IT23024

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

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

i) Within same package any class can access protected member

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

Let's look at both the scenarios →

Case1: Child class in same package

File: Parent.java

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

File: child.java

```
package mypackage;
  public void showMessage(){
      system.out.println(message);
  }
public static void main(String[] args){
      child c = new child();
      e.show message();
  }
}
```

Output:

Hello from parent

case2: child class in a different package

File: parent.java

```
package mypackage;
public class parent{
  protected String msg = "Hellow I am Mimi";
}
```

IT23024

File: child.java [In another package]

```
package child package;
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:

Hellow I am mini.

So, from the example it is seen that protected member can be accessed both from the same package subclasses and from different package's subclasses.

IT23024

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

Ans: Comparison Between Abstract Classes and Interfaces with respect to Multple inheritance is given below:

Object-oriented programming provides abstract classes and interfaces to archive abstraction. Although both are used to defined abstract behavion, they differ significantly in terms of multiple inheritance and usage.

Abstract class

A class can inherit only one abstract class. Therefore abstract classes do not support multiple inheritance. This restriction exist to avoid ambiguity such as the diamond problem, where multiple paren classes contain the same method implementation.

Example:

```
class B extends A { }
class C extends A,D { }
```

IT23024

Interface

A class can implement multiple interfaces, which means interfaces support multiple inheritance. Since interfaces mainly define method declarations, conflicts on avoided.

Example:

```
class A implements X, Y, Z { }
```

Key difference between Abstract class and Interface is given below:

Feature	Abstract class	Interface
multiple inheritance	Not supported	Supported
Method implementation	Allowed	Allowed using default method
Instance variable	Allowed	Not allowed
constructors	Allowed	Not allowed
Access modifier	Any	methods are default
keyword used	extends	implements

IT23024

When to use an Abstract class

An abstract class should be used when:

1. There is a clean IS-A relationship between classes
2. Common fields and method implementations need to be shared
3. classes are closely related
4. Future modifications to base class behaviours are expected.

Example:

```
abstract class Vehicle{
    int speed;
   void start(){
    System.out.println("Vehicle started");
   }
  abstract void move();
}
```

When to use an Interface

An interface should be used when:

1. multiple inheritance is required
2. Only behavior needs to be defined
3. Classes are unrelated but share common functionality.
4. A loosely coupled design is preferred

Example:

```
interface Flyable{
    void fly();
}
class Bird implements Flyable{}
class Airplane implements Flyable{}
```

IT23024

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

Answer:

Encapsulation is a fundamental concept of object-oriented programming that binds data (variables) and methods together into a single unit and restricts direct access to the data. This is achieved by declaring variables as private and accessing them only throw public, validat method.

Encapsulation ensures data security and data integri in the following way:

1. Data Hiding (security):
   i) private variables cannot be accessed or modified directly from outside the class.
   ii) This prevents unauthorized or accident change to sensitive data

2. Controlled Access
   i) public setter method validate input before assigning values.
   ii) Invalid data such as null, empty strings, or negative values are rejected.
   iii) This ensure the object always remains in a valid and consistent state.

3. Prevention of Invalid states
   i) Encapsulation enforces business rules.
   ii) Any attempt to break these rules is blocked by validation logic

Example: BankAccount class Demonstrating Encapsulation

```
public class BankAccount {

    private string accountNumber;
    private double balance;

    public void setAccountNumber (string accountNumber){
if( accountNumber == null || accountNumber.trim().is
Empty ()){
Throw new IllegalArgmentException ("Account number
cannot be null or empty.");
}
this.AccountNumber = accountNumber;
}
```

```
public void setInitialBalance(double balance){
  if(balance <0){
throw new IllegalArgumentException("Initial balance
cannot be negative");
   }
this.balance = balance;
}
public string getAccountNumber(){
    return accountNumber;
  }
public double getBalance(){
      return balance;
    }
}
```

How this example Ensure security and Integrity

1. The variables accountNumber and balance are private, so they cannot be accessed directly.
2. The method setAccountNumber() reject null on empty value, preventing invalid account data.
3. The method setInitialBalance() rejects negative values, ensuring financial correctness.
4. The Object's state can only be change through validation method, maintaing consistancy

I123024

5. Developing a multithreading-based project to simulate a car parking management system with classes namely. RegistarParking - Represents a parking request made by a car; paringPool - Aets as a shared synchronized queue; ParkingAgent Represents a thread that continously checks the pool and parks car from the queue and a MainClass - Simulates N cars arriving concurrently to request parking

car ABC123 requested parking.
Car XYZ456 requested parking.
Agent1 parked car ABC123.
Agent2 parked car XYZ456.

Answer:

To simulate a car parking management system using multithreading, where multiple cars request parking concurrent and multiple parking agents process these requests from a shared synchronized queue.

IT22024

System Design

1. Registran Parking
   i) Represents a parking request made by a car
   ii) stone a car number.

2. Parking Pool
   i) Acts as a shared synchronized queue
   ii) Holds pending paring requests
   iii) Ensure thread safety

3. Parking Agent
   i) Represent a thread
   ii) continously checks the pool
   iii) parks cars from the queue

4. MainClass
   i) Simulates N cars anniving concunnently
   ii) starts multiple paning Agents

IT23024

1. RegistranParking class

```
public class RegistranParking {
        private String CarNumber;
        public RegIstnan Panking (strings carNumber) {
      this.carNumber = carNumber;
    }
  public String getCarNumber (){
  return carNumber;
  }
}
```

2. ParkingPool class

```
import java.util.Linkedlist;
impont java.util.Queue;

public class ParkingPool {
    private Queue< RegistarParking > Queue = new LinkedList<>

  // Add car to parking request queue
  public synchronized void addRequest (Registranparking
  request) {
  queue.add(request);
  System.out.println ("Car " + request.getCarNumber()+"
    requested parking.");
  notifyAll();
}
```

```
// Remove can from queue for parking
public synchronized RegisteranParking getPeaod(){
    while (queue.isEmpty()){
      try{
        wait();
      } cateh (InternuptedException e) {
        e.printstackTnace();
      }
    }
    return queque.poll();
  }
}
```

3. parking Agent class

```
public class PankingAgent extends Thread {
    poivate PankingPool pool;
    poivate int agentId;

    public PankingAgent (PankingPool pool, int agentId
    this.pool = pool;
    this.agentId = AgentId;

  }
```

```
@Ovennide
public void nun(){
   while (true){
     RegistranParking request = pool.getRequest();
    System.out.printIn( "Agent " + agentId + " panked can"
    + request.getCanNumben() + ", ")

   try {
    thnead.sleep(1000); // simulate panking time
   } catch (IntennnuptedException e){
       e.printStackTnee()'
     }
   }
}
```

4. Mainclass

```
public class Mainclass{
      public static void main (stning[] angs){
        panking Pool pool = new PankingPool();

        // start panking Agents
        PankingAgent agent1 = new PaningAgent (pool,1);
        PankingAgent agent1 = new PankingAgent(pool,2);
```

```
        agent1.start();
        agent2.start();

        // simulate cars arriving concurrently
        String[] cars = { "AB123", "XYZ456", "DEF789",
          "LMN321" };

        for (String car : cars) {
            new Thread(() -> {
             pool.addRequest(new RegistrantParking(car));
            }).start();
            }
    }
}
```

Sample Output:

```
Car ABC123 requeted parking.
Car XYZ456 requested parking.
Agent 1 parked car ABC123.
Agent 2 parked car XYZ456.
Car DEF789 requested parking.
Agent 1 parked car DEF789.
Car LMN321 requested parking.
Agent 2 parked car LMN321.
```

IT23024

4. Write Program to -

i) Find the kth smallest element in an ArrayList

ii) Create a TreeMap to store the Mappings of words to their frequency in a given text.

iii) Implement a Queue and stack using the priority Queue class with custom comparator

iv) Create a TreeMap to store the mappings of student IDs to their departments

Answer:

Array list of kth element

```
import java.util.*;

public class KthSmallest {
    public static void main(String[] args) {
        ArrayList<Integer> list = new ArrayList<>();
        List.add(7);
        list.add(2);
        list.add(9);
        list.add(1);
        int k=3;
```

```
        collection.sort(list);
        system.out.println("Kth smallest element:"+list.get(k-1));
    }
}
```

11) Tree Map word frequency count

```
import java.util.*;
public class WordFrequency{
    public static void main(String[] args){
        String text="Java is easy and java is powerful";
        String[] words = text.split(" ");
        TreeMap<String, Integer> map = new TreeMap<>();
        for (String word: words){
            map.put(word, map.getOrDefault(word, 0)+1);
        }
        System.out.println("word frequency:");
        for(Map.Entry<String, Integer> entry: map.entrySe
            system.out.println(entry.getKey()+":"+entry.ge
    }
}
```

IT23024

iii) Using priority Queue stack and Queue implementation

```
import java.util.*;
class PQQueue{
    static class Element{
        int value, priority;
        Element(int value, int priority){
        this.value = value;
        this.priority = priority;
        }
    }
    static int count = 0;
    static PriorityQueue<Element> pq = new priority Queue<>(
        (a, b) -> a.priority - b.priority
    );
    static void enqueue (int x){
        pq.add (new element(x, count++));
    }
```

```
static int dequeue(){
    return pq.poll().value;
}
public static void main(String[] args){

    enqueue(10);
    enqueue(20);
    enqueue(30);
    System.out.println(dequeue());
    System.out.println(dequeue());
}
}
```