Inheritance in Java

Inheritance is an important pillar of OOP(Object-Oriented Programming). It is the mechanism in java by which one class is allowed to inherit the features(fields and methods) of another class. In Java, inheritance means creating new classes based on existing ones. A class that inherits from another class can reuse the methods and fields of that class. In addition,we can add new fields and methods to your current class as well.

The new class that is created is known as subclass (child or derived class) and the existing class from where the child class is derived is known as superclass (parent or base class).

The extends keyword is used to perform inheritance in Java. For example,

```
class Animal {
  // methods and fields
}

// use of extends keyword
// to perform inheritance
class Dog extends Animal {

  // methods and fields of Animal
  // methods and fields of Dog
}
```

In the above example, the Dog class is created by inheriting the methods and fields from the Animal class.
Here, Dog is the subclass and Animal is the superclass.

Example 1: Java Inheritance

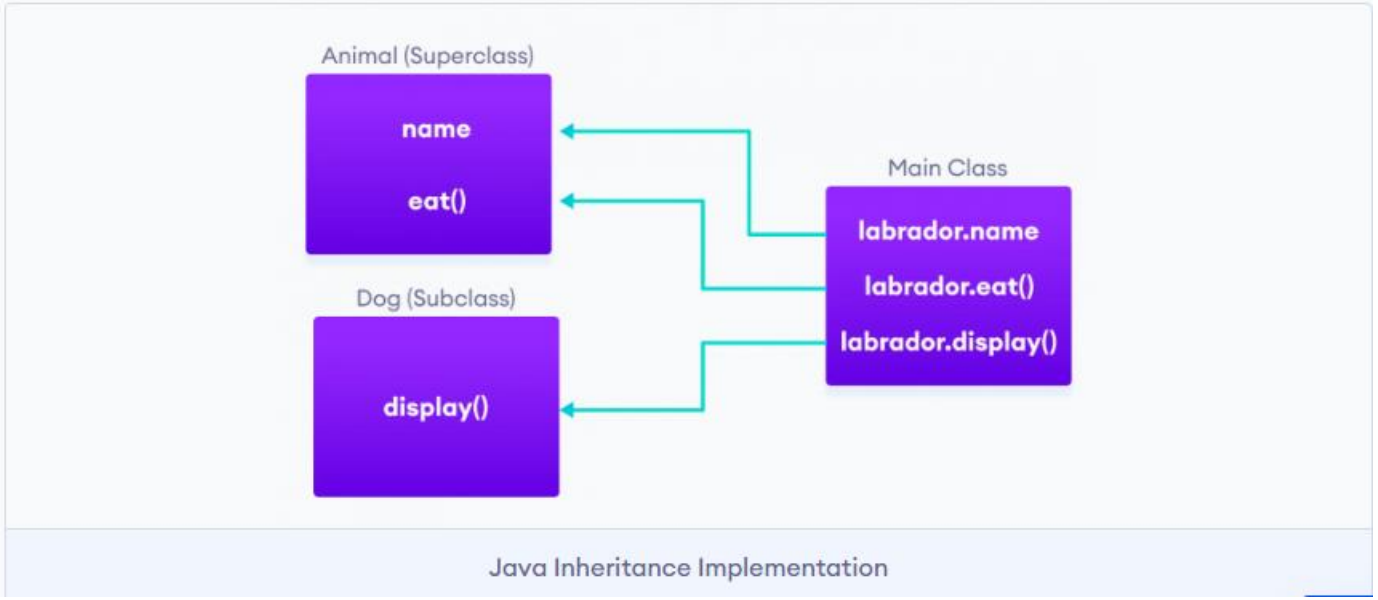
```
class Animal
 {

  // field and method of the parent class
  String name;
  public void eat() {
    System.out.println("I can eat");
  }
}

// inherit from Animal
class Dog extends Animal {

  // new method in subclass
  public void display() {
    System.out.println("My name is " + name);
  }
}

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

    // create an object of the subclass
    Dog labrador = new Dog();

    // access field of superclass
    labrador.name = "Rohu";
    labrador.display();

    // call method of superclass
    // using object of subclass
    labrador.eat();

  }
}
```

Output

```
My name is Rohu
I can eat
```

Here, `labrador` is an object of `Dog`. However, `name` and `eat()` are the members of the `Animal` class.

Since `Dog` inherits the field and method from `Animal`, we are able to access the field and method using the object of the `Dog`.



Java Inheritance Implementation

is-a relationship

In Java, inheritance is an **is-a** relationship.

That is, we use inheritance only if there exists an is-a relationship between two classes. For example,

- **Car** is a **Vehicle**
- **Orange** is a **Fruit**
- **Surgeon** is a **Doctor**
- **Dog** is an **Animal**

Here, **Car** can inherit from **Vehicle**, **Orange** can inherit from **Fruit**, and so on.

Method Overriding in Java Inheritance

In **Example 1**, we see the object of the subclass can access the method of the superclass.

However, if the same method is present in both the superclass and subclass, what will happen?

In this case, the method in the subclass overrides the method in the superclass. This concept is known as method overriding in Java.

Note: We have used the @Override annotation to tell the compiler that we are overriding a method. However, the annotation is not mandatory. To learn more, visit Java Annotations.

Example 2: Method overriding in Java Inheritance

```
class Animal {

  // method in the superclass
  public void eat() {
    System.out.println("I can eat");
  }
}

// Dog inherits Animal
class Dog extends Animal {

  // overriding the eat() method
  @Override
  public void eat() {
    System.out.println("I eat dog food");
  }

  // new method in subclass
  public void bark() {
    System.out.println("I can bark");
  }
}

class Main2 {
  public static void main(String[] args) {

    // create an object of the subclass
    Dog labrador = new Dog();

    // call the eat() method
    labrador.eat();
    labrador.bark();
  }
}
```

super Keyword in Java Inheritance

Previously we saw that the same method in the subclass overrides the method in superclass.

In such a situation, the super keyword is used to call the method of the parent class from the method of the child class.

Example 3: super Keyword in Inheritance

```
class Animal {

  // method in the superclass
  public void eat() {
    System.out.println("I can eat");
  }
}

// Dog inherits Animal
class Dog extends Animal {

  // overriding the eat() method
  @Override
  public void eat() {

    // call method of superclass
    super.eat();
    System.out.println("I eat dog food");
  }

  // new method in subclass
  public void bark() {
    System.out.println("I can bark");
  }
}

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

    // create an object of the subclass
    Dog labrador = new Dog();

    // call the eat() method
    labrador.eat();
    labrador.bark();
  }
}
```

Output

```
I can eat
I eat dog food
I can bark
```

super.eat();
Here, the super keyword is used to call the eat() method present in the superclass.

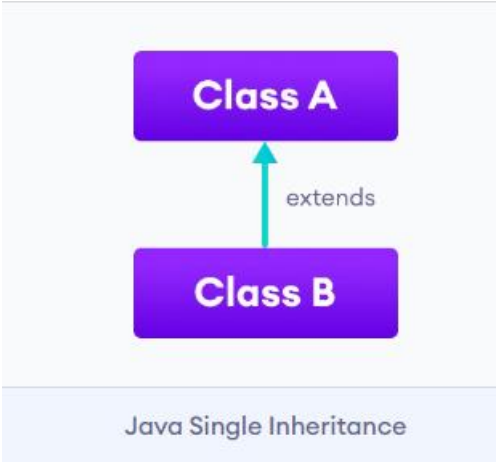
We can also use the super keyword to call the constructor of the superclass from the constructor of the subclass.

Types of inheritance

There are five types of inheritance.

1. Single Inheritance

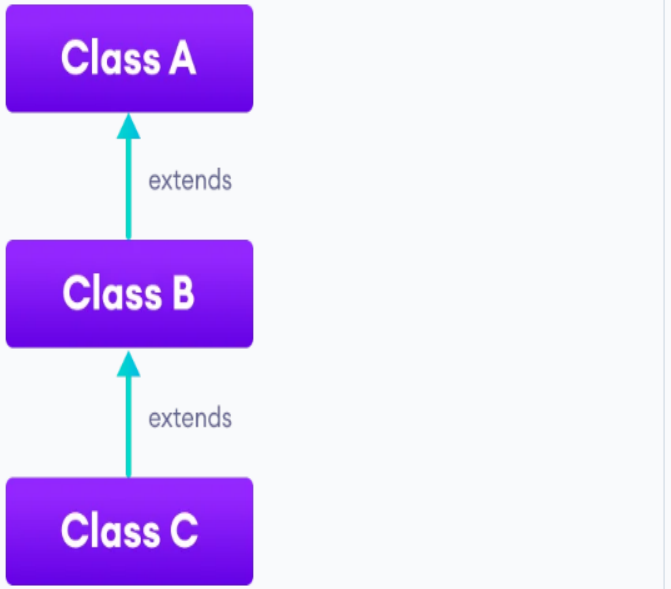
In single inheritance, a single subclass extends from a single superclass. For example,



Java Single Inheritance

2. Multilevel Inheritance

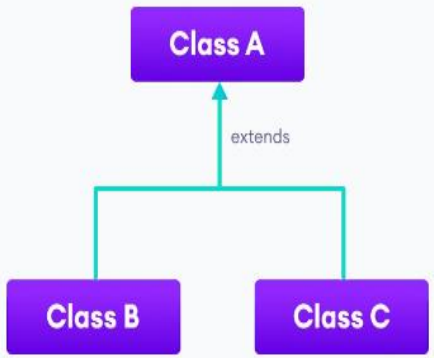
In multilevel inheritance, a subclass extends from a superclass and then the same subclass acts as a superclass for another class. For example,



Java Multilevel Inheritance

3. Hierarchical Inheritance

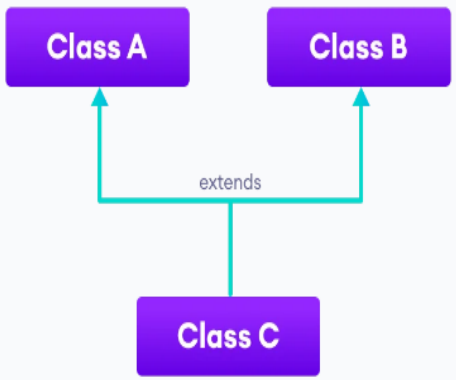
In hierarchical inheritance, multiple subclasses extend from a single superclass. For example,



Java Hierarchical Inheritance

4. Multiple Inheritance

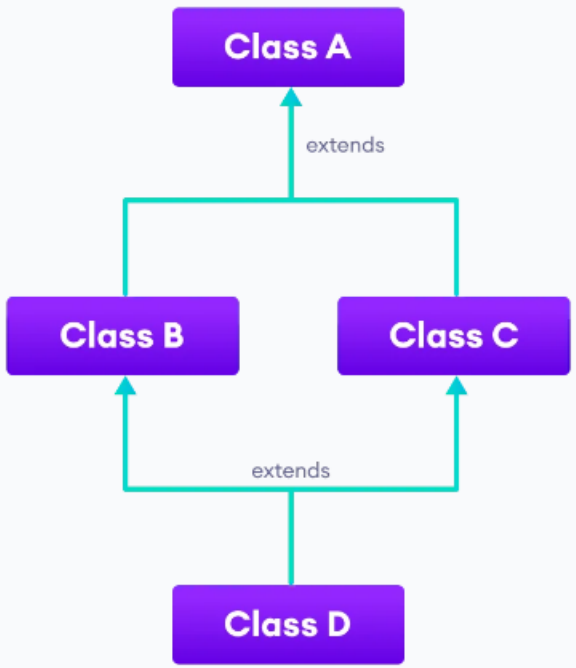
In multiple inheritance, a single subclass extends from multiple superclasses. For example,



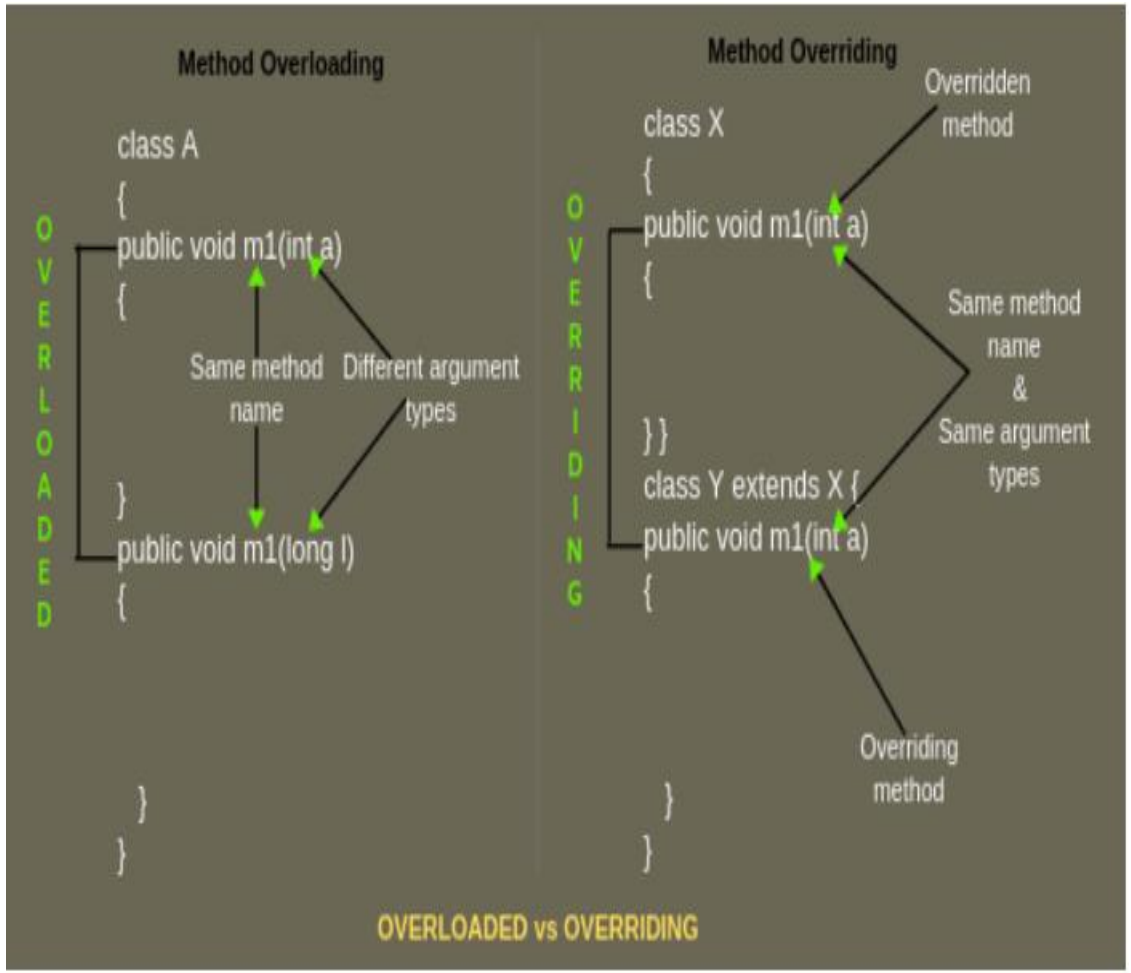
Java Multiple Inheritance

5. Hybrid Inheritance

Hybrid inheritance is a combination of two or more types of inheritance. For example,



Java Hybrid Inheritance


Method Overloading
class A
{
public void m1(int a)
{
Same method name
Different argument types
}
public void m1(long l)
{
}
}
OVERLOADED
Method Overriding
Overridden method
class X
{
public void m1(int a)
{
}}
class Y extends X {
public void m1(int a)
{
Same method name & Same argument types
Overriding method
}
}
OVERRIDING
OVERLOADED vs OVERRIDING

Polymorphism in Java

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

Real-life Illustration: Polymorphism

A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person possesses different behavior in different situations. This is called polymorphism. Polymorphism is considered one of the important features of Object-Oriented Programming. Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations. The word “poly” means many and “morphs” means forms, So it means many forms.

In Java polymorphism is mainly divided into two types:

- Compile-time Polymorphism
- Runtime Polymorphism

Type 1: Compile-time polymorphism

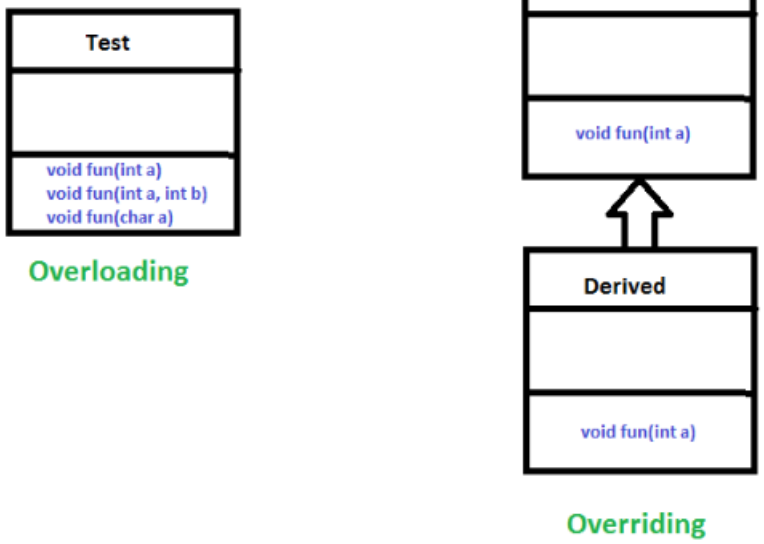
It is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading.

In Java polymorphism is mainly divided into two types:

- Compile-time Polymorphism
- Runtime Polymorphism

Type 1: Compile-time polymorphism

It is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading.



Method Overloading: When there are multiple functions with the same name but different parameters then these functions are said to be **overloaded**. Functions can be overloaded by change in the number of arguments or/and a change in the type of arguments.

```
// Java Program for Method overloading
// By using Different Types of Arguments

// Class 1
// Helper class
class Helper {

        // Method with 2 integer parameters
        static int Multiply(int a, int b)
        {

                // Returns product of integer numbers
                return a * b;
        }

        // Method 2
        // With same name but with 2 double parameters
        static double Multiply(double a, double b)
        {

                // Returns product of double numbers
                return a * b;
        }
}
// Class 2
// Main class
class GFG {

        // Main driver method
        public static void main(String[] args)
        {
                // Calling method by passing
                // input as in arguments
                System.out.println(Helper.Multiply(2, 4));
                System.out.println(Helper.Multiply(5.5, 6.3));
        }
}
```

What is Constructor and Destructor in Java?

A constructor is used to initialize a variable that means it allocates memory for the same A constructor is nothing but automatic initialization of the object. Whenever the program creates an object at that time constructor, is gets called automatically. You don’t need to call this method explicitly. Destructor is used to free that memory allocated during initialization. Generally, in java, we don’t need to call the destructor explicitly. Java has a feature of automatic garbage collection.

Why do we Need a Constructor and Destructor in Java?

Constructor and destructor are mostly used to handle memory allocation and de-allocation efficiently. Constructor and destructor do a very important role in any programming language of initializing and destroying it after use to free up the memory space.

How Constructor and Destructor Works in Java

A constructor is just a method in java, which has the same name as the class name. The constructor method does not have any return type to it.
Look at the following example for more clarity:

How to create Constructors and Destructors in java?

```
class Employee {
Employee() { //This is constructor. It has same name as class name.
System.out.println("This is the default constructor");
}
}
```

Types of Constructor

There are two types of constructors; depending upon the type, we can add and remove variables.

Default Constructor

Parameterized Constructor

With this, we are also going to see constructor overloading.

1. Default Constructor

1. Default Constructor

This is the one type of constructor. By default, without any parameters, this constructor takes place. This constructor does not have any parameters in it.

Example:

```
class Abc{
Abc(){
System.out.println("This is the example of  default constructor.");
}
}
```

2. Parameterized Constructor

As the name suggests, the parameterized constructor has some parameters or arguments at the time of initializing the object.

Example:

```
import java.util.*;
class Square
{ int width,height;
Square( int a , int b)
{ width = a; height = b;
 }
int area()
{ return width * height;
 }
} class Cal
{ public static void main(String[] args)
{
{
Square s1 = new Square(10,20); int
area_of_sqaure = s1.area();
System.out.println("The area of square is:" +
area_of_sqaure);
 } } }
```