Create a custom java package and class for the following problem:

i) Given the curent array of numbers [10, 0, 20, 30,5, 21, 30), find the largest and smallest value in the unsorted array.

Ans:

Step 1: Custom Package and Class (ArrayHelper.java)

```
// File: ArrayHelper.java package
myutils;

public class ArrayHelper {

   public static int findMax(int[] arr) {
int max = arr[0];       for (int num :
arr) {          if (num > max) max =
num;
      }
      return max;
   }

   public static int findMin(int[] arr) {
int min = arr[0];       for (int num :
arr) {          if (num < min) min =
num;
      }
      return min;
   }
}
```

Step 2: Use the class in main program (Main.java)

```
// File: Main.java import
myutils.ArrayHelper; public
```

```
class Main {     public static
void main(String[] args) {
int[] numbers = {10, 0, 20,
30, 5, 21, 30};

      int max = ArrayHelper.findMax(numbers);
int min = ArrayHelper.findMin(numbers);

      System.out.println("Maximum: " + max);
      System.out.println("Minimum: " + min);
   }
}
```

Maximum: 30

Minimum: 0

***☑ Why Interfaces are Useful:**

1) **Support multiple inheritance**
2) **Ensure a class follows certain behavior**
3) **Used in event handling, multithreading (e.g., Runnable interface), etc. ***Describe the various form of implementing interfices. Give examples for cach case.**

***Describe the various form of implementing interfaces. Give examples for each case.

In Java, **interfaces** define a set of abstract methods that must be implemented by classes. There are **four main ways** to implement interfaces:

1. Using a Class

A class uses `implements` keyword to implement an interface.

Java Copy code

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

2. Using an Anonymous Class

Used when implementation is needed only once.

Java Copy code

```
Animal a = new Animal() {
    public void sound() {

System.out.println("Anonymous dog
barks");
    }
};
```

3. Using Multiple Interfaces

A class can implement more than one interface.

Java Copy code

```
interface A { void show(); }
interface B { void display(); }

class Demo implements A, B {
    public void show()
{ System.out.println("Show"); }
    public void display()
{ System.out.println("Display"); }
}
```

✅ 2. Implementation by an Abstract Class

An **abstract class** can implement an interface **partially** (i.e., not all methods). The remaining methods must be implemented by a subclass.

📌 **Example:**

java

Copy Edit

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

abstract class WildAnimal implements Animal {
    public void eat() {
        System.out.println("Eating...");
    }
}

class Tiger extends WildAnimal {
    public void makeSound() {
        System.out.println("Roar");
    }
}
```

3. Using Lambda Expression

For interfaces with only one method (functional interface).

java

```java
interface Calculator {
    int add(int a, int b);
}

public class Main {
    public static void main(String[] args) {
        Calculator c = (a, b) -> a + b;
        System.out.println(c.add(5, 3));
    }
}
```

*****The Collections class of Java Collection Framework has several methods (e.g. sort(), min(), max()) that require natural ordering of the classes to be handled. Explain the two possible conditions that must be satisfied on the classes to be able to use the sort() method to arrange the list of objects of these classes. (Marks: 4)**

☑ **Answer:**

To use the Collections.sort() method, two conditions must be fulfilled:

1. Class must implement Comparable interface

– The class should override compareTo() method to define how objects will be compared.

2. All elements must be mutually comparable

– The list should not contain incompatible object types, or ClassCastException will occur

Example:

```
class Student implements  Comparable<Student>
{
   public int compareTo(Student s) {
      return this.id - s.id;
   }
}
```

***

Threads can be used to make your Java applications faster. They allow you to run multiple things at the same time. You must think of a good scenario for using threads in real-world and discuss your scenario. *(Marks: 4)*

✅ **Answer:**
Threads allow multiple tasks to run at the same time, increasing speed and efficiency.

Scenario: File Downloader
When downloading multiple files, each thread can download one file. This allows all files to download simultaneously.

Example:

```
Thread t1 = new Thread(new
Downloader("file1"));
t1.start();
```

Advantages:
– Faster download
– Saves time
– Better performance

***** Define an interface `Animal` with methods `eat()` and `sleep()`. Create classes `Dog` and `Cat` that implement the `Animal` interface. Write a program (Do not write full code) to demonstrate interface implementation by creating objects of both classes and calling the methods.** *(Marks: 5)*

Interface Definition:

```java
public interface Animal {
    public void eat();
    public void sleep();
}
```

Class Implementations:

```java
// Dog implements Animal
public class Dog implements Animal {
    public void eat() { /*...*/ }
    public void sleep() { /*...*/ }
}

// Cat implements Animal
public class Cat implements Animal {
    public void eat() { /*...*/ }
    public void sleep() { /*...*/ }
}
```

Main method (partial):

```java
Animal d = new Dog();
d.eat();
d.sleep();

Animal c = new Cat();
c.eat();
c.sleep();
```

✅ Full Java Code:

java

```java
// File: TestInterface.java

// Interface declaration
public interface Animal {
    public void eat();
    public void sleep();
}

// Dog class implements Animal
public class Dog implements Animal {
    public void eat() {
        System.out.println("Dog eats bones.");
    }

    public void sleep() {
        System.out.println("Dog sleeps in kennel.");
    }
}

// Cat class implements Animal
public class Cat implements Animal {
    public void eat() {
        System.out.println("Cat eats fish.");
    }

    public void sleep() {
        System.out.println("Cat sleeps on couch.");
    }
}

// Main class to demonstrate interface implementation
public class TestInterface {
    public static void main(String[] args) {
        // Creating Animal references to Dog and Cat objects
        Animal d = new Dog();
        Animal c = new Cat();

        // Calling methods using interface references
        d.eat();    // Output: Dog eats bones.
        d.sleep();  // Output: Dog sleeps in kennel.

        c.eat();    // Output: Cat eats fish.
        c.sleep();  // Output: Cat sleeps on couch.
    }
}
```

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

class Dog implements Animal {
    public void eat() {
        System.out.println("Dog is eating");
    }
    public void sleep() {
        System.out.println("Dog is sleeping");
    }
}

class Cat implements Animal {
    public void eat() {
        System.out.println("Cat is eating");
    }
    public void sleep() {
        System.out.println("Cat is sleeping");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal d = new Dog();
        d.eat();
        d.sleep();

        Animal c = new Cat();
        c.eat();
        c.sleep();
    }
}
```

***What is a package? How can you define your own package?

A package is a collection of related classes and interfaces that provides a way to organize Java programs.

Packages help to **organize code logically**, **avoid name conflicts**, and **provide access protection**.

There are two types of packages in Java:

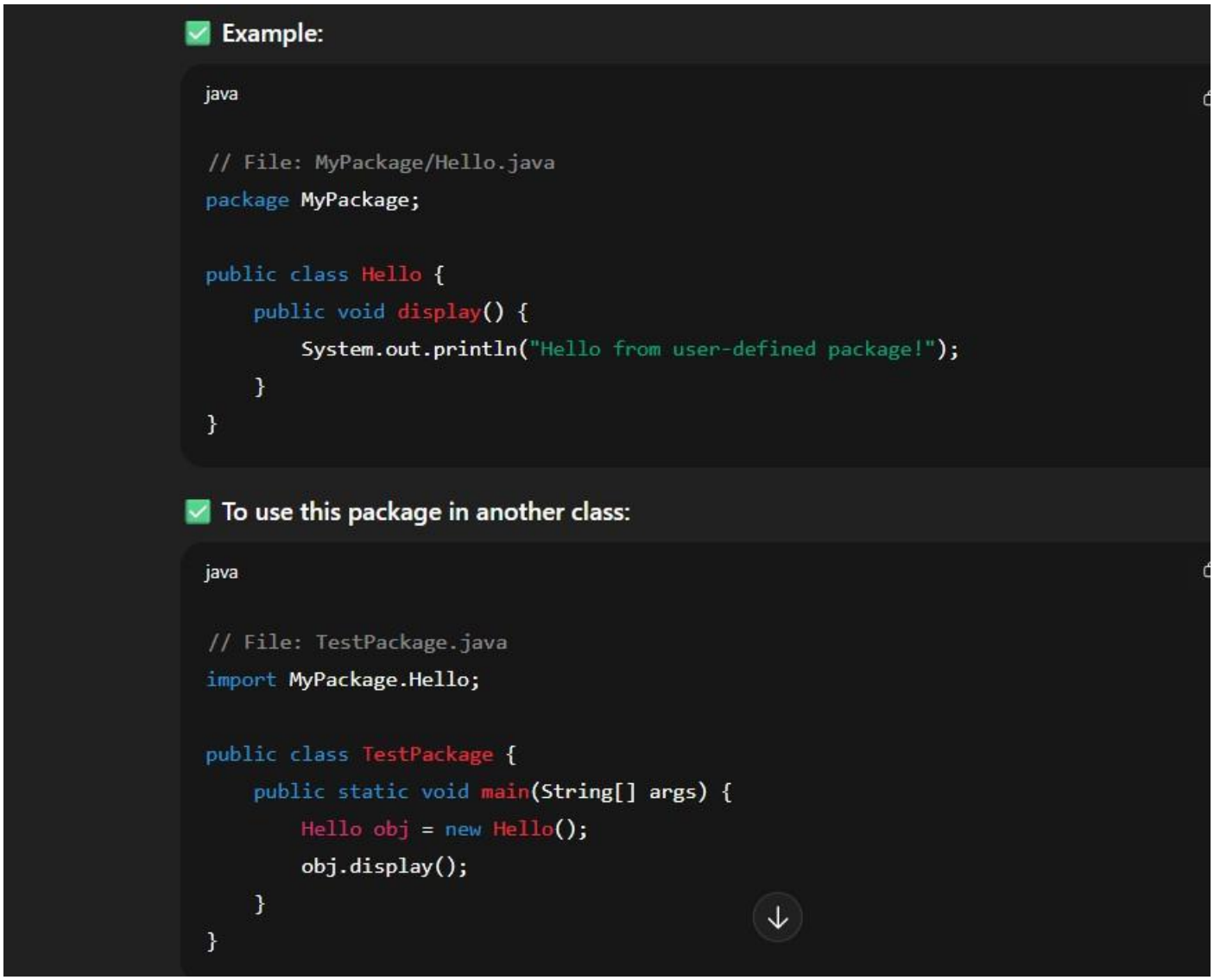
1. **Built-in Packages** – Provided by Java API (e.g., `java.util`, `java.io`)
2. **User-defined Packages** – Created by programmers to group their own classes

---

How to Define Your Own Package

To define your own package, follow these steps:

1. **Use the `package` keyword** at the very top of your Java file.
2. **Create a directory** with the same name as your package.
3. **Save your class file** inside that directory.



***Write a Java program to create a class Person with attributes name, age, and gender. Add a method displayDetails() to print the person's details. Create an object of the Person class in the main method and demonstrate its usage.

Using constructor

```java
public class Person {
    // Attributes (Instance variables)
    String name;
    int age;
    String gender;

    // Constructor to initialize the object
    Person(String n, int a, String g) {
        name = n;
        age = a;
        gender = g;
    }

    // Method to display person details
    public void displayDetails() {
        System.out.println("Name   : " + name);
        System.out.println("Age    : " + age);
        System.out.println("Gender : " + gender);
    }

    // main method to test the class
    public static void main(String[] args) {
        // Creating object of Person class
        Person p1 = new Person("Rafi", 25, "Male");

        // Displaying details using method
        p1.displayDetails();
    }
}
```

✅ Without Constructor Version (exam er jonno আর সহজ)

java

```java
public class Person {
    // Attributes
    String name;
    int age;
    String gender;

    // Method to display person details
    public void displayDetails() {
        System.out.println("Name   : " + name);
        System.out.println("Age    : " + age);
        System.out.println("Gender : " + gender);
    }

    // main method
    public static void main(String[] args) {
        Person p1 = new Person(); // default constructor used

        // manually assign values
        p1.name = "Rafi";
        p1.age = 25;
        p1.gender = "Male";

        p1.displayDetails(); // method call
    }
}
```

*****Design class Book with attributes title, author, and price. Implement multiple constructors to initialize a book object with:**

Only the title.

. Title and author.

. Title, author, and price.

Demonstrate the use of constructor overloading in the main method.

```java
public class Book {
    // Attributes
    String title;
    String author;
    double price;

    // Constructor 1: Only title
    Book(String title) {
        this.title = title;
        this.author = "Unknown";
        this.price = 0.0;
    }

    // Constructor 2: Title and author
    Book(String title, String author) {
        this.title = title;
        this.author = author;
        this.price = 0.0;
    }

    // Constructor 3: Title, author, and price
    Book(String title, String author, double price) {
        this.title = title;
        this.author = author;
        this.price = price;
    }
```

Copy Edit

```java
    // Method to display book details
    void display() {
        System.out.println("Title : " + title);
        System.out.println("Author: " + author);
        System.out.println("Price : " + price);
        System.out.println("--------------------");
    }

    // main method to demonstrate constructor overloading
    public static void main(String[] args) {
        // Creating book objects using different constructors
        Book b1 = new Book("Java Basics");
        Book b2 = new Book("OOP Concepts", "John Smith");
        Book b3 = new Book("Data Structures", "Jane Doe", 450.0);

        // Displaying the book details
        b1.display();
        b2.display();
        b3.display();
    }
}
```

Or,

```java
// Book.java
public class Book {
    // Attributes
    String title;
    String author;
    double price;

    // Constructor 1: Only title
    public Book(String t) {
        title = t;
        author = "Unknown";
        price = 0.0;
    }

    // Constructor 2: Title and author
    public Book(String t, String a) {
        title = t;
        author = a;
        price = 0.0;
    }

    // Constructor 3: Title, author, and price
    public Book(String t, String a, double p) {
        title = t;
        author = a;
        price = p;
    }

    // Method to display book details
    public void display() {
        System.out.println("Title : " + title);
        System.out.println("Author: " + author);
        System.out.println("Price : $" + price);
        System.out.println();
    }

    // Main method to demonstrate constructor overloading
    public static void main(String[] args) {
        // Using different constructors
        Book b1 = new Book("Java Programming");
        Book b2 = new Book("Python Basics", "Alice");
        Book b3 = new Book("C++ Guide", "Bob", 399.99);

        // Displaying book details
        b1.display();
        b2.display();
        b3.display();
    }
}
```

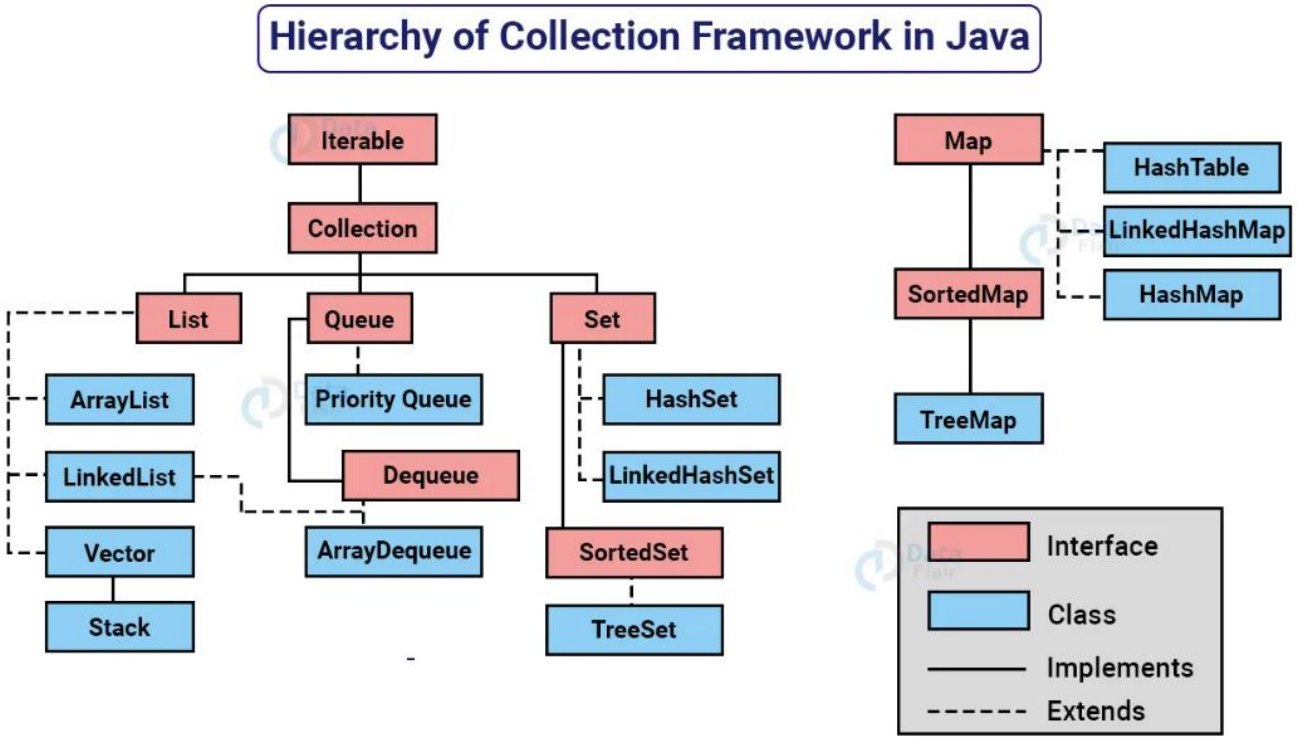
*****Create a class BankAccount with private attributes account Number, balance, and account Holder. Provide getter and setter methods to securely access and modify these attributes. Write a Java program structure (do not write full code) to demonstrate encapsulation with proper validations in the setter methods (e.g., balance cannot be negative).**

java

```java
public class BankAccount {
    // Private attributes
    private String accountNumber;
    private double balance;
    private String accountHolder;

    // Getter and Setter for accountNumber
    public String getAccountNumber() {
        return accountNumber;
    }

    public void setAccountNumber(String acc) {
        accountNumber = acc;
    }
```

```java
    // Getter and Setter for balance with validation
    public double getBalance() {
        return balance;
    }

    public void setBalance(double bal) {
        if (bal >= 0) {
            balance = bal;
        } else {
            System.out.println("Balance cannot be negative");
        }
    }

    // Getter and Setter for accountHolder
    public String getAccountHolder() {
        return accountHolder;
    }

    public void setAccountHolder(String holder) {
        accountHolder = holder;
    }
}
```

```java
class Main {
    public static void main(String[] args) {
        // Creating object
        BankAccount acc = new BankAccount();

        // Using setters to assign values
        acc.setAccountNumber("A101");
        acc.setAccountHolder("Rafi");
        acc.setBalance(5000);

        // Using getters to print values
        System.out.println("Name: " + acc.getAccountHolder());
        System.out.println("Balance: " + acc.getBalance());
    }
}
```



*** **Question:**
Create a class `Calculator` with a method `divide(int a, int b)` that performs division and throws an Arithmetic Exception if `b` is zero. Write a program (without full-code) to handle this exception using a `try-catch` block, and display an appropriate error message.

```java
class Calculator {
    public int divide(int a, int b) {
        if (b == 0)
            throw new ArithmeticException("Division by zero not allowed");
        return a / b;
    }
}
```

✅ **Using try-catch in main:**

```java
Calculator c = new Calculator();
try {
    int result = c.divide(10, 0);
    System.out.println("Result: " + result);
} catch (ArithmeticException e) {
    System.out.println("Error: " + e.getMessage());
}
```

✅ **Output:**

```vbnet
Error: Division by zero not allowed
```

***Define a base class Vehicle with attributes make, model, and year, and a method displayInfo(). Create a derived class Car that adds an attribute numberOfDoors. Write a program structure (do not write full code) in Java to demonstrate inheritance by creating objects of both Vehicle and Car.

```java
// Base class Vehicle
class Vehicle {
    String make;
    String model;
    int year;

    void displayInfo() {
        System.out.println("Make: " + make);
        System.out.println("Model: " + model);
        System.out.println("Year: " + year);
    }
}

// Derived class Car extends Vehicle
class Car extends Vehicle {
    int numberOfDoors;

    void displayCarInfo() {
        displayInfo();  // Call base class method
        System.out.println("Number of Doors: " + numberOfDoors);
    }
}
```

```java
// Main class to create objects and demonstrate inheritance
class Main {
    public static void main(String[] args) {
        Vehicle vehicle = new Vehicle();
        vehicle.make = "Toyota";
        vehicle.model = "Corolla";
        vehicle.year = 2020;
        vehicle.displayInfo();

        Car car = new Car();
        car.make = "Honda";
        car.model = "Civic";
        car.year = 2022;
        car.numberOfDoors = 4;
        car.displayCarInfo();
    }
}
```

*****Suppose you have a Piggie Bank with an initial amount of $50 and you have to add some more amount to it. Create a class 'AddAmount' with a data member named 'amount' with an initial value of $50. Now make two constructors of this class as follows:**

.without any parameter no amount will be added to the Piggie Bank

.having a parameter which is the amount that will be added to Piggie Bank

Create an object of the 'AddAmount' class and display the final amount in Piggie Bank.

```
class AddAmount {
    int amount = 50;  // Initial amount in Piggie Bank

    // Constructor without parameter: no amount added
    AddAmount() {
        // No change to amount
    }

    // Constructor with parameter: add given amount
    AddAmount(int add) {
        amount += add;  // Add the given amount to Piggie Bank
    }

    // Method to display final amount
    void displayAmount() {
        System.out.println("Final amount in Piggie Bank: $" + amount);
    }
}

public class Main {
    public static void main(String[] args) {
        // Case 1: No amount added
        AddAmount obj1 = new AddAmount();
        obj1.displayAmount();

        // Case 2: Add $100
        AddAmount obj2 = new AddAmount(100);
        obj2.displayAmount();
    }
}
```

Final amount in Piggie Bank: $50

Final amount in Piggie Bank: $150

***Create a Java program that implements an abstract class Animal that has a Name property of type text and three methods SetName (string name), GetName and Eat. The Eat method will be an abstract method of type void.

You will also need to create a Dog class that implements the above Animal class and the Eat method that says the 'Dog is Eating".

To create a new Dog type object from the Main of the program using the dog's name, give the Dog object a name, and then execute the GetName and Eat methods.

```
// Abstract class Animal
abstract class Animal {
    String name;  // Name property

    // SetName method
    void SetName(String name) {
        this.name = name;
    }

    // GetName method
    String GetName() {
        return name;
    }

    // Abstract method
    abstract void Eat();
}

// Dog class inherits Animal
class Dog extends Animal {
    // Implementation of abstract method Eat
    void Eat() {
        System.out.println("Dog is Eating");
    }
}
```

Dog's Name: Tommy

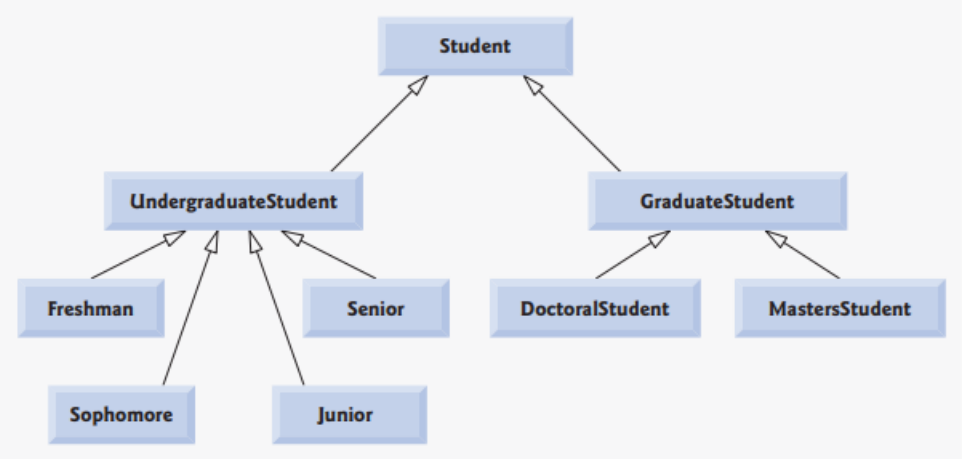
Dog is Eating

```
// Main class to test
public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();              // Creating Dog object
        myDog.SetName("Tommy");             // Setting the name
        System.out.println("Dog's Name: " + myDog.GetName()); // Getting the name
        myDog.Eat();                        // Calling Eat method
    }
}
```

***Write a code using inheritance to create an exception superclass (called ExceptionA) and exception subclasses ExceptionB and ExceptionC, where ExceptionB inherits from ExceptionA and ExceptionC inherits from ExceptionB. Write a program to demonstrate that the catch block for type ExceptionA catches exceptions of types ExceptionB and ExceptionC.

```java
// Superclass ExceptionA
class ExceptionA extends Exception {
    public ExceptionA(String message) {
        super(message);
    }
}

// Subclass ExceptionB inherits ExceptionA
class ExceptionB extends ExceptionA {
    public ExceptionB(String message) {
        super(message);
    }
}

// Subclass ExceptionC inherits ExceptionB
class ExceptionC extends ExceptionB {
    public ExceptionC(String message) {
        super(message);
    }
}

public class ExceptionTest {
    public static void main(String[] args) {
        try {
            // Throwing an object of ExceptionC
            throw new ExceptionC("This is ExceptionC");
        }
        catch (ExceptionA e) {
            // This block catches ExceptionA, ExceptionB, and ExceptionC
            System.out.println("Caught: " + e.getMessage());
        }
    }
}
```

*****Write the code and draw an inheritance hierarchy for students at a university. Use Student as the superclass of the hierarchy, and then extend Student with classes Undergraduate Student and GraduateStudent. Continue to extend the hierarchy as deep (ie, as many levels) as possible. For example, Freshman, Sophomore, Junior and Senior might extend Undergraduate Student, and DoctoralStudent and MastersStudent might be subclasses of GraduateStudent. After drawing the hierarchy, discuss the relationships that exist between the classes**.



This hierarchy contains many *is-a* (inheritance) relationships. An `UndergraduateStudent` *is a* `Student`. A `GraduateStudent` *is a* `Student`, too. Each of the classes `Freshman`, `Sophomore`, `Junior` and `Senior` *is an* `UndergraduateStudent` and *is a* `Student`. Each of the classes `DoctoralStudent` and `MastersStudent` *is a* `GraduateStudent` and *is a* `Student`. Note that there could be *many* more classes. For example, `GraduateStudent` could have subclasses like `LawStudent`, `BusinessStudent`, `MedicalStudent`, etc.

Discussion (Short Version)

- `Student` is the base class with common properties like `name`, `id`, and `displayInfo()`.
- `UndergraduateStudent` and `GraduateStudent` inherit from `Student` and get its features.
- `Freshman`, `Sophomore`, `Junior`, and `Senior` are levels of undergraduate students.
- `MastersStudent` and `DoctoralStudent` are types of graduate students.
- This shows an **"is-a" relationship**:
  - `Freshman` is an `UndergraduateStudent`
  - `UndergraduateStudent` is a `Student`
  - `DoctoralStudent` is a `GraduateStudent`
- Each class extends the one above and adds specific behavior.

```java
// Superclass
class Student {
    String name;
    int id;

    void displayInfo() {
        System.out.println("Name: " + name + ", ID: " + id);
    }
}

// Undergraduate Student class
class UndergraduateStudent extends Student {
    void study() {
        System.out.println("Undergraduate student studying...");
    }
}

// Freshman subclass
class Freshman extends UndergraduateStudent {
    void level() {
        System.out.println("Level: Freshman");
    }
}

// Sophomore subclass
class Sophomore extends UndergraduateStudent {
    void level() {
        System.out.println("Level: Sophomore");
    }
}
```

```java
// Junior subclass
class Junior extends UndergraduateStudent {
    void level() {
        System.out.println("Level: Junior");
    }
}

// Senior subclass
class Senior extends UndergraduateStudent {
    void level() {
        System.out.println("Level: Senior");
    }
}

// Graduate Student class
class GraduateStudent extends Student {
    void research() {
        System.out.println("Graduate student doing research...");
    }
}

// Masters student subclass
class MastersStudent extends GraduateStudent {
    void program() {
        System.out.println("Program: Master's");
    }
}
```

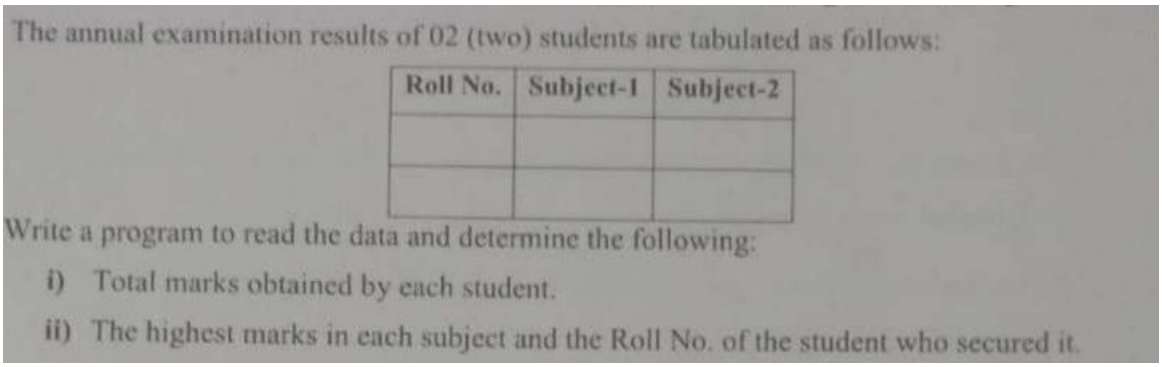
```java
// Doctoral student subclass
class DoctoralStudent extends GraduateStudent {
    void program() {
        System.out.println("Program: PhD");
    }
}

// Main class to test
public class UniversityTest {
    public static void main(String[] args) {
        Freshman f = new Freshman();
        f.name = "Alice";
        f.id = 101;
        f.displayInfo();
        f.level();
        f.study();

        DoctoralStudent d = new DoctoralStudent();
        d.name = "Dr. Bob";
        d.id = 501;
        d.displayInfo();
        d.program();
        d.research();
    }
}
```

*****You have three tasks. All of the three tasks need to be done in parallel. In the first task "TaskA" you need to output "Hello World" 100 times. In second task "TaskB" you need to add integers from 1 to 1000, and in the final task "TaskC" print "I can do it" 200 times and wait for 1 second before writing the line each time. Now ensure that "TaskC" will be completed first, then "TaskA" and then "TaskB". Write down java code for the above problem.**

```java
// TaskA: Print "Hello World" 100 times
class TaskA extends Thread {
    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.println("Hello World");
        }
    }
}

// TaskB: Add numbers from 1 to 1000
class TaskB extends Thread {
    public void run() {
        int sum = 0;
        for (int i = 1; i <= 1000; i++) {
            sum += i;
        }
        System.out.println("Sum = " + sum);
    }
}

// TaskC: Print "I can do it" 200 times, wait 1 second each time
class TaskC extends Thread {
    public void run() {
        for (int i = 0; i < 200; i++) {
            System.out.println("I can do it");
            try {
                Thread.sleep(1000); // wait 1 second
            } catch (Exception e) {
                System.out.println("Error in TaskC");
            }
        }
    }
}
```

```java
// Main class to run all tasks
public class Main {
    public static void main(String[] args) {
        // Create object of each task
        TaskA a = new TaskA();
        TaskB b = new TaskB();
        TaskC c = new TaskC();

        // Start all threads
        a.start();
        b.start();
        c.start();

        try {
            // Wait for TaskC to finish first
            c.join();
            // Then wait for TaskA to finish
            a.join();
            // Then wait for TaskB to finish
            b.join();
        } catch (Exception e) {
            System.out.println("Main thread error");
        }

        // Final message after all tasks done
        System.out.println("All tasks completed in order: C -> A -> B");
    }
}
```

***



The annual examination results of 02 (two) students are tabulated as follows:

Roll No.	Subject-1	Subject-2

Write a program to read the data and determine the following:

i) Total marks obtained by each student.

ii) The highest marks in each subject and the Roll No. of the student who secured it.

```java
import java.util.Scanner;

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

        // Data for 2 students: roll, subject1, subject2
        int roll1, sub1_1, sub2_1;
        int roll2, sub1_2, sub2_2;

        // Input for Student 1
        System.out.print("Enter Roll No of Student 1: ");
        roll1 = sc.nextInt();
        System.out.print("Enter marks in Subject 1: ");
        sub1_1 = sc.nextInt();
        System.out.print("Enter marks in Subject 2: ");
        sub2_1 = sc.nextInt();

        // Input for Student 2
        System.out.print("Enter Roll No of Student 2: ");
        roll2 = sc.nextInt();
        System.out.print("Enter marks in Subject 1: ");
        sub1_2 = sc.nextInt();
        System.out.print("Enter marks in Subject 2: ");
        sub2_2 = sc.nextInt();

        // 1. Total marks of each student
        int total1 = sub1_1 + sub2_1;
        int total2 = sub1_2 + sub2_2;

        System.out.println("\nTotal Marks:");
        System.out.println("Roll " + roll1 + ": " + total1);
        System.out.println("Roll " + roll2 + ": " + total2);

        // 2. Highest in Subject 1
        if (sub1_1 > sub1_2) {
            System.out.println("\nHighest in Subject 1: " + sub1_1 + " by Roll " + roll1);
        } else {
            System.out.println("\nHighest in Subject 1: " + sub1_2 + " by Roll " + roll2);
        }

        // 3. Highest in Subject 2
        if (sub2_1 > sub2_2) {
            System.out.println("Highest in Subject 2: " + sub2_1 + " by Roll " + roll1);
        } else {
            System.out.println("Highest in Subject 2: " + sub2_2 + " by Roll " + roll2);
        }
    }
}
```

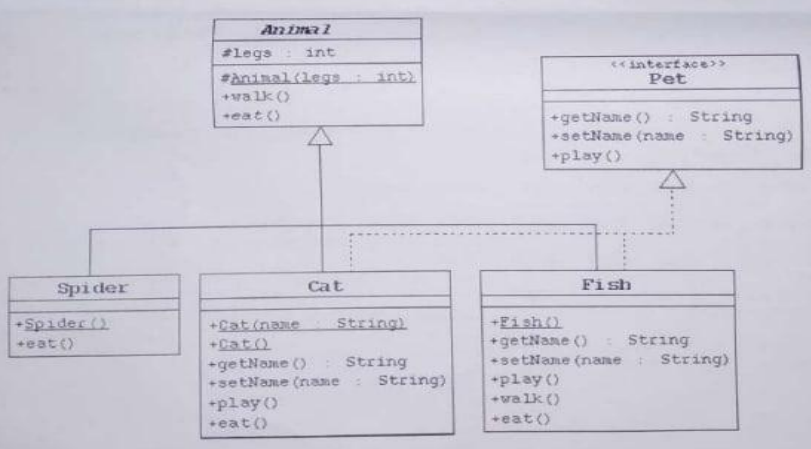
3.a) Design a class to represent Bank Account. Include the following members:

Data members:	Methods:
• Name of the depositor • Account number • Type of account • Balance amount in the account	• To assign initial values • To deposit an amount • To withdraw an amount • To display (name and balance)

N.B: You do not need to write complete source code.

```java
class BankAccount {

    // 🔸 Data Members
    String name;           // Name of the depositor
    String accountNumber;  // Account number
    String accountType;    // Type of account (e.g., Savings, Current)
    double balance;        // Balance amount in the account

    // 🔸 Method 1: To assign initial values
    void setDetails(String n, String accNo, String type, double initialBalance) {
        name = n;
        accountNumber = accNo;
        accountType = type;
        balance = initialBalance;
    }

    // 🔸 Method 2: To deposit an amount
    void deposit(double amount) {
        balance += amount;
    }

    // 🔸 Method 3: To withdraw an amount
    void withdraw(double amount) {
        if (amount <= balance) {
            balance -= amount;
        } else {
            System.out.println("Insufficient balance");
        }
    }

    // 🔸 Method 4: To display name and balance
    void display() {
        System.out.println("Depositor Name: " + name);
        System.out.println("Balance: " + balance);
    }
}
```

b) In the following **Class Diagram**, you will see a hierarchy of animals that is rooted in an abstract class ***Animal***. Several of the animal classes will implement an interface called ***Pet***. 6

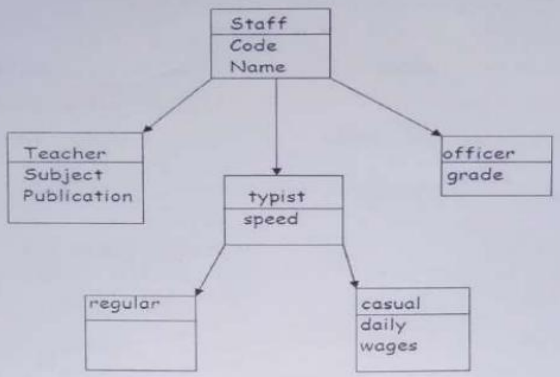


Create the **Animal** class, which is the abstract superclass of all animals.

i) Declare a protected integer attribute called legs, which records the number of legs for this animal.
ii) Define a protected constructor that initializes the legs attribute.
iii) Declare an abstract method eat.
iv) Declare a concrete method walk that prints out something about how the animals walks (include the number of legs).

```java
public abstract class Animal {

    // i) Protected integer attribute
    protected int legs;

    // ii) Constructor to initialize legs
    protected Animal(int legs) {
        this.legs = legs;
    }

    // iii) Abstract method
    public abstract void eat();

    // iv) Concrete method to describe walking
    public void walk() {
        System.out.println("This animal walks using " + legs + " legs.");
    }
}
```

b) Barishal University expects to keep a database of its staffs. The database is separated into several classes whose hierarchical connections are shown in following figure. The figure also shows the minimum details required for each class. Specify all classes and define functions to create the database and retrieve individual information as and when required.




```java
// Base class
class Staff {
    String code;
    String name;

    Staff(String code, String name) {
        this.code = code;
        this.name = name;
    }

    void display() {
        System.out.println("Code: " + code);
        System.out.println("Name: " + name);
    }
}

// Derived classes
class Teacher extends Staff {
    String subject;
    String publication;

    Teacher(String code, String name, String subject, String publication) {
        super(code, name);
        this.subject = subject;
        this.publication = publication;
    }
```

```
    void display() {
        super.display();
        System.out.println("Subject: " + subject);
        System.out.println("Publication: " + publication);
    }
}

class Officer extends Staff {
    String grade;

    Officer(String code, String name, String grade) {
        super(code, name);
        this.grade = grade;
    }

    void display() {
        super.display();
        System.out.println("Grade: " + grade);
    }
}

class Typist extends Staff {
    int speed;

    Typist(String code, String name, int speed) {
        super(code, name);
        this.speed = speed;
    }
```

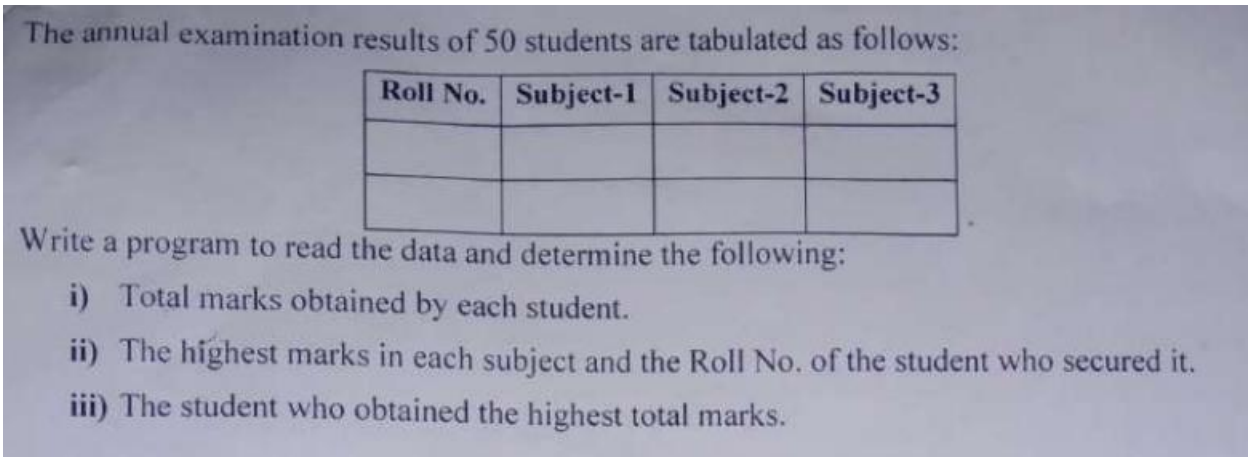
```
    void display() {
        super.display();
        System.out.println("Typing Speed: " + speed + " wpm");
    }
}

// Subclasses of Typist
class RegularTypist extends Typist {
    RegularTypist(String code, String name, int speed) {
        super(code, name, speed);
    }

    void display() {
        super.display();
        System.out.println("Typist Type: Regular");
    }
}
```

```
class CasualTypist extends Typist {
    double dailyWages;

    CasualTypist(String code, String name, int speed, double dailyWages) {
        super(code, name, speed);
        this.dailyWages = dailyWages;
    }

    void display() {
        super.display();
        System.out.println("Typist Type: Casual");
        System.out.println("Daily Wages: " + dailyWages);
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Teacher t = new Teacher("T101", "Dr. Alam", "Physics", "Quantum Mechanics");
        Officer o = new Officer("O202", "Mr. Karim", "Grade A");
        RegularTypist rt = new RegularTypist("R303", "Ms. Rina", 60);
        CasualTypist ct = new CasualTypist("C404", "Mr. Hasan", 50, 500.0);

        t.display();
        System.out.println();
        o.display();
        System.out.println();
        rt.display();
        System.out.println();
        ct.display();
    }
}
```

*****What is multilevel inheritance? A class named SuperClass has a method Display() [It prints the message "I am super class"]. Another class named ChildClass inherits SuperClass. Now, ChildClass should have the same Display() method but it should print the message "I am child class". You need to implement this scenario using OOP. You can choose your favorite language for implementation.**

```java
// SuperClass definition
class SuperClass {
    void Display() {
        System.out.println("I am super class");
    }
}

// ChildClass inherits SuperClass and overrides Display()
class ChildClass extends SuperClass {
    void Display() {
        System.out.println("I am child class");
    }
}

// Main class to test
public class Main {
    public static void main(String[] args) {
        SuperClass obj1 = new SuperClass();
        obj1.Display();  // Output: I am super class

        ChildClass obj2 = new ChildClass();
        obj2.Display();  // Output: I am child class
    }
}
```



The annual examination results of 50 students are tabulated as follows:

Roll No.	Subject-1	Subject-2	Subject-3

Write a program to read the data and determine the following:

i) Total marks obtained by each student.

ii) The highest marks in each subject and the Roll No. of the student who secured it.

iii) The student who obtained the highest total marks.

```java
import java.util.Scanner;

public class StudentResult {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int[] roll = new int[50];
        int[][] marks = new int[50][3];
        int[] total = new int[50];

        // Input Roll and Marks
        for (int i = 0; i < 50; i++) {
            System.out.print("Enter Roll No and 3 subject marks: ");
            roll[i] = sc.nextInt();
            for (int j = 0; j < 3; j++) {
                marks[i][j] = sc.nextInt();
            }
        }

        // i) Total marks of each student
        for (int i = 0; i < 50; i++) {
            total[i] = marks[i][0] + marks[i][1] + marks[i][2];
            System.out.println("Roll: " + roll[i] + ", Total = " + total[i]);
        }

        // ii) Highest in each subject
        for (int j = 0; j < 3; j++) {
            int max = marks[0][j];
            int topRoll = roll[0];
            for (int i = 1; i < 50; i++) {
                if (marks[i][j] > max) {
                    max = marks[i][j];
                    topRoll = roll[i];
                }
            }
            System.out.println("Subject " + (j + 1) + " Highest = " + max + " by Roll " + topRoll);
        }

        // iii) Highest total marks
        int maxTotal = total[0], top = roll[0];
        for (int i = 1; i < 50; i++) {
            if (total[i] > maxTotal) {
                maxTotal = total[i];
                top = roll[i];
            }
        }
        System.out.println("Highest Total: " + maxTotal + " by Roll " + top);
    }
}
```

☑ **Explain the Principles of Object-Oriented Programming (OOP) with Real-Life**

🔹 1. Encapsulation

Definition:

Encapsulation is the process of **wrapping data (variables)** and **code (methods)** into a single unit (class).

It protects data by making it private and accessible only through public methods.

Real-life Example:

A **bank account**: Balance is private; only deposit and withdraw methods can access it.

Code Example:

java Copy Edit

```java
class BankAccount {
    private double balance;

    public void deposit(double amount) {
        balance += amount;
    }

    public double getBalance() {
        return balance;
    }
}
```

🔹 2. Inheritance

Definition:

Inheritance allows a class (**child**) to acquire properties and behavior from another class (**parent**).

It supports **code reuse**.

Real-life Example:

A **Car** is a type of **Vehicle**. So Car inherits features from Vehicle.

Code Example:

java Copy Edit

```java
class Vehicle {
    void start() {
        System.out.println("Vehicle started");
    }
}

class Car extends Vehicle {
    void drive() {
        System.out.println("Car is driving");
    }
}
```

🔹 3. Polymorphism

Definition:

Polymorphism means **one interface, many forms**.

It allows methods to behave differently based on the object that calls them.

Types:

- **Compile-time (Method Overloading)**
- **Runtime (Method Overriding)**

Real-life Example:

A person can be a **teacher**, **father**, or **musician** — same person, different roles.

Code Example (Overriding):

java Copy Edit

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

class Dog extends Animal {
    void sound() {
        System.out.println("Dog barks");
    }
}
```

🔹 4. Abstraction

Definition:

Abstraction means hiding **complex details** and showing only **essential features**.

It reduces complexity.

Real-life Example:

You **drive a car** without knowing how the engine works — that's abstraction.

java

```java
abstract class Shape {
    abstract void draw();
}

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