**Inheritance: i**s one in which a new class is created that inherits the properties of the already exist class. It supports the concept of **code reusability** and **reduces the length of the code** in object-oriented programming.

A class that is derived from another class is called a **subclass** (also a **derived class**, **extended class**, or **child class**). The class from which the subclass is derived is called a **superclass** (also a **base class** or a **parent class**).

**Types of Inheritance are:**

1. Single/simple inheritance
2. Multi-level inheritance
3. Multiple inheritances
4. Hybrid inheritance
5. Hierarchical inheritance



### Multiple inheritance in Java

### Single Inheritance:

In Single Inheritance one class extends another class (one class only).

### Multiple Inheritance:

Multiple Inheritance is one of the inheritance in Java types where one class extending more than one class. Java does not support multiple inheritance.

As per above diagram, Class C extends Class A and Class B both.

### Multilevel Inheritance:

In Multilevel Inheritance, one class can inherit from a derived class. Hence, the derived class becomes the base class for the new class.

As per shown in diagram Class C is subclass of B and B is a of subclass Class A.

### Hierarchical Inheritance:

In Hierarchical Inheritance, one class is inherited by many sub classes.

As per above example, Class B, C, and D inherit the same class A.

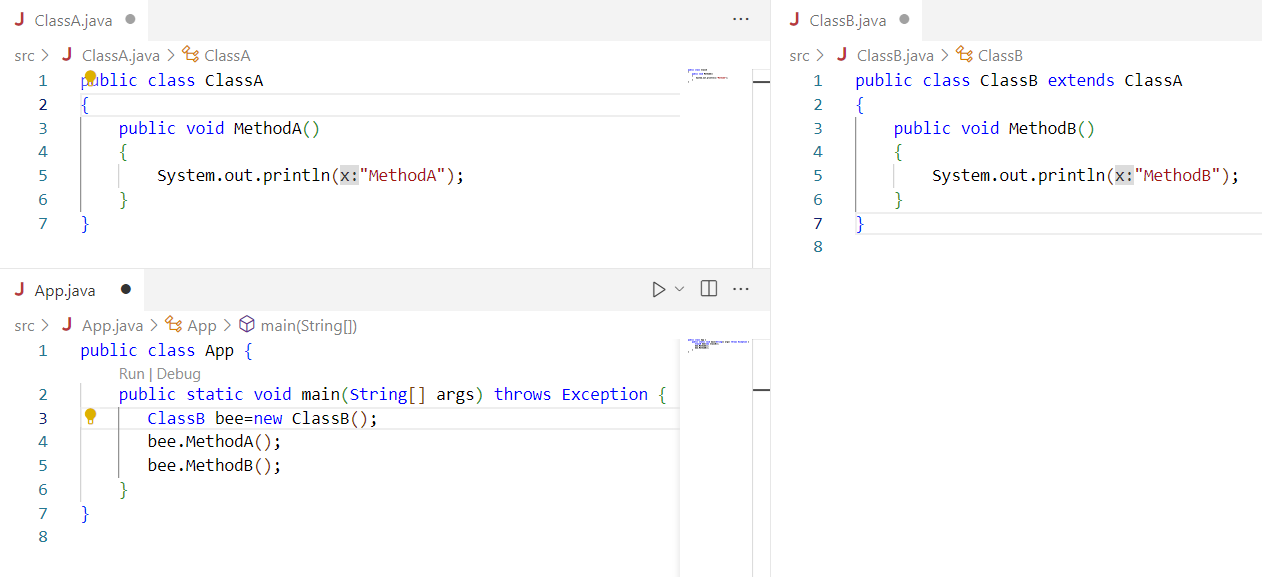
### Hybrid Inheritance:

Hybrid inheritance is one of the inheritance types in Java which is a combination of Single and Multiple inheritance.

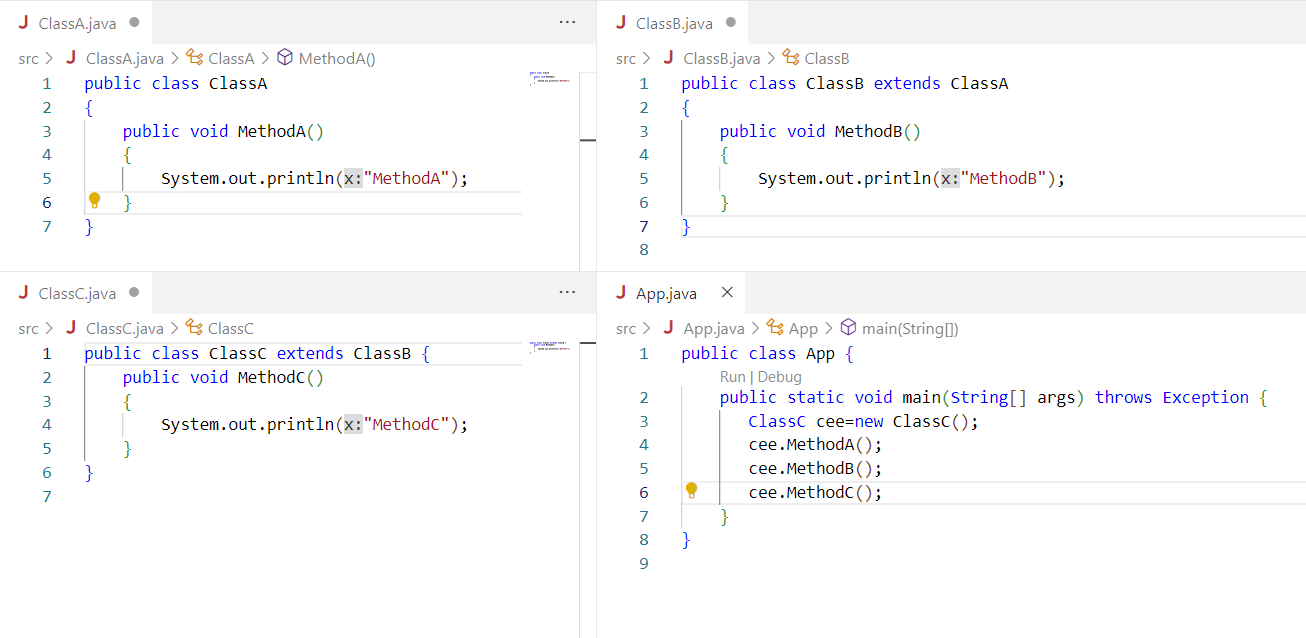
As per above example, all the public and protected members of Class A are inherited into Class D, first via Class B and secondly via Class C.

**Note:** Java doesn’t support hybrid/Multiple inheritance

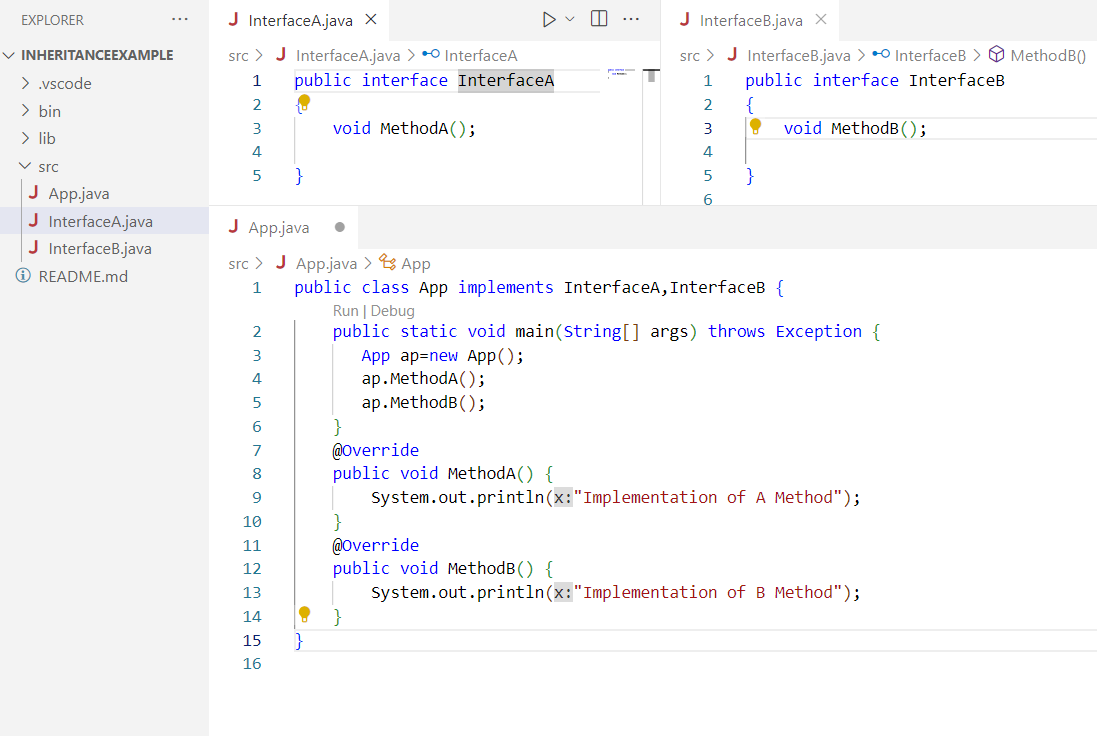
**Single Inheritance Example:**

****

**Multilevel Inheritance Example:**

****

**Multiple Inheritance using interface**



Lab: Write a java program to Calculate area of rectangle and perimeter using multiple inheritance

public class App {

    public static void main(String[] args) throws Exception {

        Calculation cal=new Calculation(5, 6);

        System.out.println(cal.calculateArea());

        System.out.println(cal.calculatePerimeter());

     }

    }

 interface Area{

    double calculateArea();

 }

 interface Perimeter{

    double calculatePerimeter();

 }

 class Calculation implements Area,Perimeter{

    private double length,breadth;

    public Calculation(double l, double b)

    {

        this.length=l;

        this.breadth=b;

    }

    public double calculatePerimeter() {

        return length\*breadth;

    }

    public double calculateArea() {

       return 2\*(length+breadth);

    }

 }

**Extend and Super Keyword  
The extends keyword** extends a class (indicates that a class is inherited from another class).  
In Java, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:

**subclass** (child) - the class that inherits from another class  
**superclass** (parent) - the class being inherited from

To inherit from a class, use the **extends** keyword.

**The super keyword** refers to superclass (parent) objects.It is used to call **superclass** methods, and to access the superclass **constructor**.The most common use of the **super** keyword is to eliminate the confusion between **superclasses** and **subclasses** that have methods with **the same name.**

# Constructor Chaining

Constructor chaining is the process of calling one constructor from another constructor with respect to current object.

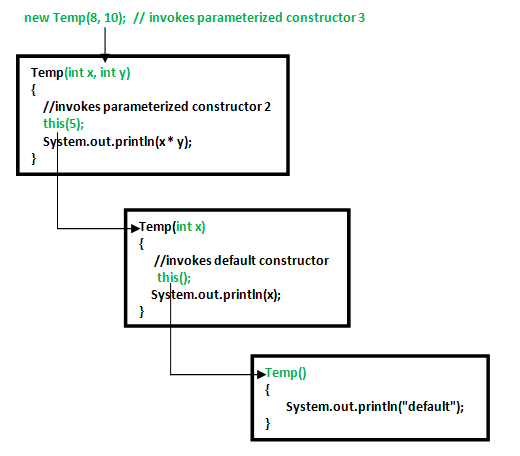
One of the main use of constructor chaining is to avoid duplicate codes while having multiple constructor (by means of constructor overloading) and make code more readable.

**Constructor chaining can be done in two ways:**

* **By using this() keyword:** It is used when we want to call the current class constructor within the same class.
* **By using super() keyword:** It is used when we want to call the superclass constructor from the base class.

**Why do we need constructor chaining?**

This process is used when we want to perform multiple tasks in a single constructor rather than creating a code for each task in a single constructor we create a separate constructor for each task and make their chain which makes the program more readable.



**Example**: Using **this** Keyword

public class App {

    public static void main(String[] args) throws Exception {

        ConstructorChain ch=new ConstructorChain();

    }

}

class ConstructorChain

{

    ConstructorChain()

    {

    this("Phoenix College of Management");

    System.out.println("Default Constructor is Called");

    }

    ConstructorChain(String str)

    {

    System.out.println("Parametrize Constructor is Called");

    }

}

**Output:**

Parametrize Constructor is Called

Default Constructor is Called

**Invoking Base class Constructor:**

A derived class can call a constructor in its base class using the **super** keyword.

Example: using **Super()** keyword

public class App {

    public static void main(String[] args) throws Exception {

        ChildClass cls=new ChildClass();

    }

}

class BaseClass

{

    BaseClass()

    {

    System.out.println("Base Class Constructor is Called");

    }

}

class ChildClass extends BaseClass

{

    ChildClass()

    {

        super();

        System.out.println("From Base Class Constructor");

    }

}

**Output:**

Base Class Constructor is Called

From Base Class Constructor

**Overriding methods:**

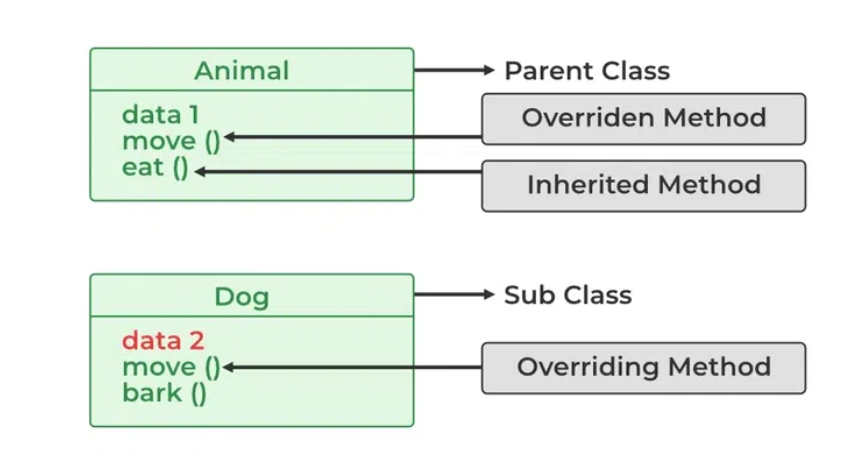
**Method overriding** is a feature that allows a subclass, or a *child class*, to specifically implement a method already given in one of its super-classes, or *parent classes*, in any object-oriented programming language.

Thus, the process of redefining a parent class’s method in a subclass is known as *method overriding*. It is also called **run time polymorphism** or **dynamic binding.**

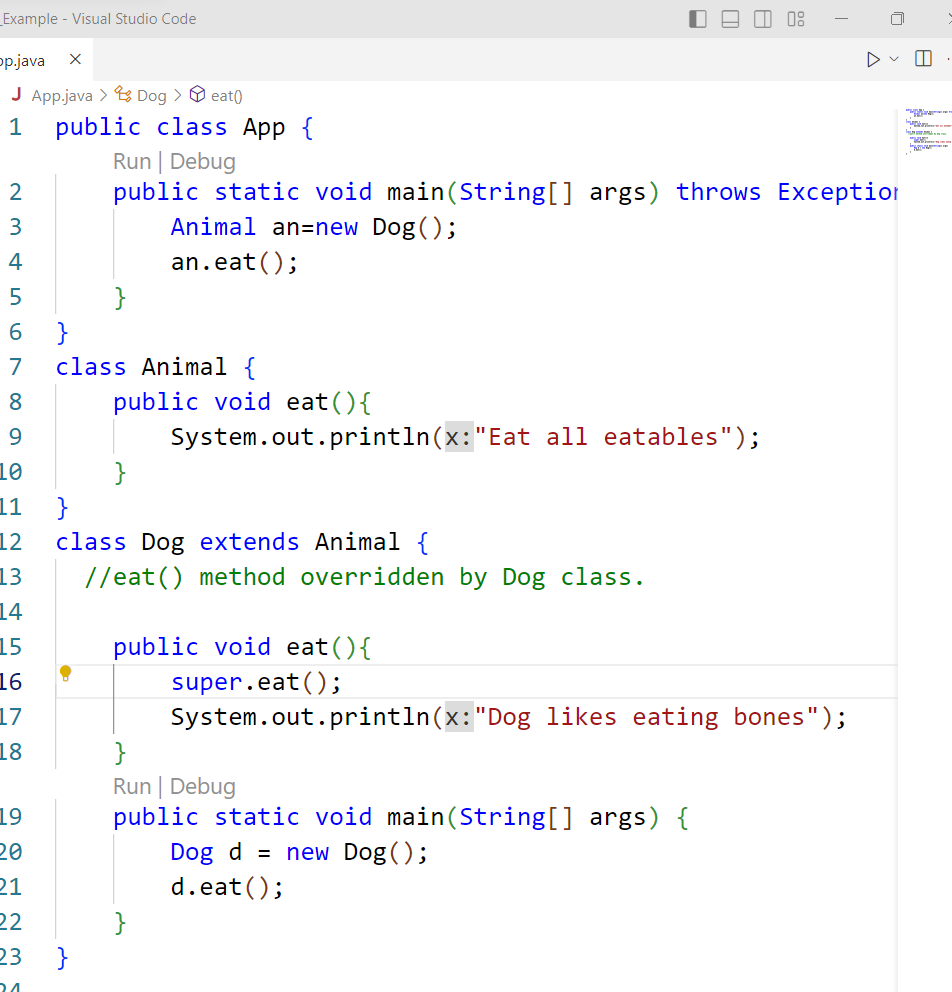
The purpose of overriding is achieved so that a sub class can provide its own implementation to a method that a superclass already provides.

**Why should we use method overloading in Java?**

Method overloading increases the **readability** of the program. This provides flexibility to programmers so that they can **call the same method for different types of data**. This makes the code look clean. This **reduces the execution time** because the binding is done in **compilation time** itself.



Example:

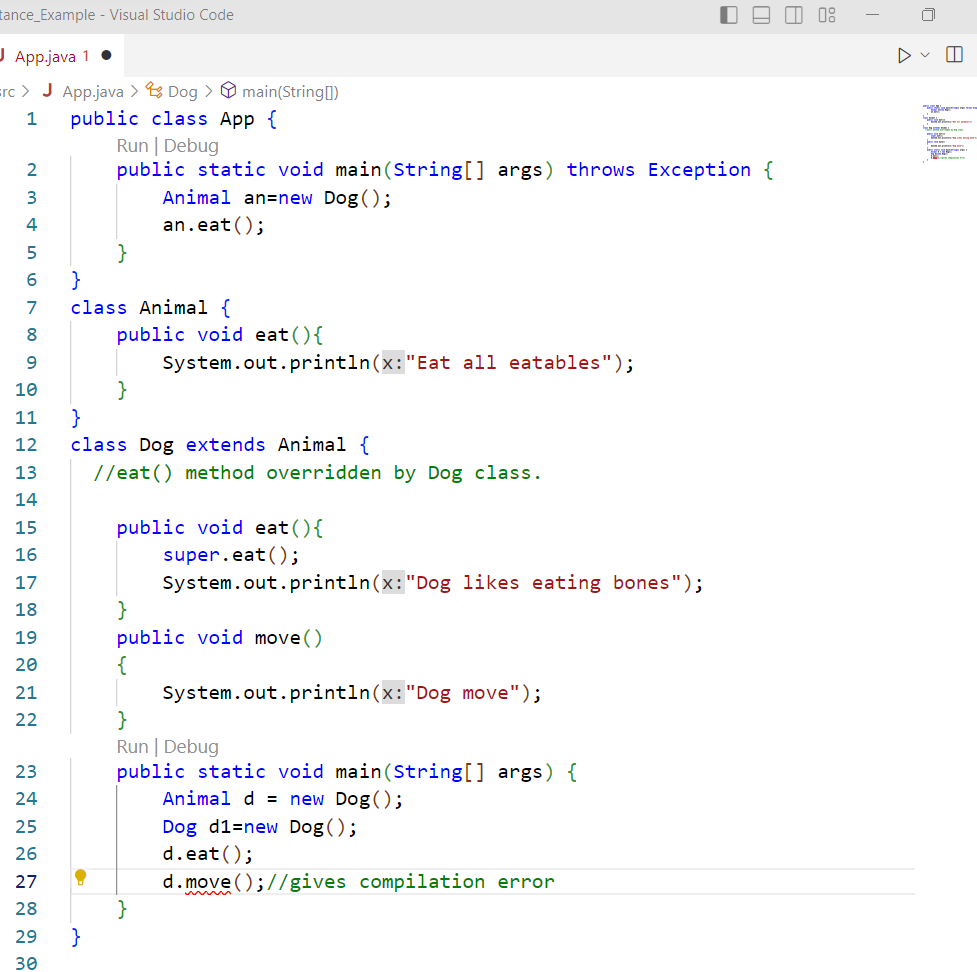


**Output:**Eat all eatables  
Dog likes eating bones

### Dynamic method dispatch in Overriding

To accomplish Java **runtime polymorphism**, overriding methods are utilized. The object that is used to call the method **determines the version of the method** that is being run. When a method is called with an object from a superclass, the **parent class’s version** is executed, but when a method is called with an object from a subclass, the **child class’s version** is executed.

That is, which version of the **overridden method** is performed is determined by the type of the **referenced object**. The practice of resolving overridden method calls at runtime is known as **dynamic method dispatch**. To understand this, see the example below:



**Benefits of method overriding in Java**

* It is used for the implementation of **runtime or dynamic polymorphism**.
* It is used to provide a **specific implementation or definition** of a method in a class, which is **already in an existence** in its superclass.
* It is also used to define **what** behavior a class can have and **how** that behavior will be implemented by the class that will inherits it.

**Rules for Method Overriding**

1. The method name should be common and the same as it is in the parent class.
2. The method signature (parameter list, return type) in the method must be the same as in the parent class.
3. There must be an inheritance connection between classes.
4. All the abstract methods in the parent class should be overridden in the child class.
5. If it declared the methods as **static or final**, then those methods cannot be overridden.