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.

A Protected member of a Parent class can be accessed by its child class within the same Package directly. It can also be accessed by a child class in a differe Package through inheritance, but not through a Parent class object.

Same Package Example:-

```
class Parent {
    Protected int x = 10;
}
```

```
class Child Extends Parent {
 void show() {
 System.out.Println (x) ;    // allowed
 }
}
```

Different Package example :-

```
Package Pack2;
import Pack1.Parent;

class Child extends Parent {
void show() {
System.out.println (x);
}
}
```

Protected members are accessable in the same Package and in subclasses of other Packages, but not accessible using Parent objects outside the Package.

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

Multiple Inheritance :-

- A class cannot extend more than one abstract class, so abstract classes donot support multiple inheritance.
- A class can implement multiple interfaces, So, interfaces support multiple inheritance in java.

When we use an Abstract class :-

- when classes are closely related
- when you want to Provide some implemented methods.
- when you want to use instance variables and constructors.

When we use an interface:

- when unrelated classes need to follow a common contract.
- when multiple inheritance is required.
- when you want to achieve 100% abstraction (method declarations only)

use an abstract class for shared base behavior and an interface for defining a common capability accross multiple classes.

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

Encapsulation and Data security & Integrity:-

Encapsulation ensures data security and integrity by hiding data members using the private access modifier and allowing access only through public methods with validation. This prevents unauthorized or invalid modification of data.

Example :- BankAccount class

```
Public class BankAccount {
// Private data members (data hiding)
 Private String accountNumber;
 Private double balance;
// validation null or empty account number
Public Void setAccountNumber (String accountNumber){
if (accountNumber == null || accountNumber.trim(
   isEmpty()) {
System.out.Println ("Invalid account Number");
}
this.accountNumber = accountNumber;
}
// Validation negative Baiance
Public void setInitialBalance (double balance){
 if (balance < 0) {
  System.out.Println ("Balance cannot be negative");
return;
}
  this.balance = balance;
}
```

```
// getter methods
Public String getAccountNumber() {
    return accountNumber;
}

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

By restricting direct access to variables and enforcing validation through setter methods, encapsulation protects data (security) and maintains correct and consistent values (integrity) in a Program.

(4) Write Program to any three :

(i) Find the kth smallest element in an ArrayList

(v) Check if two linklist are equal.

(vi) Create a Hashmap to store the mappings of employee IDs to their departments.

(i)

```
import java.util.*;
Public class kthSmallest{
Public static void main (String[] args){
ArrayList<Integer> list = new ArrayList<>
(Arrays.asList(7,2,5,1,9));
int K = 3;
Collections.sort(list);
System.out.Println("kth smallest element :"
    + list.get(K-1));
}
}
```

(V):-

```
import java.util.*;
Public class LinkedListEqual{
 Public static void main (string[] args){
LinkedList<Integer> l1 = new LinkedList
<>(Arrays.asList(1,2,3));

LinkedList<Integer> l2 = new LinkedList
<>(Arrays.asList(1,2,3));

System.out.Println(l1.equals(l2)); //true
}
}
```

(vi)

```
import java.util.*;
Public class EmployeeMap{
 Public static void main (String[] args){
    HashMap <Integer, String> map = new HashMap<
        map.put (1, "HR");
        map.put (2,"IT");
        map.put (3, "Finance");
   System.out.println(map);
  }
}
```

⑤ Developing a multithreading based Project to simulate a car Parking management system with classes namely — Registrate Parking — Represents a Parking request made by a Car ; Parking Pool — Acts as a shared synchronized queue ; Parking Agent - Represents a thread that continuously checks the Pool and Parks Cars from the queue and a Main class — simulates N Cars arriving concurrently to request Parking.

Car ABC123 requested Parking
Car XYZ456 requested Parking.
Agent 1 Parked Car ABC123.
Agent 2 Parked Car XYZ456.

RegistrarParking :-

```
Class RegistrarParking {
String CarNO ;
RegistrarParking (string CarNO) {
 this.CarNO = carNO ;
System.out.Println ("Car" + CarNO + "requested Parkin
 }
}
```

ParkingPool (synchronized queue):-

```
import java.util.* ;
Class ParkingPool {
Queue<RegistrarParking> q =new LinkedList<>()
Synchronized void addCar(RegistrarParking c) {
 q.add(c) ; notify() ;
}
Synchronized RegistrarParking getCar () throw.
Exception {
 while (q.isEmpty ()) wait();
 return q.Poll();
 }
```

ParkingAgent (Thread):-

```
Class ParkingAgent extends Thread {
ParkingPool Pool;
ParkingAgent (ParkingPool P, String n) {
{super(n); Pool = P;}
Public void run(){
try{
  while (true)
    System.out.println(getName() + "Parked Car"
                       + Pool.getCar().carNo);

  } catch (Exception e){ }

}
}
```

Mainclass:-

```
Public class Mainclass {
Public static void main (String [] args);
ParkingPool P = new ParkingPool ();
new ParkingAgent (P, "Agent 1"). start ();
new ParkingAgent (P, "Agent 2"). start ();

P.addCar (new RegistrarParking ("ABC123"));
P. addCar (newRegistrarParking ("XYZ456"));
    }
}
```

Cars request Parking concurrently, a synchronized queue stores requests, and multiple agent threads park cars using the wait – notify mechanism.

⑥ How does Java handle XML data 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 preferred over DOM.

Java Processes XML using DOM and SAX Parsers.

- DOM Parser: Loads the entire XML document into memory as a tree. It use more memory, is slower for large files, but allows random access and modify of XML data.

- SAX Poster: Reads XML sequentially in an event-driven manner. It uses less memory, is faster, but does not allow modification

Comparrison :

DOM → high memory, slower, good for small XML files.

SAX → low memory, faster, good for large XML files.

SAX Preferred Scenario :-

SAX is preferred when Processing very large XML files such as server logs, where loading the entire document into memory is inefficient.

(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.

Virtual DOM is React improves Performance by minimizing direct manipulation of the real DOM, which is slow. React keeps a lightweight copy of the DOM in memory (virtual DOM). when the state changes.

1. Virtual DOM updates first.
2. Diffing algorithm compares the new virtual DOM with the previous version to find the minimal changes.
3. Only those changes are applied to the real DOM.

Example :-

jsk

// Before : <h1> Hello </h1>

// After state change : <h1> Hello, React ! </...>

- React updates only the text node instead of re-rendering the whole DOM.

Comparison :-

Feature	Traditional DOM	React Virtual Dom
Updates	Direct, slow	Batched & minimal
Performance	Low for many changes	High
Re-render	Entire subtree	only changed node

Key :- Virtual DOM + diffing = fewer costly DOM operations → better performance.

(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 Delegration :-

Event delegration is a technique where instead of attaching event listeners to individual child elements, you attach a single listener to a Parent element. The event bubbles up from the target to the parent, allowing you to handle events for current and future (dynamically added) child elements.

Why it improves Performances :-

- Fewer event listeners → less memory usage.
- Works for dynamically added elements without needing to re-attach listeners.

Example :

```
// Parent element
Cons list = document.getElementById('myList');
// Event delegation
list.addEventListener('click', function(event){
if (event.target && event.target.nodeName === 'LI'
console.log('Clicked item:', event.target.textContent
}
});
// Dynamically adding a new element
cost newItem = document.createElement('li');
newItem.textContent = 'Item 4';
list.appendChild(newItem);
```

(9) Explain how Java regular expressions 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.

Java regular expression for input validation :-

Regular expressions (regex) define patterns that input strings must match. In Java, you can use the Pattern and Matcher classes to validate input like emails, phone numbers, or Passwords.

Example :- Email validation

```
import java.util.regex.*;
Public class EmailValidation {
  Public static void main (String[] args) {
    String email = "user@ example.com ;
    String regex = "^[\\w.-]+@[\\w.-]+\\.[a-zA-Z]
                    {2,6}$ ";   // email pattern
```

```
Pattern pattern = Pattern.compile(regex);
Matcher matcher = pattern.matcher(email);
if (matcher.matches()) {
    System.out.println("valid email");
} else {
    System.out.println("Invalid email");
    }
}
```

How it works :-

1. Pattern.compile(regex) → Compiles the regex into a pattern.

2. Pattern.matcher(email) → Creates a matcher for the input string

3. matcher.matches() → Returns true if

Regex Breakdown :

- ^[\w.-]+ → start with letters, digits, -, . or - (username)
- @ → Literal @ symbol.
- [\w.-]+ → Domain name
- \.[a-zA-Z]{2,6}$ → Dot followed by 2-6 letters (TLD).

This ensures only properly formatted emails pass validation.

(10) What is custom annotations in Java, and how can they be used to influence Program behaviour at runtime using reflection? Design a simple custom animation and show how it can be Processed with annotated elements.

Custom Annotations in Java:

Custom annotations are user-defined metadata added to classes, methods, or fields. Using reflection, programs can read these annotations at runtime to influence behaviour, like invoking certain methods or enabling features dynamically.

Step1 :- Define custom annotation.

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

Step 2 : Apply annotation to methods

```
class MyClass {
@ RunMe
Public void test() {
System.out.Println ("Test executed" );
}
@ RunMe ("Custom run").
 Public void hello () {
System.out.Println (" Hello executed!" ); }
}
```

Step 3 :- Process annotation using reflection

```
import java.lang.reflect.*;
Public Class AnnotationDemo {
Public Static Void main (String [] args) throws Exception
for(Method m : MyClass.class.getDeclaredMethods ()){
if (m.isAnnotationPresent (RunMe.class.)) {
RunMe ann = m.getAnnotation (RunMe.class);
```

```
        System.out.println("Running: " + ann.value());
        m.invoke(new MyClass());
    }
}
}
```

Output:-

Running : Execute method.
Test executed
Running : Custom run
Hello executed.

Explanation:-

- @Retention(RetentionPolicy.RUNTIME) → annotation available at runtime.
- Method.isAnnotationPresent() → checks if annotation exists.
- method.invoke() → executes method dynamically based on annotation.

Use Case:- Can mark methods for testing, logging, or conditional runtime execution.