Experiment number: 05

Experiment name: Write a Java program to create thread using thread class.

Objectives:

- Illustrate how to create threads in Java utilizing the Thread class.
- Understand how to make programs that can do multiple things simultaneously.
- Explore the lifecycle of a thread, from creation to termination, within a Java program.

Theory:

A thread represents a single line of execution within a program, allowing for concurrent execution of multiple tasks. In Java, threads are integral for achieving parallelism and responsiveness in applications. There are two ways of creating threads:

- By creating a thread class.
- Implementing the runnable interface.

Java provides the Thread class, which serves as a blueprint for creating and managing threads. For creating thread class it typically starts by invoking the **start()** method. This method initializes the thread and invokes its **run()** method. The **run()** method contains the code that the thread will execute concurrently with other threads.

The **start()** method is crucial because it manages the thread's lifecycle, including initialization, execution, and termination. Once the **start()** method is called, the thread becomes eligible for execution. After choosing the thread for execution, it invokes its **run()** method, allowing the thread to execute its code.

Source Code:

```
class A extends Thread {

  public void run() {

    for (int i = 1; i <= 5; i++) {

      System.out.println("\t From Thread A : i = " + i);

    }
```

```
        System.out.println("Exit from A");
    }
}
class B extends Thread {
    public void run() {
        for (int j = 1; j <= 5; j++) {
            System.out.println("\t From Thread B : j = " + j);
        }
        System.out.println("Exit from B");
    }
}

class C extends Thread {
    public void run() {
        for (int k = 1; k <= 5; k++) {
            System.out.println("\t From Thread C : k = " + k);
        }
        System.out.println("Exit from C");
    }
}
public class ThreadTest {
    public static void main(String args[]) {
        new A().start();
        new B().start();
        new C().start();
```

```
    }
}
```

Output:

```
From Thread A : i = 1
     From Thread A : i = 2
     From Thread A : i = 3
     From Thread A : i = 4
     From Thread A : i = 5
Exit from A
     From Thread C : k = 1
     From Thread B : j = 1
     From Thread B : j = 2
     From Thread B : j = 3
     From Thread B : j = 4
     From Thread B : j = 5
Exit from B
     From Thread C : k = 2
     From Thread C : k = 3
     From Thread C : k = 4
     From Thread C : k = 5
Exit from C
```

Experiment number: 06

Experiment name: Write a Java program to call threads using run() method.

Objectives:

- Develop a Java program to practice calling threads using the run() method.
- Write a simple Java application to understand thread invocation with the run() method.

Theory:

In Java, threads can be invoked directly by calling the **run()** method of a Thread object. Unlike the **start()** method, which initiates a new thread of execution, invoking **run()** executes the thread's code in the context of the current thread, leading to sequential execution rather than concurrent execution.

Using the **run()** method to call threads results in the execution of the thread's code within the same thread that made the invocation. This approach does not create a new execution context or enable true concurrency, as the thread runs in a sequential manner alongside the main thread of the program.

We can use the run() method by using a thread object.

Step 1: Create run method.
Step 2: Create an object for the class.
Step 3: This will call the run() method.

Source code:

```
class MyThread extends Thread {
    private String threadName;
    public MyThread(String name) {
        this.threadName = name;
    }
    public void run() {
        System.out.println("Thread " + threadName + " is running.");
    }
```

```
}
public class ThreadsWithRunMethod {
    public static void main(String[] args) {
        MyThread thread1 = new MyThread("Thread 1");
        MyThread thread2 = new MyThread("Thread 2");
        MyThread thread3 = new MyThread("Thread 3");

        // Calling run() method directly
        thread1.run();
        thread2.run();
        thread3.run();
    }
}
```

Output:

```
Thread Thread 1 is running.
Thread Thread 2 is running.
Thread Thread 3 is running.
```

Experiment number: 07

Experiment name: Write a Java program to illustrate yield() and sleep() method using thread.

Objectives:

- Create a Java program to show how the yield() and sleep() methods work in threads.
- Write a Java program that demonstrates the use of yield() and sleep() to manage thread behavior easily.

Theory:

Thread control is essential in Java programming, especially in concurrent applications. Two key methods for managing threads are yield() and sleep().

yield() Method: The yield() method in Java is used to pause the execution of the currently running thread and allow other threads of the same priority to run. It doesn't guarantee that the thread will immediately stop execution; rather, it gives the scheduler a hint that other threads can run. This method is useful in scenarios where threads need to share resources fairly.

sleep() Method: The sleep() method in Java is used to pause the execution of the current thread for a specified amount of time. It temporarily suspends the thread, allowing other threads to execute. This method is often used for tasks such as delaying execution or implementing timed operations.

In Java programming, when a thread calls the yield() method, it temporarily relinquishes its current CPU usage, allowing other threads to execute. This can lead to better resource utilization and fairness in thread execution. On the other hand, the sleep() method causes the current thread to sleep for a specified period, providing a way to introduce delays in thread execution or time-based operations.

Source code:

```
class Thread1 extends Thread {

   public void run() {

      String name = Thread.currentThread().getName();
```

```
        for (int i = 1; i <= 3; i++) {
            System.out.println(name + " running ");
            Thread.yield();
        }
    }
}
class Thread2 extends Thread {
    public void run() {
        String name = Thread.currentThread().getName();
        try {
            for (int i = 1; i <= 3; i++) {
                System.out.println(name + " running ");
                Thread.sleep(1000); // Sleeping for 1 second
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
class Thread3 extends Thread {
    public void run() {
        String name = Thread.currentThread().getName();
        for (int i = 1; i <= 3; i++) {
            System.out.println(name + " running ");
        }
```

```
    }
}
public class threadM {
    public static void main(String[] args) {
        Thread1 t1 = new Thread1();
        Thread2 t2 = new Thread2();
        Thread3 t3 = new Thread3();
        t1.setName("thread1");
        t2.setName("thread2");
        t3.setName("thread3");
        t1.start();
        t2.start();
        t3.start();
    }
}
```

Output:

```
thread2 running
thread1 running
thread3 running
thread3 running
thread3 running
thread1 running
thread1 running
thread2 running
thread2 running
```

Experiment number: 08

Experiment name: Write a Java program to use priority of thread.

Objectives:

- Develop a Java program to create multiple threads with distinct priorities.
- Illustrate thread prioritization effects on task scheduling and execution.
- Demonstrate the significance of understanding thread priority management for efficient concurrent programming.

Theory:

Java's Thread class provides methods to set and get the priority of a thread. By default, all threads inherit the priority of their parent thread, typically the main thread.When a thread is created, it inherits the priority of the thread that created it.The priority values range from 1 to 10, in increasing priority.The priority can be adjusted subsequently using the setPriority() method.The priority of a thread may be obtained using getPriority()

Priority constants are defined:

MIN_PRIORITY=1

MAX_PRIORITY=10

NORM_PRIORITY=5

The highest priority runnable thread is always selected for execution above lower priority threads.In general, the runnable thread with the highest priority is active (running). If a high-priority thread wakes up, and a low-priority thread is running,Then the high-priority thread gets to run immediately.

Source code:

```
class A extends Thread {
   public void run() {
      System.out.println(Thread.currentThread().getName());
      System.out.println(Thread.currentThread().getPriority());
```

```
    }
}
class ttt {
    public static void main(String[] args) {
        A t1 = new A();
        A t2 = new A();
        A t3 = new A();
        t1.setName("t1 thread");
        t2.setName("t2 thread");
        t3.setName("t3 thread");
        t1.setPriority(10);
        t2.setPriority(5);
        t3.setPriority(1);
        t1.start();
        t2.start();
        t3.start();

    }
}
```

Output:

```
t1 thread
10
t2 thread
5
t3 thread
1
```

Experiment number: 09

Experiment name: Write a client and server program in Java to establish a connection between them.

Objectives:

- Create a Java client-server program to connect two applications together.
- Establish a basic network connection between a client and a server using Java.

Theory:

Client-server communication is a fundamental concept in networking where two software components, the client and the server, interact with each other over a network. In client-server architecture, the server provides services or resources, while the client requests and utilizes these services. The client initiates the connection by sending a request to the server, and the server responds by providing the requested service or data.

In Java networking, the **Socket** class represents a client's connection to a server, while the **ServerSocket** class enables servers to listen for incoming connections.To establish communication, the client creates a socket by specifying the server's IP address and port number. Meanwhile, the server listens for connections on a designated port using a **ServerSocket**. Once a connection is established, both sides utilize input and output streams associated with the socket for bidirectional data exchange, enabling seamless interaction between client and server applications.

Source code:

Server model:

```
import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.io.PrintWriter;

import java.net.ServerSocket;

import java.net.Socket;


public class ServerModel {
```

```
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        try {
            System.out.println("Waiting for the clients...");
            ServerSocket ss= new ServerSocket(4999);
            Socket soc= ss.accept();
            System.out.println("Client Connected...");
            BufferedReader in= new BufferedReader(new
InputStreamReader(soc.getInputStream()));
            String str=in.readLine();
            System.out.println("Client Sent :"+str);
            PrintWriter out= new PrintWriter(soc.getOutputStream(),true);
            out.println("I got your msg");
            out.flush();
        }
 catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

Client model:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
```

```
public class ClientModel {
   public static void main(String[] args) {
      try {
         Socket soc= new Socket("localhost",4999);
         String str="hello guys";
         PrintWriter output=new PrintWriter(soc.getOutputStream());
         output.println(str);
         output.flush();
         BufferedReader in= new BufferedReader(new
InputStreamReader(soc.getInputStream()));
         String strinput=in.readLine();
         System.out.println("Server Sent :"+strinput);
      } catch (IOException e) {
         // TODO Auto-generated catch block
         e.printStackTrace();
      }
   }
}
```

Output:

Server model:

```
PS D:\3-2\Network programming\Lab\lab-9>  & 'C:\Program Files\Java\jdk-21\bin\java.exe' '--enable-preview' '-XX:+ShowCodeDetailsInEx
ceptionMessages' '-cp' 'C:\Users\Rakib\AppData\Roaming\Code\User\workspaceStorage\b2351fb0724ef950bd7fd49cfbdee071\redhat.java\jdt_w
s\lab-9_2016c7f\bin' 'ServerModel'
Waiting for the clients...
```

Client model:

```
PS D:\3-2\Network programming\Lab\lab-9>  & 'C:\Program Files\Java\jdk-21\bin\java.exe' '--enable-preview' '-XX:+ShowCodeDetails
InExceptionMessages' '-cp' 'C:\Users\Rakib\AppData\Roaming\Code\User\workspaceStorage\b2351fb0724ef950bd7fd49cfbdee071\redhat.ja
va\jdt_ws\lab-9_2016c7f\bin' 'ClientModel'
Server Sent :I got your msg
PS D:\3-2\Network programming\Lab\lab-9>
```