Batch: D3	**Roll No.:**	**16010123294**
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:

- **Initialize Components**:
  - Create a `RandomNumberGenerator` thread to generate random integers.
  - Create a `NumberProcessor` class that handles processing even and odd numbers.
- **Random Number Generation**:
  - In the `RandomNumberGenerator` thread:
    1. Enter an infinite loop:
       - Generate a random integer between 0 and 99.
       - Print the generated integer.
       - Pass the generated integer to the `processNumber` method of the `NumberProcessor`.
       - Wait for 1 second before generating the next integer.
- **Processing Numbers**:
  - In the `NumberProcessor` class:
    1. Create two threads: one for processing even numbers and another for processing odd numbers.
    2. Start both threads.
- **Handle Number Processing**:
  - In the `processNumber` method:
    1. Check if the number is even or odd:
       - If even:
         - Call the method to process the even number.
         - Notify the even processing thread.
       - If odd:
         - Call the method to process the odd number.
         - Notify the odd processing thread.
- **Even and Odd Processing**:

- For even numbers:
    1. Calculate the square of the number.
    2. Print the square result.
- For odd numbers:
    1. Calculate the cube of the number.
    2. Print the cube result.

- **Thread Management**:

    - Ensure that both the even and odd processing threads remain active to continuously handle numbers as they are generated.

- **Termination**:

- The program runs indefinitely until manually stopped, processing random integers and printing results as specified.

Implementation details:

```
import java.util.Random;


class RandomNumberGenerator extends Thread {
   private final NumberProcessor processor;


   public RandomNumberGenerator(NumberProcessor processor) {
      this.processor = processor;
   }


   @Override
   public void run() {
      Random random = new Random();
      while (true) {
```

```
        int number = random.nextInt(100); // Generate random integer between 0 and
99
        System.out.println("Generated: " + number);
        processor.processNumber(number);
        try {
            Thread.sleep(1000); // Sleep for 1 second
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            break;
        }
    }
  }
}

class NumberProcessor {
  private final Thread evenThread;
  private final Thread oddThread;

  public NumberProcessor() {
    evenThread = new Thread(this::processEven);
    oddThread = new Thread(this::processOdd);
    evenThread.start();
    oddThread.start();
  }
```

```
public synchronized void processNumber(int number) {
    if (number % 2 == 0) {
        notifyAll(); // Notify the even thread
        processEvenNumber(number);
    } else {
        notifyAll(); // Notify the odd thread
        processOddNumber(number);
    }
}

private void processEvenNumber(int number) {
    try {
        while (!Thread.currentThread().getName().equals("EvenThread")) {
            wait();
        }
        System.out.println("Even Thread - Square: " + (number * number));
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
}

private void processOddNumber(int number) {
    try {
```

```
            while (!Thread.currentThread().getName().equals("OddThread")) {
                wait();
            }
            System.out.println("Odd Thread - Cube: " + (number * number * number));
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    private void processEven() {
        Thread.currentThread().setName("EvenThread");
        while (true) {
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }

    private void processOdd() {
        Thread.currentThread().setName("OddThread");
        while (true) {
            try {
```

```
            Thread.sleep(100);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
  }
}

public class MultiThreadedApp {
  public static void main(String[] args) {
    NumberProcessor processor = new NumberProcessor();
    RandomNumberGenerator generator = new RandomNumberGenerator(processor);
    generator.start();
  }
}
```

Output:

```
java -cp /tmp/5X8AI2K4MG/MultiThreadedApp
Generated: 20
```

Conclusion:

Learned to implement multithreading

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 is a method used in concurrent programming to ensure that multiple threads can safely access shared data or resources without causing inconsistencies or unexpected outcomes. It is essential because threads operate concurrently, and without adequate synchronization, they can interfere with one another, resulting in race conditions, data corruption, or unpredictable behavior.

Reasons Why Thread Synchronization is Necessary

1. Race Conditions: When multiple threads attempt to access and modify shared data simultaneously, the final result can vary depending on the order of execution, leading to unpredictable outcomes.

2. Data Consistency: Synchronization ensures the integrity of shared data by allowing only one thread to access it at any given time.

3. Deadlock Prevention: Effective synchronization mechanisms can help avoid situations where two or more threads are stuck waiting indefinitely for each other to release resources.

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.

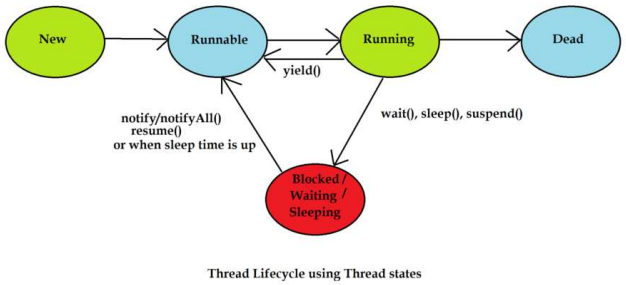
```
import java.util.Scanner;

class BankAccount {
   private double balance;

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

   // Synchronized method to handle deposits
   public synchronized void deposit(double amount) {
      if (amount > 0) {
         balance += amount;
         System.out.printf("Deposited: %.2f, New Balance: %.2f%n", amount, balance);
      }
   }

   // Synchronized method to handle withdrawals
   public synchronized void withdraw(double amount) {
      if (amount > 0 && amount <= balance) {
         balance -= amount;
         System.out.printf("Withdrew: %.2f, New Balance: %.2f%n", amount, balance);
      } else if (amount > balance) {
         System.out.printf("Withdrawal of %.2f denied. Insufficient funds. Current
Balance: %.2f%n", amount, balance);
      }
   }

   public double getBalance() {
      return balance;
   }
}

class TransactionThread extends Thread {
   private BankAccount account;
   private String transactionType;
   private double amount;

   public TransactionThread(BankAccount account, String transactionType, double
amount) {
      this.account = account;
```

```
        this.transactionType = transactionType;
        this.amount = amount;
    }

    @Override
    public void run() {
        if (transactionType.equals("deposit")) {
            account.deposit(amount);
        } else if (transactionType.equals("withdraw")) {
            account.withdraw(amount);
        }
    }
}

public class BankAccountSystem {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        BankAccount account = new BankAccount(1000.00); // Initial balance of $1000

        while (true) {
            System.out.print("Enter transaction type (deposit/withdraw) or 'exit' to quit: ");
            String transactionType = scanner.nextLine();

            if (transactionType.equalsIgnoreCase("exit")) {
                break;
            }

            System.out.print("Enter amount: ");
            double amount = scanner.nextDouble();
            scanner.nextLine(); // Consume newline

            TransactionThread transactionThread = new TransactionThread(account,
transactionType, amount);
            transactionThread.start();
        }

        // Wait for threads to finish (optional)
        // You can implement a more robust way to wait for all threads if needed.

        // Final account balance
        System.out.printf("Final Account Balance: %.2f%n", account.getBalance());
        scanner.close();
    }
}
```

```
Output                                                                    Clear
java -cp /tmp/PWXz5tA7ve/BankAccountSystem
Enter transaction type (deposit/withdraw) or 'exit' to quit: deposit
Enter amount: 300
Enter transaction type (deposit/withdraw) or 'exit' to quit: Deposited: 300.00, New Balance: 1300
    .00
withdraw
Enter amount: 800
Enter transaction type (deposit/withdraw) or 'exit' to quit: Withdrew: 800.00, New Balance: 500.00
```

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

Thread lifecyclye diagram



Thread Lifecycle using Thread states

start()

The `start()` method initiates the execution of a thread. When this method is called, it creates a new thread and calls the `run()` method within that thread. The `start()` method must be called to begin the thread's lifecycle.

2. run()

The `run()` method contains the code that defines the thread's task. When a thread is started using the `start()` method, the `run()` method is executed in a separate call stack. This method can be overridden to implement the desired behavior of the thread.

3. join()

The `join()` method allows one thread to wait for another thread to finish its execution. When a thread calls `join()` on another thread, it blocks the calling thread until the specified thread completes. This is useful for coordinating thread execution and ensuring that certain tasks are completed before others proceed.

4. sleep(long millis)

The `sleep()` method is a static method that pauses the execution of the current thread for a specified number of milliseconds. This can be used to introduce delays or to control the timing of thread execution. It can throw an `InterruptedException` if the thread is interrupted while sleeping.

5. setPriority(int priority)

The `setPriority()` method assigns a priority level to a thread. Thread priorities are used by the thread scheduler to determine the order in which threads are scheduled for execution. Priorities can range from `Thread.MIN_PRIORITY` (1) to `Thread.MAX_PRIORITY` (10), with `Thread.NORM_PRIORITY` (5) as the default. Higher-priority threads are generally scheduled to run before lower-priority threads, but this behavior can vary based on the underlying operating system