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.

= In Java a protected member is a variable, method or constructor. And these members can be accessed by -

i) Within same package any class can access protected members.

ii) In different package only subclass can access protected members and only through inheritence.

Let's look at both the scenarios →

Case 1: child class in same package.

File: Parent.java

```
package mypackage;
public class Parent {
    protected String message = "Hello from parent";
```

File: Child.java

```
package mypackage;
public class child extend Parent {
    public void showMessage() {
     System.out.println(message);
    }
  Public static void main(string[] args) {
```

```
        Child c = new Child();
        c.showMessage();
    }
}
```

Output:

```
Hello from Parent
```

Case 2: Child class in a different package.

File: Parent.java

```
package mypackage;
public class Parent{
    Protected string msg = "Hello I am Maisha";
}
```

File: Child.java [in another package]

```
package childpackage;
import mypackage.parent;
public class Child extends Parent {
    public void message(){}
        system.out.println(msg);
    }
    public static void main(String[] args){
        Child c = new Child();
        c.message();
    }
}
```

Output:

```
Hello I am Maisha
```

So, from the examples it is seen that protected members can be accessed both from the same package's subclasses and from different package's subclasses (through inheritence).

2. Compare abstract class and interfaces in terms of multiple inheritance. When would you prefer to use an abstract class and when an interface.

= In OOP, multiple inheritance means that a class can inherit from more than one parent class or multiple types.

Comparison between abstract class and interfaces in terms of multiple inheritance:

Feature	Abstract Class	Interface
Inheritance Type	Single inheritence only	Multiple inheritence is supported.
Keyword	abstract class	interface
Method	can have both abstract and concrete methods	can have abstract methods, default, static and private methods.
Constructor	Can have constructors	Can't have constructors
Access Modifier	Methods can be Public, protected, private or default.	All methods are implicitly public abstract.

Fields	can have instance variables and constants	only public static final constants.

When to use abstract class:

i) You want to share code (common implementation methods)

ii) You want to control access modifiers.

iii) You need constructors.

iv) When the classes are closely related.

When to use interface:

i) You need multiple inheritence

ii) You're defining capabilities.

iii) You're using Functional Programming patterns.

iv) When the classes can be unrelated.

3. How does encapsulation ensure data security and integrity ? Show with a BankAccount class using private variables and validate methods such as setAccountNumber(String), setInitialBalance(double) that rejects null, negative and empty values.

= Encapsulation a core principle of object oriented programming, enhances data security and integrity by bundling data and methods that operate on that data within a single unit and restrict direct access to the data.

It protects data from -

i) Direct unauthorized access
ii) Invalid or harmful input
iii) Inconsistent object states

Work process of encapsulation is given below with a BankAccount class :

Code :

```
public class BankAccount{
    private String accountNumber;
    private double balance;
public void setAccountNumber(String accountNumber){
    if (accountNumber==null||accountNumber.trim.isEmpty
                                                ()){
    System.out.println("Invalid number.");
}
```

```
    else{
        this.accountNumber = accountNumber;
    }
}

public void setInitialBalance (double balance){
    if (balance <0)
        System.out.println("Invalid balance!");
    else
        this.balance=balance;
}

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

public void deposit(double amount){
        if (amount <=0){
            system.out.println("Invalid amount");
        }
    else{
        balance += amount;
        system.out.println("Deposited:" +amount);
    }
}
}
```

```
public class BankApp {
    public static void main(String[] args){
    BankAccount acc = new BankAccount ();

    acc.setAccountNumber(" ");      # invalid
    acc.setAccountNumber("ACC2304"); # valid

    acc.setInitialBalance(-1000); # invalid
    acc.setInitialBalance(1000); # valid

    acc.deposit(-500); # invalid
    acc.deposit(500);  # valid

    System.out.println("Final Balance: "+acc.getBalance());
    }
}
```

So the benefit of using encapsulation here is:

i) private variables → Prevent direct external manipulation.

ii) Validate setters → Rejects bad input such as: null, empty, negative etc.

iii) Controlled methods → Ensures proper business logic (deposit).

4. Write Program to (Any 3) →

i) Find the kth smallest element in an ArrayList

Code:

```
import java.util.*;
public class KthsmallestElement{
   public class int KthSmallest(ArrayList<Integer>list,int K){
         Collections.sort(list);
         return list.get(K-1);
   }
public static void main (string[] args){
      Scanner sc= new Scanner(System.in);
   ArrayList<Integer>list= new ArrayList<>();
    int n= sc.nextInt();
   for(int i=0;i<n;i++)
      list.add(sc.nextInt());
   int K= sc.nextInt();
     int smallest = find KthSmallest(list,k);
     System.out.println("The"+K+"th"+"smallest element
                                  is:" + smallest);
     sc.close();
   }
}
```

ii) Create a TreeMap to store the mapping of words to their frequencies in a given text.

Code:

```
import java.util.*;
public class WordFrequencyTreeMap{
    public static void main (string[ ]args){
        string text= "I am Maisha";
        String [ ]words= text.split(" ");
      TreeMap <String;Integer>map= new TreeMap <>();
        for (String word: words){
            map.put(word, map.getOrderDefault (word,0)+1);
      }
    for (Map.Entry<String;Integer>entry:map.entryset()){
      System.out.println(entry.getKey()+" -> " +entry.getValue
                                                    ());
}   }   }
```

iii) Check if two linked lists are equal.

Code:

```
import java.util.*;
public class LinkedListEqualify{
  public static void main(String[ ] args){
    LinkedList <Integer>list1=new LinkedList<>(Arrays.asList
                                                (1,2,3));
LinkedList<Integer>list2=new LinkedList<>(Arrays.asList(1,2,3));
LinkedList<Integer>list3=new LinkedList<>(Arrays.asList(1,3,2));
System.out.println("list1==list2:"+list1.equals(list2));
```

```
        System.out.println("list1 == list3 : " + list1.equals(list3));
    }
}
```

6. How does java handle XML file using DOM and SAX parsers? Compare both approaches with respect to memory usage, processing speed and use cases. Provide a scenario where SAX would be prefferred over DOM.

XML stands for extensible Markup Language. It is a text-based format used to store, organize and transport data in a structured and readble way. It is used in-

i) Web services
ii) configuration files
iii) Data interchange between systems etc.

Java provides two main ways to parse and handle XML data → i) DOM parser
ii) SAX Parser

- Use of DOM parser to handle XML data:

i) Loads the entire XML file into memory and creates a tree structure.
ii) The tree can be traversed in any direction.

iii) Allows random access to nodes.

- Advantages of using DOM:

i) Easy to navigate and modify XML

ii) Best for small to medium sized XML files.

- Use of SAX Parser to handle XML data:

i) Reads XML sequentially using event driven call backs.

ii) Does not load the whole XML file into directory.

iii) Only forward traversal is possible.

- Advantages of using SAX:

i) Low memory usage.

ii) Very fast for large XML files.

Comparison between DOM and SAX is given below:

DOM Parser	SAX Parser
i) It loads entire XML into memory.	i) It processes XML sequentially don't load fully.
ii) Its processing speed is slower.	ii) Its processing speed is faster
iii) Access data randomly.	iii) Access data sequentially.
iv) Can read and modify XML.	iv) Only can read and process XML once.
v) Used when XML files are reasonably small.	v) Used when XML files are large.

Scenario when SAX is preffered over DOM:

Scenario:

You work in a library system that receives a daily XML file named books.xml. It contains thousands of books. You only need to print the titles of books published after 2015 and don't need to modify or store the full XML

For handling this problem SAX is best choice because:

i) It reads line by line, triggers an event when it finds <book> or <year>

ii) You can check the year, print the title if matches and move on.

iii) Fast and memory efficient.

7. How does the virtual DOM in React improve performance. Compare it with the traditional DOM and explain the diffing algorithm with a simple component update example.

React's Virtual DOM (VDOM) is a key innovation that boots performance and efficiency, especially for dynamic and interactive UIs. VDOM is a lightweight in-memory copy of the actual DOM. React keeps a virtual representation of the UI in javascript.

How Virtual DOM improves performance in React is given below:

1. Initial Render
2. State or Props update →

When a component's state change, React -

- Creates a new virtual DOM.
- Compares it to the previous VDOM.

3. Efficient Reconciliation :

React figures out →

- What's changed
- What needs to be added, updated or removed.

Comparison between VDOM and traditional DOM:

Feature	VDOM	Traditional DOM
Update Approach	Direct Use a virtual copy to compute minimal update	Direct DOM manipulation
Performance	Faster - because avoid unnecessary operations	slower
Re-rendering	Automatic and optimized through diffing	Manual updates or full re-renders.
Ease of maintenance	Easier	Harder for dynamic UIs

Explanation of diffing algo with a simple component update example:

React's diffing algo is a part of its reconciliation process.

When the UI updates, React:

1. Builds a new Virtual DOM.
2. Compare it with the previous VDOM.
3. Calculates the minimum set of changes needed.
4. Efficiently updates the real DOM based on the differences.

Simple component example:

initial state

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

→

```
{
  type: 'h1',
  props: {
    children: "Hello, Afita!"
  }
}
```

After state change

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

→

```
{type: 'h1',
 props: {
   children: "Hello, Tahsin!"
 }
}
```

8. What is Event Delegation in Javascript? and how does it optimize performance? Explain with an example of a click event on dynamically added elements.

Event delegation is a technique where a single event listener is added to a parent element, and it handles events for its child element using event bubbling.

Event delegation reduces/optimizes the performance by reducing the number of event listeners needed in a web page. Instead of adding individual listeners to each child element, a single listener is attached to a parent element. This take advantages of event budding, where events on child elements propagate up to their parents. When an event occurs on a child, parent's listener handles it, effectively managing event for all descendants.

Example of click event on dynamically added elements:

Context:

You have an empty <ul> list. Users can click a button to add new <li> items. When any <li> is clicked, you want to show an alert with the item's text. If you use event deligation, you add just one listener

to the <ul> but if you use no event deligation then, youll need to attach a click listener to every new <li>.

9. Explain how java regular expressions can be used for input validation. Write a regrex pattern to validate an email address and describe how it worcks using the Pattern and Matcher classes.

Java Regular expressions are now a powerful toolkit for input validation, allowing you to define patterns that strings must match to be considered valid. They're particularly useful for validating formats like email-address, phone numbers etc.

Java provides regex functionality through the java.util.regex package, primarily using two classes:

Pattern: Represented a compiled regular expression.
Matcher: Performs matching operations on a character sequence using a pattern.

Regex pattern for a basic email:

^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,6}$

Here,

^ → start of the string

[a-zA-Z0-9._%+-]+ → Local part

@ → must include @ symbol

[a-zA-Z0-9.-]+ → Domain name

\. → Dot (.)

[a-zA-Z]{2,6} → Top level domain (com, org)

$ → End of the string

How the regex pattern works using Pattern and Matcher Class is given below:

Pattern Class:

- Pattern.compile(regex) - Compiles regex into a pattern object.
- Pattern.matches(regex, input) → static method for one time matching

Matcher Class:

- matcher.matches() → Tests if entire string matches the pattern.
- matcher.find() → Finds the next occurrence of the pattern
- matcher.reset() → Resets the matcher to reuse it.

10. What are custom annotations in Java and how can they be used to influence program behaviour at runtime using reflection? Design a simple custom annotation and show how it can be processed with annotated elements.

Custom annotations are user defined metadata that can be attached to code elements. When combined with reflection they let your program →

i) Detect annotations at runtime
ii) Read their values
iii) Change behaviour dynamically

Step by step process of how custom annotation works is given below →

i) Define a custom annotation using @interface.
ii) Use @Retenation to make it accessible at runtime.
iii) Attach the annotation to classes, methods or fields.
iv) Use Java Reflection API to read and process the annotations.
v) Apply logic/behaviour based on the annotation value.

How custom annotation influence program behaviour at runtime using Reflection is given below:

1. Define a custom annotation

```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface RUNNow{
    int times() default;
}
```

2. Use the Annotation in a Class

```
public class MyTask{
  @RunNow(times = 3)
  public void printHello(){
    System.out.println("Hello");}
  public void notAnnotated(){
    System.out.println("This won't be called");}
}
```

3. Process it with reflection at Runtime

```
import java.lang.reflect.Method;
public class AnnotationRunner{
  public static void main(String[] args) throws Exception{
    MyTask task = new MyTask();
  for(Method method: task.getClass().getDeclaredMethods()){
```

```
if (method.isAnnotationPresent(RunNow.class)){
RunNow annotation= method.getAnnotation(RunNow.
                                        class);
  int times = annotation.times();
for (int i=0; i<times; i++)
    method.invoke(task);
} } } }
```

Output:

```
Hello
Hello
Hello
```

11. Discuss the singleton design pattern in java. What problem what does it solve and how does it ensure only one instance of a class is created? Extend your answer to explain how thread safety can be achieved in a Singleton implementation

The Singleton design Pattern is one of the most well known creational patterns that ensures a Class has only one instance throughout the application lifecycle while providing global access to that instance.

The Singleton Pattern addresses several key issues:

i) Preventing multiple instances

ii) Only one object is needed to coordinate actions across the system.

iii) Resource management.

The pattern uses several mechanism to guarantee that only one instance is created. They are -

i) Private constructor

ii) Static Factory Method

iii) Handling reflection attacks.

iv) Serialization safety

v)

To achieve thread safety in a Singleton implementation their are some methods:

1) Synchronized method

Code:

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

```
            instance = new Singleton();
        }
        return instance;
    }
}
```

2. Bill Pugh Singleton

Code:

```
public class Singleton{
    prive singleton(){}
private static class Holder{
    private static final Singleton INSTANCE=new
                                        Singleton();
}
public static Singleton getInstance(){
    return Holder.INSTANCE;}
}
}
```

12. Describe how JDBC manages communication between a Java application and a relational database. Outline the steps involved in executing a SELECT query and fetching results. Include error handling with try-catch and finally blocks.

JDBC is an API that allows Java applications with to interact with relational databases like MySQL, PostgreSQL and Oracle etc.

How JDBC manages communication:

JDBC acts as a bridge between Java and the database using:

i) DriverManager - loads the JDBC driver.
ii) Connection - estabishes the session
iii) Statement - sends SQL commands
iv) ResultSet - holds the data returned from queries.

Here's a concise outline of the steps to execute a SELECT query using JDBC with error handling:

1. Load the JDBC Driver
2. Establish a connection
3. Prepare SQL query.

4. Set parameters
5. Execute the query
6. Process the Resultset
7. Handle Exceptions
8. Closure resource in finally block.

Example of catch and finally block:

```
try {
      //set parameters, execute
   }
catch (SQLException e) {
      //handle sql errors
    }
finally {
      //close resultset, statement
   }
```

13. How do servlets and JSPs work together in a web application following the MVC architecture? Provide a brief use case showing the servlet as a controller, JSP as a view and a Java class as the model.

In a Java web application using MVC architecture, servlets, JSPs and Java classes each play distinct roles to cleanly seperate logic, presentation and data. How they work together is given below:

1. Client sends an HTTP request
2. Servlet receives the request
3. Extracts data from the request
4. Forward result to JSP for display
5. Model performs data processing
6. JSP (view) displays the result.

Use case:

Java class as model

UserValidator.java

```
public class UserValidator{
    public static boolean isValid(String username, String
                                  password){
        return username.equals("admin") && password.
                                  equals("1234");
    }
}
```

View using JSP

result.jsp

```
<%
Boolean isValid = (Boolean)request.getAttribute("isValid");
if(isValid != null && isValid) {
%>
  <h2> Welcome, admin! </h2>
<% } else {
  <h2> login failed. Try again. </h2>
<% } %>
```

Controller using servlet

LoginServlet.java

```
@WebServlet("/login")
public class LoginServlet extends HttpServlet {
protected void doPost(HttpServletRequest request,
        HttpServlet Response response)
throws ServletException, IOException{
String user = request.getParameter("username");
String pass = request.getParameter("password");
boolean isValid = UserValidator.isValid (user, pass);
request.setAttribute("isValid", isValid);
RequestDispatcher rd = request.getRequestDispatcher
                        ("result.jsp");
rd.forward(request, response); }
}
```

14. Explain the life cycle of Java servlet. What are the roles of init(), service(), destroy() methods? Discuss how servlets handle concurrent requests and how thread safety issues may arise.

The servlet life cycle defines the stages a servlet goes through from its creation to destruction in a servlet. The Javax.servlet package provides interfaces for handling these stages.

Roles of methods:

init()

→ Called once after the servlet is instantiated.
→ Used to initialize resources.
→ The ServletConfig object provides access to initialization parameters.

service()

→ Called every time the servlet receives a request.
→ It dispatches to doGet(), doPost() etc, depending on request method.

destroy()

→ Called once before the servlet is taken out of service.
→ Clean up resources like closing DB connections.

How servlet handles concurrent Requests is given below and also why thread safety matters -

Servlet Request Handling

- The servlet container (Tomcat) creats one instance of each servlet class.
- For every incoming request, the container spwans a new thread and invokes the servlets service method -
- This means many threads can be executing the servlet at the same time.

Thread safety concerns

Since multiple threads share the same servlet instance, any shared or mutable instance variables can be accessed by multiple threads concurrently. This can lead to -

- Race conditions
- Inconsistent results
- Unexpected behaviour.

15. A single instance of a servlet handles multiple requests using threads. What problem can occur if shared resources are accessed by multiple threads? Illustrate your answer with an example and suggest a solution using synchronization.

When multiple threads access shared resources then the given problems arises:

In a servlet one instance serves many requests simultaneously. The servlet container spawns a new thread for each request.

If instance variables are read or modified by these threads without coordination, problems arise - such as:

i) Race conditions → threads interface with each other unpredictably.

ii) Data inconsistency → variables can have unexpected values.

iii) Incorrect results → one user's data may leak into another user's response.

An example problem using race condition and its solution using synchronization is given below:

Example:

```
private int counter=0;
counter++;
response.getWriter().println("Request number:"+counter
```

In this problem 2 threads run nearly at the same time.

Solution:

To prevent this problem, we can use synchronization. Synchronization means access to the shared variable so only one thread at a time can execute that criticalsection.

```
synchronization(this){
 counter++;
response.getWriter().println("Request number:"+counter);
```

Here the synchronization –

- Locks the servlet instance before entering the block.
- Esures only one thread updates counter at a time.