Name: Abdullah Al Mamun Sagar
ID: IT22059

Q-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.

Ans: In Java protected members of a class:

- Can be accessed within the same package by any class.
- Can be accessed in a different package only by subclass.

Case: 1

Parent and child int the same package:

```
Package Package1;
public class parent{
  protected String message = "Hello from Parent";
}

package package1;
public class Child extends Parent{
  public void showMessage(){
    System.out.println(message);
  }
```

```
public static void main(String[] args){
    Child c = new Child();
    c.showMessage();
}
```

Result: Child can directly access the protected member because it's in the same package.

Case: 2 Child class in different Package.

A protected member is still accessible in a subclass from a different package but only through inheritance.

```
package package1;
public class Parent{
    protected String message= "Hello from Parent!";
}

Package Packge2;
public class child extent Parent{
    public void displayMessage(){
    System.out.println(message);
}
public void anothermethod(){
Parent P = new Parent();
}
}
```

IT22059

Lab-1

Q-2 Ans:

Comparison Abstract Classes vs Interfaces:

Feature	Abstract Class	Interface
Multiple Inheritance	Not supported	Fully supported
Keyword	abstract	interface
Method	Can have both abstract and concrete method.	Can have abstract, default & static method.
Constructor	Yes	No constructe
Access Modifier	Can use any	Method are public by default.

Lab:1

* Use interface when:
  - Need multiple inheritance
  - Want to define a contract
  - Implement Classes come from different inheritance classes.
  - Example: Comparable, Runnable, Serilizable.

* Use abstract Class when:
  - Want to share code.
  - Need constructor or non-static instance variable
  - Classes to be closely relate in hierarchy.
  - Example: Animal → extended by Dog, Cat

Example:

Interface:

```
interface Flyable{
    void fly();
}
interface Swimmable{
    void swim();
}
Class Duck implements Flyable, Swimmable{
    public void fly(){System.out.println("Duck flies");}
    public void swim(){System.out.println("Duck swims");}
}
```

Abstract Class:

```
abstract Class Animal{
    void eat(){System.out.println("Eating");}
    abstract void makesound();
}
Class Dog extends Animal{
    void makesound(){System.out.println("Bark");}
}
```

IT22059

Lab-2

Q-3 * Encapsulation ensure data security and integrity:

Ans: Encapsulation is the process of hiding internal data and exposing it through controll access methods.

It ensures;

- Data security by preventing direct access to fields.
- Data integrity by using validation logic inside setters.

Example: BankAccount Class (Lab2)

```
public class BankAccount{
    private String accountNumber;
    private double balance;
public void setAccountNumber(String accountNumber){
    if(accountNumber == null || accountNumber.trim().isEmpty())
    { System.out.println("Error: Account number
            can not be null or empty.");
        return; }
    this.account.Number = accountNumber;
    }
Public void setInitialBalance(double balance){
  if (balance<0){
```

```
System.out.println("Error: Initial balance can not
                    be negative.");
        return;
    }
this.balance = balance;
}

public String getAccountNumber() {
    return accountNumber;
    }
public double getBalance(){
    return balance;
}

public void displayAccount(){
System.out.println("Account Number: "+accountNumber);
System.out.println("Balance: $" + balance);
}
}
}
```

Q-4 Ans: i) Kth smallest element in an Arraylist

```
import java.util.*;
public class KthSmallest {
  public static void main (String[] args){
    ArrayList<Integer> list = new Arraylist<>
                (Arrays.asList(5,2,8,1,9));
      int K = 3;
  Collection.sort(list);
   System.out.println(K+"rd smallest element is:
                " + list.get(K-1));
}
```

ii) Treemap: word → Frequency

```
import java.util.*;
public clas WordFrequency{
   public static void main (String[]args){
     String text = "helb world hello java";
     String[] words = text.split(" ");
TreeMap<String, Integer> map= new TreeMap<>();
  for (String word : words){
  map.put(word, map.getOrDefault(words) +1);
}
```

```
        System.out.println("Word Frequencies: "+map);
    }
}
```

v) Check if two linked list are equal.

```
import java.util.*;
Public class LinkedList Equality {
    Public static void main (String[] args){
    LinkedList <Integer> list1 = new LinkedList <>
                            Arrays.asList (1,2,3));
    LinkedList <Integer> list2 = new LinkedList <>
                            Array.asList (1,2,3));
    boolean isEqual = list1.equals(list2);
    System.out.println("Lists are equal: "+isEqual);
    }
}
```

Lab-3

Q-5 Java multithreadding project that simulate a car Parking Management System:

Code:

1. Class: RegistrcarParking.java

```
public class RegistrcarParking{
    private final String carNumber;
public RegistrcarParking (String carNumber){
  this.carNumber = carNumber;
}
  public String getCarNumber(){
    return carNumber;
}}
```

2. Class: Parking pool

```
import java.util.LinkedList;
import java.util.Queue;
Public class ParkingPool{
  private final Queue<RegistrcarParking> ParkingQueue=
                    new LinkedList<>();
Public synchronized void requestParking(RegistrcarParking car){
    parkingQueue.add(car);
```

```
System.out.println("Car " + car.getCarNumber() +
                   " requested Parking.");
    notify();
}

public synchronized RegistrarParking getNextCar(){
    while (ParkingQueue.isEmpty()){
    try {
        wait();
    } catch (InterruptedException e){
        Thread.currentThread().interrupt();
    }
    return ParkingQueue.poll();
}
}
```

3. Class: ParkingAgent.java

```
public class ParkingAgent extends Thread{
    private final int agentId;
    private final ParkingPool pool;
public ParkingAgent(int agentId, ParkingPool pool){
    this.agentId = agentId;
    this.pool = pool;
}
```

```
@Override
public void run () {
While (true)
Registrar ParkingCar = pool.getNextCar();
System.out.println ("Agent." + agentId + "Parked car" +
                    carGetCarNumber()+ ".");
try {
Thread.sleep (1000);
} catch (InterruptException e)
   break;
}
}
}
```

4. Class: MainClass.java

```
public class MainClass
   public static void main (String[] args){
   parkingPool pool = new ParkingPool();

   new ParkingAgent (1, pool).start();
   new ParkingAgent (2, pool).start();
String [] cars [] cars = {"ABC123", "XYZ456", "LMN789",
                          "DEF111", "PQR222"};
```

```
for (String carNum : cars) {
    new Thread(() -> {
        RegistrarParking car = new RegistrarParking(carNum);
        pool.requestParking(car);
    }).start();
    try {
        Thread.sleep(500);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
}
}
}
```

Q-6 How does Java handle XML data using DOM and SAX Parsers? Compare them. Where SAX prefer than DOM.

Ans:

Java Prefer two main XML Parser:

DOM: (Document Object Model):

- Load entire XML into memory as a Tree.
- Allow random access and modification.

SAX: (Simple API for XML) Parser;

- Parses XML sequentially using events.
- Does not load full XML into memory

Feature	DOM	SAX
Memory	High	Low
Speed	Slower for large file	Faster for large file
Access	Random	Sequential only
Modifiable	Yes	No
Use case	Small/moderate XML, editable	Large read-only XML

Scenario where SAX is preferred:

Parsing a large server log XML file to extract specific tags (e.g. error messages) without needing to modify or load the whole file into memory.

Q-7 Virtual DOM in React improve performance and comparison with traditional DOM

Ans: Virtual DOM in React improve performance by minimizing direct manipulation of the actual Document Object Model (DOM), which is a costly operation. Instead of directly updating the real DOM every time a change occurs, React employs the following process:

- In-memory representation
- Diffing Algorithm
- Batching and Reconcilation
- Minimal DOM Updates

Virtual DOM vs Traditional DOM

Feature	Traditional DOM	Virtual DOM (React)
Update Method	Direct Manipulation	Calculate and batches update
Performance	Slower	Faster
Re-rendering	Full DOM updates	Only diffs and patches
Control	Manual	Declarative (via state)

Diffing Algorithm:

1. React compares old and new virtual DOM trees.
2. Identify minimal changes.
3. Update only the affected parts in the real DOM

Example:

Initial:

<P> Hello </P>

After update:

<P> Hi </P>

React Change: "Hello" → "Hi"

Q-8 What is event delegation in JavaScript, how does it optimize performance? Explain with an example?

Ans: **Event Delegation:** Event delegation is a technique where a single event listener is attached to a parent element to handle events for current and future child elements, using event bubbling.

Optimize Performance:

1. Reduce memory by attaching fewer event listener.
2. Work for dynamically added elements.
3. Improves performance in Large or dynamic DOM trees.

Example: Click on dynamically Added Buttons:

```
<div id="container">
</div>
<script>
const container = document.getElementById("container");
```

```
container.addEventListener("click", function(e){
  if(e.target.tagName === "BUTTON"){
  alert("Button clicked: "+e.target.textContent);
  }
});

for(let i=1; i<=3; i++){
  const btn = document.createElement("button");
  btn.textContent = "Button " + i;
  container.appendChild(btn);
  }
</script>
```

Event delegation uses event bubbling to handle event efficiently.

Q-9 : Explain how Java Regular Expression can be used for input validation. Write a regex pattern to validate an email address and describe how it works using the Pattern and Matcher classes:

Ans:

Java Regular Expression (Regex) are a powerful tool for input validation, ensuring that user-provided data conforms to specific format and constraint. This is achieved by defining a pattern that the input string must match.

Step:

1. Define the Regex Pattern
2. Use String.matches()
3. Use pattern and Matcher Classes
4. Handle validation Results.

Email Validation Example:

```
import java.util.regex.*;
public class EmailValidator {
    public static void main (String[] args){
    String email = "User@example.com";
    String regex ="^[\\w.-]+@[\\w.-]+\\.[a-zA-Z]{2,}$";
    Pattern pattern = Pattern.compile (regex);
    Matcher matcher = pattern.matcher (emai);
 if (matche.matches()) {
    System.out.println ("Valid email");
 } else {
    System.out.println ("Invalid email");
 }
}
}
```

Here:

Pattern.compile() → compiles in regex

matcher.matches() → check if the input matches the pattern.

This regex validates standard email format.

Q-10 Ans: Custom Annotation:

- Custom annotation are user defined metadata in java.
- Declared using @ interface.
- Can be processed at runtime using reflection to influence program behavior.

Example:

1. Define Annotation

```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
Public @interface RunMe{
    String value() default "Running";
}
```

2. Annotation Methods.

```
Publi class TestClass
@RunMe("start task")
Public void task1(){
    System.out.println("Task 1");
}
```

```
@RunMe
public void Task2() {
    System.out.println("Task 2");
}
}
```

3. Process with Reflection:

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

Q-11 : Singleton Design Pattern in Java.

Ans:

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

Problem it Solves:

- Control access to shared resources.
- Prevent multiple instantiations of a class, saving memory and ensuring consistency

Basic Example:

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

Thread Safe Version (Double-Checked Locking):

```
Public class Singleton {
private static volatile Singleton instance;
private Singleton () {}
Public static Singleton getInstance() {
  if (instance == null) {
    Synchronized (Singleton.class) {
      if (instance == null) {
        instance = new Singleton ();
      }
    }
  }
  return instance;
}
```

Lab-4

Q-12 JDBC communication with Database:

Ans: JDBC uses a driver (MySQL connector/J) to translate Java API calls into database-specific protocols enabling standardized interaction with relational databases.

Steps for SELECT Query & Results

1. Load Driver (Modern JDBC auto-loads)
2. Establish Connection:

```
Connection conn = DriverManager.getConnection
                                (url, user, pass);
```

3. Create statement:

```
Statement stmt = conn.createStatement();
```

4. Execute Query:

```
ResultSet rs = stmt.executeQuery("SELECT---");
```

5. Process ResultSet.

```
while (rs.next()) {
  int id = rs.getInt("id");
String name = rs.getString("name");
}
```

6. Close Resources: ResultSet → Statement → Connection

Error Handling Essentials:

```
try {
}
catch (SQLException e) {
    System.err.println("Database error: " + e.getMessage());
}
finally
    try { if (rs != null) rs.close(); } catch (SQLException e) {
try { if (stmt != null) stmt.close(); } catch (SQLException e) {
try { if (conn != null) conn.close(); } catch (SQLException e) {
}
```

Key Point:

- Use try-catch for SQLException
- finally ensures resources are closed even if exception occurs.

Q-13 Servlet and JSPs in MVC Architecture

Ans: They work Together:

- Servelet (Controller): Handles client requests, processes input and control application flow.
- JSP (View): Displays the data (UI) returned by the controller.
- Java class (Model): Contains business logic and data.

Use case Example:

1. Model – Student.java

```
public class student {
    private String name;
    private int roll;
  Public Student (String name, int roll) {
    this.name = name;
    this.roll = roll;
  }
  public String getName() { return name; }
    public int getRoll() { return roll; }
}
```

2. Controller - StudentServlet.java

```
@WebServlet("/student")
public class StudentServlet extend HttpServlet{
    protected void doGet(HttpServletRequest req,
        HttpServletResponse res) throws IOException,
                                        ServletException{
    Student s=new Student("Alice",101);
    req.setAttribute("student",s);
RequestDispatcher rd = req.getRequestdispatcher("student.jsp");
    rd.forward(req,res);
}}
```

3. View → student.jsp

```
<%@ page contentType="text/html" %>
<%
  Student s = (student) request.getAttribute("student");
%>
<h2> Student Info</h2>
<P> Name: <%= s.getName() %></P>
<P> Roll : <%= s.getRoll() %></P>
```

Q-14: Java Servlet Related

Ans:

Life cycle of JavaServlet:

1. Loading and installation – Container loads the servlet and crete an instance.
2. Initialization (init()) — called onee when the servlet is first created.
3. Request Handling (service()) — called for each client request
4. Destruction (destroy()) → called once when the servlet is being taken out of service.

* Role of:

init() → Initialized servled (called once)
service() → Handles requests (doGet(), doPost(), etc)
destroy() → Cleans up resources before servlet removal

Handling Concurrent Requests:

- The servlet container uses a single instance of the servlet.
- For each client request, it spawns a separate thread that calls service().
- This allows concurrent request handling

Thread Safety Issue:

If the servlet uses shared mutable variable without synchronization, multiple threads can corrupt data or cause race conditions.

Solution:
Use:

- Synchronized blocks/methods
- Thread-safe classes
- Avoid shared mutable state.

Q-15: Problems occurs if shared resources are accessed by multiple thread:

Ans: In Java Servlet, a single instance handles multiple client requests simultaneously. each in its own thread. If shared resources are access without protection, it can lead race condition, inconsistent data or thread interference.

Example: Thread - Unsafe Servlet

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

Issue: Multiple threads may update counter at the same time, causing incorrect or skipped counts.

Solution: Use synchronized for Thread Safety

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

Q-16: Describe how the MVC pattern separates concern in a Java web application. Explain the advantage of this structure in terms of maintainability and scalability, using a student registration system as an example.

Ans:

How it separate concern:

Model: Manage data and logic
View: Display data to users
Controller: Access user input, updates model and select view.

Advantage of MVC:

Maintability → Easy to update UI or logic without affecting others.
Scalability → Better structure of growing codebases and terms.

Example: Student Registration System.

1. Model - Student.java

```
public class student{
    private string name;
    private string email;
}
```

2. Controller – StudentServlet.java

```
@WebServlet("/register")
public class StudentServlet extends HttpServlet {
protected void doPost(HttpServletRequest req,
    HttpServletResponse res) throws ServletException,
                                    IOException {
    String name = req.getParameter("name");
    String email = req.getParameter("email");
  Student student = new Student();
   Student.setName(name);
   Student.setEmail(email);
  req.setAttribute("student", student);
   RequestDispatcher rd = req.getRequestDispatcher("success.jsp");
   rd.forward(req, res);
}}
```

3. View – success.jsp

```
<%@ page contentType = "text/html" %>
<%
  Student s = (Student) request.getAttribute("student");
%>
<p> Student <%= s.getName() %> registered successfully! </p>
```

Q-17

Servlet controller in Java EE - [Lab 5]

Servlet controller manage the flow between the model and the view.

Step by Step Flow:

1. Recieve Request (servlet recieve input from the HTTP request.)
2. Process Input (The servlet create or uses a model to process or store the data.)
3. Store data in Request Scope (It uses a ~~mode~~ request.setAttribute() to attach the model.)
4. Forwards to JSP (It forwards the request to a JSP page using RequestDispatcher)

Simple Example:

1. Model - Student.java

```
public class student {
    private String name;
    public Student(String name){this.name=name;}
    public String getName (){return name;}
}
```

2. Controller - StudentServlet.java

```
@WebServlet("/register")
public class StudentServlet extends HttpServlet {
    protected void doPost(HttpServletRequest req,
        HttpServletResponse res) throws ServletException, IOException {
        String name = req.getParameter("name");
        Student student = new Student(name);
        req.setAttribute("student", student);
        RequestDispatcher rd = req.getRequestDispatcher("student.jsp");
        rd.forward(req, res);
    }
}
```

3. View - student.jsp

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%
    Student s = (Student) request.getAttribute("student");
%>
<h2>Hello, <%= s.getName() %>! Registration successful.</h2>
```

Q-18: Compare and contrast cookies, URL rewritting, and HttpSession as methods for session tracking in servelts. Discuss their advantage, limitations and ideal use case.

Ans: In Java Servlet, comparison of cookies, URL rewriting and HttpSession:

Feature	Cookies	URL Rewriting	HttpSession
Storage	Cliend side	Session ID is appended to the URL	Server-side
Visibility	Not visible.	Visible in th URL	Not visible to users.
Security	Can be stolen or modified.	Less secure.	More secure
Data capacity	Limited	Only passes session-ID not data.	Can store full object and large data.
Ease of use	Simple to implement	Required manual appending	Very easy using

Advantages & Limitation:

Method	Advantages	Disadvantages
Cookies	Simple, automatic with browser support.	Disable in some browsers require session ID management.
URL Rewritting	Works without cookies	Exposes session ID in URL, manual effort
HttpSession	Server-side secure, store full user data	Memory overhead on server require session ID management

Ideal Use case:

Cookies → Store small client preferences

URL Rewritting → When cookies are disabled but session is still needed.

HttpSession → For secure and full-featured session management.

Q-19 A web application stores user login information using HttpSession. Explain how the session works across multiple request and how session timeout or invalidation is handled securely.

Ans:

Multiple Requests:

On each request the browser sends back the session ID, allowing the server to:

- Identify the user
- Retrive user data from the session.

Such as:

```
String user = (String) session.getAttribute("username");
```

Session Timeout and Invalidation:

1. Session Timeout in web.xml:

```
<session-config>
<session-timeout> 15 </session-timeout>
</session-config>
```

2. Session Invalidation:

Use when user logs out or want to end the session

```
session.invalidate();
```

- After invalidation, any access to session attribute will return null.

Secure Practices:

Session.invalidate() → Prevent reuse after logout

Use HTTPS → Protects JSESSIONID from theft

Regenerate session ID → On login to prevent session fixation

Set timeout → Auto expire inactive sessions.

Q-20 Ans:

How Spring MVC handles and HTTP Request:

→ Spring MVC Follows the Model-View-controller pattern to clearly separate business logic, request handling and presentation.

Role of:

@controller → Mark a class as a controller to handle HTTP request.

@RequestMapping → Maps HTTP URL Paths to controller method.

Model Object → Passes data from the controller to the view.

Request Flow: Login Form Example

1. User submits login form (/login)
2. DispatcherServlet intercepts the request
3. Controller handle the request.

Such as:

```
@controller
public class Logincontroller{
  @RequestMapping (value= "/login", method=RequestMethod.POST)
  public string login(@ RequestParam String username,
                      @ RequestParam string password,
                      Model mode){
  if ('admin'. equals (username && '1234'. equals (password)){
    model. addAttribute ("message", "Login Successful!");
     return "welcome";
  } else{
    model. addAttribute("error", "Invalid credentials");
     return "login";
  }
}
```

4. Business Logic (Add results to Model)
5. View (JSP / Thymleaf)

Q-21 Ans:

Dispatcher Servlet is the front controller in Spring MVC. It manages the entire request processing workflow.

1. Recieves the HTTP request.
2. Uses HandlerMapping to find the appropriate @controller method.
3. Calls the method and gets the view name.
4. Uses ViewResolver to map the view name to a physical view life.
5. Renders the view and sends the response back to the client.

Interaction:

1. HandlerMapping → Maps URL to controller.
2. ViewResolver → Maps view name to actual view (like /WEB-INF/views/home.jsp

[Lab-6]

Q-22 Ans: Prepared Statement is better than Statement in aspect of performance and security.

Performance → Precompiled SQL and reduce DB overhead in Prepared Statement. In Statement SQL recompile every execution that's why it is slow.

Security → Prepared Statement automatic escaping but Statement concatenation.

Example:

MySQL Table:

```
Create Table students (
id Int Primary key auto_increment,
name varchar (30),
email varchar (50)
);
```

Java Code:

```
import java.sql.*;
public class InsertStudent{
    public static void main(String[] args){
    String url = "jdbc:mysql://localhost:3306/school";
    String user = "root";
    String password = "password";
try{
    Connection con=DriverManager.getConnection
                        (url,user,password);
    String sql = "Insert into student
                        (name, email) values(?,?)";
PreparedStatement pst = con.prepareStatement(sql);

    pst.setString(1,"Sagor");
    pst.setString(2,"sagor@example.com");
    int rows = pst.executeUpdate();
System.out.println(rows+"row(s) inserted.");

    pst.close();
    con.close();
}catch(Exception e){
    e.printStackTrace();
}}}
```

Q-23 Ans:

ResultSet in JDBC:

A ResultSet is a table like object in JDBC that holds data retrieved from a database using a SELECT query.

Use of:

next() — moves to the next row

getString(column) — gets a string from the current row

getInt(column) → gets an integer from the current row.

Simple Example: Reading from MySQL

MySQL Table:

```
CREATE TABLE Students (
    id INT,
    name VARCHAR(50)
);
```

IT22059

Java Code:

```
import java.sql.*;
public class ReadStudents {
  public static void main (String [] args){
  try{
   Connection con=DriverManager.getConnection(
     "jdbc:mysql://localhost:3306/school", "root", "password");
   Statement stmt = conn.createStatement();
   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 +": " +name);
   }
   rs.close();
   Stmt close();
   conn close();
  } catch(Exception e){
     e.printStackTrace();
}}}
```

Q-24 Ans:

JPA (Java Rest Persistance API) Automatically maps Java classes (Object) to relational database tables using annotations.

@Entity → Marks the class as a database entity
@Id → Marks the primary key field
@GeneratedValue → Auto generate primary key values.

Example: Mapping a student Class.

```
import jakarta.persistence.*;
@Entity
Public Class Student{
@Id
@GeneratedValue(Strategy=GenerationType.IDENTITY)
 Private Long id;
 private String name;
 private string email;
}
```

This maps to a table like:

```
CREATE TABLE student (
id BIGINT AUTO_INCREMENT PRIMARY KEY,
name VARCHAR (255),
email VARCHAR (255)
);
```

Advantage of JPA Over JDBC:

Feature	JDBC	JPA
SQL writting	Manual	Auto-generated
Result Mapping	Manual	Automatic Object mapping.
Relationship	Must handle manually	Easy with annotation
Transaction	Manual	Declarative
Productivity	Lower	Higher

Q-25 Ans:

EntityManager Methods In JPA : Persist(), merge() and remove()

1. persist(entity):

purpose: Insert new entity in database

Use case: When saving a new object.

Example:

```
Student s= new Student ("sagor");
entityManager: persist(s);
```

2. Merge(entity):

Purpose: Update an existing entity or saves a detached entity.

Use case: When updating data

Example:

```
Student s = entityManager.find(Student.class, 1L);
s.setName("sagor");
entityManager.merge(s);
```

3. remove(entity);

Purpose: Delete the entity from the database.

Use case: When deleting a record.

Example:

```
Student s = entityManager.find(Student.class, 1L);
entityManager.remove(s);
```

[Lab-8]

Q-27 Ans:

Springboot & RestFul Services:

Springboot simplifies REST API development by:

- Auto configuring the server and dependencies.
- Handling JSON automatically using Jackson
- Providing easy-to-use annotation like @RestController, @GetMapping @postMapping

Example : Simple REST controller

```
@RestController
@RequestMapping("/students")
public class StudentController {
List<Student> students = new ArrayList<>();
@GetMapping
  public List<Student> getAll(){
    return students;
  }
  @PostMapping
  public Student addStudent(@RequestBody Student student){
    students.add(student);
    return student;
  }
}
```

- @RestController → Marks th class as a REST API
- @GetMapping → Handles GET requests
- @PostMapping → Handles POST request with JSON

Q-28 Ans:

@ Rest Controller:

- Used for REST APIs.
- Returns JSON / XML
- Combine @ controller + @ ResponseBody

@ Controller:

- Used for web apps
- Return HTML views (eg. Thymleaf)

Example:

```
@ RestController
@ RequestMapping ("/books")
Public class BookController {

@ GetMapping
  Public List <Books> getAllBooks () {
  }   // return all books

@ GetMapping ("/{id}")
  Public Book getBook (@ PathVariable Long id) {
      // return book by id
  }
```

```
@PostMapping
public Book addBook (@RequestBody Book book){
    // save and return book
}

@PutMapping ("/{id}")
public Book UpdateBook(@PathVariable Long id,
                       @RequestBody Book book){
    //Update and return book
}

@DeleteMapping ("/{id}")
public void deleteBook (@PathVariable Long id){
    // delete book
}
```

Q-30 Ans:

Aspect	Maven	Gradle
Syntax	XML-base	Groovy/Kotlin
Performance	Slower - builds, no built in caching	Faster build with incremental build
Dependency Handling	Used <dependency> tags, transitive dependencies managed	Uses implementation or api keyword flexible conflict resolution.

Example: build.gradle for Simple REST API

```
plugis {
    id 'org.springframework.boot' version '3.20'
    id 'io.spring.dependency-management' version '1.1.4'
    id 'java'
}

group = com.example
version = '1.0.0'
sourceCompatibility = '17'

repositories {
    mavenCentral()
}
```

```
dependencies {
    implementation 'org.springframework.boot:spring-boot
                    -starter-web'
    testImplementation 'org.springframework.boot:
                        Sprinboot-Starter-test'
}
test {
    useJUnitPlatform()
}
```

Q-31 Ans:

1. Project Structure

```
parent-project/
|
├── api/
├── service/
├── database/
└── pom.xml or build.gradle
```

2. Using Maven

Parent pom.xml

```
<modules>
<module>database</module>
<module>service</module>
<module>api</module>
</modules>
```

Inter module dependencies:

- service pom.xml depend on database
- api/pom.xml depend on service

3. Using Gradle:

Parent build.gradle

```
include 'api', 'service', 'database'
```

Module dependencies:

In api/build.gradle

```
dependencies {
    implementation project(':database')
                           (':service')
}
```

In service/build.gradle:

```
dependencies {
    implementation project(':database')
}
```

IT22059

[Lab-9]

Q-32

Project Title : Language Translator

Project Layers :

1. Model — Data Classes Like UserInput, LdResult Translated Output, Spell, sentences

2. Service — contains Translation service for business logic

3. controller — Translation controller handles web requests.

4. View — HTML pages using Thymleaf (translation.html, result.html)

Project Structure:

```
src
+main
| + java/com.topcoder/
| | + controller/
| | + service/
| | + model/
| + resources
| | + template/  → HTML UI (Thymleaf)
| | L application.properties
```