10. Threading in Java

Aim/Objective: To explore the basics of concurrent programming and understand the concepts of threading in Java.

Description: The student will gain an understanding of Java threading, encompassing the fundamentals of threads, their creation, synchronization, and communication, enabling concurrent execution of multiple tasks within a program.

Pre-Requisites: Knowledge on Classes and Objects, Understanding of threads and their execution model in JAVA.

Tools: Eclipse IDE for Enterprise Java and Web Developers

Pre-Lab:

1) How can you implement five functions (printTwo, printThree, printFour, printFive, and printNumber) using multiple threads to print numbers from 1 to 15, where each function prints a message if the number is divisible by 2, 3, 4, or 5, and printNumber prints the number if none of these conditions are met?
   Program:

```
public class NumberPrinter {

  private static int counter = 1;
  private static final int MAX_NUMBER = 15;

  public static void main(String[] args) {
    Runnable printTask = () -> {
      while (counter <= MAX_NUMBER) {
        synchronized (NumberPrinter.class) {
        if (counter % 2 == 0) System.out.println(counter + " is divisible by 2");
     else if (counter % 3 == 0) System.out.println(counter + " is divisible by 3");
     else if (counter % 4 == 0) System.out.println(counter + " is divisible by 4");
     else if (counter % 5 == 0) System.out.println(counter + " is divisible by 5");

          else System.out.println(counter);
          counter++;
        }
```

```
}
      };

      Thread t1 = new Thread(printTask);
      Thread t2 = new Thread(printTask);
      Thread t3 = new Thread(printTask);
      Thread t4 = new Thread(printTask);
      Thread t5 = new Thread(printTask);

      t1.start(); t2.start(); t3.start(); t4.start(); t5.start();

      try {
         t1.join(); t2.join(); t3.join(); t4.join(); t5.join();
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
   }
}
```

OUTPUT

```
1
2 is divisible by 2
3 is divisible by 3
4 is divisible by 2
5 is divisible by 5
6 is divisible by 2
7
8 is divisible by 2
9 is divisible by 3
10 is divisible by 2
11
12 is divisible by 2
13
14 is divisible by 2
15 is divisible by 3
```

In-Lab:

1) Design the Account Class
    - Create a class named Account in the bank package.
    - The Account class should have the following attributes:
        - int accountNumber
        - double balance
    - Provide methods for deposit(double amount) and withdraw(double amount) that update the balance accordingly.
2) Create a Thread Using the Runnable Interface:
    - Create a class named "TransactionRunnable" in the bank package that implements the "Runnable" interface.
    - This class should take an "Account" object and an "amount" as parameters, and perform either a deposit or withdrawal in its run method.
3) Create a Thread by Extending the Thread Class:
    - Create a class named "TransactionThread" in the bank package that extends the "Thread" class.
    - This class should also take an "Account" object and an "amount" as parameters, and perform either a deposit or withdrawal in its run method.
4) Running the Experiment:
    - Create a Main class in the "bank" package with a "main" method to execute the experiment.
    - Instantiate an Account object.
    - Create multiple threads using both "TransactionRunnable" and "TransactionThread" to perform concurrent deposits and withdrawals.

Expected Results:

- The Account balance should be updated correctly by each thread.
- The synchronized methods in the Account class ensure thread safety, preventing race conditions.
- The final balance should reflect all deposits and withdrawals performed by the threads.

Program:

```
class Account {
   private double balance;

   public synchronized void deposit(double amount) {
      balance += amount;
      System.out.println("Deposited: " + amount + ", Balance: " + balance);
   }

   public synchronized void withdraw(double amount) {
      if (balance >= amount) {
         balance -= amount;
         System.out.println("Withdrawn: " + amount + ", Balance: " + balance);
      } else {
         System.out.println("Insufficient funds for withdrawal: " + amount);
      }
   }

   public double getBalance() {
      return balance;
   }
}

class Transaction extends Thread {
   private final Account account;
   private final double amount;
   private final boolean isDeposit;

   public Transaction(Account account, double amount, boolean isDeposit) {
      this.account = account;
      this.amount = amount;
      this.isDeposit = isDeposit;
   }

   public void run() {
      if (isDeposit) account.deposit(amount);
      else account.withdraw(amount);
   }
```

```
}

public class Main {
   public static void main(String[] args) {
      Account account = new Account();
      Transaction[] transactions = {
         new Transaction(account, 1000, true),
         new Transaction(account, 500, false),
         new Transaction(account, 2000, true),
         new Transaction(account, 1500, false)
      };

      for (Transaction t : transactions) t.start();
      for (Transaction t : transactions) {
         try { t.join(); } catch (InterruptedException e) { e.printStackTrace(); }
      }

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

OUTPUT

```
Deposited: 1000.0, Balance: 1000.0
Deposited: 2000.0, Balance: 3000.0
Withdrawn: 1500.0, Balance: 1500.0
Withdrawn: 500.0, Balance: 1000.0
Final Account Balance: 1000.0
```

✓ **Data and Results:**

Data:

The bank account processes deposits and withdrawals using multiple threads.

Result:

Final balance reflects concurrent transactions executed in a multithreaded environment.

✓ **Analysis and Inferences:**

Analysis:

Synchronized methods ensure thread safety but may cause minor delays.

Inferences:

Proper synchronization prevents race conditions and maintains account balance integrity.

VIVA-VOCE Questions (In-Lab):

1) What is a thread? How does it enable concurrent execution in Java?

A thread is a lightweight unit of execution within a program. Java enables concurrent execution by running multiple threads in parallel, allowing tasks to be performed simultaneously, thus improving performance.

2) Describe the different ways to create a thread in Java.

- **Extend** `Thread` **class**: Override the `run()` method and call `start()`.
- **Implement** `Runnable` **interface**: Implement the `run()` method and pass the `Runnable` to a `Thread` object, then call `start()`.

3) What is thread safety? How do you achieve thread safety in Java?

Thread safety ensures correct behavior when multiple threads access shared data. It can be achieved using synchronized methods/blocks, locks (e.g., `ReentrantLock`), or atomic variables.

4) explain the terms "synchronization" and "thread safety" in the context of threading.

- **Synchronization**: Prevents concurrent access to a block of code by multiple threads.
- **Thread safety**: Ensures correct execution of code in a multithreaded environment.

5) How can you prevent race conditions in multithreaded programs?

1. **Mutex & Locks** – Restrict access to shared resources.
2. **Semaphores** – Manage thread execution with counters.
3. **Atomic Variables** – Ensure safe updates without locks.
4. **Monitors & Condition Variables** – Synchronize thread execution.
5. **Avoid Shared State** – Use thread-local storage or immutable data.

Post-Lab:

1) Write a JAVA program which will generate the threads: -
    - To display 10 terms of Fibonacci series.
    - To display 1 to 10 in reverse order.

Program:

```
public class Main {
   public static void main(String[] args) {
      Thread fibonacciThread = new Thread(new Fibonacci());
      Thread reverseThread = new Thread(new ReverseOrder());

      fibonacciThread.start();
      reverseThread.start();
   }
}

class Fibonacci implements Runnable {
   @Override
   public void run() {
      int n = 10, firstTerm = 0, secondTerm = 1;
      System.out.println("Fibonacci Series:");
      for (int i = 1; i <= n; ++i) {
         System.out.print(firstTerm + " ");
         int nextTerm = firstTerm + secondTerm;
         firstTerm = secondTerm;
         secondTerm = nextTerm;
      }
   }
}

class ReverseOrder implements Runnable {
   @Override
   public void run() {
      System.out.println("\nReverse Order:");
      for (int i = 10; i >= 1; i--) {
         System.out.print(i + " ");
      }
   }
}
```

OUTPUT

Fibonacci Series:
0 1 1 2 3 5 8 13 21 34
Reverse Order:
10 9 8 7 6 5 4 3 2 1

- ✓ **Data and Results:**

Data:

Program generates two threads for Fibonacci series and reverse order.

Result:

Fibonacci series and reverse order numbers are printed concurrently.

- ✓ **Analysis and Inferences:**

Analysis:

Multithreading enables parallel execution, improving efficiency and responsiveness.

Inferences:

Thread execution order may vary, demonstrating concurrent behavior in Java.

Evaluator Remark (if Any):	**Marks Secured _____ out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.