Object-Oriented Programming (OOP) in Java: A Detailed Guide

Object-Oriented Programming (OOP) is a powerful programming paradigm that structures software design around data, or objects, rather than functions and logic. Java is a pure object-oriented language, meaning almost everything in Java is an object, making it an ideal language for understanding and implementing OOP principles.

This document will delve into the core concepts of OOP, explaining each with clear definitions, real-world analogies, and practical Java code examples.

1. Introduction to OOP

At its heart, OOP aims to model real-world entities as software objects. It focuses on creating reusable, modular, and maintainable code by organizing it into self-contained units called "objects." These objects combine both data (attributes or properties) and the functions (methods or behaviors) that operate on that data.

The primary goals of OOP include:

- **Modularity:** Breaking down complex problems into smaller, manageable parts.
- **Reusability:** Writing code once and using it multiple times.
- **Maintainability:** Making it easier to update and debug code.
- **Scalability:** Allowing applications to grow and evolve gracefully.

2. Core Concepts (Pillars) of OOP

OOP is built upon several fundamental concepts, often referred to as its "pillars." These are:

2.1. Classes and Objects

The most basic building blocks of OOP.

- **Class:**
  - A class is a **blueprint** or a **template** for creating objects. It's a logical entity that defines the common characteristics (attributes/data) and behaviors (methods/functions) that all objects of that type will possess.
  - Think of a class like a cookie cutter: it defines the shape and design of the cookies, but it's not a cookie itself.
  - In Java, a class is declared using the class keyword.

  // Example: A 'Dog' class blueprint

```
class Dog {
    // Attributes (data members/properties)
    String name;
    String breed;
    int age;

    // Constructor: A special method to initialize objects
    public Dog(String name, String breed, int age) {
        this.name = name;
        this.breed = breed;
        this.age = age;
    }

    // Methods (behaviors/functions)
    public void bark() {
        System.out.println(name + " barks: Woof woof!");
    }

    public void eat() {
        System.out.println(name + " is eating.");
    }

    public void displayInfo() {
        System.out.println("Name: " + name + ", Breed: " + breed + ", Age: " + age);
    }
}
```

- **Object:**
  - An object is an **instance** of a class. It's a concrete, real-world entity created from the class blueprint.
  - Each object has its own unique set of attribute values, but shares the same behaviors defined by its class.
  - Using the cookie cutter analogy, the actual cookies you bake are the objects. You can bake many cookies (objects) from one cutter (class).
  - In Java, objects are created using the new keyword.

```
public class Kennel {
    public static void main(String[] args) {
        // Creating objects (instances) of the Dog class
        Dog myDog = new Dog("Buddy", "Golden Retriever", 3); // 'myDog' is an object
        Dog anotherDog = new Dog("Lucy", "Labrador", 5);   // 'anotherDog' is another object

        // Calling methods on the objects
```

```
        System.out.println("--- My Dog ---");
        myDog.displayInfo();
        myDog.bark();
        myDog.eat();

        System.out.println("\n--- Another Dog ---");
        anotherDog.displayInfo();
        anotherDog.bark();
        anotherDog.eat();
    }
}
```

2.1.1. Constructors

A constructor is a special type of method that is used to initialize objects. It is called automatically when an object of a class is created using the new keyword.

- **Characteristics of Constructors:**
  - It has the same name as the class.
  - It does not have a return type, not even void.
  - It can have parameters (parameterized constructor) or no parameters (default/no-arg constructor).
  - If you don't define any constructor, Java provides a default no-argument constructor implicitly.
- **Purpose:** To set the initial state of an object (initialize its attributes).

```
class Book {
    String title;
    String author;
    int publicationYear;

    // Default (No-argument) Constructor
    public Book() {
        this.title = "Untitled";
        this.author = "Unknown";
        this.publicationYear = 0;
        System.out.println("Default Book constructor called.");
    }

    // Parameterized Constructor
    public Book(String title, String author, int publicationYear) {
        this.title = title;
```

```
        this.author = author;
        this.publicationYear = publicationYear;
        System.out.println("Parameterized Book constructor called for: " + title);
    }

    public void displayBookInfo() {
        System.out.println("Title: " + title + ", Author: " + author + ", Year: " +
publicationYear);
    }
}

public class Library {
    public static void main(String[] args) {
        Book book1 = new Book(); // Calls the default constructor
        book1.displayBookInfo();

        Book book2 = new Book("The Great Gatsby", "F. Scott Fitzgerald", 1925); //
Calls the parameterized constructor
        book2.displayBookInfo();
    }
}
```

2.1.2. The this Keyword

The this keyword in Java is a reference variable that refers to the current object. It can be used for several purposes:

- **To differentiate between instance variables and local variables (when they have the same name):** This is common in constructors and setter methods.
- **To invoke the current class's constructor (constructor chaining):** this() can be used to call another constructor in the same class.
- **To return the current class instance from a method.**
- **To pass the current object as an argument to a method.**

```
class Student {
    String name;
    int age;
    String studentId;

    // Constructor using 'this' to refer to instance variables
    public Student(String name, int age) {
```

```
        this.name = name; // 'this.name' refers to the instance variable
        this.age = age;   // 'age' refers to the local parameter
        System.out.println("Student (name, age) constructor called.");
    }

    // Constructor chaining using 'this()'
    public Student(String name, int age, String studentId) {
        this(name, age); // Calls the (String, int) constructor
        this.studentId = studentId;
        System.out.println("Student (name, age, id) constructor called.");
    }

    public void setAge(int age) {
        this.age = age; // Use 'this' to assign local 'age' to instance 'age'
    }

    public void displayStudentInfo() {
        System.out.println("Name: " + this.name + ", Age: " + this.age + ", ID: " +
this.studentId);
    }
}

public class School {
    public static void main(String[] args) {
        Student s1 = new Student("Alice", 20);
        s1.displayStudentInfo();

        Student s2 = new Student("Bob", 22, "S12345"); // Calls chained constructor
        s2.displayStudentInfo();

        s1.setAge(21);
        s1.displayStudentInfo();
    }
}
```

2.2. Encapsulation

Encapsulation is the mechanism of **bundling the data (attributes) and the methods (behaviors) that operate on the data into a single unit (a class)**. It also involves

restricting direct access to some of an object's components, meaning you control how the data is accessed and modified. This is often achieved using private access modifiers for attributes and providing public methods (getters and setters) to interact with them.

- **Analogy:** Think of a smartphone. You can use its features (make calls, send messages) through its public interface (buttons, touchscreen), but you don't directly access its internal circuits or battery. Encapsulation hides the complexity and protects the internal state.
- **Benefits:**
  - **Data Hiding/Protection:** Prevents unauthorized or accidental modification of an object's internal state.
  - **Flexibility:** Allows you to change the internal implementation of a class without affecting the code that uses the class.
  - **Maintainability:** Makes code easier to debug and update.

```
class BankAccount {
    // Private attributes: Data is hidden and can only be accessed via methods
    private String accountNumber;
    private double balance;

    // Constructor to initialize the object
    public BankAccount(String accountNumber, double initialBalance) {
        this.accountNumber = accountNumber;
        // Validate initial balance to ensure it's not negative
        if (initialBalance >= 0) {
            this.balance = initialBalance;
        } else {
            System.out.println("Initial balance cannot be negative. Setting to 0.");
            this.balance = 0;
        }
    }

    // Public "getter" method to read the account number
    public String getAccountNumber() {
        return accountNumber;
    }

    // Public "getter" method to read the balance
    public double getBalance() {
        return balance;
    }

    // Public "setter" method (or modifier method) to deposit funds
```

```
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            System.out.println("Deposited: $" + amount + ". New balance: $" + balance);
        } else {
            System.out.println("Deposit amount must be positive.");
        }
    }

    // Public "setter" method (or modifier method) to withdraw funds
    public void withdraw(double amount) {
        if (amount > 0 && balance >= amount) {
            balance -= amount;
            System.out.println("Withdrew: $" + amount + ". New balance: $" + balance);
        } else if (amount <= 0) {
            System.out.println("Withdrawal amount must be positive.");
        } else {
            System.out.println("Insufficient funds. Current balance: $" + balance);
        }
    }
}

public class FinancialApp {
    public static void main(String[] args) {
        BankAccount myAccount = new BankAccount("987654321", 500.00);

        System.out.println("Account Number: " + myAccount.getAccountNumber());
        System.out.println("Current Balance: $" + myAccount.getBalance());

        myAccount.deposit(200.00);
        myAccount.withdraw(100.00);
        myAccount.withdraw(700.00); // This will fail due to insufficient funds

        // myAccount.balance = -100; // This direct access is prevented by 'private'
modifier
    }
}
```

2.3. Inheritance

Inheritance is a mechanism where a new class (called the **subclass**, **derived class**, or **child class**) acquires the properties (attributes) and behaviors (methods) of an existing class (called the **superclass**, **base class**, or **parent class**). This promotes **code reusability** and establishes an **"is-a" relationship** (e.g., a "Dog is an Animal," a

"Car is a Vehicle").

- **Analogy:** A child inherits traits from its parents. The child has its own unique traits but also shares many from its parents.
- **Keyword:** extends is used in Java to indicate inheritance.
- Java supports **single inheritance** (a class can only extend one direct superclass) but **multiple inheritance of interfaces** (a class can implement multiple interfaces).

```
// Superclass (Parent Class)
class Vehicle {
    String brand;
    int year;

    public Vehicle(String brand, int year) {
        this.brand = brand;
        this.year = year;
    }

    public void startEngine() {
        System.out.println(brand + " vehicle engine starting...");
    }

    public void stopEngine() {
        System.out.println(brand + " vehicle engine stopping.");
    }

    public void displayVehicleInfo() {
        System.out.println("Brand: " + brand + ", Year: " + year);
    }
}

// Subclass (Child Class) inheriting from Vehicle
class Car extends Vehicle {
    int numberOfDoors;

    // Constructor for Car, calling superclass constructor using 'super()'
    public Car(String brand, int year, int numberOfDoors) {
        super(brand, year); // Calls the constructor of the Vehicle superclass
        this.numberOfDoors = numberOfDoors;
```

```
    }

    // Car-specific method
    public void drive() {
        System.out.println(brand + " car is driving with " + numberOfDoors + "
doors.");
    }

    // Overriding a method from the superclass (optional, but common)
    @Override
    public void displayVehicleInfo() {
        super.displayVehicleInfo(); // Call superclass method first
        System.out.println("Number of Doors: " + numberOfDoors);
    }
}

// Another Subclass
class Motorcycle extends Vehicle {
    boolean hasSidecar;

    public Motorcycle(String brand, int year, boolean hasSidecar) {
        super(brand, year);
        this.hasSidecar = hasSidecar;
    }

    public void ride() {
        String sidecarStatus = hasSidecar ? "with a sidecar" : "without a sidecar";
        System.out.println(brand + " motorcycle is being ridden " + sidecarStatus +
".");
    }
}

public class Garage {
    public static void main(String[] args) {
        Car myCar = new Car("Toyota", 2023, 4);
        Motorcycle myMotorcycle = new Motorcycle("Harley-Davidson", 2020, true);

        System.out.println("--- My Car ---");
        myCar.displayVehicleInfo(); // Calls overridden method in Car
```

```
        myCar.startEngine();       // Inherited method from Vehicle
        myCar.drive();             // Car-specific method
        myCar.stopEngine();

        System.out.println("\n--- My Motorcycle ---");
        myMotorcycle.displayVehicleInfo(); // Calls method from Vehicle
        myMotorcycle.startEngine();       // Inherited method from Vehicle
        myMotorcycle.ride();              // Motorcycle-specific method
        myMotorcycle.stopEngine();
    }
}
```

2.3.1. The super Keyword

The super keyword in Java is a reference variable that is used to refer to the immediate parent class object. It can be used for:

- **To refer to immediate parent class instance variables:** If the parent class and child class have the same named instance variables.
- **To invoke immediate parent class methods:** If the child class has overridden a method from the parent class, super.methodName() can call the parent's version.
- **To invoke immediate parent class constructors:** super() is used to call the parent class's constructor from the child class's constructor. This *must* be the first statement in the child constructor.

```
class Parent {
    String message = "Hello from Parent";

    Parent() {
        System.out.println("Parent constructor called.");
    }

    void displayMessage() {
        System.out.println("Parent message: " + message);
    }
}

class Child extends Parent {
    String message = "Hello from Child";

    Child() {
```

```
        super(); // Invokes the immediate parent class constructor (optional, automatically added if not present)
        System.out.println("Child constructor called.");
    }

    void displayMessage() {
        System.out.println("Child message: " + message); // Refers to Child's message
        System.out.println("Parent message (via super): " + super.message); // Refers to Parent's message
        super.displayMessage(); // Invokes the immediate parent class method
    }
}

public class SuperKeywordExample {
    public static void main(String[] args) {
        Child c = new Child();
        c.displayMessage();
    }
}
```

2.4. Polymorphism

Polymorphism means "many forms." In OOP, it refers to the ability of an object to take on many forms. More specifically, it allows objects of different classes to be treated as objects of a common type (their superclass or an interface they implement). This enables a single interface to represent different underlying forms.

- **Analogy:** A person can be a "student," a "son/daughter," a "friend," or an "employee" depending on the context. Each role implies different behaviors, but it's still the same person.
- **Types of Polymorphism in Java:**
    - **Compile-time Polymorphism (Method Overloading):**
        - Achieved when a class has multiple methods with the same name but different parameters (different number of arguments, different types of arguments, or different order of arguments).
        - The compiler decides which method to call based on the arguments provided during the method call.
        - This is also known as **static binding**.

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

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

    // Method to add two doubles (overloaded)
    public double add(double a, double b) {
        return a + b;
    }
}

public class OverloadingExample {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println("Sum of 2 and 3: " + calc.add(2, 3));
        System.out.println("Sum of 2, 3, and 4: " + calc.add(2, 3, 4));
        System.out.println("Sum of 2.5 and 3.5: " + calc.add(2.5, 3.5));
    }
}
```

- **Runtime Polymorphism (Method Overriding):**
  - Achieved when a subclass provides a specific implementation for a method that is already defined in its superclass. The method signature (name, return type, and parameters) must be the same.
  - The method to be executed is determined at runtime based on the actual object type, not the reference type.
  - This is also known as **dynamic method dispatch** or **late binding**.
  - The @Override annotation is highly recommended to ensure you are correctly overriding a method.

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

// Subclass overriding makeSound()
```

```
class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Cat says: Meow!");
    }
}

// Subclass overriding makeSound()
class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Dog says: Woof!");
    }
}

public class PolymorphismExample {
    public static void main(String[] args) {
        // Creating objects with superclass reference, but subclass instance
        Animal myCat = new Cat(); // Animal reference, Cat object
        Animal myDog = new Dog(); // Animal reference, Dog object
        Animal genericAnimal = new Animal(); // Animal reference, Animal object

        myCat.makeSound();        // Calls Cat's makeSound()
        myDog.makeSound();        // Calls Dog's makeSound()
        genericAnimal.makeSound(); // Calls Animal's makeSound()

        System.out.println("\n--- Using an array of Animal references ---");
        Animal[] animals = new Animal[3];
        animals[0] = new Cat();
        animals[1] = new Dog();
        animals[2] = new Animal();

        for (Animal animal : animals) {
            animal.makeSound(); // The correct makeSound() is called at runtime
        }
    }
}
```

2.5. Abstraction

Abstraction is the concept of **showing only essential features of an object or system and hiding the complex implementation details**. It focuses on "what" an object does rather than "how" it does it. In Java, abstraction is achieved using **abstract classes** and **interfaces**.

- **Analogy:** When you drive a car, you use the steering wheel, pedals, and gear shift (the essential features). You don't need to know the intricate details of how the engine works or how the transmission shifts gears (the hidden implementation).
- **Benefits:**
  - **Simplification:** Reduces complexity by hiding unnecessary details.
  - **Flexibility:** Allows changes to the internal implementation without affecting the external view.
  - **Security:** Prevents users from accessing sensitive data or methods.
- **Abstract Classes:**
  - A class declared with the abstract keyword.
  - Cannot be instantiated (you cannot create objects of an abstract class directly).
  - Can have both abstract methods (methods without a body, declared with abstract keyword) and concrete methods (methods with a body).
  - Subclasses must implement all abstract methods of their abstract superclass, or they must also be declared abstract.
  - Used when you want to provide a common base for related classes, but some methods need to be implemented differently by each subclass.

```
// Abstract Class
abstract class Shape {
    String color;

    public Shape(String color) {
        this.color = color;
    }

    // Concrete method (has a body)
    public void displayColor() {
        System.out.println("This shape is " + color);
    }

    // Abstract method (no body, must be implemented by subclasses)
    public abstract double calculateArea();

    // Another abstract method
    public abstract void draw();
}

// Concrete subclass of Shape
class Circle extends Shape {
    double radius;
```

```
    public Circle(String color, double radius) {
        super(color);
        this.radius = radius;
    }

    @Override
    public double calculateArea() {
        return Math.PI * radius * radius;
    }

    @Override
    public void draw() {
        System.out.println("Drawing a " + color + " circle with radius " + radius);
    }
}

// Concrete subclass of Shape
class Rectangle extends Shape {
    double length;
    double width;

    public Rectangle(String color, double length, double width) {
        super(color);
        this.length = length;
        this.width = width;
    }

    @Override
    public double calculateArea() {
        return length * width;
    }

    @Override
    public void draw() {
        System.out.println("Drawing a " + color + " rectangle with length " + length + "
and width " + width);
    }
}

public class AbstractClassExample {
    public static void main(String[] args) {
        // Shape s = new Shape("Green"); // ERROR: Cannot instantiate abstract class

        Circle c = new Circle("Blue", 5.0);
        Rectangle r = new Rectangle("Red", 4.0, 6.0);
```

```
        c.displayColor();
        c.draw();
        System.out.println("Area of Circle: " + c.calculateArea());

        System.out.println();

        r.displayColor();
        r.draw();
        System.out.println("Area of Rectangle: " + r.calculateArea());

        // Polymorphism with abstract class
        Shape[] shapes = new Shape[2];
        shapes[0] = c;
        shapes[1] = r;

        System.out.println("\n--- Drawing all shapes ---");
        for (Shape shape : shapes) {
            shape.draw();
            System.out.println("Area: " + shape.calculateArea());
        }
    }
}
```

- **Interfaces:**
  - A blueprint of a class. It contains only abstract methods (before Java 8) and constants. From Java 8 onwards, interfaces can also have default and static methods. From Java 9, they can also have private methods.
  - Declared with the interface keyword.
  - A class uses the implements keyword to implement an interface.
  - A class can implement multiple interfaces (achieving a form of multiple inheritance of behavior).
  - Used when you want to define a contract or a set of behaviors that multiple unrelated classes must adhere to.

```
// Interface
interface Flyable {
    // Abstract method (implicitly public and abstract)
    void fly();
    // Default method (since Java 8)
    default void takeOff() {
        System.out.println("Taking off...");
    }
    // Static method (since Java 8)
```

```
    static void announceFlight() {
        System.out.println("Prepare for flight!");
    }
}

// Another Interface
interface Swimmable {
    void swim();
}

// Class implementing one interface
class Bird implements Flyable {
    String name;
    public Bird(String name) { this.name = name; }

    @Override
    public void fly() {
        System.out.println(name + " is flying high!");
    }
}

// Class implementing multiple interfaces
class Duck implements Flyable, Swimmable {
    String name;
    public Duck(String name) { this.name = name; }

    @Override
    public void fly() {
        System.out.println(name + " is flying, but not very high.");
    }

    @Override
    public void swim() {
        System.out.println(name + " is swimming gracefully.");
    }
}

public class InterfaceExample {
    public static void main(String[] args) {
        Bird eagle = new Bird("Eagle");
        Duck donald = new Duck("Donald");

        Flyable.announceFlight(); // Calling static method of interface

        System.out.println("\n--- Eagle Actions ---");
```

```
        eagle.takeOff(); // Calling default method
        eagle.fly();

        System.out.println("\n--- Donald Actions ---");
        donald.takeOff(); // Calling default method
        donald.fly();
        donald.swim(); // Calling method from Swimmable interface
    }
}
```

2.6. Access Modifiers

Access modifiers in Java control the visibility and accessibility of classes, fields (attributes), constructors, and methods. They are crucial for implementing encapsulation and determining what parts of your code can be accessed from where.

There are four types of access modifiers in Java:

- **public**:
  - **Visibility:** Accessible from anywhere (within the same class, same package, subclasses, and different packages).
  - **Use Case:** Used for methods and variables that need to be universally accessible, often for the public API of a class.
- **private**:
  - **Visibility:** Accessible only within the class where it is declared.
  - **Use Case:** Primarily used for fields to enforce encapsulation (data hiding). Data can only be accessed or modified via public getter/setter methods.
- **protected**:
  - **Visibility:** Accessible within the same package and by subclasses (even if they are in a different package).
  - **Use Case:** Used when you want to allow subclasses to access or override certain members, while still restricting access from unrelated classes outside the package.
- **Default (no keyword)**:
  - **Visibility:** Accessible only within the same package.
  - **Use Case:** Used when you want to restrict access to members to classes within the same package, often for internal helper classes or utility methods that are not part of the public API.

```
// Package 1: com.example.model
package com.example.model;

public class AccessModifierExample {
```

```
    public String publicVar = "Public variable";
    private String privateVar = "Private variable";
    protected String protectedVar = "Protected variable";
    String defaultVar = "Default variable"; // No modifier means default/package-private

    public void publicMethod() {
        System.out.println("This is a public method.");
    }

    private void privateMethod() {
        System.out.println("This is a private method.");
    }

    protected void protectedMethod() {
        System.out.println("This is a protected method.");
    }

    void defaultMethod() {
        System.out.println("This is a default method.");
    }

    public void demonstratePrivate() {
        System.out.println("Accessing privateVar from inside class: " + privateVar);
        privateMethod();
    }
}
```java
// Package 1: com.example.model (Same package as AccessModifierExample)
package com.example.model;

public class SamePackageClass {
    public static void main(String[] args) {
        AccessModifierExample obj = new AccessModifierExample();

        System.out.println(obj.publicVar);      // OK: Public
        // System.out.println(obj.privateVar);  // ERROR: Private
        System.out.println(obj.protectedVar);   // OK: Protected (same package)
        System.out.println(obj.defaultVar);     // OK: Default (same package)

        obj.publicMethod();                 // OK: Public
        // obj.privateMethod();             // ERROR: Private
        obj.protectedMethod();              // OK: Protected (same package)
        obj.defaultMethod();                // OK: Default (same package)

        obj.demonstratePrivate(); // Can call public method that accesses private
```
```

members
    }
}
```java
// Package 2: com.example.app (Different package)
package com.example.app;

import com.example.model.AccessModifierExample;

public class DifferentPackageClass {
    public static void main(String[] args) {
        AccessModifierExample obj = new AccessModifierExample();

        System.out.println(obj.publicVar);      // OK: Public
        // System.out.println(obj.privateVar);  // ERROR: Private
        // System.out.println(obj.protectedVar); // ERROR: Protected (different package,
not subclass)
        // System.out.println(obj.defaultVar);   // ERROR: Default (different package)

        obj.publicMethod();                  // OK: Public
        // obj.privateMethod();               // ERROR: Private
        // obj.protectedMethod();              // ERROR: Protected (different package, not
subclass)
        // obj.defaultMethod();               // ERROR: Default (different package)

        obj.demonstratePrivate(); // Can call public method that accesses private
members
    }
}
```java
// Package 2: com.example.app (Different package, Subclass)
package com.example.app;

import com.example.model.AccessModifierExample;

class SubclassInDifferentPackage extends AccessModifierExample {
    public void testAccess() {
        System.out.println(publicVar);      // OK: Public
        // System.out.println(privateVar);  // ERROR: Private
        System.out.println(protectedVar);   // OK: Protected (subclass)
        // System.out.println(defaultVar);   // ERROR: Default (different package)

        publicMethod();                  // OK: Public
        // privateMethod();               // ERROR: Private
        protectedMethod();                 // OK: Protected (subclass)

```
    // defaultMethod();              // ERROR: Default (different package)
  }

  public static void main(String[] args) {
    SubclassInDifferentPackage subObj = new SubclassInDifferentPackage();
    subObj.testAccess();
  }
}
```

2.7. The final Keyword

The final keyword in Java is used to restrict the user. It can be applied to:

- **final variable:**
  - Once a final variable is initialized, its value cannot be changed. It becomes a constant.
  - Must be initialized at the time of declaration or within the constructor (for instance variables) or static block (for static variables).

```
class Constants {
  final int VALUE = 10; // Final instance variable
  final static double PI = 3.14159; // Final static variable

  void changeValue() {
    // VALUE = 20; // Compile-time error: cannot assign a value to a final variable
  }
}
```

- **final method:**
  - A final method cannot be overridden by any subclass.
  - This is useful when you want to ensure that a specific implementation of a method is used across all subclasses without modification.

```
class ParentClass {
  final void display() {
    System.out.println("This is a final method in ParentClass.");
  }
}

class ChildClass extends ParentClass {
  // @Override
  // void display() { // Compile-time error: cannot override final method
  //     System.out.println("Trying to override final method.");
  // }
}
```

- **final class:**
  - A final class cannot be inherited by any other class.
  - This is useful for security reasons or when you want to prevent extension of a class (e.g., String class in Java is final).

```
final class CannotBeExtended {
   void show() {
      System.out.println("This is a final class.");
   }
}

// class MyClass extends CannotBeExtended { // Compile-time error: cannot inherit
from final class
// }
```

3. The Four Pillars of OOP (Recap)

To summarize, the four fundamental pillars of OOP are:

1. **Encapsulation:** Binding data and methods into a single unit (class) and restricting direct access to data.
2. **Inheritance:** Creating new classes (subclasses) from existing classes (superclasses) to reuse code and establish "is-a" relationships.
3. **Polymorphism:** The ability of an object to take on many forms, allowing a single interface to represent different underlying implementations.
4. **Abstraction:** Hiding complex implementation details and showing only essential features.

4. Benefits of OOP

Adopting the OOP paradigm offers numerous advantages in software development:

- **Code Reusability:** Through inheritance, existing classes can be reused, reducing development time and effort.
- **Modularity and Organization:** Code is organized into self-contained objects, making it easier to understand, manage, and debug.
- **Easier Maintenance:** Changes in one part of the code (e.g., internal implementation of a class) are less likely to affect other parts, thanks to encapsulation.
- **Improved Scalability:** New features and functionalities can be added more easily by extending existing classes or creating new ones.
- **Increased Flexibility:** Polymorphism allows for more adaptable and extensible

code, as objects can be treated generically.
- **Better Security:** Encapsulation helps protect data from unauthorized access or modification.
- **Real-World Modeling:** OOP allows developers to model real-world entities and their interactions more naturally, leading to more intuitive and understandable designs.

5. Relationships Between Objects

Beyond the four pillars, understanding how objects relate to each other is crucial for good OOP design.

- **Association:**
  - A general "uses a" or "has a" relationship between two independent classes.
  - It represents a structural relationship where objects of one class are connected to objects of another.
  - It can be one-to-one, one-to-many, many-to-one, or many-to-many.
  - **Example:** A Student is associated with a Course. A Teacher is associated with a Student.

```
class Student {
    String name;
    // Association: Student 'has a' Course (or is associated with)
    // In a real scenario, this would likely be a List<Course>
    Course enrolledCourse;

    public Student(String name, Course course) {
        this.name = name;
        this.enrolledCourse = course;
    }

    public void displayInfo() {
        System.out.println("Student: " + name + ", Enrolled in: " +
enrolledCourse.getCourseName());
    }
}

class Course {
    String courseName;

    public Course(String courseName) {
        this.courseName = courseName;
    }
```

```
    public String getCourseName() {
        return courseName;
    }
}

public class AssociationExample {
    public static void main(String[] args) {
        Course javaCourse = new Course("Java Programming");
        Student alice = new Student("Alice", javaCourse);
        alice.displayInfo();
    }
}
```

- **Aggregation:**
  - A specialized form of association that represents a "has-a" relationship where one class (the "whole") contains another class (the "part"), but the part can exist independently of the whole.
  - It's a "weak" relationship. If the "whole" object is destroyed, the "part" object can still exist.
  - **Example:** A Department "has a" Professor. If the department ceases to exist, the professor can still exist and potentially join another department.

```
class Professor {
    String name;
    String subject;

    public Professor(String name, String subject) {
        this.name = name;
        this.subject = subject;
    }

    public void teach() {
        System.out.println(name + " is teaching " + subject + ".");
    }
}

class Department {
    String departmentName;
    // Aggregation: Department 'has a' Professor (Professor can exist without
Department)
    Professor head; // A department has a head professor

    public Department(String departmentName, Professor head) {
        this.departmentName = departmentName;
```

```
        this.head = head;
    }

    public void displayDepartmentInfo() {
        System.out.println("Department: " + departmentName);
        if (head != null) {
            System.out.println("Head of Department: " + head.name);
        }
    }
}

public class AggregationExample {
    public static void main(String[] args) {
        Professor profSmith = new Professor("Dr. Smith", "Computer Science");
        Department csDept = new Department("Computer Science", profSmith);

        csDept.displayDepartmentInfo();
        profSmith.teach(); // Professor object exists independently
    }
}
```

- **Composition:**
  - A stronger form of aggregation, also representing a "has-a" relationship, but where the "part" cannot exist independently of the "whole."
  - It's a "strong" relationship. If the "whole" object is destroyed, the "part" object is also destroyed.
  - **Example:** A House "has a" Room. If the house is demolished, the rooms inside it no longer exist as independent entities.

```
class Room {
    String type;
    double area;

    public Room(String type, double area) {
        this.type = type;
        this.area = area;
    }

    public void displayRoomInfo() {
        System.out.println("  Room Type: " + type + ", Area: " + area + " sq.ft.");
    }
}

class House {
```

```
    String address;
    // Composition: House 'has a' Room (Room cannot exist without House)
    // Rooms are created and managed by the House itself
    private Room livingRoom;
    private Room bedroom;

    public House(String address, double livingRoomArea, double bedroomArea) {
        this.address = address;
        this.livingRoom = new Room("Living Room", livingRoomArea); // Room is created
with House
        this.bedroom = new Room("Bedroom", bedroomArea);
    }

    public void displayHouseInfo() {
        System.out.println("House Address: " + address);
        System.out.println("Rooms:");
        livingRoom.displayRoomInfo();
        bedroom.displayRoomInfo();
    }
}

public class CompositionExample {
    public static void main(String[] args) {
        House myHouse = new House("123 Main St", 300.5, 150.0);
        myHouse.displayHouseInfo();
        // If 'myHouse' object is garbage collected, 'livingRoom' and 'bedroom' objects
also cease to exist.
    }
}
```