What is Inheritance?

Inheritance is an object-oriented programming feature where one class (called subclass or child class) inherits fields and methods from another class (called superclass or parent class).

Keyword used: extends

```
class SubclassName extends SuperclassName {
   // subclass body
}
```

Key Points:

- Promotes code reusability
- Every class has object as a superclass and by default every class extends object class.
- Supports method overriding and polymorphism
- A subclass cannot inherit private members of its superclass directly
- Java supports single, multilevel, and hierarchical inheritance
- Java does not support multiple inheritance with classes (only via interfaces)
- A class cannot inherit from itself.

A Superclass Variable Can Reference a Subclass Object

```
Superclass ref = new Subclass();
```

- The reference type determines what methods can be accessed.
- Object will determine which method will be called.
- Only the methods and variables in Superclass are accessible directly.
- Useful in runtime polymorphism.

A Sub-class Variable Cannot Reference a Superclass Object

```
Subclass ref = new Superclass();
```

Points related to super keyword:

- If super( ) is not used in subclass constructor, then the default constructor of each superclass will be executed.
- super() must be the first statement inside subclass constructor.
- super( ) always refers to the constructor in the closest superclass.

Use of super keyword

1. Calling superclass constructor.
2. Used to access the variable of super class whose name is same as the name of sub-class variable. Because "this" will by-default access the variable of sub-class.

```
class Parent{
    int a;
    int b;

    Parent(int a,int b){
        this.a = a;
        this.b = b;
    }
}

class Child extends Parent{
    int b;

    Child(int a,int b){
        super(a,10);
        this.b=b;

        System.out.println(this.b); //4
        System.out.println(super.b); //10
    }
}

public class Main
{
      public static void main(String[] args) {
          Child obj = new Child(3,4);
      }
}
```

3. Accessing superclass methods/fields that are hidden in the subclass

```
class Box {
    private double width, height, depth;

    Box(double w, double h, double d) {
        width = w;
```

```
        height = h;
        depth = d;
    }

    Box(Box ob) {
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }

    // Getter methods to access private variables
    double getWidth() { return width; }
    double getHeight() { return height; }
    double getDepth() { return depth; }
}

class BoxWeight extends Box {
    double weight;

    BoxWeight(double w,double h,double d,double weight){
        super(w,h,d);
        this.weight=weight;
    }

    BoxWeight(BoxWeight ob) {
        super(ob); // Calls Box constructor
        weight = ob.weight;
    }
}

public class Main
{
     public static void main(String[] args) {
         BoxWeight obj1 = new BoxWeight(2,3,4,5);
         BoxWeight obj2 = new BoxWeight(obj1);

         System.out.println("Width: " + obj2.getWidth()); //2
        System.out.println("Height: " + obj2.getHeight()); //3
        System.out.println("Depth: " + obj2.getDepth()); //4
        System.out.println("Weight: " + obj2.weight); //5
     }
}
```

- Types of Inheritance

Type	Supported?	Description

Single	✅	One subclass inherits one superclass
Multi-level	✅	Class inherits from a subclass of another class
Hierarchical	✅	Multiple sub-classes inherited from the same superclass
Multiple	❌	When one sub-class is inherited from more than one superclass. Not possible through classes in java but it is possible through interfaces. Also known as **diamond problem.**
Hybrid	❌	Combination of single and multiple as java does not support multiple inheritance therefore hybrid inheritance is also not supported by java. Achieved using interfaces.

🔹 **Constructor Chaining in Inheritance**

- Constructors are called in **top-down order** from superclass to subclass.
- If a superclass has parameterized constructors, subclass must explicitly call super() with arguments.
- If super() is not used, the **default constructor** of the superclass is invoked.

🔹 **What is Polymorphism?**

- Poly means **"Many"** and morphism means **"ways to represent single entity".**
- Polymorphism does not apply to instance variables.

🔶 **Types of Polymorphism**

1. Compile-Time Polymorphism (Static Binding)

- Achieved using **method overloading** and **operator overloading** (though operator overloading is not supported in Java).
- Resolved by the **compiler** at compile-time.

👉 **Method Overloading:**

- Same method name with different number of parameters or different types of parameters or different return type in the **same class**.

- In Java, it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call. In some cases, Java's automatic type conversions can play a role in overload resolution.

```
class Math {
    int add(int a, int b) {
        return a + b;
    }

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

---

2. Run-Time Polymorphism (Dynamic Binding)

- Achieved using **method overriding**.
- Resolved by the **JVM** at runtime.

👉 Method Overriding:

- Same method name and parameters in a **subclass**.
- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time. Let's begin by restating an important principle: a superclass reference variable can refer to a subclass object. When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.
- In other words, it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.

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

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

```
    }
}

public class Main {
    public static void main(String[] args) {

// here which method will be called depends on the object and
// this is also known as up-casting
        Animal obj = new Dog();  // Upcasting
        obj.sound();             // Output: Dog barks
    }
}
```

🎯 Key Concepts

✅ Upcasting:

- Assigning a child class object to a parent class reference.
  `Animal obj = new Dog();`

✅ Downcasting:

- Casting parent class reference back to child class (must be done manually and carefully).
  `Dog d = (Dog) obj;`

🔹 final Keyword in Inheritance

1. **final variable**: Cannot be reassigned
2. **final method**: Cannot be overridden
3. **final class**: Cannot be extended

🔹 Static Methods

- Static methods are inherited, but **cannot be overridden**
- The static method of the superclass will always be called regardless of the object reference because static method is not object dependent.

Data Hiding:

- Data hiding is the principle of hiding internal object details (like data members) from the outside world to prevent unauthorized access and modification.
- By making variables private, they cannot be accessed directly outside the class.

Encapsulation:

- It is process of wrapping data (variables) and methods (functions) together into a single unit (i.e., a class) and restricting direct access to some of the object's components.
- Encapsulation enables data hiding.

Abstraction	Encapsulation
Abstraction is a feature of OOPs that hides the **unnecessary** detail but shows the essential information.	Encapsulation is also a feature of OOPs. It hides the code and data into a **single** entity or unit so that the data can be protected from the outside world.
It solves an issue at the **design** level.	Encapsulation solves an issue at **implementation** level.
It focuses on the **external** lookout.	It focuses on **internal** working.
It can be implemented using **abstract classes** and **interfaces**.	It can be implemented by using the **access modifiers** (private, public, protected).
It is the process of **gaining** information.	It is the process of **containing** the information.
In abstraction, we use **abstract classes** and **interfaces** to hide the code complexities.	We use the **getters** and **setters** methods to hide the data.
The objects are **encapsulated** that helps to perform abstraction.	The object need not to **abstract** that result in encapsulation.