Batch: E-2 **Roll No.: 16010123325**

Experiment / assignment / tutorial No.08

Grade: AA / AB / BB / BC / CC / CD /DD

Signature of the Staff In-charge with date

TITLE : Multithreading Programming

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

Expected OUTCOME of Experiment:

CO1: Understand the features of object oriented programming compared with procedural approach with C++ and Java

CO4: Explore the interface, exceptions, multithreading, packages.

Books/ Journals/ Websites referred:

1. Ralph Bravaco , Shai Simoson , “Java Programming From the Group Up” Tata McGraw-Hill.

2.Grady Booch, Object Oriented Analysis and Design .

Pre Lab/ Prior Concepts:

Java provides built-in support for multithreaded programming. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. A multithreading is a specialized form of multitasking. Multithreading requires less overhead than multitasking processing.

Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.

Creating a Thread:

Java defines two ways in which this can be accomplished:

1. You can implement the Runnable interface.
2. You can extend the Thread class itself.

Create Thread by Implementing Runnable:

The easiest way to create a thread is to create a class that implements the Runnable interface.

To implement Runnable, a class needs to only implement a single method called run( ), which is declared like this:

public void run( )

You will define the code that constitutes the new thread inside run() method. It is important to understand that run() can call other methods, use other classes, and declare variables, just like the main thread can.

After you create a class that implements Runnable, you will instantiate an object of type Thread from within that class. Thread defines several constructors. The one that we will use is shown here:

Thread(Runnable threadOb, String threadName);

Here, threadOb is an instance of a class that implements the Runnable interface and the name of the new thread is specified by threadName.

After the new thread is created, it will not start running until you call its start( ) method, which is declared within Thread. The start( ) method is shown here:

void start( );

Here is an example that creates a new thread and starts it running:

```
class NewThread implements Runnable {
  Thread t;
  NewThread() {
    t = new Thread(this, "Demo Thread");
    System.out.println("Child thread: " + t);
    t.start(); // Start the thread
  }
    public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Child Thread: " + i);
        // Let the thread sleep for a while.
        Thread.sleep(50);
      }
    } catch (InterruptedException e) {
      System.out.println("Child interrupted.");
    }
    System.out.println("Exiting child thread.");
  }
}

public class ThreadDemo {
  public static void main(String args[]) {
    new NewThread();
```

```
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Main Thread: " + i);
        Thread.sleep(100);
      }
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted.");
    }
    System.out.println("Main thread exiting.");
  }
}
```

The second way to create a thread is to create a new class that extends Thread, and then to create an instance of that class.
The extending class must override the run( ) method, which is the entry point for the new thread. It must also call start( ) to begin execution of the new thread.

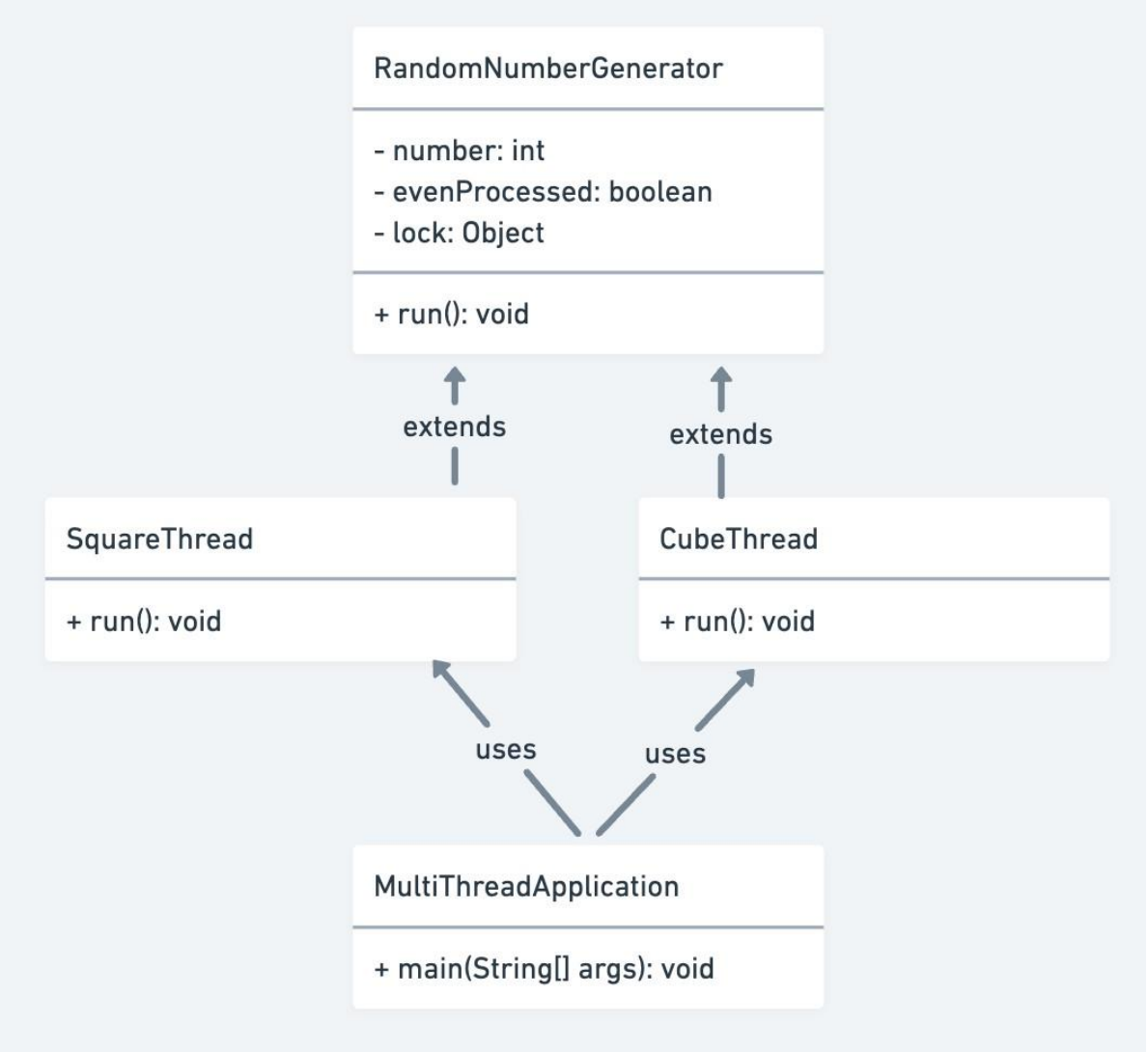
```
class NewThread extends Thread {
  NewThread() {
    super("Demo Thread");
    System.out.println("Child thread: " + this);
    start(); // Start the thread
  }
   public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Child Thread: " + i);
                // Let the thread sleep for a while.
        Thread.sleep(50);
      }
    } catch (InterruptedException e) {
      System.out.println("Child interrupted.");
    }
    System.out.println("Exiting child thread.");
  }
}

public class ExtendThread {
  public static void main(String args[]) {
    new NewThread(); // create a new thread
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Main Thread: " + i);
        Thread.sleep(100);
      }
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted.");
    }
    System.out.println("Main thread exiting.");
```

```
  }
}
```

Some of the Thread methods

Methods	Description
void setName(String name)	Changes the name of the Thread object. There is also a getName() method for retrieving the name
Void setPriority(int priority)	Sets the priority of this Thread object. The possible values are between 1 and 10. 5
boolean isAlive()	Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion.
void yield()	Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled.
void sleep(long millisec)	Causes the currently running thread to block for at least the specified number of milliseconds.
Thread currentThread()	Returns a reference to the currently running thread, which is the thread that invokes this method.

Class Diagram:



Algorithm:

1. **RandomNumberGenerator Thread**:
    - Generates random numbers in an infinite loop.
    - It checks whether the even number has been processed (evenProcessed flag) before generating the next number.
    - After generating a number, it notifies waiting threads (Square or Cube) to process it.

2. **SquareThread**:
    - Waits for an even number to be generated and processes it.
    - It calculates the square and then sets the evenProcessed flag to true, indicating the even number has been processed.

3. **CubeThread**:
    - Waits for an odd number and computes its cube.
    - Once processed, it sets the evenProcessed flag to true.

Synchronization:

- Threads use a shared lock (lock) to coordinate access to the shared number variable.
- Threads use wait() and notifyAll() to ensure only one thread is working on the number at a time based on its even or odd nature.

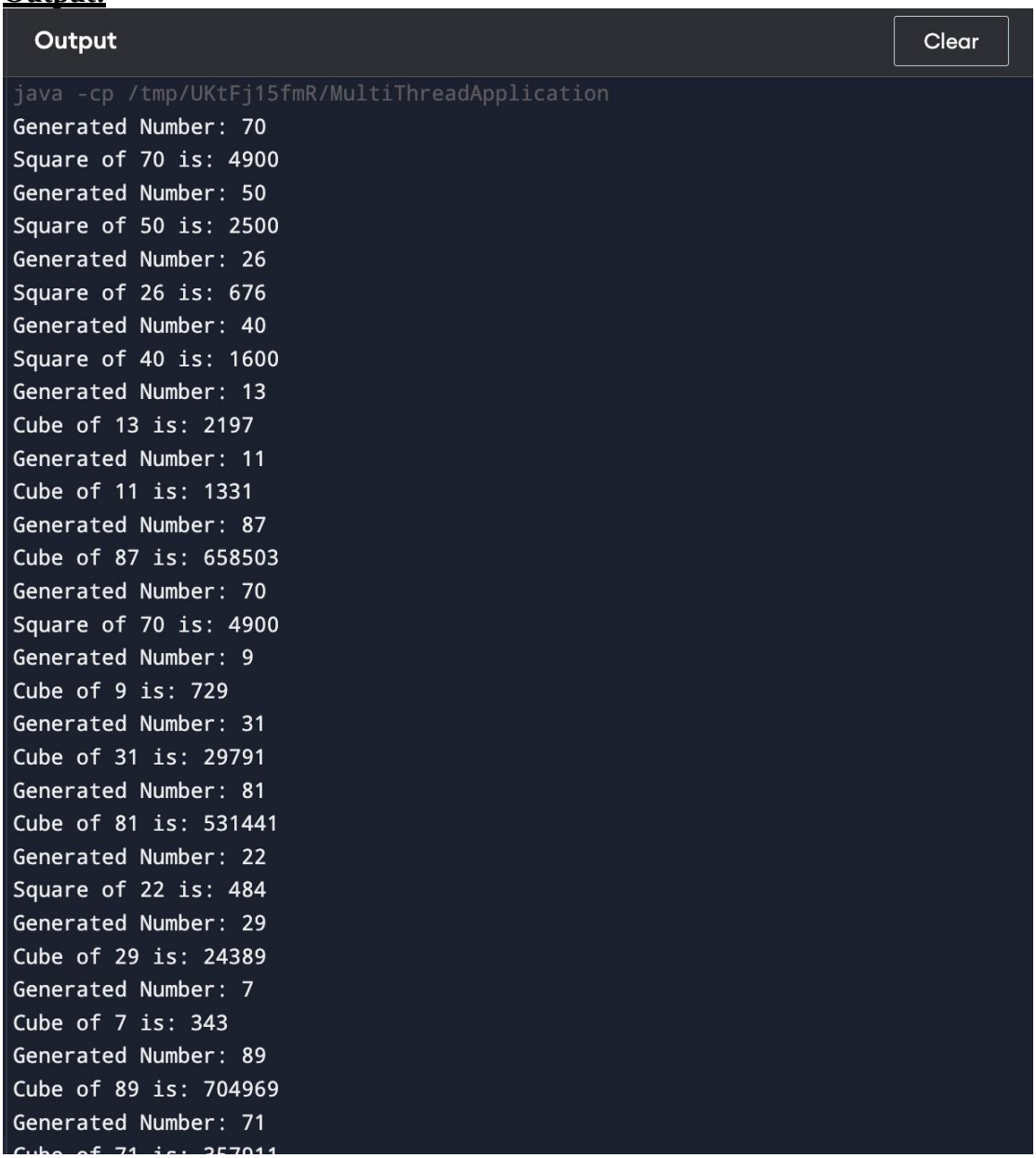
Implementation details:

```
import java.util.Random;

class RandomNumberGenerator extends Thread {
    public static int number;
    public static boolean evenProcessed = true;
    public static final Object lock = new Object();

    @Override
    public void run()
    {
        Random rand = new Random();
        while (true)
        {
            synchronized (lock)
            {
                while (!evenProcessed)
                {
                    try
                    {
                        lock.wait();
                    }
                    catch (InterruptedException e)
                    {
                        e.printStackTrace();
                    }
                }
                number = rand.nextInt(100);
                System.out.println("Generated Number: " + number);
                evenProcessed = false;
                lock.notifyAll();
            }
            try
            {
                Thread.sleep(1000);
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
    }
}

class SquareThread extends Thread
{
    @Override
    public void run() {
        while (true) {
            synchronized (RandomNumberGenerator.lock) {
```

```
                while (RandomNumberGenerator.number % 2 != 0 ||
RandomNumberGenerator.evenProcessed) {
                    try
                    {
                        RandomNumberGenerator.lock.wait();
                    }
                    catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                int square = RandomNumberGenerator.number *
RandomNumberGenerator.number;
                System.out.println("Square of " +
RandomNumberGenerator.number + " is: " + square);
                RandomNumberGenerator.evenProcessed = true;
                RandomNumberGenerator.lock.notifyAll();
            }
        }
    }
}

class CubeThread extends Thread
{
    @Override
    public void run()
    {
        while (true) {
            synchronized (RandomNumberGenerator.lock)
            {
                while (RandomNumberGenerator.number % 2 == 0 ||
RandomNumberGenerator.evenProcessed)
                {
                    try
                    {
                        RandomNumberGenerator.lock.wait();
                    }
                    catch (InterruptedException e)
                    {
                        e.printStackTrace();
                    }
                }
                int cube = RandomNumberGenerator.number *
RandomNumberGenerator.number * RandomNumberGenerator.number;
                System.out.println("Cube of " + RandomNumberGenerator.number
+ " is: " + cube);
                RandomNumberGenerator.evenProcessed = true;

                RandomNumberGenerator.lock.notifyAll();
            }
        }
```

```
    }
}

public class MultiThreadApplication
{
    public static void main(String[] args)
    {
        RandomNumberGenerator generator = new RandomNumberGenerator();
        SquareThread squareThread = new SquareThread();
        CubeThread cubeThread = new CubeThread();

        generator.start();
        squareThread.start();
        cubeThread.start();
    }
}
```

Output:


Output
Clear
java -cp /tmp/UKtFj15fmR/MultiThreadApplication
Generated Number: 70
Square of 70 is: 4900
Generated Number: 50
Square of 50 is: 2500
Generated Number: 26
Square of 26 is: 676
Generated Number: 40
Square of 40 is: 1600
Generated Number: 13
Cube of 13 is: 2197
Generated Number: 11
Cube of 11 is: 1331
Generated Number: 87
Cube of 87 is: 658503
Generated Number: 70
Square of 70 is: 4900
Generated Number: 9
Cube of 9 is: 729
Generated Number: 31
Cube of 31 is: 29791
Generated Number: 81
Cube of 81 is: 531441
Generated Number: 22
Square of 22 is: 484
Generated Number: 29
Cube of 29 is: 24389
Generated Number: 7
Cube of 7 is: 343
Generated Number: 89
Cube of 89 is: 704969
Generated Number: 71

Conclusion:

Hence, through the above experiment we learn the concept of multithreading and its application.

Date:_____ **Signature of faculty in-charge**

Post Lab Descriptive Questions

1. What do you mean by Thread Synchronization ? Why is it needed? Explain with a program.

Thread Synchronization ensures that only one thread at a time can access shared resources, preventing data inconsistency and race conditions when multiple threads modify shared data simultaneously. It makes operations thread-safe.

Why is it needed?

Without synchronization, multiple threads can modify the same data concurrently, leading to unpredictable results. Synchronization ensures that critical sections of the code are executed by only one thread at a time.

Code-

```
class Counter {
    int count = 0;
    public void increment() {
        count++;
    }
}

public class ThreadWithoutSync {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) counter.increment();
        });
        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) counter.increment();
        });

        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println("Final Counter Value: " + counter.count);
    }
}
```

Output-

Final Counter Value: 1965

2.Write a program for multithreaded Bank Account System

Implement a multithreaded bank account system in Java such that the system should simulate transactions on a bank account that can be accessed and modified by multiple threads concurrently. Your goal is to ensure that all transactions are handled correctly and that the account balance remains consistent.

```
class BankAccount
{
    private int balance;

    public BankAccount(int initialBalance)
    {
        this.balance = initialBalance;
    }

    public synchronized void deposit(int amount)
    {
        balance += amount;
        System.out.println(Thread.currentThread().getName() + " deposited " +
amount + ". Current balance: " + balance);
    }

    public synchronized void withdraw(int amount)
    {
        if (balance >= amount)
        {
            balance -= amount;
            System.out.println(Thread.currentThread().getName() + " withdrew
" + amount + ". Current balance: " + balance);
        }
        else
        {
            System.out.println(Thread.currentThread().getName() + " tried to
withdraw " + amount + " but insufficient balance.");
        }
    }

    public int getBalance()
    {
        return balance;
    }
}

class BankTransaction implements Runnable
{
    private BankAccount account;

    public BankTransaction(BankAccount account)
    {
```

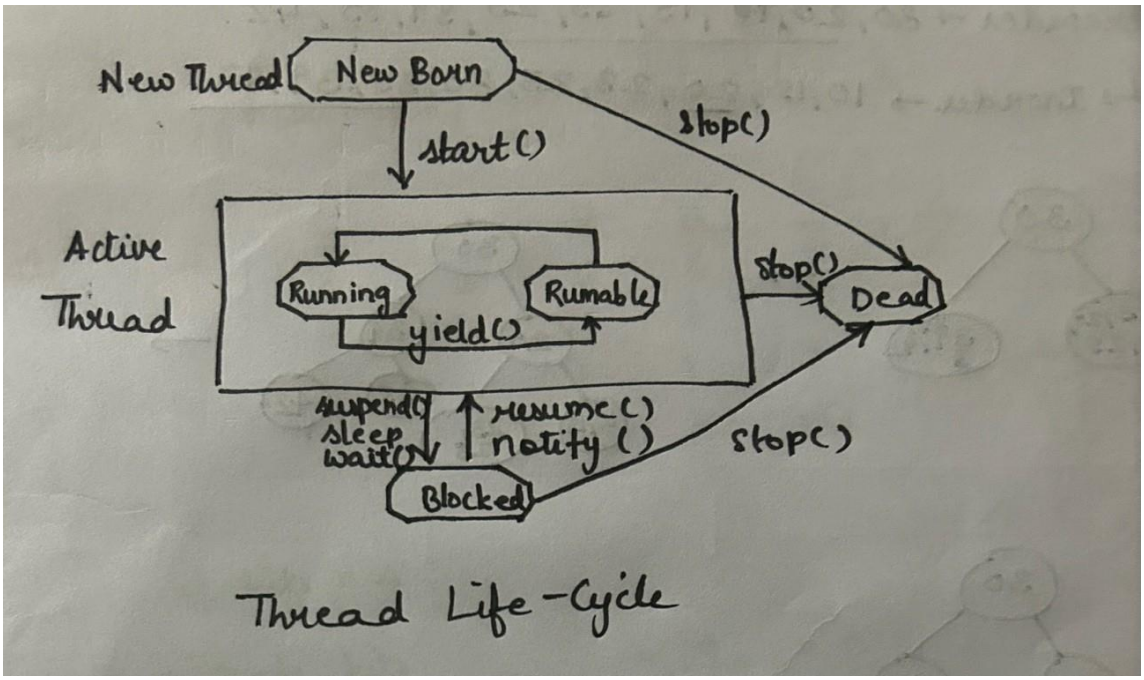
```
        this.account = account;
    }

    @Override
    public void run()
    {
        for (int i = 0; i < 5; i++)
        {
            int amount = (int) (Math.random() * 200) + 1;
            if (Math.random() > 0.5) {
                account.deposit(amount);
            }
            else
            {
                account.withdraw(amount);
            }

            try {
                Thread.sleep(100);
            }
            catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public class MultithreadedBankAccountSystem
{
    public static void main(String[] args)
    {
        BankAccount account = new BankAccount(1000);
        Thread t1 = new Thread(new BankTransaction(account), "User 1");
        Thread t2 = new Thread(new BankTransaction(account), "User 2");
        Thread t3 = new Thread(new BankTransaction(account), "User 3");

        t1.start();
        t2.start();
        t3.start();

        try {
            t1.join();
            t2.join();
            t3.join();
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Final Account Balance: " + account.getBalance());
```

```
    }
}
```

Output-

```
java -cp /tmp/YgaLeELQMx/MultithreadedBankAccountSystem
User 1 deposited 165. Current balance: 1165
User 2 withdrew 119. Current balance: 1046
User 3 deposited 85. Current balance: 1131
User 1 deposited 25. Current balance: 1156
User 2 withdrew 119. Current balance: 1037
User 3 withdrew 102. Current balance: 935
User 1 deposited 134. Current balance: 1069
User 2 deposited 178. Current balance: 1247
User 3 withdrew 52. Current balance: 1195
User 1 deposited 97. Current balance: 1292
User 2 deposited 141. Current balance: 1433
User 3 deposited 147. Current balance: 1580
User 1 deposited 189. Current balance: 1769
User 2 deposited 166. Current balance: 1935
User 3 withdrew 20. Current balance: 1915
Final Account Balance: 1915

=== Code Execution Successful ===
```

4. Draw thread lifecycle diagram. Explain any five methods of Thread class with Example ?



1. start()

Starts the thread, moving it from New to Runnable and executing run().

Eg:

```
class MyThread extends Thread {
   public void run() { System.out.println("Thread running..."); }
}
new MyThread().start();
```

2. run()

Contains the task to be executed by the thread.

Eg:

```
public void run() { System.out.println("Task in thread"); }
```

3. sleep()

Pauses the current thread for a given time.

Eg:

Thread.sleep(1000); // Pauses for 1 second

4. join()

Waits for the thread to **complete** before proceeding

Eg:

t1.join(); // Waits for t1 to finish

5. isAlive()

Returns true if the thread is still running.

Eg:

if (t1.isAlive()) { System.out.println("Thread is active"); }