1] Demostrate how a child class can access a Protected member of its parent class within the same package.

=>

Accessing protected member in same package

Parent.java.

```
Package. Jack1 ;
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.

Protected in different classes:

```
Package pack1;
public class parent {
    protected string message = "Hello";
}
Package pack2;
import pack1.Parent;
public class child extends parent {
public void showmessage() {
public void sho
System.out.println("Access from child:" + message);
}
public static void main(string [] args) {
child c = new child();
c.showmessage(); } }
```

A sub class in a different package can access protected member of the parent class only through inheritance, not through the object of the parent.

Lab-1

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

⇒

Feature	Abstract class	Interface
Multiple inheritance	Not supported	Fully supported
extends	Class A extends class B	Class A implements X,Y,Z
Code reuse	Can have method bodies and member variable.	Java 8+ can have defaul 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 inheritance of type.

III) Classes are unrelated but share common capabilities.

3] How does encapsulation ensure data security and integrity ? Show with an bank account class using private variables 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 + "rd smallest " + list.get (k-1));
}
}
```

Output : 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.Tree map<>();
```

```
for (String word : text.split(" "))
    map.put(word, map.getOrDefault(word, 0) + 1);
map.forEach((k, v) -> System.out.print(k + "=" + v));
}
}
```

Output: apple = 2
banana = 1
mango = 1

iii) Queue & Stack using PriorityQueue.

⇒

```
import java.util.*;
public class PQStackQueue {
    static class Element {
    int val, order;
    Element(int v, int o) { val = v; order = o; }
}
blic static void main(String[] args)
```

```
PriorityQueue <Element> Stack = new priorityQueue <>
((a,b) -> b.order - a.order);
PriorityQueue <Element> Queue = new PriorityQueue <>
(Comparator.comparingInt (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);
}
}
```

Lab - 3

5] Multithread based project:

```
=> import java.util.*;
import java.util.concurrent.*;
Class ParkingPool {
   private final Queues <String> queue = new LinkedList <> ();

public syncronized void addcar (string car) {
queue.add (car);
System.out.println ("car"+ car + " requested parking.
notify ();
}
public Syncronized String getcar () {
While (queue.isEmpty()) {
try {
wait ();
}
catch (Interrupted Exception 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 agent. Id){
this. Pool = Pool;
this. agent Id = agentId;
}
```

```
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[] args) {
ParkingPool pool = new ParkingPool();
new ParkingPoolAgent(Pool, 1).start();
new ParkingAgent (Pool, 2).start();
new RegisterParking ("ABC123", Pool).start();
new RegisterParking ("XYZ456", Pool).start();
 }
}
```

Output:

Car AB123 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 fit
Navigation	Easy (tree structure)	Hard
motification	Yes	No
Best for	Small xml, editing	Large xml

7] How does the virtual Dom in React improve performance?

⇒ React creats a virtual cop of Dom.
On update React :

1) Compares (diffs) old and new virtual Dom.
2) Finds whats changed.
3) Applies only the changes to real Dom.

8] Event delegation in JavaScript.

⇒ A technique where a single event listener is attached to a parent element to handles events from current and future child elements.

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

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

9]

In Java, Regular Expression (regex) are used to validate, search or extract patterns from string, such as validating emails, Passwords, Phone numbers etc.

Java provide two core classes for reg regex.

- Java.util.regex.Pattern: Compiles the regex pattern.
- Java.util.regex.Matcher: Applies the pattern to input string.

Email validation Regex:

```
String regex = "^[A-Za-z0-9+_.-]+@[A-Za-z0-9.-]+$";
```

This pattern checks:

username : letter . digits , + , - , _ , .

@ symbol

Domain name - letters, digits, dots, hyphen

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

Pattern.compile() → compiles the regex
matcher() → applies the pattern to the input string.
matcher() → checks if the input fully matches the pattern.

10)
Custom annotations are user defined metadata that can be added to code elements (classes methods etc) and processed at runtime using reflection.

Steps to use custom Annotation with Reflection:

i) Define the annotation using @interface
ii) Annotate elements with it
iii) Use reflection to read and process the annotation at runtime.

Let's build a custom annotation @RunImmediately that marks method to be invoked automatically at runtime.

Define the annotation:

```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME) // keeps annotation
                                    info during runtime.
@Target(ElementType.METHOD) // can be applied to
                            methods only.
Public @Interface RunImmediately {
  int times() default 1;
}
```

times(): Optinal attribute to control how many times to run the method.

Use Annotation in a class:

```
Public class myService {
   @RunImmediately(times=3)
   Public void sayHello() {
       System.out.println("Hello");
   }
}
```

Use reflection to process Annotation at RunTime

```
import java.lang.reflect.method;
Public class AnnotationProcessor {
    public static void main (String [] args) throws Exe{
    myService service = new myService();
    method [] methods = MyService.class.getDeclared
                                        methods ();
    for (method method : methods) {
    if (method.isAnnotationPresent (Run -- class) {
    Run ... annotation = method.getAnno -- (
    for (int i=0; i<annotation times (); i++) {
    method.invoke (Service);
        }
    }
}
}
```

11] Discuss the sigletion design pattern in java.

=>

The singleton pattern ensures that only one instance of a class exists and provides a global access point to it.

it solves :-

① Prevents multiple instance.

② Controls resource usage and ensures consistency.

Basic Singleton implementation

```
public class Singleton {
    private static Singleton instance;
    private Singleton () { }
  public static Singleton getInstance () {
    if (instance == null) {
    instance = new Singleton ();
    }
return instance;
  }
}
```

Thread - Safe Singleton:

```
public class Singleton {
private static volatite singleton instance;
private Singleton() { }
public static Singleton getInstance() {
 if (instance == null) {
    Synchronized (Singleton.class) {
 if (instance == null) {
    instance = new Singleton();
    }
  }
}
return instance.
  }
}
```

Lab 4

12] Describe how JDBC ~~mangg~~ manages communication between a Java application and a relational database.

⇒

JDBC (Java Database connectivity) is an API, that allows Java applications to connect to and interact with relational database like mySQL, PostgreSQL, or oracle.

Steps to Execute a SELECT Query:

```
import java.sql.*;
public class DBExample {
public static void main (String [] args) {
String url = "Jdbc:mysql://localhost:3306/mydb";
String user = "root";
String pass = "password";
Connection conn = null;
Statement stml = null;
Resultset rs = null;
```

```
try {
Conn = Drivermanager.getconnection(url, user, Pass);
 Stmt = Conn.createStatement();
 rs = Stmt.executeQuery("Select * from employee");
while (rs.next()) {
 System.out.println("Name: "+rs.getString("Name"
 }
 catch (SQLExecption e) {
 System.out.println("Error;" + e.getmessage());
}
Finally {
try {
if (rs! = null) rs.close();
if (Stmt != null) stmt.close();
if (Conn != null) conn.close();
}
catch (SQLException ex) {
  ex.printStackTrace();
 }
```

13] How do servelets and JSPs work together in a web application following the mvc (model-view-Controller) architecture?

⇒ Servelet acts as the controller. JSP is the view and Java class is the model in a Java EE web application using the mvc Architecture.

How they work together:

(i) Client sends an HTTP request to the Servlet.

(ii) Servlet processes the request:

(iii) JSP retrives the data and renders the HTML response to the client.

Sample use case:

Displaying user Profile.

model · User · Java :

```
public class User {
    private String name;
    private String email;
public User (String name, String email) {
    this.name = name;
    this.email = email;
  }
public String getName () {
return name;
}.
public String getEmail () {
return email;
  }
}
```

Controller : UserServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class UserServlet extends HttpServelet {
protected void doGet (HttpServletRequest request,
        HttpServletResponse response)
throws ServletException, IOException {
User user = new user ("Tahmina Nipa", "Tah@example
                                        .com");
request.setAttribute ("User", user);
RequestDispatcher dispatcher = request.getRequestDispatcher
                                (profile.jsp);
    dispatcher.forward (request, response);
    }
}
```

profile.jsp:

```
<%@ page import = "yourpackage.User" %>
<%
   User user = (User) request.getAttribute ("User")
%>.
<Html>
<Lead><fittle>User profile </fittle></Lead>
<body>
<L2> User Profile </L2>
<P> Name ; <%= User.getName () %></P>
<P> Email ; <%= User.getEmail () %></P>
</body>
</Html>
```

14] Explain the life cycle of a java servlet.

⇒

The life cycle of a java servlet is managed by the Servlet Container. It involves three main stages, handles by three important methods:

1. init() method

- The init() method is called once when the servlet is first baded into memory.
- It is used to initialized resources, such as database connections or configure settings.

2. Service() method

- The service() method is called each time a Client makes a request.
- It determines the Http request type (Get, POST etc) and calls the appropriate method.

- This is the where the main request handing logic goes

3. Destroy () method

- The destroy() method is called once when the servlet is being removed from memory.
- It is used to release resources, such as closing database connections.

Handling Concurrent Requests:

- A servlet is single instance and multithreaded.
- This means the servlet container creates only one instance of the servlet, but can handle multiple requests at the same time using different threads.

Thread Safety Issues:

Since multiple threads share the same servlet instance, shared variables can lead to race conditions.

For example:

```
private int counter = 0;
protected void doget (---) {
counter ++;
}
```

15] A single instance of a servlet handles multiple requests using threads. What problems can occure if shared resources are accessed by multiple threads

⇒ In a servlet, the container creates only one instance, and multiple requests are handled using seperate threads. If shared resources are

accessed or modified by these threads simultaneously, it can lead to thread safety issues like:

- Race conditions
- Incorrect data
- Unpredictable behavior

Example: Suppose we have a servlet that counts how many users visited a page.

```
public class CounterServlet extends HttpServlet {
private int counter=0;
protected void doGet (HttpServletRequest req,
                      HttpServletResponse res)
throws ServletException, IOException {
    counter++;
res.getWriter().println("Visitor number:" +
                        counter);
}
}
```

Solution using synchronization:

```
public class CounterServlet extends HttpServlet {
  private int counter = 0;
protected void doGet (HTTPServletRequest req,
                      HTTPServletResponse res)
throws ServletException, IOException {
    Synchronized (this) {
      counter++;
res.getWriter().println("visitor number:" +
                        counter);
    }
  }
}
```

The synchronization block ensures that only one thread can access the counter at a time.

This prevents race conditions and ensure data consistency.

16] Describe how the mvc pattern separates concern in a java web application. Explain the advantages of this structure in terms of maintainability and scalabil

=>

The Model - View - Controller pattern is a design architecture that separates concerns in a Java web application. It divides the application into three components.

① Model (M):

- Represents the data and business Logic.
- Contains Java classes.
- Handles database interactions, validations and compute

② View (V):

- Represents the presentation layer.
- Implemented Using JSPs, HTML, CSS
- Displays data received from the controller but doesn't handle logic.

③ Controller (C):

- Acts as the intermediary between model and view.
- Implemented Using Servlets.
- Handles requests, processes input, updates model and selects the appropriate view.

Example: Student Registration System

- View (JSP): Register.jsp - displays a registration form to the user.
- Controller (Servlet): RegisterServlet.java - receives the form data, calls the model, and forwards to use a success/failure page.
- Model (Java class): Student.java, StudentDAO.java - Handles business logic and stores student data into the database.

Advantages:

(i) **Separation of concerns:** Each component has a distinct role, making the codebase clean and organized.

(ii) **Maintainability:** Changes in UI don't affect business logic and vice versa.

(iii) **Reusability:** Same model logic can be reused in different views.

(iv) **Scalability:** Easier to scale and add new features.

(v) **Testability:** Logic is isolated in the model, which makes unit testing easies.