**Qno 1.** Write a Java program to calculate the area of a rectangle. Create a class called "Rectangle" with attributes for length and width, and implement a method to calculate the area. Demonstrate the usage of data types, variables, and object-oriented programming concepts.

Main.java :

class Main {

    public static void main(String[] args) {

        Rectangle rect = new Rectangle(15, 20);

        System.out.printf("\nArea of rectangle is: %d", rect.area());

    }

}

Rectangle.java :  
  
class Rectangle {

    int length;

    int breadth;

    public Rectangle(int l, int b) {

        this.length = l;

        this.breadth = b;

    }

    public int area() {

        return length \* breadth;

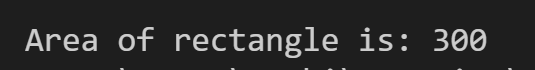
    }

}

In this program, we define a Rectangle class with two attributes: length and breadth. The class has a constructor that takes the length and breadth as parameters and initializes the attributes. We also have a method named area() that calculates the area of the rectangle by multiplying the length and breadth.

In the Main class, we create an instance of the Rectangle class called rect with a length of 10 and breadth of 20. We then call the area() method on the rect object and print the result using System.out.printf(). The output will be the area of the rectangle, which is 200 in this case.

**Output:**



**Qno 2.** Create a package called "com.example.shapes" and place the Rectangle class from question 1 inside it. Demonstrate the concept of packages and explain their significance in organizing and managing Java code.

Main.java :

import com.example.shapes.Rectangle;

public class Main {

    public static void main(String[] args) {

        Rectangle rect = new Rectangle(10, 20);

        System.out.printf("\nArea of rectangle is: %d", rect.area());

    }

}

Rectangle.java :

package com.example.shapes;

public class Rectangle {

    int length;

    int breadth;

    public Rectangle(int l, int b) {

        this.length = l;

        this.breadth = b;

    }

    public int area() {

        return length \* breadth;

    }

}

**Output:**

**Qno 3**. Implement an abstract class called "Shape" with an abstract method for calculating the area. Modify the Rectangle class to inherit from the Shape class and provide the necessary implementation. Discuss the concept of inheritance and how it promotes code reusability.

Main.java :

public class Main {

    public static void main(String[] args) {

        Rectangle rectangle = new Rectangle(5, 5);

        double area = rectangle.calculateArea();

        System.out.println("Area of rectangle is: " + area);

    }

}

Rectangle.java :

class Rectangle extends Shape {

    private int length;

    private int breadth;

    public Rectangle(int length, int breadth) {

        this.length = length;

        this.breadth = breadth;

    }

    @Override

    public double calculateArea() {

        return length \* breadth;

    }

}

Shape.java :

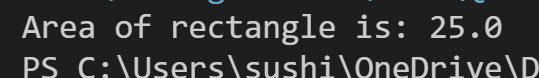
abstract class Shape {

    public abstract double calculateArea();

}

Inheritance is a fundamental concept in object-oriented programming (OOP) where a class can inherit properties and behavior from another class. In Java, classes can inherit from other classes using the extends keyword. The class being inherited from is called the superclass or parent class, and the class inheriting from it is called the subclass or child class.

Inheritance promotes code reusability by allowing subclasses to inherit and reuse the fields, methods, and behavior defined in the superclass. This reduces code duplication and promotes a hierarchical structure where common characteristics and behaviors are defined in the superclass, while subclasses can specialize or extend that functionality.

**Output:**

**Qno 4**. Write a Java program that handles exceptions. Prompt the user to enter a number and handle the NumberFormatException that may occur if a non-numeric value is entered. Explain the importance of exception handling.

Main.java :

import java.util.Scanner;

public class ExceptionHandlingExample {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        try {

            System.out.print("Enter a number: ");

            int number = Integer.parseInt(scanner.nextLine());

            System.out.println("You entered: " + number);

        } catch (NumberFormatException e) {

            System.out.println("Invalid input. Please enter a valid number.");

        } finally {

            scanner.close(); // Close the scanner to avoid resource leak

        }

    }

}

Exception handling is important in Java programming for several reasons:

*Error handling:* Exceptions allow you to gracefully handle errors and exceptional situations that may occur during program execution. Instead of crashing the program or displaying cryptic error messages, you can provide meaningful error handling and recovery mechanisms.

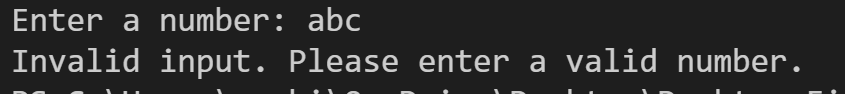
*Program stability*: Exception handling helps ensure the stability of your program. By catching and handling exceptions, you prevent the program from terminating abruptly and provide a controlled flow of execution. This allows you to handle errors gracefully and continue the program's execution if possible.

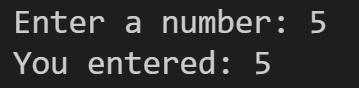
*Debugging*: Exceptions provide valuable information when debugging your program. When an exception occurs, the stack trace provides information about the error's location, allowing you to identify and fix the issue more effectively.

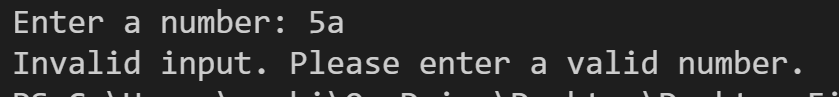
*Robustness*: Exception handling improves the robustness of your code. By handling exceptions, you can anticipate and handle exceptional situations, making your code more resilient to unexpected errors or user input.

*Maintainability*: Exception handling promotes code maintainability. By handling exceptions at appropriate points in your code, you separate the error handling logic from the normal flow of your program. This makes your code easier to understand, modify, and maintain.

**Output:**

When Input is “5” : When Input is “abc” :



When Input it “5a” :   


**Qno 5**: Write a Java program that implements an interface called "Drawable". The interface should have a method called draw() which displays a message indicating the object being drawn. Create a class called "Circle" that implements the "Drawable" interface and overrides the draw() method to display "Drawing a circle". Demonstrate the usage of interfaces and how they provide a way to achieve multiple inheritances in Java.

Main.java :

// Define the Drawable interface

interface Drawable {

    void draw();

}

// Implement the Drawable interface in the Circle class

class Circle implements Drawable {

    @Override

    public void draw() {

        System.out.println("Drawing a circle");

    }

}

// Main class to demonstrate the usage of interfaces

public class Main {

    public static void main(String[] args) {

        // Create an instance of the Circle class

        Circle circle = new Circle();

        // Call the draw() method on the Circle object

        circle.draw();

    }

}

In this program, we define the Drawable interface with a single method draw(). The Circle class implements the Drawable interface and overrides the draw() method to display the message "Drawing a circle".

The Main class demonstrates the usage of interfaces. We create an instance of the Circle class and call the draw() method on it. Since Circle implements the Drawable interface, it is able to provide an implementation for the draw() method. When we call circle.draw(), it displays the message "Drawing a circle".

Interfaces in Java provide a way to achieve multiple inheritances, as a class can implement multiple interfaces. This allows a class to inherit and provide implementation for multiple sets of behavior defined in different interfaces. It promotes code reusability and allows for flexible design by separating the definition of behavior from its implementation.

In the given example, the Circle class can implement other interfaces in addition to the Drawable interface if needed, enabling it to inherit and implement different sets of behaviors from multiple interfaces.

**Output:**



**Qno 6.** Write a Java program that uses reflection to dynamically load and instantiate a class. Create a class called "DynamicClass" with a method called displayMessage() that prints "Hello, Reflection!". In the main program, use reflection to load the "DynamicClass" and invoke the displayMessage() method. Explain the concept of reflection and how it allows Java programs to examine or modify their own structure at runtime.

import java.lang.reflect.Method;

class DynamicClass {

    public void displayMessage() {

        System.out.println("Hello, Reflection!");

    }

}

public class Main {

    public static void main(String[] args) {

        try {

            // Load the DynamicClass dynamically

            Class<?> dynamicClass = Class.forName("DynamicClass");

            // Create an instance of the DynamicClass

            Object dynamicObject = dynamicClass.getDeclaredConstructor().newInstance();

            // Get the displayMessage method

            Method displayMessageMethod = dynamicClass.getDeclaredMethod("displayMessage");

            // Invoke the displayMessage method on the dynamicObject

            displayMessageMethod.invoke(dynamicObject);

        } catch (Exception e) {

            e.printStackTrace();

        }

    }

}

Reflection in Java allows programs to examine and modify their own structure at runtime. It provides the ability to inspect and manipulate classes, interfaces, fields, methods, and other components of a Java program dynamically. Reflection enables tasks such as:

Obtaining information about classes and their members (methods, fields, constructors, etc.).

Creating new instances of classes dynamically.

Accessing and modifying the values of fields and methods at runtime.

Invoking methods dynamically.

Working with annotations and annotations processing.

**Output:**

**Qno 7:** Write a Java program that demonstrates the usage of an inner class. Create an outer class called "Outer" with a private instance variable and a method called display() that prints the value of the instance variable. Inside the "Outer" class, create an inner class called "Inner" with a method called changeValue(int newValue) that modifies the value of the instance variable. In the main program, instantiate the "Outer" class, invoke the display() method, and then invoke the changeValue() method from the inner class to update the value. Discuss the concept of inner classes and their relationship with the outer class.

class Outer {

    private int value;

    public Outer(int value) {

        this.value = value;

    }

    public void display() {

        System.out.println("Value in outer class: " + value);

    }

    class Inner {

        public void changeValue(int newValue) {

            value = newValue;

        }

    }

}

public class Main {

    public static void main(String[] args) {

        Outer outer = new Outer(10);

        outer.display();

        Outer.Inner inner = outer.new Inner();

        inner.changeValue(20);

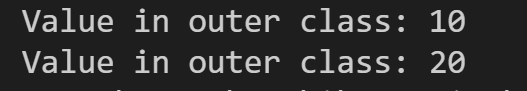
        outer.display();

    }

}

Inner classes in Java are defined within the scope of another class. They have access to the members of the outer class, including private members. Inner classes have a close relationship with the outer class, and they can access and modify the variables and methods of the outer class. Inner classes provide encapsulation and allow you to logically group related classes together. They can be useful when the inner class needs to access and modify the state of the outer class or when you want to define a class that is closely tied to the outer class and doesn't make sense to be used independently.

**Output:**

****