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);
}
});
```

2]

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 Single to n 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 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.get Email () %X/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 from 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.

Lab - 5

17] In a Java EE application, how does a servlet Controller manage the flow between the model and the views?

⇒

How Servlet Controller Manages Flow (MVC):-

In a Java EE web app, the Servlet acts as a Controller.

It:

- Takes user input (request)
- Uses the model (Java class) to process data.
- Then forwards data to JSP (views) to show results.

Sample Example:

① Model (User.java)

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

```
public String getName () {
 return name;
 }
public String getEmail () {
 return email;
  }
}
```

ii) Controller (UserController.java)

```
@WebServlet ("/User")
public class UserController extends HttpServlet {
protected void doPost (HttpServletRequest req,
                       HttpServletResponse res)
throws ServletException, IOException {
 String name = req.getparameter ("name");
 String email = req.getparameter ("emil");

 User user = new User (name, emil);
 req.setAttribute ("User", users);
RequestDispatcher rd = req.getRequestDispatcher
                       ("UserInfo.JSP");
 rd.forward (req, res);
```

iii) View (UserInfo.jsp)

```
<%@ page import = "model.User" %>

<%
User user = (User) request.getAttribute("user")
%>
<h2> User Info </h2>
<p> Name : <%= user.getName() %> </p>
<p> Email : <%= user.getEmail() %> </p>
```

18

Aspect	Cookies	URL Rewriting	HttpSession
Whole data is stored	On client browser	In the URL as query parameters	On the server.
Visibility to user	Yes	Yes	No
Security level	Low	Low	High
Data limit	Limited (~4KB per cookie)	Limited (due to URL length)	Large (depends on server memory
Implementation Complexity	Medium	High	Low

Advantages, Limitations & Ideal use cases:

Cookies:

Advantages: Can persist data even after browsers is closed and Good for user preference.

Limitations: Can be disabled by user. Low security - data is visible & modifiable.

Ideal use cases: Remembering usernames as preference. Tracking returning users.

URL rewriting

Advantages: Works when cookies are disables. Easy to implement for small apps or links.

Limitations: Exposes data in URL (security risk) must rewrite every URL manually.

Ideal use cases: Temporary session tracking when cookies are turned off. Basic apps without login.

HttpSession:

Advantages: Secured and easy to use with built-in support in Java. Automatically expires after timeout.

Limitations: Uses server memory. Needs cleanup to avoid memory leaks in large apps.

Ideal use cases: Login systems, shopping carts, personal dashboards.

[2] A web application stores user login information using Httpsession. Explain how the session works across multiple requests and how session timeout or invalidation is handled securely.

⇒ In a web application, Httpsession is used to store user-specific data, such as login information, across multiple requests. When a user logs in, the server

creats a session objects. Using request.getSession() and stores attributes like username or userID in that session.

Each user is assigned a unique session ID, which is typically stored in a cookie (JSESSIONID) on the user's browser. This ID is sent automatically with each requests, allowing the server to recognize the user and retrive their session data.

- Working accross multiple Requests:

— When a user logs in, the server creats a sessi and gives a unique session ID. This session ID sent with every request. The server uses the I to find the session and show user-specific data

- Session Timeout and Invalidation:

— If user is inactive, the session ends automatically. The session can also be ended using session.invalidate(). After timeout or logout, the user must log in again.

- Security Handling:

- Users data is kept on the server, not on the browsers. Session ID should be sent over HTTPS to avoid hacking.

20] Explain how spring mvc handles an HTTP request from a browser.

=> @Controller: marks the class as a controller that handles web requests.

@RequestMapping: Maps a specific URL (like /login) to a method inside a controller.

model: A container used to pass data from the controller to the views.

In Spring mvc, when a browser sends an HTTP request, the framework follows the model-View-Controller (mvc) design pattern to handle and respond to it in a structured way.

Request Handling Flow:

(i) The request first goes to the DispatcherServlet (the front controller).

(ii) It looks for the correct method to handle the request using annotation like @RequestMapping.

(iii) The method is inside a class marked with @Controller, which acts as the controller in MVC.

(iv) Business logic is executed and necessary data is added to a model object.

(v) Finally the view is selected and returned with the model data for display.

Example:

(Login from submission)

1. User submits a form to /login.
2. Spring calls the controller method:

```
@Controller
public class LoginController {
  @RequestMapping ("/login")
public String login (@RequestParam String username,
                     Model model) {
if (username.equals("admin")) {
model.addAttribute ("message", "Login Successful");
else {
model.addAttribute ("message", "Login failed");
return "login";
}
}
```

21] Spring mvc uses the DispacherServlet as a front Controller. Describe its role in the request processing workflow. How does it interact with view resolvers and handlers mappings?

⇒

In spring mvc, the DispatcherServlet acts as the front Controller which means it is the central point for receiving all HTTP requests in a web application.

Role of DispatcherServlet in Request workflow:-

(i) Receives Request: The DispatcherServlet receives the request from the browser.

(ii) Finds Handler: It consults Handle mapping to find the correct @Controller method based on the URL.

(iii) Calls Controller: It calls the mapped method in the Controller to execute business logic.

(IV) Gets view Name: The controller method returns a logic view name.

(V) Resolves View: The dispatcherServlet uses a view Resolver to map the logical name to an actual view file.

(VI) Renders view: The view is rendered with the data from the model, and the final HTML is sent back to the browser.

How dispatchServlet Interacts with Handle mappings and View Resolves:

Handler mapping: DispatcherServlet uses it to find the matching @Controller method based on the request URL.

View Resolver: After the controller returns a view name, DispatcherServlet uses the view resolver to map that name to an actual view file.

Lab-6

22] How does prepared Statement improve performan and security over. statement in JDBC ? write a short example to insert a record int o mySQL table using prepared statement.

=>

- performance : Prepared Statement precompiles the SQL query once and reuses it with different parameters. This makes it faster than statement especially for repeated oprations.
- Security : It prevents SQL injection by safely binding users input as parameters instead of inserting raw strings into the query.

Example: (Insert record using prepared statement)

```
import java.sql.*;
public class InsertExample {
public static void main (String args[]){
try {
Connection con = DriverManager.getConnection
("Jdbc:mysql://localhost:3306/testdb", "root",
                                    password);
String query = "INSERT INTO students (name, email)
                                    VALUES(?,?)";
PreparedStatement pst = con.prepareStatement(query);
pst.setString(1, "Alice");
pst.setString(2, "alice@example.com");
int rows = pst.executeUpdate();
System.out.println(rows + "record inserted");
con.close();
}
catch (Exception e){
e.printStackTrace();
}
}
```

Lab-7

23] What is ResultSet in JDBC and how is it used to retrive data from a mySQL database?

⇒

In JDBC a ResultSet is an object that holds the result of a SQL SELECT query. It acts like a table in memory and allows you to retrive and process rows one by one.

To retrive data:-

(i) A SQL SELECT query is executed using statement or PreparedStatement.

(ii) The result is stored in a ResultSet Object.

(iii) The next() method is used to move through each row.

(iv) Methods like: getInt(), getString() etc are used to read column values from the current row.

Example:

```
ResultSet 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: " + id + ", Name: " + name);
}
```

24] How does JPA manage the mapping between Java objects and relational tables?

=>

JPA (Java Persistence API) is a standard for mapping Java objects to relational database tables. It simplifies database operations by allowing developers to work with Java objects instead of writing raw SQL queries.

Mapping works!

① @Entity: Marks a Java class as a database entity.

② @Id: Marks a field as the primary key.

③ @Generated value: Automatically generates the primary key value.

JPA (Java Presistance API) manages the mapping between Java objects and relational database tables using annotations. Each Java class is treated as Entity.

Example!

```
import Jakarta.persistence.*;
@Entity
Public class Student {
@Id
@Generated value (Strategy = GenerationType.IDENTITY)
Private Long id;
Private String name;
Private String email;
```

Here, the Student class will be mapped to a table named student. The id field is the primary key, and its value will be automatically generated. The name and email fields will be mapped to columns in the table.

- Advantages of JPA over JDBC :-

(I) Less code : JPA reduces boilerplate (no need to write SQL for basic operations)

(II) Object Oriented : Works directly with Java objects instead of rows and columns.

(III) Automatic mapping : Maps classes to tables using annotations like @Entity.

(IV) Built-in caching : Improves performance with automatic caching.

(V) Database Independent : Easier to switch databases without changing much code.

25] Describe the difference between the Entity manager's persist(), merge(), and remove() operations. When would you use each method in a typical database transaction scenario?

⇒

Difference between persist(), merge() and remove() in JPA is as follows:-

Method	Purpose	When to use
Persist (obj)	Adds a new entity to the database.	When you want to insert a new record.
merge (obj)	Updates an existing entity.	When you want to update an existing record or reattach a detached object.
remove (obj)	Deletes an entity from the database.	When you want to delete a record

Usage in Transaction Seenario :-

- Persist () → Use to insert a new student.
- Example:

```
Student s = new Student ("Nipa", "Nipa@mail.com");
entity manager. persist (s);
```

- merge () → Use to update an existing student.

Example:

```
s. setName ("Nipa update");
entityManager . merge (s);
```

- remove () → Use to delete a student.

Example:

```
entity manage . remove (s)
```

All three methods should be used within a transection typically managed by @Transactional or EntityTransection.

Lab-8

26] Design a simple CRUD application using Spring Boot and mySQL to manage student records.

=>

Goal: manage student records with fields like id, name, email and course.

Technologies Used:

- Java
- Spring Data JPA
- Spring Boot
- mySQL

1] Entity class : Student.java

```
import Jakarta.persistence.*;
@Entity
Public class Student {
@Id
@Generated Value (Strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    pritvate String email;
    private String course;
}
```

2] Repository Interface : StudentRepository.java

```
import org.springframework.data.jpa.repository.
                                    JPARepository;
public interface StudentRepository extends JpaRepository
                                        <Student, long>
{
}
```

3] Service Layer (StudentService.java)

=>

```
import org.springframework.beans.factory.annotation.
                                                Autowired;
import org.springframework.stereotype.Service;
import java.util.List;
import java.util.Optional;

@Service
public class StudentService{

@Autowired
 private StudentRepository repository;
 // Creat
 public Student SaveStudent (Student student){
     return repository.save (student);
 }
```

```
// Read all
public List<Student> getAllStudents() {
    return repository.findAll();
}
public optimal<Student> getStudentById(Long id){
return repository.findById(id);
}
public Student UpdateStudent(Long id, Student updateData)
{Student student =repository.findById(id).
                    orElseThrow();
 Student.setName(UpdateData.getName());
 Student.setCourse(updateData.getcourse());
 return repository.save(student);
}
public void deleteStudent(Long id){
    repository.deleteById(id);
}
}
```

4] Controllers (StudentController.java)

=>

```
import org.springframework.beans.factory.annotation.
                                        Autowired;
import org.springframework.web.bind.annotation.*;
import java.util.List;
import java.util.Optional;
@RestController
@RequestMapping("/students")
public class StudentController {

@Autowired
private StudentService service;

@PostMapping
public Student createStudent(@RequestBody Student stu
return service.saveStudent(student);
}
@Get mapping
public List<Student> getAllStudents() {
   return service.getAllStudents();
}
```

```
@Get mapping ("/{id}")
public Optional <Student> getStudent (@pathvariable
                                      Long id){
    return Service.getStudentById(id);
}

@PutMapping ("/{id}")
public Student updateStudent (@pathvariable Long id,
                              @RequestBody Student
                                      Update){
return Service.updateStudent (id, update);
}

@DeleteMapping ("/{id}")
public String deleteStudent (@pathvariable Long id){
    Service.deleteStudent(id);
    return "Student delete with id:" +id;
}
}
```

Lab-9

27] How does Spring Boot simplify the development of RESTful service ?

=> The process how Spring Boot Simplifies RESTFUL service Development :-

- Auto Configuration: Spring Boot auto-configures REST API → using built-in support for JSON, Tomcat Servers, etc.
- Reduced Boilerplate: No need to write xml or complex or complex setup
- Embedded servers: Comes with an embedded server (Like Tomcat) to run immediately.
- Built-in JSON Support: Automatically converts Java objects to JSON and vice versa using Jackson.
- Install: adds the project to local repo.
- deploy: Deploys to remote server.

Structure of a Typical pom.xml

=>

```
<project xmlns = "http://maven.apache.org/pom/4.0.0" ->
  <modelVersion> 4.0.0 </modelVersion>

  <groupId> com.example </groupId>
  <artifactId> Student-crud </artifactId>
  <version> 0.0.1 -SNAPSHOT </version>

<parent>
  <groupId> org.springframework.boot </groupId>
  <artifactId> spring-boot-starter-parent </artifactId>
  <version> 3.2.0 </version>
  </parent>

<dependencies>
<dependency>
  <groupId> org.springframework.boot </groupId>
  <artifactId> spring-boot-starter-web </artifactId>
</dependency>
```

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
</dependency>

<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>
</dependencies>

<build>
<plugins>
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
</plugins>
</build>
</project>
```

- How the spring Boot starter Dependencies, Help:-

Starter Name	What It does
Spring-boot-starter-web	Adds REST APIS, Tomcat JSON support
Spring-boot-starter-data-JPA	Adds Hibernate + JPA for database access
Spring-boot-starter-test	Adds JUNIT & mockito for testing

30] Compare maven and Gradles as build tools in spring Boot project.

=)

Feature	maven	Gradle
Syntax	xML (pom.xML)	Groovy / kotlin (build.gradle)
Readability	verbose, structured	Concise, flexible
Performance	Slower	Faster
Dependency	<dependency> blocks (xML format)	implementation, /compile only
Build speed	Slower on large projects	Generally faster

Maven example:

xml

```
<dependency>
<groupId> org.springframework.boot </groupId>
<artifactId> spring-boot-starter-web </artifactId>
</dependency>
```

Gradle example: build.gradle for REST API

```
plugins {
  id 'org.springframework.boot' version '3.2.0'
  id 'io.spring.dependency-management' version '1.1.0'
  id 'java'
}
group = 'com example'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '17'
repositories {
    mavenCentral()
}
```

```
dependencies {
    implementation 'org.springframework.boot:
        spring-boot-starter-web'
    implementation 'org.springframework.boot:
        spring-boot-starter-data-jpa'

    implementation 'mysql-connector-java:8.0.33'
    testImplementation 'org.springframework.boot:
        spring-boot-starter-test'
}
```