Name: Abdur Rahim

ID: IT22031

Subject: ~~Software Project Ma~~ SPEM

Department: ICT

Answer to the question no: 1

In ~~java~~ java protected members of a class can be accessed within the same pakage by any class.

Example:

```
// Parent class
Package packageone;

Public class Parent {
    Protected String message = "Hello from Parent";
}

// child Same class.
Package packageone;

Public class childSamePackage extends Parent {
    System.out.println(message);
}

Public static void main (String[] args) {
    childSamePackage child = new childSamePackage();
    child.showMessage();
```

When the child class is different package it can access the protected field via inheritence. It can not access the protected field via a parent object reference.

Example:

Parent ~~Java~~ in Package packageone

```
package packageone;
Public class parent{
    Protected String messege = "Hello ~~world~~ world!"
    }
```

child in different package packagetwo

```
Package two
import Packageone.parent;
public class child extends to parent{
    Public void showMessege(){
    System.out.println(this.messeage);
    }}
Public static void main (String[] args){
    child child = new child()
    Child.showMesseage();
}}
```

Chid extends parcents.. Inside the child class this.messege works. It refers to the inherited field from Parent. This is called via inheritence

Answer to the question no: 3

Encapsulation is a core principle of object-oriented programming (OOP) that helps ~~enforce~~ enforce data ~~secu~~ security and integrity by:

1. Restricting direct access to an object's internal data using private fields.
2. Allowing controlled access via public methods (getters/ setters) that validate input before changing state.

Example:

Bank Account using Encapsulation:

```
Public class BankAccount {
    private String Account_number
    Private double balance
```

```
Public void setAccountNumber ( string accountNumber) {
 if (accountNumber == null || accountNumber.trim().isempty())
 {
   System.out.println ("Error: Account number can't be
                        Empty or null");
 }
 else {
      this.accountNumber = accountNumber;
      System.out.println ("Account number set to: "
                          + this.accountnumber)
 }}
Public void setInitialBalance (double initialBalance)
 if (initialBalance <0 ) {
    system.out.println ("Balance can't be negetive");
 }
 else {
      this.balance =initialBalance;
      System.out.println ("Initial balance set to: $"
                          +this.balance);
 }
}
Public String getAccountNumber () {
    return accountnumber;
 }
```

```
Public double getBalance(){
    return balance;
}
}
```

Main class :

```
Public class Test {

    Public static void main (String[] args) {

        BankAccount account = new BankAccount();
        account.setAccountNumber ("Acc123a3");
        account.setInitialBalance (1000);
        System.out.println ("Final Account Number: " + account.getAccount
                                                        Number());
        System.out.println ("Final Balance : $" + account.getBalance());
    }
}
```

* Prevent direct outside access private fields.
* Validate set methods, Ensure only valid values are set.
* Controlled access; maintain object state integrity.

Answer to the question no: 2

~~Compara~~

Comparison between Abstract class and Interfaces Interms of Multiple Inheritanee and use cases.

Featuree	Abstract class	Interface
Supports multiple Inheritenee	No (can only extend One class).	Yes (can Implement multiple Interface)
~~Can~~ extend / implement multiple	Can extend 1 class	Can Implement many interface
Common for use in,	Shared base functionality	Shared capability

Example:

```
abstract class Animal {
    abstract void makeSound();
    void eat() {
        System.out.println("Animal is eating");
    }
}

interface Flyable {
    void fly();
}

interface Swimmable {
    void swim();
}

class Duck extends Animal implements Flyable, Swimmable {
    @Override
    void makeSound() {
        System.out.println("Quack");
    }

    @Override
    public void fly() {
        System.out.println("Duck is flying");
    }

    @Override
    public void swim() {
        System.out.println("Duck is swimming");
    }
}
```

When to use abstract class vs ~~Intef~~ Interface

Use Case	Prefer Abstract class	Prefer Interface
Shared base class	When objects have a common parent	When object have a common behaviour
code reuse	You want to provide default implementations or shared fields	You only want to define metho signatures
Extensibility	When evolving a base class hierarchy	When defining a contract multiple classes must follow
Single inheritance	You can only extend one abstract class	You can implement mar interface

Answer to the question no: 4

(i) Find the kth smallest element

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

[Output: 4]

(ii) Word frequency using tree map

```
import ~~java~~ java.util.*;
public class wordfreq {
public static void main (String[] args) {
String text = "apple, banana apple mango";
TreeMap <string, Integer> map = new. TreeMap <>();
for (string word : text.split (", "))
map.put (word, map.get or default (word, 0)+1);
```

map.forEach (k,v) -> system.out.println (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; }}
  public static void main (String[] args) {
  priority queue <Element> stack = new PriorityQueue<>
((a,b) -> b.order - o.order);
Priority Queue < Element > Queue = new PriorityQueue <>
(comparator.company 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);
}
}
```

Answer to the question no: 5

Multithread based project

```
import java.util.*;
import java.util.concurrent.*;
Class Parkingpool {
 private final Queue <string> queue = new LinkedList<>();
 public syncronized void addCar (String car) {
 Queue.add (car)
 System.out.println ("Car "+ car + "requested Parking");
 notify (); }
 Public synerconized string getCar () {
 while (queue.isemptyisEmpty ()) {
   try {wait ();}
   catch (interrupted Exception) {}
    return queue.poll ();
 }
}
```

```
class RegisterParking extends Thread {
Private final string carNumber;
Private final ParkingPool pool;
Public RegisterParking (String carNumber, Parking
                                    pool .pool) {
this. carNumber = carNumber.
this. pool =pool; }
public void run () {
pool. oddcar (carNumber); }}
class ParkingAgent extends Thread {
private final ParkingPool pool
private final int agentId;
public ParkingAgent (ParkingPool pool, int agent.Id){
this. pool = pool;
this. agentId = agentId; }
public void run () {
while(true) {
  String car = pool. getCar ();
  System. out. println (" Agent " + agentId + " Parked car
                          + car + ". ");
try { Thread. sleep (500); }
catch (Interrupted Exception e) { } } }
```

```
Public class CarParkingSystem {
Public static void main (String [] arg) {
ParkingPool pool = new ParkingPool ();
new ParkingAgent (pool, 1). start ();
new ParkingAgent (pool, 2). start ();
new RegisterParking ("xyz456", pool). start ();
```

Output:

Car ABC123 requested parking
Car XYZ456 requested Parking
Agent 1 Parked car ABC123
Agent 2 parked car XYZ456

Answer to the question no: 6

Comparison between DOM vs SAX

Feature	DOM	SAX
Memory	High (loads whole XML)	Low (reads line by line
Speed	Slower for big files	faster for large file
Navigation	Easy (Tree structure)	Hard
Modification	Yes	No
Best for	Small xml editing	Large xml

Answer to the question no: 7

How does the virtual DOM in React improve performance

=>

React creates a virtual copy with DOM. On update, React:

(i) Compares (diffs) old and new virtual DOM.

(ii) Finds whats changed.

(iii) Applies only the changes to real DOM

Example:

Simple component update

```
function Greeting ({name}) {
  return <h1> Hello, {name}! </h1>;
```

Initial Render

```
<Greeting name = "Alice" />
```

Virtual DOM

```
{
  type: 'h1',
  Props: {
    children: ['Hello', 'Alice', '!']
```

Answer to the question no: 8

Event delegation in JavaScript

A techniques where a single event listener in attached to a parent element to element to handlen exception 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.destroy.tagname === "LI"){
  alert("Clicked: " + e.target.innerText);
  }});
```

Answer to the question no: 9

Explain how java Regular Expression can be used for input validation.

⇒ Java Provides the pattern and Matches classes in the java.util.regex package to work with regular expressions.

```
import java.util.regex.*;
Public class Email validator{
Public static void main (string [] args){
string email = "User@example.com.
string regex = "^[\\w-]+@[\\w.-]+\.
                [a-zA-Z]{2,6}$";
Pattern pattern = Pattern.compile(regex);
Matcher matcher = pattern.matcher(email);
if (matcher.matches()){
   system.out.println ("Valid email");
else{
   system.out.println ("Invalid email.");
}}
```

Answer to the question no: 10

Custom annotations allow to define metadata for the javacode.

Defining a custom Annotation:

```
import java.lang.annotation.*;
@Retention (RetentionPolicy.Runtime)
@ Target (ElementType.Method)
Public @interface My annotation {
  String value();
}
```

Using the annotation:

```
Public class TEST{
  @MyAnnotation (value = "example")
  Public void display(){
    System.out.println ("Display method");
  }}
```

Processing the Annotation with Reflection:

```
import java.lang.reflection.*;
Public class AnnotationProcessor {
Public static void main (String[] args)
      throws exception {
  for(method method: Test.class.getDeclare.methods()) {
```

```
if (method.isAnnotationPresent(MyAnnotation.class)) {
    MyAnnotation annotation = method.getAnnotation
                              (MyAnnotation.class);
    System.out.println("Annotation value: " + annotation
                       .value());
```

Answer to the question no: 11

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

It solves:

(i) Prevents multiple instances.

(ii) 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;
  }
```

~~if (method.isAnnotationPresent(MyAnnotation.class)) {~~

~~MyAnnotation annotation = method.getAnnotation~~
~~(MyAnnotation.class);~~

~~System.out.println("Annotation value: " + annotation.~~
~~value());~~
~~}}}~~

~~Answer to the question no: 11~~

```
Thread - safe Singleton {
Private Singleton getInstance () {
   if(instance == null ) {
      synchronized (singleton.class){
   if (instance == null) {
      instance = ~~new~~ new Singleton ();
   } } }
   return instance;
} }
```

Answer to the question no: 12

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

Step to Execute a SELECT Query:

```
import java.sql.*;
Public class DBExample {
  Public static void main (String [] args) {
  String url = "jdbc:Jdbc:MySQL://localhost:3306/mydb";
  String user = "root"
  String pass = "Password";
```

```
Connection conn=null;
Statement stmn=null;
Result set rs=null;
try {
conn = DriverManager.getconnection (url.user.pass);
stmn: conn.create Statement();
rs = stmn.execute query ("Select * from employees")
while (rs.next()) {
 System.out.println ("Name.11 +rs. getstring (Name);
}
  catch (SQLExeeptione){
    system.out.println( "Error "+e.getmessege);
 }
 Finally{
  try {
   if (rs! =null) stmt.close();
if (stmt! =null) stmt.close
if (conn! =null) conn.close();
}
 catch (SQLExeeption ex){
    ex.printStackTrace();
}
```

Answere to the question no: 13

Servlet acts as the controller, JSP is the view and java class in the model in a javaEE web application using the MVC architecture.

How they work together:

(i) Client sends an HTTP request to the servlet
(ii) Servlet process the request.
(iii) Jsp retrives the data and renders the HTML response to the client.

Sample usecase: Display user profile.

Model class User.java

```
Private string Name;
Private string email;
Public User (string name, String email){
    this.name = name
    this.name = email.
}
Public string getName () { return name; }
Public string getEmail () { return email; }
```

Controller: User.java

```
import Java.Io.*;
import Javax.servlet.http;
Public class UserServlet extends HttpServlet {
```

```
protected void doget (HttpServletRequest request,
                      HttpServletResponse response)
throws ServletException, IOException;
User User = new user ("Abdur Rahim", rahim@example.com);
request.setattribute("User", user);
RequestDispatcher dispatcher = request.getrequestDispatcher
dispatcher.forward (request, response, (Profile.jsp));
}
```

Profile.jsp

```
<%@page import = "your package, user" %>
<% User user = (User) request.getAttribute ("User"); %>
<html>
<head> <title> User profile </title> </head>
<body>
<h2> User profile </h2>
<P> Name <%= user.getName() %> </P>
<P> Email : <%= user.getEmail() %> </P>
</body>
</html>
```

Answere to the question no: 14

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

(i) **init() Method:** The init() Method is called once when the servlet is first baded into memory.

(ii) **Service() method:** The service method is called each time a client make a request.

(iii) **Destroy() method:** The destroy method is called once when the servlet is being removed from memory.

Answere to the question no: 15

In a servlet, the container creates only one instance, and multiple requests are handled using separate threads. If shared resources are accessed or modified by these threads simultaneously, it can lead to thread safety issues like.

- Race condition
- Incorrect data.
- Unpredictable behaviour.

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

```
Public class CounterServlet extends HttpServlet {
 private int cnt = 0;
 protected void doget (HttpServletRequest req, HttpServletResp
 throws ServletException, IOException{
  counter++;
 res.getwritter() println ("Visitor numbers" + counter
 }}
```

Answer to the question no: 18

Aspect	Cookies	Url Rewritting	Httpsession
1. Where 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	Limited	Large
Implement	~~Mediu~~ Medium	High	low