Object Oriented Programming (JAVA)

ID: IT23048

Q-01: Write a JAVA program that reads a series of numbers from a file (input.txt). Determine the ~~sur~~ highest number in the series, calculate the sum of ~~the~~ natural numbers up to that highest number and write the result in another file (output.txt). Use 'scanner' to read from the file and 'PrintWriter' to write to the file.

JAVA code to read a series of numbers -

```
import java.io.*;
import ~~util~~ java.util.*;

  public class NumberProcessor {
    public state void main (string [] args) {
      Scanner input = new Scanner (new File ("input.txt"));
      int max = 0;
      while (input.hasNextInt()) {
        int n = input.nextInt();
        if (n > max) max = n~~um~~;
      }
      input.dose();

      int sum = max * (max+1)/2;
      PrintWriter output = new PrintWriter ("output.txt");
      output.println(sum);
      output.close();
    }
}
```

Q-02: What are the differences you have ever found between static and final fields and methods? Exemplify what will happen if you try to access the static method/field with the object instead of class name.

Answer:

The differences I came across while coding in JAVA OOP between static and final fields and methods are given below.

Feature	Static	Final
i) Belongs to	class	Object (field) / Class (method)
ii) Accessed by	class name	Object reference
iii) Value change?	Yes, unless also final	No. (for fields)
iv) Inheritance	Can be inherited	Methods cannot be overriden
v) Instantiation needed?	No. (can be used without object)	Yes (for fields).

Example of Accessing a static field/method with an object:

```
class Example {
    static int statField = 10;
    static void statMeth() {
        System.out.println("static method called");
    }
}
public class Test {
    public static void main(String[] args) {
        Example obj = new Example();
```

Q-03: Write a JAVA program to find all factorion numbers within a given range. The program should take user input for the lower and upper bounds of the range

Q-04: Distinguish the differences among class, local and instance variables. What is significance of this keyword?

Answer: Differences among Class, Local and Instance Variables

Variable Type	Scope	Declared Inside	Memory	Default value	Modifier Allowed
Class (static)	Whole class	Class (outside methods)	Shared	Yes	Yes
Instance	Method/ Block	(Class Outside a method/block)	Separate for each object	Yes	Yes
Local	Method/ Block	Inside a method/block	Created / destroyed with method	No	No

Significance of this keyword:

- Refers to current object.
- Differentiate instance variables from local ones.
- Used to call constructors and methods in the same class.

Example:

```
class Example {
  int x;
  Example (int x) { this.x = x; }
}
```

Q-05: Write a JAVA program that defines a method to calculate the sum of all elements in an integer array. The method should take an integer array as a parameter and return the sum. Demonstrate this method by passing an array of integers from the main method.

Answer:

```
class SumofArray {
  static int sum (int[] arr){
    int total=0;
    for (int num: arr) total += num;
    return total;
  }
  public static void main (String[] args) {
    int [] numbers = {1,2,3,4,5};
    System.out.println ("sum: " + sum (numbers));
  }
}
```

Here,

1. Sum (int[] arr) method loops through the array and adds all elements.
2. Main method creates an array and calls sum() to calculate the total,
3. Output screen displays the sum of array elements.

Q-06: What is called Access Modifier? Compare the accessibility of Public, Private and Protected modifier. Describe different types of variable in JAVA with example.

Answer:

An access modifier controls the visibility of class members in object oriented programming.

Types of Access Modifiers:

1. Public: Accessible from anywhere (inside and outside the class).
2. Private: Accessible only within the same class.
3. Protected: Accessible within the same class and derived (subclass) classes, but not from outside.

Comparison of Accessibility:

Modifier	Same class	Subclass	Other classes
Public	Yes	Yes	Yes
Private	Yes	No	No
Protected	Yes	Yes	No

Example:

```
class MyClass {
public:
  int publicVar;

private:
  int privateVar;

protected
  int protectedVar;
```

```
int main () {
  MyClass obj;
  obj. publicVar = 5;
  return 0;
}
```

Q-07: Write a JAVA program to find the smallest positive root of a quadratic equation of the form — $ax^2 + bx + c = 0$ using quadratic formula : $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

Answer:

```
import java.util.Scanner;

public class QuadEquation {
  public static void main (String[] args) {
    Scanner read = new Scanner (System.in);
    System.out.println ("Enter a,b,c : ");
    int a = read.nextInt(), b = read.nextInt(), c = read.nextInt();
    double D = b*b - 4*a*c;
    if (D >= 0) {
      double root1 = -b + (Math.sqrt(D)) / (2*a);
      double root2 = -b - (Math.sqrt(D)) / (2*a);
      if (root1 > 0 && root2 > 0) {
        System.out.println ("Smallest positive root: " + Math.min(ro
      else if (root1 > 0)
        System.out.println ("Smallest positive root: " + root1);
```

```
        else if (root2 >0)
            System.out.println ("Smallest positive root : "+ root2);
        else System.out.println ("No real roots");
    }
    else System.out.println ("No real roots");
  }
}
```

Q-08: Write a program that can determine the letter, whitespace and digit. How do we pass an array to a function? Write an example.

Answer:

```
import java.util.Scanner;

public class CharChecker {
  public static void main( String [] args ) {
    Scanner read = new Scanner (System.in);
    System.out.print ("Enter a character : ");
    char ch = read.next().charAt(0);
    if ( ch >='A' && ch <= 'Z' || ch >='a' && ch <= 'z')
      System.out.println ("It is a letter.");
    else if (ch >= '0' && ch <='9')
      System.out.println ("It is a digit.");
    else if (ch == ' ')
      System.out.println ("It is a whitespace.");
    else
      System.out.println ("It is a special character.");
  }
}
```

In JAVA, arrays are passed to functions by reference. This means that when an array is passed as an argument, the function receives a reference (memory address) to the original array, not a copy of it. Any changes made to that array inside the function affect the original array.

A breakdown to how the process works —

i) Define the function (create function with array param

ii) Pass the array (as an argument)

iii) Transfer reference. The function receives a copy of the array's reference by memory address.

iv) Modify or use array using the function affecting the original array.

Here's a code to demonstrate the process:

```
public class ArrayPass {
  static void modify (int[] arr) {
    arr[0] = 100;
  }
  public static void main (String[] args) {
    int[] numbers = {1, 2, 3};
    modify (numbers);

System.out.println (numbers[0]);
  }
}
```

Q-09: In Java, explain how method overriding works in the context of inheritance. What happens when a subclass overrides a method from its superclass? How does super keyword help in calling the superclass method, and what are the potential issues when overriding methods, especially when dealing with constructors?

Answer:

In Java, method overriding ~~works~~ occurs when a subclass provides its own implementation of a method that is already defined in its superclass. The method in the subclass must have the same signature (name, return type and parameters) as the one in the superclass.

When a subclass overrides a method:

- The subclass method is called instead of the superclass method, even when using a reference of the superclass type.
- The super keyword is used to explicitly call the superclass' overridden method from within the subclass.

Potential issues with method overriding.

- Constructor overriding: Constructors cannot be overriden, but they can be called in subclasses using super() to invoke the superclass constructor.
- Method Signature conflicts: If the method signature does not match or is ~~sim~~ improperly overridden, it may lead to compile-time errors or unintended behaviour.

Q-01. Differentiate between static and non-static members including necessary examples. Write a program that is able to check either a number or string is palindrome or not.

Answer:

Static Members:

- Belong to the class, not instances of the class.
- Can be accessed without creating an object of the cl
- Shared among all instances of the class.
- Defined using the 'static' keyword.

Non-static Members:

- Belong to an instance of the class.
- Require creating an object of the class to be accessed.
- Each instance has its own copy of non-static members.

```
class Example {
    static int statVar = 10;
    int nonStatVar = 20;

    static void statMethod() {
        System.out.println("static method");
    }
    void nonStatMethod() {
        System.out.println("Non-static method");
    }
}
```

P.T.O.

```
public class Main {
  public static void main (String [] args.) {
    System.out.println (Example.statVar);
      Example.statMethod ();
      Example obj = new Example ();
System.out.println (obj.nonStatVar);
      Obj.nonStatMethod ();
  }
}
```

```
import java.util.Scanner;

public class Palindrome {
    public static void main (String[] args) {
        Scanner read = new Scanner (System.in);
        System.out.print ("Enter string : ");
        String s = read.nextLine ();
        System.out.println(str.equals (new StringBuilder (s).
                                reverse().toString()) ?
         "It is a palindrome." : "Not a palindrome");
        System.out.print ("Enter number : ");
        Int num = read.nextInt();
        System.out.println (num==new StringBuilder(String
         valueOf(num)).reverse().toString().toInt() ?
         "It is a palindrome." : "No a palindrome?");

        read.close();
    }
}
```

Q-11: What is called abstraction and encapsulation ? Describe with example. What are differences between Abstract class and Interface?

Answer:

Abstraction is the process of hiding implementation details and showing only the essential features of an object. It helps to reduce complexity and increase code reusability.

Example:

```
abstract class Bees {
   abstract void ~~start~~ honey();
}
class Extract extends Bees {
   void honey() {
      System.out.println("Honey is extracted from bees.");
   }
}
```

Here, Bees defines an abstract method honey(), hiding the internal details. The Extract class provides the specific implementation.

Encapsulation is the process of wrapping data (variables) and methods into a single unit (class) and restricting direct access to some of the object's components.

Example:

```
class Cat {
  private String name;
  public void setName (String name) {
     this.name = name ;
  }
  public String getName() {
     return name;
  }
}
```

Here, the variable name is private, and access is controlled through getter and setter methods.

Differences between Abstract Class and Interface

Feature	Abstract Class	Interface
Methods	Can have both abstract and non-abstract methods.	Only abstract methods (before Java 8).
Fields	Can have instance variables.	Only static and final variable
Multiple Inheritance	Cannot be implemented by multiple classes.	Can be implemented by multiple classes.
Default Implementation	Can have implemented methods.	Cannot have implemented methods (before Java 8).
Usage	Used for sharing code among related classes.	Used for defining a contract that multiple classes must follow

Q-12 : Create a Java program using inheritance to perform multiple numerical operations. Implement a BaseClass with common functionalities and extend it into four specialized classes to handle different tasks. Use a MainClass to execute all methods.

```
class BaseClass {
    void print (String label, String result) {
        System.out.println (label + ": " + result);
    }
}

class SumClass extends BaseClass {
    double computeSum () {
        double sum = 0;
        for (double i = 1.0 ; i >= 0.01 ; i -= 0.1)
            sum += i ;
        return sum;
    }
}

class DivisorMultipleClass Extends BaseClass {
    int gcd (int a, int b){
        return b==0 ? a: gcd (b, a%b);
    }
    int lcm (int a, int b) {
        return (a*b) / gcd (a, b);
    }
```

~~class~~

```
NumberConversionClass ncObj = new NumberConversionClass();
    int num = 29;
    ncObj.print("Binary", ncObj.toBinary(number));
    ncObj.print("Hex", ncObj.toHex(number));
    ncObj.print("Octal", ncObj.toOctal(number));
    new CustomPrintClass().PR("End of Program");
  }
}
```

```
class NumberConversionClass extends BaseClass {
    String toBinary (int num) {
        return Integer.toBinaryString(num);
    }
    String toHex (int num) {
        return Integer.toHexString(num).toUpperCase();
    }
    String toOctal (int num) {
        return Integer.toOctalString(num);
    }
}

class CustomPrintClass extends BaseClass {
    void PR (String message) {
        System.out.println("\033[1m" + message + "\033[0m");
    }
}

public class MainClass {
    public static void main (String[] args) {
        SumClass sumObj = new SumClass();
        sumObj.print("Series Sum", String.valueOf(sumObj.computeSum()));

        DivisorMultipleClass dmObj = new DivisorMultipleClass();
        int num1 = 36, num2 = 48;
        dmObj.print("GCD", String.valueOf(dmObj.gcd(num1, num2)));
        dmObj.print("LCM", String.valueOf(dmObj.lcm(num1, num2)));
```

~~Hemlock~~

Q-14: What is the significance of BigInteger? Write a program which gives a method that can return a factorial of any Integer. (For example 50)

Answer:

BigInteger is a data type provided in languages like Java to handle arbitrarily large integers. It allows performing arithmetic operations without worrying about integer overflow.

Code:

```
import java.math.BigInteger;

public class BigIntegerFactorial {
    static BigInteger factorial(int n) {
        BigInteger fact = BigInteger.ONE;
        for (int i = 2; i <= n; ++i) {
            fact = fact.multiply(BigInteger.valueOf(i));
        }
        return fact;
    }

    public static void main(String[] args) {
        ~~System.out.println(factorial(50));~~
        int n = 50;
        System.out.println(n + "!" = " + factorial(n));
    }
}
```

Q-15: Compare & contrast abstraction achieved through abstract classes and interfaces in Java. In what scenarios would you prefer to use an abstract class over an interface?
Can a class implement multiple interfaces in Java? Explain the implications of using multiple interfaces, especially when dealing with conflicting method signatures.

Answer:

Comparison between abstraction achieved through abstract classes and interfaces in Java:

Both abstract classes and interfaces achieve abstraction, which means -

i) They define what a class should do rather than how it does it.

ii) They allow for code reuse and polymorphism.

iii) They cannot be initiated directly and must be implemented by a subclass.

Aspect	Abstract Classes	Interfaces
1. Level of Abstraction	Support partial abstraction (can have both abstract and concrete methods)	Only support abstraction. (prior to Java 8)
2. Implementation	Allow method implementation (concrete methods).	No method implementation (After Java 8, default methods allow some implementation.
3. Flexibility	Less flexible; can only extend one abstract class.	More flexible; can implement multiple interfaces.
4. Encapsulation	Can have instance variables to maintain state, reducing abstraction.	Cannot have instance variables, ensuring pure abstraction.
5. Purpose	Used when some methods need a predefined implementation along with abstract ones.	Used when only the method contract is needed without enforcing implementation

Use an abstract class over an interface when —

1. Shared Behaviour & State :- You need to provide common functionality along with instance variables.

2. Partial Abstraction :- Some methods should have predefined implementations, while others remain silent.

3. Single Inheritance Suffices :- You don't need multiple inheritance (a class can extend only one abstract class).

4. Code Evolution :- Easier to add new methods without breaking existing subclasses.

5. Encapsulation : When you need constructors or protected/private methods to control behaviour.

A class can implement multiple interfaces in Java.

Implications of Multiple Interfaces —

1) Code Reusability — Allows a class to inherit multiple behaviors from multiple sources.

2) Avoids Diamond Problem — Java resolves ambiguity by requiring explicit method overrides.

3) Conflicting Method Signatures — If two interfaces have methods with the same signature but different implementations (default methods), the class must override and provide its own implementation to resolve conflicts.

4) Interface Segregation — Encourages breaking large interfaces into smaller, more specific ones for flexibility.

Using multiple interfaces improves modularity but requires careful conflict resolution.