

CHAROTAR UNIVERSITY OF SCIENCE TECHNOLOGY
DEVANG PATEL INSTITUTE OF ADVANCE TECHNOLOGY &
RESEARCH

Department of Computer Science & Engineering

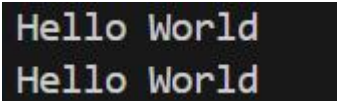
Subject Name: Java Programming

Semester: 3rd

Subject Code: CSE201

Academic year: 2024

SET- 6

No	Aim of the Practical
32.	<p>Write a program to create thread which display “Hello World” message. A. by extending Thread class B. by using Runnable interface.</p> <p><u>PROGRAM CODE :</u></p> <pre>public class pract32 { public static void main(String[] args) { Thread t1 = new Thread1(); t1.start(); Thread t2 = new Thread(new Thread2()); t2.start(); } }</pre> <pre>class Thread2 implements Runnable { public void run() { System.out.println("Hello World"); } }</pre> <pre>class Thread1 extends Thread { public void run() { System.out.println("Hello World"); } }</pre> <p><u>OUTPUT:</u></p> 

	<p><u>CONCLUSION:</u></p> <p>The Java code demonstrates two approaches to creating threads: extending the Thread class and implementing the Runnable interface. It starts a thread that prints "Hello world" and another that prints "hello world runnable interface." This example illustrates the flexibility of thread creation in Java and showcases basic multithreading concepts..</p>
33	<p>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.</p> <p><u>PROGRAM CODE :</u></p> <pre>import java.util.*; public class prac33 { public static void main(String[] args) { Scanner scanner = new Scanner(System.in); System.out.print("Enter the number N (total numbers to sum): "); int N = scanner.nextInt(); System.out.print("Enter the number of threads: "); int numThreads = scanner.nextInt(); int sum = 0; Thread[] threads = new Thread[numThreads]; Summation.sum = new int[numThreads]; for (int i = 0; i < numThreads; i++) { threads[i] = new Thread(new Summation(N, i, numThreads)); threads[i].start(); } for (int i = 0; i < numThreads; i++) { try { threads[i].join(); } catch (InterruptedException e) {</pre>

```
        e.printStackTrace();
    }
}

for (int i = 0; i < numThreads; i++) {
    sum += Summation.sum[i];
}

System.out.println("Sum: " + sum);
}
}

class Summation implements Runnable {
    static int[] sum;
    int N, start, numThreads;

    Summation(int N, int start, int numThreads) {
        this.N = N;
        this.start = start;
        this.numThreads = numThreads;
    }

    public void run() {
        for (int i = start + 1; i <= N; i += numThreads) {
            sum[start] += i;
        }
    }
}
```

OUTPUT:

```
Enter the number N (total numbers to sum): 5
Enter the number of threads: 4
Sum: 15
```

CONCLUSION:

The Java code efficiently calculates the sum of the first N natural numbers

	<p>using multiple threads. Each thread contributes to the total by summing a portion of the range, and the results are aggregated at the end. This example highlights the use of multithreading for parallel computation and demonstrates thread management in Java.</p>
34	<p>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.</p>

PROGRAM CODE :

```
public class prac34 {
    public static void main(String[] args) {
        Thread1 t1 = new Thread1();
        t1.start();
    }
}
class Thread1 extends Thread {
    public void run() {
        while (true) {
            int n = (int) (Math.random() * 100);
            System.out.println("Generated number: " + n);
            if (n % 2 == 0) {

                new Thread2(n).start();
            } else {

                new Thread3(n).start();
            }
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

class Thread2 extends Thread {
    private int n;

    Thread2(int n) {
        this.n = n;
        setName("EvenThread");
    }

    public void run() {
        System.out.println(getName() + ": Square of " + n + " is " + (n * n));
    }
}
```

```
}  
  
class Thread3 extends Thread {  
    private int n;  
  
    Thread3(int n) {  
        this.n = n;  
        setName("OddThread");  
    }  
  
    public void run() {  
        System.out.println(getName() + ": Cube of " + n + " is " + (n * n * n));  
    }  
}
```

OUTPUT:

```
Generated number: 73  
OddThread: Cube of 73 is 389017  
Generated number: 34  
EvenThread: Square of 34 is 1156  
Generated number: 49  
OddThread: Cube of 49 is 117649  
Generated number: 57  
OddThread: Cube of 57 is 185193  
Generated number: 17  
OddThread: Cube of 17 is 4913  
Generated number: 36  
EvenThread: Square of 36 is 1296  
Generated number: 55  
OddThread: Cube of 55 is 166375
```

CONCLUSION:

The Java code implements a multithreaded application that generates random numbers and spawns new threads based on whether the number is even or odd. Even numbers trigger the creation of a thread that calculates and prints their square, while odd numbers create a thread that computes their cube. This example effectively demonstrates dynamic thread creation and the handling of different tasks using Java's threading capabilities.

- 35** 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 :

```
public class prac35 {  
    public static void main(String[] args) {  
        Thread1 t1 = new Thread1();  
        t1.start();  
    }  
}  
  
class Thread1 extends Thread {  
    public void run() {  
        int n = 0;  
        while (true) {  
            n++;  
            System.out.println(n);  
            try {  
                Thread.sleep(1000);  
            } catch (InterruptedException e) {  
                System.out.println("Thread interrupted.");  
            }  
        }  
    }  
}
```

OUTPUT:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
```

CONCLUSION:

The Java code creates a simple multithreaded application that continuously increments and displays a variable every second. By extending the Thread class, it demonstrates basic thread functionality, including looping and sleeping. This example effectively illustrates the fundamental principles of thread execution and timing in Java.

- 36** 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.

PROGRAM CODE :

```
public class prac36 {
    public static void main(String[] args) {
        Thread first = new Thread(new Runnable() {
            public void run() {
                System.out.println(Thread.currentThread().getName() + " Priority:
" + Thread.currentThread().getPriority());
            }
        });

        Thread second = new Thread(new Runnable() {
            public void run() {
                System.out.println(Thread.currentThread().getName() + " Priority:
" + Thread.currentThread().getPriority());
            }
        });

        Thread third = new Thread(new Runnable() {
            public void run() {
                System.out.println(Thread.currentThread().getName() + " Priority:
" + Thread.currentThread().getPriority());
            }
        });

        first.setPriority(3);
        second.setPriority(5);
        third.setPriority(7);

        first.setName("FIRST");
        second.setName("SECOND");
        third.setName("THIRD");

        first.start();
        second.start();
        third.start();
    }
}
```

```
}  
}
```

OUTPUT:

```
SECOND Priority: 5  
FIRST Priority: 3  
THIRD Priority: 7
```

CONCLUSION:

The provided Java code demonstrates the creation and management of threads by extending the Thread class. It showcases setting custom priorities for three threads and starts their execution, which will display their names in the console. This example highlights basic threading concepts, including priority management in Java.

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

PROGRAM CODE :

```
public class prac37 {
    public static void main(String[] args) {
        Buffer buffer = new Buffer();
        Producer producer = new Producer(buffer);
        Consumer consumer = new Consumer(buffer);

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

class Buffer {
    private int[] buffer;
    private int size;
    private int count;

    Buffer() {
        size = 5;
        buffer = new int[size];
        count = 0;
    }

    public synchronized void produce(int item) {
        while (count == size) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        buffer[count] = item;
        count++;
        System.out.println("Produced: " + item);
        notify();
    }
}
```

```
}  
public synchronized int consume() {  
    while (count == 0) {  
        try {  
            wait();  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
    int item = buffer[count - 1];  
    count--;  
    System.out.println("Consumed: " + item);  
    notify();  
    return item;  
}  
}  
  
class Producer extends Thread {  
    private Buffer buffer;  
  
    Producer(Buffer buffer) {  
        this.buffer = buffer;  
    }  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            buffer.produce(i);  
        }  
    }  
}  
  
class Consumer extends Thread {  
    private Buffer buffer;  
  
    Consumer(Buffer buffer) {  
        this.buffer = buffer;  
    }  
  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            buffer.consume();  
        }  
    }  
}
```

```
}  
}
```

OUTPUT:

```
Produced: 0  
Produced: 1  
Produced: 2  
Produced: 3  
Produced: 4  
Consumed: 4  
Consumed: 3  
Consumed: 2  
Consumed: 1  
Consumed: 0  
Produced: 5  
Produced: 6  
Produced: 7  
Produced: 8  
Produced: 9  
Consumed: 9  
Consumed: 8  
Consumed: 7  
Consumed: 6  
Consumed: 5
```

CONCLUSION:

This program demonstrates the classic Producer-Consumer problem using multithreading and synchronization in Java. The producer generates items and adds them to a shared buffer, while the consumer removes and processes them. Synchronization ensures proper coordination between the threads.