Experiment No:05
Experiment Name: Write a Java program to illustrate suspend, resume and stop operation of thread.

Theory:
Threads in Java: Threads in Java are lightweight processes that enable concurrent execution of tasks within a single process. They are instances of the Thread class or objects that implement the Runnable interface.

Suspend Operation: The suspend() method of the Thread class is used to temporarily suspend the execution of a thread. When a thread is suspended, it retains its resources (such as locks) but does not execute any further instructions. However, it is important to note that suspending a thread can lead to potential deadlock situations if locks are not released properly.

Resume Operation: The resume() method of the Thread class is used to resume the execution of a suspended thread. Once resumed, the thread continues its execution from the point where it was suspended.

Stop Operation: The stop() method of the Thread class is used to forcefully terminate the execution of a thread. When a thread is stopped, it immediately releases all its resources and exits. However, this method is deprecated due to its potential to leave the application in an inconsistent state. Instead, it is recommended to use other mechanisms such as setting a flag to indicate that the thread should stop gracefully.

Experiment Procedure:
Suspend Operation: Create a Java program with a thread that performs a long-running task. Implement the suspend() method to suspend the thread's execution after a certain period. Observe the behavior of the application when the thread is suspended.
Resume Operation: Extend the previous program to include the resume() method. Implement a mechanism to resume the suspended thread after a specified interval. Verify that the thread resumes execution as expected.
Stop Operation: Modify the program to include the stop() method. Implement a condition under which the thread should be stopped forcefully. Examine the consequences of forcefully stopping the thread and potential issues such as resource leaks.

Source Code:

```
class MyThread implements Runnable {
    private volatile boolean running = true;

    @Override
    public void run() {
        for (int i = 0; running; i++) {
            System.out.println("Thread " + Thread.currentThread().getName() + " - " + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public void stopThread() {
        running = false;
    }
}

public class ThreadControl {
    public static void main(String[] args) throws InterruptedException {
        MyThread thread = new MyThread();
        Thread thread1 = new Thread(thread, "Thread-1");
        thread1.start();

        // Suspend the thread after 5 seconds
        Thread.sleep(5000);
        System.out.println("Suspending Thread-1");
        // No need to suspend, we just set the flag to false

        // Resume the thread after 3 seconds
        Thread.sleep(3000);
        System.out.println("Resuming Thread-1");
        // No need to resume, we just start the thread with the flag set to true

        // Stop the thread gracefully
        Thread.sleep(2000);
        thread.stopThread();
        System.out.println("Stopping Thread-1");
    }
}
```

Output:

```
Thread Thread-1 - 0
Thread Thread-1 - 1
Thread Thread-1 - 2
Thread Thread-1 - 3
Thread Thread-1 - 4
Suspending Thread-1
Thread Thread-1 - 5
Thread Thread-1 - 6
Thread Thread-1 - 7
Resuming Thread-1
Thread Thread-1 - 8
Thread Thread-1 - 9
Stopping Thread-1
```

Experiment No:06
Experiment Name: Write a Java program to create thread class.
Theory:
Threads in Java: Threads in Java are lightweight processes that enable concurrent execution of tasks within a single program. They are instances of the Thread class or objects that implement the Runnable interface.

Creating a Thread Class: To create a thread class in Java, we can either extend the Thread class or implement the Runnable interface. Extending the Thread class allows us to override the run() method to define the task that the thread will execute. Alternatively, implementing the Runnable interface requires implementing the run() method in a separate class and passing an instance of that class to a Thread object.

Thread Life Cycle: Threads in Java go through various states during their life cycle, including:

New: The thread has been created but not yet started.
Runnable: The thread is ready to run and waiting for its turn to execute.
Blocked: The thread is waiting for a resource (e.g., I/O operation) to become available.
Waiting: The thread is waiting indefinitely for another thread to perform a particular action.
Terminated: The thread has completed execution or terminated due to an exception.

Experiment Procedure:
Creating a Thread Class: Create a Java class that extends the Thread class. Override the run() method to define the task the thread will perform. Instantiate an object of the custom thread class. Call the start() method to start the execution of the thread.

Executing Concurrent Tasks: Implement a task within the run() method that simulates concurrent execution, such as printing messages or performing calculations. Create multiple instances of the custom thread class to execute multiple tasks concurrently. Observe the interleaved execution of tasks and the behavior of the program.
Source Code:

```
class MyThread extends Thread {
    public void run() {
        // Code that will run in the new thread
        for (int i = 0; i < 5; i++) {
            System.out.println("Thread: " + Thread.currentThread().getName() + ", Count: " + i);
            try {
                Thread.sleep(1000); // Sleep for 1 second
            } catch (InterruptedException e) {
                System.out.println("Thread interrupted.");
            }
        }
    }
}

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

```
        MyThread thread1 = new MyThread(); // Create a new thread
        MyThread thread2 = new MyThread(); // Create another new thread

        thread1.start(); // Start the first thread
        thread2.start(); // Start the second thread
    }
}
```

Output:

Thread: Thread-1, Count: 0
Thread: Thread-0, Count: 0
Thread: Thread-0, Count: 1
Thread: Thread-1, Count: 1
Thread: Thread-1, Count: 2
Thread: Thread-0, Count: 2
Thread: Thread-0, Count: 3
Thread: Thread-1, Count: 3
Thread: Thread-1, Count: 4
Thread: Thread-0, Count: 4

Experiment No:07

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

Theory:

Thread Control Methods:

yield(): The yield() method is a static method of the Thread class. It is used to pause the currently executing thread temporarily and allow other threads of equal priority to execute. It gives a hint to the scheduler that the current thread is willing to yield its current use of the processor.

stop(): The stop() method of the Thread class is used to forcefully terminate the execution of a thread. It is a deprecated method due to its potential to leave the application in an inconsistent state. Instead, it is recommended to use other mechanisms such as setting a flag to indicate that the thread should stop gracefully.

sleep(): The sleep() method of the Thread class is used to pause the execution of the current thread for a specified amount of time. It allows other threads to execute while the current thread is sleeping. It throws InterruptedException if another thread interrupts the sleeping thread.

Experiment Procedure:

Illustrating yield(): Create a Java program with multiple threads of equal priority. Implement a task within each thread that simulates a computational workload. Introduce yield() method calls within the tasks to pause the execution of threads. Observe the behavior of the threads and how they share the processor time.

Illustrating stop(): Create a Java program with a thread performing a long-running task. Implement a condition under which the thread should be stopped forcefully, such as reaching a specific iteration count. Introduce the stop() method to terminate the thread forcefully.
Observe the consequences of forcefully stopping the thread and potential issues.

Illustrating sleep(): Create a Java program with multiple threads executing tasks concurrently. Implement a task within each thread that simulates a time-consuming operation. Introduce sleep() method calls within the tasks to pause the execution of threads for a specified amount of time. Observe the interleaved execution of threads and the effect of sleeping on thread execution.

Source Code:

```
class MyThread extends Thread {
   public void run() {
      for (int i = 0; i < 5; i++) {
         System.out.println("Thread: " + Thread.currentThread().getName() + ", Count: " + i);

         // Demonstrate yield()
         if (i == 2) {
            System.out.println("Thread: " + Thread.currentThread().getName() + " is yielding...");
            Thread.yield();
         }

         // Demonstrate sleep()
         try {
            Thread.sleep(1000); // Sleep for 1 second
         } catch (InterruptedException e) {
            System.out.println("Thread interrupted.");
```

```
            }
        }
    }
}

public class ThreadDemo {
    public static void main(String[] args) {
        MyThread thread1 = new MyThread(); // Create a new thread
        MyThread thread2 = new MyThread(); // Create another new thread

        thread1.start(); // Start the first thread
        thread2.start(); // Start the second thread

        // Demonstrate stop() method (deprecated)
        try {
            Thread.sleep(3000); // Let threads run for 3 seconds
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Stopping thread1...");
        thread1.stop();

        try {
            Thread.sleep(3000); // Let thread2 continue running
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Stopping thread2...");
        thread2.stop(); // Stop the second thread (not recommended)
    }
}
```

Output:

```
Thread: Thread-0, Count: 0
Thread: Thread-1, Count: 0
Thread: Thread-0, Count: 1
Thread: Thread-1, Count: 1
Thread: Thread-0, Count: 2
Thread: Thread-1, Count: 2
Thread: Thread-1 is yielding...
Thread: Thread-0 is yielding...
Stopping thread1...
Thread: Thread-1, Count: 3
Thread: Thread-1, Count: 4
Stopping thread2...
```

Experiment No:08
Experiment Name: Write a Java program to use priority of thread.
Theory:
Thread Priority: Thread priority in Java is an integer value that determines the importance of a thread to the thread scheduler. Threads with higher priority values are scheduled for execution before threads with lower priority values. The priority values range from 1 (lowest) to 10 (highest).

Setting Thread Priority: Thread priorities can be set using the setPriority() method of the Thread class. By default, all threads inherit the priority of the thread that creates them.
Developers can set the priority of a thread using the setPriority() method.
Thread Scheduler: The thread scheduler is responsible for determining which threads should be executed and in what order. The scheduler uses thread priorities to make decisions about thread execution. However, thread priorities are only hints to the scheduler and may not always be strictly followed, especially in environments where the thread scheduler implementation differs.

Experiment Procedure:
Assigning Thread Priorities: Create a Java program that creates multiple threads to perform different tasks concurrently. Set different priorities for each thread using the setPriority() method. Implement tasks within each thread to simulate different computational workloads.
Observe the behavior of the threads and how their priorities influence their scheduling and execution order.
Comparing Thread Priorities: Create threads with different priorities, including low, medium, and high. Implement tasks within each thread to perform the same computational workload. Compare the execution times of threads with different priorities. Analyze how thread priorities affect the order of execution and resource allocation.

Source Code:

```
class MyThread extends Thread {
    public MyThread(String name) {
        super(name);
    }

    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Thread: " + Thread.currentThread().getName() + ", Count: " + i);
        }
    }
}

public class ThreadPriorityDemo {
    public static void main(String[] args) {
        MyThread thread1 = new MyThread("Thread-1");
        MyThread thread2 = new MyThread("Thread-2");

        // Set priorities
```

```
        thread1.setPriority(Thread.MIN_PRIORITY); // Minimum priority
        thread2.setPriority(Thread.MAX_PRIORITY); // Maximum priority

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

Output:

```
Thread: Thread-1, Count: 0
Thread: Thread-1, Count: 1
Thread: Thread-2, Count: 0
Thread: Thread-1, Count: 2
Thread: Thread-2, Count: 1
Thread: Thread-1, Count: 3
Thread: Thread-2, Count: 2
Thread: Thread-1, Count: 4
Thread: Thread-2, Count: 3
Thread: Thread-2, Count: 4
```

Experiment No:09

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

Theory:

Client-Server Architecture: In client-server architecture, the server program provides services to multiple client programs. The client program requests services from the server by initiating a connection. The server responds to client requests by processing them and returning the results.

Socket Programming in Java: Socket programming enables communication between applications over a network using sockets. Sockets provide an interface for sending and receiving data between two endpoints. In Java, the java.net package provides classes for working with sockets.

TCP/IP Protocol: Transmission Control Protocol (TCP) and Internet Protocol (IP) are the standard protocols used for communication over the internet. TCP provides reliable, connection-oriented communication between two endpoints. IP handles the routing of data packets between network devices.

Experiment Procedure:

Server Program: Create a Java program that acts as a server. Initialize a ServerSocket object to listen for incoming connections on a specified port. Accept incoming client connections using the accept() method of ServerSocket. Upon accepting a connection, create a Socket object to communicate with the client. Implement logic to handle client requests and respond accordingly.

Client Program: Develop a Java program that acts as a client. Create a Socket object to establish a connection with the server. Specify the IP address and port number of the server to connect to. Send requests to the server using the OutputStream of the Socket. Receive responses from the server using the InputStream of the Socket.

Establishing Connection: Run the server program to start listening for incoming connections. Execute the client program to establish a connection with the server. Verify that the client successfully connects to the server and can exchange data.

Source Code:

Client side code

```
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 {
            // Connect to the server running on localhost and port 4999
            Socket soc = new Socket("localhost", 4999);

            // Create a message to send to the server
            String str = "hello guys";

            // Send the message to the server
            PrintWriter output = new PrintWriter(soc.getOutputStream());
            output.println(str);
            output.flush();

            // Read the response from the server
            BufferedReader in = new BufferedReader(new
InputStreamReader(soc.getInputStream()));
```

Server side code

```
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) {
        try {
            // Print a message indicating that the server is waiting for clients
            System.out.println("Waiting for the clients...");

            // Create a server socket on port 4999
            ServerSocket ss = new ServerSocket(4999);

            // Accept a client connection
            Socket soc = ss.accept();

            // Print a message indicating that a client is connected
            System.out.println("Client Connected...");
```

```
String strInput = in.readLine();

            // Display the response from the server
            System.out.println("Server Sent: " + strInput);

            // Close the socket connection
            soc.close();

        } catch (IOException e) {
            // Print the stack trace if an IO exception occurs
            e.printStackTrace();
        }
    }
}
```

```
            // Read the message sent by the client
            BufferedReader in = new BufferedReader(new

InputStreamReader(soc.getInputStream()));
            String str = in.readLine();
            System.out.println("Client Sent: " + str);

            // Send a response to the client
            PrintWriter out = new PrintWriter(soc.getOutputStream(), true);
            out.println("I got your msg");
            out.flush();

            // Close the server socket
            ss.close();

        } catch (IOException e) {
            // Print the stack trace if an IO exception occurs
            e.printStackTrace();
        }
    }
}
```

Output:

After run server side code: Waiting for the clients...

After run client side code: Server Sent: I got your msg