Object-Oriented Programming (OOP) in Java is a programming paradigm that organizes software design around objects rather than functions or logic. The four main principles of OOP are **Encapsulation**, **Inheritance**, **Polymorphism**, and **Abstraction**. Here's a breakdown of each concept with its definition and real-world examples:

---

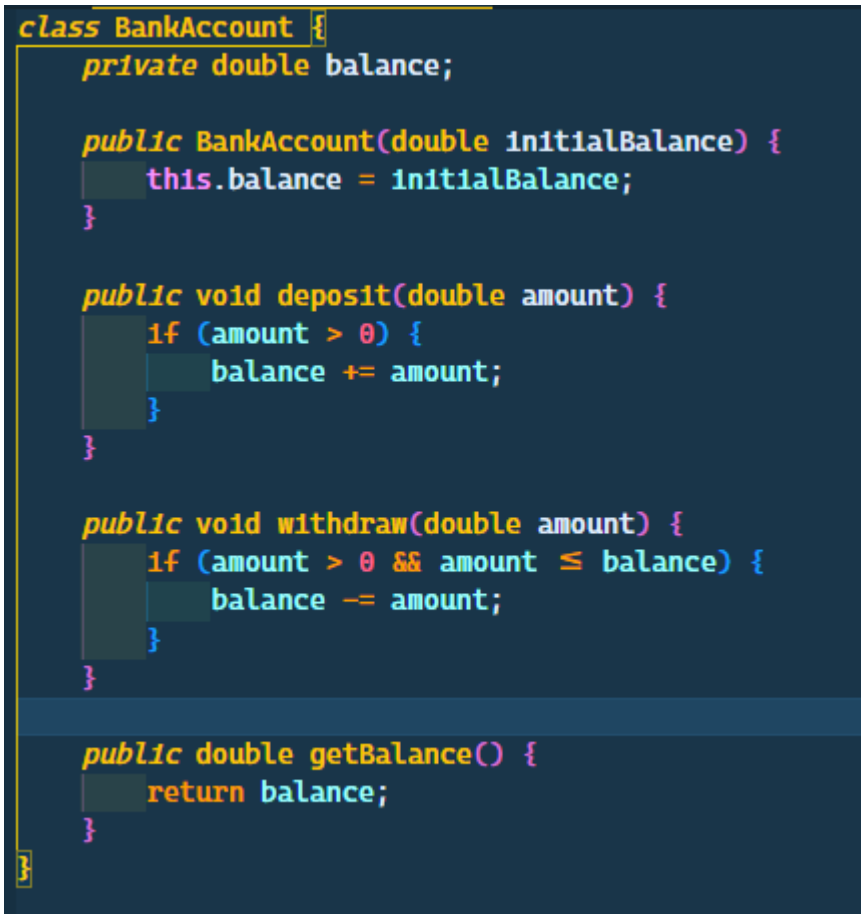
1. Encapsulation

Definition: Encapsulation is the process of bundling data (variables) and methods (functions) that operate on the data into a single unit, or class. It also involves restricting direct access to some of the object's components to ensure controlled access through getters and setters.

Real-World Example: A **bank account**:

- **Data**: Account number, account balance.
- **Methods**: Deposit, withdraw, getBalance.
- Direct access to the balance is restricted, but you can use methods like getBalance() to check it or withdraw() to deduct an amount.

Java Example:



2. Inheritance

Definition: Inheritance allows a class (child class) to inherit properties and behavior from another class (parent class). This promotes code reusability and hierarchical relationships.

Real-World Example: A **vehicle**:

- A **Car** is a type of **Vehicle** and inherits properties like speed and methods like start or stop.
- Specialized behaviors, such as air conditioning or sunroof, are added in the child class.

Java Example:

```java
class Vehicle {
    void start() {
        System.out.println("Vehicle is starting.");
    }
}

class Car extends Vehicle {
    void playMusic() {
        System.out.println("Playing music.");
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Car car = new Car();
        car.start();  // Inherited from Vehicle
        car.playMusic(); // Defined in Car
    }
}
```

3. Polymorphism

Definition: Polymorphism allows a single interface to represent different underlying data types. It enables one action to behave differently based on the object performing it.

- **Compile-time Polymorphism**: Achieved via method overloading.
- **Runtime Polymorphism**: Achieved via method overriding.

Real-World Example: A **printer**:

- A printer can print different types of documents, such as PDFs or images. The action (printing) is the same, but the implementation differs based on the document type.

Java Example: **Method Overloading (Compile-time)**:

```java
class Printer {
    void print(String text) {
        System.out.println("Printing text: " + text);
    }

    void print(int pages) {
        System.out.println("Printing " + pages + " pages.");
    }
}
```

Method Overriding (Runtime):

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

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

public class Main {
    public static void main(String[] args) {
        Animal animal = new Dog(); // Upcasting
        animal.sound(); // Dog's sound() is called
    }
}
```

4. Abstraction

Definition: Abstraction focuses on exposing only the necessary details and hiding implementation complexity. This is achieved using abstract classes or interfaces.

Real-World Example: A **TV remote**:

- The remote lets you increase volume or change channels without showing the internal circuitry.

Java Example: **Using an Abstract Class**:

```java
abstract class Shape {
    abstract void draw();
}

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

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

Using an Interface:

```java
interface Animal {
    void eat();
}

class Cat implements Animal {
    @Override
    public void eat() {
        System.out.println("Cat eats fish.");
    }
}
```

Summary of Real-World Scenarios

OOP Concept	Real-World Example
Encapsulation	Bank account (private balance)
Inheritance	Vehicle and Car hierarchy
Polymorphism	Printer printing PDFs vs images
Abstraction	TV remote buttons

Why Java doesn't support multiple inheritance?

Java does not support multiple inheritance through classes to avoid **ambiguity problems** that arise from the **Diamond Problem**.
The Diamond Problem occurs when a class inherits from two classes that have the same method. If both parent classes define a method with the same name, it becomes unclear which one should be used by the child class.

Example:

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

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

// class C extends A, B { // This would create ambiguity
// }
```

Instead, Java uses **interfaces** to achieve multiple inheritance. Interfaces provide method declarations without implementation, which eliminates ambiguity since there is no concrete implementation to inherit.

What are Constructors?

A **constructor** is a special method in Java used to initialize objects. It is called automatically when an object of the class is created. It shares the same name as the class and does not have a return type (not even void).

Characteristics:

- Used for initialization.
- Invoked implicitly during object creation.
- Can be overloaded to provide multiple ways of initializing an object.

Types of Constructors

1. **Default Constructor**:
    - A no-argument constructor automatically provided by Java if no constructors are explicitly defined.
    - Initializes object with default values.

Example:

```java
class Example {
    int value;

    public static void main(String[] args) {
        Example obj = new Example(); // Default constructor called
        System.out.println(obj.value); // 0 (default int value)
    }
}
```

Parameterized Constructor:

- A constructor that accepts parameters to initialize an object with specific values.

```java
class Example {
    int value;

    Example(int value) {
        this.value = value; // Initialize using parameter
    }

    public static void main(String[] args) {
        Example obj = new Example(10);
        System.out.println(obj.value); // 10
    }
}
```

Constructor Overloading

Constructor overloading means defining multiple constructors with the same name but different parameter lists in the same class. This provides multiple ways to create an object.

Example:

```java
class Example {
    int value;

    Example() { // Default constructor
        value = 0;
    }

    Example(int value) { // Parameterized constructor
        this.value = value;
    }

    public static void main(String[] args) {
        Example obj1 = new Example();       // Calls default constructor
        Example obj2 = new Example(20);     // Calls parameterized constructor

        System.out.println(obj1.value);     // 0
        System.out.println(obj2.value);     // 20
    }
}
```

What is an Interface?

An **interface** in Java is a blueprint of a class. It contains abstract methods (methods without implementation) and constants (static and final by default).

Characteristics:

- Cannot have method implementations (prior to Java 8).
- Java 8 introduced default and static methods in interfaces.
- Classes can implement multiple interfaces to achieve **multiple inheritance**.

Example:

```java
interface Animal {
    void eat(); // Abstract method
}

class Cat implements Animal {
    @Override
    public void eat() {
        System.out.println("Cat eats fish");
    }
}
```

What are Abstract Classes?

An **abstract class** in Java is a class that can have both abstract methods (without implementation) and concrete methods (with implementation).

Characteristics:

- Cannot be instantiated directly.
- Used to provide a base class for subclasses.
- Useful when a class needs partial implementation and partial abstraction.

Example:

```java
abstract class Vehicle {
    abstract void start(); // Abstract method

    void stop() { // Concrete method
        System.out.println("Vehicle stopped.");
    }
}

class Car extends Vehicle {
    @Override
    void start() {
        System.out.println("Car starts with a key.");
    }
}
```

Needs of Interfaces and Abstract Classes

Feature	Interface	Abstract Class
Purpose	Total abstraction (from Java 8: partial)	Partial abstraction
Use Case	For defining behavior (what to do)	For sharing common functionality
Multiple Inheritance	Achieves multiple inheritance	Does not support multiple inheritance
Concrete Methods	Allowed (Java 8: default/static methods)	Allowed

How is Multiple Inheritance Achieved Using Interfaces?

Java allows a class to implement multiple interfaces. Since interfaces only provide method declarations, there's no conflict in method implementation.

Example:

```java
interface A {
    void methodA();
}

interface B {
    void methodB();
}

class C implements A, B {
    public void methodA() {
        System.out.println("Method A from Interface A");
    }

    public void methodB() {
        System.out.println("Method B from Interface B");
    }
}

public class Main {
    public static void main(String[] args) {
        C obj = new C();
        obj.methodA();
        obj.methodB();
    }
}
```

Can Static Methods Be Overridden?

No, static methods cannot be overridden in Java because they belong to the class rather than the object. Overriding works with instance methods that are associated with objects.

Example:

```java
class Parent {
    static void display() {
        System.out.println("Static method in Parent");
    }
}

class Child extends Parent {
    static void display() { // This is method hiding, not overriding
        System.out.println("Static method in Child");
    }
}

public class Main {
    public static void main(String[] args) {
        Parent obj = new Child();
        obj.display(); // Output: Static method in Parent
    }
}
```

If a subclass declares a static method with the same name and signature as in the parent class, the subclass's method hides the parent class's method rather than overriding it.

Differences b/w Encapsulation and Abstraction:

1. **Purpose**:
    - **Encapsulation**: Protects object data and restricts unauthorized access.
    - **Abstraction**: Hides implementation details and exposes only relevant information.
2. **Focus**:
    - **Encapsulation**: Deals with the internal state and representation of the object.
    - **Abstraction**: Deals with the behavior and functionality of the object.
3. **Implementation**:
    - **Encapsulation**: Uses access modifiers (like private and public) and methods (getters/setters).
    - **Abstraction**: Uses abstract classes or interfaces to hide details.
4. **Real-world Analogy**:
    - **Encapsulation**: Think of a car's engine. The engine's details are hidden from the driver (the data is encapsulated), but the driver can interact with the car using controls (methods).
    - **Abstraction**: The car's control interface (steering, pedals) provides an abstraction, allowing the driver to drive without needing to know the inner workings of the engine or transmission.

Here are the differences between **Encapsulation** and **Abstraction** in a concise format:

Aspect	**Encapsulation**	**Abstraction**
Definition	Bundling data and methods that operate on that data within a single unit, while restricting access to the internal details.	Hiding complex implementation details and showing only the essential features or functionalities.
Goal	To protect the internal state of an object and control access to it.	To simplify complex systems by exposing only the necessary parts of the object.
Focus	Internal state and representation of the object.	Behavior and functionality of the object.
Implementation	Achieved through access modifiers (private, public) and getter/setter methods.	Achieved using abstract classes or interfaces.
Access Control	Controls how data is accessed or modified (via getter/setter methods).	Focuses on providing an abstract interface, hiding implementation details.
Real-World Analogy	A car's engine is encapsulated; the driver interacts with the controls but cannot access the engine's internals directly.	A car's controls (steering, pedals) provide an abstraction of driving without needing to know how the engine works.
Example	private String engine; public void setEngine(String engine)	abstract class Animal { abstract void sound(); }

Access Modifiers

Access modifiers control the visibility and accessibility of classes, methods, and variables. Java provides four main access modifiers:

1. **public**:
    - **Visibility**: The member (class, method, variable) is accessible from any other class, regardless of the package.
    - **Usage**: Typically used for classes and methods that need to be accessible globally.
    - **Example**: A public method can be called from any other class in any package.
2. **private**:
    - **Visibility**: The member is only accessible within the same class where it is defined.
    - **Usage**: Used for variables and methods that should not be accessed directly outside the class.
    - **Example**: Private instance variables are used to protect the object's internal state and ensure data encapsulation.
3. **protected**:
    - **Visibility**: The member is accessible within the same package and by subclasses (even if they are in different packages).
    - **Usage**: Often used when you want a method or variable to be visible to subclasses but not to the entire world.
    - **Example**: A protected method or field is accessible in the subclass, even if the subclass is in another package.
4. **Default (Package-Private)**:
    - **Visibility**: If no access modifier is specified, it is known as the default access modifier. The member is accessible only within the same package.
    - **Usage**: Used when you want to restrict access to members within the same package but not outside.
    - **Example**: A method with default access is accessible to other classes in the same package but not outside.

Types of Variables

1. **Local Variables**:
    - **Definition**: Variables declared inside a method, constructor, or block.
    - **Scope**: The scope of a local variable is limited to the block of code (method or constructor) where it is defined.
    - **Initialization**: Must be initialized before use.
    - **Usage**: Typically used for temporary storage or working with values within a specific method.

2. **Instance Variables**:
    - **Definition**: Variables declared within a class but outside any method, constructor, or block.
    - **Scope**: Instance variables are associated with an instance of the class (an object). Each object has its own copy of the instance variables.
    - **Initialization**: Instance variables are initialized when an object is created. If not explicitly initialized, they are assigned default values (e.g., 0 for numbers, null for objects).
    - **Usage**: Used to represent the properties or attributes of an object.
3. **Static Variables**:
    - **Definition**: Variables declared with the static keyword within a class.
    - **Scope**: Static variables are associated with the class itself rather than any specific instance of the class. They are shared by all instances of the class.
    - **Initialization**: Static variables are initialized only once when the class is loaded into memory.
    - **Usage**: Used when you want to have a common property for all instances of a class. For example, a counter that tracks how many objects of a class have been created.
4. **Final Variables**:
    - **Definition**: Variables declared with the final keyword. These variables can be initialized only once and their value cannot be changed after initialization.
    - **Scope**: Final variables can be instance, static, or local variables.
    - **Usage**: Used when you want to define constants or prevent the modification of a variable once it is initialized.
    - **Initialization**: A final variable must be initialized either during declaration or in the constructor (for instance variables) or within a static block (for static variables).

Summary of Differences

Type of Variable	Scope	Initialization	Usage
Local Variable	Inside a method, constructor, or block	Must be initialized before use	Temporary storage for values in a specific method.
Instance Variable	Inside a class (but outside methods)	Initialized when an object is created	Represents attributes of an object.
Static Variable	Associated with the class (shared by all instances)	Initialized when the class is loaded	Shared by all instances of the class.
Final Variable	Can be local, static, or instance	Must be initialized once and cannot be changed	Represents constants or values that should not change.

- **Access Modifiers** control the visibility and accessibility of classes, methods, and variables, ensuring proper encapsulation.
- **Variable Types** define the scope, initialization behavior, and the role of the variables in the program.

Constructors in Java

A **constructor** in Java is a special method that is called when an object of a class is created. The constructor's main purpose is to initialize the newly created object. It has the same name as the class and does not have a return type (not even void).

Key Features of Constructors:

1. **Name**: A constructor must have the same name as the class.
2. **No Return Type**: Constructors do not have a return type, not even void.
3. **Automatic Call**: A constructor is automatically called when an object is created using the new keyword.
4. **Initialization**: Its primary role is to initialize the object's state (assign values to instance variables).

Types of Constructors

There are **two main types of constructors** in Java:

1. **Default Constructor (No-argument Constructor)**:
    - **Definition**: A constructor that does not take any parameters.
    - **Purpose**: It provides default values to the instance variables of the object when no arguments are passed during the object creation.
    - **Behavior**: If no constructor is explicitly defined in the class, Java automatically provides a default constructor that initializes instance variables to their default values (e.g., 0 for numbers, null for objects).
    - **Example**:

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

    // Default constructor
    Car() {
        model = "Unknown";
        year = 2020;
    }
}
```

2. **Parameterized Constructor**:

- **Definition**: A constructor that accepts parameters to initialize the object with specific values.
- **Purpose**: It allows you to set the initial state of an object at the time of creation by passing arguments to the constructor.

- **Behavior**: You define this constructor explicitly and it allows object creation with customized data.
- **Example**:

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

    // Parameterized constructor
    Car(String model, int year) {
        this.model = model;
        this.year = year;
    }
}
```

Constructor Overloading

- **Definition**: Constructor overloading occurs when a class has multiple constructors with different parameter lists. It allows an object to be initialized in different ways.
- **Usage**: You can provide different constructors for different use cases by varying the number or types of parameters.
- **Example**:

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

    // Constructor with no arguments
    Car() {
        this.model = "Unknown";
        this.year = 2020;
    }

    // Constructor with arguments
    Car(String model, int year) {
        this.model = model;
        this.year = year;
    }
}
```

Key Differences Between Default and Parameterized Constructors

Feature	Default Constructor	Parameterized Constructor
Parameters	No parameters	Takes parameters to initialize the object with custom values
Initialization	Initializes with default values (like 0, null, etc.)	Initializes with specific values passed during object creation
Automatic Creation	Automatically created by Java if no constructor is defined	Must be explicitly defined by the programmer
Usage	Used when no initial values are provided	Used when specific values need to be passed during object creation

What is an Interface in Java?

An **interface** in Java is a reference type, similar to a class, that can contain only **abstract methods**, **default methods**, **static methods**, and **constants**. It cannot contain instance fields (variables) and cannot be instantiated directly. Interfaces are used to define a contract that classes can implement, specifying what methods the class must provide without dictating how these methods should be implemented.

Key Features of Interfaces:

1. **Abstract Methods**: Methods in an interface are abstract by default (they do not have a body) and must be implemented by any class that implements the interface.
2. **No Constructors**: Interfaces cannot have constructors because they cannot be instantiated.
3. **Multiple Inheritance**: Interfaces allow multiple inheritance in Java, which means a class can implement more than one interface.
4. **Cannot contain instance variables**: Interfaces can contain only constants (public, static, and final variables).
5. **Methods in Interfaces**:
    - **Abstract methods** (default type for methods before Java 8)
    - **Default methods** (since Java 8, with a default implementation)
    - **Static methods** (since Java 8, with a body)

Types of Methods in Interfaces

1. **Abstract Methods**:
    - The traditional method type in interfaces, which has no body and must be implemented by the implementing class.
    - **Syntax**:
    -
```
interface Vehicle {
    void start();  // Abstract method
}
```

2. **Default Methods** (since Java 8):

- A default method has a body and provides a default implementation. If a class implements the interface and does not override the default method, it will use the default implementation.
- **Syntax:**

```
interface Vehicle {
    default void honk() {
        System.out.println("Honking");
    }
}
```

3. **Static Methods** (since Java 8):

- Static methods in an interface can have a body and are called on the interface itself, not the implementing class.
- **Syntax**:

```
interface Vehicle {
    static void displayInfo() {
        System.out.println("Vehicle Info");
    }
}
```

4. **Private Methods** (since Java 9):

- Private methods can be used within the interface to share code between default methods without being accessible from outside the interface.
- **Syntax:**

```
interface Vehicle {
    private void internalMethod() {
        System.out.println("Internal method");
    }

    default void start() {
        internalMethod();
        System.out.println("Starting the vehicle");
    }
}
```

Difference Between Interface and Abstract Class

Aspect	Interface	Abstract Class
Methods	Can only have abstract methods (prior to Java 8), default methods, and static methods.	Can have abstract methods, concrete methods (with a body), and static methods.
Multiple Inheritance	A class can implement multiple interfaces.	A class can inherit only one abstract class (Java does not support multiple inheritance of classes).
Constructor	Cannot have a constructor.	Can have a constructor.
Instance Variables	Can only have constants (static, final variables).	Can have instance variables that can be initialized in constructors.
Access Modifiers	All methods are implicitly public (except private methods).	Can have methods with any access modifier (e.g., private, protected).
Usage	Used to define a contract for what classes can do.	Used to provide a common base for related classes with shared implementation.
Inheritance	A class can implement multiple interfaces.	A class can extend only one abstract class.

Achieving Multiple Inheritance Using Interfaces

In Java, **multiple inheritance** (i.e., a class inheriting from more than one class) is not allowed for classes. However, you can achieve **multiple inheritance** using **interfaces**. A class can implement multiple interfaces, thus inheriting the behavior and contract from each interface.

How Multiple Inheritance Works in Interfaces:

- A class can implement multiple interfaces, which allows it to inherit the abstract methods of all those interfaces. The class must then provide concrete implementations for all the abstract methods.
- Interfaces allow you to achieve multiple inheritance by providing a way for a class to have multiple types (interfaces) without inheriting multiple class hierarchies.

Example of Multiple Inheritance with Interfaces:

```java
interface Animal {
    void eat();
}

interface Flyable {
    void fly();
}

class Bird implements Animal, Flyable {
    @Override
    public void eat() {
        System.out.println(x:"Bird is eating.");
    }

    @Override
    public void fly() {
        System.out.println(x:"Bird is flying.");
    }
}

public class Main {
    Run | Debug
    public static void main(String[] args) {
        Bird bird = new Bird();
        bird.eat(); // Inherited from Animal interface
        bird.fly(); // Inherited from Flyable interface
    }
}
```

In this example, the Bird class implements both the Animal and Flyable interfaces, effectively inheriting behaviors from both interfaces.

Why Use Interfaces?

1. **Decoupling**: Interfaces allow for decoupling between the class definition and the method implementation, which makes code more flexible and easier to maintain.
2. **Multiple Inheritance**: As Java does not allow multiple inheritance through classes, interfaces provide a way to achieve multiple inheritance by allowing a class to implement multiple interfaces.
3. **Polymorphism**: Interfaces provide a mechanism for polymorphism, as different classes can implement the same interface but provide different implementations for the same methods.
4. **Contract Definition**: Interfaces help in defining a contract that classes can adhere to, making the design cleaner and more modular.

Summary

- **Interfaces** are used to define a contract (methods that a class must implement), and they can contain abstract methods, default methods, static methods, and constants.
- **Abstract Classes** are used to provide partial implementation to be shared among related classes, and they can contain both abstract and concrete methods.
- **Multiple inheritance** is possible in Java using interfaces because a class can implement multiple interfaces, inheriting their abstract methods.