Rudra
IT-22002

1) Demonstrate how a child class can access a protected member of its parent class within the same package. Explain what happens when child class is in a different package.

Ans. In Java, the protected access modifier allows members to be accessed only within the same package and through subclasses even if they are under a different package.

When the child class is in the same package as the parent, it can directly access the protected members of the parent. Example:

```
package com.parent;

public class Parent {

    protected int protectedField = 20;

    protected void protectedMethod() {
        System.out.println("Protected members in parent.");
    }
}
```

~~package~~

Again when the child class is in a diffenent package, it can only access the protected members through inheritance. However, it cannot access them using a parent class reference. From the above example we get,

```
package com.child;
import com.parent.Parent;
public class Child extends Parent {
    public void accessProtected() {
        System.out.println(protected
                Field);
        protectedMethod();
    }
}
```

2) Compare abstract classes and interfaces in terms of multiple inheritance. When would you prefer to use an abstract class and when an interface? [Lab 1]

Feature	Abstract class	Interface
Multiple Inheritance	A class can extend only one abstract class	A class can implement multiple interfaces
Method implementation	Can have both abstract and concrete method	After Java 8+, it can have both default and static methods
Fields	It can have non-static fields	It can have public static final (constants)
Construction	Can have construction.	No constructors
Usage	Shared code and states among related classes	Multiple inheritance of type

Abstract classes are used for-

1) Shared code: If subclasses need common method implementations .. Example - Animal with eat(), sleep(), sound() etc.

2) State management: If subclasses need instance variables (e.g shape with color)

3) Construction logic: If initialization logic is needed.

Example:

```
abstract class Animal {
    String name;
    Animal(name) {
        this.name = name;
    }
    void sleep() {
        System.out.println(name + "is
                sleeping.");
        abstract void makeSound();
}
```

Interfaces are used for -

1) Multiple Behaviours: A class need to implement multiple contracts. Example - flyable, runnable etc.

2) Loose coupling: Focuses on what an object do- not how. Example: Runnable, Comparable.

3) Default methods: Since Java 8, interfaces can provide default implementations.

Example-

```
interface flyable {
    void fly();
    default void takeOff() {
        System.out.println("Taking off...");
    }
}

interface Swimmable {
    void swim();
}

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

3) How does encapsulation ensure data security and integrity? Show with a BankAccount class using private variables and validated methods such as setAccountNumber(String). [Lab 2]

Ans. In Java, encapsulation is a fundamental concept in object-oriented programming (OOP) that involves bundling variables and methods that operate on the data into a single unit or class. It restricts direct access in some of the object's components. Encapsulation predominantly involves and ensures:

1) Data security: It prevents unauthorized access or modification. By hiding the internal implementation encapsulation enhances the security, preventing unauthorized access to sensitive data.

2) Data Integrity: Ensures data remains valid and consistent. It promotes the use of modular code, allowing for reuse of code

in different parts of a program or future projects.

BankAccount Example:

```
public class BankAccount {

    private String accountNumber;
    private double balance;

    public void deposit(double amount) {
        if(amount <= 0) {
            throw new IllegalArgument
            Exception("Account must be
                positive.");
        }
        balance += amount;
    }

    public void withdraw(double amount) {
        if(amount <= 0) {
            throw new IllegalArgument
            Exception("
                Account should be positive.");
        }
        if(amount > balance) {
            throw new IllegalArgument
            Exception("
                Insufficient funds.");
        }
```

```
        balance -= amount;
    }
    public double getBalance(){
        return balance;
    }
}
```

Here, balance is private and cannot be modified directly. No direct balance = balance - 1000 is possible. Both methods validate amounts. This minimal version show core benefits of encapsulation using just two methods while maintaining all critical security and integrity checks.

4) Write program to any of -

i) Find the kth smallest element in an Array List

Ans.

```
//importing required classes from Java standard library
```

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Scanner;
public class KthSmallest {
  //Method to find kth smallest element from
    ArrayList
    public static int KthSmallest(ArrayList
            <Integer> Array, int x)
    {
        //Check for valid input
        if(x<0 || x>Array.size())
        {
          // Throw exception with a
             message
          throw new IllegalArgument
          Exception("Invalid value of
                    x.");
        }
        else{
          Collections.sort(Array); //sort
            in ascending order
```

```
        return Array.get(x-1); // Return (x-1)th
                               index value.
    }
}

public static void main(String[] args) {
    //Create scanner to read input
    Scanner sc=new Scanner(System.in);
    ArrayList<Integer> Array=new ArrayList();
    System.out.println("Enter the number of
                        elements:");
    int n=sc.nextInt(); //Read number of
                         elements.
    System.out.println("Enter"+n+"ele
                        ments:");
    for(int i=0;i<n;i++)
    {
        Array.add(sc.nextInt);
    }
    System.out.println("Enter the x th
                        element to be searched:");
```

```
int x= sc.nextInt();
try{
    int kthsmallest= KthSmallest(Array,x) ; //Call function
    System.out.println("The "+x+"th smallest element is: "+kthsmallest);
}
catch(IllegalArgumentException e) {
    System.out.println(e.getMessage()) ; //
        Display error message if an exception occurs.
}
}
}
```

Output:

Enter the number of elements :

5

Enter 5 elements:

10 9 8 5 6

Enter the xth smallest element to be searched:

4

The 4th smallest element is 9.

ii) Create a TreeMap to store the mappings of words to the frequencies in a given text

Ans.

```
import java.util.Map;
import java.util.Scanner;
import java.util.TreeMap;

public class FreqWords {

    //Function to count frequency of
      words

    static void count_freq(String str)
    {
      Map<String, Integer> mp = new
                TreeMap<>();

      //Splitting to find word

      String arr[] = str.split(" ");

      //Loop to iterate over words.
```

```
for(int i=0;i<arr.length;i++)
{
  //Conditions to check if array elements present
    in hashmap
  if(mp.containsKey(arr[i]))
   {
      mp.put(arr[i], mp.get(arr[i])+1);
   }
  else{
     mp.put(arr[i],1);
  }
}
//Loop to iterate over elements of map
for(Map.Entry<String, Integer> entry:
                mp.entrySet())
{
   System.out.println(entry.getKey()+
          "-"+entry.getValue());
  }
}
```

```
//Entry point of the program
    public static void main (String[] args) {
        Scanner sc=new Scanner(System.in);
        String str=sc.nextLine();
        //Function call
        count_freq(str);
    }
}
```

Output:

Software Engineering is the best passion in the world

Engineering-1

Software-1

best-1

in-1

is-1

passion-1

the-1

world-1

v) Check if two linked lists are equal.

Ans.

```
class Node{
    int data;
    Node next;
    Node(int data)
    {
        this.data=data;
        this.next= NULL;
    }
}
class IdenticalLinkedList{
    // Return true if two linked lists are
        equal.
    static boolean areIdentical(Node head1,
                    Node head2)
    {
        while(head1!=null && head2!=null) {
            if(head1.data!=head2.data)
                return false;
```

```
// Move to next nodes in both list
    head1 = head1.next;
    head2 = head2.next;
}

// If linked lists are identical, both head1
   and head2 must be null.
   return (head1 == null && head2 == null);
}

public static void main(String[] args) {
   // Create first linked list 3→2→1
   Node head1 = new Node(3);
   head1.next = new Node(2);
   head1.next.next = new Node(1);

   // Create second linked list: 1→2→3
   Node head2 = new Node(1);
   head2.next = new Node(2);
   head2.next.next = new Node(3);

   // Function call
if (areIdentical(head1, head2) != true)
```

```
{
    System.out.println("The linked lists are
        identical.");
}
else{
    System.out.println("The linked lists are not
        identical.");
}
}
}
```

Output:

The linked lists are not identical.

5) Develop a multithreading-based project to simulate a car-parking management system with classes namely- Registrar Parking- Represents a parking made by a car, Parking Pool acts as a shared synchronized queue, Parking Agent- represents a thread that continuously checks the pool and parks car from queue and a

Main Class - Simulates N cars arriving concurrently to request parking [Lab-3]

Ans. The code is given below!

```
import java.util.concurrent.*;

class RegistrarParking{
    private String carLicense;
    public RegistrarParking(String
                carLicense) {
        this.carLicense=carLicense;
    }
    public String getCarLicense(){
        return carLicense;
    }
}

class ParkingPool {
    private BlockingQueue<Registrar
        Parking> parkingQueue;
    public ParkingPool (int capacity){
        parkingQueue=new LinkedBlocking
                Queue<>(capacity);
    }
```

```
public void addParkingRequest(RegistrarParking
    request) throws InterruptedException{
    parkingQueue.put(request);
    System.out.println("Car" + request.getCar
        License() + "requested parking.");
    }

public RegistrarParking takeParkingRequest()
    throws InterruptedException{
        return parkingQueue.take();
    }
}

class ParkingAgent implements Runnable
{
    private ParkingPool parkingPool;
    private int agentId;
}
@Override
    public void run(){
    try{
        while(true){
```

```
RegistrarParking request = parkingPool.
        takeParkingRequest();
        parkCar(request);
    }
}
catch(InterruptedException e) {
    Thread.currentThread().interrupt
            ();
    }
}

private void parkCar(RegistrarParking
    request) throws InterruptedException {
    Thread.sleep(100);
    System.out.println("Agent" +
        agentId + "parked car" + request
            .getCarLicense() + ".");
    }
}

public class MainClass {
```

```
public static void main(String[] args) {
    int parkingCapacity = 10;
    int numberOfAgents = 2;
    int numberOfCars = 10;
    ParkingPool parkingPool = new ParkingPool
            (parkingCapacity);
    for(int i=1; i<=numberOfAgents; i++)
    {
        Thread agentThread = new Thread
            (new parkingAgent(parkingPool,
                i));
        agentThread.start();
    }
    for(int i=1; i<=numberOfCars; i++) {
        final String carLicense = generate
            LicensePlate(i);
        new Thread() -> {
            try {
                parkingPool.addParkingRequest
                (new RegistrainParking(carLicense));
```

```
        }
        catch(InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }).start();

    try {
    Thread.sleep((long) (Math.random() * 200));
    }
    catch(InterruptedException e) {
        Thread.currentThread().interrupt();
    }
  }
}

private static String generateLicensePlate
        (int index) {
    if(index==1) return "ABC123";
    if(index==2) return "XYZ123";
    return "CAR"+(100+index);
  }
}
```

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.

Ans. Java mainly provides two approaches for processing XML data namely: DOM (Document Object Model) and SAX (Simple API for XML). These parsers differ significantly in their architecture and use cases. Java handles using DOM and SAX parser in the following way:

DOM Parser:

1) Tree-based model that loads the entire XML document into memory.
2) Creates a hierarchial structure tree of nodes
3) Allows random access to any part of the document.
4) Supports both reading and writing XML.

SAX Parsen:

1) Event-based model that reads XML sequentially

2) Doesn't store entire document in the memory

3) Triggers events when encountering XML events.

4) Read-only access to XML.

Comparision of DOM and SAX in the terms of processing speed, memory usage and uses cases are shown below:

Feature	DOM Parsen	SAX Parsen
Memory usage	High (entire document in memory)	Low (stream processing)
Processing speed	Slower for larger files	Faster for larger files
Access pattern	Random access	Sequential access
Learning curve	Easier to use	More complex event handling
Use case	Small files	Large files

SAX is preferred more than DOM parsers due to following reasons:

1) Memory efficiency: SAX can process files much larger than the available primary memory.

2) Performance: Only processes relevant parts of the document

3) Early termination: Can stop processing once needed data is found.

Example:

XML file:

```
<products>
    <product>
        <id>1</id>
        <name>Laptop</name>
    </product>
    <product>
        <id>1000001</id>
        <name>Monitor</name>
    </product>
</products>
```

DOM approach:

```
import javax.xml.parsers.DocumentBuilderFactory;
import org.w3c.dom.Document;
public class DomParserExample {
    public static void main(String[]
        args) throws Exception{
    Document doc=DocumentBuilder
        Factory.newInstance().new
            DocumentBuilder().
            parse("large_data
                xml");
    System.out.println("DOM parsed
        successfully");
```

SAX approach:

```
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;
```

```
public class tinySAX {

    public static void main(String[] args) {

        SAXParserFactory.newInstance().
            newSAXParser().parse("data.xml",
                new DefaultHandler(){

                boolean inName;

                public void startElement(String u,
                    String l, String q, Attributes a) {

                    inName = q.equals("name");
                }

                public void endElement(String u,
                        String l, String q) {

                    inName = false;

                }
            });
    }
```

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

Ans. DOM (virtual) improves the performance of reactin the following way:

1) Minimizes different direct DOM operations that are expensive.

2) Batches updates to reduce reflow.

3) Efficient diffing algorithm determines exactly what exchanged.

4) Reconciliation process updates only necessary parts of DOM.

5) Avoids unnecessary rerendering of the unchanged components.

Comparision between virtual and traditional DOM is mentioned below:

Feature	Virtual DOM	Traditional DOM
Type	Lightweight JS Object	Browser's actual DOM tree
Updates	Fast (in-memory)	Slow (triggers reflow)
Manipulation	Changes first apply here	Directly modified
Performance	Optimized by diffing and batching	Expensive per operation
Example Update	Only updates changed nodes	Re-renders entire subtree if changed

Diffing Algorithm follows the below principles:

1) Tree Diffing: Compares tree hierarchially

2) Component-level: If component type changes entire subtree is rebuilt

3) Element-level: For elements of same type only attributes/children are compared

4) Keys: Uses keys to match children in the lists efficiently.

Example of single component update:

```
function UserProfile({name, age}) {
    return (
        <div className="profile">
            <h2>{name}</h2>
            <p>Age: {age}</p>
        </div>
    );
}

<UserProfile name="Atif" age={23} />
```

After update:

```
<UserProfile name="Atif" age={24} />
```

Here, React create a new virtual DOM tree with updated props. It compares with