BIRLA INSTITIUTE OF TECHNOLOGY AND SCIENCE, PILANI
CS/IS F213 and CS/IS C313
LAB-10

AGENDA **DATE: 30/10/2013** **TIME: 02 Hours**

1. **Creating a Thread**
2. **Major Thread Concepts**
3. **Using Multithreading**
4. **Inter-thread communication**
5. **Thread deadlocks**
6. **Ordering locks**
7. **Thread control**

1. Creating a Thread:
Java defines two ways in which this can be accomplished:

- You can implement the Runnable interface.
- You can extend the Thread class, itself.

1.1 Create Thread by Implementing Runnable:

The easiest way to create a thread is to create a class that implements the **Runnable** interface.To implement Runnable, a class need only implement a single method called **run()**, which is declared like this:

```
public void run()
```

You will define the code that constitutes the new thread inside run() method. It is important to understand that run() can call other methods, use other classes, and declare variables, just like the main thread can.
After you create a class that implements Runnable, you will instantiate an object of type Thread from within that class. Thread defines several constructors. The one that we will use is shown here:

```
Thread(Runnable threadOb, String threadName);
```

Here threadOb is an instance of a class that implements the Runnable interface and the name of the new thread is specified by threadName.After the new thread is created, it will not start running until you call its start() method, which is declared within Thread. The start() method is shown here:

```
void start();
```

Example-1: Here is an example that creates a new thread and starts it running:

```
// Create a new thread.
class NewThread implements Runnable {
   Thread t;
   NewThread() {
      // Create a new, second thread
      t = new Thread(this, "Demo Thread");
      System.out.println("Child thread: " + t);
      t.start(); // Start the thread
   }

   // This is the entry point for the second thread.
   public void run() {
      try {
         for(int i = 5; i > 0; i--) {
            System.out.println("Child Thread: " + i);
            // Let the thread sleep for a while.
            Thread.sleep(500);
         }
     } catch (InterruptedException e) {
         System.out.println("Child interrupted.");
     }
```

```
     System.out.println("Exiting child thread.");
   }
}

class ThreadDemo {
   public static void main(String args[]) {
      new NewThread(); // create a new thread
      try {
         for(int i = 5; i > 0; i--) {
           System.out.println("Main Thread: " + i);
           Thread.sleep(1000);
         }
      } catch (InterruptedException e) {
         System.out.println("Main thread interrupted.");
      }
      System.out.println("Main thread exiting.");
   }
}
```

1.2 Create Thread by Extending Thread:
The second way to create a thread is to create a new class that extends Thread, and then to create an instance of that class. The extending class must override the **run( )** method, which is the entry point for the new thread. It must also call **start( )** to begin execution of the new thread.

Example-2: Here is the preceding program rewritten to extend Thread:

```
// Create a second thread by extending Thread
class NewThread extends Thread {
   NewThread() {
      // Create a new, second thread
      super("Demo Thread");
      System.out.println("Child thread: " + this);
      start(); // Start the thread
   }

   // This is the entry point for the second thread.
   public void run() {
      try {
         for(int i = 5; i > 0; i--) {
            System.out.println("Child Thread: " + i);
                        // Let the thread sleep for a while.
            Thread.sleep(500);
         }
      } catch (InterruptedException e) {
         System.out.println("Child interrupted.");
      }
      System.out.println("Exiting child thread.");
   }
}

class ExtendThread {
   public static void main(String args[]) {
      new NewThread(); // create a new thread
      try {
         for(int i = 5; i > 0; i--) {
            System.out.println("Main Thread: " + i);
            Thread.sleep(1000);
         }
      } catch (InterruptedException e) {
         System.out.println("Main thread interrupted.");
      }
      System.out.println("Main thread exiting.");
   }
}
```

Example-3: The following ThreadClassDemo program demonstrates some of these methods of the Thread class:

```
// File Name : DisplayMessage.java
```

```
// Create a thread to implement Runnable
public class DisplayMessage implements Runnable{
   private String message;
   public DisplayMessage(String message){
      this.message = message;
   }
   public void run(){
      while(true) {
         System.out.println(message);
      }
   }
}

// File Name : GuessANumber.java
// Create a thread to extentd Thread
public class GuessANumber extends Thread{
   private int number;
   public GuessANumber(int number){
      this.number = number;
   }
   public void run(){
      int counter = 0;
      int guess = 0;
      do{
          guess = (int) (Math.random() * 100 + 1);
          System.out.println(this.getName()+ " guesses " + guess);
          counter++;
      }while(guess != number);
      System.out.println("Correct! " + this.getName() + " in "
                                                  + counter + " guesses.**");
   }
}

// File Name : ThreadClassDemo.java
public class ThreadClassDemo{
   public static void main(String[] args){

      Runnable hello = new DisplayMessage("Hello");
      Thread thread1 = new Thread(hello);
      thread1.setDaemon(true);
      thread1.setName("hello");
      System.out.println("Starting hello thread...");
      thread1.start();

      Runnable bye = new DisplayMessage("Goodbye");
      Thread thread2 = new Thread(hello);
      thread2.setPriority(Thread.MIN_PRIORITY);
      thread2.setDaemon(true);
      System.out.println("Starting goodbye thread...");
      thread2.start();

      System.out.println("Starting thread3...");
      Thread thread3 = new GuessANumber(27);
      thread3.start();
      try{
         thread3.join();
      }catch(InterruptedException e){
         System.out.println("Thread interrupted.");
      }
      System.out.println("Starting thread4...");
      Thread thread4 = new GuessANumber(75);
      thread4.start();
      System.out.println("main() is ending...");
   }
}
```

2. Major Thread Concepts:
While doing Multithreading programming, you would need to have following concepts very handy:

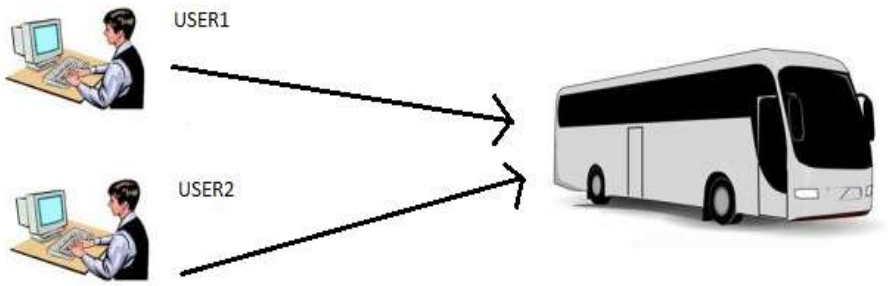
- Thread Synchronization
- Interthread Communication
- Thread Deadlock
- Thread Control: Suspend, Stop and Resume

3. Using Multithreading:

The key to utilizing multithreading support effectively is to think concurrently rather than serially. For example, when you have two subsystems within a program that can execute concurrently, make them individual threads. With the careful use of multithreading, you can create very efficient programs. A word of caution is in order, however: If you create too many threads, you can actually degrade the performance of your program rather than enhance it. Remember, some overhead is associated with context switching. If you create too many threads, more CPU time will be spent changing contexts than executing your program! When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this synchronization is achieved is called *thread synchronization*. The synchronized keyword in Java creates a block of code referred to as a critical section. Every Java object with a critical section of code gets a lock associated with the object. To enter a critical section, a thread needs to obtain the corresponding object's lock.

3.1 Thread Synchronization

Example- 4:



Let us consider an example where we have a travel website happybus.com through which you can book your bus tickets. Now there are only three seats left and two people are trying to book the tickets at the same time. Only one will be able to proceed and other will fail. If you consider in both the cases, the trouble arises when different threads try to work on the same data at same time. To avoid this problem, the code synchronization should be implemented which restricts multiple threads to work on the same code at the same time. Now, Let us see a simple demo of code without thread safety. Given below is the code which demonstrates the Bus reservation example without Thread Safety.

BusReservation.java

```
public class BusReservation implements Runnable{
    private int totalSeatsAvailable = 2;

    public void run() {
       //BusReservation br = new BusReservation();
       PassengerThread pt = (PassengerThread) Thread.currentThread();
       boolean ticketsBooked = this.bookTickets(pt.getSeatsNeeded(), pt.getName());
       if(ticketsBooked==true){
              System.out.println("CONGRATS" + Thread.currentThread().getName()
                     + "..The number of seats requested "
                     + "(" + pt.getSeatsNeeded() + ") are BOOKED");
       }
       else{
              System.out.println("Sorry Mr. " +Thread.currentThread().getName()
                     + " .. The number of seats requested "
                     + "(" + pt.getSeatsNeeded() + ")" are not available");
       }
    }

    public boolean bookTickets(int seats, String name){
       System.out.println("Welcome to HappyBus "
```

```
                    + Thread.currentThread().getName()
                    + " Number of Tickets Available are: "
                    + this.getTotalSeatsAvailable());

      if (seats>this.getTotalSeatsAvailable()) { return false; }
      else { totalSeatsAvailable = totalSeatsAvailable-seats; return true; }
    }

    private int getTotalSeatsAvailable() { return totalSeatsAvailable; }
}
```

PassengerThread.java

```
// This represent a Passenger.
public class PassengerThread extends Thread{
      private int seatsNeeded;
      public PassengerThread(int seats, Runnable target, String name) {
            super(target,name);
            this.seatsNeeded = seats;
      }
      public int getSeatsNeeded() {
            return seatsNeeded;
      }
}
```

ThreadSafetyDemo.java

```
public class ThreadSafetyDemo{
      public static void main(String[] args){
            BusReservation br = new BusReservation();
            PassengerThread pt1 = new PassengerThread(2,br,"Ramesh");
            PassengerThread pt2 = new PassengerThread(2, br, "Surresh");
            //PassengerThread pt3 = new PassengerThread(1, br, "Satish");
            pt1.start();
            pt2.start();
            //pt3.start();
      }
}
```

Output:
Welcome to HappyBus Ramesh Number of Tickets Available are: 2
Welcome to HappyBus Sumesh Number of Tickets Available are: 2
CONGRATS Ramesh .. The number of seats requested(2) are BOOKED
Sorry Mr. Sumesh .. The number of seats requested(2) are not available

In the above output, you can see that both user have got the number of tickets available as 2. But when the second user thread tried, the sorry message came out as the first user thread was able to successfully block the ticket. This kind of code behaviour **creates confusion in users and can have adverse effects in the data consistency**. To avoid this situation it is important to restrict the threads to act in queue. Code synchronization helps to restrict the multiple thread access for a particular part of the code.

3.2 Code Synchronization:

Code Synchronization helps in preventing threads working on the same code at the same time. The synchronization works based on the locking concept. Threads that work on the synchronized code should acquire the lock of the object or class file respectively. Please remember an important point that there is only one lock per object or loaded class file. Synchronization can only be applied for method or block of code. But not for classes or variables.

Syntax for using synchronize method:
public synchronize void method(){ }
public static synchronize void method{…}
synchronize(this){…}

Exercise 1: Now modify the code of BusReservation.java class and add synchronized keyword to the bookTickets() method. Other classes remain same. let us see how the bus reservation work when we put synchronization in place.

The code synchronization will restrict both the threads to enter bookTickets method at the same time which avoids the number of tickets available to be displayed for both of them at the same time.

Exercise2: Uncomment the comments in ThreadSafetyDemo.java and to add more threads(or users) to the same program and find the difference in the output when synchronisation is used. Observe the advantages of synchronisation in a multi user scenario.

4. Inter-thread Communication

Consider the classic queuing problem, where one thread is producing some data and another is consuming it. To make the problem more interesting, suppose that the producer has to wait until the consumer is finished before it generates more data.
In a polling system, the consumer would waste many CPU cycles while it waited for the producer to produce. Once the producer was finished, it would start polling, wasting more CPU cycles waiting for the consumer to finish, and so on. Clearly, this situation is undesirable.
To avoid polling, Java includes an elegant inter-process communication mechanism via the following methods:

- **wait( ):** This method tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify( ).
- **notify( ):** This method wakes up the first thread that called wait( ) on the same object.
- **notifyAll( ):** This method wakes up all the threads that called wait( ) on the same object. The highest priority thread will run first.

These methods are implemented as **final** methods in Object, so all classes have them. All three methods can be called only from within a **synchronized** context. These methods are declared within Object. Various forms of wait( ) exist that allow you to specify a period of time to wait.

Example 5:

The following sample program consists of four classes: Q, the queue that you're trying to synchronize; Producer, the threaded object that is producing queue entries; Consumer, the threaded object that is consuming queue entries; and PC, the tiny class that creates the single Q, Producer, and Consumer.

The proper way to write this program in Java is to use wait( ) and notify( ) to signal in both directions, as shown here:

```
class Q {
   int n;
   boolean valueSet = false;
   synchronized int get() {
      if(!valueSet)
      try {
         wait();
      } catch(InterruptedException e) {
         System.out.println("InterruptedException caught");
      }
      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");
      }
      this.n = n;
      valueSet = true;
      System.out.println("Put: " + n);
      notify();
   }
}

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

   public void run() {
      int i = 0;
      while(true) {
```

```
            q.put(i++);
        }
    }
}

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

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

Inside get(), wait() is called. This causes its execution to suspend until the Producer notifies you that some data is ready. When this happens, execution inside get() resumes. After the data has been obtained, get() calls notify(). This tells Producer that it is okay to put more data in the queue. Inside put(), wait() suspends execution until the Consumer has removed the item from the queue. When execution resumes, the next item of data is put in the queue, and notify() is called. This tells the Consumer that it should now remove it.

5. Thread Deadlock

A special type of error that you need to avoid that relates specifically to multitasking is deadlock, which occurs when two threads have a circular dependency on a pair of synchronized objects.

For example, suppose one thread enters the monitor on object X and another thread enters the monitor on object Y. If the thread in X tries to call any synchronized method on Y, it will block as expected. However, if the thread in Y, in turn, tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete.

Example 6:

In this example, there are two thread objects myThread and yourThread. When myThread starts running it holds lock on str1 object. If now yourThread starts it gets lock on str2. Now when first thread continues and wants lock on str2 then it is not available because second thread already holds lock on str2. Now both thread will wait for each other. myThread waits for str2 to be released by yourThread and yourThread is waiting for str1 to be released by myThread.. This program will never complete. As this example illustrates, if your multithreaded program locks up occasionally, deadlock is one of the first conditions that you should check for.

Use the below code for class **DeadLockDemo.java** and observe the output.

```
public class DeadLockDemo extends Thread {
       public static String str1 = "str1";
       public static String str2 = "str2";

       public static void main(String[] args) {
              Thread myThread = new MyThread();
              Thread yourThread = new YourThread();
              myThread.start();
              yourThread.start();
       }

       private static class MyThread extends Thread {
       public void run() {
          synchronized (str1) {
              System.out.println("MyThread Holds lock on object str1");
              try {
```

```
                Thread.sleep(10);
            } catch (InterruptedException e) {}

            System.out.println("MyThread waiting for lock on object str2");
            synchronized (str2) {
            System.out.println("MyThread Holds lock on objects str1, str2");
            }
         }//end of synchronized }//end of run }//end of MyThread
      private static class YourThread extends Thread {
            public void run() {
                synchronized (str2) {
                    System.out.println("YourThread Holds lock on object str2");
                    try {
                       Thread.sleep(10);
                    }
                    catch (InterruptedException e) {}
                    System.out.println("YourThread waiting for lock"
                                        + "on object str1");
                    synchronized (str1) {
                       System.out.println("YourThread Holds lock on"
                                        + "objects str1, str2");
                    } //end of inner synchronized block
                }//end of outer synchronized block
            }//end of run method
       }//end of inner class YourThread
}//end of class DeadLockDemo
```

6. Ordering Locks:
A common threading trick to avoid the deadlock is to order the locks. By ordering the locks, it gives threads a specific order to obtain multiple locks.

Deadlock Example 7: Following is the depiction of a dead lock:

```
// File Name ThreadSafeBankAccount.java
public class ThreadSafeBankAccount{

   private double balance;
   private int number;

   public ThreadSafeBankAccount(int num, double initialBalance){
      balance = initialBalance;
      number = num;
   }

   public int getNumber(){
      return number;
   }

   public double getBalance(){
      return balance;
   }

   public void deposit(double amount){
      synchronized(this){
        double prevBalance = balance;
        try{
           Thread.sleep(4000);
        }catch(InterruptedException e){}
        balance = prevBalance + amount;
      }
   }

   public void withdraw(double amount){
      synchronized(this){
         double prevBalance = balance;
         try{
            Thread.sleep(4000);
         }catch(InterruptedException e){}
```

```
         balance = prevBalance - amount;
      }
   }
}

// File Name LazyTeller.java
public class LazyTeller extends Thread{
   private ThreadSafeBankAccount source, dest;

   public LazyTeller(ThreadSafeBankAccount a, ThreadSafeBankAccount b){
      source = a;
      dest = b;
   }

   public void run(){
      transfer(250.00);
   }

   public void transfer(double amount) {
      System.out.println("Transferring from " + source.getNumber() +
                                                  " to " + dest.getNumber());
      synchronized(source){
          Thread.yield();
          synchronized(dest){
             System.out.println("Withdrawing from " + source.getNumber());
             source.withdraw(amount);
             System.out.println("Depositing into " + dest.getNumber());
             dest.deposit(amount);
          }
      }
   }
}

public class DeadlockDemo{
   public static void main(String [] args){
      System.out.println("Creating two bank accounts...");
      ThreadSafeBankAccount checking = new ThreadSafeBankAccount(101, 1000.00);
      ThreadSafeBankAccount savings = new ThreadSafeBankAccount(102, 5000.00);
      System.out.println("Creating two teller threads...");
      Thread teller1 = new LazyTeller(checking, savings);
      Thread teller2 = new LazyTeller(savings, checking);
      System.out.println("Starting both threads...");
      teller1.start();
      teller2.start();
   }
}
```

The problem with the LazyTeller class is that it does not consider the possibility of a race condition, a common occurrence in multithreaded programming. After the two threads are started, teller1 grabs the checking lock and teller2 grabs the savings lock. When teller1 tries to obtain the savings lock, it is not available. Therefore, teller1 blocks until the savings lock becomes available. When the teller1 thread blocks, teller1 still has the checking lock and does not let it go. Similarly, teller2 is waiting for the checking lock, so teller2 blocks but does not let go of the savings lock. This leads to one result: deadlock!

Deadlock Solution Example 8:
Here transfer() method, in a class named OrderedTeller, instead of arbitrarily synchronizing on locks, this transfer() method obtains locks in a specified order based on the number of the bank account.

```
// File Name ThreadSafeBankAccount.java
public class ThreadSafeBankAccount{
   private double balance;
   private int number;

   public ThreadSafeBankAccount(int num, double initialBalance){
      balance = initialBalance;
      number = num;
   }

   public int getNumber(){ return number; }
```

```
   public double getBalance(){ return balance; }

   public void deposit(double amount){
      synchronized(this) {
        double prevBalance = balance;
        try {
           Thread.sleep(4000);
        }catch(InterruptedException e){}
        balance = prevBalance + amount;
      }
   }

   public void withdraw(double amount) {
      synchronized(this){
             double prevBalance = balance;
         try{
            Thread.sleep(4000);
         }catch(InterruptedException e){}
         balance = prevBalance - amount;
      }
   }
}

// File Name OrderedTeller.java
public class OrderedTeller extends Thread{
   private ThreadSafeBankAccount source, dest;

   public OrderedTeller(ThreadSafeBankAccount a,ThreadSafeBankAccount b){
      source = a;
      dest = b;
   }

   public void run() {
      transfer(250.00);
   }

   public void transfer(double amount) {
       System.out.println("Transferring from " + source.getNumber()
           + " to " + dest.getNumber());
       ThreadSafeBankAccount first, second;
       if(source.getNumber() < dest.getNumber()){
          first = source;
          second = dest;
       }
       else{
          first = dest;
          second = source;
       }
       synchronized(first){
          Thread.yield();
          synchronized(second){
             System.out.println("Withdrawing from " + source.getNumber());
             source.withdraw(amount);
             System.out.println("Depositing into "+ dest.getNumber());
             dest.deposit(amount);
          }
      }
   }
}

// File Name DeadlockDemo.java
public class DeadlockDemo {
   public static void main(String [] args) {
      System.out.println("Creating two bank accounts...");
      ThreadSafeBankAccount checking = new ThreadSafeBankAccount(101, 1000.00);
      ThreadSafeBankAccount savings =  new ThreadSafeBankAccount(102, 5000.00);
```

```
      System.out.println("Creating two teller threads...");
      Thread teller1 = new OrderedTeller(checking, savings);
      Thread teller2 = new OrderedTeller(savings, checking);
      System.out.println("Starting both threads...");
      teller1.start();
      teller2.start();
   }
}
```

7. Thread Control

While the suspend( ), resume( ), and stop( ) methods defined by **Thread** class seem to be a perfectly reasonable and convenient approach to managing the execution of threads, they must not be used for new Java programs and obsolete in newer versions of Java. The following example illustrates how the wait( ) and notify( ) methods that are inherited from Object can be used to control the execution of a thread. This example is similar to the program in the previous section. However, the deprecated method calls have been removed. Let us consider the operation of this program. The NewThread class contains a boolean instance variable named suspendFlag, which is used to control the execution of the thread. It is initialized to false by the constructor. The run( ) method contains a synchronized statement block that checks suspendFlag. If that variable is true, the wait( ) method is invoked to suspend the execution of the thread. The mysuspend( ) method sets suspendFlag to true. The myresume( ) method sets suspendFlag to false and invokes notify( ) to wake up the thread. Finally, the main( ) method has been modified to invoke the mysuspend( ) and myresume( ) methods.

Example 9:

```
// Suspending and resuming a thread for Java 2
class NewThread implements Runnable {
   String name; // name of thread
   Thread t;
   boolean suspendFlag;

   NewThread(String threadname) {
      name = threadname;
      t = new Thread(this, name);
      System.out.println("New thread: " + t);
      suspendFlag = false;
      t.start(); // Start the thread
   }

   // This is the entry point for thread.
   public void run() {
      try {
      for(int i = 15; i > 0; i--) {
         System.out.println(name + ": " + i);
         Thread.sleep(200);
         synchronized(this) {
            while(suspendFlag) {wait();}
          }
        }
      } catch (InterruptedException e) {
         System.out.println(name + " interrupted.");
      }
      System.out.println(name + " exiting.");
   }

   void mysuspend() {
      suspendFlag = true;
   }

   synchronized void myresume() {
      suspendFlag = false;
      notify();
   }
}

class SuspendResume {
   public static void main(String args[]) {
      NewThread ob1 = new NewThread("One");
      NewThread ob2 = new NewThread("Two");
      try {
```

```
            Thread.sleep(1000);
            ob1.mysuspend();
            System.out.println("Suspending thread One");
            Thread.sleep(1000);
            ob1.myresume();
            System.out.println("Resuming thread One");
            ob2.mysuspend();
            System.out.println("Suspending thread Two");
            Thread.sleep(1000);
            ob2.myresume();
            System.out.println("Resuming thread Two");
         } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
         }
         // wait for threads to finish
         try {
            System.out.println("Waiting for threads to finish.");
            ob1.t.join();
            ob2.t.join();
         } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
         }
         System.out.println("Main thread exiting.");
      }
   }
```