|  |  |  |  |
| --- | --- | --- | --- |
| **Course Name:** | **Object Oriented Programming** | **Semester:** | **III** |
| **Date of Performance:** | **25 / 08 / 2023** | **Batch No:** | **Batch A - 3** |
| **Faculty Name:** | **Pragya Gupta** | **Roll No:** | **16014022050** |
| **Faculty Sign & Date:** |  | **Grade/Marks:** | **\_\_\_ / 25** |

|  |  |  |
| --- | --- | --- |
| **Writing Program (07)** | **Performance in lab**  **and Viva (05 + 03)** | **Post lab questions, conclusion and**  **Completion (03 + 02 + 05)** |
|  |  |  |

**Experiment No.: 5**

**Title: Use of Interface**

|  |
| --- |
| **Aim and Objective of the Experiment:** |
| Learn the how to use Interfaces. |

|  |
| --- |
| **COs to be achieved:** |
| **CO2:** Understand concepts of Object-Oriented Programming and basic characteristics of Java. |

|  |
| --- |
| **Tools used:** |
| 1. Java Development Kit (JDK) 2. Visual Studio Code 3. [Interfaces in Java - GeeksforGeeks](https://www.geeksforgeeks.org/interfaces-in-java/) |

|  |
| --- |
| **Theory:** |
| 1. (About Interfaces)   In Java, an interface is a reference type that defines a contract of methods that a class must implement. It provides a way to achieve abstraction and multiple inheritance in Java, allowing you to define a set of method signatures that classes can implement without specifying the actual implementation details. Interfaces are used to define common behavior that can be shared among unrelated classes.  Here are the key aspects of interfaces in Java:   1. **Declaring an Interface**: To declare an interface, you use the interface keyword followed by the interface name and its method signatures. An interface can include constant fields (implicitly public, static, and final) and abstract methods (implicitly public and abstract).   public interface MyInterface {  int CONSTANT\_VALUE = 10; // Constant field  void doSomething(); // Abstract method  void doSomethingElse(); // Abstract method  }   1. **Implementing Interfaces:** To implement an interface, a class uses the implements keyword followed by the interface's name. The class must provide concrete implementations for all the methods declared in the interface.   public class MyClass implements MyInterface {  @Override  public void doSomething() {  // Implementation  }  @Override  public void doSomethingElse() {  // Implementation  }  }   1. **Multiple Interface Implementation:** A class can implement multiple interfaces, separated by commas. This is Java's way of achieving multiple inheritance without the complexities that come with class-based multiple inheritance.   public class MyOtherClass implements Interface1, Interface2 {  // Implement methods from both interfaces  }   1. **Default Methods:** Java 8 introduced the concept of default methods in interfaces. Default methods have implementations provided within the interface itself. This allows you to add new methods to existing interfaces without breaking the classes that implement them.   public interface MyInterface {  void doSomething();    default void doSomethingNew() {  // Default implementation  }  }   1. **Static Methods:** Java 8 also introduced static methods in interfaces. These methods can be called on the interface itself and are not inherited by implementing classes.   public interface MyInterface {  static void staticMethod() {  // Implementation  }  }   1. **Functional Interfaces:** A functional interface is an interface with a single abstract method. Functional interfaces are used extensively in Java's lambda expressions and the java.util.function package. 2. **Marker Interfaces:** Marker interfaces are interfaces with no method declarations, used to indicate a certain capability or feature in classes that implement them. For example, the Serializable interface is a marker interface used to indicate that a class can be serialized. 3. **Access Modifiers:** All methods in an interface are implicitly public and abstract. Fields are implicitly public, static, and final.   Interfaces play a crucial role in Java's concept of polymorphism and abstraction, enabling you to design more flexible and modular code. They allow you to define a contract that classes must adhere to while promoting loose coupling between classes. |

|  |
| --- |
| **Code:** |
| 1. **Write a program to design interface named *stack* with the following methods**    1. ***push* and *pop* elements from the stack**    2. **check for stack *empty* and *full* using method**   interface Stack // interface named stack  {      void push(int element);      int pop();      boolean isEmpty();      boolean isFull();      void display();  }  class NewStack implements Stack  {      private int maxSize;      private int top;      private int[] stackArray;      public NewStack(int maxSize)      {          this.maxSize = maxSize;          this.top = -1;          this.stackArray = new int[maxSize];      }      @Override      public void push(int element)      {          if (!isFull()) {              stackArray[++top] = element;              System.out.println("pushed element: " + element);          } else {              System.out.println("stack is full, cannot push element");          }      }      @Override      public int pop()      {          if (!isEmpty()) {              int poppedElement = stackArray[top--];              System.out.println("popped element: " + poppedElement);              return poppedElement;          } else {              System.out.println("stack is empty, cannot pop element");              return -1;          }      }      @Override      public boolean isEmpty()      {          return top == -1;      }      @Override      public boolean isFull()      {          return top == maxSize - 1;      }      @Override      public void display()      {          System.out.println("\nstack elements: ");          for (int i = 0; i <= top; i++) {              System.out.print(stackArray[i] + "\n");          }          System.out.println();      }  }  public class exp5\_q1 {      public static void main(String[] args)      {          int maxSize = 5;          NewStack stack = new NewStack(maxSize);          stack.push(10);          stack.push(20);          stack.push(30);          stack.push(40);          stack.push(50);          stack.push(60); // trying add elements when stack is full          stack.display(); // displaying elements after pushing          stack.pop();          stack.pop();          stack.pop();          stack.pop();          stack.pop();          stack.pop(); // trying to pop when stack is empty      }  }   1. **Design class of *college* which uses two interfaces named *student* and *teacher*. Class *college* will use methods declared in interface to *display* student and teacher personal information.**   // student interface  interface Student  {      void displayStudentInfo();  }  // teacher interface  interface Teacher  {      void displayTeacherInfo();  }  // college class implementing both interfacess  class College implements Student, Teacher  {      private String studentName;      private int studentRollNo;      private String teacherName;      private String subjectTaught;      public College(String studentName, int studentRollNo)      {          this.studentName = studentName;          this.studentRollNo = studentRollNo;      }      public College(String teacherName, String subjectTaught)      {          this.teacherName = teacherName;          this.subjectTaught = subjectTaught;      }      // implementing displayStudentInfo method from Student interface      @Override      public void displayStudentInfo()      {          System.out.println("Student Name: " + studentName);          System.out.println("Student Roll Number: " + studentRollNo);      }      // implementing displayTeacherInfo method from Teacher interface      @Override      public void displayTeacherInfo()      {          System.out.println("Teacher Name: " + teacherName);          System.out.println("Subject Taught: " + subjectTaught);      }  }  public class exp5\_q2  {      public static void main(String[] args)      {          // student object          Student student = new College("ketaki mahajan", 050);            // teacher object          Teacher teacher = new College("mr. Patil", "mathematics");          System.out.println("Student Information:");          student.displayStudentInfo();            System.out.println();          System.out.println("Teacher Information:");          teacher.displayTeacherInfo();      }  } |

|  |
| --- |
| **Output:** |
|  |

|  |
| --- |
| **Post Lab Subjective/Objective type Questions:** |
| 1. **What is interface? How is it different from an abstract class?**   An interface and an abstract class are both Java constructs that facilitate the implementation of certain programming principles, but they serve different purposes and have distinct characteristics.  Interface:   * An interface is a reference type in Java, similar to a class, but it only contains method signatures and constants (static final fields). * It does not have any method implementations; it only provides a contract that the classes implementing the interface must adhere to. * All methods declared in an interface are implicitly public and abstract (no method body). * A class can implement multiple interfaces, allowing for a form of multiple inheritance. * Since Java 8, interfaces can have default methods (methods with implementations) and static methods. * Interfaces can be used to achieve loose coupling in the codebase by providing a common contract without dictating the internal implementation.   Abstract Class:   * An abstract class is a class that cannot be instantiated on its own; it's meant to be subclassed. * It can have both abstract methods (methods without implementations) and concrete methods (methods with implementations). * Abstract methods in an abstract class define a contract that subclasses must implement. * Abstract classes can have instance variables that are inherited by subclasses. * Java supports single inheritance only, meaning a class can extend only one abstract class. * Abstract classes can provide a mix of both concrete and abstract methods, which can be beneficial for creating reusable code.   In summary, interfaces are more focused on defining contracts that classes must adhere to, while abstract classes provide a base for inheritance and code sharing by combining abstract and concrete methods. Your choice between using an interface or an abstract class will depend on the specific requirements of your design and the level of code reusability you aim to achieve.   1. **Explain how multiple inheritance can be implemented using interfaces.**   Multiple inheritance in Java can be achieved using interfaces. While Java does not support multiple inheritance of classes (a class cannot extend multiple classes), it does allow a class to implement multiple interfaces. This allows a class to inherit behavior from multiple sources through interfaces. Here's how multiple inheritance is implemented using interfaces:  interface A {  void methodA();  }  interface B {  void methodB();  }  class C implements A, B {  @Override  public void methodA() {  System.out.println("Method A implementation");  }  @Override  public void methodB() {  System.out.println("Method B implementation");  }  }  public class MultipleInheritanceDemo {  public static void main(String[] args) {  C instance = new C();  instance.methodA();  instance.methodB();  }  }  Suppose we have two interfaces: A and B, each with its own set of methods. A class C can implement both interfaces A and B, inheriting the methods declared in both interfaces.  In this example:   * Interface A declares the method methodA. * Interface B declares the method methodB. * Class C implements both A and B interfaces, providing implementations for both methods. * The main method creates an instance of class C and calls both methodA and methodB.   By implementing multiple interfaces, class C effectively inherits the behavior specified by both A and B. This is a way to achieve multiple inheritance in Java without the issues associated with multiple inheritance of classes (like the diamond problem). Each interface serves as a contract that class C must fulfill, and the class can provide its own implementations for the methods declared in these interfaces.   1. **What will be function of the following code? will this code work? if not, why?**     The provided code snippet appears to have a few syntax errors and typos, which would prevent it from compiling and running successfully:   1. The code uses colons (:) instead of semicolons (;) in the loop constructs (for and if statements). 2. The code uses single quotes (') instead of double quotes (") to define strings. 3. In the second if statement, the opening curly brace { is missing.   Due to these syntax errors, the code would not run successfully. To make it run, the syntax errors need to be corrected, such as using semicolons instead of colons and fixing the quotes for string literals. Additionally, there are issues with using raw Vector without specifying its generic type and some typos that would need to be addressed. |

|  |
| --- |
| **Conclusion:** |
| In this experiment, we explored the practical usage of Java interfaces. Interfaces allow us to define contracts for classes, promoting flexible and modular code design through adherence to specified behaviors. This experience enhanced our understanding of how interfaces contribute to effective programming in Java. |

|  |
| --- |
| **Signature of faculty in-charge with Date:** |