

Java Programming

Multithreading - II

Module 8

Agenda

1

Thread Priorities

2

Thread Synchronization

3

Inter Thread Communication

Objectives

- At the end of this module, you will be able to:
 - Set Thread priorities
 - Appreciate the need for Thread Synchronization
 - Implement a java program that uses Synchronized methods
 - Implement java program for inter thread communication

Thread Priorities

Thread Priorities

- A thread priority decides:
 - The importance of a particular thread, as compared to the other threads
 - when to switch from one running thread to another
- The term that is used for switching from one thread to another is **context switch**
- Threads which have higher priority are usually executed in preference to threads that have lower priority

Thread Priorities (Contd.).

- When the thread scheduler has to pick up from several threads that are runnable, it will check the thread priority and will decide when a particular has to run
- The threads that have higher-priority usually get more CPU time as compared to lower-priority threads
- A higher priority thread can also preempt a lower priority thread
- Actually, threads of equal priority should evenly split the CPU time

Thread Priorities (Contd.).

- Every thread has a priority
- When a thread is created it inherits the priority of the thread that created it
- The methods for accessing and setting priority are as follows:
 - `public final int getPriority();`
 - `public final void setPriority (int level);`

Thread Priorities (Contd.).

- JVM selects a Runnable thread with the highest priority to run
- All Java threads have a priority in the range 1-10
- Top priority is 10, lowest priority is 1
- Normal priority i.e.. priority by default is 5
- Thread.MIN_PRIORITY - minimum thread priority
- Thread.MAX_PRIORITY - maximum thread priority
- Thread.NORM_PRIORITY - normal thread priority

Thread Priorities (Contd.).

- When a new Java thread is created it has the same priority as the thread which created it
- Thread priority can be changed by the `setPriority()` method

```
t1.setPriority(Thread.NORM_PRIORITY + 1); //priority  
t2.setPriority(Thread.NORM_PRIORITY - 1); //priority  
t3.setPriority(Thread.MAX_PRIORITY - 1); //priority
```

```
t1.start();  
t2.start();  
t3.start();
```

Deciding on a Context Switch

- A thread can voluntarily relinquish control by explicitly yielding, sleeping, or blocking on pending Input/ Output
- All threads are examined and the highest-priority thread that is ready to run is given the CPU
- A thread can be preempted by a higher priority thread
- A lower-priority thread that does not yield the processor is superseded, or preempted by a higher-priority thread

This is called preemptive multitasking.

- **When two threads with the same priority are competing for CPU time, threads are time-sliced in round-robin fashion in case of Windows like OSs**

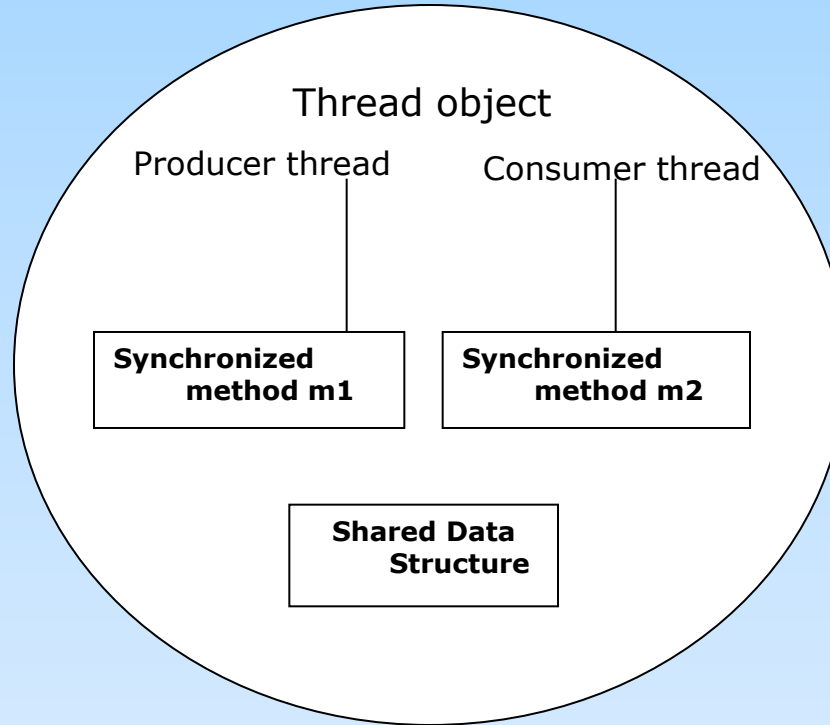
Thread Synchronization

Synchronization

- It is normal for threads to be sharing objects and data
- Different threads shouldn't try to access and change the same data at the same time
- Threads must therefore be synchronized
- For example, imagine a Java application where one thread (which let us assume as Producer) writes data to a data structure, while a second thread (consider this as Consumer) reads data from the data structure

Synchronization (Contd.).

This example use concurrent threads that share a common resource: a data structure.



Synchronization (Contd.).

- The current thread operating on the shared data structure, must be granted mutually exclusive access to the data
- The current thread gets an exclusive lock on the shared data structure, or a **mutex**
- A **mutex** is a concurrency control mechanism used to ensure the integrity of a shared data structure

Synchronization (Contd.).

- Mutex is not assured, if, the methods of the object, accessed by competing threads are ordinary methods
- It might lead to a race condition when the competing threads will race each other to complete their operation
- A **race condition** can be prevented by defining the methods accessed by the competing threads as **synchronized**

Synchronization (Contd.).

- Synchronized methods are an elegant variation on a time-tested model of interprocess-synchronization: the monitor
- The monitor is a thread control mechanism
- When a thread enters a monitor (synchronized method), all other threads, that are waiting for the monitor of same object, must wait until that thread exits the monitor
- The monitor acts as a concurrency control mechanism

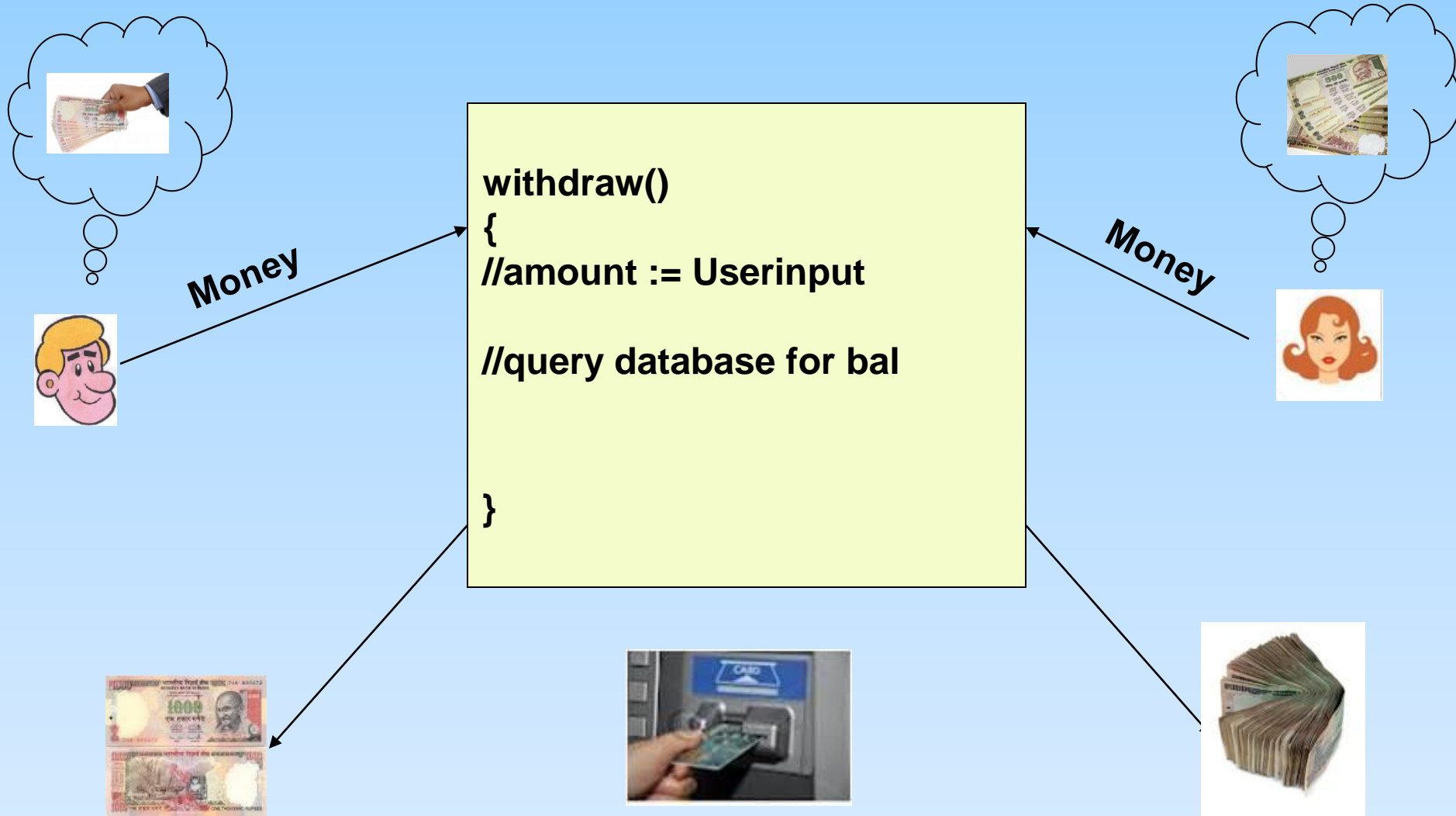
Synchronization (Contd.).

- Threads often need to share data.
- There is a need for a mechanism to ensure that the shared data will be used by only one thread at a time
- This mechanism is called synchronization.
- Key to synchronization is the concept of the monitor (also called a semaphore).

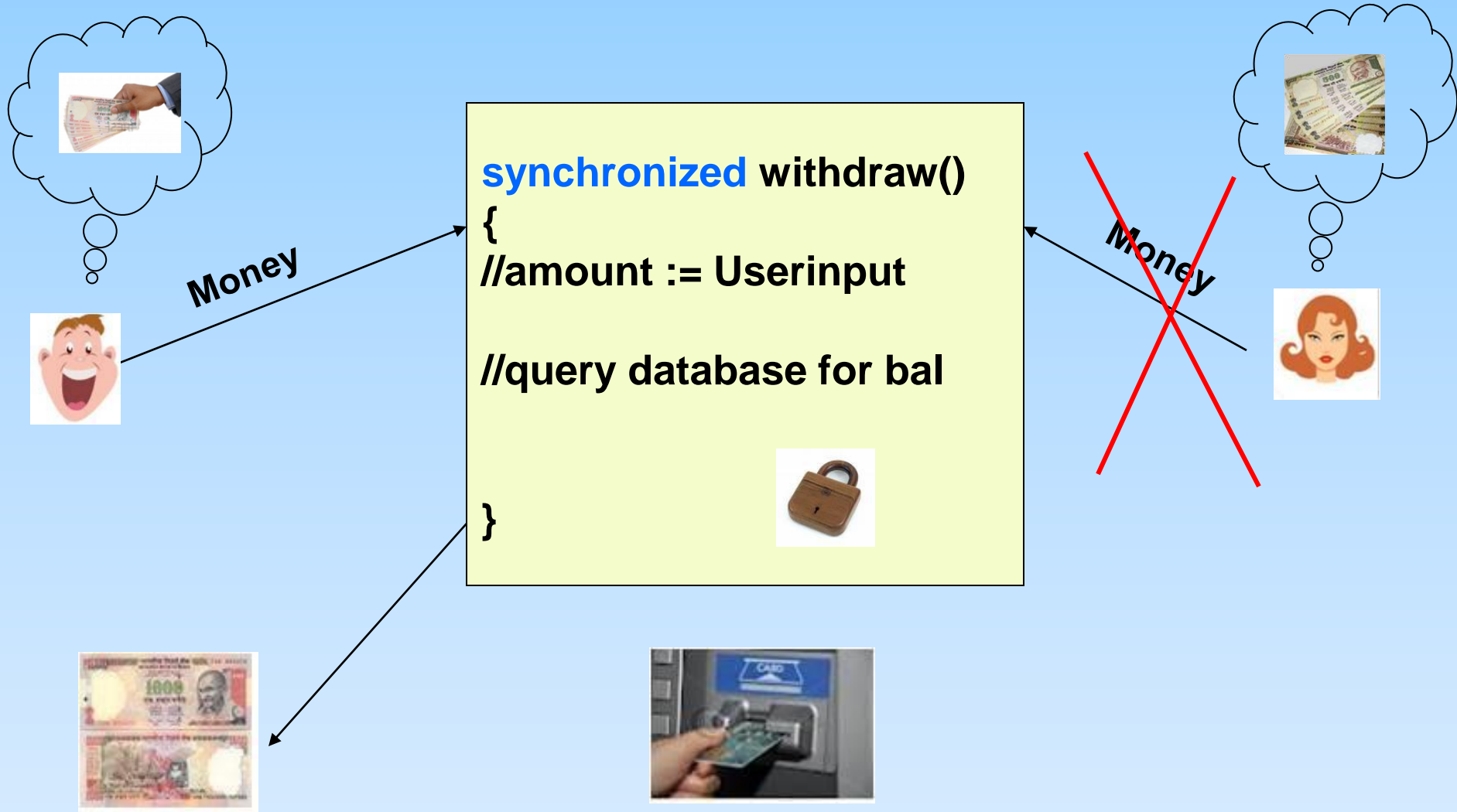
Using Synchronized Methods

- Implementation of concurrency control mechanism is very simple because every Java object has its own implicit monitor associated with it
- If a thread wants to enter an object's monitor, it has to just call the synchronized method of that object
- While a thread is executing a synchronized method, all other threads that are trying to invoke that particular synchronized method or any other synchronized method of the same object, will have to wait

Using Synchronized Methods (Contd.).



Using Synchronized Methods (Contd.).



Synchronization

- Every object in Java has a lock
- Using *synchronization* enables the lock and allows only one thread to access that part of code
- Synchronization can be applied to:
 - A method
`public synchronized withdraw(){...}`
 - A block of code
`synchronized (objectReference){...}`
- Synchronized methods in subclasses use same locks as their superclasses

The Synchronized Statement


Syntax for block of code

```
public void run()  
{  
    synchronized(obj)  
    {  
        obj.withdraw(500);  
    }  
}
```

How to get the required output?

```
class SharedObject {  
    void sharedMethod(String arg){  
        System.out.print("[");  
        System.out.print(arg);  
        try{  
            Thread.sleep(1000);  
        }  
        catch (InterruptedException e){  
            System.out.println("Interrupted");  
        }  
        System.out.println("]");  
    }  
}
```

This program is spread
across 3 slides



The instance of “SharedObject” is accessed by three Threads and through each of these threads we pass a String argument, which needs to be printed in the following format :

[Spirit]
[Of]
[Wipro]

How to get the required output? (Contd.).

```
class Thread1 implements Runnable{
    String arg;
    SharedObject obj1;
    Thread t;
    public Thread1(SharedObject obj1, String arg){
        this.obj1 = obj1;
        this.arg = arg;
        t = new Thread(this);
        t.start();
    }
    public void run(){
        obj1.sharedMethod(arg);
    }
}
```


How to get the required output? (Contd.).

```
class Synchro{  
    public static void main(String args[]){  
        SharedObject obj1 = new SharedObject();  
        Thread1 x1 = new Thread1(obj1, "Spirit");  
        Thread1 x2 = new Thread1(obj1, "Of");  
        Thread1 x3 = new Thread1(obj1, "Wipro");  
        try{  
            x1.t.join();  
            x2.t.join();  
            x3.t.join();  
        }  
        catch (InterruptedException e){  
            System.out.println("Interrupted");  
        }  
    }  
}
```

Will this program, in the current form, print the desired output?

No..! The output is
[[[SpiritOfWipro]
]
]

What you must do, to get the desired output?

Inter Thread Communication

Thread Messaging

- In Java, you need not depend on the OS to establish communication between threads
- All objects have predefined methods, which can be called to provide inter-thread communication

Inter-Thread Communication

- Threads are often interdependent - one thread depends on another thread to complete an operation, or to service a request.
- The words wait and notify encapsulate the two central concepts to thread communication
 - A thread waits for some condition or event to occur.
 - You notify a waiting thread that a condition or event has occurred.
- To avoid polling, Java's elegant inter-thread communication mechanism uses:
 - `wait()`
 - `notify()`, and `notifyAll()`

Inter-Thread Communication (Contd.).

- wait(), notify() and notifyAll() are:
 - Declared as final in Object
 - Hence, these methods are available to all classes
 - These methods can only be called from a synchronized context
- wait() directs the calling thread to surrender the monitor, and go to sleep until some other thread enters the monitor of the same object, and calls notify()
- notify() wakes up the other thread which was waiting on the same object(*that had called wait() previously on the same object*)

Inter-Thread Communication (Contd.).

- The following sample program incorrectly implements a simple form of the Developer/Client problem
- It consists of four classes namely:
 - Q, the queue that you are trying to synchronize
 - Developer, the threaded object that is producing queue entries
 - Client, the threaded object that is consuming queue entries
 - PC, the class that creates the single Queue, Developer, and Client

Inter-Thread Communication (Contd.).

```
public class Q {  
    int n;  
    synchronized int get( ) {  
        System.out.println( "Got: " + n);  
        return n;  
    }  
    synchronized void put( int n) {  
        this.n = n;  
        System.out.println( "Put: " + n);  
    }  
}
```

Inter-Thread Communication (Contd.).

```
class Developer implements Runnable {
    Q q;
    Developer ( Q q) {
        this.q = q;
        new Thread( this, "Developer").start( );
    }
    public void run( ) {
        int i = 0;
        while (true) {
            q.put (i++);
        }
    }
}
```


Inter-Thread Communication (Contd.).

```
class Client implements Runnable {  
    Q q;  
    Client ( Q q) {  
        this.q = q;  
        new Thread (this, "Client").start( );  
    }  
    public void run( ) {  
        while (true) {  
            q.get( );  
        }  
    }  
}
```

Inter-Thread Communication (Contd.).

```
class PC {  
  
    public static void main (String args [ ] ) {  
        Q q = new Q( );  
        new Developer (q);  
        new Client (q);  
        System.out.println("Press Control-C to  
stop");  
    }  
}
```

Inter-Thread Communication (Contd.).

Using wait () and notify ()

```
public class Q {  
    int n;  
    boolean valueset = false;  
    synchronized int get( ) {  
        if (!valueset)  
            try {  
                wait( );  
            }  
        catch (InterruptedException e) {  
  
            System.out.println("InterruptedException  
caught");  
        }  
    }  
}
```

Inter-Thread Communication (Contd.).

```
        System.out.println( "Got: " + n);
        valueset = false;
        notify( );
        return n;
    }
    synchronized void put( int n) {
        if (valueset)
            try {
                wait( );
            }
            catch (InterruptedException e) {

                System.out.println("InterruptedException
                caught");
            }
    }
```

Inter-Thread Communication (Contd.).

```
        this.n = n;
        valueset = true;
        System.out.println( "Put: " + n);
        notify( );
    }
}

class Developer implements Runnable {
    Q q;
    Developer ( Q q) {
        this.q = q;
        new Thread( this, "Developer").start( );
    }
}
```

Inter-Thread Communication (Contd.).

```
public void run( ) {  
    int i = 0;  
    while (true) {  
        q.put (i++);  
    }  
}  
class Client implements Runnable {  
    Q q;  
    Client ( Q q) {  
        this.q = q;  
        new Thread (this, "Client").start( );  
    }  
    public void run( ) {  
        while (true) {  
            q.get( );  
        }  
    }  
}
```

Inter-Thread Communication (Contd.).

```
class PC {  
    public static void main (String args [ ] ) {  
        Q q = new Q( );  
        new Developer (q);  
        new Client (q);  
        System.out.println("Press Control-C to  
stop");  
    }  
}
```

Review

Fill in the blanks :

1. wait(), notify() and notifyAll() methods are defined in _____ class.
2. In java, concurrency control mechanism is implemented by prefixing the method with the keyword _____.
3. In java, a thread can have a minimum priority of _____, while maximum is _____.
4. You can specify the priority of a thread by invoking _____ method on the thread object.
5. wait() and notify() methods can only be called from a _____ context.

Summary

- In this module, you were able to:
 - Set Thread priorities
 - Appreciate the need for Thread Synchronization
 - Implement a java program that uses Synchronized methods
 - Implement java program for inter thread communication

References

1. Schildt, H. Java: The Complete Reference. J2SETM. Ed 5. New Delhi: McGraw Hill-Osborne, 2005.
2. Tutorial point (2012). Java: Multithreading. Retrieved on March 29, 2012, from,
http://www.tutorialspoint.com/java/java_multithreading.htm
3. Oracle (2012). The Java Tutorials: Concurrency. Retrieved on March 29, 2012, from,
<http://docs.oracle.com/javase/tutorial/essential/concurrency/>

Thank You