CHAROTAR UNIVERSITY OF SCIENCE & TECHNOLOGY

DEVANG PATEL INSTITUTE OF ADVANCE TECHNOLOGY & RESEARCH

Department of Computer Science & Engineering

Subject Name: Java Programming

Semester: 3rd

Subject Code:CSE201

Academic year: 2024-25

Part - 7

No.	Aim of the Practical

32.	**AIM:**
	
	Write a program to create thread which display “Hello World” message. A. by extending Thread class B. by using Runnable interface.
	
	PROGRAM CODE:
	
	//A. By Extending the Thread Class
	
	/*public class p32 extends Thread {
	
	public void run() {
	
	System.out.println("Hello World");

```
    }


    public static void
main(String[] args) {

        p32 thread = new p32();

        thread.start();

    }

}*/


//B. By Implementing the
Runnable Interface

public class p32 implements
Runnable {

    public void run() {

        System.out.println("Hello
World");

    }


    public static void
main(String[] args) {

        Thread thread = new
Thread(new p32());

        thread.start();

    }

}
```

OUTPUT:

```
C:\Users\saumy\OneDrive\Documents\JAVA\Practical 7>java p32
Hello World
```

CONCLUSION:

Extending Thread: When we extend the `Thread` class, we have access to the thread's methods and variables. However, this approach can lead to tight coupling between the thread's implementation and the `Thread` class.

Implementing Runnable: When we implement the `Runnable` interface, we decouple the thread's implementation from the `Thread` class. This approach is more flexible and allows for easier reuse of the thread's logic.

33.

AIM:

Write a program which takes N and number of threads as an argument. Program should distribute the task of summation of N numbers amongst number of threads and final result to be displayed on the console.

PROGRAM CODE:

```
import java.util.Scanner;


class SumThread extends Thread {

    private int start;

    private int end;

    private int result;


    public SumThread(int start, int end) {

        this.start = start;
```

```
        this.end = end;
    }

    public void run() {
        result = 0;
        for (int i = start; i <= end; i++) {
            result += i;
        }
    }

    public int getResult() {
        return result;
    }
}

public class p33 {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        System.out.print("Enter the value of N: ");
        int n = s.nextInt();
        System.out.print("Enter the number of threads: ");
        int Threads = s.nextInt();
```

```
int Size = n / Threads;
SumThread[] threads = new SumThread[Threads];

for (int i = 0; i < Threads; i++) {
    int start = i * Size + 1;
    int end = (i == Threads - 1) ? n : (i + 1) * Size;
    threads[i] = new SumThread(start, end);
    threads[i].start();
}

int sum = 0;
for (int i = 0; i < Threads; i++) {
    try {
        threads[i].join();
        int threadResult = threads[i].getResult();
        System.out.println("Thread " + (i + 1) + " value: " + threadResult);
        sum += threadResult;
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

System.out.println("The sum of the numbers from 1 to " + n + " is: " + sum);

}

}

OUTPUT:

```
C:\Users\saumy\OneDrive\Documents\JAVA\Practical 7>java p33
Enter the value of N: 5
Enter the number of threads: 3
Thread 1 value: 1
Thread 2 value: 2
Thread 3 value: 12
The sum of the numbers from 1 to 5 is: 15
```

CONCLUSION:

In terms of limitations, this program assumes that the number of threads is fixed and known beforehand. In a real-world scenario, the number of threads might need to be dynamically adjusted based on the system's resources and workload.

34.

AIM:

Write a java program that implements a multi-thread application that has three threads. First thread generates random integer every 1 second and if the value is even, second thread computes the square of the number and prints. If the value is odd, the third thread will print the

value of cube of the number.

PROGRAM CODE:

import java.util.Scanner;

```
class Even extends Thread {
    private int n;

    public Even(int n) {
        this.n = n;
    }

    public void run() {

            int square = n * n;
            System.out.println("Square of " + n + "= " + square);

    }
}

class Odd extends Thread {
    private int n;

    public Odd(int n) {
        this.n = n;
    }
```

```
    public void run() {


        int cube = n * n * n;

        System.out.println("Cube of " + n + "= " + cube);


    }

}


public class p34 {

    public static void main(String[] args) {

        Scanner s = new Scanner(System.in);

        for (int i = 1; i <= 10; i++) {


            System.out.println("number is: " + i);

            if (i % 2 == 0) {

                Even e2 = new Even(i);

                e2.start();

            } else {

                Odd o3 = new Odd(i);

                o3.start();

            }

            try {
```

```
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
  }
}
```

OUTPUT:

```
C:\Users\saumy\OneDrive\Documents\JAVA\Practical 7>java p34
number is: 1
Cube of 1= 1
number is: 2
Square of 2= 4
number is: 3
Cube of 3= 27
number is: 4
Square of 4= 16
number is: 5
Cube of 5= 125
number is: 6
Square of 6= 36
number is: 7
Cube of 7= 343
number is: 8
Square of 8= 64
number is: 9
Cube of 9= 729
number is: 10
Square of 10= 100
```

CONCLUSION:

In terms of limitations, this program assumes that the threads will run indefinitely, and there is no mechanism to stop or terminate the threads. In a real-world scenario, thread management and termination mechanisms would be necessary to ensure proper resource cleanup and system stability.

35.

AIM:

Write a program to increment the value of one variable by one and display it after one second using thread using sleep() method.

PROGRAM CODE:

```
class MyThread extends Thread
{
	public void run()
	{
		for(int i=0;i<10;i++)
		{
			System.out.println("Call no.="+i);
			try
			{
			sleep(1000);
			}
			catch(InterruptedException e)
			{
			e.printStackTrace();
			}
		}
	}
}
```

```
class extra3
{
	public static void main(String []args)
	{
		MyThread t=new MyThread();
		t.start();
		for(int i=0;i<10;i++)
		{
			System.out.println("main thread");
		}
	}
}
```

OUTPUT:

```
C:\Users\saumy\OneDrive\Documents\JAVA\Practical 7>java p35
main thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread
Call no.=0
Call no.=1
Call no.=2
Call no.=3
Call no.=4
Call no.=5
Call no.=6
Call no.=7
Call no.=8
Call no.=9
```

CONCLUSION:

Thread-based increment: The program demonstrates how to use a thread to increment a variable's value periodically.

Sleep() method: The program uses the `sleep()` method to pause the thread's execution for a specified duration (1 second in this case).
Concurrency: The program shows how a thread can run concurrently with the main thread, allowing for asynchronous execution of tasks.
Simple synchronization: The program uses the `Thread.sleep()` method to introduce a delay between increments, ensuring that the variable's value is updated at a fixed interval.

AIM:

Write a program to create three threads ‘FIRST’, ‘SECOND’, ‘THIRD’. Set the priority of the ‘FIRST’ thread to 3, the ‘SECOND’ thread to 5(default) and the ‘THIRD’ thread to 7.

36.

PROGRAM CODE:

```
class First extends Thread
{
public void run()
   {
        System.out.println("First");


   }
}
class Second extends Thread
{
public void run()
   {
        System.out.println("Second");


   }
}
class Third extends Thread
{
public void run()
   {
        System.out.println("Third");
```

```
    }
}
public class p36
{
    public static void main(String[] args)
    {
        First f1=new First();
        Second s1=new Second();
        Third t1=new Third();

        f1.getPriority();
        s1.getPriority();
        t1.getPriority();

        f1.setPriority(5);
        s1.setPriority(4);
        t1.setPriority(6);

        f1.start();
        s1.start();
        t1.start();
    }
```

}

OUTPUT:

```
C:\Users\saumy\OneDrive\Documents\JAVA\Practical 7>java p36
Second
First
Third

C:\Users\saumy\OneDrive\Documents\JAVA\Practical 7>javac p36.java

C:\Users\saumy\OneDrive\Documents\JAVA\Practical 7>java p36
First
Second
Third
```

CONCLUSION:

37. In Java, thread priorities are a way to indicate the relative importance of threads to the JVM thread scheduler. The `setPriority()` method allows you to change the priority of a thread, with values ranging from `Thread.MIN_PRIORITY` (1) to `Thread.MAX_PRIORITY` (10).

However, it's important to note that thread priority behavior is platform-dependent, meaning that on some systems, thread priorities may not have much effect on the order of execution or overall performance. The JVM might not strictly enforce priority-based execution, as thread scheduling is ultimately determined by the underlying operating system.

This program demonstrates how to create and manage thread priorities in Java, but in practice, thread priority is typically not relied on as a robust mechanism for controlling thread execution in multi-threaded applications.

AIM:

Write a program to solve producer-consumer problem using thread synchronization.

PROGRAM CODE:

class ProducerConsumer {

```
// Shared buffer and control variables
private final int[] buffer;
private int size;
private int count = 0;  // Tracks the number of items in the buffer
private int in = 0;     // Points to the next empty position to produce
private int out = 0;    // Points to the next filled position to consume

public ProducerConsumer(int size) {
    this.size = size;
    buffer = new int[size];
}

// Producer method
public synchronized void produce(int item) throws InterruptedException {
    while (count == size) {
        // Buffer is full, wait for consumer to consume
        wait();
    }

    // Simulate production delay
    Thread.sleep(500); // 500 milliseconds delay
```

```
        // Produce an item into the buffer
        buffer[in] = item;
        in = (in + 1) % size;
        count++;

        System.out.println("Produced: " + item);

        // Notify the consumer that an item has been produced
        notify();

        // Wait until the consumer consumes the item before producing the next one
        wait();
    }

    // Consumer method
    public synchronized void consume() throws InterruptedException {
        while (count == 0) {
            // Buffer is empty, wait for producer to produce
            wait();
        }
```

```
        // Simulate consumption delay
        Thread.sleep(500); // 500 milliseconds delay

        // Consume an item from the buffer
        int item = buffer[out];
        out = (out + 1) % size;
        count--;

        System.out.println("Consumed: " + item);

        // Notify the producer that an item has been consumed
        notify();
    }
}

class Producer extends Thread {
    private final ProducerConsumer sharedBuffer;

    public Producer(ProducerConsumer sharedBuffer) {
        this.sharedBuffer = sharedBuffer;
    }
```

```
    @Override
    public void run() {
        try {
            for (int i = 1; i <= 10; i++) {
                sharedBuffer.produce(i);
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}

class Consumer extends Thread {
    private final ProducerConsumer sharedBuffer;

    public Consumer(ProducerConsumer sharedBuffer) {
        this.sharedBuffer = sharedBuffer;
    }

    @Override
    public void run() {
```

```
        try {
            for (int i = 1; i <= 10; i++) {
                sharedBuffer.consume();
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}


public class p37 {
    public static void main(String[] args) {
        ProducerConsumer sharedBuffer = new ProducerConsumer(1); // 
Buffer of size 1


        Producer producer = new Producer(sharedBuffer);
        Consumer consumer = new Consumer(sharedBuffer);


        producer.start();
        consumer.start();
    }
}
```

OUTPUT:

```
C:\Users\saumy\OneDrive\Documents\JAVA\Practical 7>java p37
Produced: 1
Consumed: 1
Produced: 2
Consumed: 2
Produced: 3
Consumed: 3
Produced: 4
Consumed: 4
Produced: 5
Consumed: 5
Produced: 6
Consumed: 6
Produced: 7
Consumed: 7
Produced: 8
Consumed: 8
Produced: 9
Consumed: 9
Produced: 10
Consumed: 10
```

CONCLUSION:

This implementation illustrates how to solve a common threading problem using basic synchronization tools available in Java, avoiding common pitfalls such as deadlock, race conditions, and buffer overflow/underflow.