

OOPS

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects," which are instances of classes. These objects encapsulate data in the form of attributes and behavior in the form of methods. OOP emphasizes modularity, reusability, and abstraction, making it easier to design and maintain complex systems.

Key Features of OOP:

1. **Encapsulation:** Bundling data and methods that operate on the data within a single unit (class) and restricting direct access to some components.
2. **Inheritance:** Enabling new classes to derive properties and behaviors from existing classes, promoting code reuse.
3. **Polymorphism:** Allowing objects to be treated as instances of their parent class or as different types based on context.
4. **Abstraction:** Hiding implementation details and showing only the essential features of an object.

Procedural Programming vs. OOP:

Feature	Procedural Programming	Object-Oriented Programming
Structure	Follows a top-down approach with functions as building blocks.	Follows a bottom-up approach with objects as building blocks.
Data Handling	Data and functions are separate.	Data and functions are encapsulated within objects.
Code Reusability	Reusability is limited, achieved through function calls.	Promotes reusability through inheritance and polymorphism.
Scalability	Less suited for large, complex systems.	Well-suited for large, scalable systems.
Security	Limited control over data access.	Offers better control with access modifiers (e.g., private, public).
Example Languages	C, Pascal, Fortran	Java, Python, C++, Ruby

What does it mean by top bottom and bottom up:

1. Top-Down Approach:

In the **top-down** approach, the focus is on breaking down a larger problem into smaller, more manageable sub-problems or tasks. The overall structure is designed first, and the details are filled in later.

- **Process:**
 1. Start with a high-level design or goal.
 2. Divide the goal into smaller sub-tasks or modules.
 3. Implement each module step-by-step.
 4. Combine all modules to form the complete system.
 - **Key Characteristics:**
 1. **Design-driven:** Begin with the "big picture."
 2. Focuses on the **flow of control** (what happens next).
 3. Commonly used in **procedural programming**.
 - **Example:** In a program to calculate the average of numbers:
 1. Decide the main steps: input, calculate, output.
 2. Break them into smaller tasks: accept user input, sum the numbers, divide by count.
 3. Implement these tasks one by one.
-

2. Bottom-Up Approach:

In the **bottom-up** approach, the focus is on designing and implementing smaller, reusable components (like classes or functions) first, which are then combined to form the complete system.

- **Process:**
 1. Start by creating small, independent modules or components.
 2. Integrate these modules to build the larger system.
 3. The complete program emerges from assembling the components.
- **Key Characteristics:**
 1. **Component-driven:** Start with building blocks.
 2. Focuses on **data** and how it's managed.
 3. Commonly used in **object-oriented programming**.
- **Example:** In the same program to calculate the average:
 1. First, create a function to add numbers.
 2. Create another function to divide the sum by the count.
 3. Combine these functions into the main program.

3. Classes and Objects:

1. Class:

- A **class** is a blueprint or template for creating objects. It defines the structure and behavior that the objects created from the class will have.
- A class encapsulates **attributes** (data) and **methods** (functions) to define the properties and actions of an object.

2. Object:

- An **object** is an instance of a class. It is a concrete entity that has a state (data) and behavior (methods) defined by the class.
- Think of objects as real-world entities (e.g., a car, a person, a book), and the class as the design plan for such entities.

Example:

Let's consider the example of a car:

- **Class:** A class represents the general structure of a car (e.g., Car).
 - **Attributes** (data): color, model, brand, speed.
 - **Methods** (behavior): drive(), stop(), accelerate().
- **Object:** Specific cars are objects of the class "Car".
 - Example 1: A red Honda Civic that can drive at 100 km/h.
 - Example 2: A blue Toyota Corolla that can accelerate to 120 km/h.

```
// Defining a Class
class Car {
    // Attributes of the class
    String brand;
    String color;

    // Constructor to initialize the object
    Car(String brand, String color) {
        this.brand = brand;
        this.color = color;
    }

    // Method to display car details
    void displayInfo() {
        System.out.println("This is a " + color + " " + brand + ".");
    }

    // Method to simulate driving
    void drive() {
        System.out.println("The " + color + " " + brand + " is driving.");
    }
}
```

```
// Main class to run the program
public class Main {
    public static void main(String[] args) {
        // Creating objects of the Car class
        Car car1 = new Car("Honda", "Red");
        Car car2 = new Car("Toyota", "Blue");

        // Using methods of the Car class
        car1.displayInfo(); // Output: This is a Red Honda.
        car1.drive();       // Output: The Red Honda is driving.

        car2.displayInfo(); // Output: This is a Blue Toyota.
        car2.drive();       // Output: The Blue Toyota is driving.
    }
}
```

PLEASE NOTE: The dot (.) is used to access the object's attributes and methods.

To call a method in Java, write the method name followed by a set of parentheses (), followed by a semicolon (;). Eg. `classname.FnName()`;

You will often see Java programs that have either static or public attributes and methods.

In the example above, we created a `dswdsstatic` method, which means that it can be accessed without creating an object of the class, unlike public, which can only be accessed by objects:

You can also modify attribute values:

```
public class Main {  
    int x;  
  
    public static void main(String[] args) {  
        Main myObj = new Main();  
        myObj.x = 40;  
        System.out.println(myObj.x);  
    }  
}
```

If you don't want the ability to override existing values, declare the attribute as `final`:

```
public class Main {  
    final int x = 10;  
  
    public static void main(String[] args) {  
        Main myObj = new Main();  
        myObj.x = 25; // will generate an error: cannot assign a value to  
a final variable  
        System.out.println(myObj.x);  
    }  
}
```

The `final` keyword is useful when you want a variable to always store the same value, like PI (3.14159...).

If you create multiple objects of one class, you can change the attribute values in one object, without affecting the attribute values in the other:

Change the value of `x` to 25 in `myObj2`, and leave `x` in `myObj1` unchanged:

```
public class Main {  
  
    int x = 5;  
  
    public static void main(String[] args) {  
  
        Main myObj1 = new Main(); // Object 1  
  
        Main myObj2 = new Main(); // Object 2  
  
        myObj2.x = 25;  
  
        System.out.println(myObj1.x); // Outputs 5  
  
        System.out.println(myObj2.x); // Outputs 25  
  
    }  
  
}
```

4. Constructors in Java

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

Key Features of Constructors:

1. **Initialization:** Constructors initialize the object's attributes when it is created.
2. **Implicit Call:** A constructor is automatically called when an object is instantiated.
3. **Same Name as Class:** The constructor name must match the class name.
4. **No Return Type:** Constructors don't return a value, not even `void`.

Types of Constructors:

There are three types of constructors in Java:

1. Default Constructor:

- A default constructor is one without parameters. It is provided by Java automatically if no constructor is defined in the class.
- If a custom constructor is defined, Java does not provide a default one.

```
class Car {  
    String brand;  
    String color;  
  
    // Default Constructor  
    Car() {  
        brand = "Unknown";  
        color = "White";  
    }  
  
    void displayInfo() {  
        System.out.println("Car Brand: " + brand + ", Color: " + color);  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Car car = new Car(); // Default constructor is called  
        car.displayInfo();    // Output: Car Brand: Unknown, Color: White  
    }  
}
```



2. Parameterized Constructor:

- A constructor that takes arguments to initialize the object with specific values.

```
class Car {  
    String brand,color;  
  
    // Parameterized Constructor  
    Car(String brand, String color) {  
        this.brand = brand;  
        this.color = color;  
    }  
  
    void displayInfo() {  
        System.out.println("Car Brand: " + brand + ", Color: " + color);  
    }  
}
```

3. Copy Constructor:

- A copy constructor creates a new object by copying the attributes of an existing object.
- Java does not provide a built-in copy constructor, but it can be implemented manually.

```
class Car {  
    String brand;  
    String color;  
  
    // Parameterized Constructor  
    Car(String brand, String color) {  
        this.brand = brand;
```



```

        this.color = color;
    }

    // Copy Constructor
    Car(Car car) {
        this.brand = car.brand;
        this.color = car.color;
    }

    void displayInfo() {
        System.out.println("Car Brand: " + brand + ", Color: " + color);
    }
}

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

        car1.displayInfo(); // Output: Car Brand: Honda, Color: Red
        car2.displayInfo(); // Output: Car Brand: Honda, Color: Red
    }
}

```

Constructor Overloading:

- You can define multiple constructors in the same class with different parameter lists (number or type of parameters). This is called **constructor overloading**.

5. Access Modifiers in Java

Access modifiers in Java define the **visibility** or **scope** of classes, methods, and variables. They determine how accessible a particular component is from other parts of the program. There are **four types** of access modifiers in Java:

1. Public:

- **Scope:** Accessible **everywhere** in the program, including outside the package.
- **Usage:** Used when the method, variable, or class needs to be visible to all other classes and packages.

```
package example;


public class Car {
    public String brand; // Public variable

    public void displayBrand() { // Public method
        System.out.println("Brand: " + brand);
    }
}

package test;

import example.Car;

public class Main {
    public static void main(String[] args) {
        Car car = new Car();
        car.brand = "Honda"; // Accessible because it's public
        car.displayBrand();  // Accessible because it's public
    }
}
```



2. Private:

- **Scope:** Accessible **only within the class** where it is declared.
- **Usage:** Used to protect sensitive data by restricting access. Commonly used with **getter** and **setter** methods.

```
class Car {  
    private String brand; // Private variable  
  
    // Setter method to set the value of the private variable  
    public void setBrand(String brand) {  
        this.brand = brand;  
    }  
  
    // Getter method to access the value of the private variable  
    public String getBrand() {  
        return brand;  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Car car = new Car();  
        // car.brand = "Honda"; // Error: brand has private access  
        car.setBrand("Honda"); // Set the brand using setter  
        System.out.println(car.getBrand()); // Access the brand using getter  
    }  
}
```

3. Protected:

- **Scope:** Accessible:
 - Within the same package.
 - In **subclasses** (child classes) even if they are in different packages.

- **Usage:** Commonly used for inheritance when you want the child class to access the parent class's methods or variables.

```
package example;
```

```
public class Car {  
    protected String brand; // Protected variable  
  
    protected void displayBrand() { // Protected method  
        System.out.println("Brand: " + brand);  
    }  
}
```

```
package test;
```

```
import example.Car;
```

```
public class Main extends Car { // Subclass of Car  
    public static void main(String[] args) {  
        Main car = new Main();  
        car.brand = "Toyota";    // Accessible because of inheritance  
        car.displayBrand();      // Accessible because of inheritance  
    }  
}
```

4. Default (No Modifier):

- **Scope:** Accessible:
 - **Only within the same package** (also known as package-private).
- **Usage:** Used when you want to restrict access to classes or members within the same package

package example;

```
class Car { // No modifier: default access
    String brand; // Default access variable

    void displayBrand() { // Default access method
        System.out.println("Brand: " + brand);
    }
}
```

package test;

import example.Car;

```
public class Main {
    public static void main(String[] args) {
        // Car car = new Car(); // Error: Car is not public and cannot be accessed outside its
package
    }
}
```

Comparison Table:

Modifier	Same Class	Same Package	Subclass (Different Package)	Other Packages
Public	✓	✓	✓	✓
Private	✓	✗	✗	✗
Protected	✓	✓	✓	✗
Default	✓	✓	✗	✗

Non-Access Modifiers

For classes, you can use either `final` or `abstract`:

Modifier	Description
<code>final</code>	The class cannot be inherited by other classes (You will learn more about inheritance in the Inheritance chapter)
<code>abstract</code>	The class cannot be used to create objects (To access an abstract class, it must be inherited from another class. You will learn more about inheritance and abstraction in the Inheritance and Abstraction chapters)

For attributes and methods, you can use the one of the following:

Modifier	Description
<code>final</code>	Attributes and methods cannot be overridden/modified
<code>static</code>	Attributes and methods belongs to the class, rather than an object
<code>abstract</code>	Can only be used in an abstract class, and can only be used on methods. The method does not have a body, for example <code>abstract void run();</code> . The body is provided by the subclass (inherited from). You will learn more about inheritance and abstraction in the Inheritance and Abstraction chapters

Abstract

An `abstract` method belongs to an `abstract` class, and it does not have a body. The body is provided by the subclass:

```
// Code from filename: Main.java
```

```
// abstract class
```

```
abstract class Main {  
    public String fname = "John";  
    public int age = 24;  
    public abstract void study(); // abstract method  
}
```

```
// Subclass (inherit from Main)
```

```
class Student extends Main {  
    public int graduationYear = 2018;  
    public void study() { // the body of the abstract method is  
        provided here  
        System.out.println("Studying all day long");  
    }  
}
```

```
// End code from filename: Main.java
```

```
// Code from filename: Second.java
```

```
class Second {  
    public static void main(String[] args) {  
        // create an object of the Student class (which inherits  
        attributes and methods from Main)  
        Student myObj = new Student();  
  
        System.out.println("Name: " + myObj.fname);  
        System.out.println("Age: " + myObj.age);  
        System.out.println("Graduation Year: " + myObj.graduationYear);  
  
        myObj.study(); // call abstract method  
    }  
}
```

6. Java Packages & API

A package in Java is used to group related classes. Think of it as a folder in a file directory. We use packages to avoid name conflicts, and to write a better maintainable code. Packages are divided into two categories:

- Built-in Packages (packages from the Java API)
- User-defined Packages (create your own packages)

Built-in Packages

The Java API is a library of prewritten classes, that are free to use, included in the Java Development Environment.

The library is divided into packages and classes. Meaning you can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package.

To use a class or a package from the library, you need to use the `import` keyword:

```
import package.name.Class;    // Import a single class
```

```
import package.name.*;       // Import the whole package
```

Import a Class

If you find a class you want to use, for example, the `Scanner` class, which is used to get user input, write the following code:

Example

```
import java.util.Scanner;
```

In the example above, `java.util` is a package, while `Scanner` is a class of the `java.util` package.

To use the `Scanner` class, create an object of the class and use any of the available methods found in the `Scanner` class documentation. In our example, we will use the `nextLine()` method, which is used to read a complete line:

```
import java.util.*;
```

To import a whole package, end the sentence with an asterisk sign (*). The following example will import ALL the classes in the `java.util` package:

User-defined Packages

To create your own package, you need to understand that Java uses a file system directory to store them. Just like folders on your computer:

MyPackageClass.java

```
package mypack;
class MyPackageClass {
    public static void main(String[] args) {
        System.out.println("This is my package!");
    }
}
```

Encapsulation in Java

Encapsulation is one of the fundamental principles of Object-Oriented Programming (OOP). It refers to the bundling of **data** (fields) and **methods** (functions) that operate on the data into a single unit (class) and restricting direct access to the data from outside the class.

Memory updated

Encapsulation in Java

Encapsulation is one of the fundamental principles of Object-Oriented Programming (OOP). It refers to the bundling of **data** (fields) and **methods** (functions) that operate on the data into a single unit (class) and restricting direct access to the data from outside the class.

Key Characteristics of Encapsulation:

1. **Data Hiding:**
 - By declaring fields as **private**, the internal details of an object are hidden from outside access.
 - Access to the data is provided using **public** getter and setter methods.
2. **Controlled Access:**
 - Encapsulation allows us to control how the data is accessed and modified.
3. **Improved Maintainability:**
 - Changes in the internal implementation of a class do not affect the code that uses the class.
4. **Enhanced Security:**
 - Sensitive data can be protected by restricting direct access.

// Encapsulation Example: Student Class

```
class Student {  
  
    // Private fields  
  
    private String name;  
  
    private int age;  
  
  
    // Public getter for name  
  
    public String getName() {  
        return name;  
    }  
  
  
    // Public setter for name  
  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

```
// Public getter for age
```

```
public int getAge() {
```

```
    return age;
```

```
}
```

```
// Public setter for age
```

```
public void setAge(int age) {
```

```
    // Adding validation for age
```

```
    if (age > 0) {
```

```
        this.age = age;
```

```
    } else {
```

```
        System.out.println("Age must be a positive number.");
```

```
    }
```

```
}
```

```
}
```

```
public class Main {
```

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

```
        // Creating a Student object
```

```
        Student student = new Student();
```

```
        // Using setter methods to set values
```

```
        student.setName("John");
```

```
        student.setAge(20);
```

```
// Using getter methods to access values

System.out.println("Student Name: " + student.getName()); // Output: Student Name: John

System.out.println("Student Age: " + student.getAge()); // Output: Student Age: 20


// Invalid age input

student.setAge(-5); // Output: Age must be a positive number.

}

}
```

Advantages of Encapsulation:

1. **Data Protection:**
 - Prevents unauthorized access and accidental modification of data.
2. **Flexibility:**
 - You can modify the internal logic (e.g., add validation) without changing the external interface.
3. **Reusability:**
 - Encapsulation promotes modularity, making code easier to reuse.
4. **Easy Maintenance:**
 - Changes are localized within the class, reducing the ripple effect on other parts of the application.

Real-World Use Cases of Encapsulation

Encapsulation is widely used in real-world applications to create modular, maintainable, and secure code. Here are some examples that illustrate the practical applications of encapsulation:

1. Banking System

In a banking application, sensitive information such as account numbers, balances, and transaction history should be private and accessible only through controlled methods.

```
class BankAccount {

    private String accountNumber; // Private field

    private double balance;        // Private field
```

```
// Constructor to initialize account

public BankAccount(String accountNumber, double initialBalance) {

    this.accountNumber = accountNumber;

    this.balance = initialBalance;

}


// Getter for account number

public String getAccountNumber() {

    return accountNumber;

}


// Getter for balance

public double getBalance() {

    return balance;

}


// Method to deposit money

public void deposit(double amount) {

    if (amount > 0) {

        balance += amount;

        System.out.println("Deposited: " + amount);

    } else {
```

```
        System.out.println("Invalid deposit amount.");
    }
}

// Method to withdraw money
public void withdraw(double amount) {
    if (amount > 0 && amount <= balance) {
        balance -= amount;
        System.out.println("Withdrew: " + amount);
    } else {
        System.out.println("Invalid withdrawal amount or
insufficient balance.");
    }
}

}

public class Main {
    public static void main(String[] args) {
        BankAccount account = new BankAccount("1234567890", 5000.0);

        // Accessing account details through methods
        System.out.println("Account Number: " +
account.getAccountNumber());
    }
}
```

```
        System.out.println("Current Balance: " +
account.getBalance());

        // Depositing and withdrawing money

        account.deposit(2000.0);

        account.withdraw(3000.0);

        System.out.println("Updated Balance: " +
account.getBalance());

    }

}
```

Inheritance in Java

Inheritance is an Object-Oriented Programming (OOP) concept where one class (the **child** or **subclass**) derives or inherits properties and behaviors (fields and methods) from another class (the **parent** or **superclass**). This promotes **code reusability** and establishes a natural hierarchy between classes.

Key Features of Inheritance:

1. **Code Reusability:**
 - Common code in the parent class can be reused by child classes.
2. **Extensibility:**
 - Child classes can add additional features or override parent class features.
3. **Method Overriding:**
 - The child class can provide a specific implementation for a method already defined in the parent class.
4. **Types of Relationships:**
 - Represents an "is-a" relationship. For example, a "Dog is an Animal."

Syntax of Inheritance

Java uses the `extends` keyword for inheritance.

java

CopyEdit

```
class Parent {  
  
    // Fields and methods of the parent class  
  
}  
  
class Child extends Parent {  
  
    // Fields and methods of the child class  
  
}
```

Types of Inheritance in Java:

1. **Single Inheritance:**
A class inherits from a single parent class.
Example: `Dog extends Animal`.
2. **Multilevel Inheritance:**
A class inherits from another class, which in turn inherits from another class.
Example: `Puppy extends Dog, Dog extends Animal`.
3. **Hierarchical Inheritance:**
Multiple classes inherit from a single parent class.
Example: `Dog` and `Cat` both extend `Animal`.
4. **Multiple Inheritance through Interfaces:**
Java does not support multiple inheritance with classes, but it supports it with interfaces.

Real-Life Examples of Inheritance

1. Vehicle Example (Single Inheritance)

```
// Parent class

class Vehicle {

    int wheels;

    String fuelType;

    public void displayDetails() {

        System.out.println("Wheels: " + wheels);

        System.out.println("Fuel Type: " + fuelType);

    }

}

// Child class

class Car extends Vehicle {

    String brand;

    int seats;

    public void displayCarDetails() {

        displayDetails(); // Call parent method

        System.out.println("Brand: " + brand);

        System.out.println("Seats: " + seats);

    }

}
```

```
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Car car = new Car();  
        car.wheels = 4;  
        car.fuelType = "Petrol";  
        car.brand = "Toyota";  
        car.seats = 5;  
  
        car.displayCarDetails();  
    }  
}
```

Output:

Wheels: 4

Fuel Type: Petrol

Brand: Toyota

Seats: 5

2. Banking System (Multilevel Inheritance)

```
// Grandparent class  
class Account {  
    String accountNumber;
```

```
        public void showAccountNumber() {  
            System.out.println("Account Number: " + accountNumber);  
        }  
    }  
  
    // Parent class  
    class SavingsAccount extends Account {  
        double interestRate;  
  
        public void showInterestRate() {  
            System.out.println("Interest Rate: " + interestRate + "%");  
        }  
    }  
  
    // Child class  
    class FixedDeposit extends SavingsAccount {  
        int maturityPeriod;  
  
        public void showMaturityPeriod() {  
            System.out.println("Maturity Period: " + maturityPeriod + "  
years");  
        }  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        FixedDeposit fd = new FixedDeposit();  
        fd.accountNumber = "12345678";  
        fd.interestRate = 5.5;  
        fd.maturityPeriod = 5;  
  
        fd.showAccountNumber();  
        fd.showInterestRate();  
        fd.showMaturityPeriod();  
    }  
}
```

Output:

Account Number: 12345678

Interest Rate: 5.5%

Maturity Period: 5 years

3. Animal Example (Hierarchical Inheritance)

```
// Parent class

class Animal {

    String name;

    public void eat() {

        System.out.println(name + " is eating.");

    }

}

// Child class 1

class Dog extends Animal {

    public void bark() {

        System.out.println(name + " is barking.");

    }

}

// Child class 2

class Cat extends Animal {

    public void meow() {

        System.out.println(name + " is meowing.");

    }

}
```

```
}

public class Main {

    public static void main(String[] args) {

        Dog dog = new Dog();

        dog.name = "Buddy";

        dog.eat();

        dog.bark();


        Cat cat = new Cat();

        cat.name = "Kitty";

        cat.eat();

        cat.meow();

    }

}
```

Output:

Buddy is eating.

Buddy is barking.

Kitty is eating.

Kitty is meowing.

4. Employee Management System (Method Overriding)

```
// Parent class
class Employee {
    public void work() {
        System.out.println("Employee is working.");
    }
}

// Child class
class Manager extends Employee {
    @Override
    public void work() {
        System.out.println("Manager is managing the team.");
    }
}

// Another child class
class Developer extends Employee {
    @Override
    public void work() {
        System.out.println("Developer is writing code.");
    }
}

public class Main {
    public static void main(String[] args) {
        Employee emp1 = new Manager();
        emp1.work(); // Manager's work method will be called

        Employee emp2 = new Developer();
        emp2.work(); // Developer's work method will be called
    }
}
```

Output:

```
Manager is managing the team.
Developer is writing code.
```

So, when you write `X a = new X();`, here's what's happening:

- **X a**: You're declaring a variable `a` of type `X`.
- **new X()**: You're creating a new object of class `X` using the constructor `X()`, and assigning that object to the variable `a`.

This is a standard way of declaring and initializing an object in Java.

for method overriding why we do `Employee emp1 = new Manager();` but we do `FixedDeposit fd = new FixedDeposit();` for multilevel etc

This difference arises due to the concept of **polymorphism** in Object-Oriented Programming (OOP), which allows an object of a **child class** to be referenced by a **parent class type**. Let's break it down:

The line `Employee emp1 = new Manager();` in Java is an example of **polymorphism** and **inheritance**. Here's what it means:

1. **Employee** is the type of the variable `emp1`. This means `emp1` can reference any object of type `Employee` or any subclass of `Employee`.
2. **new Manager()** creates a new instance of the `Manager` class. Even though `Manager` is a subclass of `Employee`, it's an object of type `Manager`, not `Employee`.
3. **Employee emp1 = new Manager();** means that the reference variable `emp1` is of type `Employee`, but it's actually holding an object of type `Manager`. This is allowed because `Manager` is a subclass of `Employee`, and every `Manager` is also an `Employee` (since inheritance works this way).

The line `Employee emp1 = new Manager();` in Java is an example of **polymorphism** and **inheritance**. Here's what it means:

1. **Employee** is the type of the variable `emp1`. This means `emp1` can reference any object of type `Employee` or any subclass of `Employee`.
2. **new Manager()** creates a new instance of the `Manager` class. Even though `Manager` is a subclass of `Employee`, it's an object of type `Manager`, not `Employee`.
3. **Employee emp1 = new Manager();** means that the reference variable `emp1` is of type `Employee`, but it's actually holding an object of type

`Manager`. This is allowed because `Manager` is a subclass of `Employee`, and every `Manager` is also an `Employee` (since inheritance works this way).

This line demonstrates **upcasting**, where an object of a subclass (`Manager`) is referenced by a variable of the superclass type (`Employee`). At runtime, when `emp1.work()` is called, Java will look for the `work()` method in the `Manager` class due to method overriding (dynamic dispatch). Therefore, even though `emp1` is of type `Employee`, it will call the overridden `work()` method from the `Manager` class.

This is an example of runtime polymorphism, where the method that gets executed is determined at runtime based on the actual type of the object (`Manager`) rather than the reference type (`Employee`).

The final Keyword

If you don't want other classes to inherit from a class, use the `final` keyword:

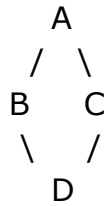
```
final class Vehicle{
    ..
}
class Car extends Vehicle{
    ..
} // cant inherit.
```

Java does **not support multiple inheritance** for classes. This means that a class cannot directly inherit from more than one class.

Why Java Doesn't Support Multiple Inheritance for Classes:

The primary reason is to avoid ambiguity and complexity in the code. If a class could inherit from multiple classes, it could lead to situations where the compiler or runtime can't decide which method or property to use when there are conflicting definitions in the parent classes.

The most common issue that arises is the **diamond problem**, where a class inherits from two classes that have the same method. This would create ambiguity in which method should be called.



Example of the Diamond Problem:

```
class A {
    void show() {
        System.out.println("Class A show()");
    }
}

class B extends A {
    void show() {
        System.out.println("Class B show()");
    }
}

class C extends A {
    void show() {
        System.out.println("Class C show()");
    }
}

// This will lead to a conflict in the next class
class D extends B, C { // This is illegal in Java
    // Which "show" method should be inherited?
}
```

In the above example:

- D would inherit `show()` from both B and C, but which one should it use? Should it use the method from B or from C?
- Java does not allow this type of ambiguity.

How Java Solves This Issue:

While Java doesn't support multiple inheritance with classes, it **does support multiple inheritance through interfaces**. A class can implement multiple interfaces, which allows it to inherit behavior from multiple sources, without the complications of the diamond problem.

Example with Interfaces:

```
interface A {
    void show();
}

interface B {
    void display();
}

class C implements A, B {
    public void show() {
        System.out.println("Class C show()");
    }

    public void display() {
        System.out.println("Class C display()");
    }
}

public class Main {
    public static void main(String[] args) {
        C obj = new C();
        obj.show();    // Class C show()
        obj.display(); // Class C display()
    }
}
```

```
}
```

In this case, `C` implements both `A` and `B` interfaces, which is allowed in Java. Since interfaces do not contain method implementations, there is no conflict. The class `C` is responsible for providing implementations for the methods `show()` and `display()`.

Difference bw Abstract class and Interface:

Abstract Class: An abstract class is used to represent a base class that provides partial implementation of some methods, but cannot be instantiated directly. It can be used to define common behavior for subclasses while leaving some methods to be implemented by the subclasses.

Interface: An interface is a contract that specifies a set of methods that a class must implement. It doesn't provide any implementation (except for default methods introduced in Java 8). A class that implements an interface must provide concrete implementations for all the methods defined in the interface.

```
abstract class Animal {
    abstract void makeSound(); // abstract method (no body)

    void sleep() { // concrete method (with body)
        System.out.println("Animal is sleeping.");
    }
}

abstract class Dog extends Animal {
    void makeSound() {
        System.out.println("Woof!");
    }
}
```

Interface: Interfaces, by default, can only contain **abstract methods** (until Java 8). However, starting from Java 8, interfaces can have **default methods** (with implementation) and **static methods** (also with implementation).

```

interface Animal {
    void makeSound(); // abstract method (no body)

    default void sleep() { // default method (with body)
        System.out.println("Animal is sleeping.");
    }
}

```

```

interface Animal {
    void makeSound();
}

```

```

interface Mammal {
    void nurse();
}

```

```

class Dog implements Animal, Mammal {
    public void makeSound() {
        System.out.println("Woof!");
    }

    public void nurse() {
        System.out.println("Nursing puppies.");
    }
}

```

Polymorphism in Java

Polymorphism is one of the core principles of Object-Oriented Programming (OOP). The term "polymorphism" means "many forms." In Java, it allows objects to behave in multiple ways based on their runtime type or context.

Types of Polymorphism in Java

Java supports two main types of polymorphism:

1. Compile-Time Polymorphism (Static Binding):

- Achieved through **method overloading**.
- The method to be called is determined at **compile time**.
- Example: Multiple methods in the same class with the same name but different parameters.

2. Runtime Polymorphism (Dynamic Binding):

- Achieved through **method overriding**.
- The method to be called is determined at **runtime**, based on the actual object type.
- Example: A parent class reference pointing to a child class object.

1. Compile-Time Polymorphism (Method Overloading)

Method Overloading occurs when multiple methods in the same class have the **same name** but different **parameter lists** (different type, number, or sequence of parameters).

Example:

```
class Calculator {  
    // Method to add two integers  
    public int add(int a, int b) {  
        return a + b;  
    }  
  
    // Overloaded method to add three integers  
    public int add(int a, int b, int c) {  
        return a + b + c;  
    }  
  
    // Overloaded method to add two double values  
    public double add(double a, double b) {  
        return a + b;  
    }  
}
```

```

    }

    public class Main {
        public static void main(String[] args) {
            Calculator calc = new Calculator();

            System.out.println(calc.add(5, 10));           // Calls
add(int, int)
            System.out.println(calc.add(5, 10, 15));      // Calls
add(int, int, int)
            System.out.println(calc.add(5.5, 10.5));      // Calls
add(double, double)
        }
    }
}

```

Output:

```

15
30
16.0

```

2. Runtime Polymorphism (Method Overriding)

Method Overriding allows a child class to provide a **specific implementation** of a method that is already defined in the parent class. It enables **dynamic method dispatch**, where the method to be executed is determined at runtime based on the object's actual type.

Rules for Method Overriding:

- The method in the child class must have the **same name, return type,** and **parameters** as in the parent class.
- The access modifier of the overriding method cannot be more restrictive than the method in the parent class.
- The overriding method can call the parent method using `super`.

Example:

```
class Animal {
    public void sound() {
        System.out.println("Animal makes a sound");
    }
    public void try(){
        System.out.println("Called using super keyword");
    }
}
class Dog extends Animal {
    @Override
    public void sound() {
        super.try();
        System.out.println("Dog barks");
    }
}
class Cat extends Animal {
    @Override
    public void sound() {
        System.out.println("Cat meows");
    }
}
public class Main {
    public static void main(String[] args) {
        Animal myAnimal; // Parent reference

        myAnimal = new Dog(); // Dog object
        myAnimal.sound();      // Calls Dog's sound() method
        myAnimal = new Cat(); // Cat object
        myAnimal.sound();      // Calls Cat's sound() method
    }
}
```

Output:

```
Dog barks
Cat meows
```


Why Use Polymorphism?

1. Code Flexibility:

- Write generic code that works with multiple related classes.
- Example: Using a `List` interface to refer to `ArrayList` or `LinkedList`.

2. Dynamic Behavior:

- Allows behavior to change at runtime, depending on the actual object type.

3. Promotes Abstraction:

- Focus on what an object can do (its interface) rather than its specific implementation.

Difference Between Compile-Time and Runtime Polymorphism

Aspect	Compile-Time Polymorphism	Runtime Polymorphism
Achieved Through	Method Overloading	Method Overriding
Binding	Static Binding (at compile time)	Dynamic Binding (at runtime)
Method Resolution	Compiler determines the method to call	JVM determines the method to call
Flexibility	Less flexible	More flexible (based on object type)
Example	Calculator with overloaded <code>add</code> method	Animal classes with overridden <code>sound</code> method

Abstraction in Java

Abstraction is a core principle of Object-Oriented Programming (OOP) that focuses on hiding implementation details and showing only the essential features of an object or system.

In Java, abstraction can be achieved using:

1. **Abstract Classes**
2. **Interfaces**

Why Use Abstraction?

- **Simplifies Code:** Users interact with high-level interfaces rather than worrying about the underlying implementation.
- **Promotes Modularity:** Helps divide complex systems into smaller, manageable components.
- **Improves Maintainability:** Changes in the internal implementation don't affect users.
- **Encourages Reusability:** Abstraction isolates functionality, making it reusable in different contexts.

1. Abstract Classes

An **abstract class** is a class that cannot be instantiated directly. It can have:

- Abstract methods (methods without a body).
- Concrete methods (methods with a body).

Syntax:

```
abstract class ClassName {  
    abstract void methodName(); // Abstract method (no  
implementation)  
    void anotherMethod() {      // Concrete method (with  
implementation)  
        // Method body  
    }  
}
```

Key Points:

- Abstract classes provide a base for subclasses to build upon.
- If a class contains at least one abstract method, it must be declared as **abstract**.
- Subclasses of an abstract class must override all abstract methods unless the subclass itself is declared abstract.

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

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

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

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

public class Main {
    public static void main(String[] args) {
        Animal dog = new Dog();
        dog.makeSound(); // Output: Dog barks.
        dog.eat();       // Output: This animal eats food.

        Animal cat = new Cat();
        cat.makeSound(); // Output: Cat meows.
        cat.eat();       // Output: This animal eats food.
    }
}
```

2. Interfaces

An **interface** is a blueprint for a class that specifies what a class must do, but not how to do it. It defines a contract that implementing classes must fulfill.

Key Features:

- All methods in an interface are **abstract by default** (in older versions of Java).
- From Java 8 onward:
 - You can include **default methods** (methods with implementation).
 - You can also include **static methods**.
- From Java 9 onward:
 - Private methods are allowed inside interfaces.

```
interface InterfaceName {  
    void methodName(); // Abstract method  
}
```

Example:

```
interface Shape {  
    void draw(); // Abstract method  
}
```

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

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

```
public class Main {
```

```

public static void main(String[] args) {
    Shape circle = new Circle();
    circle.draw(); // Output: Drawing a circle.

    Shape rectangle = new Rectangle();
    rectangle.draw(); // Output: Drawing a rectangle.
}
}

```

Abstract Classes vs Interfaces

Feature	Abstract Class	Interface
Inheritance	A class can extend only one abstract class (single inheritance).	A class can implement multiple interfaces (multiple inheritance).
Methods	Can have abstract and concrete methods.	Methods are abstract by default (can have default and static methods in Java 8+).
Fields	Can have instance variables (non-final).	Can only have public static final variables (constants).
Access Modifiers	Abstract methods can have public, protected, or default access.	Methods are always public by default.
Use Case	When there is a shared state or implementation among subclasses.	When defining a contract for unrelated classes.

So abstract class can contain both abstract method and concrete method. If abstract method is there you have to implement it in the child class.

Interfaces (Before Java 8)

Before **Java 8**, **interfaces** were very different from **abstract classes** in Java. Interfaces could only contain **abstract methods** (i.e., methods with no implementation) and **constants** (i.e., **static final** variables). The key

difference was that **interfaces could not provide any method implementation at all.**

Interfaces could only:

- Declare abstract methods (methods without bodies).
- Declare **public static final** fields (constants).
- A class implementing the interface was required to provide an implementation for all the methods declared in the interface.

4. Key Differences Before Java 8:

Feature	Abstract Class (Before Java 8)	Interface (Before Java 8)
Methods	Can have both abstract and concrete methods.	Can only have abstract methods.
Instance Variables	Can have instance variables (fields).	Can only have <code>public static final</code> constants.
Constructors	Can have constructors.	Cannot have constructors.
Method Implementation	Can provide method implementations.	Cannot provide method implementations (unless it's <code>default</code> or <code>static</code> in Java 8).
Multiple Inheritance	Can only inherit from one abstract class.	Can implement multiple interfaces.
Access Modifiers	Can have private, protected, public modifiers.	Methods are <code>public</code> by default.

In an **abstract class**, you can declare and initialize instance variables (fields) just like you would in a regular class. These variables can have any access modifier (`private`, `protected`, or `public`), and they can be used by the class's methods or subclasses.

Before **Java 8**, **interfaces** could **only** have **constants** (i.e., `public static final` fields). All fields in interfaces were implicitly `public`, `static`, and `final`, meaning they were **constants** and could not be modified.

Key Points for Interfaces (Before Java 8):

- **Variables are implicitly `public`, `static`, and `final`** (constants).
- You cannot have instance variables (fields) that belong to individual objects.
- You cannot have non-static variables or fields in interfaces.

Other imp points:

Final Keyword

- **Final variables:** Constants.
- **Final methods:** Cannot be overridden.
- **Final classes:** Cannot be subclassed.

Discuss use cases like immutable classes (e.g., `String` in Java).

Nested and Inner Classes

- **Static nested classes:** Associated with the outer class, accessed without an instance.
- **Non-static inner classes:** Require an instance of the outer class.

Show how inner classes can enhance encapsulation.

Garbage Collection in Java

Garbage Collection (GC) in Java is a **process of automatic memory management**. It aims to reclaim memory occupied by objects that are no longer in use, preventing memory leaks and ensuring efficient use of system resources.

How Garbage Collection Works in Java

1. **Managed Heap:**
 - The JVM allocates memory in the **heap** for objects.
 - Objects are created, used, and eventually discarded when no longer referenced.
2. **Garbage Collector:**

- It automatically identifies and deallocates memory for unreachable objects (objects that are no longer referenced in the program).
- It operates in the background as part of the JVM.
- 3. **Reachability:** An object is considered for garbage collection when it becomes unreachable. This happens when:
 - There are no references to the object.
 - All references to the object go out of scope.

Steps in Garbage Collection

1. **Marking Phase:**
 - The GC identifies all the objects that are reachable from the root references.
 - Roots include **static fields, local variables in methods, and references from the current thread stack.**
2. **Deletion Phase:**
 - Unreachable objects are removed, and their memory is reclaimed.
3. **Compacting Phase** (optional):
 - After deletion, the remaining objects are compacted to free up contiguous memory space.

Heap Memory Structure in Java

The heap is divided into regions for better memory management:

1. **Young Generation (New Space):**
 - Divided into:
 - **Eden Space:** Where new objects are created.
 - **Survivor Spaces (S0 and S1):** Holds objects that survive garbage collection in Eden.
2. **Old Generation (Tenured Space):**
 - Stores objects that have a long lifespan (promoted from the young generation).
3. **Metaspace:**
 - Stores class metadata (replaced **PermGen** in Java 8+).

Types of Garbage Collection

1. **Minor GC:**
 - Cleans up the **Young Generation**.
 - Frequent but fast.

2. **Major GC:**

- Cleans up the **Old Generation**.
- Less frequent but slower.

3. **Full GC:**

- Cleans up the entire heap (Young + Old + Metaspace).
- Typically triggered explicitly or by memory pressure.