OOPs

What is OOPs?

OOPs stands for **Object-Oriented Programming System**.
It's a programming style that uses **objects** and **classes** to design programs.

Think of it like building with **LEGO blocks** — each block (object) has its own features and work, and you combine them to create big things (programs)!

Key Idea:

OOP is all about **organizing code** in a way that:

- Is easy to **understand**
- Can be **reused**
- Is easier to **maintain** and **debug**

Example of Real Life:

Let's take a **Car**:

- **Properties (Data)**: color, model, speed
- **Behaviors (Functions)**: start(), drive(), brake()

We can make a **Car class**, and then create many **Car objects** from it — like red car, blue car, sports car, etc.

4 Pillars of OOP

Pillar	Meaning in Simple Terms
1. Encapsulation	Wrapping data and code together (like a capsule) — hides inner details
2. Abstraction	Showing only important stuff, hiding complex parts (like using a TV remote)
3. Inheritance	A child class gets properties from a parent class (like son inherits from father)
4. Polymorphism	One thing, many forms (like one person: student, gamer, artist)

Advantages of OOP

- Makes code **clean and reusable**
- Easy to **update and maintain**
- Helps with **real-world modeling**
- Improves **security** through encapsulation

Simple Example in Java:

```
class Car {
    String color;

    void drive() {
        System.out.println("Car is driving");
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();  // Create object
        myCar.color = "Red";
        myCar.drive();          // Call method
    }
}
```

Class and Object

◇ What is a Class?

A **class** is like a **blueprint** or **template** for creating objects.

Think of it like a **recipe** in cooking. A recipe tells you what ingredients you need and how to make a dish — but the dish itself doesn't exist until you actually cook it.

In the same way, a class **describes** how an object will look and behave, but **it doesn't exist in memory** until you create an object.

Example:

```
class Car {
    String color;
    int speed;

    void drive() {
        System.out.println("Car is driving");
    }
}
```

Here, `Car` is a class. It has:

- **Attributes (variables)**: `color`, `speed`
- **Behaviors (methods)**: `drive()`

◇ What is an Object?

An **object** is a **real-world instance** of a class. It is created **based on the class** and has its own values for the class's properties.

Think of it like **cooking a dish using the recipe**. Once you cook, you have the actual dish — not just the idea.

Example:

```
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car(); // object created
        myCar.color = "Red";
        myCar.speed = 100;

        myCar.drive();
    }
}
```

Here:

- `myCar` is an **object** of class `Car`
- It has its own `color` and `speed`
- When we call `myCar.drive()`, it runs the behavior defined in the class

Analogy

Real Life	OOP Concept
Blueprint of a house	Class
Actual house built from blueprint	Object
Recipe	Class
Cooked dish using recipe	Object
Mobile phone model	Class
Your own mobile phone	Object

Encapsulation

Encapsulation is one of the core concepts in Object-Oriented Programming (OOP). It's about **hiding the internal details** of how an object works and providing **public methods** to access and modify that data. This keeps the object safe and secure from outside interference and misuse.

Think of **encapsulation** like a **capsule** that wraps something inside. You can only interact with the capsule through a controlled interface (like pressing buttons on a remote control). The internal working of the capsule is hidden from you.

Why Encapsulation?

Encapsulation helps to:

1. **Protect the data**: It prevents unauthorized access and modification.
2. **Simplify the code**: By hiding complex details, it's easier to understand how to use objects.
3. **Maintain code**: You can change the internal workings of an object without affecting how other code interacts with it.

How Does Encapsulation Work?

In OOP, we achieve encapsulation by:

1. **Making variables private** – this prevents direct access to them.
2. **Providing public methods (getters and setters)** – these allow controlled access to the variables.

Example of Encapsulation

```
class BankAccount {
    // Private variables - Cannot be accessed directly outside this class
    private double balance;

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

    // Public setter method to set the balance
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        } else {
            System.out.println("Deposit amount must be positive");
        }
    }

    // Public method to withdraw money
    public void withdraw(double amount) {
        if (amount <= balance) {
            balance -= amount;
        } else {
            System.out.println("Insufficient funds");
        }
    }
}
```

Explanation of the Example

- **Private variables**: `balance` is declared as `private`, meaning no other class can directly change or access it.
- **Public methods**: We use `getBalance()`, `deposit()`, and `withdraw()` to interact with the `balance`. These methods control how the balance is modified or accessed, ensuring:
    - Deposits are only positive values.
    - Withdrawals are limited to available funds.

Real-Life Analogy

Imagine you have a **safe** where you store money (the `balance`). The **safe** has a lock, so you can't just open it whenever you want. To access or modify the money inside, you need to use a **key** — this is like the getter and setter methods.

- The **lock** is like making the `balance` variable private.
- The **key** is like the public methods (`deposit()`, `withdraw()`).

You don't have to worry about how the safe works inside; you just use the key to safely manage your money!

Abstraction

Abstraction is another important concept in Object-Oriented Programming (OOP). It refers to the **process of hiding the complex implementation details** of a system and **exposing only the necessary parts** to the user.

In simple terms, abstraction is like using a **TV remote** — you don't need to know how the remote works inside or how it communicates with the TV; you only need to know what buttons to press to change the channel or volume. The internal working is hidden, but the interface is clear and easy to use.

Why Abstraction?

Abstraction helps to:

1. **Simplify** the interface: It allows you to focus on the important things, without worrying about the complex details.
2. **Improve code maintainability**: Since the internal workings can change without affecting how users interact with the system.
3. **Enhance security**: By exposing only necessary parts and hiding the rest, abstraction keeps sensitive data and operations safe.

How Does Abstraction Work?

Abstraction is typically achieved through:

1. **Abstract classes**
2. **Interfaces**

1. Abstract Class

An **abstract class** is a class that cannot be instantiated (i.e., you can't create objects of it directly). It can have both **abstract methods** (methods without a body) and **concrete methods** (methods with a body).

Example of Abstract Class:

```
abstract class Animal {
    // Abstract method (does not have a body)
    abstract void sound();

    // Regular method (does have a body)
    void sleep() {
        System.out.println("The animal is sleeping");
    }
}

class Dog extends Animal {
    // Implementing the abstract method
    void sound() {
        System.out.println("Woof");
    }
}

class Cat extends Animal {
    // Implementing the abstract method
    void sound() {
        System.out.println("Meow");
```

```
    }
}
```

In this example:

- `Animal` is an **abstract class**. It provides an abstract method `sound()` that doesn't have a body, meaning the subclasses must define how it works.
- **Concrete method**: The `sleep()` method is implemented in the abstract class.
- **Subclasses**: `Dog` and `Cat` extend the `Animal` class and provide their own specific implementation of the `sound()` method.

2. Interface

An **interface** is similar to an abstract class, but it can only have **abstract methods** (methods without a body) and **constant variables**. Any class that implements the interface must provide implementations for all of its methods.

Example of Interface:

```
interface Vehicle {
    void start(); // abstract method
    void stop();  // abstract method
}

class Car implements Vehicle {
    public void start() {
        System.out.println("Car is starting");
    }

    public void stop() {
        System.out.println("Car is stopping");
    }
}

class Bike implements Vehicle {
    public void start() {
        System.out.println("Bike is starting");
    }

    public void stop() {
        System.out.println("Bike is stopping");
    }
}
```

Here:

- `Vehicle` is an **interface** with abstract methods `start()` and `stop()`.
- **Implementing the interface**: `Car` and `Bike` implement the `Vehicle` interface, providing their own versions of `start()` and `stop()`.

Real-Life Analogy

Imagine you have a **smartphone**. The phone has **multiple apps** like a **calculator**, **camera**, and **music player**. Each app has a **specific functionality**, but the **complex workings** of how each app functions (like how calculations are done or how images are captured) are hidden from you.

- The **interface** is like the app's **main screen**. You interact with the app using buttons, and you don't need to know how the app works inside.
- The **abstract class** is like a **generic smartphone** that provides some basic features (like calling or texting), but apps (like camera or calculator) need to define specific functions (how to calculate or take a photo).

Inheritance

Inheritance is a fundamental concept in Object-Oriented Programming (OOP). It allows one class to **inherit** the properties and methods of another class. This helps to **reuse code**, **extend functionality**, and **create a relationship between classes**.

Think of inheritance like **family inheritance**. A child inherits traits and qualities from their parents, such as eye color, height, and certain behaviors. Similarly, in OOP, a **child class** inherits attributes and behaviors from a **parent class**.

Why Inheritance?

Inheritance helps to:

1. **Promote code reusability**: You can reuse the functionality of an existing class.
2. **Extend functionality**: You can add more specific features to a class that already exists.
3. **Create hierarchical relationships**: It helps to create a hierarchy or structure in your classes.

How Does Inheritance Work?

In OOP, a class can **inherit** another class using the `extends` keyword (in languages like Java or C#) or by subclassing (in languages like Python).

Example of Inheritance in Java

Let's say we have a parent class called `Animal` and we want to create two child classes: `Dog` and `Cat`. Both `Dog` and `Cat` should have some common properties, like the ability to `eat` and `sleep`, which they can **inherit** from the `Animal` class.

Example:

```
// Parent class
class Animal {
    // Properties
    String name;

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


void sleep() {
        System.out.println(name + " is sleeping");
    }
}

// Child class 1 (Dog)
class Dog extends Animal {
```

```
    // Dog-specific method
    void bark() {
        System.out.println(name + " is barking");
    }
}

// Child class 2 (Cat)
class Cat extends Animal {
    // Cat-specific method
    void meow() {
        System.out.println(name + " is meowing");
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating objects of Dog and Cat
        Dog dog = new Dog();
        dog.name = "Buddy";
        dog.eat();
        dog.bark();

        Cat cat = new Cat();
        cat.name = "Whiskers";
        cat.eat();
        cat.meow();
    }
}
```

Explanation of the Example:

1. **Parent class (`Animal`)**:
   - It has properties (e.g., `name`) and methods (`eat()` and `sleep()`).
   - These are common to **all animals**, so the `Dog` and `Cat` classes will inherit them.
2. **Child classes (`Dog` and `Cat`)**:
   - Both classes `Dog` and `Cat` extend `Animal`, meaning they inherit the properties and methods from `Animal`.
   - Each child class can also have its own specific methods. For example, `Dog` has a `bark()` method, and `Cat` has a `meow()` method.
3. **Main method**:
   - We create objects of `Dog` and `Cat` and call both the inherited methods (`eat()`) and their own specific methods (`bark()` and `meow()`).

Types of Inheritance

1. **Single Inheritance**: A class inherits from only one parent class.
   - Example: A `Dog` inherits from `Animal`.
2. **Multilevel Inheritance**: A class inherits from a parent class, and then another class inherits from that child class.
   - Example: `Dog` inherits from `Animal`, and `Bulldog` inherits from `Dog`.
3. **Hierarchical Inheritance**: Multiple classes inherit from a single parent class.
   - Example: Both `Dog` and `Cat` inherit from `Animal`.
4. **Multiple Inheritance** (not supported directly in some languages like Java, but supported in Python via mixins):
   - Example: A class can inherit from two or more classes (though this is avoided in some languages due to complexity).

1. Single Inheritance

Single inheritance occurs when a class (child class) inherits from **only one parent class**. This is the most basic form of inheritance.

Example:

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

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

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat(); // Inherited from Animal class
        dog.bark(); // Defined in Dog class
    }
}
```

Explanation:

- The `Dog` class inherits the `eat()` method from the `Animal` class.
- `Dog` can have additional methods, like `bark()`, specific to it.

2. Multilevel Inheritance

Multilevel inheritance occurs when a class inherits from another class, and then a **third class** inherits from that child class. In other words, it forms a **chain** of inheritance.

Example:

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

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

// Grandchild class
class Puppy extends Dog {
    void play() {
        System.out.println("Puppy is playing");
    }
```

```
}

public class Main {
    public static void main(String[] args) {
        Puppy puppy = new Puppy();
        puppy.eat();  // Inherited from Animal
        puppy.bark(); // Inherited from Dog
        puppy.play(); // Defined in Puppy
    }
}
```

Explanation:

- `Puppy` inherits from `Dog`, and `Dog` inherits from `Animal`.
- So, `Puppy` inherits methods from both `Dog` and `Animal`.

3. Hierarchical Inheritance

Hierarchical inheritance occurs when multiple classes inherit from a **single parent class**. In this type of inheritance, different child classes share a common base class.

Example:

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

// Child class 1
class Dog extends Animal {
    void bark() {
        System.out.println("Dog is barking");
    }
}

// Child class 2
class Cat extends Animal {
    void meow() {
        System.out.println("Cat is meowing");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat();  // Inherited from Animal
        dog.bark(); // Defined in Dog

        Cat cat = new Cat();
        cat.eat();  // Inherited from Animal
        cat.meow(); // Defined in Cat
    }
}
```

Explanation:

- Both `Dog` and `Cat` inherit from the `Animal` class.
- They can have their own specific methods (`bark()` and `meow()`), but they share the common behavior (`eat()`) from the parent `Animal`.

4. Multiple Inheritance (via Interfaces)

Multiple inheritance occurs when a class inherits from **more than one class**. However, in some languages like Java, **multiple inheritance** is **not directly supported** because it can cause ambiguity (e.g., if two parent classes have the same method, it's unclear which method to inherit). But multiple inheritance is still possible using **interfaces**.

Example in Java (using interfaces):

```
// Interface 1
interface Animal {
    void eat();
}

// Interface 2
interface Pet {
    void play();
}

// Child class
class Dog implements Animal, Pet {
    public void eat() {
        System.out.println("Dog is eating");
    }

    public void play() {
        System.out.println("Dog is playing");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat();  // Implemented from Animal interface
        dog.play(); // Implemented from Pet interface
    }
}
```

Explanation:

- The `Dog` class **implements two interfaces**: `Animal` and `Pet`.
- The interfaces provide method definitions, and the `Dog` class **implements** them with its own behavior.

5. Hybrid Inheritance

Hybrid inheritance is a **combination** of multiple types of inheritance, often combining **single inheritance**, **multilevel inheritance**, and **multiple inheritance** (via interfaces).

Example in Java (using interfaces and classes):

```
// Interface 1
interface Animal {
    void eat();
}

// Parent class
class Mammal implements Animal {
    public void eat() {
        System.out.println("Mammal is eating");
```

```
    }
}

// Child class 1
class Dog extends Mammal {
    void bark() {
        System.out.println("Dog is barking");
    }
}

// Child class 2
class Cat extends Mammal {
    void meow() {
        System.out.println("Cat is meowing");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat();  // Inherited from Mammal
        dog.bark(); // Defined in Dog

        Cat cat = new Cat();
        cat.eat();  // Inherited from Mammal
        cat.meow(); // Defined in Cat
    }
}
```

Explanation:

- `Mammal` implements the `Animal` interface.
- `Dog` and `Cat` inherit from `Mammal` (which already implements `Animal`), creating a **hybrid inheritance** system.

Real-Life Analogy

- Imagine you have a **parent** (a `Vehicle`), and you create different **child classes** like `Car` and `Bike`.
    - The parent `Vehicle` might have general properties like `speed` and methods like `start()`.
    - The `Car` and `Bike` can inherit those properties and methods, and add their own specific features like `airConditioning()` for the car or `kickStart()` for the bike.

Polymorphism

Polymorphism is a core concept of Object-Oriented Programming (OOP), and the word literally means "**many forms**". In programming, **polymorphism allows a single function, method, or operator to behave differently based on the context**—like depending on the data type or the class calling it.

Real-Life Example

Let's say you have a **smartphone**. You use the same "play" button to:

- Play music in the music app
- Play a video in the video app
- Play a game in the games app

Even though you're pressing the same button ("play"), the **behavior changes depending on the context**. That's polymorphism!

Why Polymorphism?

- **Code reusability** – Use the same interface/method name for different behaviors.
- **Scalability** – Add new types/classes without changing existing code.
- **Flexibility** – Change or extend behavior without rewriting code.

Types of Polymorphism

There are **two main types** of polymorphism in OOP:

1. Compile-Time Polymorphism (also called Static Polymorphism)

This happens when the method call is resolved at **compile time**. It's usually achieved using **method overloading** or **operator overloading**.

➤ Method Overloading

- Same method name, different parameter list (number/type/order).
- Decided during compilation.

Example (Java):

```
class MathOperations {
    // Method with 1 int parameter
    void add(int a) {
        System.out.println("Sum: " + (a + 10));
    }

    // Method with 2 int parameters
    void add(int a, int b) {
        System.out.println("Sum: " + (a + b));
    }

    // Method with 2 double parameters
    void add(double a, double b) {
        System.out.println("Sum: " + (a + b));
    }
}

public class Main {
    public static void main(String[] args) {
        MathOperations obj = new MathOperations();
        obj.add(5);          // Calls first method
        obj.add(5, 15);      // Calls second method
        obj.add(2.5, 3.5);   // Calls third method
    }
}
```

2. Run-Time Polymorphism (also called Dynamic Polymorphism)

This occurs when the method to be executed is determined **during program execution**. Achieved using **method overriding** with **inheritance**.

➤ Method Overriding

- A **subclass** provides its own implementation of a method already defined in the **parent class**.
- Happens at runtime using **dynamic dispatch**.

Example (Java):

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

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

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

public class Main {
    public static void main(String[] args) {
        Animal obj1 = new Dog(); // Upcasting
        Animal obj2 = new Cat();

        obj1.sound(); // Output: Dog barks
        obj2.sound(); // Output: Cat meows
    }
}
```

Explanation:

- Both `Dog` and `Cat` override the `sound()` method of `Animal`.
- Although `obj1` and `obj2` are declared as `Animal`, the **actual method that runs depends on the object type at runtime**.

Constructor and Destructor

Constructor

A **constructor** is a special method in a class that is **automatically called** when an object is created. Its main job is to **initialize the object**.

Features of Constructors:

- Same name as the class.
- No return type (not even `void`).
- Called automatically when an object is created.
- Can be **default**, **parameterized**, or **copy** constructors.

Types of Constructors

1. Default Constructor

A **default constructor** is a constructor that takes **no arguments**.

Example:

```
class Car {
    String model;

    // Default constructor
    Car() {
        model = "Unknown";
        System.out.println("Default constructor called.");
    }

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

public class Main {
    public static void main(String[] args) {
        Car c = new Car(); // default constructor is called
        c.display();
    }
}
```

2. Parameterized Constructor

A **parameterized constructor** is a constructor that takes **arguments** to initialize fields with specific values.

Example:

```
class Car {
    String model;

    // Parameterized constructor
    Car(String m) {
        model = m;
        System.out.println("Parameterized constructor called.");
    }

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

public class Main {
    public static void main(String[] args) {
        Car c = new Car("Tesla");
        c.display();
    }
}
```

3. Copy Constructor

A **copy constructor** creates a **new object by copying values from another object** of the same class.

Java doesn't have a built-in copy constructor like C++, but we can define one manually.

Example:

```
class Car {
    String model;

    // Parameterized constructor
    Car(String m) {
        model = m;
    }

    // Copy constructor
    Car(Car c) {
        model = c.model;
    }
    void display() {
        System.out.println("Model: " + model);
    }
}

public class Main {
    public static void main(String[] args) {
        Car car1 = new Car("BMW");
        Car car2 = new Car(car1); // Copy constructor

        car2.display(); // Output: Model: BMW
    }
}
```

Destructor (in C++)

A **destructor** is used to **free up resources** when an object is destroyed. Java doesn't use destructors like C++—instead, it uses **garbage collection**.

In C++:

```
class Car {
public:
    Car() {
        cout << "Constructor called" << endl;
    }

    ~Car() {
        cout << "Destructor called" << endl;
    }
};
```

Access Specifiers (Access Modifiers)

Access specifiers (or modifiers) define the **visibility** or **access level** of classes, variables, methods, and constructors in object-oriented programming. They help in **encapsulation** by controlling **who can access what**.

Types of Access Specifiers in Java

Modifier	Access Level
`private`	Within the same class only
`default`	Within the same package (no keyword used)
`protected`	Same package + subclasses in other packages
`public`	Accessible from **anywhere**

1. Private

- **Least accessible**
- Used to **hide internal details** from outside classes.
- Common for **class variables (fields)**.

Example:

```
class Person {
    private String name;

    public void setName(String n) {
        name = n;
    }

    public void getName() {
        System.out.println("Name: " + name);
    }
}
```

- `name` can only be accessed inside the `Person` class.
- Outside access is only possible through public methods (`getName()`, `setName()`).

2. Default (Package-Private)

- **No keyword** used.
- Accessible **only within the same package**.

Example:

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

- Can't be accessed from a class in another package.

3. Protected

- More accessible than default.
- Accessible in:
    - Same package

- Subclasses (even if in a different package)

Example:

```
class Animal {
    protected void sleep() {
        System.out.println("Animal is sleeping");
    }
}

class Dog extends Animal {
    void demo() {
        sleep(); // Accessible due to protected
    }
}
```

4. Public

- **Most accessible**.
- Accessible from **anywhere** in the program.

Example:

```
public class Hello {
    public void greet() {
        System.out.println("Hello, World!");
    }
}
```

- You can call `greet()` from any class, any package.

Static Members (Variables and Methods)

In Java (and other OOP languages), the keyword `static` means **belonging to the class** rather than to any specific object of the class (means that it is not a part of any object it is a part of class).

So, static members (variables or methods) are **shared** across all objects of a class (means that it is not the part of object, instead all the object can use its value).

1. Static Variables

Also known as **class variables**.

- Only **one copy** (unique value) exists for all objects.
- Shared among all instances of the class.
- Useful for constants or values that are common for all objects.

Example:

```
class Student {
    int rollNo;
    static String college = "ABC College"; // shared by all students

    Student(int r) {
        rollNo = r;
    }
```

```
    void display() {
        System.out.println("Roll No: " + rollNo + ", College: " + college);
    }
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student(1);
        Student s2 = new Student(2);

        s1.display();
        s2.display();
    }
}
```

Output:

```
Roll No: 1, College: ABC College
Roll No: 2, College: ABC College
```

2. Static Methods

- Can be called **without creating an object**.
- Can **only access static data** directly.
- Can't use `this` or access instance (non-static) variables directly.

Example:

```
class Calculator {
    static int square(int x) {
        return x * x;
    }
}

public class Main {
    public static void main(String[] args) {
        int result = Calculator.square(5); // no object needed
        System.out.println("Square: " + result);
    }
}
```

'this' and 'super' Keywords

`this` Keyword

- Refers to the **current object** of the class.
- Used when variable names are **same in constructor and class**.

Example:

```
class Car {
    String model;

    Car(String model) {
        this.model = model; // `this` refers to the instance variable
    }

    void display() {
        System.out.println("Model: " + this.model);
```

```
    }
}
```

`super` Keyword

- Refers to the **parent class** (superclass).
- Used to:
    1. Call **superclass constructor**
    2. Access **superclass methods/variables** if overridden

Example 1: Calling superclass constructor

```
class Animal {
    Animal() {
        System.out.println("Animal constructor");
    }
}

class Dog extends Animal {
    Dog() {
        super(); // calls Animal's constructor
        System.out.println("Dog constructor");
    }
}
```

Example 2: Accessing superclass method

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

class Dog extends Animal {
    void sound() {
        super.sound(); // Calls Animal's sound()
        System.out.println("Dog barks");
    }
}
```

final Keyword

The `final` keyword in Java means **"you can't change it."**
You can use `final` with:

- **Variables**
- **Methods**
- **Classes**

Let's go through each one with simple examples.

1. Final Variable

A **final variable** means **you can assign a value only once** — after that, you **can't change it**.

Think of it like a constant — once set, it stays the same forever.

Example:

```
class Demo {
    final int speedLimit = 60; // final variable

    void show() {
        // speedLimit = 80; ❌ Not allowed
        System.out.println("Speed Limit: " + speedLimit);
    }
}
```

2. Final Method

A **final method** means **you can't override it in a child class**.

Think of it like a rule that children (subclasses) can't change.

Example:

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

class Dog extends Animal {
    // void sound() ❌ Not allowed - can't override final method
}
```

3. Final Class

A **final class** means **no other class can inherit (extend) it**.

It's like saying "This class is complete. No changes allowed."

Example:

```
final class Vehicle {
    void run() {
        System.out.println("Vehicle is running");
    }
}

// class Car extends Vehicle ❌ Not allowed - final class can't be extended
```

Abstract Classes and Interfaces

What is an Abstract Class?

- A class that **cannot be fully used on its own**.
- It can have **both abstract methods (without body)** and **normal methods (with body)**.
- You **must extend** it in a child class and complete the abstract methods.

Think of it like a half-done template — you need to complete it before using.

Example:

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

    void eat() {
        System.out.println("This animal eats food.");
    }
}

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

What is an Interface?

- Like a **contract** or **set of rules** that a class must follow.
- Contains only **abstract methods** (until Java 7).
- From Java 8+, it can also have **default and static methods**.
- A class **implements** an interface.

Think of it like a list of instructions — any class that signs the contract must follow all the rules.

Example:

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

class Cat implements Animal {
    public void sound() {
        System.out.println("Cat meows");
    }
}
```

Method Overloading and Overriding

Method Overloading (Compile-time Polymorphism)

What is it?

- **Same method name**, but with **different number or type of parameters** in the **same class**.
- Happens during **compile time**.

Think of it like a multi-function tool: One button, many uses depending on what input you give.

Example:

```
class MathTool {
    // Method 1
    int add(int a, int b) {
        return a + b;
    }
```

```
    // Method 2
    double add(double a, double b) {
        return a + b;
    }

    // Method 3
    int add(int a, int b, int c) {
        return a + b + c;
    }
}
```

All methods are called `add()`, but they work with different inputs.

Method Overriding (Run-time Polymorphism)

What is it?

- When a **child class provides its own version** of a method that is already defined in the **parent class**.
- Method name and parameters must be **exactly same**.
- Happens during **runtime**.

Think of it like a child changing a rule from the parent — same method, but different behavior.

Example:

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

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

The `Dog` class **overrides** the `sound()` method of the `Animal` class.

Difference Between Overloading and Overriding

Feature	Method Overloading	Method Overriding
Based on	Number or type of parameters	Inheritance (parent-child relationship)
Class Type	Same class	Different classes (Inheritance)
Method Signature	Different	Same
Return Type	Can be same or different	Should be same or compatible
Timing	Compile time	Run time
Keyword	Not required	`@Override` (optional but recommended)

Real-Life Example:

Overloading:

- **Printer** can print:
  - Only text: `print(String text)`
  - A photo: `print(Image photo)`
  - A file: `print(File file)`

Overriding:

- **Parent** says: "Wake up at 7 AM"
- **Teen child** says: "Nah, I'll wake up at 9 AM" — same instruction, new version

Exception Handling in OOP

What is an Exception?

An **exception** is an **error** that happens **while your program is running**.

Imagine: You're dividing 10 by 0 in your code — that's a problem!
The program crashes unless you **handle the exception**.

What is Exception Handling?

Exception Handling means writing special code to **catch errors** and **keep your program from crashing**.

It's like putting a **seatbelt** in your car — it **protects** your program from accidents (errors).

Why is it important in OOP?

- Helps write **safe, crash-free** programs.
- Keeps your code **clean and understandable**.
- Follows the **OOP principle of robustness** (your code can handle problems).

How do we handle exceptions in Java?

We use **5 main keywords**:

◇ `try`

Write code that **might throw an error** inside this block.

◇ `catch`

Used to **catch and handle** the exception.

◇ `finally`

Optional — this block **always runs**, whether there's an error or not.

- **`throw`**

Used to **manually throw** an exception.

- **`throws`**

Used to **declare** that a method might throw an exception.

Simple Example:

```
public class Example {
    public static void main(String[] args) {
        try {
            int result = 10 / 0; // This will cause an exception
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Error: You can't divide by zero!");
        } finally {
            System.out.println("This always runs, even if there's an error.");
        }
    }
}
```

Output:

```
Error: You can't divide by zero!
This always runs, even if there's an error.
```

Common Types of Exceptions:

Exception Type	What it means
`ArithmeticException`	Dividing by zero, math errors
`NullPointerException`	Using an object that is null
`ArrayIndexOutOfBounds`	Accessing an invalid array index
`IOException`	Input/output error (like file not found)
`ClassNotFoundException`	Class not found while loading

Real-Life Example:

- **Try**: "Let me open this file."
- **Catch**: "Oops! File not found. Let me show a message instead of crashing."
- **Finally**: "I'll close the file anyway, if it was open."