1] Demonstrate how a child class can access a protected member of its parent class with the same package.

Accessing protected member in same package ~~parent.java~~

Parent.Java

```
Package.pack1;
public class Parent {
    protected String message = "Hello from P";
```

Child.Java

```
Package.Pack1;
Public class child extends parent {
    Public void showMessage() {
        System.out.println("child accessed." + message);
    }
    public static void main(String[] args) {
        child c = new child();
        c.showMessage();
    }
}
```

Since both classes in the same package the protected member message is accessable in child class.

Proted in different classes:

```
Package Pack1;
Public class Parent {
    protected String message = "Hello";
}
Package Pack2;
import Pack1.Parent;
Public class child extend Porent {
Public void showMessage() {
    System.out.println("Accen from child: " + message);
  }
public static void main (String[] args) {
Child c = new child();
c.showMessage(); } }
```

A subclass in a different ~~packge~~ package can accen protected member of the porent class only through inheritance, not through ~~the~~ object of the porent.

2] Compare abstract classes and interfaces in terms of multiple inheritance.

Feature	Abstract class	Interface
multiple inheritance	Not supported	Fully supported
Extends	ClassA extends classB	ClassA implements X, Y, Z
code reuse	Can have method bodies and member variable.	Java 8+ can be default and static methods.
state	Can have instance variable	Only constants (Public static final)
constructor	Yes (Can initialize field)	NO constructor allowed

when to use an abstract class:-

i) You want to provide base functionality and shared states.

ii) You need constructor or non-static instance variables.

iii) You expected closely related classes with an 'is-a' relationship.

when to use interface:-

i) You want to define pure behaviour, not implementation.

ii) You want to use multiple interface of type.

iii) Classes are unrelated but share common capabilities.

350

3] How does encapsulation ensure data security and integrity? show with a bank account class using private variable and validated methods such as setAccountNumber(string) etc.

Encapsulation is a key principle in object oriented programming that hide internal data. It helps data security, data integrity, maintainability.

```
Public class BankAccount {
    private String accountNumber;
    private double balance;
Public void setAccountNumber(String AccountNumber) {
if (accountNumber == null || accountNumber.trim().isEmpty()) {
    throw new IllegalArgumentException("Can't be null");
}
this.accountNumber = accountNumber;
}
Public void setInitialBalance(double balance) {
    if (balance < 0) {
    throw new IllegalArgumentException("Balance can't be
                                        negative");
}
this.balance = balance;
}
```

```
Public String getAccountNumber(){
    return accountNumber;
}
Public double getBalance(){
    return balance;
}
Public void deposit(double amount){
    if(amount>0) this.balance += amount;
}
}
```

4/ i) find kth smallest element

```
import java.util.*;
Public class kthSmallest{
Public static void main(String[] args){
List<Integer> list = Arrays.asList(8,2,5,1,9,4);
Collection.sort(list);
int k = 3;
System.out.println(k+"th smallest "+ list.get(k-1));
}
}
```

output:

3th smallest 4

ii) word frequency using treemap

```
import java.util.*;
public class wordfreq {
    public static void main(String[] args) {
    String text = "apple banana apple mango";
Treemap<String, Integer> map = new TreeMap<>();
for (String word : text.split(" "))
  map.put(word, map.getOrDefault(word, 0)+1);
~~map.put(word~~
  map.forEach((k,v) → System.out.print(k + "=" + v));
  }
  }
```

output:

apple = 2
banana = 1
mango = 1

iii) Queue & stack using Priority Queue

```
import java.util.*;
public ~~class~~ class pQStackQueue {
    static class Element {
    int val, order;
    Element(int v, int o) {val = v; order = o;}
  }
```

```
Public static void main(String[] args) {
 priority Queue <Element> stack = new priority Queue<>
 ((a, b) → b.order - a.order);
 priority Queue <Element> queue = new Priority Queue<>
(Comparator.comparing Int(a→a.order));
 int order = 0;
  Stack.add(new Element(10, order++));
  Stack.add (new Element (20, order++));
System.out.println("Stack pop:" + stack.poll().val);
  queue.add (new Element (100, 0));
  queue.add (new Element (200, 1));
System.out.println("Queue poll:" + queue.poll().val);
 }
}
```

5| Multithread based project:

```
import java.util.*;
import java.util.concurrent.*;
 Class ParkingPool{
    private tainal Queue <String> queue = new Linked List<>();
public synchronized void addCar(String car){
  queue.add (Car);
System.out.println("Car" + car + " requested parking.");
```

```
notify();
}
public synchronized String getCar() {
    while (queue.isEmpty()) {
try {
    wait;
}
catch (InterruptedException e) { }
return queue.poll();
}
}

Class RegisterParking extends Thread {
  private final String CarNumber;
  private final ParkingPool pool;

public RegisterParking (String CarNumber, ParkingPool pool) {
 this.carNumber = CarNumber;
 this.pool = pool;
 }

public void run() {
    pool.addCar(CarNumber);
  }
}

Class ParkingAgent extends Thread {
  private final ParkingPool pool;
  private final int agentId;
```

```
public ParkingAgent(ParkingPool pool, int agentId) {
    this.pool = pool;
    this.agentId = agentId;
~~public ParkingAgent(Parking po~~
}
public void run() {
    while(true) {
        String car = pool.getCar();
System.out.println("Agent" + agentId + "Parked car" + car + ".");
try {
    Thread.sleep(500);
    }
catch (InterruptedException e) { }
}
}
}
public class CarParkingSystem {
    public static void main(String[] arg) {
    ParkingPool pool = new ParkingPool();
    new ParkingAgent(pool, 1).start();
    new ParkingAgent(pool, 2).start();
    new RegistrarParking("ABC123", pool).start();
    new RegistrarParking("XYZ456", pool).start();
    }
}
```

Output:

Car ABC123 requested Parking

Car XYZ456 requested Parking

Agent 1 Parked Car ABC123

Agent 2 Parked Car XYZ456.

6| Comparison between Dom Vs SAX

Feature	DOM	SAX
Memory	High (loads whole XML)	Low (reads line by line)
Speed	Slower for big files	Faster for large file
Navigation	Easy (tree structure)	Hard
modification	Yes	No
Best for	Small XML, editing	Large XML

7| How ~~does~~ does the Virtual DOM in React ~~imp~~ improve Performance?

React creates a Virtual copy of Dom

On update, React:

1) Compares (diffs) old and new Virtual Dom

2) Finds what changed

3) Applies only the changes to real Dom.

8) Event delegation in JavaScript:

A technique where a sigle event Listener is attached to a porent element to handels events from current and future child elements

```
<ul id="menu">
<li> Home </li>
<li> About </li>
</ul>
```

```
document.getElementById("menu").addEvenListener(
"click", function(e){
  if(e.target.tagName === "LI"){
    alert("Clicked: " + e.target.innerText);
  }
});
```

9) Java Regular Expressions for input validation:

Java provides the pattern and Matcher classes from the java.util.regex package to use Regular Expression (Regex) for validating input formates like email, phone number, etc.

```
import java.util.regex.*;
public class EmailValidation{
    public static void main (String[ ] args) {
        String email = "test.user@example.com";
        String regex = "^[A-Za-z0-9+_.-]+@[A-Za-z0-9.
            -]+\\.[A-Za-z]{2,6}$";
Pattern pattern = Pattern.compile(regex);
Matcher matcher = pattern.matcher(email);
 if(matcher.matches()){
    System.out.println("valid email.");
  }
  else {
    System.out.println("Invalid email.");
    }
  }
}
```

Regex Explanation:

⇒ ^[A-Za-z0-9+_.-]+ → Username part

⇒ @ → At symbol

⇒ [A-Za-z0-9.-]+ → Domain name

⇒ \\. → Dot before domain extension

⇒ [A-Za-z]{2,6}$ → Domain extension like .com, .org

This checks that the email format is valid using regex pattern matching.

10. Custom Annotations in Java (With Reflection)

Custom annotations are user-defined metadata created using @interface. They can be processed at runtime using reflection to influence program behavior.

```
import java.lang.annotation.*;
public @interface RunMe {
    String value() default "Running";
}
public class MyService {
    public void taskOne() {
        System.out.println("Task one Executed");
    }
}

import java.lang.reflect.*;
public class AnnotationProcessor {
    public static void main(String[] args) throws Exception
    {
        MyService service = new MyService();
    for(Method method : service.getClass().getDeclaredMethods()) {
        if(method.isAnnotationPresent(RunMe.class)) {
            RunMe ann = method.getAnnotation(RunMe.class);
            System.out.println("Message: " + ann.value());
            method.invoke(service);
        }
    }
}
```

Output:

Message : start task
task one Executed

Custom annotations with reflection let you define and control behaviors dynamically at runtime based on metadata.

12/ JDBC Communication with a Relational Database:

JDBC (Java Database Connectivity) is an API that allows Java applications to interact with relational databases like MySQL, Oracle, PostgreSQL, etc. It provides methods to connect, send SQL queries, and retrieve results.

- Steps to execute a SELECT Query:
  1. Load the JDBC Driver
  2. Establish Connection to the database
  3. Create Statement or prepared Statement
  4. Execute SELECT query
  5. Process ResultSet
  6. Close resources

```
import java.sql.*;

Public class JDBCSelectExample {
    public static void main(String[] args) {

        Connection conn = null;
        Statement stmt = null;
        ResultSet rs = null;

try {

    Class.forName("com.mysql.cj.jdbc.Driver");
    conn = DriverManager.getConnection(
     "jdbc:mysql://localhost:3306/mydb", "username", "password");
     stmt = conn.createStatement();
     rs = stmt.executeQuery("SELECT id, name FROM
                                    students");

  while(rs.next()){
     int id = rs.getInt("id");
     String name = rs.getString("name");
     System.out.println(id + ": " + name);
  }
} catch(Exception e) {
        e.printStackTrace();
    }
 finally {
        try{ if(rs != null) rs.close(); } catch(Exception e){ }
        try { if(stmt != null) stmt.close(); } catch(Exception e){ }
        try { if(conn != null) conn.close(); } catch(Exception e){
    }

}}
```