Java module 3

1) What is the importance of super keyword in inheritance? Give example.

The ***super keyword*** in Java is used in **inheritance** to refer to the parent class. It plays an important role in the following ways:

Uses of super Keyword:

1. *Access Parent Class Constructor:*
   - It is used to call the constructor of the parent class.
2. *Access Parent Class Methods*:
   - It is used to call a method of the parent class that has been overridden.
3. *Access Parent Class Variables:*
   - It is used to refer to a variable of the parent class when there is a name conflict.

Syntax:

```
super(); // Calls parent class constructor
super.methodName(); // Calls parent class method
super.variableName; // Refers to parent class variable
```

Example:

```
class Parent {
   int age = 50;
}

class Child extends Parent{
   int age = 20;

   void displayAge()
 {
      System.out.println("Parent age: " + super.age);
      System.out.println("Child age: " + age);
   }
}

public class Main {
   public static void main(String[] args)
 {
      Child c = new Child();
      c.displayAge();
```

```
    }
}
```

Output

```
Parent age: 50
Child age: 20
```

2) *Explain how interface is used to achieve multiple inheritance in java.*

Java does not support ***multiple inheritance*** with classes to avoid ambiguity. However, multiple inheritance can be achieved using interfaces.

Key Features:

1.An Interface is a Blueprint :-

- It contains abstract methods and constants.

2.Multiple Inheritance :-

- A class can implement multiple interfaces.

3.No Ambiguity :-

- There is no diamond problem because interfaces do not store implementation.

Example :

```
interface Animal {
   void eat();
}

interface Bird {
   void fly();
}

class Bat implements Animal, Bird {
   public void eat() {
      System.out.println("Bat eats insects.");
   }

   public void fly() {
      System.out.println("Bat can fly.");
   }
}

public class Main {
   public static void main(String[] args) {
      Bat bat = new Bat();
      bat.eat();
      bat.fly();
   }
}
```

Output :

```
Bat eats insects.
Bat can fly.
```

3) What is abstract class and abstract method? Explain with example.

An **abstract class** is a class that is declared using the *abstract keyword* and cannot be instantiated. It is used to provide a base for subclasses to extend and implement abstract methods.

An **abstract method** is a method declared without implementation.

Key Points:

1. *An abstract class may have both abstract and concrete methods.*
2. *Abstract methods must be implemented by subclasses.*
3. *If a class has at least one abstract method, it must be declared abstract*.

Syntax:

```
abstract class AbstractClass {
   abstract void abstractMethod(); // Abstract method
   void concreteMethod() { // Concrete method
      System.out.println("This is a concrete method.");
   }
}
```

Example:

```
abstract class Shape {
   abstract void draw(); // Abstract method

   void display() {
      System.out.println("This is a shape.");
   }
}

class Circle extends Shape {
   void draw() {
      System.out.println("Drawing a circle.");
   }
}

public class Main {
   public static void main(String[] args) {
      Circle circle = new Circle();
      circle.display();
      circle.draw();
   }
}
```

Output:

```
This is a shape.
Drawing a circle.
```

4) Define an Exception. Key Terms in Exception Handling

An ***exception*** is an event that disrupts the normal flow of a program. It occurs during runtime and can be caused by various factors like invalid input, division by zero, or file not found.

Key Terms in Exception Handling:

1. ***try Block:***
   - Contains the code that may throw an exception.
2. ***catch Block:***
   - Handles the exception.
3. ***finally Block:***
   - Executes code after try and catch, regardless of exception occurrence.
4. ***throw:***
   - Used to explicitly throw an exception.
5. ***throws:***
   - Declares exceptions in a method signature.

Example :

```
try {
   // Code that may throw an exception
} catch (ExceptionType e) {
   // Code to handle the exception
} finally {
   // Cleanup code
}
```

Example :

```
public class Main {
   public static void main(String[] args) {
      try {
         int divideByZero = 10 / 0; // Throws ArithmeticException
      } catch (ArithmeticException e) {
         System.out.println("Exception caught: " + e.getMessage());
      } finally {
         System.out.println("This block is always executed.");
      }
   }
}
```

Output :

```
Exception caught: / by zero
This block is always executed.
```

5. Explain the method overriding with a suitable example.

Method Overriding is a feature in **Java where a subclass provides a specific implementation of a method that is already defined in its superclass.** This is done to modify or extend the functionality of the method in the subclass.

- The method in the subclass must have the same name, return type, and parameters as the method in the superclass.
- Method Overriding is used to achieve runtime polymorphism, where the call to an overridden method is resolved at runtime.

Key Points:

- Method overriding happens in the context of inheritance.
- The **@Override** annotation is often used to indicate that a method is being overridden.
- The overridden method in the subclass must have the same signature as the one in the superclass.

Syntax:

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

class Subclass extends Superclass {
    @Override
    void display() {
        System.out.println("Subclass display()");
    }
}
```

Example

```
class Animal {
    // Method in superclass
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    // Overriding the sound() method in subclass
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Animal();
        Animal myDog = new Dog();

        myAnimal.sound();  // Outputs: Animal makes a sound
        myDog.sound();     // Outputs: Dog barks
    }
}
```

6. Inheritance in Java

Inheritance is an object-oriented programming concept **where one class (called the subclass or child class) inherits the properties and behaviors (methods) from another class** (called the superclass or parent class). It allows the subclass to reuse the code in the superclass and extend or modify it.

Key Concepts of Inheritance:

1. **Superclass (Parent class):** The class that is inherited from. It provides common properties and methods to be shared by subclasses.

2. **Subclass (Child class):** The class that inherits from the superclass. It can add additional features or modify the behavior of inherited methods.

3. **Access to Superclass Members:** The subclass inherits both fields and methods from the superclass, but private members of the superclass are not directly accessible from the subclass.

Types of Inheritance in Java:

1. **Single Inheritance:** A class inherits from a single class.
   - Example: Class B inherits from Class A.
2. **Multilevel Inheritance:** A class inherits from a subclass, forming a chain.
   - Example: Class C inherits from Class B, and Class B inherits from Class A.

7. Java Program for Multilevel Inheritance (3 Levels)

In multilevel inheritance, a class is derived from another class, which is itself derived from another class, forming a chain of inheritance.

Here's an example with 3 levels of inheritance:

```
// Level 1: Base class
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

// Level 2: Derived class from Animal
class Dog extends Animal {
    void bark() {
        System.out.println("Dog barks");
    }
}

// Level 3: Derived class from Dog
class Puppy extends Dog {
    void play() {
        System.out.println("Puppy plays with a ball");
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating an object of Puppy, which inherits from Dog and Animal
        Puppy myPuppy = new Puppy();

        // Calling methods from all levels
        myPuppy.sound();   // Inherited from Animal
        myPuppy.bark();    // Inherited from Dog
        myPuppy.play();    // Defined in Puppy
    }
}
```

output

```
Animal makes a sound
Dog barks
Puppy plays with a ball
```

Explanation:

- Animal class is at Level 1 and has a sound() method.
- Dog class is at Level 2 and inherits from Animal, adding its own bark() method.
- Puppy class is at Level 3 and inherits from Dog, adding a play() method.
- In the main method, an object of Puppy is created, which can access methods from all three levels of the inheritance hierarchy.

8. What is single-level inheritance? Write a Java program to implement single-level inheritance.

Definition:
Single-level inheritance is a type of inheritance where a class (child or subclass) is derived from another class (parent or superclass). The subclass inherits the properties and behaviors (methods) of the superclass.

Java Program to Implement Single-Level Inheritance

```java
// Superclass (Parent class)
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

// Subclass (Child class) that inherits from Animal
class Dog extends Animal {
    void bark() {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating an object of Dog
        Dog myDog = new Dog();

        // Calling methods from both Animal (superclass) and Dog (subclass)
        myDog.sound();  // Inherited from Animal
        myDog.bark();   // Defined in Dog
    }
}
```

Output

```
Animal makes a sound
Dog barks
```

Explanation:

- Animal class is the superclass, which has a method sound().
- Dog class is the subclass that extends Animal and adds a new method bark().
- In the main method, an object of Dog is created, which can call methods from both Dog (its own class) and Animal (its superclass).

9. Explain method overloading and method overriding with suitable examples.

Both method overloading and method overriding are concepts in Java that **allow us to define methods with the same name**. However, they differ in terms of their behavior and how they are used.

1. Method Overloading

Definition:
Method overloading occurs when a class has multiple methods with the same name but different parameters (either in number or type). The return type can be the same or different. Overloading is resolved at compile time (static polymorphism).

Key Points:

- Methods must have the same name.
- The number or type of parameters must differ.
- It is resolved at compile time.

Example of Method Overloading:

```
class Calculator {
    // Method with one parameter
    int add(int a) {
        return a + a;
    }

    // Method with two parameters
    int add(int a, int b) {
        return a + b;
    }

    // Method with three parameters
    int add(int a, int b, int c) {
        return a + b + c;
    }
}

public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();

        // Calling overloaded methods
        System.out.println(calc.add(5));        // Outputs: 10
        System.out.println(calc.add(5, 10));    // Outputs: 15
        System.out.println(calc.add(5, 10, 15)); // Outputs: 30
    }
}
```

Explanation:

- The add() method is overloaded with different numbers of parameters.
- Java resolves which version of add() to call based on the number and types of arguments passed.

2. Method Overriding

Definition:
Method overriding occurs **when a subclass provides a specific implementation of a method that is already defined in its superclass.** The method in the subclass must have the same name, return type, and parameters as the method in the superclass. Overriding allows us to change or extend the behavior of the inherited method.

Key Points:

- The method must have the same name, return type, and parameters.
- Overriding is resolved at runtime (dynamic polymorphism).
- It is used for providing specific implementations in subclasses.

Example of Method Overriding:

```
// Superclass
class Animal {
    // Method in superclass
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

// Subclass
class Dog extends Animal {
    // Overriding the sound() method
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}

// Another subclass
class Cat extends Animal {
    // Overriding the sound() method
    @Override
    void sound() {
        System.out.println("Cat meows");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Animal();
        Animal dog = new Dog();
        Animal cat = new Cat();

        // Calling the sound() method for each object
        animal.sound();  // Outputs: Animal makes a sound
        dog.sound();     // Outputs: Dog barks
        cat.sound();     // Outputs: Cat meows
    }
}
```

Explanation:

- The sound() method is defined in the Animal superclass.
- The Dog and Cat subclasses override the sound() method to provide their own specific implementations.
- When the sound() method is called on objects of Animal, Dog, and Cat, Java dynamically calls the overridden version of the method, based on the object's type at runtime.

Summary:

- Method Overloading: Occurs when you have multiple methods in the same class with the same name but different parameters.
- Method Overriding: Occurs when a subclass provides its own implementation of a method already defined in its superclass.

Experiment 3: class shape with circle, rectangle and square

```
// Superclass Shape
public class Shape {
    public void draw() {
        System.out.println("Drawing a shape");
    }

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

// Circle class inherits from Shape and overrides draw and erase methods
class Circle extends Shape {
    public void draw() {
        System.out.println("Drawing a circle");
    }

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

// Triangle class inherits from Shape and overrides draw and erase methods
class Triangle extends Shape {
    public void draw() {
        System.out.println("Drawing a triangle");
    }

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

// Square class inherits from Shape and overrides draw and erase methods
class Square extends Shape {

    public void draw() {
        System.out.println("Drawing a square");
    }

    public void erase() {
        System.out.println("Erasing a square");
    }
```

```java
}

// Main class to test the behavior
class Main {
    public static void main(String[] args) {
        // Creating objects of Circle, Triangle, and Square
        Circle c = new Circle();
        Triangle t = new Triangle();
        Square s = new Square();

        // Using Circle object
        System.out.println("Using circle object:");
        c.draw();
        c.erase();

        // Using Triangle object
        System.out.println("Using triangle object:");
        t.draw();
        t.erase();

        // Using Square object
        System.out.println("Using square object:");
        s.draw();
        s.erase();
    }
}
```

Op:-

```
Using circle object:
Drawing a circle
Erasing a circle
Using triangle object:
Drawing a triangle
Erasing a triangle
Using square object:
Drawing a square
Erasing a square
```

10. Explain the concept of nesting using interfaces.

Nesting of interfaces refers to the concept of defining an interface inside another interface. In Java, you can define one interface within another. This allows you to logically group related interfaces within a specific context. Nested interfaces can be either static or non-static.

Types of Nested Interfaces:

1. **Static Nested Interface:**
   - A nested interface defined inside a class or another interface can be declared static.
   - The static nested interface can be accessed independently of the outer class or interface.
   - It can be implemented by any class outside the outer interface or class.

2. **Non-Static Nested Interface:**
   - A non-static nested interface is implicitly associated with an instance of the outer class.
   - It is accessible only from an instance of the outer class or interface.

Syntax of Nesting an Interface:

```
interface OuterInterface {
    // Static Nested Interface
    interface InnerInterface {
        void display();
    }
}

class ImplementingClass implements OuterInterface.InnerInterface {
    public void display() {
        System.out.println("Implementation of Inner Interface");
    }
}

public class Main {
    public static void main(String[] args) {
        ImplementingClass obj = new ImplementingClass();
        obj.display();
    }
}
```

2. Non-Static Nested Interface

```
class OuterClass {
    // Non-static Nested Interface
    interface InnerInterface {
        void show();
    }

    class InnerClass implements InnerInterface {
        public void show() {
            System.out.println("Inside Inner Class, implementing InnerInterface");
        }
    }
}

public class Main {
    public static void main(String[] args) {
        OuterClass outer = new OuterClass();
        OuterClass.InnerClass inner = outer.new InnerClass();
        inner.show();
    }
}
```

1. **Static Nested Interface:**

- In the first example, OuterInterface contains the static InnerInterface.
- The InnerInterface is implemented in the class ImplementingClass.
- The static nested interface can be accessed without creating an instance of the outer class.

2. **Non-Static Nested Interface:**

- In the second example, the InnerInterface is non-static, which means it is defined inside the non-static inner class OuterClass.
- You cannot directly access the non-static nested interface without creating an instance of the outer class first.

Advantages of Nesting Interfaces:

- Logical Grouping: It allows related interfaces to be grouped logically, making code more organized.
- Encapsulation: Inner interfaces can be used to encapsulate behavior that's only relevant to the outer class or interface, improving modularity.
- Access Control: You can control the visibility of nested interfaces, allowing them to be accessible only within the outer class or interface.

11. Difference between class and interface

1. **Definition:**
   - A class is a blueprint for creating objects that define properties and methods.
   - An interface is a contract that defines a set of abstract methods that a class must implement.

2. **Methods:**
   - A class can have both concrete (implemented) methods and abstract methods.
   - An interface can only have abstract methods (until Java 8, which introduced default and static methods).

3. **Instantiation:**
   - A class can be instantiated to create objects.
   - An interface cannot be instantiated directly. It needs to be implemented by a class.

4. Implementation:
   - A class is used to create objects and define the behavior and state of those objects.
   - An interface is used to define a contract that classes can implement. A class can implement multiple interfaces.

5. **Constructors:**
   - A class can have constructors to initialize objects.
   - An interface cannot have constructors because it cannot be instantiated.

6. **Access Modifiers:**
   - A class can have various access modifiers like private, protected, public, and default.
   - An interface's methods are implicitly public, and you cannot define access modifiers for methods in an interface.

7. **Multiple Inheritance:**
   - A class supports single inheritance, meaning a class can only inherit from one superclass.
   - An interface supports multiple inheritance, meaning a class can implement multiple interfaces.

8. **Variables:**
   - A class can have instance variables that hold state.
   - An interface can only have constants (static final variables).

9. **Use Cases:**
   - A class is used to define objects and their behaviors.
   - An interface is used to define common behavior that can be shared across multiple classes without enforcing a class hierarchy.

10. **Abstract Nature:**
    - A class can be either abstract (not providing full implementation) or concrete (providing complete implementation).
    - An interface is inherently abstract, meaning it cannot provide method implementations (except default or static methods from Java 8 onward).

Module 4

1) *Explain the concept of importing packages in java and provide an example demonstrating the usage of the import statement.*

In Java, **packages** are used to group related classes and interfaces. The import statement allows you to use classes from other packages without needing to specify their full path.

Syntax:

```
import package_name.ClassName; // Import a specific class
import package_name.*; // Import all classes from a package
```

Example :-

```
import java.util.Scanner; // Importing Scanner class

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in); // Using the imported class
        System.out.println("Enter your name:");
        String name = sc.nextLine();
        System.out.println("Hello, " + name);
    }
}
```

2) How do you create your own exception class? Explain with a program.

You can create a custom exception by extending the **Exception class.**

Syntax

```
class MyException extends Exception {
    MyException(String message) {
        super(message);
    }
}
```

Example :

```
class InvalidAgeException extends Exception {
    InvalidAgeException(String message) {
        super(message);
    }
}

public class Main {
    public static void main(String[] args) {
        try {
            int age = -1;
            if (age < 0) {
                throw new InvalidAgeException("Age cannot be negative.");
            }
        } catch (InvalidAgeException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Output :

```
Age cannot be negative.
```

3) Nested Try Block Example

A **nested try block** means having one try block inside another. It allows handling multiple exceptions at different levels.

Example:

```
public class Main {
    public static void main(String[] args) {
        try {
            try {
                int result = 10 / 0; // Throws ArithmeticException
            } catch (ArithmeticException e) {
                System.out.println("Inner catch: Division by zero.");
            }

            String str = null;
            System.out.println(str.length()); // Throws NullPointerException
        } catch (NullPointerException e) {
            System.out.println("Outer catch: Null pointer exception.");
        }
    }
}
```

Output

```
Inner catch: Division by zero.
Outer catch: Null pointer exception.
```

4) Use of Interface in Java

An **interface** defines a contract for what a class should do, without specifying how. It allows for multiple inheritance in Java.

Syntax:

```
interface InterfaceName {
    void method(); // Abstract method
}
```

Example :

```
interface Animal {
    void sound();
}

class Dog implements Animal {
    public void sound() {
        System.out.println("Woof");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.sound();
    }
```

5. Define package. Explain the steps involved in creating a user-defined package with an example.

A package in Java is a **way to group related classes, interfaces, and sub-packages. It helps in organizing code into namespaces, avoiding naming conflicts, and providing access control and modularity.** Packages also make it easier to maintain and manage large projects.

Java provides two types of packages:

1. **Predefined packages:** These are built-in Java packages like java.util, java.io, etc.

2. **User-defined packages:** These are packages created by the programmer to organize custom classes and interfaces.

Steps to Create a User-Defined Package

1. **Define the package:**
   - At the top of your Java file, use the package keyword.
   - Example:

```
package mypackage;
```

2. **Create a class inside the package:**
   - Write a simple class inside the package.

```
package mypackage;

public class HelloWorld {
    public void showMessage() {
        System.out.println("Hello from the package!");
    }
}
```

3. **Save the file:**
   - Save the file in a folder matching the package name.
   - If your package is mypackage, save the file as HelloWorld.java inside a folder named mypackage.

4. **Compile the class:**
   - Open the terminal and navigate to the parent folder of mypackage.
   - Run this command:

```
javac mypackage/HelloWorld.java
```

5. **Use the package in another program:**

- To use the HelloWorld class, import it into another Java program.

```
import mypackage.HelloWorld;

public class Test {
    public static void main(String[] args) {
        HelloWorld obj = new HelloWorld();
        obj.showMessage();  // Prints: Hello from the package!
    }
}
```

6. **Compile and run the program:**
- Compile both classes:

Summary:
- A package groups related classes.
- Use the package keyword at the top of your file.
- Save the file in a folder with the package name.
- Import and use the package in another file.

6. Write a program that contains one method that will throw an IllegalAccessException and use proper exception handles so that the exception should be printed.

```
public class ExceptionDemo {

    // Method that throws IllegalAccessException
    public static void checkAccess() throws IllegalAccessException {
        // Throwing the exception
        throw new IllegalAccessException("Access is not allowed!");
    }

    public static void main(String[] args) {
        try {
            // Calling the method that throws the exception
            checkAccess();
        } catch (IllegalAccessException e) {
            // Handling the exception
            System.out.println("Exception caught: " + e.getMessage());
        }
    }
}
```

Explanation:

1. **Method checkAccess:** This method throws an Illega**lAccessException when called.**
2. **Try-Catch Block:** In the main method, the checkAccess method is called inside a try block. If it throws an exception, the catch block catches it and prints the error message.

Output:

```
Exception caught: Access is not allowed!
```

7. Define an exception. What are the key terms used in exception handling? Explain.

An exception in Java is an event that disrupts the normal flow of program execution. It occurs when the program encounters an unexpected condition, like dividing by zero, accessing an invalid array index, or trying to read a file that doesn't exist.

Key Terms in Exception Handling:

1. **Exception:**
   - An exception is an event that disrupts the normal flow of a program.
   - **Example:** ArithmeticException, NullPointerException.

2. **Throwable:**
   - The superclass of all errors and exceptions in Java.
   - It has two main subclasses: Error (severe errors) and Exception (which can be handled).

3. **Try Block:**
   - The block of code where exceptions might occur.
   - **Syntax:**

```
try {
   // Code that may throw an exception
}
```

4. **Catch Block:**
   - It catches and handles exceptions thrown in the try block.
   - **Syntax:**

```
catch (ExceptionType e) {
   // Handle the exception
}
```

5. **Finally Block:**
   - It is optional and executes no matter what, even if an exception was caught or not.
   - Typically used to close resources (e.g., files).
   - **Syntax:**

```
finally {
   // Cleanup code
}
```

6. **Throw:**
- Used to explicitly throw an exception from a method.
- **Syntax:**

```
throw new ExceptionType(“Message");
```

7. **Throws:**
- Used in a method signature to declare that a method might throw an exception.
- **Syntax:**

```
public void myMethod() throws IOException {
   // Code that may throw an exception
}
```

Example

```
public class Example {
   public static void main(String[] args) {
      try {
         int result = 10 / 0;  // This will cause an ArithmeticException
      } catch (ArithmeticException e) {
         System.out.println("Exception caught: " + e.getMessage());
      } finally {
         System.out.println("This is the finally block, it runs always.");
      }
   }
}
```

Explanation:

1. Try block contains code that might throw an exception.
2. Catch block catches the exception and handles it.
3. Finally block runs whether or not an exception occurred.

8. Explain various levels of accessing protections available for packages and their implications with suitable example

Access Modifiers in Java:

In Java, access modifiers are used to define the visibility and accessibility of classes, methods, variables, and constructors. These modifiers control how the members of a class can be accessed, either from within the same class, package, or outside the package.

The four access levels in Java are:

1. Public Access Modifier:

- Definition: The public modifier allows the class, method, or variable to be accessible from anywhere in the program.
- Implication: Any class, method, or variable marked as public can be accessed by any other class, whether it's in the same package or a different package.

- **Example:**

```
// Class with public access
public class MyClass {
    public int num = 10;  // Public variable

    public void display() {  // Public method
        System.out.println("Value: " + num);
    }
}

// Accessing the public class and its members from another class
public class Test {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        System.out.println(obj.num);  // Accessing public variable
        obj.display();  // Accessing public method
    }
}
```

2. Protected Access Modifier:

- Definition: The protected modifier allows access to the class, method, or variable within the same package or from subclasses (even if they are in different packages).

- Implication: It provides more restricted access than public, allowing access only within the package or through inheritance (i.e., subclass).
- **Example:**

```
// Class with protected access
class MyClass {
    protected int num = 20;  // Protected variable

    protected void display() {  // Protected method
        System.out.println("Value: " + num);
    }
}

// Accessing protected class and its members in a subclass
public class Test extends MyClass {
    public static void main(String[] args) {
        Test obj = new Test();
              System.out.println(obj.num);    // Accessing protected variable in
subclass
        obj.display();  // Accessing protected method in subclass
    }
}
```

3. Default (Package-Private) Access Modifier:

- Definition: If no access modifier is specified, Java applies default access (also called package-private). The class, method, or variable is accessible only within the same package.
- Implication: Members with default access are not visible to classes outside their package.

- **Example:**

```
// Class with default access (no access modifier)
class MyClass {
    int num = 30;  // Default variable

    void display() {  // Default method
        System.out.println("Value: " + num);
    }
}

// Accessing default class and its members in the same package
public class Test {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        System.out.println(obj.num);  // Accessing default variable
        obj.display();  // Accessing default method
    }
}
```

4. Private Access Modifier:

- **Definition:** The private modifier restricts access to the class, method, or variable to only within the same class. It cannot be accessed by other classes, even within the same package.

- **Implication:** This is the most restrictive level of access, used for hiding data and enforcing encapsulation. It ensures that the data is private to the class and cannot be accessed directly from outside.

- **Example:**

```
public class MyClass {
    private int num = 40;  // Private variable

    private void display() {  // Private method
        System.out.println("Value: " + num);
    }

    public void accessPrivate() {  // Public method to access private members
        System.out.println("Accessing private value: " + num);
    }
}

public class Test {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        // System.out.println(obj.num);  // Error: num has private access
        // obj.display();  // Error: display() has private access
        obj.accessPrivate();  // Accessing private method via public method
    }
}
```

Implications:

- **Public:** No restriction on access. Any class can use the public class or its members.

- **Protected:** Access within the same package and through inheritance (subclassing).

- **Default:** Access within the same package. No access from outside the package.

- **Private:** Access only within the same class. Data is hidden from other classes to ensure security and encapsulation.

This structure helps maintain data hiding and security in Java, allowing you to control how and where your classes, methods, and variables can be accessed.

9. Develop a program to create package "balance" containing account class with class displayBalance() method and import this package in another program to access method of account class

Step 1: Create the Package (balance)

1. Save the following code in a file named Account.java.
2. Ensure the file is in a folder named balance (same as the package name).

File: balance/Account.java

```
package balance; // Declaring the package

public class Account {
    public void displayBalance() {
        System.out.println("Your account balance is: ₹10,000");
    }
}
```

Step 2: Use the Package in Another Program

1. Save the following code in a file named TestBalance.java.
2. Place it outside the balance folder.

File: TestBalance.java

```
import balance.Account; // Importing the Account class from the balance package

public class TestBalance {
    public static void main(String[] args) {
        Account acc = new Account(); // Creating an object of Account class
        acc.displayBalance();        // Calling the displayBalance() method
    }
}
```

How to Compile and Run:

1. Compile the Package:
   - Open the terminal and go to the folder containing the balance folder.
   - Compile the Account.java file:

```
javac balance/Account.java
```

2. **Compile the Main Program:**

- Compile the TestBalance.java file:

```
javac TestBalance.java
```

3. **Run the Program:**

- Run the TestBalance program:

```
java TestBalance
```

Output :-

```
Your account balance is: ₹10,000
```

10. Build a Java program for a banking application to throw an exception, where a person tries to withdraw the amount even though he/she has lesser than minimum balance (Create a custom exception

```
// Custom Exception Class
class InsufficientBalanceException extends Exception {
    public InsufficientBalanceException(String message) {
        super(message); // Pass the message to the Exception class
    }
}

// Main Class
public class BankingApp {
    public static void main(String[] args) {
        double balance = 5000; // Fixed account balance

        try {
            double amountToWithdraw = 6000; // Trying to withdraw ₹6000
            if (amountToWithdraw > balance) {
                    // Throw custom exception if withdrawal amount is greater than
balance
                 throw new InsufficientBalanceException("Insufficient balance! You
have only ₹" + balance);
            } else {
                balance -= amountToWithdraw;
                 System.out.println("Withdrawal successful! Remaining balance: ₹"
+ balance);
            }
        } catch (InsufficientBalanceException e) {
            // Handle the exception
            System.out.println("Exception: " + e.getMessage());
        }

        try {
            double amountToWithdraw = 3000; // Trying to withdraw ₹3000
            if (amountToWithdraw > balance) {
                    // Throw custom exception if withdrawal amount is greater than
balance
                 throw new InsufficientBalanceException("Insufficient balance! You
have only ₹" + balance);
            } else {
                balance -= amountToWithdraw;
                System.out.println("Withdrawal successful! Remaining balance: ₹"
+ balance);
            }
        } catch (InsufficientBalanceException e) {
            // Handle the exception
            System.out.println("Exception: " + e.getMessage());
        }
    }
}
```

Op;-

```
Exception: Insufficient balance! You have only ₹5000

Withdrawal successful! Remaining balance: ₹2000
```

11. creating a package named mypack

File 1: mypack.java

```
// File: mypack/mypack.java
package mypack;

public class mypack {
    public void display() {
        System.out.println("HELLO FROM MYPACK");
    }
}
```

File 2: Testpack.java

```
// File: mypack/Testpack.java
package mypack;

public class Testpack {
    public static void main(String[] args) {
        mypack obj = new mypack();
            obj.display();   // Call the display() method from mypack
class
    }
}
```

12. Custom Exception for Zero Division

```
import java.util.Scanner;

// Custom exception class for division by zero
public class ZeroD extends Exception {
    public ZeroD(String msg) {
        super(msg);
    }
}

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

        System.out.println("Enter 2 numbers: ");
        int a = sc.nextInt();
        int b = sc.nextInt();

        try {
            // Check for division by zero
            if (b == 0) {
                throw new ZeroD("Zero division not allowed");
            }
            float res = (float) a / b;
            System.out.println("Result: " + res);  // Perform division if no
error
        } catch (ZeroD e) {
            System.out.println(e.getMessage());  // Handle the exception
if division by zero occurs
        } finally {
            sc.close();  // Close the scanner to avoid resource leak
        }
    }
}
```