1. Demonstrate how a child class can access a protected member of its parent class within the same package. Explain with example what happens when the child class is in a different package.

Accessing Protected Members in Java

In Java, a protected member of a class can be accessed within the same package and in subclasses (child classes) even if they are in different packages but only through inheritance.

Case 1: Child class is in the Same Package

When the parent class and child class are in the same package, the child class can directly access the protected member.

Example

```
// Parent Class
package pack1;
public class parent {
    protected int num = 10;
  }
// Child Class
package pack1;
public class child extends parent {
        void display () {
            System.out.println ("Number: " + num);}
    }
```

The variable number is declared as protected in the parent class. Since child is

in the same package, it can access number directly. No object creation of Parent is required. The program runs successfully and prints the value.

Case 2 : Child Class is in a Different Package

When the child class is in a different package, it can still access the protected member only through inheritance, not thrugh a parent class object.

Example

```
//Parent class,
package pack1;
public class Parent {
    protected int num = 20; }

// Child class
package pack 2;
```

```
import pack1.Parent;
public class Child extends Parent {
    void display( ) {
    System.out.println ("Number "+ num);}
}
```

2. Compare abstract classes and interfaces in terms of multiple inheritance. When would you prefer to use an abstract class and when an interface?

Multiple inheritance means a class inheriting features from more than one parent class.

Java does not support multiple inheritance using classes to avoid ambiguity, but it supports it using interfaces.

Abstract Class & Multiple Inheritance:

A class can not extend more than one abstract class. Therefore, abstract classes do not support multiple inheritance.

Example-

```
abstract class A {
    abstract void show(); }
abstract class B {
    abstract void display(); }
//Not allowed x
Class C extends A, B { }
```

Java allows only one class to be extended. This avoids confusion when two parent classes have methods with the same name.

Interface & Multiple Inheritance

A class can implement multiple interfaces. Therefore, interfaces support multiple inheritance.

Example –

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

interface B{
    void display();}

Class C implements A, B{

    public void show(){
        System.out.println("show");}

    public void display(){
        System.out.println("Display");}
    }
```

Interfaces only contain method declarations. Since there is no implementation conflict,

Java allows multiple inheritance using interfaces.

Differences:

Feature	Abstract Class	Interface
Multiple inheritance	not supported	supported
Number of parent types	only one	multiple
Method types	Abstract + non-abstract	Abstract methods
Variables	Can have instance variables	only public static final constants
Constructors	Allowed	Not allowed

When to use an Abstract Class

i. Classes are closely related

ii. You want to share a common code

iii. You need instance variables

iv. Partial ~~imp~~ implementation is required.

Example –

~~Vichel~~ Vehicle → Car. Bike

Common methods like start() can be implemented once.

When to use an Interface

i. Multiple inheritance is needed

ii. Classes are not closely related

iii. You want to define common behavior

iv. You want to achieve 100% abstraction.

Example —

Runnable, Serializable

Different classes implementing the same behavior.

3. How does encapsulation ensure data integrity and security?

Encapsulation is an oop concept where - data and methods are wrapped together in a class, Class data is hidden using access modifier (like private), access is provided only through controlled methods and helps in data security and data integrity.

How Encapsulation Ensures Data Security

1. Data Hiding : Variables are declared as private. They can not be accessed directly from outside the class.

2. Controlled Access : Data can only be modified using public methods. Prevents ~~autho~~ unauthorized or accidental changes.

3. Prevents Misuse : Invalid values (null, negative, empty) can be blocked. Ensures safe use of data.

How Encapsulation Ensures Data Integrity

1. Validation Logic : Setter methods check input values before assigning.

ii. Consistent State: Object always remains in a valid and meaningful state.

iii. Error Prevention: Prevents invalid balance on account number.

~~4. How does encapsulation ensure ?~~

3. Show with a BankAccount class using private variables and validated methods such as setAccountNumber(String), setInitialBalance(double) that rejects null, negative or empty values.

```
public class BankAccount {

    //private data members
    private String accountNumber;
    private double balance;

    //Method to set account number with validation
    public void setAccountNumber(String accNo){
```

```
if (accNo == null || accNo.isEmpty()){
    System.out.println("Invalid!"); }
else{
    this.accountNumber = accNo ;}
}

//Method to set initial balance with validation
public void setInitialBalance(double amount){
    if (amount < 0) {
System.out.println ("Negative!");}
    else {
        this.balance = amount;}
    }
//Getter methods
public String getAccountNumber( ) {
```

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

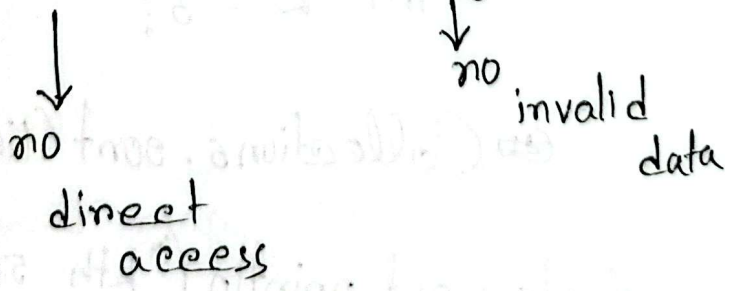
Explanation:

accountNumber and balance are private, so they can not be accessed directly.

setAccountNumber() rejects null, empty string.

setInitialBalance() rejects negative values, only valid data is stored in the object.

This ensures security and integrity.



4. Write program to—

I. find the kth smallest element in an Array List.

```
import java.util.*;
public class kthSmallest {
    public static void main (String[] args){
        ArrayList<Integer> list = new ArrayList<>();
        list.add(5);
        list.add(2);
        list.add(8);
        list.add(1);
        list.add(4);
        int k = 3;
        Collections.sort(list);
    System.out.println("kth smallest = "+ list.get(k-1));
    }
}
```

An ArrayList is first sorted using Collections.sort(). After sorting, the element at index k-1 represents the kth smallest value. This approach is simple and efficient for small to medium datasets.

ii. Create a TreeMap to store the mappings of words to their frequencies in a given text.

```
import java.util.*;
public class WordFrequency{
  public static void main(String[] args){
    String text = "java is easy and java is powerful";
    String[] words = text.split(" ");
  TreeMap<String, Integer> map = new TreeMap<>();
  for(String word : words){
     map.put(word, map.getOrDefault(word, 0)+1);}
   System.out.println(map);}
  }
```

The input text is split into words and stored in a TreeMap. TreeMap keeps the keys in sorted order automatically. Each word's frequency is updated whenever it appears again.

To implement a Queue and Stack using the PriorityQueue class, we'll create custom comparators.

Implement your code:

```
class StackUsingPQ {
    static class Element {
        int value, priority;
        Element(int v, int p) {
            value = v;
            priority = p;
        }
    }
}
```

The input text is split into words and stored in a TreeMap. TreeMap keeps the keys in sorted order automatically. Each word's frequency is updated whenever it appears again.

iii. Implement a Queue and Stack using the Priority Queue class with a custom comparator.

```
import java.util.*;
class StackUsingPQ {
    static class Element {
        int value, priority;
        Eelement (int v, int p) {
            value = v;
            priority = p;
        }
    }
```

```
    static int count = 0;
public static void main (String[] args){
        PriorityQueue<Element> stack =
new PriorityQueue<>((a, b) -> b.priority - a.priority);
        stack.add (new Element (10, count++));
        stack.add (new Element (20, count++));
        stack.add (new Element(30, count++));

    System.out.println (stack.poll().value);

    }
}
```

Priority Queue normally follows priority order, but using a custom comparator, it can simulate Stack and Queue behavior.

Higher priority gives LIFO (stack) behavior, while lower priority gives FIFO (Queue) behavior.

IV. Create a TreeMap to store the mappings of student IDs ~~an~~ to their details.

```
import java.util.*;
public class StudentMap{
    public static void main(String[] args) {
        TreeMap<Integer, String> students = new
                        TreeMap<>();
        students.put(101, "Rahim");
        students.put(102, "Karim");
        students.put(103, "Sadia");
    System.out.println(students);}
    }
```

A TreeMap is used to store student IDs as keys and their details as values. It automatically maintains the data ascending order of student IDs. This helps in organized and stored data storage.

V. Check if two LinkedLists are equal

```
import java.util.*;
public class CompareLists {
  public static void main(String[] args) {
    LinkedList<Integer> list1 = new LinkedList<>();
    LinkedList<Integer> list2 = new LinkedList<>();
    list1.add(1);
    list2.add(2);
    list2.add(1);
    list2.add(2);
    if (list1.equals(list2)){
        System.out.println("Lists are equal");}
    else{
        System.out.println("Lis not equal");}}}
```

The equals() method of LinkedList compare both size and elements in order. If both lists contain the same elements in the same sequence, it returns true.

vi. Create a HashMap to store the mappings of employee IDs to their departments.

```
import java.util.*;
public class EmployeeMap {
  public static void main (String[] args) {
    HashMap<Integer, String> empMap = new HashMap<>();
    empMap.put(1, "HR");
    empMap.put(2, "IT");
    empMap.put(3, "Finance");
    System.out.println(empMap);}
  }
```

A HashMap stores data in key-value pairs with fast access time. Employee IDs are used as keys and department names as values. It allows efficient insertion, deletion and retrieval of data.

5. Developing a multithreading based project to simulate a car parking management system with classes namely - RegisteranParking - Represents a parking request made by a car; ParkingPool - Acts as a shared synchronized queue; ParkingAgent Represents a thread that continuously checks the pool and parks cars from the ~~que~~ queue and a MainClass - simulates N cars arriving concurrently to request parking.

Car ABC123 requested parking.

Car XYZ456 requested parking.

Agent 1 parked car ABC123.

Agent 2 parked car XYZ456.

Ans:

RegistrarParking Class

```
public class RegistrarParking {
    private String carNumber;
    public RegistrarParking(String carNumber){
        this.carNumber= carNumber;
System.out.println("Car " + carNumber + " requested
        parking.");
    }
public String getCarNumber(){
        return carNumber;}
    }
```

ParkingPool class

```
import java.util.*;
public class ParkingPool {
    private Queue<RegistrarParking> queue = new
            LinkedList<>();
    public synchronized void addCar(RegistrarParking
            car) {
        queue.add(car);
        notify(); }
    public synchronized RegistrarParking getCar() {
        while (queue.isEmpty()) {
            try {
                wait(); }
            catch (InterruptedException e) {
                e.printStackTrace(); }
        }
        return queue.poll();
    }
}
```

ParkingAgent Class

```
public class ParkingAgent extends Thread {
    private ParkingPool pool;
    private String agentName;
public ParkingAgent (ParkingPool pool, String name){
    this.pool = pool;
    this.agentName = name; }
public void run() {
    while (true){
        RegistrarParking car = pool.getCar();
System.out.println (agentName + "parked car "+
car.getCarNumber() + ".");
    try {
        thread.sleep(500);}
    catch (InterruptedException e){
        e.printStackTrace();}
}}
```

Main Class

```
public class Main Class {
public static void main (String[] args) {
    ParkingPool pool = new ParkingPool ();
ParkingAgent agent1 = new ParkingAgent (pool,
        "Agent 1 ");
ParkingAgent agent2 = new ParkingAgent (pool,
            "Agent 2 ");
agent1.start();
agent2.start();
new.RegistranParking ("ABC123");
pool.addCar (new RegistrarParking ("ABC123"));
new RegistranParking ("XYZ456");
pool.addCar (new RegistranParking ("XYZ456"));
  }
}
```

RegistrarParking - Represents a parking request made by a car. It stores the car number

ParkingPool - Acts as a shared resource (queue). It is synchronized to avoid race conditions when multiple threads access it.

Parking_Agent - Represents a thread that continously checks the parking pool and parks cars from the queue.

Main Class - Simulates multiple cars arriving at the same time and multiple agents handling them.

Cars arrive concurrently and place parking requests into a shared parking pool. Parking agents run as ~~spe~~ separate threads and take cars from the pool one by one. Synchronization ensures that only one agent parks a car at a time.