1)

Prerequisites:

* Basic understanding of computer systems and operating systems.
* Access to a computer with a supported operating system (Windows, macOS, Linux).

Underlying Concept / Theory:

* Java: A popular programming language known for its platform independence and object-oriented nature.
* PATH variable: An environment variable that specifies the locations where the operating system should look for executable files.

Problem Statement:

To install Java and set up the PATH variables in order to compile and run simple Java programs.

Algorithm / Step-by-step instructions:

1. Determine the appropriate Java Development Kit (JDK) version to install based on your requirements and operating system compatibility.
2. Visit the official Java website (<https://www.oracle.com/java/>) or an alternative trusted source to download the JDK installer.
3. Run the JDK installer and follow the on-screen instructions to complete the installation process.
4. After installation, verify the Java installation by opening a command prompt (or terminal) and typing java -version. This should display the installed Java version.
5. Next, set up the PATH variable:
   * On Windows:
     + Right-click on "My Computer" or "This PC" and select "Properties."
     + Click on "Advanced system settings" or "Advanced" tab.
     + Click on "Environment Variables" button.
     + In the "System Variables" section, select the "Path" variable and click on "Edit."
     + Add the path to the JDK's "bin" directory (e.g., C:\Program Files\Java\jdk1.8.0\_XXX\bin) at the end of the existing value, separating each entry with a semicolon (;).
     + Click "OK" to save the changes.
   * On macOS and Linux:
     + Open a terminal window.
     + Edit the .bash\_profile or .bashrc file in your home directory using a text editor (e.g., nano ~/.bash\_profile).
     + Add the following line at the end of the file: export PATH="/usr/local/java/jdkXXX/bin:$PATH" (replace jdkXXX with the appropriate JDK version).
     + Save the file and exit the text editor.
     + In the terminal, run source ~/.bash\_profile or source ~/.bashrc to apply the changes immediately.

Output:

* Successful installation and configuration of Java.
* Verification of the installed Java version using java -version.
* Proper setup of the PATH variable.

Conclusion / Remarks:

Installing Java and configuring the PATH variables is crucial for developing and running Java applications. It allows the operating system to locate the Java compiler and runtime environment when executing Java programs. It's important to choose the correct JDK version and follow the platform-specific instructions to ensure a successful setup.

Exercises / Challenges:

1. Write a simple Java program that prints "Hello, World!" to the console.
2. Save the program in a file with a .java extension.
3. Compile the program using the Java compiler (javac).
4. Run the compiled program using the Java Virtual Machine (java).
5. Experiment with modifying the program and recompiling/running it to observe the changes.

CODE:

1. code

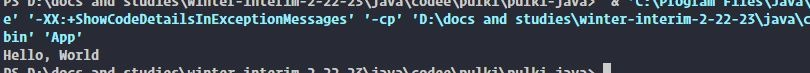
class App {

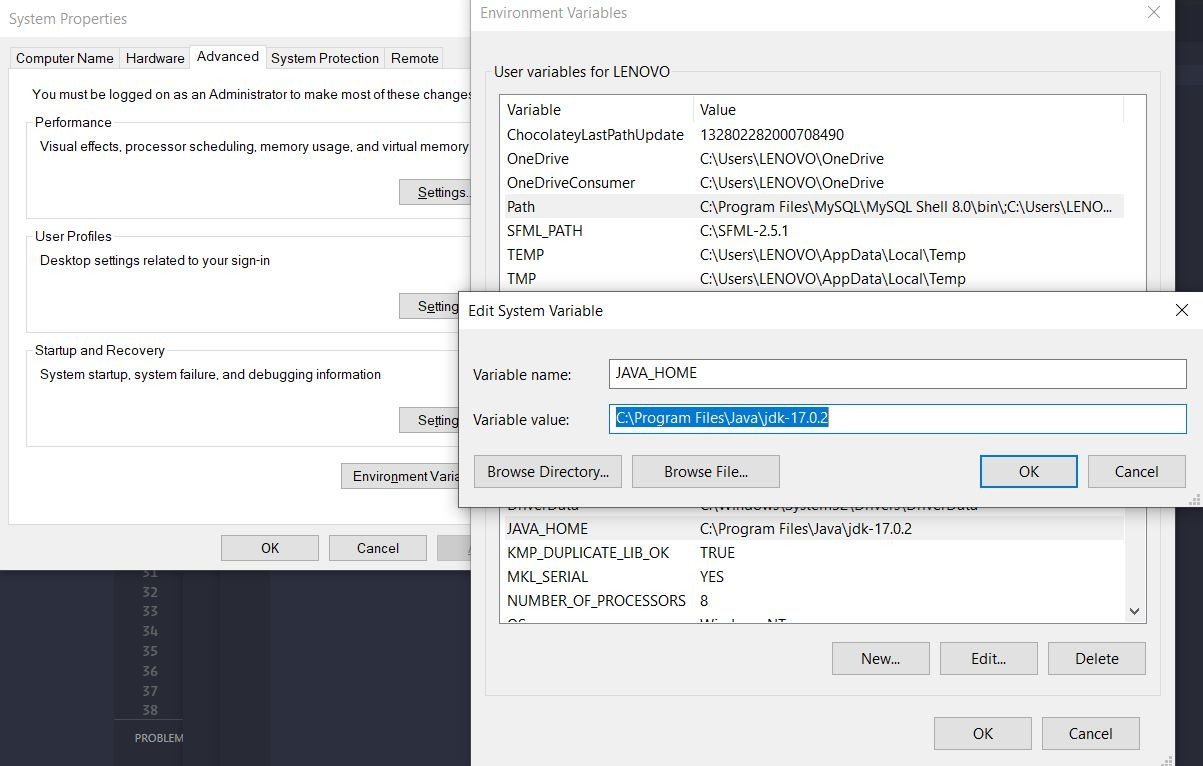
public static void main(String args[]) {

System.*out*.println("Hello, World")**;**

}

}





\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

2)

Prerequisites:

* Basic understanding of programming concepts
* Familiarity with the Java programming language

Underlying Concept/Theory:

* Class: A blueprint or template for creating objects in Java. It defines the data and behavior of the objects.
* Main method: The entry point of a Java program. It is the starting point for the execution of the program.
* Console input/output (I/O): The process of reading input from the console and displaying output to the console.

Problem Statement:

Write a Java program that prompts the user to enter their name, age, and favorite color. The program should then display a message with the entered information.

Algorithm/Step-by-step instructions:

1. Create a Java class called "UserInfo".
2. Inside the class, declare the main method.
3. Inside the main method, declare variables to store the user's name, age, and favorite color.
4. Use the console input/output functions to prompt the user for their name, age, and favorite color, and store the input in the respective variables.
5. Concatenate the variables with a message string to form the output message.
6. Display the output message using the console output function.

Conclusion/Remarks:

This program demonstrates the basic structure of a Java program, including the class definition, main method, and console input/output operations. It shows how to read user input from the console and display output to the console.

Exercises/Challenges:

1. Modify the program to ask for additional information, such as the user's address and phone number.
2. Implement error handling for invalid user inputs, such as entering a non-numeric value for the age.
3. Create a program that calculates the sum of two numbers entered by the user and displays the result.

Code:

import java.util.Scanner**;**

public class UserInfo {

public static void main(String[] args) {

Scanner input = new Scanner(System.*in*)**;**

System.*out*.print("Enter your name: ")**;**

String name = input.nextLine()**;**

System.*out*.print("Enter your age: ")**;**

int age = input.nextInt()**;**

input.nextLine()**;** // Consume the newline character

System.*out*.print("Enter your favorite color: ")**;**

String color = input.nextLine()**;**

String message = "Hello, " + name + "! You are " + age + " years old and your favorite color is " + color + "."**;**

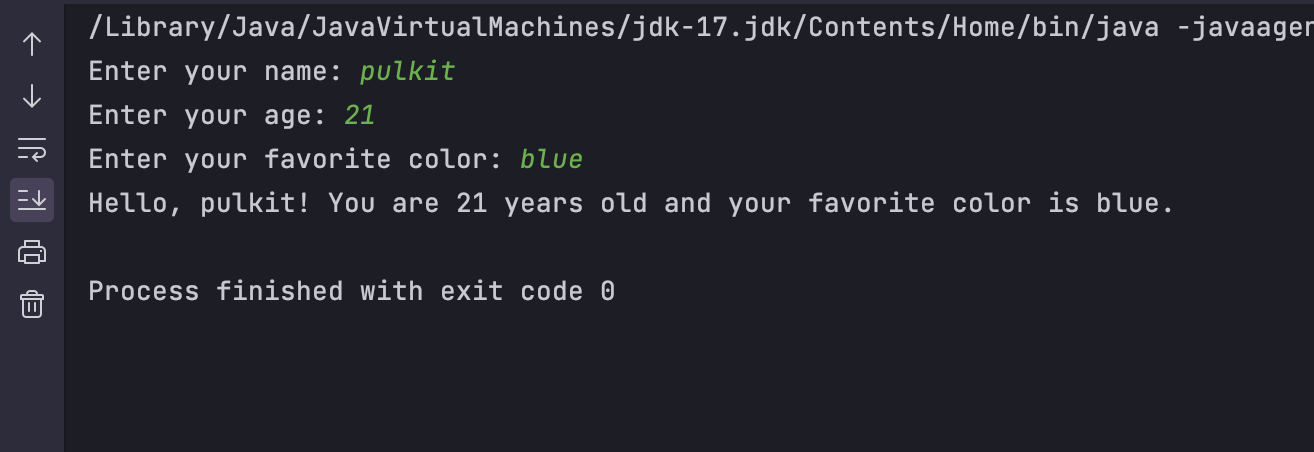
System.*out*.println(message)**;**

input.close()**;**

}

}

output:



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

3)

Prerequisites:

Basic knowledge of Java programming language.

Underlying Concept / Theory:

In Java, classes and objects are fundamental concepts of object-oriented programming. A class is a blueprint or template that defines the properties and behaviors of an object. An object is an instance of a class, representing a specific entity or concept.

Encapsulation is a mechanism in Java that bundles data (attributes) and methods (behavior) together within a class, hiding the internal implementation details from the outside world. It promotes data protection and code reusability.

Abstraction is another important concept that allows us to create abstract classes and interfaces, which define common behaviors and characteristics that can be shared by multiple classes. It helps in achieving modularity and code maintenance.

Problem Statement:

Create a Java program that demonstrates the concept of classes and objects, and showcases encapsulation and abstraction principles.

Algorithm / Step-by-step instructions:

1. Create a class named "Car" with private attributes such as "brand", "model", and "price".
2. Implement public getter and setter methods for each attribute to encapsulate the data.
3. Create a public method named "startEngine" that displays a message indicating the car has started.
4. Create a public abstract method named "drive" within the "Car" class, representing the common behavior of driving a car.
5. Create two subclasses of "Car" named "Sedan" and "SUV".
6. Implement the "drive" method in each subclass with specific instructions for driving a sedan and an SUV.
7. Create objects of both "Sedan" and "SUV" classes.
8. Use the setter methods to set the attributes of the objects.
9. Call the "startEngine" method for each object.
10. Call the "drive" method for each object.
11. Display the details and behavior of the objects using the getter methods.

Output:

The program should display the following output:

* Messages indicating that the car has started.
* Specific instructions for driving a sedan and an SUV.
* Details and behavior of the created objects.

Conclusion / Remarks:

The program successfully demonstrates the concepts of classes and objects in Java. It illustrates encapsulation by encapsulating the data within the class and providing public methods to access and modify it. It also showcases abstraction by defining a common behavior using an abstract method and implementing it in subclasses with specific instructions.

Exercises / Challenges:

1. Extend the program to include additional types of cars and implement their specific driving behaviors.
2. Modify the program to calculate and display the fuel efficiency of each car based on distance traveled and fuel consumed.
3. Create a method within the "Car" class to calculate the total cost of ownership based on the car's price, maintenance cost, and fuel efficiency.
4. Implement a sorting mechanism to sort the created objects based on their price or any other attribute.
5. Create an interface named "Convertible" and implement it in a class representing a convertible car, adding convertible-specific behaviors.

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

CODE::

// BankAccount.java

public class BankAccount {

private String accountNumber**;**

private double balance**;**

// Encapsulation: Accessors and Mutators

public String getAccountNumber() {

return accountNumber**;**

}

public void setAccountNumber(String accountNumber) {

this.accountNumber = accountNumber**;**

}

public double getBalance() {

return balance**;**

}

public void setBalance(double balance) {

this.balance = balance**;**

}

// Abstraction: Methods to perform operations

public void deposit(double amount) {

balance += amount**;**

System.*out*.println("Deposit of $" + amount + " processed. New balance: $" + balance)**;**

}

public void withdraw(double amount) {

if (balance >= amount) {

balance -= amount**;**

System.*out*.println("Withdrawal of $" + amount + " processed. New balance: $" + balance)**;**

} else {

System.*out*.println("Insufficient funds. Withdrawal not processed.")**;**

}

}

}

Second file

// App.java

public class App {

public static void main(String[] args) {

BankAccount account1 = new BankAccount()**;**

account1.setAccountNumber("123456")**;**

account1.setBalance(1000.0)**;**

System.*out*.println("Account Number: " + account1.getAccountNumber())**;**

System.*out*.println("Balance: $" + account1.getBalance())**;**

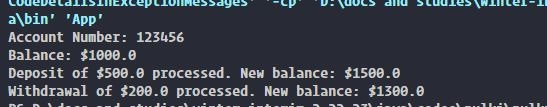
account1.deposit(500.0)**;**

account1.withdraw(200.0)**;**

}

}

+++++++++++++++++++++++++++++++++++++++++++++++++++++++



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

4)

Exercise (Objective): Demonstrate the use of access modifiers

Prerequisites:

* Basic understanding of object-oriented programming concepts
* Familiarity with programming languages that support access modifiers (such as Java, C++, or C#)

Underlying Concept / Theory:

Access modifiers are keywords used in object-oriented programming languages to define the accessibility of classes, methods, and variables. They control the level of visibility and enforce encapsulation and information hiding.

Problem Statement:

Create a class called "Person" with private variables for name, age, and salary. Implement public methods to set and retrieve these values. Demonstrate the use of different access modifiers.

Algorithm / Step-by-step instructions:

1. Create a class named "Person."
2. Declare private variables inside the class for name, age, and salary.
3. Implement public methods (getters and setters) to access and modify these private variables.
4. Use the "private" access modifier for the variables and the "public" access modifier for the methods.
5. In the setter methods, validate the input values to ensure they meet certain criteria (e.g., age should be positive, salary should be non-negative).
6. Instantiate an object of the "Person" class.
7. Use the setter methods to set the values of name, age, and salary for the object.
8. Use the getter methods to retrieve and print the values of name, age, and salary for the object.

Output:

The output will display the values of name, age, and salary for the person object, which were set using the setter methods.

Conclusion / Remarks:

Access modifiers are crucial for maintaining encapsulation and controlling the visibility of variables and methods in object-oriented programming. By using access modifiers appropriately, we can enforce data protection and ensure that only the necessary components are accessible to other parts of the program.

Exercises / Challenges:

1. Extend the "Person" class to include a private variable for email address. Implement a public method to retrieve the email address and another method to set it. Use the appropriate access modifier.
2. Create a subclass of "Person" called "Employee" with an additional private variable for employee ID. Implement public methods to retrieve and set the employee ID. Use the appropriate access modifier for the variables and methods.
3. Create multiple instances of the "Person" and "Employee" classes and demonstrate the use of access modifiers by accessing and modifying their variables from different parts of the program.

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

CODE :

class base{

private int a**;**

protected int b**;**

public int c**;**

int d**;**

public base()

{

a=5**;**

b=6**;**

c=7**;**

d=8**;**

}

public void display()

{

System.*out*.println("Private Accessible by Member function but not object "+a)**;**

}

}

class child extends base{

public child()

{System.*out*.println("Private Inaccessible "+a)**;**

System.*out*.println("Protected is Accessible by Inherited Class but not outside package "+b)**;**

System.*out*.println("Default is accessible inside every file in package "+d)**;**

}

}

public class AccessSpecifier {

public static void main(String[] args)

{

child c=new child()**;**

System.*out*.println(c.c)**;**

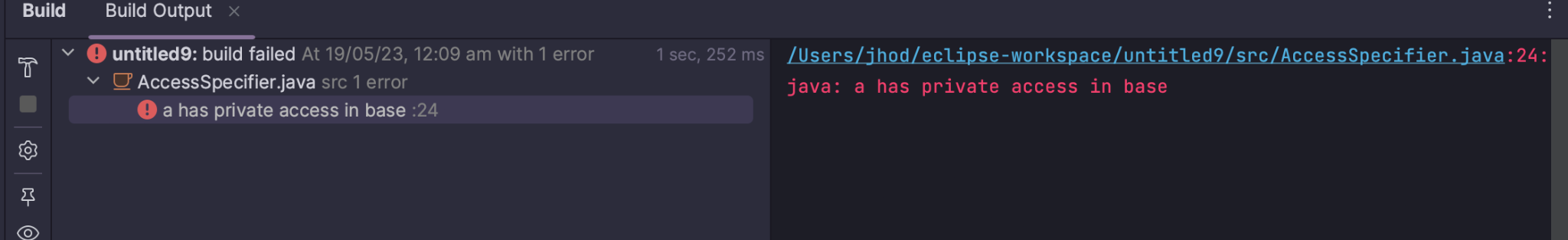
System.*out*.println(c.b)**;**

c.display()**;**

}

}

Output:



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

5)

Prerequisites:

* Basic understanding of object-oriented programming concepts
* Familiarity with the Java programming language

Underlying Concept / Theory:

Inheritance is a fundamental concept in object-oriented programming that allows a class to inherit properties and behaviors from another class. It promotes code reuse and helps in creating a hierarchical relationship between classes. There are different types of inheritance, including simple inheritance and multilevel inheritance.

Problem Statement:

Create a Java program to demonstrate simple inheritance and multilevel inheritance. Implement different types of inheritance relationships to showcase the concept effectively.

Algorithm / Step-by-step instructions:

1. Create a base class called "Vehicle" with properties such as "make" and "model" and a method called "displayInfo" to print the vehicle details.
2. Create a derived class called "Car" that inherits from the Vehicle class. Add additional properties specific to cars, such as "numberOfDoors" and "fuelType."
3. Implement the displayInfo method in the Car class to display car-specific details along with the inherited vehicle details.
4. Create another derived class called "SportsCar" that inherits from the Car class. Add a property called "topSpeed" specific to sports cars.
5. Implement the displayInfo method in the SportsCar class to display sports car-specific details along with the inherited car and vehicle details.
6. Instantiate objects of the Vehicle, Car, and SportsCar classes in the main method.
7. Call the displayInfo method on each object to observe the inheritance and method overriding in action.

Output:

The output of the program should display the vehicle details, car details, and sports car details, demonstrating the inheritance relationship and method overriding.

Conclusion / Remarks:

Inheritance is a powerful feature in object-oriented programming that enables code reuse and facilitates the creation of class hierarchies. It allows derived classes to inherit properties and behaviors from their base classes, promoting modularity and extensibility in software development.

Exercises / Challenges:

1. Modify the program to include another type of inheritance relationship, such as multiple inheritance or hierarchical inheritance, and demonstrate it.
2. Implement additional derived classes to showcase other types of inheritance, such as hybrid inheritance or multiple inheritance using interfaces.
3. Create a class hierarchy representing different animals using inheritance and demonstrate polymorphism by calling a common method on different animal objects.
4. Implement an example of method overriding in the derived classes and demonstrate how the behavior of a method can be changed in the derived class.
5. Create a class representing a university with properties such as name and location. Implement derived classes for different types of departments, such as Computer Science, Mathematics, and English, showcasing the concept of inheritance in a real-world scenario.

CODE:

class Level1{

int a**;**

public Level1()

{

a=5**;**

}

}

class Level2 extends Level1{

int b**;**

public Level2() {

b = 3**;**

}

}

class Level3 extends Level2{

int c**;**

public Level3() {

c = 4**;**

}

}

public class Main {

public static void main(String[] args) {

Level3 l=new Level3()**;**

System.*out*.println("Multi-Level Inheritance ")**;**

System.*out*.println("Class Element is "+l.c)**;**

System.*out*.println("Parent Class Element is "+l.b)**;**

System.*out*.println("Grand Parent Class Element is "+l.a)**;**

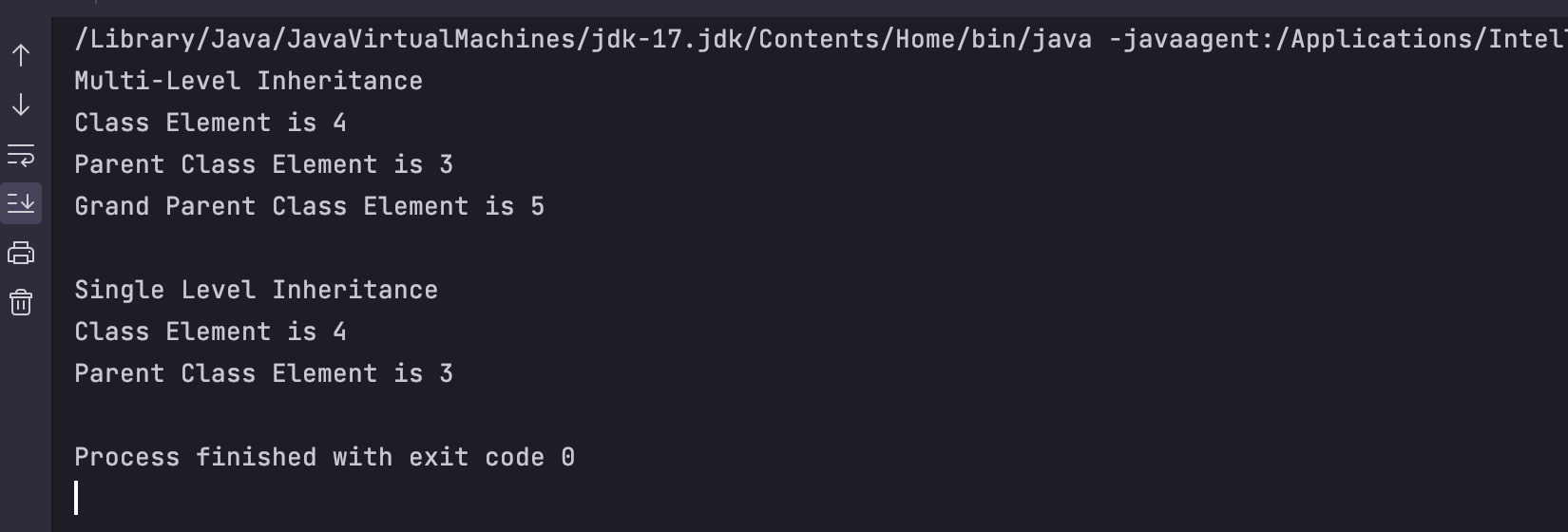
System.*out*.println("\nSingle Level Inheritance")**;**

System.*out*.println("Class Element is "+l.c)**;**

System.*out*.println("Parent Class Element is "+l.b)**;**

}}

output:



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

6)

Prerequisites:

* Basic knowledge of Java programming language
* Understanding of inheritance and interfaces in Java

Underlying Concept / Theory:

* In Java, multiple inheritance refers to a class inheriting from multiple classes. However, Java does not support multiple inheritance for classes. Instead, multiple inheritance can be achieved through interfaces.
* An interface is a collection of abstract methods that a class can implement. It defines a contract that the implementing class must adhere to.
* By implementing multiple interfaces, a class can inherit behavior and functionality from multiple sources.

Problem Statement:

Implement a Java program to demonstrate the concept of multiple inheritance using interfaces. Create interfaces A and B, and a class C that implements both interfaces. Test the program by creating an object of class C and invoking methods from both interfaces.

Algorithm / Step-by-step instructions:

1. Declare interface A with abstract methods.
2. Declare interface B with abstract methods.
3. Implement interface A in class C by providing method implementations.
4. Implement interface B in class C by providing method implementations.
5. Create an object of class C.
6. Invoke methods from both interfaces using the object created in the previous step.
7. Display the output.

Output:

The output will vary based on the method implementations in interfaces A and B, as well as the implementations in class C. The output should demonstrate the successful inheritance of behavior and functionality from both interfaces.

Conclusion / Remarks:

By using interfaces in Java, we can achieve the concept of multiple inheritance. Interfaces provide a way to define a contract that a class must adhere to and inherit behavior from multiple sources. This allows for better code organization, flexibility, and code reuse.

Exercises / Challenges:

1. Modify the program to include a third interface D. Implement D in class C and test the program by invoking methods from all three interfaces.
2. Create additional classes that implement interfaces A and B. Test the program by creating objects of these classes and invoking their respective methods.
3. Explore the diamond problem in the context of interfaces and multiple inheritance. Try to create a scenario where the diamond problem arises and find a solution to resolve it.

CODE:

interface parent1{

default void mem1()

{

System.*out*.println("This is member of interface 1")**;**

} }

interface parent2{

default void mem2()

{

System.*out*.println("This is member of interface 2")**;**

} }

class childinheritance implements parent1**,**parent2{

}

public class Multiple{

public static void main(String[] args)

{

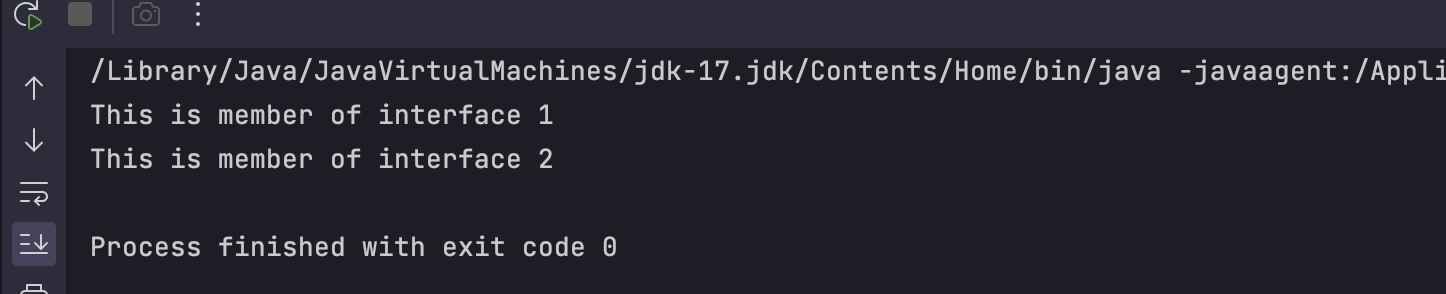
childinheritance c=new childinheritance()**;**

c.mem1()**;**

c.mem2()**;**

}

}



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

7)

Prerequisites:

* Basic knowledge of Java programming language
* Understanding of variables and data types in Java

Underlying Concept / Theory:

* Arrays are used to store multiple values of the same data type in a single variable.
* Each value in an array is called an element, and each element is accessed by its index.
* Arrays have a fixed size, which is specified during their declaration.
* Array indices start from 0, so the first element is accessed using index 0.

Problem Statement:

Create a Java program that demonstrates the use of arrays. The program should ask the user to enter five integers and then display the sum and average of those numbers.

Algorithm / Step-by-step instructions:

1. Declare an integer array named "numbers" with a size of 5.
2. Create a loop that iterates five times to prompt the user for five integers.
3. Inside the loop, read the user's input and store each number in the array using the loop index.
4. After the loop, declare two variables: "sum" to store the sum of the numbers and "average" to store the average.
5. Initialize "sum" to 0.
6. Create a loop that iterates through the array.
7. Inside the loop, add each element to the "sum" variable.
8. Calculate the average by dividing the sum by the size of the array (5 in this case).
9. Display the sum and average to the user.

Output:

Enter number 1: [user input]

Enter number 2: [user input]

Enter number 3: [user input]

Enter number 4: [user input]

Enter number 5: [user input]

Sum: [sum of the numbers]

Average: [average of the numbers]

Conclusion / Remarks:

Arrays are a useful concept in programming for storing and manipulating multiple values. They provide a convenient way to work with collections of data in a structured manner.

Exercises / Challenges:

1. Modify the program to find the minimum and maximum values entered by the user.
2. Extend the program to handle a dynamic number of inputs from the user.
3. Implement a sorting algorithm to sort the numbers in the array in ascending order.
4. Create a program that asks the user for a sentence and stores each word in an array, then displays the number of words and the longest word.

CODE:

public class Array {

public static void main(String[] args) {

int[] a={5**,**3**,**7**,**2**,**9}**;**

System.*out*.println("The elements in the array are ")**;**

for (int x:a)

System.*out*.println(x)**;**

}

}

