**1. Write java code for creating 3 methods called even, odd and prime in parent class and override them from child class. The parent class should print 1 to 20 range in all method, but child print 1 to 30 array range in all methods. And access all the methods using derived class object**

**Pseudocode:**

BEGIN

DECLARE variables of types:

byte b

short s

int i

long l

float f

double d

char c

boolean bool

ASSIGN appropriate values to each variable:

b = 10

s = 100

i = 1000

l = 100000L

f = 3.14

d = 3.14159

c = 'A'

bool = true

PRINT all the values of the variables:

PRINT "Byte value: " followed by b

PRINT "Short value: " followed by s

PRINT "Int value: " followed by i

PRINT "Long value: " followed by l

PRINT "Float value: " followed by f

PRINT "Double value: " followed by d

PRINT "Char value: " followed by c

PRINT "Boolean value: " followed by bool

END

**Java Code:**

// Parent class

class NumberPrinter {

// Method to print even numbers from 1 to 20

public void even() {

System.out.println("Even numbers from 1 to 20:");

for (int i = 1; i <= 20; i++) {

if (i % 2 == 0) {

System.out.print(i + " ");

}

}

System.out.println();

}

// Method to print odd numbers from 1 to 20

public void odd() {

System.out.println("Odd numbers from 1 to 20:");

for (int i = 1; i <= 20; i++) {

if (i % 2 != 0) {

System.out.print(i + " ");

}

}

System.out.println();

}

// Method to print prime numbers from 1 to 20

public void prime() {

System.out.println("Prime numbers from 1 to 20:");

for (int i = 2; i <= 20; i++) {

boolean isPrime = true;

for (int j = 2; j <= i / 2; j++) {

if (i % j == 0) {

isPrime = false;

break;

}

}

if (isPrime) {

System.out.print(i + " ");

}

}

System.out.println();

}

}

// Child class extending NumberPrinter

class ExtendedNumberPrinter extends NumberPrinter {

// Override method to print even numbers from 1 to 30

@Override

public void even() {

System.out.println("Even numbers from 1 to 30:");

for (int i = 1; i <= 30; i++) {

if (i % 2 == 0) {

System.out.print(i + " ");

}

}

System.out.println();

}

// Override method to print odd numbers from 1 to 30

@Override

public void odd() {

System.out.println("Odd numbers from 1 to 30:");

for (int i = 1; i <= 30; i++) {

if (i % 2 != 0) {

System.out.print(i + " ");

}

}

System.out.println();

}

// Override method to print prime numbers from 1 to 30

@Override

public void prime() {

System.out.println("Prime numbers from 1 to 30:");

for (int i = 2; i <= 30; i++) {

boolean isPrime = true;

for (int j = 2; j <= i / 2; j++) {

if (i % j == 0) {

isPrime = false;

break;

}

}

if (isPrime) {

System.out.print(i + " ");

}

}

System.out.println();

}

}

// Main class to test the functionality

public class Main {

public static void main(String[] args) {

ExtendedNumberPrinter printer = new ExtendedNumberPrinter();

// Access methods using the child class object

printer.even();

printer.odd();

printer.prime();

}

}

**2. Override a default constructor, which contains a=10,b=15 in java using parametrized constructor which contains a=20,b=25. Create a method called disp(). And call both the constructors associating it with disp()**

Pseudocode:

BEGIN

DEFINE Class `Example`

// Default constructor

FUNCTION DefaultConstructor()

SET a = 10

SET b = 15

// Parameterized constructor

FUNCTION ParameterizedConstructor(int aValue, int bValue)

SET a = aValue

SET b = bValue

// Method to display values

FUNCTION disp()

PRINT "Value of a: " followed by a

PRINT "Value of b: " followed by b

// Main program execution

CREATE an instance of Example using the default constructor

CALL disp() method on this instance

CREATE an instance of Example using the parameterized constructor with values 20 and 25

CALL disp() method on this instance

END

**Java Code:**

// Define the class Example

class Example {

int a;

int b;

// Default constructor

public Example() {

a = 10;

b = 15;

}

// Parameterized constructor

public Example(int aValue, int bValue) {

a = aValue;

b = bValue;

}

// Method to display values

public void disp() {

System.out.println("Value of a: " + a);

System.out.println("Value of b: " + b);

}

// Main method to test the constructors and disp() method

public static void main(String[] args) {

// Create an instance of Example using the default constructor

Example obj1 = new Example();

obj1.disp(); // Display values using default constructor

// Create an instance of Example using the parameterized constructor

Example obj2 = new Example(20, 25);

obj2.disp(); // Display values using parameterized constructor

}

}

**3. Java program to create a Base class with a method called ‘void area(int a, int b)’ which print area of a square. Now override the method from Derived class and make it print area of a rectangle.**

**Pseudocode:**

BEGIN

DEFINE Base Class

FUNCTION area(int a, int b)

PRINT "Area of the square: " followed by (a \* a)

DEFINE Derived Class that extends Base

OVERRIDE FUNCTION area(int a, int b)

PRINT "Area of the rectangle: " followed by (a \* b)

// Main program execution

CREATE an instance of Derived Class

CALL area() method on this instance with values 5 and 10

END

**Java Code:**

// Define the Base class

class Base {

// Method to calculate area of a square

public void area(int a, int b) {

// b is not used here, as the method calculates the area of a square

System.out.println("Area of the square: " + (a \* a));

}

}

// Define the Derived class that extends Base

class Derived extends Base {

// Override method to calculate area of a rectangle

@Override

public void area(int a, int b) {

// Calculate the area of a rectangle

System.out.println("Area of the rectangle: " + (a \* b));

}

}

// Main class to test the functionality

public class Main {

public static void main(String[] args) {

// Create an instance of Derived class

Derived obj = new Derived();

// Call the overridden area() method with values for rectangle

obj.area(5, 10);

}

}

**4. Write java code to overload a method called ‘int sum(int a, int b)’ by the 3 ways.**

**.a) By increasing and decreasing no. of parameters**

**Pseudocode:**

BEGIN

DEFINE Class `Calculator`

// Method to sum two integers

FUNCTION sum(int a, int b)

RETURN a + b

// Overloaded method to sum three integers

FUNCTION sum(int a, int b, int c)

RETURN a + b + c

// Main program execution

CREATE an instance of Calculator

CALL sum() method with two integers

CALL sum() method with three integers

END

**Java code:**

// Define the Calculator class

class Calculator {

// Method to sum two integers

public int sum(int a, int b) {

return a + b;

}

// Overloaded method to sum three integers

public int sum(int a, int b, int c) {

return a + b + c;

}

}

// Main class to test the overloaded methods

public class Main {

public static void main(String[] args) {

// Create an instance of Calculator

Calculator calc = new Calculator();

// Call the sum() method with two integers

System.out.println("Sum of 5 and 10: " + calc.sum(5, 10));

// Call the sum() method with three integers

System.out.println("Sum of 1, 2, and 3: " + calc.sum(1, 2, 3));

}

}

**b)      By changing the data types of parameters**

**Pseudocode:**

BEGIN

DEFINE Class `Calculator`

// Method to sum two integers

FUNCTION sum(int a, int b)

RETURN a + b

// Overloaded method to sum two doubles

FUNCTION sum(double a, double b)

RETURN a + b

// Main program execution

CREATE an instance of Calculator

CALL sum() method with two integers

CALL sum() method with two doubles

END

**Java code:**

// Define the Calculator class

class Calculator {

// Method to sum two integers

public int sum(int a, int b) {

return a + b;

}

// Overloaded method to sum two doubles

public double sum(double a, double b) {

return a + b;

}

}

// Main class to test the overloaded methods

public class Main {

public static void main(String[] args) {

// Create an instance of Calculator

Calculator calc = new Calculator();

// Call the sum() method with two integers

System.out.println("Sum of 5 and 10: " + calc.sum(5, 10));

// Call the sum() method with two doubles

System.out.println("Sum of 2.5 and 3.5: " + calc.sum(2.5, 3.5));

}

}

**c)      By interchanging the parameters**

**Pseudo code:**

BEGIN

DEFINE Class `Calculator`

// Method to sum integer and double

FUNCTION sum(double a, int b)

RETURN a + b

// Overloaded method to sum double and integer

FUNCTION sum(int a, double b)

RETURN a + b

// Main program execution

CREATE an instance of Calculator

CALL sum() method with a double and an integer

CALL sum() method with an integer and a double

END

**Java Code:**

// Define the Calculator class

class Calculator {

// Method to sum a double and an integer

public double sum(double a, int b) {

return a + b;

}

// Overloaded method to sum an integer and a double

public double sum(int a, double b) {

return a + b;

}

}

// Main class to test the overloaded methods

public class Main {

public static void main(String[] args) {

// Create an instance of Calculator

Calculator calc = new Calculator();

// Call the sum() method with a double and an integer

System.out.println("Sum of 2.5 and 3: " + calc.sum(2.5, 3));

// Call the sum() method with an integer and a double

System.out.println("Sum of 3 and 2.5: " + calc.sum(3, 2.5));

}

}

**5. Create an abstract class with 2 abstract methods(total() and average()) and 3 concrete methods(mean(), mode(), median()). Now extend the abstract class from a concrete class and use all the methods in that abstract class**

**Pseudocode:**

BEGIN

DEFINE Abstract Class `Statistics`

// Abstract method to calculate total

FUNCTION total() RETURN int

// Abstract method to calculate average

FUNCTION average() RETURN double

// Concrete method to calculate mean

FUNCTION mean(data[]) RETURN double

// Implementation for mean

// Concrete method to calculate mode

FUNCTION mode(data[]) RETURN int

// Implementation for mode

// Concrete method to calculate median

FUNCTION median(data[]) RETURN double

// Implementation for median

DEFINE Concrete Class `DataStatistics` EXTENDS Statistics

// Implement abstract method total

FUNCTION total() RETURN int

// Implementation of total calculation

// Implement abstract method average

FUNCTION average() RETURN double

// Implementation of average calculation

// Main program execution

CREATE an instance of DataStatistics

CALL total() method

CALL average() method

CALL mean() method with sample data

CALL mode() method with sample data

CALL median() method with sample data

END

Java Code:

// Define the abstract class Statistics

abstract class Statistics {

// Abstract method to calculate total

public abstract int total();

// Abstract method to calculate average

public abstract double average();

// Concrete method to calculate mean

public double mean(int[] data) {

if (data.length == 0) return 0;

double sum = 0;

for (int num : data) {

sum += num;

}

return sum / data.length;

}

// Concrete method to calculate mode

public int mode(int[] data) {

if (data.length == 0) return -1;

java.util.HashMap<Integer, Integer> freqMap = new java.util.HashMap<>();

int mode = data[0];

int maxCount = 0;

for (int num : data) {

int count = freqMap.getOrDefault(num, 0) + 1;

freqMap.put(num, count);

if (count > maxCount) {

maxCount = count;

mode = num;

}

}

return mode;

}

// Concrete method to calculate median

public double median(int[] data) {

if (data.length == 0) return 0;

java.util.Arrays.sort(data);

int middle = data.length / 2;

if (data.length % 2 == 0) {

return (data[middle - 1] + data[middle]) / 2.0;

} else {

return data[middle];

}

}

}

// Define the concrete class DataStatistics extending Statistics

class DataStatistics extends Statistics {

// Implement the abstract method total

@Override

public int total() {

// Example implementation

int[] data = {1, 2, 3, 4, 5}; // Sample data

int sum = 0;

for (int num : data) {

sum += num;

}

return sum;

}

// Implement the abstract method average

@Override

public double average() {

// Example implementation

int[] data = {1, 2, 3, 4, 5}; // Sample data

return mean(data);

}

// Additional methods can be overridden or used directly

}

// Main class to test the methods

public class Main {

public static void main(String[] args) {

// Create an instance of DataStatistics

DataStatistics stats = new DataStatistics();

// Call the abstract methods implemented in DataStatistics

System.out.println("Total: " + stats.total());

System.out.println("Average: " + stats.average());

// Sample data for mean, mode, and median

int[] data = {1, 2, 2, 3, 4, 5};

// Call the concrete methods

System.out.println("Mean: " + stats.mean(data));

System.out.println("Mode: " + stats.mode(data));

System.out.println("Median: " + stats.median(data));

}

}

**6. Create an interface with 4 methods called add(), sub(), mul() and div(). Then give implementation for all in the implementing class**

Pseudocode:

BEGIN

DEFINE Interface `Calculator`

// Method to add two numbers

FUNCTION add(int a, int b) RETURN int

// Method to subtract two numbers

FUNCTION sub(int a, int b) RETURN int

// Method to multiply two numbers

FUNCTION mul(int a, int b) RETURN int

// Method to divide two numbers

FUNCTION div(int a, int b) RETURN double

DEFINE Class `BasicCalculator` IMPLEMENTS Calculator

// Implement add method

FUNCTION add(int a, int b) RETURN int

RETURN a + b

// Implement sub method

FUNCTION sub(int a, int b) RETURN int

RETURN a - b

// Implement mul method

FUNCTION mul(int a, int b) RETURN int

RETURN a \* b

// Implement div method

FUNCTION div(int a, int b) RETURN double

IF b == 0

THROW ArithmeticException("Division by zero")

RETURN (double) a / b

// Main program execution

CREATE an instance of BasicCalculator

CALL add() method with sample inputs

CALL sub() method with sample inputs

CALL mul() method with sample inputs

CALL div() method with sample inputs

END

Java Code:

// Define the Calculator interface

interface Calculator {

// Method to add two numbers

int add(int a, int b);

// Method to subtract two numbers

int sub(int a, int b);

// Method to multiply two numbers

int mul(int a, int b);

// Method to divide two numbers

double div(int a, int b);

}

// Implement the Calculator interface in BasicCalculator class

class BasicCalculator implements Calculator {

**7. Create 3 interfaces with 1 method each sum(), avg(), percentage() respectively. Now implement all the 3 interfaces in your class**

**Pseudocode:**

BEGIN

BEGIN

DEFINE Interface `SumInterface`

// Method to calculate the sum

FUNCTION sum(int a, int b) RETURN int

DEFINE Interface `AvgInterface`

// Method to calculate the average

FUNCTION avg(int a, int b) RETURN double

DEFINE Interface `PercentageInterface`

// Method to calculate the percentage

FUNCTION percentage(int total, int part) RETURN double

DEFINE Class `Calculator` IMPLEMENTS SumInterface, AvgInterface, PercentageInterface

// Implement sum method

FUNCTION sum(int a, int b) RETURN int

RETURN a + b

// Implement avg method

FUNCTION avg(int a, int b) RETURN double

RETURN (a + b) / 2.0

// Implement percentage method

FUNCTION percentage(int total, int part) RETURN double

IF total == 0

RETURN 0

RETURN (part / (double) total) \* 100

// Main program execution

CREATE an instance of Calculator

CALL sum() method with sample inputs

CALL avg() method with sample inputs

CALL percentage() method with sample inputs

END

Java Code:

// Define the SumInterface interface

interface SumInterface {

int sum(int a, int b);

}

// Define the AvgInterface interface

interface AvgInterface {

double avg(int a, int b);

}

// Define the PercentageInterface interface

interface PercentageInterface {

double percentage(int total, int part);

}

// Implement all three interfaces in the Calculator class

class Calculator implements SumInterface, AvgInterface, PercentageInterface {

// Implement sum method

@Override

public int sum(int a, int b) {

return a + b;

}

// Implement avg method

@Override

public double avg(int a, int b) {

return (a + b) / 2.0;

}

// Implement percentage method

@Override

public double percentage(int total, int part) {

if (total == 0) {

return 0;

}

return (part / (double) total) \* 100;

}

}

// Main class to test the implemented methods

public class Main {

public static void main(String[] args) {

// Create an instance of Calculator

Calculator calc = new Calculator();

// Test sum method

System.out.println("Sum: 10 + 5 = " + calc.sum(10, 5));

// Test avg method

System.out.println("Average: (10 + 5) / 2 = " + calc.avg(10, 5));

// Test percentage method

System.out.println("Percentage: 5 out of 20 = " + calc.percentage(20, 5) + "%");

}

}

**8. Create an interface called Tree and extend 2 classes from it called Branch1 and Branch2. Tree should contain methods fruits(), leaves() and flowers(), these methods contain 2,3,4 parameters respectively.**

Pseudocode:

BEGIN

DEFINE Interface `Tree`

// Method to handle fruits with 2 parameters

FUNCTION fruits(String fruitType, int quantity)

// Method to handle leaves with 3 parameters

FUNCTION leaves(String leafType, int length, int width)

// Method to handle flowers with 4 parameters

FUNCTION flowers(String flowerType, int petalCount, String color, boolean hasFragrance)

DEFINE Class `Branch1` IMPLEMENTS Tree

// Implement fruits method

FUNCTION fruits(String fruitType, int quantity)

PRINT "Branch1: Fruits - Type: " + fruitType + ", Quantity: " + quantity

// Implement leaves method

FUNCTION leaves(String leafType, int length, int width)

PRINT "Branch1: Leaves - Type: " + leafType + ", Length: " + length + ", Width: " + width

// Implement flowers method

FUNCTION flowers(String flowerType, int petalCount, String color, boolean hasFragrance)

PRINT "Branch1: Flowers - Type: " + flowerType + ", Petal Count: " + petalCount + ", Color: " + color + ", Fragrance: " + hasFragrance

DEFINE Class `Branch2` IMPLEMENTS Tree

// Implement fruits method

FUNCTION fruits(String fruitType, int quantity)

PRINT "Branch2: Fruits - Type: " + fruitType + ", Quantity: " + quantity

// Implement leaves method

FUNCTION leaves(String leafType, int length, int width)

PRINT "Branch2: Leaves - Type: " + leafType + ", Length: " + length + ", Width: " + width

// Implement flowers method

FUNCTION flowers(String flowerType, int petalCount, String color, boolean hasFragrance)

PRINT "Branch2: Flowers - Type: " + flowerType + ", Petal Count: " + petalCount + ", Color: " + color + ", Fragrance: " + hasFragrance

// Main program execution

CREATE instances of Branch1 and Branch2

CALL fruits(), leaves(), and flowers() methods on both instances

END

Java Code:

// Define the Tree interface

interface Tree {

// Method to handle fruits with 2 parameters

void fruits(String fruitType, int quantity);

// Method to handle leaves with 3 parameters

void leaves(String leafType, int length, int width);

// Method to handle flowers with 4 parameters

void flowers(String flowerType, int petalCount, String color, boolean hasFragrance);

}

// Implement the Tree interface in Branch1

class Branch1 implements Tree {

// Implement fruits method

@Override

public void fruits(String fruitType, int quantity) {

System.out.println("Branch1: Fruits - Type: " + fruitType + ", Quantity: " + quantity);

}

// Implement leaves method

@Override

public void leaves(String leafType, int length, int width) {

System.out.println("Branch1: Leaves - Type: " + leafType + ", Length: " + length + ", Width: " + width);

}

// Implement flowers method

@Override

public void flowers(String flowerType, int petalCount, String color, boolean hasFragrance) {

System.out.println("Branch1: Flowers - Type: " + flowerType + ", Petal Count: " + petalCount + ", Color: " + color + ", Fragrance: " + hasFragrance);

}

}

// Implement the Tree interface in Branch2

class Branch2 implements Tree {

// Implement fruits method

@Override

public void fruits(String fruitType, int quantity) {

System.out.println("Branch2: Fruits - Type: " + fruitType + ", Quantity: " + quantity);

}

// Implement leaves method

@Override

public void leaves(String leafType, int length, int width) {

System.out.println("Branch2: Leaves - Type: " + leafType + ", Length: " + length + ", Width: " + width);

}

// Implement flowers method

@Override

public void flowers(String flowerType, int petalCount, String color, boolean hasFragrance) {

System.out.println("Branch2: Flowers - Type: " + flowerType + ", Petal Count: " + petalCount + ", Color: " + color + ", Fragrance: " + hasFragrance);

}

}

// Main class to test the implementations

public class Main {

public static void main(String[] args) {

// Create instances of Branch1 and Branch2

Tree branch1 = new Branch1();

Tree branch2 = new Branch2();

// Test methods for Branch1

branch1.fruits("Apple", 10);

branch1.leaves("Maple", 5, 3);

branch1.flowers("Rose", 5, "Red", true);

// Test methods for Branch2

branch2.fruits("Orange", 20);

branch2.leaves("Oak", 6, 4);

branch2.flowers("Tulip", 6, "Yellow", false);

}

}

**9. Use static keyword in the following levels**

1. **Static variable**
2. **Static method**
3. **Static block**

**Static nested classes**

Pseudocode:

BEGIN

// Define a class with different static members

DEFINE Class `StaticExample`

// Static variable

DECLARE STATIC int staticVariable

// Static method

DEFINE STATIC FUNCTION staticMethod()

PRINT "This is a static method"

// Static block

DEFINE STATIC BLOCK

PRINT "This is a static block"

SET staticVariable = 10

// Static nested class

DEFINE STATIC NESTED CLASS StaticNestedClass

// Static method in nested class

DEFINE STATIC FUNCTION nestedStaticMethod()

PRINT "This is a static method in a static nested class"

// Method to access outer class static variable

DEFINE FUNCTION accessOuterStaticVariable()

PRINT "Static variable from outer class: " + StaticExample.staticVariable

// Main program execution

CREATE INSTANCE of StaticNestedClass

CALL staticMethod() from StaticExample

CALL nestedStaticMethod() from StaticNestedClass

CALL accessOuterStaticVariable() from StaticNestedClass instance

END

Java Code:

public class StaticExample {

// Static variable

static int staticVariable;

// Static method

static void staticMethod() {

System.out.println("This is a static method");

}

// Static block

static {

System.out.println("This is a static block");

staticVariable = 10;

}

// Static nested class

static class StaticNestedClass {

// Static method in nested class

static void nestedStaticMethod() {

System.out.println("This is a static method in a static nested class");

}

// Method to access outer class static variable

void accessOuterStaticVariable() {

System.out.println("Static variable from outer class: " + StaticExample.staticVariable);

}

}

// Main method to demonstrate the usage

public static void main(String[] args) {

// Call static method

StaticExample.staticMethod();

// Create an instance of StaticNestedClass

StaticNestedClass nestedClass = new StaticNestedClass();

// Call static method in static nested class

StaticNestedClass.nestedStaticMethod();

// Access outer class static variable from nested class

nestedClass.accessOuterStaticVariable();

}

}

**10. Use Final keyword in the following levels**

1. **Final variable**
2. **Final method**

**Final classes**

**Pseudocode:**

BEGIN

// Define a class with different final members

DEFINE Class `FinalExample`

// Final variable

DECLARE FINAL int finalVariable = 100

// Final method

DEFINE FINAL FUNCTION finalMethod()

PRINT "This is a final method and cannot be overridden"

// Method in the class

DEFINE FUNCTION regularMethod()

PRINT "This is a regular method"

// Define a final class

DEFINE FINAL CLASS FinalClass

// Method in final class

DEFINE FUNCTION printMessage()

PRINT "This is a method in a final class"

// Attempt to extend the final class (not allowed)

DEFINE CLASS ExtendedClass EXTENDS FinalClass

// This is not allowed, and should be commented out or omitted

// Main program execution

CREATE INSTANCE of FinalExample

CALL finalMethod() from FinalExample

CALL regularMethod() from FinalExample

// Create an instance of FinalClass

CREATE INSTANCE of FinalClass

CALL printMessage() from FinalClass instance

END

\*\*Java Code:\*\*

// Define a class with final members

public class FinalExample {

// Final variable

final int finalVariable = 100;

// Final method

final void finalMethod() {

System.out.println("This is a final method and cannot be overridden");

}

// Regular method

void regularMethod() {

System.out.println("This is a regular method");

}

}

// Define a final class

final class FinalClass {

// Method in final class

void printMessage() {

System.out.println("This is a method in a final class");

}

}

// Attempting to extend a final class will result in a compilation error

// public class ExtendedClass extends FinalClass {

// // This code is not allowed and will produce a compilation error

// }

// Main class to demonstrate the usage of final keyword

public class Main {

public static void main(String[] args) {

// Create an instance of FinalExample

FinalExample example = new FinalExample();

// Call final method

example.finalMethod();

// Call regular method

example.regularMethod();

// Create an instance of FinalClass

FinalClass finalClass = new FinalClass();

// Call method in final class

finalClass.printMessage();

}

}

**11.Use ‘this’ keyword in the following purposes**

1. **Referencing instance variable**
2. **Invoking another constructor**
3. **Passing current object as a parameter**
4. **Returning current object**

**\*\*Pseudocode:\*\***

BEGIN

// Define a class demonstrating various uses of 'this' keyword

DEFINE CLASS `ThisKeywordExample`

// Instance variables

DECLARE int instanceVariable1

DECLARE int instanceVariable2

// Constructor with parameters

DEFINE CONSTRUCTOR `ThisKeywordExample(int a, int b)`

SET this.instanceVariable1 = a

SET this.instanceVariable2 = b

// Constructor invoking another constructor

DEFINE CONSTRUCTOR `ThisKeywordExample(int a)`

CALL this(a, 0) // Invoking another constructor with default value

// Method referencing instance variable

DEFINE FUNCTION `showInstanceVariables()`

PRINT "Instance Variable 1: " + this.instanceVariable1

PRINT "Instance Variable 2: " + this.instanceVariable2

// Method passing current object as a parameter

DEFINE FUNCTION `passObject(ThisKeywordExample obj)`

CALL obj.showInstanceVariables()

// Method returning current object

DEFINE FUNCTION `returnCurrentObject() RETURNS ThisKeywordExample`

RETURN this

// Main program execution

DEFINE CLASS `Main`

DEFINE FUNCTION `main(String[] args)`

// Create an instance of ThisKeywordExample

CREATE INSTANCE of `ThisKeywordExample` with parameters 10 and 20

CALL showInstanceVariables() from instance

// Create an instance using another constructor

CREATE INSTANCE of `ThisKeywordExample` with parameter 30

CALL showInstanceVariables() from instance

// Pass current object as a parameter

CALL passObject(instance) from instance

// Return current object and use it

CREATE INSTANCE of `ThisKeywordExample` with parameters 40 and 50

CREATE INSTANCE of `ThisKeywordExample` from returnCurrentObject() method

CALL showInstanceVariables() from newInstance

END

\*\*Java Code:\*\*

public class ThisKeywordExample {

// Instance variables

int instanceVariable1;

int instanceVariable2;

// Constructor with parameters

public ThisKeywordExample(int a, int b) {

this.instanceVariable1 = a; // Referencing instance variable

this.instanceVariable2 = b; // Referencing instance variable

}

// Constructor invoking another constructor

public ThisKeywordExample(int a) {

this(a, 0); // Invoking another constructor with default value

}

// Method referencing instance variables

public void showInstanceVariables() {

System.out.println("Instance Variable 1: " + this.instanceVariable1);

System.out.println("Instance Variable 2: " + this.instanceVariable2);

}

// Method passing current object as a parameter

public void passObject(ThisKeywordExample obj) {

obj.showInstanceVariables(); // Passing current object

}

// Method returning current object

public ThisKeywordExample returnCurrentObject() {

return this; // Returning current object

}

// Main method to demonstrate the use of 'this' keyword

public static void main(String[] args) {

// Create an instance of ThisKeywordExample using the constructor with parameters

ThisKeywordExample instance = new ThisKeywordExample(10, 20);

instance.showInstanceVariables();

// Create an instance using the constructor that invokes another constructor

ThisKeywordExample instance2 = new ThisKeywordExample(30);

instance2.showInstanceVariables();

// Pass current object as a parameter

instance.passObject(instance);

// Return current object and use it

ThisKeywordExample instance3 = new ThisKeywordExample(40, 50);

ThisKeywordExample returnedInstance = instance3.returnCurrentObject();

returnedInstance.showInstanceVariables();

}

}

**12. Use ‘super’ keyword in the following purposes**

1. **Accessing superclass members**
2. **Calling superclass constructor**
3. **Invoking superclass methods**

\*\*Pseudocode:\*\*

BEGIN

// Define a superclass with some members and methods

DEFINE CLASS `SuperClass`

// Instance variable

DECLARE int superVariable

// Constructor of SuperClass

DEFINE CONSTRUCTOR `SuperClass(int value)`

SET this.superVariable = value

// Method in SuperClass

DEFINE FUNCTION `display()`

PRINT "Value from SuperClass: " + this.superVariable

// Define a subclass extending SuperClass

DEFINE CLASS `SubClass EXTENDS SuperClass`

// Instance variable

DECLARE int subVariable

// Constructor of SubClass

DEFINE CONSTRUCTOR `SubClass(int value1, int value2)`

CALL super(value1) // Calling SuperClass constructor

SET this.subVariable = value2

// Method in SubClass

DEFINE FUNCTION `display()`

CALL super.display() // Invoking SuperClass method

PRINT "Value from SubClass: " + this.subVariable

// Main program execution

DEFINE CLASS `Main`

DEFINE FUNCTION `main(String[] args)`

// Create an instance of SubClass

CREATE INSTANCE of `SubClass` with parameters 10 and 20

// Call display method to show values

CALL display() from SubClass instance

END

Java Code:

// Superclass

class SuperClass {

// Instance variable

int superVariable;

// Constructor of SuperClass

public SuperClass(int value) {

this.superVariable = value;

}

// Method in SuperClass

public void display() {

System.out.println("Value from SuperClass: " + superVariable);

}

}

// Subclass extending SuperClass

class SubClass extends SuperClass {

// Instance variable

int subVariable;

// Constructor of SubClass

public SubClass(int value1, int value2) {

super(value1); // Calling SuperClass constructor

this.subVariable = value2;

}

// Method in SubClass

@Override

public void display() {

super.display(); // Invoking SuperClass method

System.out.println("Value from SubClass: " + subVariable);

}

}

// Main class to demonstrate the use of 'super' keyword

public class Main {

public static void main(String[] args) {

// Create an instance of SubClass

SubClass subClassInstance = new SubClass(10, 20);

// Call display method to show values

subClassInstance.display();

}

}

13. Single type parameter generic class

\*\*Pseudocode:\*\*

BEGIN

// Define a generic class with a single type parameter

DEFINE CLASS `GenericClass<T>`

// Declare a variable of type T

DECLARE T data

// Constructor to initialize the data

DEFINE CONSTRUCTOR `GenericClass(T data)`

SET this.data = data

// Method to get the data

DEFINE FUNCTION `getData() RETURNS T`

RETURN this.data

// Method to display the data

DEFINE FUNCTION `displayData()`

PRINT "Data: " + this.data

// Main program execution

DEFINE CLASS `Main`

DEFINE FUNCTION `main(String[] args)`

// Create an instance of GenericClass with Integer type

CREATE INSTANCE of `GenericClass<Integer>` with parameter 100

CALL displayData() from Integer instance

// Create an instance of GenericClass with String type

CREATE INSTANCE of `GenericClass<String>` with parameter "Hello"

CALL displayData() from String instance

END

\*\*Java Code:\*\*

// Define a generic class with a single type parameter

class GenericClass<T> {

// Declare a variable of type T

private T data;

// Constructor to initialize the data

public GenericClass(T data) {

this.data = data;

}

// Method to get the data

public T getData() {

return data;

}

// Method to display the data

public void displayData() {

System.out.println("Data: " + data);

}

}

// Main class to demonstrate the use of GenericClass

public class Main {

public static void main(String[] args) {

// Create an instance of GenericClass with Integer type

GenericClass<Integer> integerInstance = new GenericClass<>(100);

integerInstance.displayData(); // Output: Data: 100

// Create an instance of GenericClass with String type

GenericClass<String> stringInstance = new GenericClass<>("Hello");

stringInstance.displayData(); // Output: Data: Hello

}

}

14. Multiple type parameter generic class

\*\*Pseudocode:\*\*

BEGIN

// Define a generic class with two type parameters

DEFINE CLASS `MultiGenericClass<K, V>`

// Declare variables of types K and V

DECLARE K key

DECLARE V value

// Constructor to initialize key and value

DEFINE CONSTRUCTOR `MultiGenericClass(K key, V value)`

SET this.key = key

SET this.value = value

// Method to get the key

DEFINE FUNCTION `getKey() RETURNS K`

RETURN this.key

// Method to get the value

DEFINE FUNCTION `getValue() RETURNS V`

RETURN this.value

// Method to display key and value

DEFINE FUNCTION `displayKeyValue()`

PRINT "Key: " + this.key

PRINT "Value: " + this.value

// Main program execution

DEFINE CLASS `Main`

DEFINE FUNCTION `main(String[] args)`

// Create an instance of MultiGenericClass with Integer and String types

CREATE INSTANCE of `MultiGenericClass<Integer, String>` with parameters 1 and "Hello"

CALL displayKeyValue() from Integer-String instance

// Create an instance of MultiGenericClass with String and Double types

CREATE INSTANCE of `MultiGenericClass<String, Double>` with parameters "Pi" and 3.14

CALL displayKeyValue() from String-Double instance

END

\*\*Java Code:\*\*

// Define a generic class with two type parameters

class MultiGenericClass<K, V> {

// Declare variables of types K and V

private K key;

private V value;

// Constructor to initialize key and value

public MultiGenericClass(K key, V value) {

this.key = key;

this.value = value;

}

// Method to get the key

public K getKey() {

return key;

}

// Method to get the value

public V getValue() {

return value;

}

// Method to display key and value

public void displayKeyValue() {

System.out.println("Key: " + key);

System.out.println("Value: " + value);

}

}

// Main class to demonstrate the use of MultiGenericClass

public class Main {

public static void main(String[] args) {

// Create an instance of MultiGenericClass with Integer and String types

MultiGenericClass<Integer, String> integerStringInstance = new MultiGenericClass<>(1, "Hello");

integerStringInstance.displayKeyValue(); // Output: Key: 1, Value: Hello

// Create an instance of MultiGenericClass with String and Double types

MultiGenericClass<String, Double> stringDoubleInstance = new MultiGenericClass<>("Pi", 3.14);

stringDoubleInstance.displayKeyValue(); // Output: Key: Pi, Value: 3.14

}

}

15. Using generics on methods example

\*\*Pseudocode:\*\*

BEGIN

// Define a class with a generic method

DEFINE CLASS `GenericMethods`

// Generic method to print an array of any type

DEFINE METHOD `printArray(T[] array)`

FOR EACH element IN array

PRINT element

// Main program execution

DEFINE CLASS `Main`

DEFINE FUNCTION `main(String[] args)`

// Create an instance of GenericMethods

CREATE INSTANCE of `GenericMethods`

// Call printArray with Integer type array

DECLARE Integer[] intArray = {1, 2, 3, 4, 5}

CALL printArray(intArray) from GenericMethods instance

// Call printArray with String type array

DECLARE String[] strArray = {"Apple", "Banana", "Cherry"}

CALL printArray(strArray) from GenericMethods instance

END

\*\*Java Code:\*\*

// Define a class with a generic method

class GenericMethods {

// Generic method to print an array of any type

public <T> void printArray(T[] array) {

for (T element : array) {

System.out.println(element);

}

}

}

// Main class to demonstrate the use of generic method

public class Main {

public static void main(String[] args) {

// Create an instance of GenericMethods

GenericMethods genericMethods = new GenericMethods();

// Call printArray with Integer type array

Integer[] intArray = {1, 2, 3, 4, 5};

genericMethods.printArray(intArray); // Output: 1 2 3 4 5

// Call printArray with String type array

String[] strArray = {"Apple", "Banana", "Cherry"};

genericMethods.printArray(strArray); // Output: Apple Banana Cherry

}

}

16. Restrict use of primitive types using generics

\*\*Pseudocode:\*\*

BEGIN

// Define a generic class with a single type parameter T

DEFINE CLASS `GenericClass<T>`

// Declare a variable of type T

DECLARE T data

// Constructor to initialize data

DEFINE CONSTRUCTOR `GenericClass(T data)`

SET this.data = data

// Method to get the data

DEFINE FUNCTION `getData() RETURNS T`

RETURN this.data

// Method to display the data

DEFINE FUNCTION `displayData()`

PRINT "Data: " + this.data

// Main program execution

DEFINE CLASS `Main`

DEFINE FUNCTION `main(String[] args)`

// Create an instance of GenericClass with Integer type (not primitive int)

CREATE INSTANCE of `GenericClass<Integer>` with parameter 100

CALL displayData() from Integer instance

// Create an instance of GenericClass with Double type (not primitive double)

CREATE INSTANCE of `GenericClass<Double>` with parameter 3.14

CALL displayData() from Double instance

// Attempting to use primitive types directly will result in a compile-time error

// Uncommenting the following line will cause a compilation error:

// CREATE INSTANCE of `GenericClass<int>` with parameter 5

END

\*\*Java Code:\*\*

// Define a generic class with a single type parameter T

class GenericClass<T> {

// Declare a variable of type T

private T data;

// Constructor to initialize data

public GenericClass(T data) {

this.data = data;

}

// Method to get the data

public T getData() {

return data;

}

// Method to display the data

public void displayData() {

System.out.println("Data: " + data);

}

}

// Main class to demonstrate the use of generic class with wrapper types

public class Main {

public static void main(String[] args) {

// Create an instance of GenericClass with Integer type (wrapper class)

GenericClass<Integer> integerInstance = new GenericClass<>(100);

integerInstance.displayData(); // Output: Data: 100

// Create an instance of GenericClass with Double type (wrapper class)

GenericClass<Double> doubleInstance = new GenericClass<>(3.14);

doubleInstance.displayData(); // Output: Data: 3.14

// Attempting to use primitive types directly will cause a compile-time error

// Uncommenting the following line will result in a compilation error:

// GenericClass<int> intInstance = new GenericClass<>(5); // Error: int cannot be used with generics

}

}

17. Use Scanner to get Char, String, Int, Float and Double input at same moment.

\*\*Pseudocode:\*\*

BEGIN

// Import the Scanner class

IMPORT java.util.Scanner

// Create a Scanner object to read input

DECLARE Scanner scanner

INITIALIZE scanner WITH new Scanner(System.in)

// Prompt user for a char input

PRINT "Enter a character: "

DECLARE charInput

SET charInput TO scanner.next().charAt(0)

// Prompt user for a string input

PRINT "Enter a string: "

DECLARE stringInput

SET stringInput TO scanner.next()

// Prompt user for an int input

PRINT "Enter an integer: "

DECLARE intInput

SET intInput TO scanner.nextInt()

// Prompt user for a float input

PRINT "Enter a float: "

DECLARE floatInput

SET floatInput TO scanner.nextFloat()

// Prompt user for a double input

PRINT "Enter a double: "

DECLARE doubleInput

SET doubleInput TO scanner.nextDouble()

// Print all collected inputs

PRINT "Character: " + charInput

PRINT "String: " + stringInput

PRINT "Integer: " + intInput

PRINT "Float: " + floatInput

PRINT "Double: " + doubleInput

END

\*\*Java Code:\*\*

import java.util.Scanner;

public class InputExample {

public static void main(String[] args) {

// Create a Scanner object to read input

Scanner scanner = new Scanner(System.in);

// Prompt user for a char input

System.out.print("Enter a character: ");

char charInput = scanner.next().charAt(0);

// Prompt user for a string input

System.out.print("Enter a string: ");

scanner.nextLine(); // Consume the newline left-over

String stringInput = scanner.nextLine();

// Prompt user for an int input

System.out.print("Enter an integer: ");

int intInput = scanner.nextInt();

// Prompt user for a float input

System.out.print("Enter a float: ");

float floatInput = scanner.nextFloat();

// Prompt user for a double input

System.out.print("Enter a double: ");

double doubleInput = scanner.nextDouble();

// Print all collected inputs

System.out.println("Character: " + charInput);

System.out.println("String: " + stringInput);

System.out.println("Integer: " + intInput);

System.out.println("Float: " + floatInput);

System.out.println("Double: " + doubleInput);

// Close the scanner

scanner.close();

}

}

18. Find System Date and Time using Date class

\*\*Pseudocode:\*\*

BEGIN

// Import the Date class

IMPORT java.util.Date

// Create an instance of Date to get the current system date and time

DECLARE Date currentDate

INITIALIZE currentDate WITH new Date()

// Print the current date and time

PRINT "Current date and time: " + currentDate.toString()

END

\*\*Java Code:\*\*

import java.util.Date;

public class DateExample {

public static void main(String[] args) {

// Create an instance of Date to get the current system date and time

Date currentDate = new Date();

// Print the current date and time

System.out.println("Current date and time: " + currentDate.toString());

}

}

19. Use UUID to generate a random Universally Unique Identifier

\*\*Pseudocode:\*\*

BEGIN

// Import the UUID class

IMPORT java.util.UUID

// Generate a random UUID

DECLARE UUID randomUUID

INITIALIZE randomUUID WITH UUID.randomUUID()

// Print the generated UUID

PRINT "Generated UUID: " + randomUUID.toString()

END

\*\*Java Code:\*\*

import java.util.UUID;

public class UUIDExample {

public static void main(String[] args) {

// Generate a random UUID

UUID randomUUID = UUID.randomUUID();

// Print the generated UUID

System.out.println("Generated UUID: " + randomUUID.toString());

}

}

20. Java toString() and equals() method

\*\*Pseudocode:\*\*

BEGIN

// Define a class with fields and override toString() and equals() methods

DEFINE CLASS `Person`

DECLARE private String name

DECLARE private int age

// Constructor to initialize fields

DEFINE CONSTRUCTOR `Person(String name, int age)`

SET this.name = name

SET this.age = age

// Override toString() method to provide a custom string representation

DEFINE FUNCTION `toString() RETURNS String`

RETURN "Person{name='" + this.name + "', age=" + this.age + "}"

// Override equals() method to compare Person objects based on name and age

DEFINE FUNCTION `equals(Object obj) RETURNS boolean`

IF obj IS NOT INSTANCE OF Person

RETURN false

CAST obj TO Person

DECLARE Person other = (Person) obj

RETURN this.name.equals(other.name) AND this.age == other.age

// Main program execution

DEFINE CLASS `Main`

DEFINE FUNCTION `main(String[] args)`

// Create two Person objects with the same data

CREATE Person person1 WITH "Alice", 30

CREATE Person person2 WITH "Alice", 30

// Print the string representation of the Person objects

PRINT person1.toString()

PRINT person2.toString()

// Compare the two Person objects

PRINT "Are person1 and person2 equal? " + person1.equals(person2)

END

**Java Code:**

public class Person {

private String name;

private int age;

// Constructor to initialize fields

public Person(String name, int age) {

this.name = name;

this.age = age;

}

// Override toString() method to provide a custom string representation

@Override

public String toString() {

return "Person{name='" + name + "', age=" + age + "}";

}

// Override equals() method to compare Person objects based on name and age

@Override

public boolean equals(Object obj) {

if (this == obj) {

return true; // Check if both references are the same

}

if (obj == null || getClass() != obj.getClass()) {

return false; // Check if obj is null or of a different class

}

Person other = (Person) obj;

return age == other.age && name.equals(other.name); // Compare name and age

}

// Main method to demonstrate the use of toString() and equals()

public static void main(String[] args) {

// Create two Person objects with the same data

Person person1 = new Person("Alice", 30);

Person person2 = new Person("Alice", 30);

// Print the string representation of the Person objects

System.out.println(person1.toString());

System.out.println(person2.toString());

// Compare the two Person objects

System.out.println("Are person1 and person2 equal? " + person1.equals(person2));

}

}