6.(a) Write a Java program to implement Multiple inheritance with suitable objects

```java
// Interface for common banking functionality

interface BankOperations {
    void deposit(double amount);
    void withdraw(double amount);
}

// Interface for transferring funds between accounts
interface FundTransfer {
    void transfer(BankAccount toAccount, double amount);
}

// Interface for calculating interest
interface InterestCalculator {
    double calculateInterest();
}

// BankAccount class implementing BankOperations, FundTransfer, and
InterestCalculator
class BankAccount implements BankOperations, FundTransfer, InterestCalculator {
    private String accountNumber;
    private double balance;

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

    public String getAccountNumber() {
        return accountNumber;
    }

    public double getBalance() {
        return balance;
    }

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

    @Override
    public void withdraw(double amount) {
        if (amount <= balance) {
            balance -= amount;
            System.out.println("Withdrawn: " + amount);
        } else {
            System.out.println("Insufficient balance");
        }
    }

    @Override
    public void transfer(BankAccount toAccount, double amount) {
        if (amount <= balance) {
            withdraw(amount);
            toAccount.deposit(amount);
            System.out.println("Transfer successful");
        } else {
            System.out.println("Insufficient balance for transfer");
        }
    }

    @Override
    public double calculateInterest() {
        // Implement interest calculation logic here
        // For example, return balance * 0.05 for 5% interest
        return balance * 0.05;
    }
}

public class Main {
    public static void main(String[] args) {
        BankAccount account1 = new BankAccount("A12345", 1000);
        BankAccount account2 = new BankAccount("B67890", 2000);

        System.out.println("Account 1 balance: " + account1.getBalance());
        System.out.println("Account 2 balance: " + account2.getBalance());

        account1.deposit(500);
        System.out.println("Account 1 balance after deposit: " +
account1.getBalance());

        account2.withdraw(300);
        System.out.println("Account 2 balance after withdrawal: " +
account2.getBalance());

        account1.transfer(account2, 200);
        System.out.println("Account 1 balance after transfer: " +
account1.getBalance());
        System.out.println("Account 2 balance after transfer: " +
account2.getBalance());

        double interest = account1.calculateInterest();
        System.out.println("Interest earned by Account 1: " + interest);
    }
}
```

```java
// Interface for common banking functionality
interface BankOperations {
    void deposit(double amount);
    void withdraw(double amount);
}

// Interface for transferring funds between accounts
interface FundTransfer {
    void transfer(BankAccount toAccount, double amount);
}

// Interface for calculating interest
interface InterestCalculator {
    double calculateInterest();
}

// BankAccount class implementing BankOperations, FundTransfer, and
InterestCalculator
class BankAccount implements BankOperations, FundTransfer, InterestCalculator {
    private String accountNumber;
    private double balance;

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

    public String getAccountNumber() {
        return accountNumber;
    }

    public double getBalance() {
        return balance;
    }

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

    @Override
    public void withdraw(double amount) {
        if (amount <= balance) {
            balance -= amount;
            System.out.println("Withdrawn: " + amount);
        } else {
            System.out.println("Insufficient balance");
        }
    }
    @Override
    public void transfer(BankAccount toAccount, double amount) {
        if (amount <= balance) {
            withdraw(amount);
            toAccount.deposit(amount);
            System.out.println("Transfer successful");
        } else {
            System.out.println("Insufficient balance for transfer");
        }
    }

    @Override
    public double calculateInterest() {
        // Implement interest calculation logic here
        // For example, return balance * 0.05 for 5% interest
        return balance * 0.05;
    }
}

public class Main {
    public static void main(String[] args) {
        BankAccount account1 = new BankAccount("A12345", 1000);
        BankAccount account2 = new BankAccount("B67890", 2000);

        System.out.println("Account 1 balance: " + account1.getBalance());
        System.out.println("Account 2 balance: " + account2.getBalance());

        account1.deposit(500);
        System.out.println("Account 1 balance after deposit: " +
account1.getBalance());

        account2.withdraw(300);
        System.out.println("Account 2 balance after withdrawal: " +
account2.getBalance());

        account1.transfer(account2, 200);
        System.out.println("Account 1 balance after transfer: " +
account1.getBalance());
        System.out.println("Account 2 balance after transfer: " +
account2.getBalance());

        double interest = account1.calculateInterest();
        System.out.println("Interest earned by Account 1: " + interest);
    }
}
```
Algorithm:

This Java program defines a basic banking system with interfaces and a `BankAccount` class to demonstrate common banking operations. Let's break down the code step by step:

1. **Interfaces**:
   - Three interfaces are defined: `BankOperations`, `FundTransfer`, and `InterestCalculator`. These interfaces specify a set of methods related to common banking functionalities.

2. **BankAccount Class**:
   - The `BankAccount` class implements all three interfaces: `BankOperations`, `FundTransfer`, and `InterestCalculator`.
   - It has two private instance variables:
     - `accountNumber`: a string representing the account number.
     - `balance`: a double representing the account balance.

3. **Constructor**:
   - The class has a constructor that initializes the `accountNumber` and `balance` when a `BankAccount` object is created. It takes an account number and an initial balance as parameters.

4. **Getter Methods**:
   - The class provides getter methods to retrieve the account number and balance:
     - `getAccountNumber()`: Returns the account number.
     - `getBalance()`: Returns the account balance.

5. **Deposit Method**:
   - The `deposit` method, as required by the `BankOperations` interface, increases the account balance by the specified `amount` and prints a message indicating the deposit.

6. **Withdraw Method**:
   - The `withdraw` method, also required by the `BankOperations` interface, checks if there is a sufficient balance to withdraw the specified `amount`. If there is, it deducts the amount from the balance and prints a withdrawal message; otherwise, it prints an "Insufficient balance" message.

7. **Transfer Method**:
   - The `transfer` method, required by the `FundTransfer` interface, transfers funds from the current account to another account (`toAccount`) if there is a sufficient balance. It calls the `withdraw` and `deposit` methods as needed and provides transfer status messages.

8. **Interest Calculation Method**:
   - The `calculateInterest` method, as required by the `InterestCalculator` interface, calculates interest on the account's balance. The actual interest calculation logic is a placeholder and can be customized to fit the specific interest rate calculation required.

9. **Main Class**:
   - The `Main` class contains the `main` method, which is the entry point of the program.

   - Inside the `main` method, two `BankAccount` objects (`account1` and `account2`) are created with initial balances.
   - The program then performs various banking operations:
     - Depositing money into `account1`.
     - Withdrawing money from `account2`.
     - Transferring money from `account1` to `account2`.
     - Calculating the interest earned by `account1`.

10. **Output**:
   - The program prints messages indicating the current balance and the results of each operation.

This code demonstrates how to structure a basic banking system using interfaces and a class that implements those interfaces. The logic for interest calculation is a simple placeholder and can be customized as needed for a real banking system.

```
OUTPUT:
Account 1 balance: 1000.0
Account 2 balance: 2000.0
Deposited: 500.0
Account 1 balance after deposit: 1500.0
Withdrawn: 300.0
Account 2 balance after withdrawal: 1700.0
Withdrawn: 200.0
Deposited: 200.0
Transfer successful
Account 1 balance after transfer: 1300.0
Account 2 balance after transfer: 1900.0
Interest earned by Account 1: 65.0
```

==

6 (b) Write a Java program to search the element in an array.

```
import java.util.Scanner;

public class ArraySearch {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Input array elements
        System.out.print("Enter the number of elements in the array: ");
        int n = scanner.nextInt();

        int[] arr = new int[n];
        System.out.println("Enter the array elements:");
        for (int i = 0; i < n; i++) {
            arr[i] = scanner.nextInt();
        }
        // Input the element to search
        System.out.print("Enter the element to search: ");
        int target = scanner.nextInt();

        // Call the search method
        int index = searchElement(arr, target);

        if (index != -1) {
            System.out.println("Element " + target + " found at index " + index);
        } else {
            System.out.println("Element " + target + " not found in the array.");
        }

        scanner.close();
    }

    // Method to search for an element in the array
    public static int searchElement(int[] arr, int target) {
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] == target) {
                return i; // Return the index if found
            }
        }
        return -1; // Return -1 if not found
    }
}
```

Algorithm:

Sure, let's explain the Java program that searches for an element in an array step by step:

1. Importing Necessary Libraries:
   import java.util.Scanner;
   ```
   This line imports the `Scanner` class from the `java.util` package, which is used to read user input.

2. Main Method and Input:
   public static void main(String[] args) {
   ```
   The program starts with the `main` method, which is the entry point of the program.
   Scanner scanner = new Scanner(System.in);
   ```
   Here, a `Scanner` object is created to read user input.

3. Array Input:
   System.out.print("Enter the number of elements in the array: ");
   ```

~~~~
   int n = scanner.nextInt();
   ```
   The program prompts the user to enter the number of elements in the array and
reads this value as `n`.
   int[] arr = new int[n];
   System.out.println("Enter the array elements:");
   for (int i = 0; i < n; i++) {
       arr[i] = scanner.nextInt();
   }
   ```
   An integer array `arr` of size `n` is created to store the array elements. The
program then prompts the user to enter the array elements one by one and stores them
in the array.

4. Element to Search:

   System.out.print("Enter the element to search: ");
   int target = scanner.nextInt();
   ```
   The program prompts the user to enter the element to be searched and reads it as
`target`.

5. SearchElement Method:
   int index = searchElement(arr, target);
   ```
   The `searchElement` method is called to search for the element in the array. The
result (the index where the element was found) is stored in the `index` variable.

6. SearchElement Method:
   public static int searchElement(int[] arr, int target) {
   ```
   The `searchElement` method takes an integer array `arr` and an integer `target`
as parameters. It returns an integer representing the index where the `target`
element is found in the array or -1 if the element is not found.


   for (int i = 0; i < arr.length; i++) {
       if (arr[i] == target) {
           return i; // Return the index if found
       }
   }
   return -1; // Return -1 if not found
   ```
   This method iterates through the array using a `for` loop. If it finds an element
in the array that matches the `target`, it returns the index of that element. If no
match is found, it returns -1.

7. Result Output:

   if (index != -1) {
~~~~

~~~
        System.out.println("Element " + target + " found at index " + index);
    } else {
        System.out.println("Element " + target + " not found in the array.");
    }
    ```
~~~
Finally, the program checks the value of `index`. If it's not -1, it means the element was found, and it prints a message indicating the element and its index. If `index` is -1, it prints a message indicating that the element was not found.

8. Closing Scanner:

```
   scanner.close();
```

The `Scanner` object is closed to release system resources.

This program allows the user to input an array and search for a specific element in that array. It demonstrates how to use methods and loops to perform a search operation.

==

.   7.(a) Write a Java program to implement own exception and throw for input mark of student is out of range (0 to 100) and also catch own and build in exceptions

```
import java.util.Scanner;

// Custom Exception Class
class InvalidMarkException extends Exception {
    public InvalidMarkException(String message) {
        super(message);
    }
}

public class MarkValidation {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        try {
            System.out.print("Enter the student's mark (0 to 100): ");
            int mark = scanner.nextInt();

            // Check if the mark is out of range
            if (mark < 0 || mark > 100) {
                throw new InvalidMarkException("Invalid mark. Mark should be between
0 and 100.");
            }

            System.out.println("Student's mark: " + mark);

        } catch (InvalidMarkException e) {
            // Custom Exception Handling
```

```
            System.out.println("Custom Exception: " + e.getMessage());
        } catch (java.util.InputMismatchException e) {
            // InputMismatchException Handling (for non-integer inputs)
            System.out.println("InputMismatchException: Invalid input. Please enter
a valid integer.");
        } catch (Exception e) {
            // Generic Exception Handling
            System.out.println("Exception: " + e.getMessage());
        } finally {
            scanner.close();
        }
    }
}
```

Algorithm:

Certainly, let's break down the Java program that implements a custom exception for handling out-of-range student marks and demonstrates how to catch both the custom exception and built-in exceptions:

1. Importing Necessary Libraries:
```
   import java.util.Scanner;
```
   The program imports the `Scanner` class to read user input.

2. Custom Exception Class:
```
   class InvalidMarkException extends Exception {
       public InvalidMarkException(String message) {
           super(message);
       }
   }
```
   Here, a custom exception class `InvalidMarkException` is defined. It extends the built-in `Exception` class and allows you to throw and catch exceptions with a specific message.

3. `MarkValidation` Class:
```
   public class MarkValidation {
```
   The main class that contains the `main` method.

4. Inside the `main` Method:
   - We create a `Scanner` object to read user input.

5. `try` Block:
   - We prompt the user to enter a student's mark with the `System.out.print` statement.
   - We use `scanner.nextInt()` to read the integer value entered by the user and store it in the `mark` variable.

6. Mark Validation:
   - We check if the `mark` is out of range (i.e., less than 0 or greater than 100). If it is out of range, we throw a custom `InvalidMarkException` with a specific error message.

7. `catch` Blocks:
   - We catch and handle exceptions:
     - `InvalidMarkException`: This block handles the custom exception for out-of-range marks and prints a message with the exception message.
     - `InputMismatchException`: This block handles exceptions when the user enters a non-integer input and provides an error message.
     - `Exception`: This block serves as a generic exception handler for any other unhandled exceptions. It also displays an error message with the exception message.

8. `finally` Block:
   - We ensure that the `Scanner` object is closed in the `finally` block to release system resources.

This program allows the user to input a student's mark and handles exceptions if the input is out of range, not an integer, or if any other unexpected exception occurs. It demonstrates the use of custom exceptions and exception handling in Java.

Output:

```
Enter the student's mark (0 to 100): 101
Custom Exception: Invalid mark. Mark should be between 0 and 100.
Enter the student's mark (0 to 100): 99
Student's mark: 99
```

===

7.b)Write a Java program to implement multi-threaded program, to create three threads, in thread one to print perfect square, in thread two to print factorial of natural number, in thread three to print multiples of 23.

```java
class PerfectSquareThread extends Thread {
    public void run() {
        for (int i = 1; i <= 10; i++) {
            int square = i * i;
            System.out.println("Perfect Square of " + i + " is " + square);
        }
    }
}

class FactorialThread extends Thread {
    public void run() {
        for (int i = 1; i <= 10; i++) {
            int factorial = calculateFactorial(i);
            System.out.println("Factorial of " + i + " is " + factorial);
        }
    }

    private int calculateFactorial(int n) {
        if (n == 0 || n == 1) {
            return 1;
        } else {
            return n * calculateFactorial(n - 1);
        }
    }
}

class MultiplesOf23Thread extends Thread {
    public void run() {
        for (int i = 1; i <= 10; i++) {
            int multiple = i * 23;
            System.out.println("Multiple of 23 for " + i + " is " + multiple);
        }
    }
}

public class MultiThreadDemo {
    public static void main(String[] args) {
        PerfectSquareThread squareThread = new PerfectSquareThread();
        FactorialThread factorialThread = new FactorialThread();
        MultiplesOf23Thread multiplesOf23Thread = new MultiplesOf23Thread();

        squareThread.start();
        factorialThread.start();
        multiplesOf23Thread.start();
    }
}
```

Algorithm:

Certainly, let's explain the code for the multi-threaded program:

1. Thread Classes:
   - Three custom thread classes are defined: `PerfectSquareThread`, `FactorialThread`, and `MultiplesOf23Thread`. Each of these classes extends the `Thread` class and overrides the `run` method, which is the entry point for thread execution.

2. `PerfectSquareThread` Class:
   - This class is responsible for calculating and printing the perfect squares of numbers from 1 to 10.
   - Inside the `run` method, there is a loop that iterates from 1 to 10.
   - For each value of `i`, it calculates the square by multiplying `i` by itself (`i * i`) and then prints the result.

3. `FactorialThread` Class:
   - This class is responsible for calculating and printing the factorials of numbers from 1 to 10.
   - Inside the `run` method, there is a loop that iterates from 1 to 10.
   - For each value of `i`, it calculates the factorial using a recursive method `calculateFactorial(i)`.
   - The `calculateFactorial` method is a private helper method that computes the factorial of a number `n`. It uses recursion to calculate `n!` (n factorial), which is defined as `n! = n * (n-1)!`. The base case is when `n` is 0 or 1, in which case the method returns 1.

4. `MultiplesOf23Thread` Class:
   - This class is responsible for calculating and printing multiples of 23 for numbers from 1 to 10.
   - Inside the `run` method, there is a loop that iterates from 1 to 10.
   - For each value of `i`, it calculates the multiple of 23 by multiplying `i` by 23 and then prints the result.

5. `MultiThreadDemo` Class:
   - In the `main` method of the `MultiThreadDemo` class, we create instances of the three custom thread classes: `PerfectSquareThread`, `FactorialThread`, and `MultiplesOf23Thread`.

   - We start each thread by calling the `start()` method on the thread instances. This initiates the execution of the `run` method in each thread.

When you run the program, it will create three threads that run concurrently and perform their respective tasks:

- `PerfectSquareThread` prints perfect squares from 1 to 10.
- `FactorialThread` prints factorials of numbers from 1 to 10.
- `MultiplesOf23Thread` prints multiples of 23 for numbers from 1 to 10.

The output from each thread will be interleaved in the console, demonstrating parallel execution of the tasks.

OUTPUT:

```
Factorial of 1 is 1
Perfect Square of 1 is 1
Factorial of 2 is 2
Perfect Square of 2 is 4
Factorial of 3 is 6
Perfect Square of 3 is 9
Factorial of 4 is 24
Factorial of 5 is 120
Perfect Square of 4 is 16
Factorial of 6 is 720
Perfect Square of 5 is 25
Factorial of 7 is 5040
```

```
Perfect Square of 6 is 36
Factorial of 8 is 40320
Perfect Square of 7 is 49
Factorial of 9 is 362880
Perfect Square of 8 is 64
Factorial of 10 is 3628800
Perfect Square of 9 is 81
Perfect Square of 10 is 100
Multiple of 23 for 1 is 23
Multiple of 23 for 2 is 46
Multiple of 23 for 3 is 69
Multiple of 23 for 4 is 92
Multiple of 23 for 5 is 115
Multiple of 23 for 6 is 138
Multiple of 23 for 7 is 161
Multiple of 23 for 8 is 184
Multiple of 23 for 9 is 207
Multiple of 23 for 10 is 230
```

===
====================================

8.a)Write a Java program to implement own exception and throw for input grade of student is out of range (O,A+,A,B+,B,C,U) and also catch own and build in exceptions

```java
import java.util.Scanner;

// Custom Exception Class
class InvalidGradeException extends Exception {
    public InvalidGradeException(String message) {
        super(message);
    }
}

public class GradeValidation {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        try {
            System.out.print("Enter the student's grade (O, A+, A, B+, B, C, U): ");
            String grade = scanner.next().toUpperCase();

            // Check if the grade is out of range
            if (!isValidGrade(grade)) {
                throw new InvalidGradeException("Invalid grade. Grade should be O, 
A+, A, B+, B, C, or U.");
            }

            System.out.println("Student's grade: " + grade);
        } catch (InvalidGradeException e) {
            // Custom Exception Handling
            System.out.println("Custom Exception: " + e.getMessage());
        } catch (Exception e) {
            // Generic Exception Handling (for other exceptions)
            System.out.println("Exception: " + e.getMessage());
        } finally {
            scanner.close();
        }
    }

    // Custom method to check if the grade is valid
    private static boolean isValidGrade(String grade) {
        String[] validGrades = {"O", "A+", "A", "B+", "B", "C", "U"};
        for (String validGrade : validGrades) {
            if (validGrade.equals(grade)) {
                return true;
            }
        }
        return false;
    }
}
```

Algorithm:

This Java program, "GradeValidation," is designed to validate a student's grade input and handle custom exceptions for invalid grades. It also demonstrates how to catch both custom and built-in exceptions. Here's a line-by-line explanation of the code:

1. Importing Necessary Packages:
   ```java
   import java.util.Scanner;
   ```
   This line imports the `Scanner` class from the `java.util` package for user input.

2. Custom Exception Class (`InvalidGradeException`):
   ```java
   class InvalidGradeException extends Exception {
       public InvalidGradeException(String message) {
           super(message);
       }
   }
   ```
   - The code defines a custom exception class, `InvalidGradeException`, which extends the built-in `Exception` class. This class allows you to create custom exceptions with a specific error message.

3. `main` Method:

```java
   public static void main(String[] args) {
       Scanner scanner = new Scanner(System.in);
       // ...
   }
   ```
   - The program's main method begins here.
   - It initializes a `Scanner` object for user input.

4. Try-Catch Block:
   ```java
   try {
       // ...
   } catch (InvalidGradeException e) {
       // ...
   } catch (Exception e) {
       // ...
   } finally {
       scanner.close();
   }
   ```
   - The code contains a try-catch-finally block for handling exceptions.
   - Within the `try` block, the program does the following:
     - Prompts the user to enter a student's grade (O, A+, A, B+, B, C, or U) and converts it to uppercase.
     - Checks if the entered grade is valid using the `isValidGrade` method.
     - If the grade is not valid, it throws a custom exception `InvalidGradeException` with an appropriate error message.
     - It then displays the student's grade.
   - In the `catch` block:
     - Custom Exception Handling: If an `InvalidGradeException` is thrown, the program catches it and displays a custom error message.
     - Generic Exception Handling: If any other exception is thrown, it catches it as a general `Exception` and displays a generic error message.
   - The `finally` block ensures that the `Scanner` object is closed to release system resources.

5. Custom Method `isValidGrade`:
   ```java
   private static boolean isValidGrade(String grade) {
       // ...
   }
   ```
   - The `isValidGrade` method is used to check if the entered grade is valid.
   - It compares the input grade with an array of valid grades (`"O", "A+", "A", "B+", "B", "C", "U"`) and returns `true` if the grade is valid, otherwise `false`.

6. Running the Program:
   ```java
   public static void main(String[] args) {
   ```

```
        // ...
    }
    ```
```
   - The `main` method initializes the program, where the user can enter a student's grade, and the program validates the input.

When you run the program, it will prompt you to enter a student's grade. If the grade is within the valid range (O, A+, A, B+, B, C, U), it will display the grade. If you enter an invalid grade, it will throw and catch the custom exception `InvalidGradeException` and display a custom error message. For any other exceptions, it will display a generic error message.

OUTPUT:

```
Enter the student's grade (O, A+, A, B+, B, C, U): c
Student's grade: C
Enter the student's grade (O, A+, A, B+, B, C, U): l
Custom Exception: Invalid grade. Grade should be O, A+, A, B+, B, C, or U.

===
==============================

8.(b)   Write a Java program to print different series like square & cube of N numbers using multi-threading

```java
// Class to print the square of N numbers
class SquareThread extends Thread {
    private int N;

    public SquareThread(int N) {
        this.N = N;
    }

    public void run() {
        System.out.println("Squares of N numbers:");
        for (int i = 1; i <= N; i++) {
            System.out.println("Square of " + i + " is " + (i * i));
        }
    }
}

// Class to print the cube of N numbers
class CubeThread extends Thread {
    private int N;

    public CubeThread(int N) {
        this.N = N;
    }

    public void run() {
        System.out.println("Cubes of N numbers:");
```

```
        for (int i = 1; i <= N; i++) {
            System.out.println("Cube of " + i + " is " + (i * i * i));
        }
    }
}

public class SeriesPrinting {
    public static void main(String[] args) {
        int N = 10; // Define the range of N numbers

        // Create two threads, one for squares and one for cubes
        SquareThread squareThread = new SquareThread(N);
        CubeThread cubeThread = new CubeThread(N);

        // Start both threads
        squareThread.start();
        cubeThread.start();

        // Wait for both threads to finish
        try {
            squareThread.join();
            cubeThread.join();
        } catch (InterruptedException e) {
            System.out.println("Thread interrupted: " + e.getMessage());
        }

        System.out.println("Main thread has finished.");
    }
}
```

Algorithm:
To print different series like squares and cubes of N numbers using multi-threading in Java, we'll create two separate threads: one for printing squares and another for printing cubes. Here's the Java program to achieve this with a line-by-line explanation:

```
// Create two threads, one for squares and one for cubes
        SquareThread squareThread = new SquareThread(N);
        CubeThread cubeThread = new CubeThread(N);
```

Now, let's explain the code step by step:

1. `SquareThread` Class:
   - This class extends `Thread` and is responsible for printing the squares of N numbers.
   - It takes an integer `N` as a parameter in its constructor to specify the range.
   - The `run` method is overridden to define the behavior of this thread. It prints the squares of numbers from 1 to `N`.

2. `CubeThread` Class:
   - Similar to the `SquareThread` class, this class is responsible for printing the

cubes of N numbers.
   - It also takes an integer `N` as a parameter in its constructor.
   - The `run` method prints the cubes of numbers from 1 to `N`.

3. `SeriesPrinting` Class (Main Class):
   - The `main` method initializes the program.
   - It defines the value of `N` to determine the range of numbers for which squares and cubes will be printed.

4. Creating Threads:
   - Two threads, `squareThread` and `cubeThread`, are created to handle the printing of squares and cubes, respectively.
   - Each thread is initialized with the value of `N` for its specific series.

5. Starting Threads:
   - `squareThread` and `cubeThread` are started using the `start` method.

6. Waiting for Threads to Finish:
   - The `join` method is used to wait for both threads to finish before the main thread continues.
   - This ensures that the series are printed sequentially.

7. Exception Handling:
   - The program catches and handles any `InterruptedException` that may occur during thread execution.

8. Completion Message:
   - Once both threads have finished, the program prints a message to indicate that the main thread has finished.

When you run the program, it will print the squares and cubes of numbers from 1 to `N` in separate threads, ensuring that both series are printed concurrently.

==

9.a)9. implement own exception and throw for input age of voter is out of range (<18) and also catch own and build in exception

```
import java.util.Scanner;

// Custom Exception Class for Age Out of Range
class AgeOutOfRangeException extends Exception {
    public AgeOutOfRangeException(String message) {
        super(message);
    }
}

public class VoterAgeValidation {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        try {
            System.out.print("Enter the age of the voter: ");
            int age = scanner.nextInt();

            // Check if the age is out of range
            if (age < 18) {
                throw new AgeOutOfRangeException("Invalid age. Voter must be at
least 18 years old.");
            }

            System.out.println("Voter's age: " + age);
        } catch (AgeOutOfRangeException e) {
            // Custom Exception Handling
            System.out.println("Custom Exception: " + e.getMessage());
        } catch (java.util.InputMismatchException e) {
            // InputMismatchException Handling (for non-integer inputs)
            System.out.println("InputMismatchException: Invalid input. Please enter
a valid integer.");
        } catch (Exception e) {
            // Generic Exception Handling (for other exceptions)
            System.out.println("Exception: " + e.getMessage());
        } finally {
            scanner.close();
        }
    }
}
```

Algorithm

We create the AgeOutOfRangeException class, which extends Exception, to handle the custom age out-of-range exception.

The user is prompted to enter their age, and we check if the age is out of range (less than 18).

If the age is out of range, we throw the AgeOutOfRangeException with a custom error message.

We catch the AgeOutOfRangeException exception and handle it with a specific message. We also handle InputMismatchException (for non-integer inputs) and generic exceptions.

The finally block is used to ensure that the Scanner is closed, whether or not an exception is thrown.

This program allows you to validate the age of a voter and handle exceptions accordingly.

OUTPUT:

```
}
```

```
Enter the age of the voter: 18
Voter's age: 18
Enter the age of the voter: 14
Custom Exception: Invalid age. Voter must be at least 18 years old.
```

==
============================

9.b)(b) Write a Java program to print different series like fibonacci & multiple of 17 using multi-threading

```
// Class to print the Fibonacci series
class FibonacciThread extends Thread {
    private int n;

    public FibonacciThread(int n) {
        this.n = n;
    }

    public void run() {
        int first = 0, second = 1;
        System.out.println("Fibonacci Series (First " + n + " terms):");
        for (int i = 0; i < n; i++) {
            System.out.print(first + " ");
            int next = first + second;
            first = second;
            second = next;
        }
        System.out.println();
    }
}

// Class to print multiples of 17
class MultiplesOf17Thread extends Thread {
    private int n;

    public MultiplesOf17Thread(int n) {
        this.n = n;
    }

    public void run() {
        System.out.println("Multiples of 17 (First " + n + " terms):");
        for (int i = 1; i <= n; i++) {
            int multiple = 17 * i;
            System.out.print(multiple + " ");
        }
        System.out.println();
    }
}
```

```
public class MultiThreadSeriesDemo {
    public static void main(String[] args) {
        int n = 10; // Number of terms to print

        // Create threads for both series
        FibonacciThread fibonacciThread = new FibonacciThread(n);
        MultiplesOf17Thread multiplesOf17Thread = new MultiplesOf17Thread(n);

        // Start both threads
        fibonacciThread.start();
        multiplesOf17Thread.start();
    }
}
```

Algorithm:

We have two custom thread classes: FibonacciThread and MultiplesOf17Thread, each responsible for printing a specific series.

The FibonacciThread class calculates and prints the Fibonacci series for the first n terms.

The MultiplesOf17Thread class prints the first n multiples of 17.

In the main method, we specify the number of terms n we want to print for both series. You can change this value to print more or fewer terms.

We create instances of the two thread classes: fibonacciThread and multiplesOf17Thread.

We start both threads using the start() method. This allows both series to be printed concurrently.

When you run this program, you'll see the Fibonacci series and multiples of 17 printed concurrently by the two threads. The use of multi-threading allows the series to be generated in parallel, improving efficiency when dealing with time-consuming tasks.

OUTPUT:

```
Fibonacci Series (First 10 terms):
Multiples of 17 (First 10 terms):
0 17 1 34 1 2 3 51 68 85 5 102 119 8 136 13 153 170 21
34
```

==

10.a)10.   (a) Write a Java program to implement dynamic dispatching for calculating area of different shapes

```java
// Interface for Shape
interface Shape {
    // Method to calculate the area
    double calculateArea();
}

// Circle class implementing Shape
class Circle implements Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public double calculateArea() {
        return Math.PI * radius * radius;
    }
}

// Rectangle class implementing Shape
class Rectangle implements Shape {
    private double length;
    private double width;

    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    @Override
    public double calculateArea() {
        return length * width;
    }
}

// Triangle class implementing Shape
class Triangle implements Shape {
    private double base;
    private double height;

    public Triangle(double base, double height) {
        this.base = base;
        this.height = height;
    }

    @Override
    public double calculateArea() {
        return 0.5 * base * height;
    }
}

public class AreaCalculator {
    public static void main(String[] args) {
        // Create instances of different shapes
        Shape circle = new Circle(5.0);
        Shape rectangle = new Rectangle(4.0, 6.0);
        Shape triangle = new Triangle(3.0, 8.0);

        // Calculate and display the areas of different shapes
        System.out.println("Area of Circle: " + circle.calculateArea());
        System.out.println("Area of Rectangle: " + rectangle.calculateArea());
        System.out.println("Area of Triangle: " + triangle.calculateArea());
    }
}
```

Algorithm:

We define an interface Shape with a method calculateArea().

We create classes for specific shapes (e.g., Circle, Rectangle, Triangle) that implement the Shape interface. Each class provides its own implementation of the calculateArea() method.

In the main method, we create instances of different shape classes (circle, rectangle, and triangle) using the Shape interface reference.

We call the calculateArea() method on each shape object, and dynamic dispatch ensures that the appropriate calculateArea() method of the specific shape class is called.

The program displays the calculated areas of the circle, rectangle, and triangle.

Using interfaces in this way allows you to define a common contract (Shape interface) that classes must adhere to, promoting loose coupling and making your code more extensible and maintainable.

OUTPUT:

```
Area of Circle: 78.53981633974483
Area of Rectangle: 24.0
Area of Triangle: 12.0
```

===

10.b)Write a Java program to check the given number is Armstrong number or not.

```
import java.util.Scanner;
```

```
public class ArmstrongNumberChecker {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter a number: ");
        int number = scanner.nextInt();

        if (isArmstrongNumber(number)) {
            System.out.println(number + " is an Armstrong number.");
        } else {
            System.out.println(number + " is not an Armstrong number.");
        }

        scanner.close();
    }

    // Function to check if a number is an Armstrong number
    public static boolean isArmstrongNumber(int num) {
        int originalNumber = num;
        int numDigits = countDigits(num);
        int sum = 0;

        while (num > 0) {
            int digit = num % 10;
            sum += Math.pow(digit, numDigits);
            num /= 10;
        }

        return sum == originalNumber;
    }

    // Function to count the number of digits in a number
    public static int countDigits(int num) {
        int count = 0;
        while (num != 0) {
            num /= 10;
            count++;
        }
        return count;
    }
}
```

Algorithm:

Sure, I'll explain the algorithm used in the Java program to check if a number is an Armstrong number.

Armstrong Number (also known as a narcissistic number or a pluperfect digital invariant) is a number that is equal to the sum of its own digits raised to the power of the number of digits.

Here's the step-by-step algorithm:

1. **User Input**: The program starts by taking input from the user. The user is asked to enter an integer, which is stored in the variable `number`.

2. **isArmstrongNumber Function**:
   - The program has a function named `isArmstrongNumber`, which takes an integer `num` as a parameter and returns a boolean value.
   - In this function:
     - A copy of the original number, `originalNumber`, is created to compare with the result later.
     - The function counts the number of digits in `num` using the `countDigits` function.
     - A variable `sum` is initialized to zero to accumulate the sum of digits raised to the power of `numDigits`.
     - The program enters a `while` loop, which iteratively processes each digit in the number:
     - `digit` stores the last digit of `num` (obtained using `num % 10`).
     - `sum` is incremented by the digit raised to the power of `numDigits` using the `Math.pow` function.
     - The last digit is removed from `num` by integer division (`num /= 10`).
   - After exiting the loop, the program compares `sum` with `originalNumber`. If they are equal, the number is an Armstrong number, and `true` is returned. Otherwise, `false` is returned.

3. **countDigits Function**:
   - The `countDigits` function is used to count the number of digits in an integer. It takes an integer `num` as a parameter and returns an integer.
   - The function initializes a `count` variable to zero and enters a `while` loop.
   - In each iteration, it divides `num` by 10 and increments `count` by 1.
   - This loop continues until `num` becomes zero, and then the `count` is returned.

4. **Output**:
   - The program then prints the result. If the `isArmstrongNumber` function returns `true`, it means the number is an Armstrong number, and the program prints "is an Armstrong number." If it returns `false`, the program prints "is not an Armstrong number."

5. **Cleanup**:
   - Finally, the `Scanner` is closed to release system resources after input reading.

The program efficiently determines if a number is an Armstrong number by checking the digit-wise sum, which is compared with the original number. If they match, it's an Armstrong number. Otherwise, it's not.

Output:

```
Enter a number: 145
145 is not an Armstrong number.
Enter a number: 153
153 is an Armstrong number.
```