Program 01 : Matrix Addition

Develop a JAVA program to add TWO matrices of suitable order N (The value of N should be read from command line arguments).

Java Code

```
public class MatrixAddition {
    public static void main(String[] args) {
        // Check if the number of command line arguments is correct
        if (args.length != 1) {
            System.out.println("Usage: java MatrixAddition <order_N>");
            return;
        }

        // Parse the command line argument to get the order N
        int N = Integer.parseInt(args[0]);

        // Check if N is a positive integer
        if (N <= 0) {
            System.out.println("Please provide a valid positive integer for the order N.");
            return;
        }

        // Create two matrices of order N
        int[][] matrix1 = new int[N][N];
        int[][] matrix2 = new int[N][N];

        // Fill the matrices with some sample values (you can modify this as needed)
        fillMatrix(matrix1, 1);
        fillMatrix(matrix2, 2);

        // Print the matrices
        System.out.println("Matrix 1:");
```

```
        printMatrix(matrix1);

        System.out.println("\nMatrix 2:");
        printMatrix(matrix2);

        // Add the matrices
        int[][] resultMatrix = addMatrices(matrix1, matrix2);

        // Print the result matrix
        System.out.println("\nResultant Matrix (Matrix1 + Matrix2):");
        printMatrix(resultMatrix);
    }

    // Helper method to fill a matrix with sequential values
    private static void fillMatrix(int[][] matrix, int startValue) {
        int value = startValue;
        for (int i = 0; i < matrix.length; i++) {
            for (int j = 0; j < matrix[i].length; j++) {
                matrix[i][j] = value++;
            }
        }
    }

    // Helper method to add two matrices
    private static int[][] addMatrices(int[][] matrix1, int[][] matrix2) {
        int N = matrix1.length;
        int[][] resultMatrix = new int[N][N];

        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                resultMatrix[i][j] = matrix1[i][j] + matrix2[i][j];
```

```
            }
        }

        return resultMatrix;
    }

    // Helper method to print a matrix
    private static void printMatrix(int[][] matrix) {
        for (int[] row : matrix) {
            for (int value : row) {
                System.out.print(value + "\t");
            }
            System.out.println();
        }
    }
}
```

In this example, the matrices are filled with sequential values for simplicity, but you can modify the **fillMatrix** method to fill the matrices with any values you prefer.

Output

```
$ java MatrixAddition 3

Matrix 1:
1       2       3
4       5       6
7       8       9

Matrix 2:
2       3       4
5       6       7
8       9       10
```

Resultant Matrix (Matrix1 + Matrix2):

3 5 7

9 11 13

15 17 19

Program 02 : Stack Operations

Develop a stack class to hold a maximum of 10 integers with suitable methods. Develop a JAVA main method to illustrate Stack operations.

Java Code

```
import java.util.Scanner;


public class Stack {
    private static final int MAX_SIZE = 10;
    private int[] stackArray;
    private int top;

    public Stack() {
        stackArray = new int[MAX_SIZE];
        top = -1;
    }

    public void push(int value) {
        if (top < MAX_SIZE - 1) {
            stackArray[++top] = value;
            System.out.println("Pushed: " + value);
        } else {
            System.out.println("Stack Overflow! Cannot push " + value + ".");
        }
    }

    public int pop() {
```

```
    if (top >= 0) {
        int poppedValue = stackArray[top--];
        System.out.println("Popped: " + poppedValue);
        return poppedValue;
    } else {
        System.out.println("Stack Underflow! Cannot pop from an empty stack.");
        return -1; // Return a default value for simplicity
    }
}

public int peek() {
    if (top >= 0) {
        System.out.println("Peeked: " + stackArray[top]);
        return stackArray[top];
    } else {
        System.out.println("Stack is empty. Cannot peek.");
        return -1; // Return a default value for simplicity
    }
}

public void display() {
    if (top >= 0) {
        System.out.print("Stack Contents: ");
        for (int i = 0; i <= top; i++) {
            System.out.print(stackArray[i] + " ");
        }
        System.out.println();
    } else {
        System.out.println("Stack is empty.");
    }
}
```

```
public boolean isEmpty() {
    return top == -1;
}

public boolean isFull() {
    return top == MAX_SIZE - 1;
}

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

    int choice;

    do {
        System.out.println("\nStack Menu:");
        System.out.println("1. Push");
        System.out.println("2. Pop");
        System.out.println("3. Peek");
        System.out.println("4. Display Stack Contents");
        System.out.println("5. Check if the stack is empty");
        System.out.println("6. Check if the stack is full");
        System.out.println("0. Exit");

        System.out.print("Enter your choice: ");
        choice = scanner.nextInt();

        switch (choice) {
            case 1:
                System.out.print("Enter the value to push: ");
```

```
                int valueToPush = scanner.nextInt();
                stack.push(valueToPush);
                break;
            case 2:
                stack.pop();
                break;
            case 3:
                stack.peek();
                break;
            case 4:
                stack.display();
                break;
            case 5:
                System.out.println("Is the stack empty? " + stack.isEmpty());
                break;
            case 6:
                System.out.println("Is the stack full? " + stack.isFull());
                break;
            case 0:
                System.out.println("Exiting the program. Goodbye!");
                break;
            default:
                System.out.println("Invalid choice. Please try again.");
        }
    } while (choice != 0);

    scanner.close();
  }
}
```

Output

```
$ java Stack
```

Stack Menu:

1. Push

2. Pop

3. Peek

4. Display Stack Contents

5. Check if the stack is empty

6. Check if the stack is full

0. Exit

Enter your choice: 4

Stack is empty.

Stack Menu:

1. Push

2. Pop

3. Peek

4. Display Stack Contents

5. Check if the stack is empty

6. Check if the stack is full

0. Exit

Enter your choice: 5

Is the stack empty? true

Stack Menu:

1. Push

2. Pop

3. Peek

4. Display Stack Contents

5. Check if the stack is empty

6. Check if the stack is full

0. Exit

Enter your choice: 6

Is the stack full? false

Stack Menu:

1. Push

2. Pop

3. Peek

4. Display Stack Contents

5. Check if the stack is empty

6. Check if the stack is full

0. Exit

Enter your choice: 1

Enter the value to push: 10

Pushed: 10

Stack Menu:

1. Push

2. Pop

3. Peek

4. Display Stack Contents

5. Check if the stack is empty

6. Check if the stack is full

0. Exit

Enter your choice: 1

Enter the value to push: 20

Pushed: 20

Stack Menu:

1. Push

2. Pop

3. Peek

4. Display Stack Contents

5. Check if the stack is empty

6. Check if the stack is full

0. Exit

Enter your choice: 4

Stack Contents: 10 20

Stack Menu:

1. Push

2. Pop

3. Peek

4. Display Stack Contents

5. Check if the stack is empty

6. Check if the stack is full

0. Exit

Enter your choice: 3

Peeked: 20

Stack Menu:

1. Push

2. Pop

3. Peek

4. Display Stack Contents

5. Check if the stack is empty

6. Check if the stack is full

0. Exit

Enter your choice: 1

Enter the value to push: 30

Pushed: 30

Stack Menu:

1. Push

2. Pop

3. Peek

4. Display Stack Contents

5. Check if the stack is empty

6. Check if the stack is full

0. Exit

Enter your choice: 4

Stack Contents: 10 20 30

Stack Menu:

1. Push

2. Pop

3. Peek

4. Display Stack Contents

5. Check if the stack is empty

6. Check if the stack is full

0. Exit

Enter your choice: 2

Popped: 30

Stack Menu:

1. Push

2. Pop

3. Peek

4. Display Stack Contents

5. Check if the stack is empty

6. Check if the stack is full

0. Exit

Enter your choice: 3

Peeked: 20

Stack Menu:

1. Push

2. Pop

3. Peek

4. Display Stack Contents

5. Check if the stack is empty

6. Check if the stack is full

0. Exit

Enter your choice: 4

Stack Contents: 10 20

Stack Menu:

1. Push

2. Pop

3. Peek

4. Display Stack Contents

5. Check if the stack is empty

6. Check if the stack is full

0. Exit

Enter your choice: 0

Exiting the program. Goodbye!

Program 03 : Employee Class

A class called Employee, which models an employee with an ID, name and salary, is designed as shown in the following class diagram. The method raiseSalary (percent) increases the salary by the given percentage. Develop the Employee class and suitable main method for demonstration.

Java Code

```
public class Employee {
    private int id;
    private String name;
    private double salary;
```

```
public Employee(int id, String name, double salary) {
    this.id = id;
    this.name = name;
    this.salary = salary;
}

public void raiseSalary(double percent) {
    if (percent > 0) {
        double raiseAmount = salary * (percent / 100);
        salary += raiseAmount;
        System.out.println(name + "'s salary raised by " + percent + "%. New salary: $" + salary);
    } else {
        System.out.println("Invalid percentage. Salary remains unchanged.");
    }
}

public String toString() {
    return "Employee ID: " + id + ", Name: " + name + ", Salary: $" + salary;
}

public static void main(String[] args) {
    // Creating an Employee object
    Employee employee = new Employee(1, "John Doe", 50000.0);

    // Displaying employee details
    System.out.println("Initial Employee Details:");
    System.out.println(employee);

    // Raising salary by 10%
    employee.raiseSalary(10);
```

```
        // Displaying updated employee details
        System.out.println("\nEmployee Details after Salary Raise:");
        System.out.println(employee);
    }
}
```

In this example, the Employee class has a constructor to initialize the employee's ID, name, and salary. The raiseSalary method takes a percentage as a parameter and raises the salary accordingly. The toString method is overridden to provide a meaningful string representation of the Employee object. The main method demonstrates the usage of the Employee class by creating an instance, displaying its details, raising the salary, and then displaying the updated details.

Output

```
$ java Employee
Initial Employee Details:
Employee ID: 1, Name: John Doe, Salary: $50000.0
John Doe's salary raised by 10.0%. New salary: $55000.0

Employee Details after Salary Raise:
Employee ID: 1, Name: John Doe, Salary: $55000.0
```

Program 04 : 2D Point Class

A class called MyPoint, which models a 2D point with x and y coordinates, is designed as follows:

- **Two instance variables x (int) and y (int).**
- **A default (or "no-arg") constructor that construct a point at the default location of (0, 0).**
- **A overloaded constructor that constructs a point with the given x and y coordinates.**
- **A method setXY() to set both x and y.**
- **A method getXY() which returns the x and y in a 2-element int array.**
- **A toString() method that returns a string description of the instance in the format "(x, y)".**
- **A method called distance(int x, int y) that returns the distance from this point to another point at the given (x, y) coordinates**
- **An overloaded distance(MyPoint another) that returns the distance from this point to the given MyPoint instance (called another)**

- **Another overloaded distance() method that returns the distance from this point to the origin (0,0) Develop the code for the class MyPoint. Also develop a JAVA program (called TestMyPoint) to test all the methods defined in the class.**

Java Code

MyPoint.java

```
public class MyPoint {
   private int x;
   private int y;

   // Default constructor
   public MyPoint() {
      this.x = 0;
      this.y = 0;
   }

   // Overloaded constructor
   public MyPoint(int x, int y) {
      this.x = x;
      this.y = y;
   }

   // Set both x and y
   public void setXY(int x, int y) {
      this.x = x;
      this.y = y;
   }

   // Get x and y in a 2-element int array
   public int[] getXY() {
      return new int[]{x, y};
   }
```

```
    // Return a string description of the instance in the format "(x, y)"
    public String toString() {
        return "(" + x + ", " + y + ")";
    }

    // Calculate distance from this point to another point at (x, y) coordinates
    public double distance(int x, int y) {
        int xDiff = this.x - x;
        int yDiff = this.y - y;
        return Math.sqrt(xDiff * xDiff + yDiff * yDiff);
    }

    // Calculate distance from this point to another MyPoint instance (another)
    public double distance(MyPoint another) {
        return distance(another.x, another.y);
    }

    // Calculate distance from this point to the origin (0,0)
    public double distance() {
        return distance(0, 0);
    }
}
```

TestMyPoint.java

```
public class TestMyPoint {
    public static void main(String[] args) {
        // Creating MyPoint objects using different constructors
        MyPoint point1 = new MyPoint();
        MyPoint point2 = new MyPoint(3, 4);

        // Testing setXY and getXY methods
```

```
        point1.setXY(1, 2);
        System.out.println("Point1 coordinates after setXY: " + point1.getXY()[0] + ", " +
point1.getXY()[1]);

        // Testing toString method
        System.out.println("Point2 coordinates: " + point2.toString());

        // Testing distance methods
        System.out.println("Distance from Point1 to Point2: " + point1.distance(point2));
        System.out.println("Distance from Point2 to Origin: " + point2.distance());
    }
}
```

This **TestMyPoint** program creates two **MyPoint** objects, sets and retrieves coordinates, and tests the various distance calculation methods. Feel free to modify and expand this code as needed.

Output

```
$ java TestMyPoint
Point1 coordinates after setXY: 1, 2
Point2 coordinates: (3, 4)
Distance from Point1 to Point2: 2.8284271247461903
Distance from Point2 to Origin: 5.0
```

Program 05 : Inheritance & Polymorphism – Shape Class

Develop a JAVA program to create a class named shape. Create three sub classes namely: circle, triangle and square, each class has two member functions named draw () and erase (). Demonstrate polymorphism concepts by developing suitable methods, defining member data and main program.

Java Code

```
class Shape {
    protected String name;

    public Shape(String name) {
        this.name = name;
    }
```

```
    public void draw() {
        System.out.println("Drawing a " + name);
    }

    public void erase() {
        System.out.println("Erasing a " + name);
    }
}

class Circle extends Shape {
    private double radius;

    public Circle(String name, double radius) {
        super(name);
        this.radius = radius;
    }

    @Override
    public void draw() {
        System.out.println("Drawing a circle with radius " + radius);
    }

    @Override
    public void erase() {
        System.out.println("Erasing a circle with radius " + radius);
    }
}

class Triangle extends Shape {
    private double base;
```

```
    private double height;

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

    @Override
    public void draw() {
        System.out.println("Drawing a triangle with base " + base + " and height " + height);
    }

    @Override
    public void erase() {
        System.out.println("Erasing a triangle with base " + base + " and height " + height);
    }
}

class Square extends Shape {
    private double side;

    public Square(String name, double side) {
        super(name);
        this.side = side;
    }

    @Override
    public void draw() {
        System.out.println("Drawing a square with side length " + side);
    }
```

```
    @Override
    public void erase() {
        System.out.println("Erasing a square with side length " + side);
    }
}

public class ShapeDemo {
    public static void main(String[] args) {
        Shape[] shapes = new Shape[3];

        shapes[0] = new Circle("Circle", 5.0);
        shapes[1] = new Triangle("Triangle", 4.0, 6.0);
        shapes[2] = new Square("Square", 3.0);

        for (Shape shape : shapes) {
            shape.draw();
            shape.erase();
            System.out.println();
        }
    }
}
```

In this program, the **Shape** class is the superclass, and **Circle**, **Triangle**, and **Square** are its subclasses. The **draw()** and **erase()** methods are overridden in each subclass. The **main** method creates an array of **Shape** objects and initializes it with instances of the different subclasses. When iterating through the array and calling the **draw()** and **erase()** methods, polymorphism allows the appropriate overridden methods in each subclass to be executed.

Output

```
$ java ShapeDemo
Drawing a circle with radius 5.0
Erasing a circle with radius 5.0
```

Drawing a triangle with base 4.0 and height 6.0

Erasing a triangle with base 4.0 and height 6.0

Drawing a square with side length 3.0

Erasing a square with side length 3.0

Program 06 : Abstract Class

Develop a JAVA program to create an abstract class Shape with abstract methods calculateArea() and calculatePerimeter(). Create subclasses Circle and Triangle that extend the Shape class and implement the respective methods to calculate the area and perimeter of each shape.

Java Code

```
abstract class Shape {
    abstract double calculateArea();
    abstract double calculatePerimeter();
}

class Circle extends Shape {
    private double radius;

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

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

    @Override
    double calculatePerimeter() {
        return 2 * Math.PI * radius;
    }
```

```
}

class Triangle extends Shape {
    private double side1;
    private double side2;
    private double side3;

    public Triangle(double side1, double side2, double side3) {
        this.side1 = side1;
        this.side2 = side2;
        this.side3 = side3;
    }

    @Override
    double calculateArea() {
        // Using Heron's formula to calculate the area of a triangle
        double s = (side1 + side2 + side3) / 2;
        return Math.sqrt(s * (s - side1) * (s - side2) * (s - side3));
    }

    @Override
    double calculatePerimeter() {
        return side1 + side2 + side3;
    }
}

public class ShapeDemo {
    public static void main(String[] args) {
        // Creating Circle and Triangle objects
        Circle circle = new Circle(5.0);
        Triangle triangle = new Triangle(3.0, 4.0, 5.0);
```

```
        // Calculating and displaying area and perimeter
        System.out.println("Circle Area: " + circle.calculateArea());
        System.out.println("Circle Perimeter: " + circle.calculatePerimeter());

        System.out.println("\nTriangle Area: " + triangle.calculateArea());
        System.out.println("Triangle Perimeter: " + triangle.calculatePerimeter());
    }
}
```

In this program, **Shape** is an abstract class with abstract methods **calculateArea()** and **calculatePerimeter()**. The **Circle** and **Triangle** classes extend **Shape** and provide their implementations for these abstract methods. The **ShapeDemo** class demonstrates creating objects of these shapes and calculating their areas and perimeters.

Output

```
$ java ShapeDemo
Circle Area: 78.53981633974483
Circle Perimeter: 31.41592653589793

Triangle Area: 6.0
Triangle Perimeter: 12.0
```

Program 07 : Resizable interface

Develop a JAVA program to create an interface Resizable with methods resizeWidth(int width) and resizeHeight(int height) that allow an object to be resized. Create a class Rectangle that implements the Resizable interface and implements the resize methods.

Java Code

```
// Resizable interface
interface Resizable {
    void resizeWidth(int width);
    void resizeHeight(int height);
}
```

```
// Rectangle class implementing Resizable interface
class Rectangle implements Resizable {
    private int width;
    private int height;

    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    // Implementation of Resizable interface
    @Override
    public void resizeWidth(int width) {
        this.width = width;
        System.out.println("Resized width to: " + width);
    }

    @Override
    public void resizeHeight(int height) {
        this.height = height;
        System.out.println("Resized height to: " + height);
    }

    // Additional methods for Rectangle class
    public int getWidth() {
        return width;
    }

    public int getHeight() {
        return height;
    }
```

```
    public void displayInfo() {
        System.out.println("Rectangle: Width = " + width + ", Height = " + height);
    }
}

// Main class to test the implementation
public class ResizeDemo {
    public static void main(String[] args) {
        // Creating a Rectangle object
        Rectangle rectangle = new Rectangle(10, 5);

        // Displaying the original information
        System.out.println("Original Rectangle Info:");
        rectangle.displayInfo();

        // Resizing the rectangle
        rectangle.resizeWidth(15);
        rectangle.resizeHeight(8);

        // Displaying the updated information
        System.out.println("\nUpdated Rectangle Info:");
        rectangle.displayInfo();
    }
}
```

In this program, the **Resizable** interface declares the methods **resizeWidth** and **resizeHeight**. The **Rectangle** class implements this interface and provides the specific implementation for resizing the width and height. The **main** method in the **ResizeDemo** class creates a **Rectangle** object, displays its original information, resizes it, and then displays the updated information.

Output

```
$ java ResizeDemo
Original Rectangle Info:
```

Rectangle: Width = 10, Height = 5

Resized width to: 15

Resized height to: 8

Updated Rectangle Info:

Rectangle: Width = 15, Height = 8

Program 08 : Outer class

Develop a JAVA program to create an outer class with a function display. Create another class inside the outer class named inner with a function called display and call the two functions in the main class.

Java Code

```
class Outer {
    void display() {
        System.out.println("Outer class display method");
    }

    class Inner {
        void display() {
            System.out.println("Inner class display method");
        }
    }
}

public class OuterInnerDemo {
    public static void main(String[] args) {
        // Create an instance of the Outer class
        Outer outer = new Outer();

        // Call the display method of the Outer class
        outer.display();
```

```
        // Create an instance of the Inner class (nested inside Outer)
        Outer.Inner inner = outer.new Inner();

        // Call the display method of the Inner class
        inner.display();
    }
}
```

In this program, the **Outer** class has a method named **display**, and it also contains an inner class named **Inner** with its own **display** method. In the **main** method of the **OuterInnerDemo** class, an instance of the outer class (**Outer**) is created, and its **display** method is called. Then, an instance of the inner class (**Inner**) is created using the outer class instance, and its **display** method is called. This demonstrates the concept of nesting classes in Java.

Output

```
$ java OuterInnerDemo
Outer class display method
Inner class display method
```

Program 09 : Custom Exception

Develop a JAVA program to raise a custom exception (user defined exception) for DivisionByZero using try, catch, throw and finally.

Java Code

```
// Custom exception class
class DivisionByZeroException extends Exception {
    public DivisionByZeroException(String message) {
        super(message);
    }
}

public class CustomExceptionDemo {
    // Method to perform division and throw custom exception if denominator is zero
    static double divide(int numerator, int denominator) throws DivisionByZeroException {
```

```
        if (denominator == 0) {
            throw new DivisionByZeroException("Cannot divide by zero!");
        }
        return (double) numerator / denominator;
    }
    public static void main(String[] args) {
        int numerator = 10;
        int denominator = 0;

        try {
            double result = divide(numerator, denominator);
            System.out.println("Result of division: " + result);
        } catch (DivisionByZeroException e) {
            System.out.println("Exception caught: " + e.getMessage());
        } finally {
            System.out.println("Finally block executed");
        }
    }
}
```

n this program:

- The **DivisionByZeroException** class is a custom exception that extends the **Exception** class.
- The **divide** method performs division and throws the custom exception if the denominator is zero.
- In the **main** method, we attempt to divide and catch the custom exception if it occurs. The **finally** block is used for code that must be executed, whether an exception is thrown or not.

When you run this program with a denominator of 0, it will throw the **DivisionByZeroException**, catch it, print the error message, and then execute the **finally** block.

Output

```
$ java CustomExceptionDemo
Exception caught: Cannot divide by zero!
Finally block executed
```

Program 10 : Packages

Develop a JAVA program to create a package named mypack and import & implement it in a suitable class.

Java Code

Package mypack

```
// Inside a folder named 'mypack'
package mypack;


public class MyPackageClass {
    public void displayMessage() {
        System.out.println("Hello from MyPackageClass in mypack package!");
    }


    // New utility method
    public static int addNumbers(int a, int b) {
        return a + b;
    }
}
```

Now, let's create the main program in a different file outside the mypack folder:

PackageDemo class using mypack Package

```
// Main program outside the mypack folder
import mypack.MyPackageClass;
//import mypack.*;



public class PackageDemo {
    public static void main(String[] args) {
        // Creating an instance of MyPackageClass from the mypack package
        MyPackageClass myPackageObject = new MyPackageClass();


        // Calling the displayMessage method from MyPackageClass
```

```
        myPackageObject.displayMessage();

        // Using the utility method addNumbers from MyPackageClass
        int result = MyPackageClass.addNumbers(5, 3);
        System.out.println("Result of adding numbers: " + result);
    }
}
```

To compile and run this program, you need to follow these steps:

Organize your directory structure as follows:

```
project-directory/
├── mypack/
│   └── MyPackageClass.java
└── PackageDemo.java
```

Compile the files:

```
javac mypack/MyPackageClass.java
javac PackageDemo.java
```

Output

```
$ java PackageDemo
Hello from MyPackageClass in mypack package!
Result of adding numbers: 8
```

Program 11 : Runnable Interface

Write a program to illustrate creation of threads using runnable class. (start method start each of the newly created thread. Inside the run method there is sleep() for suspend the thread for 500 milliseconds).

Java Code

```
class MyRunnable implements Runnable {
    private volatile boolean running = true;

    @Override
    @SuppressWarnings("deprecation")
    public void run() {
```

```
        while (running) {
            try {
                // Suppress deprecation warning for Thread.sleep()
                Thread.sleep(500);
                System.out.println("Thread ID: " + Thread.currentThread().getId() + " is running.");
            } catch (InterruptedException e) {
                System.out.println("Thread interrupted.");
            }
        }
    }

    public void stopThread() {
        running = false;
    }
}


public class RunnableThreadExample {
    public static void main(String[] args) {
        // Create five instances of MyRunnable
        MyRunnable myRunnable1 = new MyRunnable();
        MyRunnable myRunnable2 = new MyRunnable();
        MyRunnable myRunnable3 = new MyRunnable();
        MyRunnable myRunnable4 = new MyRunnable();
        MyRunnable myRunnable5 = new MyRunnable();

        // Create five threads and associate them with MyRunnable instances
        Thread thread1 = new Thread(myRunnable1);
        Thread thread2 = new Thread(myRunnable2);
        Thread thread3 = new Thread(myRunnable3);
        Thread thread4 = new Thread(myRunnable4);
```

```
        Thread thread5 = new Thread(myRunnable5);

        // Start the threads
        thread1.start();
        thread2.start();
        thread3.start();
        thread4.start();
        thread5.start();

        // Sleep for a while to allow the threads to run
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Stop the threads gracefully
        myRunnable1.stopThread();
        myRunnable2.stopThread();
        myRunnable3.stopThread();
        myRunnable4.stopThread();
        myRunnable5.stopThread();
    }
}
```

In this program, we define a MyRunnable class that implements the Runnable interface. The run method contains a loop where the thread sleeps for 500 milliseconds, printing its ID during each iteration. We also handle potential interruptions caused by thread operations.

In the RunnableThreadExample class, we create five instances of MyRunnable, each associated with a separate thread. The start method is called on each thread, initiating their concurrent execution. After a brief period of allowing the threads to run, we gracefully stop each thread using the stopThread method.

Output

```
$ java RunnableThreadExample
```

Thread ID: 24 is running.

Thread ID: 21 is running.

Thread ID: 20 is running.

Thread ID: 23 is running.

Thread ID: 22 is running.

Program 12 : Thread Class

Develop a program to create a class MyThread in this class a constructor, call the base class constructor, using super and start the thread. The run method of the class starts after this. It can be observed that both main thread and created child thread are executed concurrently.

Java Code

```
class MyThread extends Thread {
    // Constructor calling base class constructor using super
    public MyThread(String name) {
        super(name);
        start(); // Start the thread in the constructor
    }

    // The run method that will be executed when the thread starts
    @Override
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(Thread.currentThread().getName() + " Count: " + i);
            try {
                Thread.sleep(500); // Sleep for 500 milliseconds
            } catch (InterruptedException e) {
                System.out.println(Thread.currentThread().getName() + " Thread interrupted.");
            }
        }
    }
}
```

```
public class ThreadConcurrentExample {
  public static void main(String[] args) {
    // Create an instance of MyThread
    MyThread myThread = new MyThread("Child Thread");

    // Main thread
    for (int i = 1; i <= 5; i++) {
      System.out.println(Thread.currentThread().getName() + " Thread Count: " + i);
      try {
        Thread.sleep(500); // Sleep for 500 milliseconds
      } catch (InterruptedException e) {
        System.out.println(Thread.currentThread().getName() + " Thread interrupted.");
      }
    }
  }
}
```

n this program:

- The MyThread class extends Thread.
- The constructor of MyThread calls the base class constructor using super(name) to set the thread's name and starts the thread.
- The run method is overridden and contains a loop to print counts. The thread sleeps for 500 milliseconds in each iteration.
- In the main method, an instance of MyThread is created, which starts the child thread concurrently.
- The main thread also prints counts and sleeps for 500 milliseconds in each iteration.

When you run this program, you'll observe that both the main thread and the child thread are executed concurrently, and their outputs may be interleaved.

Output

```
$ java ThreadConcurrentExample
main Thread Count: 1
Child Thread Count: 1
```

main Thread Count: 2

Child Thread Count: 2

main Thread Count: 3

Child Thread Count: 3

main Thread Count: 4

Child Thread Count: 4

main Thread Count: 5

Child Thread Count: 5