Ans-1 : A protected member of a parent class is accessible by a child within the same package

Example:

```
package pack1;
class parent {
  protected int x=10;
}
class Child extends Parents {
    void show(){
        System.out.println (x); // Accesible
    }
}
```

If the parent class is in the different package, it can be still access the protected member, but only through inheritence, not by object reference. (But in same package can access protected member via parent object)

Example:

```
package pack1;
public class Parent {
    protected int x=10;
}

package pack2;
import pack1.Parent;
class Child extends Parent {
    void show(){
        System.out.println (x); // accesible
} }
```

```
Package pack2;
import pack1.Parent;
Class Other {
    void (){
        parent p= new Parent();
        System.out.println(p.x); // compile-time error
    }
}
```

Ans 2: Multiple inheritence support:

Abstract Class: Java doesn't support multiple inheritence with classes, so a class can extends only one abstract class.

Interface: Java support multiple inheritence with interfaces

A class can implement multiple interfaces

Use abstract class:

i) When you need to provide some common code

ii) When methods may have default behavior

iii) When all child classes share 'is-a' relationship

Use interface:

i) When you want to achieve multiple inheritance
ii) When you need to define only method signatures
iii) When unrelated classes need to share common capability

Example:

```
abstract class A{
    abstract void showA();
}
abstract class B{
    abstract void showB();
}
class C extends A,B{  // compilation Error
}

interface Flyable{
    void fly();
}
interface Swimmable{
    void swim();
}
class Duck implements Flyable, Swimmable{
    void fly() {System.out.println("Duck flying");
    void swim() {  "  ("Duck swimming")
}
```

Ans-3 Encapsulation is a principle of OOP where data are kept private and can only be accessed or modified through public methods (getters / setters). This ensure

i) Data security - Prevents direct access to sensitive data

ii) Data integrity - Validate inputs before changing state

iii) Controlled access - we can enforce rules

Example:

```
public class BankAccount {
    private String accountNumber;
    private double balance;
    public void setAccountNumber(String accNo) {
        if (accNo != null && !accNo.isEmpty()) {
            accountNumber = accNo;
        } else {
            System.out.println("Invalid account Number");
        }
    }
    public void setInitialBalance(double bal) {
        if (bal >= 0) {
            balance = bal;
        } else {
            System.out.println("Initial balance cannot be
                                negative");
        }
    }
```

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

Ans-6 Java provides two common ways to handle XML:

i) DOM (Document Object Model)

ii) SAX (Simple API for XML)

DOM Parser:

i) Loads entire XML file into memory as a tree

ii) Allows random access, easy navigation and modification

```
DocumentBuilderFactory factory = DocumentBuilderFactory.
                                 newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
Document doc = builder.parse("data.xml");
```

SAX Parser:

i) Reads XML sequentially

ii) Doesn't store whole document in memory

iii) Faster and memory-efficient for large XML files

```
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser parser = factory.newSAXParser();
parser.parse("data.xml", new DefaultHandler());
```

Feature	DOM	SAX
Memory Usage	High (loads whole file)	Low (reads line by line)
Processing Speed	Slower for large files	Faster for large files
Access type	Random read/write access	Sequential read-only
Use Cases	Small/medium XML with edits	Large XML, read-only parsing

If you're parsing a 500MB XML log file to extract specific tags (like <error>), SAX is preffered. It processes line by line without memory overload

* Use DOM for erase and editing

Ans 7 : In react, the virtual DOM (VDOM) is a lightweight copy of the real DOM kept in memory. It updates the actual DOM only when necessary, improving performance.

It improves performance by:

i) Updating the real DOM is slow

ii) React creates a new virtual DOM when state/props change

iii) It compares the new virtual DOM with the old one (diffing algo)

iv) Only the changed parts are updated in the real DOM - not the whole page

Feature	Virtual DOM	Traditional DOM
Updates	Efficient (only diffs)	Full re-render on change
Speed	Faster	Slower
Control	controlled by React	Manual

Diffing Algorithm Example:

```
function Greet(){
    return <h1>Hello</h1>;
}
```

If the state changes and React needs to render:

```
function Greet() {
    return <h1>Hello, World!</h1>
}
```

React:

i) Creates a new virtual DOM

ii) Compares with old virtual DOM

iii) Finds the difference ("Hello" → "Hello, World!")

iv) Updates only that in the Real DOM

Ans-8 : Event delegation is a technique where a single event listener is added to a parent element to hadle events on its child elements, even if they are added later

It optimize performance by -

i) Reducing the number of event listener

ii) Saves memory

iii) Works well with dynamically added elements

Example:

```
<ul id = "list">
   <li> Item 1 </li>
   <li> Item 2 </li>
</ul>
<script>
document.getElementById("list").addEventListener
                    ("click", function(e)) {
   if(e.target.tagName === "LI"){
      alert("Clicked: " + e.target.textContent);
   }
});
```

```
// Dynamically added item
const newItem = document.createElement("li");
newItem.textContent = "Item 3";
document.getElementById("list").appendChild(newItem);
</script>
```

Listener is added only on <ul>, not each
Works for new <li> like "Item 3" too
DOM traversal via e.target handles delegation

Ans-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 provides two core classes for 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

matches() → checks if the input fully matches the pattern

Ans-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(): Optional 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 Exc {
        MyService service = new MyService();
        Method[] methods = MyService.class.getDeclared
                                            Methods();
        for (Method method : methods) {
            if (method.isAnnotationPresent (Run... .class)) {
                Run... annotation = method.getAnno(
                                            Runtime... class)
                for(int i=0; i<annotation.times(); i++) {
                    method.invoke(service);
                }
            }
        }
    }
```

Ans 12 JDBC is an API that enables Java applications to connect and interact with relational databases (like MySQL, PostgreSQL, Oracle etc). JDBC manages communicati

i) Driver Manager loads the appropriate database driver

ii) Establishes a connection to the database

iii) Sends SQL queries to the DB engine

iv) Retrives ResultSet for SELECT queries

v) Closes resources (ResultSET, Statement, Connection) after use

```
import java.sql.*;
public class JDBCExample {
    public static void main(String args[]){
        Connection conn = null;
        Statement stmt = null;
        ResultSet rs = null;
        try{
          Class.forName("com.mysql.cj.jdbc.Driver");
          // load driver
          conn = DriverManager.getConnection("jdbc:my
                sql://localhost:3306/testdb", "root", "pass");
          //connection to DB
          stmt = conn.createStatement(); //create statement
          rs = stmt.executeQuery("SELECT * FROM users");
          // Execute SELECT query
          while(rs.next()){
              System.out.println(rs.getString("username");
          }//process result
        } catch(Exception e){
            e.printStackTrace(); // Handle exception
        }finally{
            try{
              if(rs!=null)rs.close();
              if(stmt!=null)st.close();
              if(conn!=null)conn.close();
```

```
        }catch (SQLException ex){
            ex.printStackTrace();
        }
    }
}
}
```

Ans 13 In a web application following the MVC pattern:

- Servlet acts as the controller: It handles client requests, processes input, interacts with the model, and decides which view to display.
- Java class acts as the Model: It contains the business logic and data (like fetching data from the database or processing it)
- JSP acts as the view: It presents data to the user in the form of HTML pages, displaying the results sent by the controller.

They work together:

1) The client sends a request to the Servlet (controller)
2) The servlet calls the model class to process the data or fetch required information
3) The model returns data to the Servlet

4) The servlet sets this data as request attributes and forwards the request to a JSP page (view)

5) The JSP uses the data to display the output to the user

Model (Java class):

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

Controller (Servlet):

```
protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
            IOException {
    User user = new User("Alice");
    request.setAttribute("user", user);
    RequestDispatcher rd = request.getRequestDispatcher
    rd.forward(request, response);
}
```

("welcome.jsp");

View (JSP - welcome.jsp):

```
<html>
<body>
    <h1>Welcome, ${user.name}!</h1>
<body>
<html>
```

Ans-16 The MVC (Model-View-Controller) pattern divides an application into three interconnected components:

1) Model: Represents the data and business logic
   Handles data operations like fetching, updating and validating (e.g., Student class, database interac

2) View: Presents data to the user (UI)
   Usally JSP pages that display information and user forms

3) Controller: Act as intermediary between view and model
   Handles user request, invokes model operations and selects the view to display

Advantages of MVC

Maintainability: Changes in UI (view) don't affect business logic (Model) or control flow (Controller)
Developers can work independently on M, V, C

Scalability: Easier to add new features by extanding one component without impacting othes
Supports multiple views for the same Model (web, mobile)

Model: Student class manages student data and

database operations

Controller: RegistrationServlet processes registration requests, validates input and calls model methods

View: registration.jsp shows the registration form and confirmation messages.

Flow: User submits forms → Controller handles request → Model saves student data → Controller forward view → view shows success page

Ans 17: Servlet Controller Managing Flow Between Model and View in Java EE:

① Receives client requests

② Interacts with Model to process or fetch data

③ Forwards the data to the view to generate the response

The servlet uses RequestDispatcher to forward requests along with data

Servlet (controller):

```
protected void (HttpServletRequest request, HttpServlet-
    Response response) throws ServletException, IO-
        Exception {
    User user = new User ("John");
    //Interact with model
```

```
request.setAttribute("user", user)
// Add data to request scope
RequestDispatcher dispatcher = reques.getRe-
        questDispatcher("userProfile.jsp");
dispatcher.forward(request, response);
// Forward to JSP(view)
}
```

JSP - userProfile.jsp:

```
<html>
<body>
  <h1> Welcome, ${user.name}! </h1>
<body>
<html>
```

Ans 20: Spring mvc handles an Http request in the following steps:

Request flow in Spring MVC:

1) Browser sends an HTTP request to the server
2) DispatcherServlet intercept the request
3) HandlerMapping maps the request to a method in a @Controller using @RequestMapping

4) The controller method executes:
It processes the request
Interacts with the model
Add data to a Model object
5) It returns a view name
6) The ViewResolver maps the view name to a JSP
7) The view is rendered with data from the model and returned to the browser

@Controller :- Marks a class as a controller that handles HTTP requests

@RequestMapping :- Maps HTTP requests to specific controller methods

Model :- An object used to pass data from the controller to the view, keeping business logic separate from presentation

Login form Submission:

```
@Controller
public class LoginController {
    @RequestMapping(value = "/login", method = RequestMethod.POST)
    public String login(@RequestParam("username") String username, @RequestParam("password") String password, Model model) {
        if("admin".equals(username) && "pass123".equals(password)) {
```

```
            model.addAttribute("message", "Login Successful");
            return "welcome";
        } else {
            model.addAttribute("error", "Invalid
                                credentials");
            return "login";
        }
    }
}
```

Ans 21: The DispatcherServlet is a core component of the Spring MVC framework. It acts as the entry point for all incoming HTTP requests in a Spring web application. It follows the Front Controller Design Pattern - meaning all request go through a single servlet that delegates the request to appropriate handlers.

Here is the step-by-step breakdown of how DispatcherServlet handles a request:

1) Client sends HTTP Request:

2) DispatcherServlet Receives Request from the browser

3) It consults HandlerMapping to find the appropriate @Controller and @RequestMapping method that matches the URL

4) It uses a HandlerAdapter to invoke the matched controller method

5) The controller returns a logical view name and adds model data

6) DispatcherServlet passes the view name to the View Resolver

7) The View Resolver resolves the view name to an actual view (JSP file)

8) The DispatcherServlet forwards the request, model data to the resolved view for rendering

9) The generated response is sent back to the client

Handler Mapping: Maps request URLs to controller methods. DispatcherServlet uses it to route requests

View Resolver: Converts logical view name to actual view resources. Delegates view rendering to it

Ans 22: Performance: Prepared Statement precompiles the SQL query once and reuses it for multiple execution, reducing parsing time on the database server. This improves performance especially for repeated queries with different parameters

Security: PreparedStatement uses placeholders (?) and sets parameter values safely, preventing SQL injection attacks by treating input as data, not executable code. On the other side Statement concatenates string to build SQL, making it vulnerable to injection.

Example (Insert Record using Prepared Statement)

```
String sql = "Insert INTO users (username, email)
        VALUES (?, ?)";
try (Connection conn = DriverManager.getConnection
    (dbURL, user, pass);
    PreparedStatement ps = conn.prepareStatement
                                        (sql){
        ps.setString (1, "alice");
        ps.setString (2, "alice@example.com");
        int rowInserted = ps.executeUpdate();
```

```
        System.out.println(rowInserted + "record (s) inserted");
    }catch (SQLException e){
        e.printStackTrace();
    }
```

Ans 23: A ResultSet is an object returned by executing a SQL query using JDBC. It holds the data retrieved from the database in a tabular form and allows navigation through rows

next() → Moves the cursor to the next row of the result set. Returns false when no more rows

getString() → Retrieves a column's value as a String

getInt() → Retrieves a column's value as an Int

```
String query = "SELECT id, name, email FROM users";
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(query);
while (rs.next()){
        int id = rs.getInt("id");
        String name = rs.getString("name");
        String email = rs.getString("email");
        System.out.println("ID: "+id+", Name: "+name+", Email: "
                                +email);
}
```

Ans 25: Difference between EntityManagers persist(), merge(), remove()

persist(object Entity): Adds a new entity instance to persistence context. Makes the entity managed and schedules for insert in the database when transaction commits. Use when creating and saving a new object for the first time

merge(object entity): Updates an existing entity by copying the state of the given detached entity into the manged entity. Returns managed entity. Use when you have a detached object and want to update the database with its changes

remove(object entity): Marks a managed entity for deletion from the database upon transaction commit. Use when you ~~have~~ want to delete an existing entity.

Ans 26: Student Entity Example:

```
@Entity
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String email;
}
```

Repository Interface:

```
public interface StudentRepository extends JpaRepository
                                    <Student, Long> {
}
```

CRUD operation using Repository:

Create:

```
Student student = new Student();
Student.setName("Alice");
student.setEmail("alice@example.com");
StudentRepository.save(student);
```

Read:

Get all students: `List<Student> students = StudentRepository.findAll();`

Get by ID: `Optional<Student> student = studentRepository.findById(id);`

Update: Retrieve entity, update fields and save

```
Student existing = studentRepository.findById(id)
                                    .orElseThrow();
existing.setEmail("newemail@example.com");
studentRepository.save(existing);
```

Delete:

```
studentRepository.deleteByID(id);
```

Ans 27: Spring Boot greatly simplifies the development of RESTful web series by:

i) Auto-Configuration: Automatically configures application components like Jackson (for JSON), Tomcat (server) etc

ii) Embedded Server: No need for external Tomcat/Jetty setup

iii) Starter Dependencies: Quickly adds needed dependencies with spring-boot-starter-web

iv) No XML config: Everything can be annotated using Java

v) Integrated JSON Support: Handles JSON serialization out of the box

@RestController : Combines @Controller and @ResponseBody marking the class as REST API handler that returns JSON/XML responses directly

@GetMapping and @PostMapping: Map HTTP GET and POST requests to controller methods

Example REST controller Handling JSON:

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

Ans 28: Difference Between @RestController and @Controller in Spring Boot:

Features	@RestController	@Controller
Purpose	Used for RESTful web services (API responses)	Used for MVC web applications (HTML views)
Automatically adds	@ResponseBody to all methods	No automatic response body
Returns	JSON / XML (object serialized	View name (Thymeleaf, JSP)
Use case	APIs, Mobile Backends, data services	Web UI, HTML rendering

RESTful API Endpoint Structure for Library System

HTTP Method	Endpoints	Purpose
GET	/books	Get a list of all books
GET	/books/{id}	Get details of a specific book
POST	/books	Add a new book
PUT	/books/{id}	Update a specific book
DELETE	/books/{id}	Delete a specific book

Ans 29: Maven is a build automation and dependency management tool used widely in Java projects, including Spring Boot. It helps developers manage libraries, compile code, run tests, package applications and more - all defined in a single pom.xml file.

Dependencies: Maven fetches required libraries (JAR files) from central repository and stores them locally. Declare dependencies in the <dependencies> section of pom.xml

Build Lifecycle: Maven follows a standard build lifecycle made up of phases like:

validate : validate project structure

complile : complie source code

test : Run unit tests

package : pacakage code (into a .jar)

install : install a local repo

deploy: Deploy to remote repo

Each phases triggers the previous ones in order

Typical pom.xml Structure:

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.example</groupId>
```

```
<artifactId> demo </artifactId>
<version> 1.0.0 </version>

<parent>
    <groupId > org. springframework. boot </groupId
    <artifactId > spring-boot-starter-parent <. .
    <version > 3.00 </version>
</parent>

<dependencies>
    <dependencies>
        <groupId> . . . . .
        <artifactId> spring-boot-starter-web
    </dependeeien>
</dependencies>

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

Starters bundle common dependencies needed for a specific feature (web, data JPA, security)

Avoid manual dependency management for every library. Just add starter. Ensure compitable version and

reduce configuration effort

Ans 30 Comparison of Maven and Gradle in Spring Boot

Feature	Maven	Gradle
Syntax	XML (pom.xml)	Groovy or Kotlin
Performance	Slower (no build caching by default)	Faster (supports build caching and incremental builds
Configuration	Declarative, verbose	More concise and flexible (declarative + imperative)
Dependency Hadling	Central repository (mvn-repository)	Uses same repositories but supports dynamic version
Build Customization	Difficult, plug-in diriven	Easier, programmic via code
Tooling Support	very mature, widely used	Rapidly improving, widely adopted in Android & Spring boot

build.gradle for simple Spring Boot REST API

```
plugins {
    id 'org.springframework.boot' version '3.0.0'
    id 'io.spring.dependency-management' version '1.1.0'
    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:
                    spring-boot-starter-test'
}
test {
    useJUnitPlatform()
}
```

Ans 31 Structuring a Multi-Module Boot Application

Project Structure:

Parent Project (Root): Contains common configuration and manages sub-modules

Modules:

API Module - Handles REST controllers and request handling

Service Module - contains business logic and services

Database Module - Manages repositories, entities and database access

Parent POM (pom.xml) includes:

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

api/pom.xml depends on service

```
<dependencies>
    <dependency>
        <groupId> com.example <
        <artifact> service <
        <version> ${project.version} <
    <
<
```

Managing Inter-Module Dependencies:

Use project dependencies to link modules

Keep modules loosely coupled and focused on specific concerns

Common configuration in the parent to ensure consistency

This enables independent module development and easier maintenance

Ans 14 A Java Servlet's life cycle is managed by the servlet container and includes the following key stages:

1) Initialization (init() method): called once when servlet is first loaded into memory. Used to perform one-time setup tasks

2) RequestHandling (service() method): called each time a client sends a request to the servlet. Handle request ~~type~~ processing and generate a response

Termination (destroy() method): Called once when the servlet is about to be unloaded. Used to release resources like database connections or file handles.

Ans 15 Thread Safety Problems in Servlet with Shared Resources:

In servlets, a single instance handles multiple requests concurrently by using multiple threads. If shared resources like instance variables are accessed by multiple threads without proper control, data inconsistency and race condition can occur.

```
pubic class CounterServlet extends HttpServlet {
    private int counter=0;
    protected void doGet(HttpServletRequest request,
            HttpServletResponse response) throws
            IOException {
        counter++;
        response.getWriter().println("counter: "+
                                    counter);
    }
```

Ans 11: The Singleton Pattern ensures that:

i) Only one instance of a class is created

ii) It provides a global access point to that instance

In many applications, some classes should only have one instance, such as: Logging, Configuration settings, Database Connections, Thread Pool. Without Singleton multiple objects may be created unnecessarily.

Basic Singleton:

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

Ensures one instance, but not thread-safe

Thread-safe Singleton:

```
public class Singleton {
    private static Singleton instance;
    private Singleton() { }
```

```
public static synchronized Singleton getInstance(){
    if (instance == null){
        instance = new Singleton(); // Safe for multithreading
    }
    return instance;
}
```

Thread-saf but Synchronization is slow.

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){
                    intance = new Singleton; // Efficient and thread safe
                }
            }
        }
        return instance;
    }
}
```

Ans 18

Method	How it works	Advantages	Limitations	Ideal use case
Cookies	Small data stored on clients browser sent with each request	- Easy to use - Persistent across browser sessions - Supported widely	- User can disable cookies - Limited size - Privacy concern	Tracking user preferences, remembering login in
URL Rewriting	Session ID appended as parameter to URLs	- work even if cookies are disabled - simple to implement	- URLs look cluttered - user must navigate via rewriting URLs - Security risk if session ID exposed	Applications where cookies are disabled or unreliable
Http Session	Server-side session object identified by Session ID	- Secure and stores large amounts of data - Session managed on server - Easy to use	- Consumes server memory - Requires cookies support or URL rewriting fallback	Applications needing secure, complex session management

Ans 19 Session works accross multiple requests:

i) User logs in → server creates a session and assigns a session

ii) Client stores the session ID cookie

iii) Client requests send the session ID cookie each time

iv) Server uses session ID to retrieve data from session object

Session Timeout and Invalidation:

i) Sessions have a timeout period set by the server

ii) If the user in inactive longer than this, the session is automatically invalidated to free resources and protect security

iii) Developers can manually invalidate sessions when needed

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

iv) On timeout or invalidation, session data is cleared, preventing unauthorized access if the session ID is stolen

Security consideration:

- Use HTTPS to protect session ID from interception
- Regenerate session ID on login to prevent session fixation at
- Set appropriate session timeout to balance usability and secu

Ans 24: JPA manages Mapping Between Java objects and Relational table:

- JPA maps Java classes to database tables, and class fields to table columns, enabling ORM
- Developers define entities with annotation, JPA handles SQL generation and data persistence automatically

@Entity: Marks a Java class as a persistant entity mapped to a database table

@ID: Specifies the primary key field of the entity

@GeneratedValue: Indicates that the primary key value is automatically generated

```
import Jakarta.persistence.*;
@Entity
public class User {
    @ID
    @GeneratedValue(Strategy = GenerationType.IDENTITY)
     private Long id;
     private String username;
     private String email;
}
```

User class maps to a user table

id is the primary key, auto generated by database

JPA handles SQL operations like insert, update and select

1] Write a program to find the k^{th} smallest element in an Arraylist

```
import java.util.*;
public class kthSmallest {
    public static int findkthSmallest (List <Integer>list, int k){
        Collection.sort(list);
            return list.get(k-1); }
    public static void main (String[] args) {
        List <Integer> numbers = Array.asList (7, 10, 4, 3, 20
        int k=3;
        int result = findkthSmallest (numbers, k);
        System.out.println ("The "+k+"rd smallest elemen
                            is: "+result); }
}
```

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

```
import java.util.*;
public class WordFrequencyMap {
    public static void main (String args[]) {
        String text = "hello world";
        String[] words = text.split (" ");
        TreeMap <String, Integer> frequencyMap = new
                                    TreeMap() for
                                (String word: words)
```

```
{ frequencyMap.put(word, frequencyMap.getOrDefault(word,0)+1);
System.out.println("Word frequencies:");
for(Map.Entry<String,Integer>entry:frequencyMap.entrySet()){
    System.out.println(entry.getKey()+":"+entry.getValue());}
}
```

3] Queue and stack using priority Queue with a custom compator

```
import java.util.*;
public class QueueStackPQ{
    public static void main(String[] args){
        PriorityQueue<Integer> queue = new PriorityQueue<>();
        queue.add(3);
        queue.add(1);
        queue.add(2);
        System.out.println("Queue(ascending order):");
        while(!queue.isEmpty()){
            System.out.println(queue.poll()+" ");}
        PriorityQueue<Integer> stack= new PriorityQueue<>(
                                    Collection.reverseOrder());
        stack.add(1);
        stack.add(2);
        stack.add(3);
        System.out.println("\n stack(deceending order):");
        while(!stack.empty()){
```

```
        System.out.println (Stack.poll () + " "); }

    }
}
```

4. TreeMap to store student IDs and their details.

```
import java.util.*;
class Student {
    String name;
    int age;
    Student (String name, int age) {
        this.name = name;
        this.age = age; }
    public String toString () {
        return name + " (" + age + ")"; }
}
public class StudentMap {
    public static void main (String [] args) {
        TreeMap <Integer, student> Student = new
                                        TreeMap <>();
        Students.put (101, new Student ("Adam", 20);
        Students.put (102, new Student ("Arin", 21);
    for (Map.Entry <Integer, student> entry . students.
                                        entrySet ()) {
        System.out.println ("ID " + entry.getkey () + " Details:
                "+entry.getvalue ()); }
}
```