MST-1 Java

Q1. State whether the following statement is true/false with justification "when a reference variable is assigned to another reference variable, another copy of object is created.

Ans: The statement is false.

When a reference variable is assigned to another reference variable, no new copy of the object is created. Instead, both reference variables point to the same object in memory. This means that any changes made to the object through one reference will be reflected when accessed through the other reference, as both references refer to the same object.

```java
// Java example
ClassName obj1 = new ClassName();
ClassName obj2 = obj1; // obj2 refers to the same object as obj1
```

Here, obj1 and obj2 point to the same object. No new object is created when obj2 is assigned obj1.

Example:

```java
class Dog {
    String name;

    Dog(String name) {
        this.name = name;
    }

    void bark() {
        System.out.println(name + " is barking!");
    }
}

public class Main {
    public static void main(String[] args) {
        // Create a new Dog object and assign it to dog1 reference variable
        Dog dog1 = new Dog("Buddy");

        // Assign dog1 to another reference variable dog2
        Dog dog2 = dog1;

        // Both dog1 and dog2 point to the same object
        dog1.bark();  // Output: Buddy is barking!
        dog2.bark();  // Output: Buddy is barking!

        // Change the name using dog2
        dog2.name = "Max";

        // Both dog1 and dog2 reflect the name change because they reference the same obje
        dog1.bark();  // Output: Max is barking!
        dog2.bark();  // Output: Max is barking!
    }
}
```

Key Points:

1. dog1 is an object reference variable that points to a Dog object.
2. When you assign dog2 = dog1, both dog1 and dog2 reference the same object in memory.

3. Any changes made to the object through one reference variable (like dog2) will affect the object when accessed through the other reference (like dog1).

Q2."Write Once and Run Anywhere" - Support this statement with proper reasoning.

Ans: Java's **"Write Once, Run Anywhere"** principle means that a Java program, once written and compiled into bytecode, can run on any platform that has a Java Virtual Machine (JVM) without needing to be recompiled. The JVM interprets the bytecode and adapts it to the specific platform, making Java platform-independent and allowing the same code to run on various operating systems like Windows, macOS, and Linux.

Q3.If a class defines several constructors, is it feasible to invoke one constructor from another constructor? Justify your answer.

Ans: Yes, it is feasible to invoke one constructor from another constructor in Java. This is called constructor chaining and can be done using the this() keyword.

Justification:

In Java, constructor chaining allows one constructor to call another constructor within the same class to avoid code duplication and simplify initialization logic. This is done using this(), which must be the first statement in the constructor.

Example:

java Copy code

```java
class MyClass {
    int x;
    int y;

    // Constructor 1
    public MyClass() {
        this(5);  // Calls Constructor 2
        System.out.println("Default constructor");
    }

    // Constructor 2
    public MyClass(int x) {
        this.x = x;
        System.out.println("Constructor with one parameter: " + x);
    }
}
```

Q4.a) Rewrite the following program segment using ternary operator instead of the if-else statement.

String grade;

if(marks >= 90) {

grade = "A";

} else if(marks >= 80) {

grade = "B";

} else {

grade = "C";

}

Ans:

Ans:

```java
public class ternary_op
{
    int marks=98;
    String grade;
    String print_ternary()
    {
        // if(marks >= 90)
        // {
        //  return grade = "A";
        // }
        // else if(marks >= 80)
        // {
        //     return grade = "B";
        // }
        // else
        // {
        //  return grade = "C";
        // }

        return (marks>=90?"A":marks>=80?"B":"C");
    }
    Run | Debug
    public static void main(String[] args)
    {
        ternary_op obj=new ternary_op();
        System.out.println(obj.print_ternary());
    }
}
```

b) How to create objects? Does Java support object destruction? Support your answer.

Ans: In Java, objects are created using the new keyword, which allocates memory for the object and calls the constructor to initialize it.

Example:

ClassName obj = new ClassName();

Here:

- ClassName is the name of the class.
- obj is the reference variable pointing to the object.
- new ClassName() creates a new object by invoking the class constructor.

Does Java support object destruction?

Java does not support explicit object destruction like languages such as C++ (where you manually delete objects). Instead, Java uses automatic garbage collection, which automatically frees memory for objects that are no longer referenced by any part of the program.

Justification:

- The Java Garbage Collector (GC) is responsible for reclaiming memory by identifying objects that are no longer in use (i.e., objects without any active references) and cleaning them up.
- Developers cannot explicitly destroy objects, but they can make objects eligible for garbage collection by removing references to them (e.g., setting them to null).

Example:

MyClass obj = new MyClass();

obj = null; // Now the object is eligible for garbage collection

Q5.Compare and contrast method overloading with method overriding with the help of program(s).

Ans:

Aspect	Method Overloading	Method Overriding
Definition	Multiple methods with the same name but different parameters in the same class.	Subclass provides a specific implementation of a method defined in the parent class.
Polymorphism	Compile-time polymorphism (method determined at compile time).	Runtime polymorphism (method determined at runtime).
Purpose	To achieve functionality with different inputs.	To modify the behavior of a method in the child class.
Parameters	Must differ in number, type, or order of parameters.	Must have the same name, return type, and parameters.
Inheritance	No inheritance relationship is required.	Requires an inheritance relationship (parent-child class).
Return Type	Can have different return types.	Must have the same return type (or a covariant return type).
Access Modifier	Can have any access modifier.	The overriding method must have the same or more accessible modifier.
Annotation	No special annotation is required.	`@Override` annotation is often used (optional but recommended).
Example	Adding methods with different parameters in a calculator.	Animal subclasses (Dog, Cat) overriding the `sound()` method.

Example of Method Overriding:

java Copy code

```java
class Animal {
    // Method to be overridden
    public void sound() {
        System.out.println("The animal makes a sound");
    }
}

class Dog extends Animal {
    // Overriding the sound method in the Dog class
    @Override
    public void sound() {
        System.out.println("The dog barks");
    }
}

class Cat extends Animal {
    // Overriding the sound method in the Cat class
    @Override
    public void sound() {
        System.out.println("The cat meows");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Animal();  // Animal reference, Animal object
        Animal myDog = new Dog();        // Animal reference, Dog object
        Animal myCat = new Cat();        // Animal reference, Cat object

        myAnimal.sound();   // Calls Animal's sound method
        myDog.sound();      // Calls Dog's sound method (overridden)
        myCat.sound();      // Calls Cat's sound method (overridden)
    }
}
```

Example of Method Overloading:

```java
class Calculator {
    // Method to add two integers
    public int add(int a, int b) {
        return a + b;
    }

    // Overloaded method to add three integers
    public int add(int a, int b, int c) {
        return a + b + c;
    }

    // Overloaded method to add two double values
    public double add(double a, double b) {
        return a + b;
    }

    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println("Add two integers: " + calc.add(5, 10));          // Calls add(i
        System.out.println("Add three integers: " + calc.add(5, 10, 15));     // Calls add(
        System.out.println("Add two doubles: " + calc.add(2.5, 3.5));        // Calls add(d
    }
}
```

Q6. What is the meaning of first line of java program "public static void main (String args[])?

Ans: The line public static void main(String[] args) is the entry point of a Java application. Here's a breakdown of its components:

- public: This is an access modifier that means the main method is accessible from anywhere. It is necessary because the Java Virtual Machine (JVM) needs to call this method from outside the class.
- static: This keyword means that the main method belongs to the class, rather than instances of the class. It can be called without creating an object of the class. The JVM uses this method to start the execution of the program.
- void: This is the return type of the method. void means that the method does not return any value.
- main: This is the name of the method. The JVM looks for the main method when starting the execution of a Java program.
- String[] args: This is a parameter to the main method. It represents an array of String objects, which can be used to pass command-line arguments to the program. args is the name of the parameter, and it can be any valid identifier.

Q7. Difference between recursion and iteration with valid program.

Aspect	Recursion	Iteration
Definition	A method that calls itself to solve a problem.	A method that uses loops to repeat a block of code.
Approach	Breaks the problem into smaller instances of the same problem.	Repeats a block of code until a condition is met.
Memory Usage	Uses more memory due to stack space for multiple function calls.	Uses less memory as it involves simple loops.
Performance	May be slower due to the overhead of recursive calls.	Generally faster and more efficient due to direct loop execution.
Base Case	Requires a base case to terminate recursion.	Uses loop conditions to control termination.
Function Calls	Involves multiple function calls (recursive calls).	Involves only one loop construct.
Readability	Can be more elegant for problems with a natural recursive structure (e.g., tree traversal).	Often more straightforward and easy to understand for problems solvable with loops.
Examples	Factorial calculation, Fibonacci sequence.	Factorial calculation using a loop, iterating over arrays.
Stack Overflow	Can cause stack overflow for deep recursion levels.	Does not have stack overflow issues as it uses loops.

```java
public class RecursionExample {
    // Recursive method to find factorial
    public static int factorial(int n) {
        // Base case
        if (n == 0) {
            return 1;
        }
        // Recursive case
        return n * factorial(n - 1);
    }

    public static void main(String[] args) {
        int number = 5;
        System.out.println("Factorial of " + number + " is: " + factorial(number));
    }
}
```

```java
public class IterationExample {
    // Iterative method to find factorial
    public static int factorial(int n) {
        int result = 1;
        for (int i = 1; i <= n; i++) {
            result *= i;
        }
        return result;
    }

    public static void main(String[] args) {
        int number = 5;
        System.out.println("Factorial of " + number + " is: " + factorial(number));
    }
}
```

Q8.Explain different types of operators used in java programming. Write a simple program to demonstrate operator precedence in java.

Ans: In Java, operators are special symbols that perform operations on variables and values. Here's a summary of the different types of operators in Java:

1. Arithmetic Operators

These operators are used to perform basic arithmetic operations.

- + (Addition)

- - (Subtraction)
- * (Multiplication)
- / (Division)
- % (Modulus)

2. Relational Operators

These operators are used to compare two values.

- == (Equal to)
- != (Not equal to)
- > (Greater than)
- < (Less than)
- >= (Greater than or equal to)
- <= (Less than or equal to)

3. Logical Operators

These operators are used to perform logical operations.

- && (Logical AND)
- || (Logical OR)
- ! (Logical NOT)

4. Bitwise Operators

These operators are used to perform bit-level operations.

- & (Bitwise AND)
- | (Bitwise OR)
- ^ (Bitwise XOR)
- ~ (Bitwise NOT)
- << (Left shift)
- >> (Right shift)
- >>> (Unsigned right shift)

5. Assignment Operators

These operators are used to assign values to variables.

- = (Simple assignment)
- += (Add and assign)
- -= (Subtract and assign)

- *= (Multiply and assign)
- /= (Divide and assign)
- %= (Modulus and assign)

6. Unary Operators

These operators operate on a single operand.

- + (Unary plus)
- - (Unary minus)
- ++ (Increment)
- -- (Decrement)
- ! (Logical NOT)

7. Ternary Operator

This operator is a shorthand for the if-else statement.

- ?: (Conditional operator)

8. Instanceof Operator

This operator is used to check whether an object is an instance of a specific class or subclass.

- instanceof

Example Program: Operator Precedence

Here's a simple Java program that demonstrates operator precedence:

```
public class OperatorPrecedenceDemo {
    public static void main(String[] args) {
        int a = 10;
        int b = 5;
        int c = 2;
        // Example of different operators and their precedence
        int result = a + b * c - (a / c) % c;
        System.out.println("Result: " + result);
        // Demonstrate logical operators
        boolean x = true;
        boolean y = false;
        boolean logicalResult = x && (y || (x && !y));
        System.out.println("Logical Result: " + logicalResult);
```

```
        // Demonstrate ternary operator
        int max = (a > b) ? a : b;
        System.out.println("Max value: " + max);
    }
}
```

The output of the program will be:

Result: 12

Logical Result: true

Max value: 10

Syllabus part A

1.comparison of procedural programming and object-oriented programming (OOP).

Aspect	Procedural Programming	Object-Oriented Programming (OOP)
Concept	Focuses on functions or procedures.	Focuses on objects and classes.
Structure	Organized around routines or functions.	Organized around objects and classes.
Data Handling	Data is usually separate from functions.	Data is encapsulated within objects.
Approach	Top-down approach.	Bottom-up approach.
Reusability	Code reuse is achieved through functions and libraries.	Code reuse is achieved through inheritance and polymorphism.
Data Protection	Data is exposed and can be modified by functions.	Data is protected and modified through methods (encapsulation).
Design	Focuses on actions or procedures to process data.	Focuses on modeling real-world entities as objects.
Modularity	Code is divided into functions or procedures.	Code is divided into classes and objects.
Inheritance	Not supported.	Supports inheritance, allowing new classes to inherit properties and behaviors from existing classes.
Polymorphism	Limited to function overloading.	Supports method overriding and operator overloading.
Example Languages	C, Pascal, Fortran.	Java, C++, Python, Ruby.
State Management	State is managed through global or local variables.	State is managed within objects and their attributes.
Code Maintenance	Code maintenance can be challenging due to lack of organization.	Code maintenance is easier due to encapsulation and modularity.
Flexibility	Changes in code may require changes in multiple places.	Changes in one part of the code (class) don't necessarily affect others.
Focus	Procedure-centric, focusing on the sequence of operations.	Object-centric, focusing on interactions between objects.

2. Principles of Object Oriented Programming in java.

principles with brief explanations:

1. **Encapsulation**

   Java Implementation: Encapsulation is implemented using access modifiers (e.g., private, protected, public). The data is typically made private, and access is provided through public getter and setter methods.

   - Example:

```
public class Person {
   private String name; // Private field
   public String getName() { // Getter method
      return name;
   }
   public void setName(String name) { // Setter method
      this.name = name;
   }
}
```

2. **Inheritance**

   - Definition: Inheritance allows a new class (subclass) to inherit properties and behaviors (methods) from an existing class (superclass). This promotes code reuse and establishes a natural hierarchy.
   - Java Implementation: Java supports single inheritance, where a class can inherit from one superclass using the extends keyword.
   - Example:

```
public class Animal {
   public void eat() {
      System.out.println("This animal eats food.");
   }
}
public class Dog extends Animal {
   public void bark() {
      System.out.println("The dog barks.");
   }
}
```

3. **Polymorphism**
    - Java Implementation: Polymorphism is achieved through method **overloading** and method **overriding**.
        - Method Overloading: Multiple methods with the same name but different parameters.

```
public class MathOperations {
   public int add(int a, int b) {
      return a + b;
   }
   public double add(double a, double b) {
      return a + b;
   }
}
```

        - Method Overriding: Subclasses provide a specific implementation of a method already defined in its superclass.

```
public class Animal {
   public void makeSound() {
      System.out.println("Some sound");
   }
}
public class Dog extends Animal {
   @Override
   public void makeSound() {
      System.out.println("Bark");
   }
}
```

4. **Abstraction**
    - Definition: Abstraction is the concept of hiding the complex implementation details and showing only the necessary features of an object. It focuses on what an object does rather than how it does it.
    - Java Implementation: Abstraction is achieved using abstract classes and interfaces.

- Abstract Classes: Classes that cannot be instantiated on their own and may contain abstract methods (methods without a body) that must be implemented by subclasses.

```
public abstract class Shape {
   public abstract void draw();
}
public class Circle extends Shape {
   @Override
   public void draw() {
      System.out.println("Drawing a circle.");
   }
}
```

- Interfaces: Provide a way to achieve abstraction and multiple inheritance. Interfaces can have abstract methods and default methods.

```
public interface Drawable {
   void draw();
}
public class Rectangle implements Drawable {
   @Override
   public void draw() {
      System.out.println("Drawing a rectangle.");
   }
}
```

Q3. Type casting: Type casting in Java allows you to convert variables from one type to another. There are two main types of casting:

1. **Widening Casting** (Automatic Type Conversion)
2. **Narrowing Casting** (Explicit Type Conversion)

1. **Widening Casting**

Definition: Widening casting is when you convert a smaller data type to a larger data type. This type of casting is done automatically by the Java compiler. It does not lose data and is safe.

Example:

- Converting int to long, float, or double.
- Converting char to int.

Syntax: No explicit syntax is needed; it's done automatically.

Example Code:

```
public class WideningCastingExample {
   public static void main(String[] args) {
      int intValue = 10;
      long longValue = intValue;  // Widening casting: int to long
      float floatValue = intValue; // Widening casting: int to float
      double doubleValue = intValue; // Widening casting: int to double
      System.out.println("intValue: " + intValue);
      System.out.println("longValue: " + longValue);
      System.out.println("floatValue: " + floatValue);
      System.out.println("doubleValue: " + doubleValue);
   }
}
```

2. **Narrowing Casting**

Definition: Narrowing casting is when you convert a larger data type to a smaller data type. This type of casting requires explicit syntax and can potentially lose data or cause precision loss.

Syntax: You need to use a cast operator in parentheses before the variable you want to cast.

Example:

- Converting double to float, long, int, or short.
- Converting char to byte or short.

Example Code:

```
public class NarrowingCastingExample {
   public static void main(String[] args) {
      double doubleValue = 10.5;
      int intValue = (int) doubleValue;  // Narrowing casting: double to int
      float floatValue = (float) doubleValue; // Narrowing casting: double to float
      byte byteValue = (byte) doubleValue; // Narrowing casting: double to byte
```

```
        System.out.println("doubleValue: " + doubleValue);
        System.out.println("intValue: " + intValue);
        System.out.println("floatValue: " + floatValue);
        System.out.println("byteValue: " + byteValue);
    }
}
```

4.Data Types: Primitive Data Types

Numeric

1. byte
    - Size: 1 byte (8 bits)
    - Range: -128 to 127
    - Example: byte b = 100;
2. short
    - Size: 2 bytes (16 bits)
    - Range: -32,768 to 32,767
    - Example: short s = 10000;
3. int
    - Size: 4 bytes (32 bits)
    - Range: -2^31 to 2^31 - 1 (-2,147,483,648 to 2,147,483,647)
    - Example: int i = 100000;
4. long
    - Size: 8 bytes (64 bits)
    - Range: -2^63 to 2^63 - 1 (-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)
    - Example: long l = 10000000000L; (Note the L suffix to denote a long literal)
5. float
    - Size: 4 bytes (32 bits)
    - Range: Approximately ±3.40282347E+38F (6-7 significant decimal digits)
    - Example: float f = 3.14f; (Note the f suffix to denote a float literal)
6. double
    - Size: 8 bytes (64 bits)

- Range: Approximately ±1.79769313486231570E+308 (15-16 significant decimal digits)
- Example: double d = 3.141592653589793;

Non-Numeric

7. char
    - Size: 2 bytes (16 bits)
    - Range: 0 to 65,535 (Unicode characters)
    - Example: char c = 'A';
8. boolean
    - Size: The size is JVM-dependent (usually 1 byte for convenience)
    - Range: true or false
    - Example: boolean b = true;

Reference Data Types

Reference data types include **arrays**, **classes**, and **interfaces**. They do not have a predefined size but rather a size that depends on the object or array's size and the JVM implementation.

- Example of an Array: int[] numbers = new int[5];
- Example of a Class: class Person { String name; int age; }
- Example of an Interface: interface Vehicle { void drive(); }

6.Arrays:

One-Dimensional Array

A one-dimensional array is essentially a list of elements of the same type.

Declaration and Initialization

// Declaration

Int numbers[];

// Initialization

numbers = new int[5]; // An array of 5 integers

// Declaration and Initialization in one line

int ages[] = {10, 20, 30, 40, 50};

Accessing and Modifying Elements

// Accessing elements

System.out.println(ages[0]); // Output: 10

```
// Modifying elements
ages[1] = 25;
System.out.println(ages[1]); // Output: 25
```

Example: Sum of Array Elements

```
public class OneDArrayExample {
    public static void main(String[] args) {
        int numbers[] = {1, 2, 3, 4, 5};
        int sum = 0;
        for (int i = 0; i < numbers.length; i++) {
            sum += numbers[i];
        }
        System.out.println("Sum: " + sum); // Output: Sum: 15
    }
}
```

2. Two-Dimensional Array

A two-dimensional array is an array of arrays. It can be visualized as a table with rows and columns.

Declaration and Initialization

```
// Declaration
int matrix[][];
// Initialization
matrix = new int[3][4]; // 3 rows and 4 columns
// Declaration and Initialization in one line
int predefinedMatrix[][] = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
};
```

Accessing and Modifying Elements

```
// Accessing elements
System.out.println(predefinedMatrix[0][2]); // Output: 3
```

```
// Modifying elements
predefinedMatrix[1][1] = 15;
System.out.println(predefinedMatrix[1][1]); // Output: 15
```

Example: Printing a Two-Dimensional Array

```
public class TwoDArrayExample {
    public static void main(String[] args) {
        int[][] matrix = {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        };

        for (int i = 0; i < matrix.length; i++) {
            for (int j = 0; j < matrix[i].length; j++) {
                System.out.print(matrix[i][j] + " ");
            }
            System.out.println();
        }
        // Output:
        // 1 2 3
        // 4 5 6
        // 7 8 9
    }
}
```

8.Selection Statements, Control Statements:

1. Selection Statements

Selection statements allow you to choose between different paths of execution based on conditions.

if Statement

The if statement executes a block of code if its condition is true.

```
public class IfExample {
   public static void main(String[] args) {
      int number = 10;

      if (number > 0) {
         System.out.println("The number is positive.");
      }
   }
}
```

if-else Statement

The if-else statement executes one block of code if the condition is true and another block if it is false.

```
public class IfElseExample {
   public static void main(String[] args) {
      int number = -5;
      if (number >= 0) {
         System.out.println("The number is non-negative.");
      } else {
         System.out.println("The number is negative.");
      }
   }
}
```

if-else if-else Statement

The if-else if-else statement allows you to handle multiple conditions.

```
public class IfElseIfExample {
   public static void main(String[] args) {
      int number = 0;
      if (number > 0) {
         System.out.println("The number is positive.");
      } else if (number < 0) {
         System.out.println("The number is negative.");
```

```
    } else {
        System.out.println("The number is zero.");
    }
  }
}
```

switch Statement

The switch statement selects one of many code blocks to be executed.

```
public class SwitchExample {
  public static void main(String[] args) {
    int day = 3;
    switch (day) {
      case 1:
        System.out.println("Monday");
        break;
      case 2:
        System.out.println("Tuesday");
        break;
      case 3:
        System.out.println("Wednesday");
        break;
      case 4:
        System.out.println("Thursday");
        break;
      case 5:
        System.out.println("Friday");
        break;
      case 6:
        System.out.println("Saturday");
        break;
      case 7:
        System.out.println("Sunday");
```

```
            break;
        default:
            System.out.println("Invalid day");
            break;
    }
  }
}
```

2. Control Statements

Control statements alter the flow of execution.

for Loop

The for loop executes a block of code a specific number of times.

```
public class ForLoopExample {
  public static void main(String[] args) {
    for (int i = 1; i <= 5; i++) {
      System.out.println("Iteration: " + i);
    }
  }
}
```

while Loop

The while loop executes a block of code as long as its condition is true.

```
public class WhileLoopExample {
  public static void main(String[] args) {
    int i = 1;
    while (i <= 5) {
      System.out.println("Iteration: " + i);
      i++;
    }
  }
}
```

do-while Loop

The do-while loop executes a block of code once before checking the condition, and then repeatedly executes the block as long as the condition is true.

```
public class DoWhileLoopExample {
    public static void main(String[] args) {
        int i = 1;
        do {
            System.out.println("Iteration: " + i);
            i++;
        } while (i <= 5);
    }
}
```

break Statement

The break statement exits from the loop or switch statement.

```
public class BreakExample {
    public static void main(String[] args) {
        for (int i = 1; i <= 5; i++) {
            if (i == 3) {
                break;
            }
            System.out.println("Iteration: " + i);
        }
    }
}
```

continue Statement

The continue statement skips the current iteration of the loop and proceeds with the next iteration.

```
public class ContinueExample {
    public static void main(String[] args) {
        for (int i = 1; i <= 5; i++) {
            if (i == 3) {
                continue;
            }
```

```
            System.out.println("Iteration: " + i);
        }
    }
}
```

10. Constructors: In Java, a constructor is a special type of method that is used to initialize objects. It has the same name as the class and does not have a return type, not even void. Constructors are invoked when an object of a class is created, allowing you to set initial values for object attributes.

Types of Constructors in Java:

1. Default Constructor:
    - A constructor that is automatically provided by Java if no other constructors are explicitly defined by the programmer.
    - It does not take any arguments and initializes object fields with default values (e.g., 0 for numeric fields, null for object references).

Example:

```
class Dog {
    String name;
    int age;

    // Default Constructor (provided automatically if not defined)
    Dog() {
        System.out.println("Dog created!");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog1 = new Dog(); // Calls default constructor
    }
}
```

2. Parameterized Constructor:
    - A constructor that takes arguments to initialize the object with specific values.
    - It allows initializing object attributes with values passed when the object is created.

Example:

```
class Dog {
   String name;
   int age;
   // Parameterized Constructor
   Dog(String name, int age) {
      this.name = name;
      this.age = age;
   }
}
public class Main {
   public static void main(String[] args) {
      Dog dog2 = new Dog("Buddy", 3); // Calls parameterized constructor
      System.out.println("Dog's name: " + dog2.name + ", Age: " + dog2.age);
   }
}
```

3. No-Argument Constructor:
    - A constructor with no parameters, often used to initialize objects with default values or to do some default setup.

Example:

```
class Car {
   String model;
   int year;
   // No-Argument Constructor
   Car() {
      this.model = "Unknown";
      this.year = 2020;
   }
}


public class Main {
```

```
    public static void main(String[] args) {
        Car car1 = new Car(); // Calls no-argument constructor
        System.out.println("Car model: " + car1.model + ", Year: " + car1.year);
    }
}
```

4. **Copy Constructor:**
    - A constructor that creates a new object by copying another object's properties.
    - Java does not provide a default copy constructor, but it can be explicitly defined by the programmer.

Example:

```
class Person {
    String name;
    int age;
    // Parameterized Constructor
    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    // Copy Constructor
    Person(Person person) {
        this.name = person.name;
        this.age = person.age;
    }
}
public class Main {
    public static void main(String[] args) {
        Person person1 = new Person("John", 25);
        Person person2 = new Person(person1); // Calls copy constructor
        System.out.println("Person 2's name: " + person2.name + ", Age: " + person2.age);
    }
}
```

11.Objects as Parameters, Returning Objects in java:

1. Objects as Parameters

When you pass an object to a method, the reference to that object is passed (not a copy of the object). This means any changes made to the object inside the method will affect the original object outside the method as well.

Example: Passing Object as a Parameter

```
class Rectangle {
   int length;
   int width;
   Rectangle(int l, int w) {
      length = l;
      width = w;
   }
   void displayDimensions() {
      System.out.println("Length: " + length + ", Width: " + width);
   }
}
class Main {
   // Method to change the dimensions of the rectangle
   static void modifyRectangle(Rectangle r) {
      r.length = 10;
      r.width = 5;
   }
   public static void main(String[] args) {
      Rectangle rect = new Rectangle(4, 2);
      rect.displayDimensions();  // Before modification
      // Passing object as a parameter
      modifyRectangle(rect);
      rect.displayDimensions();  // After modification
   }
}
```

Output:

Length: 4, Width: 2

Length: 10, Width: 5

In this example, the Rectangle object rect is passed to the modifyRectangle method. The method modifies the length and width, which reflects in the original rect object.

2. Returning Objects

A method can also return an object. You typically use this when creating or modifying objects in a method and want to pass the modified or newly created object back to the caller.

Example: Returning an Object from a Method

```
class Circle {
    double radius;
    Circle(double r) {
        radius = r;
    }
    void displayRadius() {
        System.out.println("Radius: " + radius);
    }
}
class Main {
    // Method that returns a new Circle object
    static Circle createCircle(double r) {
        return new Circle(r);  // Returning a Circle object
    }
    public static void main(String[] args) {
        // Call method and assign returned object
        Circle circle = createCircle(7.5);
        circle.displayRadius();  // Display radius of the new Circle
    }
}
```

Output:

Radius: 7.5

In this example, the createCircle method returns a Circle object, which is then used in the main method to display the radius.

11.Overloading Constructors: In Java, constructor overloading allows a class to have multiple constructors, each with a different set of parameters. This enables the creation of objects in different ways depending on the provided arguments. Constructor overloading follows the same principle as method overloading, where each constructor must have a unique parameter list (either in type, number, or order of parameters).

Example of Constructor Overloading

```
class Employee {
   String name;
   int id;
   double salary;
   // Constructor 1: No parameters
   public Employee() {
      name = "Unknown";
      id = 0;
      salary = 0.0;
   }
   // Constructor 2: Only name
   public Employee(String name) {
      this.name = name;
      id = 0;
      salary = 0.0;
   }
   // Constructor 3: Name and id
   public Employee(String name, int id) {
      this.name = name;
      this.id = id;
      salary = 0.0;
   }
   // Constructor 4: Name, id, and salary
   public Employee(String name, int id, double salary) {
      this.name = name;
      this.id = id;
```

```
        this.salary = salary;
    }
    // Display employee details
    public void display() {
        System.out.println("Name: " + name + ", ID: " + id + ", Salary: " + salary);
    }
}
public class Main {
    public static void main(String[] args) {
        // Using different constructors
        Employee emp1 = new Employee();
        Employee emp2 = new Employee("John");
        Employee emp3 = new Employee("Alice", 123);
        Employee emp4 = new Employee("Bob", 456, 55000.00);
        // Displaying the employee details
        emp1.display();
        emp2.display();
        emp3.display();
        emp4.display();
    }
}
```

Output:

Name: Unknown, ID: 0, Salary: 0.0

Name: John, ID: 0, Salary: 0.0

Name: Alice, ID: 123, Salary: 0.0

Name: Bob, ID: 456, Salary: 55000.0

12.This Keyword :In Java, the this keyword is a reference to the current object, used primarily in object-oriented programming to avoid ambiguity between class attributes and parameters, or to refer to the instance of the current class.

Here are some common uses of the this keyword in Java:

1. Distinguishing Between Instance Variables and Parameters

When a constructor or method has a parameter with the same name as an instance variable, this is used to refer to the instance variable.

```
public class Example {
   int num;
   // Constructor
   public Example(int num) {
      this.num = num; // 'this.num' refers to the instance variable, 'num' refers to the parameter
   }
}
```

2. Calling One Constructor from Another (Constructor Chaining)

The this() syntax can be used to call another constructor in the same class.

```
public class Example {
   int num;
   String text;
   // Default constructor
   public Example() {
      this(0, "Default"); // Calls the parameterized constructor
   }
   // Parameterized constructor
   public Example(int num, String text) {
      this.num = num;
      this.text = text;
   }
}
```

3. Referring to the Current Object

this can be used to pass the current object as an argument to another method.

```
public class Example {
   public void print() {
      System.out.println(this); // Refers to the current object
   }
}
```

4. Returning the Current Object

You can use this to return the current object, which can be useful in method chaining.

```
public class Example {
   public Example setValue(int value) {
      this.num = value;
      return this; // Returns the current object
   }
}
```

5. Calling Methods of the Current Class

You can use this to explicitly call a method of the current class.

```
public class Example {
   public void method1() {
      System.out.println("Method 1");
   }
   public void method2() {
      this.method1(); // Calls method1() of the current object
   }
}
```

13.Command Line Arguments :In Java, command-line arguments are passed to the main method of your program, which is defined as public static void main(String[] args). The args parameter is an array of String objects that contains the arguments passed to the program.

Here's a simple example to illustrate how to use command-line arguments in Java:

```
public class CommandLineExample {
   public static void main(String[] args) {
      // Check if any arguments were passed
      if (args.length > 0) {
         System.out.println("Command-line arguments passed:");
         // Iterate through the arguments and print them
         for (int i = 0; i < args.length; i++) {
            System.out.println("Argument " + i + ": " + args[i]);
         }
      } else {
```

```
            System.out.println("No command-line arguments passed.");
        }
    }
}
```

14.**Inheritance**

Inheritance is a key feature of Object-Oriented Programming (OOP) that allows one class to inherit the properties and behavior (methods) of another class. The class that inherits is called the **subclass (or child class)**, and the class from which it inherits is called the **superclass (or parent class)**.

Syntax:

```
class Parent {
    // Parent class properties and methods
}
class Child extends Parent {
    // Child class can access Parent class properties and methods
}
```

Example of Inheritance:

```
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}
class Dog extends Animal {
    void bark() {
        System.out.println("The dog barks.");
    }
}
public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat();  // Inherited from Animal class
        dog.bark(); // Defined in Dog class
    }
```

}

15.super Keyword

The super keyword in Java is used to refer to the immediate parent class. It is often used for:

- **Accessing parent class methods** that have been overridden in the child class.
- **Calling parent class constructors**.
- **Accessing Parent Class Variables:**

  1. Calling the Parent Class Constructor:

  The super() can be used to call the constructor of the parent class from the child class. If it's not explicitly called, Java automatically calls the default constructor of the parent class (if it exists).

```
class Parent {
   Parent() {
      System.out.println("Parent constructor called");
   }
}
class Child extends Parent {
   Child() {
      super();  // Calls the Parent class constructor
      System.out.println("Child constructor called");
   }
}
public class Test {
   public static void main(String[] args) {
      Child c = new Child();
   }
}
```

  Output:

  Parent constructor called

  Child constructor called

  2. Accessing Parent Class Methods:

  The super keyword can also be used to call a method from the parent class when the child class has overridden it.

```
class Parent {
   void display() {
      System.out.println("Display method of Parent class");
   }
}
class Child extends Parent {
   void display() {
      super.display();  // Calls the Parent class display method
      System.out.println("Display method of Child class");
   }
}
public class Test
{
   public static void main(String[] args) {
      Child c = new Child();
      c.display();
   }
}
```

Output:

Display method of Parent class

Display method of Child class

3. Accessing Parent Class Variables:

If the child class defines a variable with the same name as the parent class, the super keyword can be used to access the parent class version of the variable.

```
class Parent {
   int value = 10;
}
class Child extends Parent {
   int value = 20;


   void showValues() {
```

```
        System.out.println("Value in Child class: " + value);
        System.out.println("Value in Parent class: " + super.value);
    }
}
public class Test {
    public static void main(String[] args) {
        Child c = new Child();
        c.showValues();
    }
}
```

Output:

Value in Child class: 20

Value in Parent class: 10

16. Dynamic Method Dispatch: Dynamic Method Dispatch in Java is a mechanism that implements runtime polymorphism. It occurs when a method is overridden in a subclass, and the method to be executed is determined at runtime based on the type of object (rather than the reference type).

Key Concepts:

1. Inheritance: Dynamic method dispatch relies on inheritance where a subclass overrides a method in the parent class.
2. Method Overriding: The subclass provides a specific implementation of a method that is already defined in its superclass.
3. Upcasting: The reference variable of the parent class refers to the object of the child class.

How Dynamic Method Dispatch Works:

- When a method is called on a parent class reference, which refers to a child class object, the overridden method in the child class is called at runtime.
- The decision of which method to call is made dynamically during program execution, hence the term dynamic dispatch.

Example Code:

```
class Parent {
  void show() {
     System.out.println("Parent's show method");
  }
}
```

```
class Child extends Parent {
    @Override
    void show() {
        System.out.println("Child's show method");
    }
}
public class Test {
    public static void main(String[] args) {
        Parent obj = new Child(); // Upcasting
        obj.show(); // Calls Child's show method
    }
}
```

Explanation:

- Parent obj = new Child();: The reference variable obj is of type Parent, but it refers to an instance of Child.
- obj.show();: Even though the reference is of type Parent, the overridden method in Child is called. This happens because at runtime, the object type is checked, not the reference type.

17. Abstract Classes: In Java, **abstract classes** are classes that cannot be instantiated on their own and are meant to be extended by other classes. They are used to define common behaviors that multiple subclasses can inherit while allowing each subclass to provide specific implementations for abstract methods.

Key Points of Abstract Classes in Java:

1. **Cannot be Instantiated**: You cannot create an object of an abstract class directly.

```
abstract class Animal {
    abstract void sound();
}
// Animal a = new Animal(); // This will cause an error
```

2. **Abstract Methods**: These are methods that are declared without a body in the abstract class. Subclasses must provide an implementation for these methods.

```
abstract class Animal {
    // Abstract method
    abstract void sound();
}
```

```
class Dog extends Animal {
  @Override
  void sound() {
    System.out.println("Woof");
  }
}
```

3. **Non-Abstract Methods**: Abstract classes can also contain methods with implementations. These methods can be inherited and used directly by subclasses.

```
abstract class Animal {
  abstract void sound();
  // Non-abstract method
  void sleep() {
    System.out.println("Sleeping...");
  }
}
class Cat extends Animal {
  @Override
  void sound() {
    System.out.println("Meow");
  }
}
```

4. **Partial Implementation**: Abstract classes allow you to provide partial implementations, meaning some methods may be fully implemented, and others (abstract methods) may not. Subclasses can choose which ones to override or leave as-is.
5. **Constructors in Abstract Classes**: Abstract classes can have constructors, and these constructors are called when an instance of a subclass is created.

```
abstract class Animal {
  Animal() {
    System.out.println("Animal is created");
  }
  abstract void sound();
}
```

```
class Dog extends Animal {
    Dog() {
        System.out.println("Dog is created");
    }
    @Override
    void sound() {
        System.out.println("Woof");
    }
}
```

6. **Extending Abstract Classes**: A class that extends an abstract class must either implement all of its abstract methods or be declared abstract itself.

Example:

```
abstract class Animal {
    // Abstract method (does not have a body)
    abstract void sound();
    // Regular method
    void eat() {
        System.out.println("Eating...");
    }
}
class Dog extends Animal {
    // Provide implementation for the abstract method
    void sound() {
        System.out.println("Woof Woof");
    }
}
class Cat extends Animal {
    // Provide implementation for the abstract method
    void sound() {
        System.out.println("Meow");
    }
```

```
}

public class Main {

   public static void main(String[] args) {

      Animal dog = new Dog();

      dog.sound();  // Outputs: Woof Woof

      dog.eat();    // Outputs: Eating...

      Animal cat = new Cat();

      cat.sound();  // Outputs: Meow

      cat.eat();    // Outputs: Eating...

   }

}
```

17.Using Final with Inheritance: In Java, the final keyword can be used in conjunction with inheritance to enforce certain restrictions. The final keyword can be applied to classes, methods, and variables. Here's how it works in the context of inheritance:

1. Final Classes:

A class marked as final cannot be subclassed, meaning no other class can inherit from it.

Example:

```
final class FinalClass {

   public void display() {

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

   }

}

// This will cause a compile-time error:

// class SubClass extends FinalClass {}
```

In this case, any attempt to create a subclass of FinalClass will result in a compile-time error.

2. Final Methods:

A method marked as final cannot be overridden by subclasses. This is useful when you want to prevent specific behavior in a superclass from being changed by subclasses.

Example:

```
class Parent {

   public final void show() {

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

```
}
}
class Child extends Parent {
   // This will cause a compile-time error:
   // public void show() {
   //     System.out.println("Trying to override final method.");
   // }
}
```

Here, the show() method in the Parent class cannot be overridden by the Child class.

3. Final Variables:

A final variable acts like a constant. Once initialized, it cannot be changed. In terms of inheritance, if an instance variable in a class is final, subclasses cannot modify its value after it's initialized.

Example:

```
class Parent {
   public final int constant = 100;
   public void display() {
      System.out.println("Constant: " + constant);
   }
}
class Child extends Parent {
   public void changeConstant() {
      // This will cause a compile-time error:
      // constant = 200;
   }
}
```

18.Menu Driven calculator program:

```
import java.util.Scanner;
class methods {
   static int Addition(int x, int y) {
      return (x + y);
   }
```

```
static int Subtraction(int x, int y) {
    return (x - y);
 }
 static int Multiplication(int x, int y) {
    return (x * y);
 }
 static int Division(int x, int y) {
    return (x / y);
 }
 public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    int choice;
    do {
       // Display menu
       System.out.println("Menu:");
       System.out.println("1. Addition");
       System.out.println("2. Subtraction");
       System.out.println("3. Multiplication");
       System.out.println("4. Division");
       System.out.println("5. Exit");
       System.out.print("Enter your choice (1-5): ");
       choice = scanner.nextInt();

       // Perform the chosen operation
       switch (choice) {
          case 1:
             // Addition
             System.out.print("Enter first number: ");
             int num1 = scanner.nextInt();
             System.out.print("Enter second number: ");
             int num2 = scanner.nextInt();
```

```
    System.out.println("Result: " + Addition(num1, num2));
    break;

case 2:
    // Subtraction
    System.out.print("Enter first number: ");
    num1 = scanner.nextInt();
    System.out.print("Enter second number: ");
    num2 = scanner.nextInt();
    System.out.println("Result: " + Subtraction(num1, num2));
    break;

case 3:
    // Multiplication
    System.out.print("Enter first number: ");
    num1 = scanner.nextInt();
    System.out.print("Enter second number: ");
    num2 = scanner.nextInt();
    System.out.println("Result: " + Multiplication(num1, num2));
    break;

case 4:
    // Division
    System.out.print("Enter first number: ");
    num1 = scanner.nextInt();
    System.out.print("Enter second number: ");
    num2 = scanner.nextInt();
    System.out.println("Result: " + Division(num1, num2));
    break;

case 5:
```

```
                // Exit
                System.out.println("Exiting...");
                break;

            default:
                System.out.println("Invalid choice. Please select a valid option.");
                break;
        }
        System.out.println(); // Print an empty line for better readability
    } while (choice != 5);
    scanner.close();
  }
}
```