by race condition.



Sample Unsynchronized Program to Demo Race Condition

- Inventory
  - The constructor initializes the stock levels for all 5 items.
  - The replenish() method takes the index and qty of the stock being replenished, adding the specified qty to the appropriate stock level.
  - The withdraw() method takes the same arguments and does the opposite.
    - If however, the required qty for a given part is not available it wait until sufficient parts are replenished.
    - This method also prints the final stock level after withdrawing and aborts the program if it is below 0.



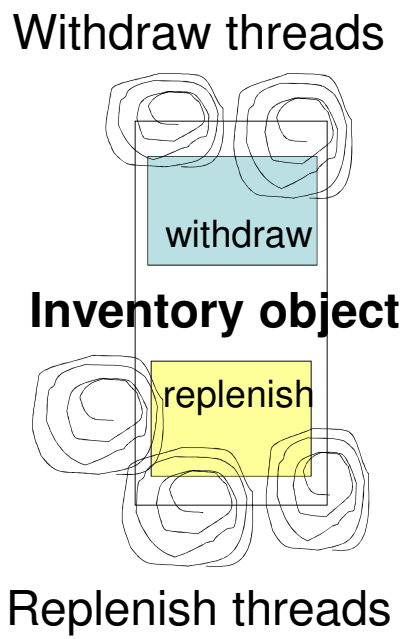
- ReplenishRunnable, WithdrawRunnable

  The constructors for these classes take references to the Inventory object and the index of the part to be replenished.

  In the run() method it repeatedly calls the replenish method or withdraw() method respectively after short intervals with randomly chosen values.

- ManipulateInventory

  The constructor creates an Inventory object with 5 stock items all with initial level of 100.
  Then it creates 10 threads for replenishing and 10 threads for withdrawing and set them in motion (by calling start()) to update the Inventory object created.
  Note, all stock items will be updated by 4 different threads at any one time.



```
public class ManipulateInventory {
    public static void main(String args[]) {
        ManipulateInventory mI = new ManipulateInventory();
    }

    public ManipulateInventory() {
        // Creating an inventory of 5 items all with 1000 items
        Inventory inv = new Inventory(5,100);

        // We create 10 threads each for withdrawing
        Thread w[] = new Thread[10];
        // and 10 threads for replenishing
        Thread r[] = new Thread[10];

        // The 20 threads will be updating one of the 5 stock items
        for (int i=0; i<10; i++) {
            w[i] = new Thread( new WithdrawRunnable(inv,i%5));
            w[i].start();
        }
        for (int i=0; i<10; i++) {
            r[i] = new Thread(new ReplenishRunnable(inv,i%5));
            r[i].start();
        }
    }
```

```
public class ReplenishRunnable implements Runnable {
    private Inventory inv;
    private int index;
    /* Constructor takes a reference to Inventory object
     and the index of the stock item it will update */
    public ReplenishRunnable(Inventory inv,int index) {
        this.inv = inv;
        this.index = index;
    }
    public void run() {
        while (true) {
            int qty = (int) (5 * Math.random());  // qty = 1- 5
            inv.replenish(index,qty);
            try {
                Thread.sleep((int) (10 * Math.random())); // sleep 1 - 10 ms
            } catch (InterruptedException ex) {
            }
        }
    }
}
public class WithdrawRunnable implements Runnable {
    private Inventory inv;
    private int index;
    /* Constructor takes a reference to Inventory object
     and the index of the stock item it will update */
    public WithdrawRunnable(Inventory inv,int index) {
        this.inv = inv;
        this.index = index;
    }
    public void run() {
        while (true) {
            int qty = (int) (100 * Math.random());  // qty = 1 - 100
            try {
                inv.withdraw(index,qty);
                Thread.sleep((int) (5 * Math.random())); // sleep 1 - 5 ms
            } catch (InterruptedException ex) {
            }
        }
    }
}
```

```
/* class for managing inventory of items */
class Inventory {
  private int stock[];  // current stock levels

  /* Allows the user to specify the number of different items
     and their current stock levels                          */
  public Inventory(int num, int initial) {
    stock = new int[num];
    for (int i=0; i<num; i++)
      stock[i] = initial;
  }

  /* specify the index of stock item and quantity to be replenished */
  public void replenish(int index, int qty) {
    System.out.println("In replenish: Stock index = " +
                    index+ " level "+ stock[index] + " qty = " + qty);
    int amount = stock[index];
    amount = amount + qty;
    stock[index] = amount;
  }

  /* specify the index of stock item and quantity to be withdrawn */
  public void withdraw(int index, int qty) throws InterruptedException {
    while ( stock[index] < qty)
      ;     // delay until stock become available

    // print current stock level and qty
    System.out.println("In withdraw: Stock index = "+index +
                   " level "+ stock[index] + " qty = " + qty);
    int amount = stock[index] - qty;
    if (amount < 0) {
      System.out.println("Stock level below 0 = " + amount);
      System.exit(1);
    }
    stock[index] = amount;
  }
 }
}
```

Race Condition Explained

Thread A
Index = 2 qty = 100

```
public void withdraw
   (int index, int qty)
{
1 while (stock[index]<qty)
     ; // until available
2 System.out.println(…);

3 int amount=stock[index]-qty;
  …
4 stock[index] = amount;
}
```

Thread B
Index = 2 qty = 80

```
public void replenish
     (int index, int qty)
{
1  int amount = stock[index];
2  amount = amount + qty;
3  stock[index] = amount;
}
```

Thread C
Index = 2 qty = 60

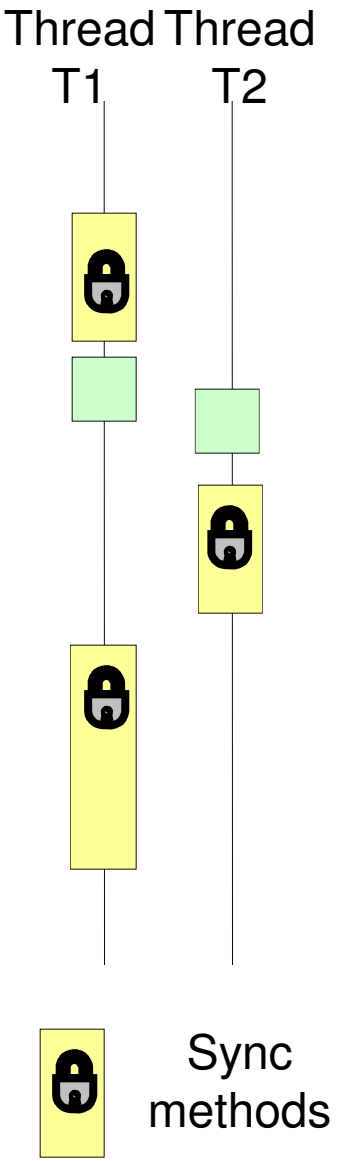
```
public void withdraw
   (int index, int qty)
{
1 while (stock[index]<qty)
     ; // until available
2 System.out.println(…);

3 int amount=stock[index]-qty;
  …
4 stock[index] = amount;
}
```

Assume that the initial stock level was at 50. A possible sequence of instructions resulting from three threads (2 WithdrawThread and 1 ReplenishThread) is shown next. This causes the stock[2] to fall below 0 resulting in an error state.

Order	stock[2]	
A1	50	
B1	50	
B2	50	
B3	130	
C1	130	
C2	130	
A1	130	
A2	130	
A3	130	
A4	30	
C3	-30	Error

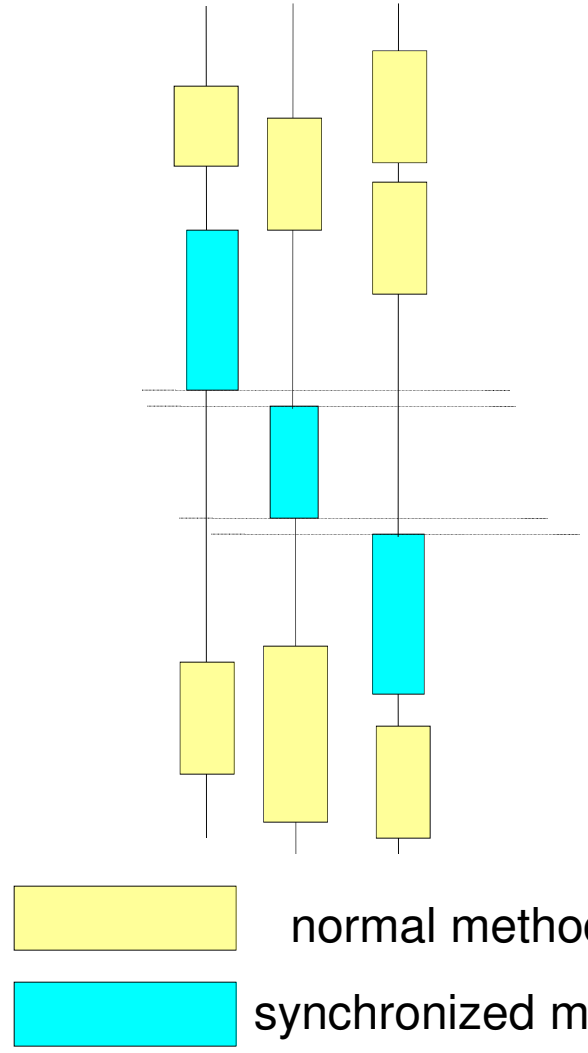
Synchronizing Statements

- To avoid race conditions we must avoid more than one thread from executing certain critical code.
- To disallow more than one thread accessing replenish() or withdraw() of Inventory object at the same time, we can add the **synchronized** keyword to these methods.
- Synchronized method acquires a lock before execution.
- Once a thread obtains a lock on an object (in our case Inventory object) no other thread can start any synchronized methods using the same lock, until the lock is released on completion of the synchronized method.
- Synchronized statement can also be used to get a lock on a class (using static method), on current instance or any object (not necessarily the instance of the class containing the method)



Synchronized Inventory Object

```
/* specify the index of stock item and quantity to be replenished */
    public synchronized void replenish(int index, int qty)
    {
        System.out.println("In replenish: Stock index = " +
              index+ " level "+ stock[index] + " qty = " + qty);
        int amount = stock[index];
        amount = amount + qty;
        stock[index] = amount;
    }

    /* specify the index of stock item and quantity to be withdrawn */
    public synchronized void withdraw(int index, int qty)
    {
       while ( stock[index] < qty)
          ;     // delay until stock become avaialble

       // print current stock level and qty
       System.out.println("In withdraw: Stock index = "+index +
                    " level "+ stock[index] + " qty = " + qty);
       int amount = stock[index] - qty;
       if (amount < 0)
       {
          System.out.println("Stock level below 0 = " + amount);
          System.exit(1);
       }
       stock[index] = amount;
    }
```



The need for Thread Cooperation

- If we run our Inventory application with synchronized withdraw and deposit methods we can avoid the race condition. However, for the threads to cooperate properly we need to use the methods wait(), notify() and notifyAll().

- Without cooperation, Withdraw thread starting the synchronized withdraw() method could end up waiting indefinitely for sufficient parts to become available. But the Replenish thread cannot proceed with the synchronized replenish method until withdraw() is complete because the Withdraw thread is holding the lock to the Inventory object.

A WithdrawThread

```
public synchronized void
    withdraw(int index, int qty)
{
    while ( stock[index] < qty)
     ; // wait until available
    …
}
```

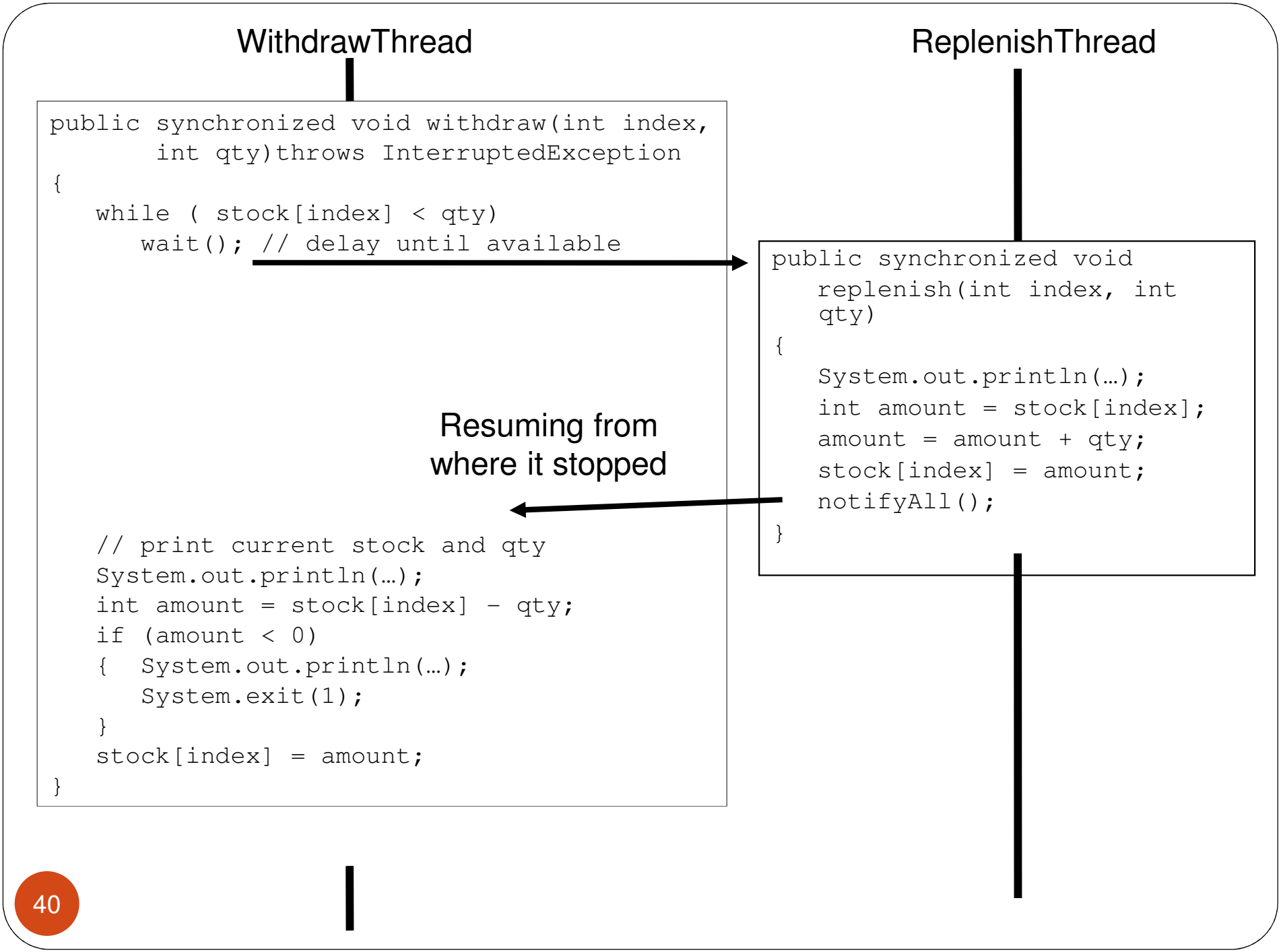
Cannot proceed

A ReplenishThread

```
public synchronized void
    replenish(int index, int qty)
{
    System.out.println(…);
    int amount = stock[index];
    amount = amount + qty;
    stock[index] = amount;
}
```

Using wait(), notify() and notifyAll()

- These methods must be called inside synchronized methods or blocks.
- The **wait**() method lets a thread wait for some condition, such as “sufficient stock” to become true. When that happen a **notify**() or **notifyAll**() method can be used to resume the waiting threads. The **notifyAll**() wakes up all waiting threads, **notify**() wakes only one of the waiting ones.
- The **wait**() method can throws **InterruptedException**
- The next slide shows the modified replenish and withdraw methods.


WithdrawThread
ReplenishThread
public synchronized void withdraw(int index,
int qty)throws InterruptedException
{
while (stock[index] < qty)
wait(); // delay until available
public synchronized void
replenish(int index, int
qty)
{
System.out.println(…);
int amount = stock[index];
amount = amount + qty;
stock[index] = amount;
notifyAll();
}
Resuming from
where it stopped
// print current stock and qty
System.out.println(…);
int amount = stock[index] - qty;
if (amount < 0)
{ System.out.println(…);
System.exit(1);
}
stock[index] = amount;
}

Block-Level Synchronization

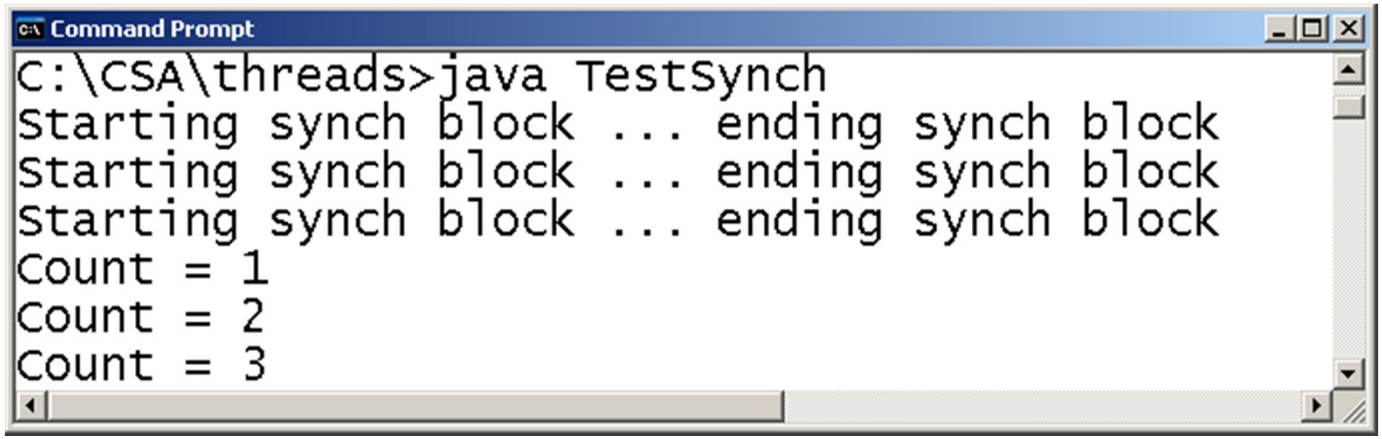
- Method level synchronization though effective cannot always be used:
  - When class used not designed to be thread-safe
- Block level synchronization allows us to place a synchronized keyword around a block of code. It is synchronized using any object as in

```
synchronized( Object o)
{
  …
}
```

- The next program uses a StringBuffer object to ensure only one thread can access count at any one time.

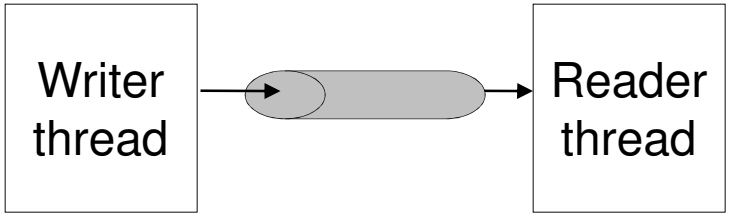
```
class SynchBlock implements Runnable {
   StringBuffer buffer;
   int counter;
   public SynchBlock(){
        buffer = new StringBuffer();
        counter = 1;
   }
   public void run(){
      synchronized(buffer){
         System.out.print("Starting synchronized block ");
         int temp = counter++;
         String message = "Count = " + temp + "\n";
         try {
             Thread.sleep(100);
         }
         catch (InterruptedException ie) {}
         buffer.append(message);
         System.out.println("... ending synch block");
      }
   }
}
```

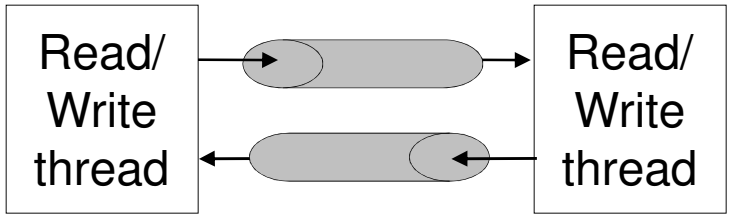
```
public class TestSynch
{  public static void main(String args[]) throws Exception
   {
      // Create a new runnable instance of SynchBlock
      SynchBlock block = new SynchBlock();
      Thread t1 = new Thread(block);
      Thread t2 = new Thread(block);
      Thread t3 = new Thread(block);
      t1.start();
      t2.start();
      t3.start();
      // wait for all three threads to finish
      t1.join(); t2.join(); t3.join();
      System.out.println(block.buffer);
   }
}
```



Inter-Thread Communication

- Two common options for thread communication are
  - wait()/notify() methods which allows a waiting thread to be notified of an event
  - Pipes
- Like multi-process communication used in OS, one thread can send data directly to another using special streams linking one thread to another.
- The sample program next creates a one-way between primary application thread and a second thread that sends text message using the pipe.
- Note that pipe is constructed before the thread is started.

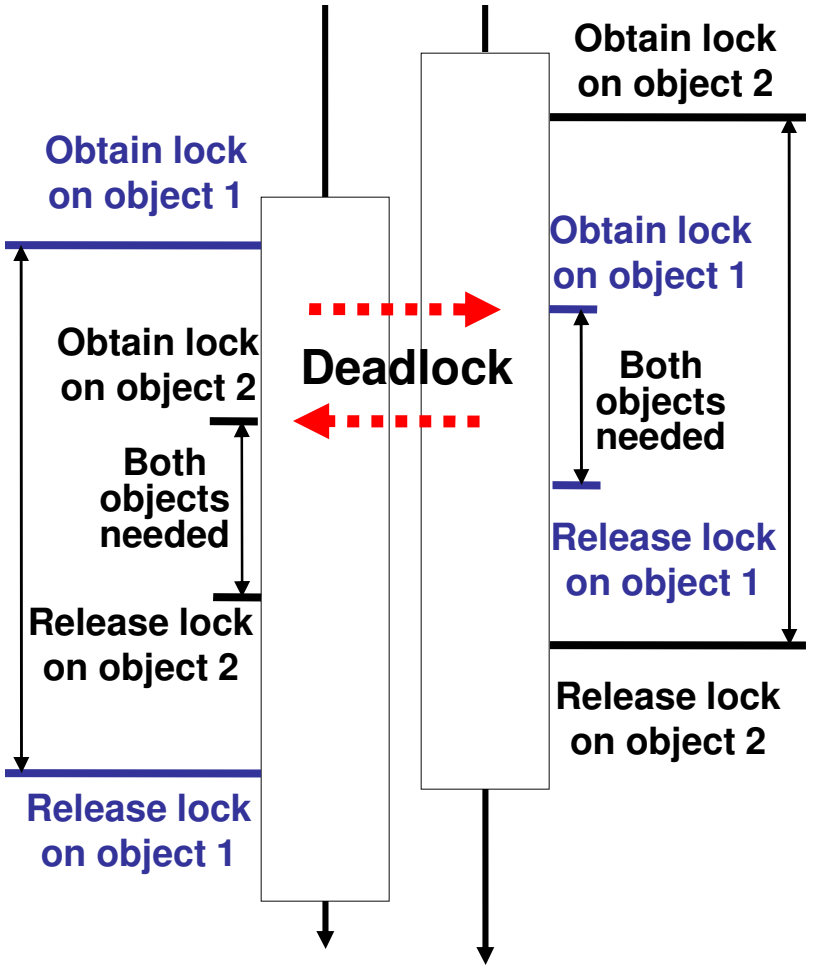




```
import java.io.*;
public class PipeDemo implements Runnable {
    PipedOutputStream output;
    // Create an instance of the PipeDemo class
    public PipeDemo(PipedOutputStream out) {   // Copy to local member variable
        output = out;
    }
    public static void main(String args[]) {
        try {
            PipedOutputStream pout = new PipedOutputStream();
            //Create a pipe for reading, and connect it to output pipe
            PipedInputStream pin = new PipedInputStream(pout);
            //create a new pipe demo thread, to write to our thread
            Thread pipeDemoThread = new Thread(new PipeDemo(pout));
            // start the thread
            pipeDemoThread.start();
            // read the thread data
            int input = pin.read();
            // Terminate when end of stream reached
            while (input != -1) {   // Print the message
                System.out.print( (char) input);
                // read the enxt byte
                input = pin.read();
            }
        }

        catch (Exception e) {
            System.err.println("Pipe error " + e);
        }
    }
    public void run() {
        try {
            // Create a printstream for convenient writing
            PrintStream p = new PrintStream(output);
            // Print message
            p.println("Hello from another thread, via pipes!");
            // close the stream
            p.close();
        } catch (Exception e) {

        }
    }
}
```

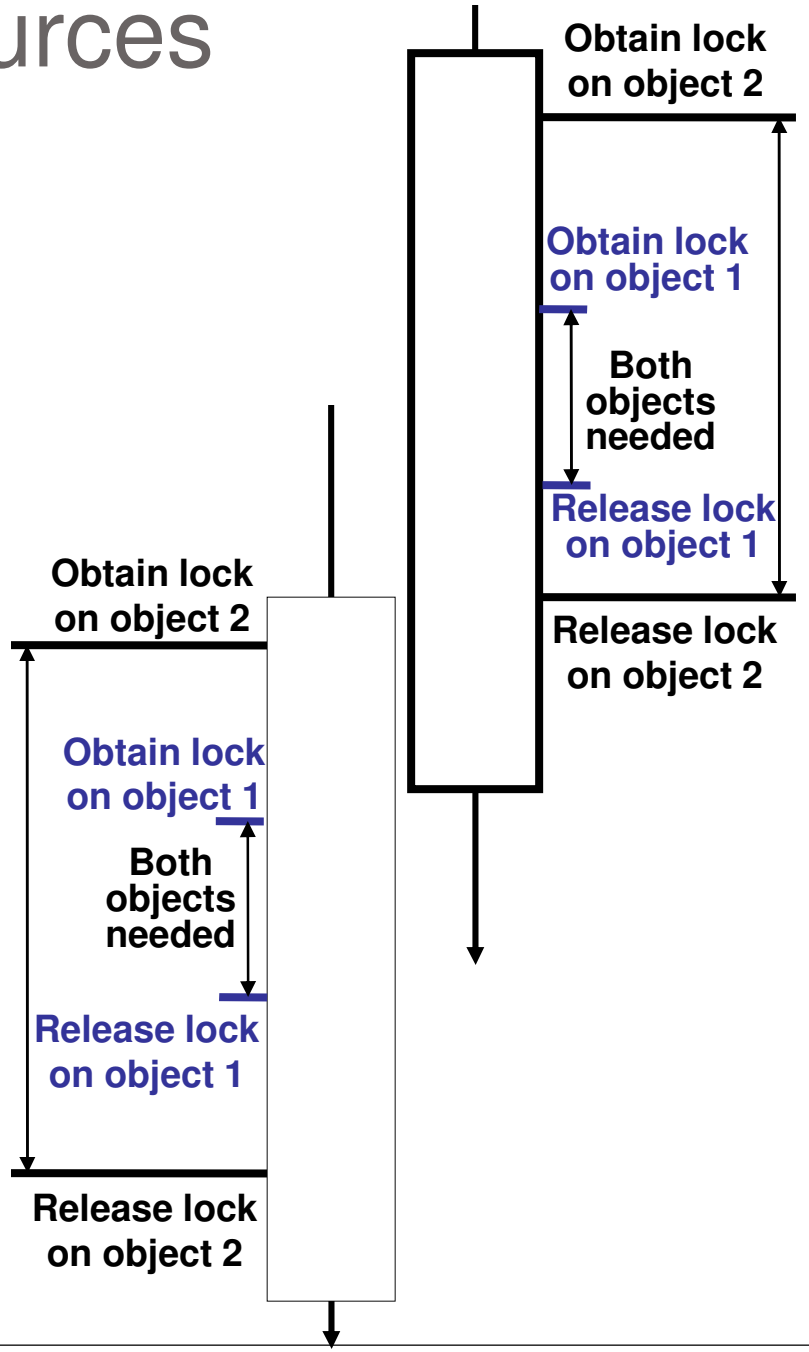
Deadlock

- When two or more threads need to acquire locks on several shared objects, a deadlock situation may arise.
- In this example thread 1 has obtained lock on object 1 and thread 2 on object 2. Now thread1 is attempting to obtain a lock on object 2, and thread 2 a lock on object1. Each thread end up waiting for the other to release – A deadlock.



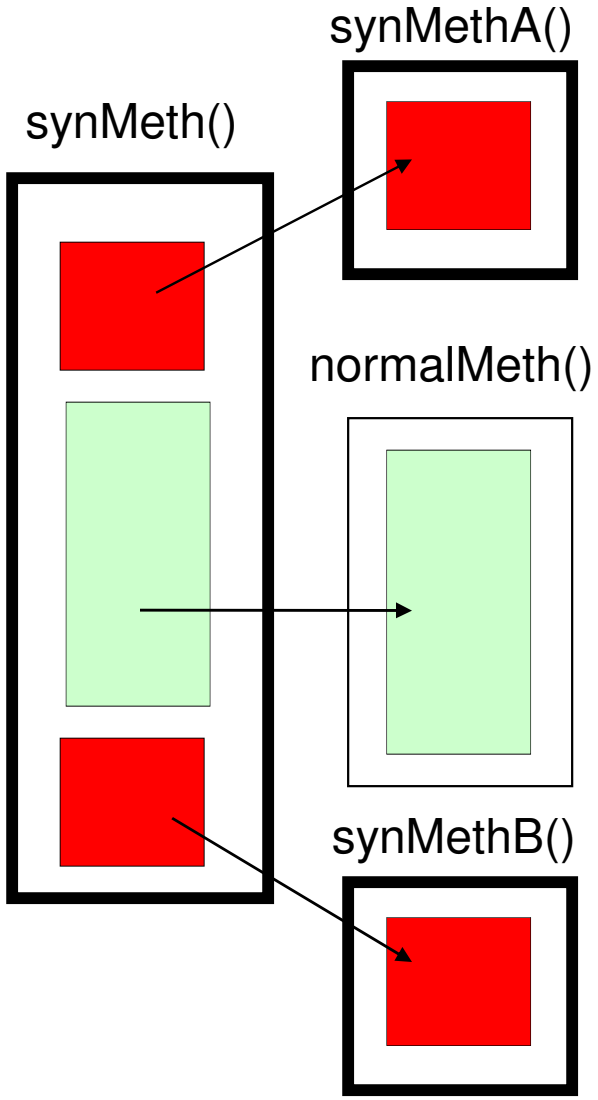
Tip #1: Reorder Resources Acquisition

- Deadlock can be avoided through resource ordering. This scheme requires locks to be obtained in the same order.
- In the example both threads obtain lock on object 2 followed by lock on object1.



Tip #2: Minimize Synchronization

- Do not synchronize longer than needed as it disallows access to all other threads.
  - Factor in the code that uses shared variables into a separate method(s) thus reducing the waiting time for other threads.



Tip #3: Use “Master Lock”

- Require a “master lock” to be acquired before acquiring any specific resource lock
- Useful when dealing with 3rd-party code when specific resource lock can’t be controlled

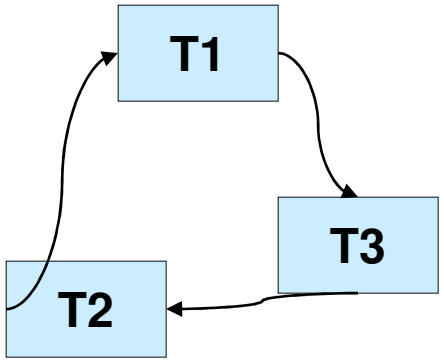
Tip #4: Use Interruptible Lock Acquisition

- Java 5's Concurrent framework offers the Lock interface and implementation like ReentrantLock

```
public interface Lock {
  void lock();
  void lockInterruptibly() throws
    InterruptedException;
  boolean tryLock();
  boolean tryLock(long timeout, TimeUnit unit)
    throws InterruptedException;
  void unlock();
  Condition newCondition();
}
```

Wait-For Graph

- A wait-for graph is a directed graph showing relationships between transactions and data
- A transaction is a set of atomic operations accessing and modifying stored data.
- The diagram shows a deadlock as transaction T1 is waiting for T3, T3 is waiting for T2 and T2 is waiting for T1 resulting in cycle.
- The wait occurs because one transaction has a lock on a resource needed by another.



Tip #5: Centralized Lock Control

- A wait for graph is updated by the lock manager based on locks granted and released by transactions.
- Each time a lock is requested lock manager examines to see if a cycle is about to be created. If so, it may abort one existing transaction.
- Detecting cycles is a well known problem.
- The transaction to be aborted may be based on:
  - How many operations in a transaction
  - How long a transaction has waited
  - How close to completion

Tip #6: Timeout Locks

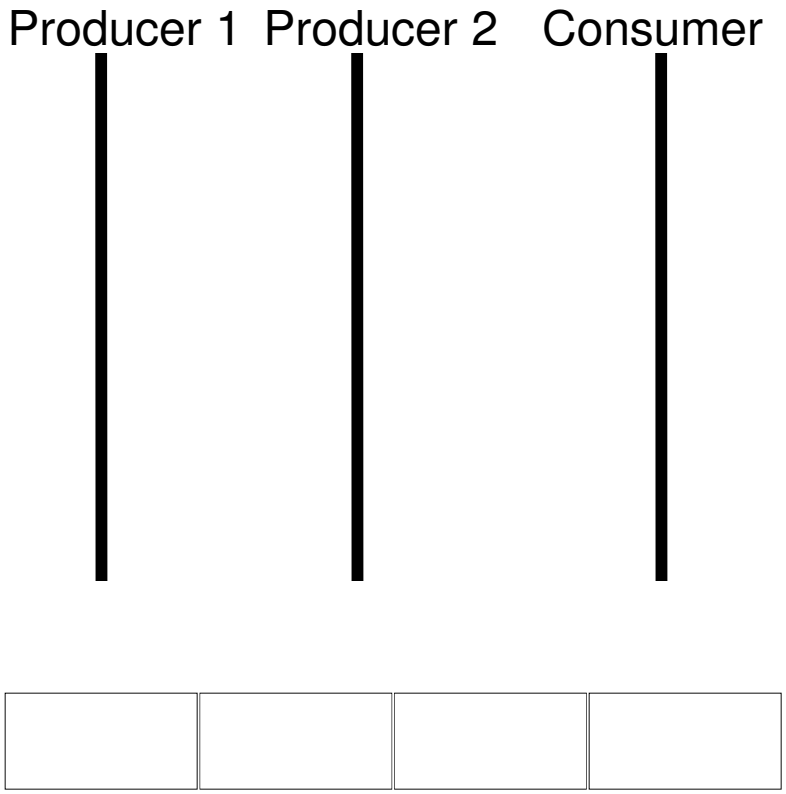
- Many database servers use timeout to eliminate deadlock.
- Each lock granted to a transaction is given a specified duration after which the transaction may be aborted if the lock is not released.
- Features of this approach:
  - Easy to implement (compared to wait-for graphs)
  - Transaction may be aborted even if no deadlock involved
  - Long running transactions have more chances to be aborted

Locking in database servers

- Most database servers obtain a read lock before reading from a table and a write lock before writing.
- Developer may not have direct control over locking
- Most database servers allow a number of lock levels:
  - Row locking
  - Page locking – page of file storage containing part of tabe
  - Table locking
  - Database locking (all the tables)

Producer / Consumer Example

- The next program creates a Queue that can store up to 10 elements.
- It then uses two producer threads to add 100 messages each to the queue and a consumer thread to retrieve them all.
- If you run this program several times you will notice that the program may not complete normally at times as well as give unexpected output.
- You will study the problem in the tutorials adding the necessary code to make the threads run without interference.



Testing the Producer / Consumer

```
//   Runs two Producer and one Consumer threads concurrently
public class ThreadTest{

   public static void main(String[] args)   {

      Queue queue = new Queue(10);
      queue.setDebug(true);
      Runnable run1 = new Producer("Hello, World!", queue, 100);
      Runnable run2 = new Producer("Goodbye, World!", queue, 100);
      Runnable run3 = new Consumer(queue, 2 * REPETITIONS);

      Thread thread1 = new Thread(run1);
      Thread thread2 = new Thread(run2);
      Thread thread3 = new Thread(run3);

      thread1.start();
      thread2.start();
      thread3.start();
   }
}
```

```
public class Queue {
private Object[] elements;
   private int head;
   private int tail;
   private int size;
   private boolean debug;
   public Queue(int capacity)
   {  elements = new Object[capacity];
      head = 0;       tail = 0;       size = 0;
   }
   // Returns the object at the head.  Assumes !isEmpty()
   public Object removeFirst()
   {
      Object r = elements[head];
      head++;
      size--;
      if (head == elements.length)
          head = 0;
      return r;
   }
   public void add(Object anObject)  // Appends object at the tail.
   {  elements[tail] = anObject;
      tail++;
      size++;
      if (tail == elements.length)
          tail = 0;
    }
   public boolean isFull()  {   return size == elements.length;   }
   public boolean isEmpty() {      return size == 0;   }

}
```

```
// repeatedly inserts a greeting into a queue.
public class Producer implements Runnable
{
   // Constructs the producer object. It takes the queue, the number of
   // repetition and greetings to insert into a queue
   public Producer(String aGreeting, Queue aQueue, int reps)
   {
      greeting = aGreeting;
      queue = aQueue;
      repetitions = reps;
   }

   public void run()
   {
      try
      {  int i = 1;
         while (i <= repetitions)
         {
            if (!queue.isFull())
            {
               queue.add(i + ": " + greeting);
               i++;
            }
            Thread.sleep((int)(Math.random() * DELAY));
         }
      }
      catch (InterruptedException exception)
      {
      }
   }
   private String greeting;
   private Queue queue;
   private int repetitions;
   private static final int DELAY = 10;
}
```

```
// repeatedly removes item from a queue
public class Consumer implements Runnable
{
   // Constructs consumer object taking the queue from which to retrieve
   // and the number of arguments
   public Consumer(Queue aQueue, int reps)
   {
      queue = aQueue;
      repetitions = reps;
   }

   public void run()
   {
      try
      {
         int i = 1;
         while (i <= repetitions)
         {
            if (!queue.isEmpty())
            {
               Object greeting = queue.removeFirst();
               System.out.println(greeting);
               i++;
            }
            Thread.sleep((int)(Math.random() * DELAY));
         }
      }
      catch (InterruptedException exception)      {      }
   }

   private Queue queue;
   private int repetitions;
   private static final int DELAY = 10;
}
```