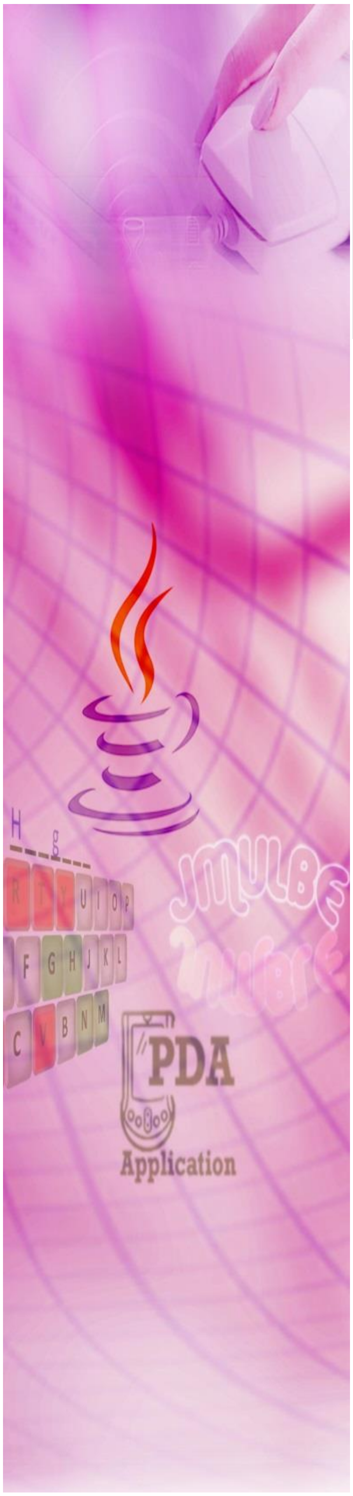


Session 7

Instructor Inputs

Session Overview

This session covers the Creating Multiple Threads and Identifying the Thread Priorities topics of the Creating Threads section of Chapter 5. In addition, it covers the Implementing Thread Synchronization section and the Implementing Atomic Variables and Locks topic of the Implementing Concurrency section of Chapter 6 of the book, Programming in Java – SG.

In this session, the students will explore the concept of synchronization and concurrency in Java. They will learn implementing thread synchronization. In addition, they will learn implementing atomic variables and locks in a multithreaded environment.

Creating Threads

Handling Tips

Start the discussion by asking the students few questions about the creation of threads. Ask them about the purpose of the `run()` and `start()` methods. Then, you can explain to the students the requirement of creating multiple threads with the help of the example given in the SG. Further, explain the code given in the SG that is used to create obstacles and make them appear at random positions on the frame. While explaining the concept of multiple threads, you can tell the students that there can be a situation where it is required to ensure that the execution of the main thread finishes after all the other threads have executed. For this, they will need to know whether a thread has finished its execution, or else, they can join the execution of a thread with the main method. This can be determined by using the `isAlive()` method and the `join()` method, respectively. Demonstrate the use of the preceding methods with the help of the information provided in the SG. Further, explain the `join()` method with the help of the example given under the Creating Multiple Threads subtopic in the Additional Examples topic. Now, ask the students what will happen if a class extends the `Thread` class and the `run()` method is not overriden. Elicit their responses and tell them that if the `run()` method is not overridden, a compilation error is not generated. However, the execution of the thread will not produce any result as the `run()` method has an empty definition in the `Thread` class.

Next, introduce the concept of thread priorities to the students. Thereafter, discuss how CPU manages execution of threads. Tell them that an advantage of using multiple threads is that the threads can be assigned a priority, which can ensure that important tasks can be assigned a higher priority. This will allow the tasks with a higher priority to take precedence over the lower priority tasks. Further, tell them that a thread in Java can be assigned a priority and a thread with a higher priority is scheduled for execution before the threads with a lower priority. The range of the thread priority is 1 to 10. Further, explain to them how to set the priority of a thread and explain the code given in the SG. Before concluding the session, explain the example given under the Additional Examples topic.

Additional Examples

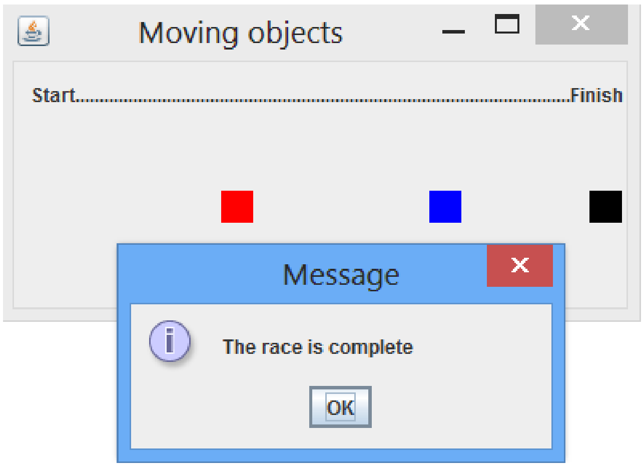
Creating Multiple Threads

The following code demonstrates the use of the `join()` method:

```
import java.awt.Color;
import java.util.Random;
import javax.swing.*;
public class ThreadRace extends Thread {
    String ThreadName;
    JLabel l;
    JPanel l1, l2, l3;
    JFrame fr;

    public ThreadRace() {
        buildGUI();
    }

    public ThreadRace(String s) {
        super(s);
    }

    public void run() {

        if (Thread.currentThread().getName().equals("RunnerA")) {
            runRunnerA();
            runRunnerB();
            runRunnerC();
        }

    }

    public void runRunnerA() {
        Random ran = new Random();
        // int s = ran.nextInt(1000);
        for (int i = 0; i < 130; i++) {

            l1.setBounds(i, 80, 20, 20);
            try {
                Thread.sleep(25);

            } catch (Exception e) {
                System.out.println(e);
            }
        }

    }

    public void runRunnerB() {
        Random ran = new Random();
        int r = ran.nextInt(180);

        for (int i = 130; i < 260; i++) {
            l2.setBounds(i, 80, 20, 20);
            try {
                Thread.sleep(15);

            } catch (Exception e) {
                System.out.println(e);
            }
        }
    }
    public void runRunnerC() {
        Random ran = new Random();
        int m = ran.nextInt(10);

        for (int i = 260; i < 360; i++) {
            l3.setBounds(i, 80, 20, 20);
            try {
                Thread.sleep(10);

            } catch (Exception e) {
                System.out.println(e);
            }
        }

    }

    public void buildGUI() {
        fr = new JFrame("Moving objects");
        fr.setVisible(true);
        fr.setSize(400, 200);
        fr.setLayout(null);

        l = new
JLabel("Start...........................................................
.......................................Finish");
        l.setBounds(10, 10, 400, 20);
        fr.add(l);

        l1 = new JPanel();
        l1.setSize(20, 20);
        l1.setBackground(Color.red);
        l1.setBounds(10, 80, 20, 20);
        fr.add(l1);

        l2 = new JPanel();
        l2.setSize(20, 20);
        l2.setBackground(Color.blue);
        l2.setBounds(130, 80, 20, 20);
        fr.add(l2);

        l3 = new JPanel();
        l3.setSize(20, 20);
        l3.setBackground(Color.black);
        l3.setBounds(260, 80, 20, 20);
        fr.add(l3);

    }

    public static void main(String args[]) {
        ThreadRace obj = new ThreadRace();
        Thread Runner1 = new Thread(obj);
        Thread Runner2 = new Thread(obj);
        Thread Runner3 = new Thread(obj);

        Runner1.setName("RunnerA");
        Runner2.setName("RunnerB");
```

```
            Runner3.setName("RunnerC");
            Runner1.start();
            try {
               Runner1.join();
            } catch (Exception e) {
            }
            Runner2.start();
            try {
                Runner2.join();
            } catch (Exception e) {
            }
            Runner3.start();
            try {
                Runner3.join();
            } catch (Exception e) {
            }
            JOptionPane.showMessageDialog(null, "The race is complete");

           }

    }
```

Once the preceding code is executed, the following output is displayed:



Moving objects
Start...Finish
Message
OK

The race is complete

The Output

In the preceding code, the `RunnerA`, `RunnerB`, and `RunnerC` threads are created and their execution is joined together sequentially one after the other. The three threads are used to simulate the baton relay game. When all the threads complete their execution, the `main()` method is executed in the end and the message, The race is complete, is displayed.

Identifying the Thread Priorities

You can use the following code to create two threads and set their priorities by using the priority constants, `MIN_PRIORITY` and `MAX_PRIORITY`:

```
public class ThreadDemo extends Thread {
    public void run()
    {
        System.out.println("Executed by: " +
Thread.currentThread().getName());
    }
    public static void main(String args[])
    {
        System.out.println("Executed by: " +
Thread.currentThread().getName());
        ThreadDemo thread1 = new ThreadDemo();
        ThreadDemo thread2 = new ThreadDemo();
        thread1.start();
        thread1.setName("1st thread");
        thread1.setPriority(MIN_PRIORITY);
        thread2.start();
        thread2.setName("2nd thread");
        thread2.setPriority(MAX_PRIORITY);
    }
}
```

Once the preceding code is executed, the following output is displayed:

```
Executed by: main
Executed by: 1st thread
Executed by: 2nd thread
```

Note
The preceding output may vary as the exact sequence of the thread execution cannot be determined.

Activity 5.1: Implementing Threads in Java

Handling Tips

Discuss the problem statement with the students.

To perform the activity, 5.1, you need to use the **SignalDemo.txt** file, which is provided at the following location in the TIRM CD:

- **Datafiles For Faculty\Activities\Chapter 05\Activity 5.1\Input Files**

The solution file, **SignalDemo.java**, for this activity is provided at the following location in the TIRM CD:

- **Datafiles For Faculty\Activities\Chapter 05\Activity 5.1\Solution**

Implementing Thread Synchronization

Handling Tips

Start the discussion by explaining the concept of synchronization by using the scenario given in the SG. Emphasize on the point that it is important to synchronize threads in a multithreaded environment. You can explain the example of a traffic signal, where the red lights are synchronized to avoid accidents.

Next, introduce the two approaches for implementing thread synchronization. While explaining the `synchronized` methods approach, emphasize on the concept of a monitor and how an object is locked so that no other synchronized method can be invoked. Focus on the point that when a thread is sleeping or waiting, it temporarily releases the lock it holds. Thereafter, tell them about the synchronized statement and how is it different from using the synchronized method. While discussing this topic, you can discuss the code given in the Synchronizing Threads subtopic under the Additional Examples topic.

After ensuring that the students have gained clarity on the concept of synchronization, move on to the next topic, Implementing Inter-threaded Communication. Start telling them that in multithreaded application it is often required that multiple thread communicates with each other, and then explain the importance of the communication between the threads. Discuss the producer consumer example, as given in the SG. Discuss the `wait()`, `notify()`, and `notifyAll()` methods and explain the usage of each method. Further, discuss the code that shows the producer-consumer problem, but without inter-threaded communication. Ensure that the students notice that the output of the program is not as required and emphasize that this is because the methods in the program are not synchronized. Then, show the inter-thread communication by using the `wait()` and `notify()` methods. Discuss the output of the program, as given in the SG, and ensure that the students notice that the output is as required. Further, while discussing the inter-thread communication, you can demonstrate the code given under the Implementing Inter-threaded Communication subtopic in the Additional Examples topic.

Additional Examples

Synchronizing Threads

The following code demonstrates how to implement and achieve synchronization in a multithreaded environment:

```
import java.util.Random;

class Person implements Runnable {

    private int id;   // id identifies each Person
    private static ATM ATMMachine = new ATM();  // only one ATM

    public Person(int id) {
        this.id = id;
    }

    public void run() {
        Random RandGen = new Random();
        int Rnd = RandGen.nextInt(200);
        performTransaction(Rnd);
        ATMMachine.useToll(this);  // current Person ATM
        performTransaction(Rnd);

    }

    public int getPersonId() {
        return this.id;
    }

    public void performTransaction(int time) {
        int limit = 500000;
        for (int j = 0; j < time; j++) {
            for (int k = 0; k < limit; k++) {// do nothing
            };
        }
    }
}

class ATM {

    boolean inUse;

    public ATM() {
        inUse = false;
    }

    public void useToll(Person Person) {
        while (true) {
            if (inUse == false) {
                synchronized (this) {
                    inUse = true;
                    System.out.println("Person " + (Person.getPersonId() + 1)
+ " enters ATM booth");
                    Person.performTransaction(50); // Person spends 50 time
units in the ATM booth
                    System.out.println("Person " + (Person.getPersonId() + 1)
+ " exits ATM booth");
                    inUse = false;
                    break;
                }

            }

        }

    }
}

public class Simulate {

    private static int noOfPersons = 3;
    private static Person[] Persons;

    public static void main(String[] args) {
        try {
            Simulate ATM = new Simulate();
            Persons = new Person[3];
            for (int i = 0; i < noOfPersons; i++) {
                Persons[i] = new Person(i);
                Thread t = new Thread(Persons[i]);
                t.start();
                Thread.sleep(10);
            }
        } catch (Exception ex) {
            System.out.println(ex);

        }

    }
}
```

Once the preceding code is executed, the following output is displayed:

```
Person 1 enters ATM booth
Person 1 exits ATM booth
Person 2 enters ATM booth
Person 2 exits ATM booth
Person 3 enters ATM booth
Person 3 exits ATM booth
```

The preceding code simulates the use of an ATM machine inside an ATM booth by three persons in a synchronized manner; only one person at a time can use the ATM booth. However, multiple threads can reach the ATM booth at a given time.

Implementing Inter-threaded Communication

The following code demonstrates how to use the `wait()` and `notify()` methods to achieve inter-thread communication:

```
class BankCustomer {

    int amount = 0;
    int checkDeposit = 0;

    public synchronized int withdraw(int amount) {
        System.out.println(Thread.currentThread().getName() + " wants to
withdraw");
        if (checkDeposit == 0) {
            try {
                System.out.println(Thread.currentThread().getName() + " is
waiting");
                wait();
            } catch (Exception e) {
            }
        }
        this.amount -= amount;
        System.out.println("Money withdrawn by " +
Thread.currentThread().getName());
        return amount;
    }
    public synchronized void depositMoney(int amt) {
        System.out.println(Thread.currentThread().getName() + " is going to 
deposit");
        this.amount += amt;
        notifyAll();
        System.out.println("Money deposited by " + 
Thread.currentThread().getName());
        checkDeposit = 1;
    }
}

public class NotifyThreads {

    public static void main(String a[]) {
        final BankCustomer Person1 = new BankCustomer();
        Thread t1 = new Thread() {
            public void run() {

                Person1.withdraw(1000);
            }
        };

        Thread t2 = new Thread() {
            public void run() {
                Person1.depositMoney(5000);
            }
        };

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

Once the preceding code is executed, the following output is displayed:

```
Thread-0 wants to withdraw
Thread-0 is waiting
Thread-1 is going to deposit
Money deposited by Thread-1
Money withdrawn by Thread-0
```

In the preceding code, a withdrawal is possible from a bank account only if a deposit has been made to the account.

Activity 6.1: Implementing Synchronization

Handling Tips

Discuss the problem statement with the students.

The solution file, **Simulate.java**, for this activity is provided at the following location in the TIRM CD:

- **Datafiles For Faculty\Activities\Chapter 06\Activity 6.1\Solution**

Implementing Concurrency

Handling Tips

Discuss with the students if they can identify any drawback of synchronization in Java. Elicit their responses and then tell them that synchronization does not ensure that a multithreaded application is utilizing the system resources in an efficient manner. Further, tell them that to overcome the preceding drawback and improve the efficiency of multithreaded applications, Java provides support for concurrent programming. Explain about the concept of concurrency to the students. Emphasize on how concurrency can be utilized to optimize the performance of an application.

Then, tell them that to implement concurrency, Java provides the `java.util.concurrent` package. Next, introduce the concept of atomic variables and operations to the students and focus on their requirements. Introduce the various classes of the `java.util.concurrent.atomic` package and explain the code given in the SG. While explaining this topic, you can discuss the code given under the Implementing Atomic Variables and Locks subtopic in the Additional Examples topic.

Thereafter, tell them that when synchronization is implemented, there is a possibility that the thread that has been waiting for the longest period of time never acquires the lock. In addition, if an exception is raised while a thread is executing the critical section, the lock may never be released. Such problems can reduce the efficiency and even hamper the execution of a multithreaded application. Then, discuss the `ReentrantLock` class and its constructors and methods. Focus on the methods of the `ReentrantLock` class. Ensure that the students have gained a clear understanding on each of the preceding methods. Further, explain the code that shows how to implement a reentrant lock, as given in the SG.

Additional Examples

Implementing Atomic Variables and Locks

The following code demonstrates an atomic operation of an atomic boolean variable:

```
import java.util.concurrent.atomic.AtomicBoolean;

public class AtomicBooleanDemo {
    AtomicBoolean ab= new AtomicBoolean(true);
    class ThreadA implements Runnable{

            @Override
            public void run() {
                    ab.compareAndSet(false, true);
                    System.out.println("Set to "+ab.get()+" by
"+Thread.currentThread().getName() );
            }

    }

    class ThreadB implements Runnable{

            @Override
            public void run() {
                    ab.compareAndSet(true, false);
```

```
                        System.out.println("Set to "+ab.get()+" by
"+Thread.currentThread().getName() );
              }

        }

       public static void main(String args[]){
           new Thread(new AtomicBooleanDemo().new ThreadA()).start();
           new Thread(new AtomicBooleanDemo().new ThreadB()).start();
       }
}
```

Once the preceding code is executed, the following output is displayed:

```
Set to true by Thread-0
Set to false by Thread-1
```

In the preceding code, an `AtomicBoolean` type object named `ab` is created. Then, by using the `compareAndSet()` method, its value is set to true, if it is false, or vice versa by the threads in the program.

FAQs

- *Is it possible to determine the thread that will resume execution when the `notifyAll()` method is invoked?*

  Ans: No, it is not possible to determine the specific thread that will resume execution when the `notifyAll()` method is invoked.

- *Can the `wait()` or `notify()` method be used outside a synchronized method?*

  Ans: No, the `wait()` or `notify()` method cannot be used outside a synchronized method.

- *Will an exception be generated if the `wait()`, `notify()`, or `notifyAll()` method is invoked from a block or method that is not synchronized?*

  Ans: Yes, the `InvalidMonitorStateException` exception will be generated if the `wait()`, `notify()`, or `notifyAll()` method is invoked from a block or method that is not synchronized.

- *Is it possible for a thread to invoke a non synchronized method when it is executing a synchronized method?*

  Ans: Yes, it is possible.

- *Is it possible to synchronize the constructor of a class?*

  Ans: No, as per the Java language specification, the constructors of a class cannot be synchronized.

- *Is it possible to use the `synchronized` keyword with the `run()` method?*

  Ans: Yes, it is possible.

- *Is the ++ operation on an integer variable an atomic operation?*

  Ans: No, it is not an atomic operation. The ++ operation on an integer comprises the following steps:

  a. Retrieve the value of the integer from memory.
  b. Increment the value.
  c. Assign the newly-incremented value to the appropriate memory location.
  d. Return the value to the caller.