Unit-I

Q1) What are the drawbacks of procedural languages? Explain the need of object- oriented programming with suitable program. Also explain the various attributes of object-oriented programming.

Ans: Drawbacks of Procedural Languages

Procedural programming languages like C, Pascal, and Fortran are based on the concept of procedure calls, where programs are organized into functions or procedures. While procedural languages are effective for small to medium-scale applications, they exhibit several drawbacks when dealing with large and complex systems:

1. **Code Reusability**: Procedural languages lack the ability to easily reuse code. Functions can be reused, but this reuse is often limited by the global scope of variables and the tight coupling between data and functions.
2. **Difficulty in Maintenance**: As the program grows, maintaining it becomes challenging because functions share global variables, making it difficult to track where and how data is modified.
3. **Scalability**: Procedural programming is not naturally suited for large projects. When the program gets more extensive, it becomes harder to break the problem down logically into smaller components. Interdependencies between functions can make scaling difficult.
4. **Poor Real-World Modeling**: Procedural languages often struggle to represent real-world objects. In a procedural program, data and behaviors (functions) are separate, which does not align well with how real-world entities behave, where data and behaviors are inherently linked.
5. **Data Security**: Global variables can be accessed and modified by any part of the program, increasing the risk of unintended side effects and data corruption.

Need for Object-Oriented Programming (OOP)

OOP was introduced to address the limitations of procedural languages by modeling programs based on real-world objects. OOP provides a way to encapsulate both data and behavior (methods) within objects, improving the organization, flexibility, and scalability of programs.

OOP offers several advantages:

- **Modularity**: Programs are divided into smaller, independent objects that interact with each other.
- **Code Reusability**: Through inheritance, existing code can be reused and extended without modifying the original code.
- **Data Abstraction**: Objects hide the implementation details and expose only essential functionalities, which simplifies the complexity of a system.
- **Encapsulation**: Data and methods that operate on the data are bundled together. Access to data is restricted to prevent unauthorized access or modifications.
- **Polymorphism**: The ability to define methods in multiple forms. This allows objects of different types to be treated in a uniform way.
- **Inheritance**: A new class can inherit properties and methods from an existing class, promoting code reuse and reducing redundancy.

Attributes of Object-Oriented Programming

1. **Encapsulation**: Encapsulation is the concept of bundling data (fields) and methods (functions) that operate on that data into a single unit or class. It restricts direct access to an object's data and only allows modification through methods, ensuring the integrity of the object.

   Example:

```
class Person {
   private String name; // Encapsulated data

   // Public method to access private data
   public String getName() {
      return name;
   }

   // Public method to modify private data
   public void setName(String name) {
      this.name = name;
   }
```

```
}
```

2. **Abstraction**: Abstraction hides the complex implementation details and only exposes the essential functionalities. In OOP, abstraction is achieved through interfaces and abstract classes.

   Example:
   ```
   abstract class Vehicle {
      abstract void start(); // Abstract method (no implementation)

      public void fuel() {
         System.out.println("Filling fuel...");
      }
   }

   class Car extends Vehicle {
      @Override
      void start() {
         System.out.println("Car is starting...");
      }
   }
   ```

3. **Inheritance**: Inheritance allows one class to inherit properties and methods from another class. It promotes code reuse and allows the creation of a hierarchy between classes.

   Example:
   ```
   class Animal {
      public void makeSound() {
         System.out.println("Some generic animal sound");
      }
   }

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

4. **Polymorphism**: Polymorphism allows a single method to behave differently based on the object it is called on. It can be achieved through method overloading (compile-time polymorphism) or method overriding (runtime polymorphism).

   Example (Method Overriding):
   ```
   class Animal {
      public void sound() {
         System.out.println("Animal makes a sound");
      }
   }

   class Cat extends Animal {
      @Override
      public void sound() {
         System.out.println("Meow");
      }
   }

   class Main {
      public static void main(String[] args) {
         Animal myAnimal = new Cat();
         myAnimal.sound();  // Calls the overridden method in Cat
      }
   }
   ```

Conclusion

Object-Oriented Programming (OOP) addresses the limitations of procedural programming by introducing key concepts like encapsulation, inheritance, and polymorphism, making programs more modular, scalable, and maintainable.

Q2) Explain different types of variable used in JAVA. Explain each with suitable JAVA program also explain the scope of each type of variable.

Ans: In Java, variables can be categorized based on their declaration and usage. There are three primary types of variables:

1. **Instance Variables (Non-Static Fields)**
2. **Class Variables (Static Fields)**
3. **Local Variables**

Each type of variable has its own characteristics, scope, and lifetime. Let's look at each type in detail with Java code examples.

1. Instance Variables (Non-Static Fields)

- **Definition**: Instance variables are non-static fields declared in a class but outside any method, constructor, or block. They are tied to a specific object of the class and are created when an object is instantiated.
- **Scope**: The scope of instance variables is within the object. They can be accessed by all non-static methods of the class.
- **Lifetime**: Instance variables are created when the object is created and destroyed when the object is destroyed.

Example of Instance Variables:

```
class Car {
    // Instance variables
    String model;
    String color;
    int year;

    // Constructor to initialize instance variables
    public Car(String model, String color, int year) {
        this.model = model;
        this.color = color;
        this.year = year;
    }

    // Method to display car details (instance variables are accessible here)
    public void displayCarDetails() {
        System.out.println("Car Model: " + model);
        System.out.println("Car Color: " + color);
        System.out.println("Car Year: " + year);
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating an object of Car (instance variables are initialized for this object)
        Car car1 = new Car("Toyota Camry", "Blue", 2020);
        car1.displayCarDetails();  // Accessing instance variables through an object

        Car car2 = new Car("Tesla Model 3", "Red", 2021);
        car2.displayCarDetails();  // Each object has its own copy of instance variables
    }
}
```

Output:

```
Car Model: Toyota Camry
Car Color: Blue
Car Year: 2020
Car Model: Tesla Model 3
Car Color: Red
```

Car Year: 2021

2. Class Variables (Static Fields)

- **Definition**: Class variables are fields declared with the static keyword. These variables are shared across all objects of the class. They belong to the class itself rather than to any specific object.
- **Scope**: The scope of a class variable is the entire class. It can be accessed by both static and non-static methods.
- **Lifetime**: Class variables are created when the class is loaded and destroyed when the class is unloaded (typically when the program ends).

Example of Class Variables:

```
class BankAccount {
    // Static variable
    static double interestRate = 3.5;

    // Instance variable
    String accountHolder;
    double balance;

    // Constructor to initialize instance variables
    public BankAccount(String accountHolder, double balance) {
        this.accountHolder = accountHolder;
        this.balance = balance;
    }

    // Static method to change the interest rate (access static variable)
    public static void setInterestRate(double newRate) {
        interestRate = newRate;
    }

    // Method to display account details (both instance and static variables)
    public void displayAccountDetails() {
        System.out.println("Account Holder: " + accountHolder);
        System.out.println("Balance: " + balance);
        System.out.println("Interest Rate: " + interestRate);
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating objects of BankAccount
        BankAccount account1 = new BankAccount("John", 5000);
        BankAccount account2 = new BankAccount("Alice", 7000);

        // Display account details before changing interest rate
        account1.displayAccountDetails();
        account2.displayAccountDetails();

        // Change interest rate using class method (static method)
        BankAccount.setInterestRate(4.0);

        // Display account details after changing interest rate
        account1.displayAccountDetails();
        account2.displayAccountDetails();
    }
}
```

Output:

```
Account Holder: John
Balance: 5000.0
Interest Rate: 3.5
Account Holder: Alice
Balance: 7000.0
Interest Rate: 3.5
```

Account Holder: John
Balance: 5000.0
Interest Rate: 4.0
Account Holder: Alice
Balance: 7000.0
Interest Rate: 4.0

Explanation:

- The interestRate is a static variable, so it is shared among all objects. Changing it through one object or class method affects all instances.

3. Local Variables

- **Definition**: Local variables are declared inside methods, constructors, or blocks and are used for temporary storage of data within those methods or blocks.
- **Scope**: The scope of a local variable is limited to the block, method, or constructor in which it is declared.
- **Lifetime**: Local variables are created when the block or method is entered and destroyed once the block or method is exited.

Example of Local Variables:

```
class Calculator {
    // Method with local variables
    public void addNumbers() {
        // Local variables
        int a = 10;
        int b = 20;
        int sum = a + b;  // sum is a local variable
        System.out.println("Sum: " + sum);
    }

    // Another method with different local variables
    public void multiplyNumbers() {
        int x = 5;
        int y = 4;
        int product = x * y;  // product is a local variable
        System.out.println("Product: " + product);
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating an object of Calculator
        Calculator calc = new Calculator();

        // Calling methods (local variables are created and destroyed in these methods)
        calc.addNumbers();
        calc.multiplyNumbers();
    }
}
```

Output:

Sum: 30
Product: 20

Explanation:

- Variables a, b, sum, x, y, and product are local to their respective methods. They are created when the method is called and are destroyed when the method finishes.

Q3) What are the primitive data types in JAVA? Mention each data type in detail with proper syntax and example.

Ans: Java has **8 primitive data types**, which are predefined by the language and serve as the most basic building blocks of data manipulation. These data types are grouped into four categories:

1. **Integer Types**: byte, short, int, long
2. **Floating-Point Types**: float, double
3. **Character Type**: char
4. **Boolean Type**: boolean

Each primitive type is associated with a fixed memory size and a specific range of values it can store. Let's go through each data type in detail:

1. byte (Integer Type)

- **Size**: 8 bits (1 byte)
- **Range**: -128 to 127
- **Use**: Typically used when memory savings are important or for storing small integer values.

Syntax:

byte variableName = value;

Example:

```
public class Main {
   public static void main(String[] args) {
      byte age = 25;
      System.out.println("Age: " + age);  // Output: Age: 25
   }
}
```

2. short (Integer Type)

- **Size**: 16 bits (2 bytes)
- **Range**: -32,768 to 32,767
- **Use**: Useful when memory is a concern, and the int type is more than needed for storing small numbers.

Syntax:

short variableName = value;

Example:

```
public class Main {
   public static void main(String[] args) {
      short year = 2024;
      System.out.println("Year: " + year);  // Output: Year: 2024
   }
}
```

3. int (Integer Type)

- **Size**: 32 bits (4 bytes)
- **Range**: -2^31 to 2^31 - 1 (-2,147,483,648 to 2,147,483,647)
- **Use**: Most commonly used for integer arithmetic. Suitable for most general-purpose calculations.

Syntax:

int variableName = value;

Example:

```
public class Main {
   public static void main(String[] args) {
```

```
        int population = 500000;
        System.out.println("Population: " + population);  // Output: Population: 500000
    }
}
```

4. long (Integer Type)

- **Size**: 64 bits (8 bytes)
- **Range**: -2^63 to 2^63 - 1 (approx. -9.2 quintillion to 9.2 quintillion)
- **Use**: Used when a wider range than int is required, for example, in high-precision or very large calculations.

Syntax:

```
long variableName = value;
```

Example:

```
public class Main {
    public static void main(String[] args) {
        long distanceToMoon = 384400000L;  // The 'L' is necessary for long literals
        System.out.println("Distance to Moon (meters): " + distanceToMoon);
    }
}
```

5. float (Floating-Point Type)

- **Size**: 32 bits (4 bytes)
- **Range**: Approximately ±3.40282347E+38F (7 decimal digits)
- **Use**: Used for single-precision floating-point arithmetic (fractional numbers).
- **Important**: Append F or f at the end of the value for float literals.

Syntax:

```
float variableName = value;
```

Example:

```
public class Main {
    public static void main(String[] args) {
        float pi = 3.14F;
        System.out.println("Value of Pi: " + pi);  // Output: Value of Pi: 3.14
    }
}
```

6. double (Floating-Point Type)

- **Size**: 64 bits (8 bytes)
- **Range**: Approximately ±1.79769313486231570E+308 (15 decimal digits)
- **Use**: Used for double-precision floating-point arithmetic (more precise than float).
- **Default**: By default, floating-point numbers are treated as double unless explicitly cast or suffixed with F.

Syntax:

```
double variableName = value;
```

Example:

```
public class Main {
    public static void main(String[] args) {
        double largeDecimal = 1234567.1234567;
        System.out.println("Large Decimal Number: " + largeDecimal);
    }
}
```

7. char (Character Type)

- **Size**: 16 bits (2 bytes)
- **Range**: 0 to 65,535 (Unicode characters)
- **Use**: Used to store a single character or a Unicode value.
- **Note**: Character literals are enclosed in single quotes (' ').

Syntax:

char variableName = 'character';

Example:

```
public class Main {
   public static void main(String[] args) {
      char letter = 'A';
      System.out.println("Letter: " + letter);  // Output: Letter: A
   }
}
```

8. boolean (Boolean Type)

- **Size**: 1 bit (but memory used depends on JVM implementation)
- **Range**: true or false
- **Use**: Used to store only two possible values: true or false. It is primarily used in conditions and control structures.

Syntax:

boolean variableName = trueOrFalse;

Example:

```
public class Main {
   public static void main(String[] args) {
      boolean isJavaFun = true;
      System.out.println("Is Java Fun? " + isJavaFun);  // Output: Is Java Fun? true
   }
}
```

Conclusion

Java's primitive data types provide a way to store simple values like numbers, characters, and logical conditions. They have fixed sizes and ranges, making them efficient for memory and performance, and are essential for all computations in Java.

Q4) Explain the types of operators in JAVA.

Ans: Here's a detailed explanation of the types of operators in Java along with examples for each:

1. Arithmetic Operators

Used for performing mathematical calculations.

Example:

```
int a = 10;
int b = 5;
System.out.println(a + b);  // Output: 15
System.out.println(a - b);  // Output: 5
System.out.println(a * b);  // Output: 50
System.out.println(a / b);  // Output: 2
System.out.println(a % b);  // Output: 0
```

2. Unary Operators

Operate on a single operand.

Example:

```
int x = 10;
System.out.println(++x);  // Output: 11 (pre-increment)
System.out.println(x--);  // Output: 11 (post-decrement, but prints before decrement)
System.out.println(x);    // Output: 10
System.out.println(-x);   // Output: -10
System.out.println(!true);// Output: false
```

3. Relational (Comparison) Operators

Used for comparing values.

Example:

```
int a = 5;
int b = 3;
System.out.println(a == b);  // Output: false
System.out.println(a != b);  // Output: true
System.out.println(a > b);   // Output: true
System.out.println(a <= b);  // Output: false
```

4. Logical Operators

Used for combining multiple conditions.

Example:

```
boolean a = true;
boolean b = false;
System.out.println(a && b);  // Output: false
System.out.println(a || b);  // Output: true
System.out.println(!a);      // Output: false
```

5. Bitwise Operators

Used to perform bit-level operations.

Example:

```
int a = 5;  // Binary: 0101
int b = 3;  // Binary: 0011
System.out.println(a & b);  // Output: 1 (Binary: 0001)
System.out.println(a | b);  // Output: 7 (Binary: 0111)
System.out.println(a ^ b);  // Output: 6 (Binary: 0110)
System.out.println(~a);     // Output: -6 (Binary: ...11111010)
System.out.println(a << 1); // Output: 10 (Binary: 1010)
System.out.println(a >> 1); // Output: 2 (Binary: 0010)
```

6. Assignment Operators

Used to assign values to variables.

Example:

```
int a = 5;
a += 3;  // a = a + 3, so a = 8
System.out.println(a);  // Output: 8
a *= 2;  // a = a * 2, so a = 16
System.out.println(a);  // Output: 16
```

7. Ternary Operator

A shorthand for if-else conditions.

Example:

```
int a = 10, b = 20;
int max = (a > b) ? a : b;  // Checks if a is greater than b
System.out.println(max);    // Output: 20
```

Q5) What is JAVA? Explain its features and also explain how JVM works.

Ans: Java is a widely used, high-level programming language known for its simplicity, portability, and robust feature set. Developed by Sun Microsystems (now owned by Oracle Corporation) and officially released in 1995, Java is designed to be platform-independent, meaning code written in Java can run on any device that has a Java Virtual Machine (JVM).

Basic Features of Java

1. **Platform Independence**
   - **Write Once, Run Anywhere (WORA)**: Java programs are compiled into bytecode, which is platform-independent. This bytecode is executed by the JVM, allowing Java applications to run on any platform that has a compatible JVM.

   Example: A Java program compiled on a Windows machine can run on a Linux or macOS system without modification.

2. **Object-Oriented**
   - **Encapsulation**: Java encourages bundling data (fields) and methods (functions) that operate on the data into a single unit called a class. Access to the data is restricted via access modifiers.
   - **Inheritance**: Java supports inheritance, allowing a new class to inherit the properties and methods of an existing class, facilitating code reusability.
   - **Polymorphism**: Java supports polymorphism, where one interface can be used to represent different underlying forms (data types). This allows for method overriding and method overloading.
   - **Abstraction**: Java allows you to define abstract classes and interfaces to provide a blueprint for other classes, focusing on essential qualities without needing to specify details.
3. **Simple and Easy to Learn**
   - Java was designed to be easy to learn and use, with a syntax that is clean and familiar to those who have used C or C++. It removes complex features like pointers and multiple inheritance, simplifying programming.
4. **Secure**
   - Java includes several features designed to ensure security:
     - **Bytecode Verification**: The JVM checks bytecode for illegal code that could violate access rights.
     - **Security Manager**: Java applications can be restricted by a security manager, which enforces security policies.
     - **Sandboxing**: Java applets (though less common now) can be executed in a restricted environment called a sandbox, limiting their ability to perform potentially harmful operations.
5. **Robust**
   - **Exception Handling**: Java provides a powerful mechanism for handling runtime errors via exceptions. This makes it easier to detect and manage errors in a controlled manner.
   - **Garbage Collection**: Java manages memory automatically with garbage collection, which removes objects that are no longer in use, helping to prevent memory leaks and optimize memory usage.
6. **Multithreaded**
   - Java has built-in support for multithreading, allowing multiple threads to run concurrently within a single program. This makes it easier to develop applications that perform multiple tasks simultaneously.
7. **Distributed**
   - Java provides extensive support for networking and distributed computing, making it easier to build applications that communicate over a network. It includes libraries for handling network communications, remote method invocation (RMI), and distributed objects.
8. **High Performance**
   - Java programs are generally faster than interpreted languages due to the JVM's Just-In-Time (JIT) compiler, which compiles bytecode to native machine code at runtime for better performance.
9. **Dynamic**
   - Java is dynamic in nature, meaning it can adapt to changing requirements and conditions. It supports dynamic class loading, which allows classes to be loaded at runtime, and dynamic method invocation.

10. **Rich Standard Library**
    - Java provides a comprehensive standard library, also known as the Java API (Application Programming Interface), which includes packages for performing various tasks such as input/output (I/O), data manipulation, networking, and graphical user interface (GUI) development.

JVM

Java Virtual Machine (JVM) is an abstract computing machine that enables a computer to run Java programs as well as programs written in other languages that are compiled to Java bytecode. The JVM provides a runtime environment in which Java bytecode can be executed, allowing Java applications to be platform-independent.

How JVM Works

1. **Compilation**:
    - Java source code (.java files) is compiled by the Java Compiler (javac) into bytecode (.class files). This bytecode is platform-independent and can be executed on any machine that has a compatible JVM.
2. **Loading**:
    - The JVM loads the compiled bytecode into memory. This is done by the **Class Loader**, which is a part of the JVM.
3. **Bytecode Verification**:
    - The bytecode verifier checks the loaded bytecode to ensure it is valid and adheres to Java's security rules. This prevents code that violates Java's safety principles from being executed.
4. **Execution**:
    - The JVM interprets or compiles the bytecode into machine code that can be executed by the underlying hardware. This is done using the **Java Interpreter** or the **Just-In-Time (JIT) Compiler**.
        - The **Interpreter** translates bytecode into machine code instruction by instruction.
        - The **JIT Compiler** compiles bytecode into native machine code at runtime, optimizing performance by reducing the number of times the code needs to be interpreted.
5. **Runtime Environment**:
    - The JVM manages system resources, such as memory management, execution of threads, and garbage collection (automatic memory management).

Summary

- The **JVM** is an essential part of the Java programming language, enabling it to run on any platform that has a compatible JVM, achieving platform independence.
- It works through a series of steps: loading, verification, and execution of bytecode.
- The **JVM architecture** comprises several components, including the class loader subsystem, runtime data areas, execution engine, and native interface, all of which work together to execute Java applications efficiently.

Q6) Explain:

a) Implicit Type Casting **b) Explicit Type Casting**

Ans:

a) Implicit Type Casting

Implicit type casting in Java, also known as automatic type conversion or widening conversion, happens when the Java compiler automatically converts a smaller data type to a larger data type. This occurs without any explicit code from the programmer because there is no risk of data loss.

Example of Implicit Type Casting:

```
public class ImplicitTypeCastingExample {
   public static void main(String[] args) {
      // Smaller to larger type conversions (automatic casting)
      int intVar = 100;
      long longVar = intVar;  // int to long
      float floatVar = longVar;  // long to float
      double doubleVar = floatVar;  // float to double

      System.out.println("int value: " + intVar);       // Output: 100
      System.out.println("long value: " + longVar);     // Output: 100
      System.out.println("float value: " + floatVar);   // Output: 100.0
      System.out.println("double value: " + doubleVar); // Output: 100.0
```

```
    }
}
```

b) Explicit Type Casting

Explicit type casting in Java, also known as manual type conversion or narrowing conversion, is when the programmer manually converts one data type into another, typically from a larger data type to a smaller data type. This is necessary because narrowing conversions can lead to data loss or loss of precision, so Java requires explicit instructions from the programmer to proceed.

Example of Explicit Type Casting:

```
public class ExplicitTypeCastingExample {
    public static void main(String[] args) {
        double doubleVar = 9.78;
        int intVar = (int) doubleVar;  // Explicit casting from double to int

        System.out.println("Double value: " + doubleVar);  // Output: 9.78
        System.out.println("Int value (after casting): " + intVar);  // Output: 9
    }
}
```

Unit-II

Q1) Explain class and object with suitable example. How classes and objects are represented in object modelling techniques? Explain with suitable diagram and example in detail.

Ans: Class and Object in Java

In Java (and object-oriented programming in general), a **class** is a blueprint for creating objects, while an **object** is an instance of a class. Classes define the properties (data fields) and behaviors (methods) that objects created from that class will have. Objects are individual instances that hold their own data but share the structure and behavior defined by the class.

Class

A class is a template or a blueprint that defines the structure and behavior (attributes and methods) common to all objects of a particular type. It encapsulates the data for the object and the methods that operate on that data.

Example of a class:

```
class Car {
    // Attributes (properties)
    String model;
    String color;
    int year;

    // Constructor to initialize the Car object
    public Car(String model, String color, int year) {
        this.model = model;
        this.color = color;
        this.year = year;
    }

    // Method to display car details
    public void displayCarDetails() {
        System.out.println("Car Model: " + model);
        System.out.println("Car Color: " + color);
        System.out.println("Car Year: " + year);
    }
}
```

Object

An object is an instance of a class. When you create an object, you allocate memory for the object and initialize its attributes according to the constructor's specification.

Example of an object:

```
public class Main {
    public static void main(String[] args) {
        // Creating an object of the Car class
        Car car1 = new Car("Toyota Camry", "Blue", 2020);

        // Calling method on the object
        car1.displayCarDetails();
    }}
```

Output:

```
Car Model: Toyota Camry
Car Color: Blue
Car Year: 2020
```

Representation of Classes and Objects in Object Modelling Techniques

In **Object-Oriented Design** and **Object Modelling Techniques** (OMT), classes and objects are represented using **UML (Unified Modeling Language)** diagrams. The primary UML diagram used to represent classes and their relationships is called a **Class Diagram**.

1. ***Class Diagram***

A class diagram is a visual representation of the classes in a system and the relationships between them. It displays the class name, attributes, methods, and associations between classes.

A **class diagram** consists of:

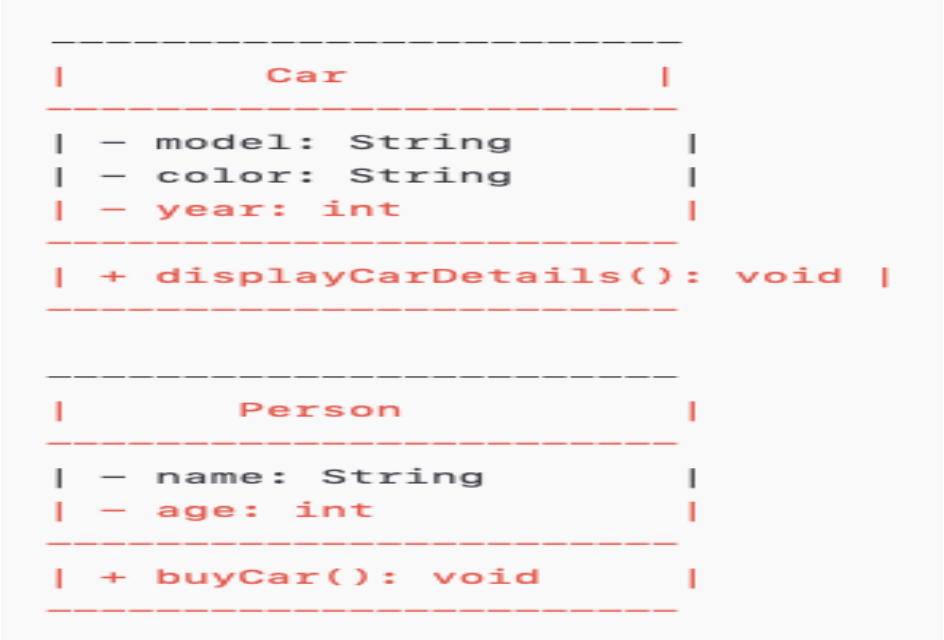
- **Class Name**: Represents the name of the class.
- **Attributes**: Data fields or properties of the class.
- **Methods (Operations)**: Functions or behaviors that the class can perform.

Each class is represented as a rectangle, which is divided into three sections: the top section for the class name, the middle section for the attributes, and the bottom section for methods.

Example UML Class Diagram:

Let's model a **Car** class and a **Person** class in a class diagram.

- **Car Class**:
  - Attributes: model, color, year
  - Methods: displayCarDetails()
- **Person Class**:
  - Attributes: name, age
  - Methods: buyCar()



In this diagram:

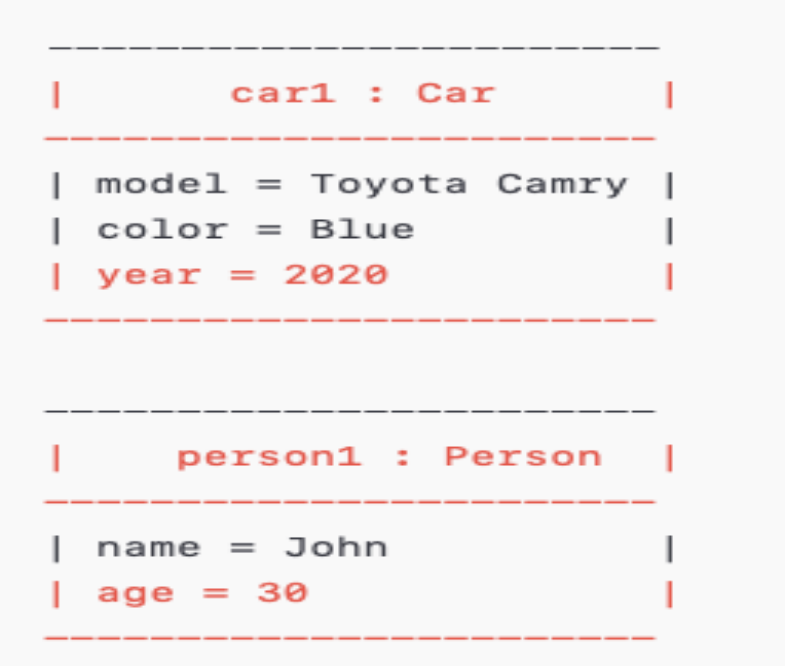
- Car is a class with private attributes (indicated by - before the attributes) and a public method displayCarDetails(indicated by + before the method).
- Person is another class that has private attributes (name, age) and a public method buyCar.

2. Object Diagram

An object diagram is a snapshot of the objects in the system at a particular point in time. It shows instances of the classes (objects) and their relationships. An object diagram is similar to a class diagram, but instead of showing the class structure, it shows the actual objects and their attribute values.

For example, if we have an object car1 of the class Car, and an object person1 of the class Person, an object diagram could be represented like this:

Object Diagram:



Here, car1 is an instance of the Car class, and person1 is an instance of the Person class. The diagram shows their current values for the attributes.

Example: Object-Oriented Modelling in Java

Let's model a simple real-world relationship between **Person** and **Car** using Java.

```
// Car class
class Car {
    String model;
    String color;
    int year;

    // Constructor
    public Car(String model, String color, int year) {
        this.model = model;
        this.color = color;
        this.year = year;
    }

    // Method to display car details
    public void displayCarDetails() {
        System.out.println("Car Model: " + model + ", Color: " + color + ", Year: " + year);
    }
}

// Person class
class Person {
    String name;
    int age;
    Car car;  // Association: A person can have a car

    // Constructor
    public Person(String name, int age, Car car) {
        this.name = name;
        this.age = age;
        this.car = car;  // Linking the car to the person
    }

    // Method to display person details
    public void displayPersonDetails() {
```

```
        System.out.println("Person Name: " + name + ", Age: " + age);
        System.out.print("Car Details: ");
        car.displayCarDetails();
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating a Car object
        Car myCar = new Car("Tesla Model S", "Red", 2022);

        // Creating a Person object with a car
        Person person = new Person("Alice", 28, myCar);

        // Displaying person and car details
        person.displayPersonDetails();
    }
}
```

Conclusion

- **Class** in Java is a blueprint that defines properties and methods.
- **Object** is an instance of a class, representing an actual entity in memory with its own data.
- Classes and objects are represented in **UML diagrams** in object-oriented modeling techniques. Class diagrams show class structure and relationships, while object diagrams capture instances of those classes at a specific point in time.

Q2) What is a constructor? What is its requirement in programming? Explain with program. What are th benefits of constructor overloading? Explain with suitable example.

Ans: A **constructor** in Java is a special method that is automatically called when an object of a class is created. It is used to initialize the object and set initial values for the object's fields.

Key Features of Constructors:

- **Same Name as the Class**: The name of the constructor must be the same as the class name.
- **No Return Type**: Constructors do not have a return type, not even void.
- **Called Automatically**: A constructor is called automatically when an object is instantiated.
- **Types of Constructors**:
    - **Default Constructor**: A constructor that takes no arguments.
    - **Parameterized Constructor**: A constructor that takes arguments to initialize fields with specific values.

Why are Constructors Required in Programming?

Constructors are essential because they allow you to initialize objects when they are created. Instead of manually setting field values after object creation, constructors ensure that objects are properly initialized with values right from the start.

Example: Basic Constructor in Java

```
class Car {
    String model;
    int year;

    // Default constructor
    Car() {
        model = "Unknown";
        year = 0;
    }

    // Parameterized constructor
    Car(String m, int y) {
        model = m;
```

```
        year = y;
    }

    void display() {
        System.out.println("Model: " + model + ", Year: " + year);
    }
}

public class ConstructorExample {
    public static void main(String[] args) {
        // Using default constructor
        Car car1 = new Car();
        car1.display();  // Output: Model: Unknown, Year: 0

        // Using parameterized constructor
        Car car2 = new Car("Tesla", 2023);
        car2.display();  // Output: Model: Tesla, Year: 2023
    }
}
```

Constructor Overloading in Java

Constructor overloading is a feature in Java that allows a class to have multiple constructors with different parameter lists. Each constructor performs different tasks depending on the number and type of parameters.

Benefits of Constructor Overloading:

1. **Flexibility in Object Initialization**: It provides flexibility to create objects with different initial values.
2. **Code Reusability**: Overloaded constructors allow code reuse by enabling different ways to initialize objects with minimal code changes.
3. **Custom Initialization**: It allows custom initialization for different object creation scenarios.

Example: Constructor Overloading in Java

```
class Employee {
    String name;
    int age;
    String department;

    // Default constructor
    Employee() {
        this.name = "Not Specified";
        this.age = 0;
        this.department = "Unknown";
    }

    // Constructor with one parameter
    Employee(String name) {
        this.name = name;
        this.age = 0;
        this.department = "Unknown";
    }

    // Constructor with two parameters
    Employee(String name, int age) {
        this.name = name;
        this.age = age;
        this.department = "Unknown";
    }

    // Constructor with three parameters
    Employee(String name, int age, String department) {
        this.name = name;
        this.age = age;
        this.department = department;
```

```
    }

    void display() {
        System.out.println("Name: " + name + ", Age: " + age + ", Department: " + department);
    }
}

public class ConstructorOverloadingExample {
    public static void main(String[] args) {
        // Using default constructor
        Employee emp1 = new Employee();
        emp1.display();  // Output: Name: Not Specified, Age: 0, Department: Unknown

        // Using constructor with one parameter
        Employee emp2 = new Employee("John");
        emp2.display();  // Output: Name: John, Age: 0, Department: Unknown

        // Using constructor with two parameters
        Employee emp3 = new Employee("Alice", 28);
        emp3.display();  // Output: Name: Alice, Age: 28, Department: Unknown

        // Using constructor with three parameters
        Employee emp4 = new Employee("Bob", 35, "HR");
        emp4.display();  // Output: Name: Bob, Age: 35, Department: HR
    }
}
```

Q3) What do you mean by the term inheritance in Java? What are the benefits of inheritance? Explain the various forms of inheritance supported by JAVA with suitable code segments.

Ans: Inheritance in Java: Inheritance **in Java is a mechanism where one class acquires the properties (fields) and behaviors (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). This is a key feature of** Object-Oriented Programming (OOP) **and promotes code reuse.**

Key Terms:

- **Superclass**: The class whose properties and methods are inherited (also known as the parent class).
- **Subclass**: The class that inherits from the superclass (also known as the child class).
- **extends keyword**: Used to indicate that a class inherits from another class.

Benefits of Inheritance:

1. **Code Reusability**: Inheritance allows you to reuse the code of the existing class, reducing redundancy.
2. **Method Overriding**: It allows you to provide a specific implementation of a method that is already defined in its parent class.
3. **Extensibility**: You can enhance the behavior of an existing class by adding new features to the subclass.
4. **Maintenance**: By grouping shared attributes and behaviors in a superclass, changes to shared logic can be made in one place, making the system easier to maintain.

Types of Inheritance Supported in Java:

Java supports several types of inheritance:

1. **Single Inheritance**
2. **Multilevel Inheritance**
3. **Hierarchical Inheritance**

Java **does not support multiple inheritance** directly (i.e., a class cannot inherit from more than one class) to avoid ambiguity. However, **multiple inheritance** can be achieved through interfaces.

1. Single Inheritance

In **single inheritance**, a subclass inherits from one superclass.

Example:

```
class Animal {
   void eat() {
      System.out.println("This animal is eating.");
   }
}

class Dog extends Animal {
   void bark() {
      System.out.println("The dog is barking.");
   }
}

public class SingleInheritance {
   public static void main(String[] args) {
      Dog dog = new Dog();
      dog.eat();  // Inherited method
      dog.bark(); // Method from Dog class
   }
}
```

Output:

```
This animal is eating.
The dog is barking.
```

2. Multilevel Inheritance

In **multilevel inheritance**, a class is derived from a class that is already derived from another class, forming a chain of inheritance.

Example:

```
class Animal {
   void eat() {
      System.out.println("This animal is eating.");
   }
}

class Dog extends Animal {
   void bark() {
      System.out.println("The dog is barking.");
   }
}

class Puppy extends Dog {
   void weep() {
      System.out.println("The puppy is weeping.");
   }
}

public class MultilevelInheritance {
   public static void main(String[] args) {
      Puppy puppy = new Puppy();
      puppy.eat();  // Inherited from Animal
      puppy.bark(); // Inherited from Dog
      puppy.weep(); // Method from Puppy class
   }
}
```

Output:

```
This animal is eating.
The dog is barking.
The puppy is weeping.
```

3. Hierarchical Inheritance

In **hierarchical inheritance**, multiple subclasses inherit from the same superclass.

Example:

```
class Animal {
   void eat() {
      System.out.println("This animal is eating.");
   }
}

class Dog extends Animal {
   void bark() {
      System.out.println("The dog is barking.");
   }
}

class Cat extends Animal {
   void meow() {
      System.out.println("The cat is meowing.");
   }
}

public class HierarchicalInheritance {
   public static void main(String[] args) {
      Dog dog = new Dog();
      Cat cat = new Cat();

      dog.eat();  // Inherited from Animal
      dog.bark(); // Method from Dog class

      cat.eat();  // Inherited from Animal
      cat.meow(); // Method from Cat class
   }
}
```

Output:

```
This animal is eating.
The dog is barking.
This animal is eating.
The cat is meowing.
```

4. Multiple Inheritance via Interfaces

Java does not support multiple inheritance of classes directly (i.e., a class cannot inherit from more than one class). However, you can achieve multiple inheritance using **interfaces**.

Example:

```
interface CanFly {
   void fly();
}

interface CanSwim {
   void swim();
}

class Duck implements CanFly, CanSwim {
   public void fly() {
      System.out.println("The duck is flying.");
   }

   public void swim() {
      System.out.println("The duck is swimming.");
   }
}
```

```
public class MultipleInheritanceWithInterfaces {
    public static void main(String[] args) {
        Duck duck = new Duck();
        duck.fly();  // Method from CanFly interface
        duck.swim(); // Method from CanSwim interface
    }
}
```

Output:

```
The duck is flying.
The duck is swimming.
```

Benefits of Inheritance:

1. **Code Reusability**: Reduces redundancy by reusing common fields and methods.
2. **Extensibility**: You can add new functionality to existing code easily by extending classes.
3. **Polymorphism**: Inheritance allows subclasses to have their own specific implementation of methods (via method overriding), enabling polymorphism.
4. **Simplified Code**: By inheriting common features, subclasses can focus on unique behaviors, simplifying the codebase.

Q4) Explain:

a) Use of this keyword **b) Use of super keyword**

a) Use of this keyword

In Java, the this keyword is used to refer to the current object within an instance method or constructor. It helps differentiate between instance variables and parameters or local variables when they have the same name. Here are some common uses of the this keyword in Java:

1. To refer to instance variables:

The this keyword can be used to distinguish between instance variables and method parameters that have the same name.
e.g.

```
class Person {
    String name;

    Person(String name) {
        this.name = name; // Refers to the instance variable 'name'
    }
}
```

2. **To invoke the current class constructor**

You can use this() to call another constructor within the same class. This is useful for constructor chaining.

e.g.

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

    Person(String name) {
        this(name, 0); // Calls the constructor with two parameters
    }

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

}
```

3. **To pass the current object as an argument**

The this keyword can be used to pass the current instance of the class to a method or constructor.
e.g.

```
class Person {
    void display(Person p) {
```

```
        System.out.println("Name: " + p.name);
    }

    void show() {
        display(this); // Passes the current object as an argument
    }
}
```

4. **To invoke the current class method**

The this keyword can be used to call a method of the current class.

e.g.

```
class Person {
    void display() {
        System.out.println("Hello");
    }

    void show() {
        this.display(); // Calls the display() method of this class
    }
}
```

b) Use of super keyword

In Java, the **super** keyword is a reference variable used to refer to the **immediate parent class** (or superclass) of the current object. The super keyword allows a subclass to access methods, constructors, and fields from its superclass, which is useful when subclass methods or constructors override or hide members of the superclass.

Uses of super Keyword:

1. **To call the superclass constructor**.
2. **To access superclass methods** (when overridden in the subclass).

1. super to Call Superclass Constructor

When a subclass constructor needs to call its parent class constructor, the super() statement is used. This is important when the superclass constructor requires parameters.

Example:

```
class Animal {
    String name;

    // Superclass constructor
    Animal(String name) {
        this.name = name;
        System.out.println("Animal constructor called");
    }
}

class Dog extends Animal {
    String breed;

    // Subclass constructor
    Dog(String name, String breed) {
        super(name);  // Calls the superclass constructor
        this.breed = breed;
        System.out.println("Dog constructor called");
    }

    void display() {
        System.out.println("Name: " + name + ", Breed: " + breed);
    }
}

public class SuperConstructorExample {
    public static void main(String[] args) {
        Dog dog = new Dog("Buddy", "Golden Retriever");
```

```
        dog.display();
    }
}
```

Output:

```
Animal constructor called
Dog constructor called
Name: Buddy, Breed: Golden Retriever
```

3. super to Call Superclass Methods

When a subclass overrides a method from its superclass, you can use super to call the overridden method from the parent class.

Example:

```
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    // Overriding the sound() method
    void sound() {
        System.out.println("Dog barks");
    }

    void callSuperSound() {
        super.sound();  // Calls the superclass method
    }
}

public class SuperMethodExample {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.sound();         // Calls overridden method (Dog barks)
        dog.callSuperSound(); // Calls superclass method (Animal makes a sound)
    }
}
```

Output:

```
Dog barks
Animal makes a sound
```