Experiment#		Student ID	
Date		Student Name	

2) 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:

```
class NumberPrinter{
    private final int MAX=15;

 public synchronized void printTwo(int number){
     if(number%2==0 && number%4!=0){
       System.out.println("Divisible by 2:"+number);
     }
 }
 public synchronized void printThree(int number){
     if(number%3==0){
       System.out.println("Divisible by 3:"+number);
     }
}
 public synchronized void printFour(int number){
     if(number%4==0){
       System.out.println("Divisible by 4:"+number);
     }
}
 public synchronized void printFive(int number){
     if(number%5==0){
       System.out.println("Divisible by 5:"+number);
     }
}
 public synchronized void printNumber(int number){
   if(number%2!=0 && number%3!=0 && number%4!=0 && number%5!=0){
```

```
        System.out.println("Number :"+number);
    }
}
}
public class Main {
    public static void main(String[] args){
        NumberPrinter numberPrinter = new NumberPrinter();
        Runnable taskTwo = () -> {
            for(int i=1; i<=15; i++) {
                numberPrinter.printTwo(i);
            }
    };
        Runnable taskThree = () -> {
            for(int i=1; i<=15; i++) {
                numberPrinter.printThree(i);
            }
    };
        Runnable taskFour = () -> {
            for(int i=1; i<=15; i++) {
                numberPrinter.printFour(i);
            }
    };
    Runnable taskFive =() -> {
            for(int i=1; i<=15; i++) {
                numberPrinter.printFive(i);
            }
    };
    Runnable taskPrintNumber =() -> {
            for(int i=1; i<=15; i++) {
```

```
            numberPrinter.printNumber(i);
        }
    };

    new Thread(taskTwo).start();
    new Thread(taskThree).start();
    new Thread(taskFour).start();
    new Thread(taskFive).start();
    new Thread(taskPrintNumber).start();
    }
}
```

Program:

1) Account class

```
package bank;

public class Account {
    private int accountNumber;
    private double balance;

    public Account(int accountNumber, double initialBalance) {
        this.accountNumber = accountNumber;
        this.balance = initialBalance;
    }
    public synchronized void deposit(double amount) {
        if(amount > 0) {
            balance += amount;
            System.out.println(Thread.currentThread().getName() + " deposited " +
            amount + ". Updated balance: " + balance);
        }
    }
    public synchronized void withdraw(double amount) {
        if(amount > 0 && balance >= amount) {
            balance -= amount;
            System.out.println(Thread.currentThread().getName() + " withdrew "
            + amount + ". Updated balance: " + balance);
        } else {
            System.out.println(Thread.currentThread().getName() + " attempted to
            withdraw " + amount + " but insufficient balance.");
        }
    }
    public double getBalance() {
        return balance;
    }
}
```

2) TransactionRunnable Class

```
package bank;

public class TransactionRunnable implements Runnable {
    private Account account;
    private double amount;
    private boolean deposit;

    public TransactionRunnable(Account account, double amount, boolean deposit) {
        this.account = account;
        this.amount = amount;
        this.deposit = deposit;
    }
    public void run() {
        if(deposit) {
            account.deposit(amount);
        }else {
            account.withdraw(amount);
        }
    }
}
```

3) TransactionThread Class

```
package bank;

public class TransactionThread extends Thread {
    private Account account;
    private double amount;
    private boolean deposit;

    public TransactionThread(Account account, double amount, boolean deposit) {
        this.account = account;
        this.amount = amount;
```

```
    this.deposit = deposit;

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

4) Main class

```
package bank;
public class Main {
    public static void main(String[] args) {
        Account account = new Account(123456, 1000);
        Runnable depositRunnable = new TransactionRunnable(account, 500, true);
        Runnable withdrawRunnable = new TransactionRunnable(account, 300, false);
        Thread t1 = new Thread(depositRunnable, "Runnable-1");
        Thread t2 = new Thread(withdrawRunnable, "Runnable-2");
        TransactionThread t3 = new TransactionThread(account, 200, true);
        TransactionThread t4 = new TransactionThread(account, 409, false);
        t1.start();
        t2.start();
        t3.start();
        t4.start();
```

```
try {
    t1.join();
    t2.join();
    t3.join();
    t4.join();
} catch (InterruptedException e) {
    e.printStackTrace();
}
System.out.println("Final Balance: "+ account.getBalance());
}
}
```

Experiment#		Student ID	
Date		Student Name	

✓ **Data and Results:**

Runnable-1 deposited 500. Updated Balance : 1500

Runnable-2 withdrew 300. Updated Balance: 1200

Thread-3 deposited 200. Updated Balance : 1400

Thread-4 withdrew 900. Updated Balance : 1000.

Final balance : 1000.

✓ **Analysis and Inferences:**

1) Account Class: Uses synchronized deposit and withdraw methods to ensure thread safety and prevent race conditions during concurrent transactions.

2) Transaction Runnable: Implements Runnable for flexible thread creation. Allows threads to either deposit or withdraw money based on a boolean flag.

3) Transaction Thread: Extends Thread for simple threading but less flexible than Runnable.

4) Concurrency: Synchronization ensures that only one thread modifies the account at a time, keeping the balance correct.

5) Final Outcome: The account balance is the accurate after all concurrent transactions complete.

Experiment#		Student ID	
Date		Student Name	

VIVA-VOCE Questions (In-Lab):

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

A thread is a small unit of execution in a program that enables concurrent tasks. In Java threads allow multiple tasks to run simultaneously, either by extending Thread or implementing Runnable, improving performance through parallel processing.

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

1) Extend Thread class : Override the run() method.

2) Implement Runnable : Implement Runnable and pass it to a Thread object.

3) Use ExecutorService : Manage threads using thread pools.

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

Thread safety ensures shared resources are accessed consistently by multiple threads without conflicts.

Achieve Thread Safety :

Synchronization : Use synchronized to lock code sections.

Atomic Variables : Use atomic classes for thread-safe operations.

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

Synchronization: Ensures only one thread accesses a shared resource at a time. to prevent data inconsistences.

Thread Safety: Guarantees correct behavior when multiple threads access shared resources. Achieved using synchronization, atomic operations, or thread-safe classes.

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

Preventing Race Conditions:

* Use synchronized blocks/methods.
* Use atomic classes like AtomicInteger.
* Use thread-safe classes (e.g., ConcurrentHashMap)
* Avoid shared mutable data or use locks for control.

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:

```
class FibonacciThread extends Thread {
    public void run() {
        int n1=0, n2=1;
        System.out.println("Fibonacci Series:");
        System.out.println(n1+" "+n2);
        for (int i=2; i<10; i++) {
```

```
        int next = n1 + n2;
        System.out.print(" " + next);
        n1 = n2;
        n2 = next;
    }
    System.out.println();
  }
}
class ReverseOrderThread extends Thread {
    public void run() {
        system.out.println("Numbers in reverse order:");
        for(int i=10; i>=1; i--) {
            System.out.print(i+" ");
        }
        System.out.println();
    }
}
public class ThreadExample{
    public static void main(String[] args) {
        FibonacciThread fibonacciThread = new FibonacciThread();
        ReverseOrderThread reverseOrderThread = new ReverseOrderThread();

        fibonacciThread.start();
        reverseOrderThread.start();
    }
}
```

Experiment#		Student ID	
Date		Student Name	

✓ **Data and Results:**

Fibonacci series:

0 1 1 2 3 5 8 13 21 34

Numbers in reverse order:

10 9 8 7 6 5 4 3 2 1

✓ **Analysis and Inferences:**

Analysis:

* Two threads are created: one to display the first 10 terms of the Fibonacci series and the other to print numbers from 10 to 1.
* Both threads run concurrently when started (start() method).

Inferences:

* The code demonstrates simple multithreading.
* The tasks run independently, with no risk of race conditions.

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.