Inheritance

What is Inheritance in Java?

Inheritance in Java is a concept that acquires the properties from one class to other classes; for example, the relationship between father and son. Inheritance in Java is a process of acquiring all the behaviours of a parent object.

The concept of inheritance in Java is that new classes can be constructed on top of older ones. You can use the parent class's methods and properties when you inherit from an existing class. You can also add additional fields and methods to your existing class.
The parent-child relationship, also known as the IS-A relationship, is represented by inheritance.

To explain further,

One object can acquire all of a parent object's properties and actions through the technique of inheritance in Java Programming. It is a crucial component of OOPs (Object Oriented programming system).

In Java, the idea of inheritance means that new classes can be built on top of existing ones. When you derive from an existing class, you can use its methods and properties. To your current class, you may also add new fields and methods.

What is inheritance and example?

A new item can inherit the traits of an older object through the process of inheritance. As an illustration, think of the class "human." You might want to add other human characteristics in your class, such as height, weight, and so on. Therefore, one approach is to redefine each of those attributes in your class. Though not a good practise, it might be a useful approach to learn object-oriented programming. Inheriting all of those properties from one particular class is the best way to go about it. All of the attributes of class "human" (or "parent") may be inherited by class "child." The term "inheritance" in object-oriented programming refers to this.

```
Class PetAnimal {



 // field and method of the parent class

 String name;

 public void eat() {

   System.out.println("I can eat");

 }

}



// inherit from PetAnimal

class Dog extends PetAnimal {
```

```
 // 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();



}
```

```
}
```

In Java, a class can inherit attributes and methods from another class. The class that inherits the properties is known as the sub-class or the child class. The class from which the properties are inherited is known as the superclass or the parent class.

In Inheritance, the properties of the base class are acquired by the derived classes.

Inheritance Syntax in Java

```
class derived_class extends base_class

{

  //methods

  //fields
```

```
}
```

General format for Inheritance

```
class superclass

{

// superclass data variables

// superclass member functions

}

class subclass extends superclass

{

// subclass data variables

// subclass member functions
```

```
}
```

Inheritance uses the “extends” keyword to create a derived class by reusing the base class code.

Extends keyword in Java

The extended keyword extends a class and is an indicator that a class is being inherited by another class. When you say class B extends a class A, it means that class B is inheriting the properties(methods, attributes) from class A. Here, class A is the superclass or parent class and class B is the subclass or child class.

Inheritance Program Example

```
class Base

{

public void M1()
```

```
{

System.out.println(“ Base Class Method ”);

}

}

class Derived extends Base

{

public void M2()

{

System.out.printIn(“ Derived Class Methods “);

}

}
```

```
class Test

{

public static void main(String[] args)

{

Derived d = new Derived(); // creating object

d.M1(); // print Base Class Method

d.M2(); // print Derived Class Method

}

}
```

Power of Inheritance: Key Benefits

- **Code Reusability:** Inheritance shines in its ability to promote code reuse. You can inherit the common functionalities of a

superclass and extend them in subclasses without duplicating code. This is particularly beneficial when dealing with related concepts that share a core set of behaviors.

- **Improved Maintainability:** When a common functionality needs modification, you only need to update the superclass code. These changes automatically propagate to all subclasses that inherit from it, streamlining maintenance efforts. It is difficult to maintain a single source for shared behaviors. So, it is best to modify the same code repeatedly across multiple classes.
- **Promoting the IS-A Relationship:** Inheritance establishes a clear and intuitive “IS-A” relationship between classes. A subclass inherits properties and behaviors from its superclass. This essentially states that the subclass “is a type of” the superclass.

For instance, a ‘Dog’ class inherits from an Animal class, demonstrating that a dog “is a type of” animal. This relationship fosters a hierarchical understanding of your class structure.

Essential Concepts for Mastering Inheritance

- **Superclasses and Subclasses:** Grasp the fundamental roles of superclasses and subclasses. A superclass (parent class) serves as the source of inherited properties and methods. A subclass (child class) inherits from the superclass and potentially adds its own unique features.
- **The 'extends' Keyword:** The extends keyword acts as the bridge between classes in inheritance. It's used in the subclass declaration to establish the inheritance relationship. The syntax for a subclass typically looks like:

```
class SubclassName extends SuperclassName {

// Subclass members (fields and methods)

}
```

- **Method Overriding:** Inheritance allows subclasses to redefine (override) methods inherited from their superclass. This empowers you to provide a specific implementation for a method

within the subclass context. Method overriding is a cornerstone of polymorphism. It enables you to write code that operates on different objects of related but distinct types at runtime.

- **The super Keyword:** The 'super' keyword plays a crucial role in inheritance scenarios. It's used within a subclass to refer to the superclass's members (variables and methods). Here are some of its key applications:

1. **Calling superclass constructors:** You can use super() to invoke the superclass constructor from within the subclass constructor.
2. **Accessing superclass methods:** The 'super' keyword allows you to access methods defined in the superclass, even if they are hidden by methods with the same name in the subclass.
3. **Accessing superclass variables:** Similar to methods, you can use 'super' to access variables defined in the superclass.

Inheritance Types in Java: Building Class Hierarchies

- **Single Inheritance:** This is the most fundamental and widely used type of inheritance. A subclass inherits from only one

superclass, establishing a clear and direct lineage. Single inheritance promotes code clarity and maintainability.

```
// Single Inheritance Example: Vehicle and Car

class Vehicle {

 public void move() {

    System.out.println("Vehicle is moving");

 }

}

class Car extends Vehicle {

 public void openTrunk() {

    System.out.println("Car trunk opened");
```

```
 }

}

public class InheritanceDemo {

 public static void main(String[] args) {

    Car car = new Car();

    car.move(); // Inherited from Vehicle

    car.openTrunk(); // Specific to Car

 }

}
```

- **Multi-Level Inheritance:** In multi-level inheritance, a subclass inherits from another subclass. This, in turn, inherits from another superclass, forming a chain of inheritance. This allows

you to create specialized classes that inherit properties and behaviors from multiple levels of abstraction.

```
// Multi-Level Inheritance Example: Animal, Mammal, and Dog

class Animal {

 public void eat() {

    System.out.println("Animal is eating");

 }

}

class Mammal extends Animal {

 public void giveBirth() {

    System.out.println("Mammal giving birth");
```

```
 }

}

class Dog extends Mammal {

 public void bark() {

    System.out.println("Dog is barking");

 }

}

public class InheritanceDemo {

 public static void main(String[] args) {

    Dog dog = new Dog();

    dog.eat(); // Inherited from Animal
```

```
    dog.giveBirth(); // Inherited from Mammal

    dog.bark(); // Specific to Dog

  }

}
```

- **Hierarchical Inheritance:** Hierarchical inheritance involves multiple subclasses inheriting from a single superclass. This creates a hierarchy of classes, where the superclass defines common functionalities inherited by the specialized subclasses.

```
// Hierarchical Inheritance Example: Shape with Square and Circle

class Shape {

 public void draw() {

    System.out.println("Drawing a shape");
```

```
 }

}

class Square extends Shape {

 public void drawSquare() {

    System.out.println("Drawing a square");

 }

}

class Circle extends Shape {

 public void drawCircle() {
```

Abstract Classes

Abstract classes are a special type of class that cannot be directly instantiated (you cannot create objects of an abstract class). They serve as

blueprints for subclasses and define a contract that subclasses must adhere to. Abstract classes can contain:

- **Abstract methods:** These methods lack an implementation within the abstract class itself. Subclasses are mandated to provide their own implementation for these methods. This enforces a common behavior across subclasses.
- **Concrete methods:** These methods have a defined implementation within the abstract class and can be inherited by subclasses.

Why use abstract classes?

- **Enforce common behavior:** Abstract methods ensure that subclasses implement a specific functionality.
- **Promote code reuse:** Concrete methods in abstract classes provide reusable code for subclasses.

- **Model real-world concepts:** Abstract classes can represent abstract concepts that cannot be directly instantiated (e.g., an abstract Shape class).

```
// Abstract Shape Example with an abstract method

abstract class Shape {

 public abstract double getArea(); // Abstract method
- subclasses must implement

 public void draw() {

   System.out.println("Drawing a shape");

 } // Concrete method - can be used by subclasses

}

class Square extends Shape {
```

```
 private double sideLength;

 public Square(double sideLength) {

   this.sideLength = sideLength;

 }

 @Override

 public double getArea() {

   return sideLength *
```

Why use Inheritance in Java?

The main advantage of inheritance is code reusability and also method overriding (runtime polymorphism).

Inheritance is also known as the IS-A relationship.

Terms Used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

Types of Inheritance in Java

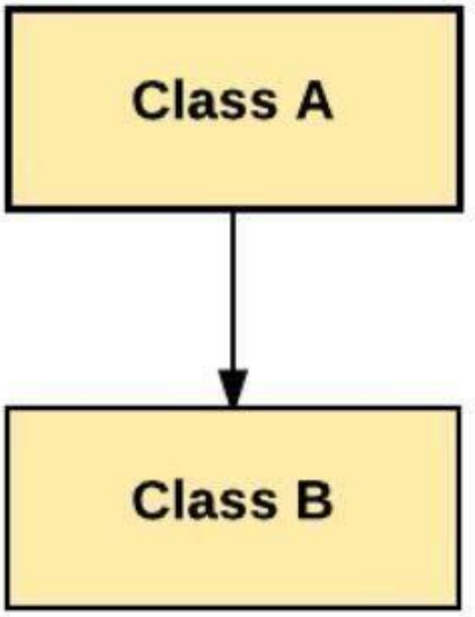
The different 6 types of Inheritance in java are:

- **Single inheritance.**

- **Multi-level inheritance.**
- **Multiple inheritance.**
- **Multipath inheritance.**
- **Hierarchical Inheritance.**
- **Hybrid Inheritance.**

Single Inheritance

As the title indicates, just one class is subject to this kind of inheritance. The parent class gives rise to just one child class. The attributes in this sort of inheritance are only descended from one parent class, at most. Code reuse and the implementation of new features are made easier because the attributes are descended from a single base class. Below is a flowchart of a single inheritance:



In Inheritance, we can access superclass methods and variables. We can also access subclass methods and variables through subclass objects only. We have to take care of superclass and subclass methods, and variable names shouldn't conflict.

Program Example:

```
class A

{
```

```
int a, b;

void display()

{

System.out.println("Inside class A values ="+a+" "+b);

}

}

class B extends A

{

int c;

void show()

{
```

```
System.out.println("Inside Class B values="+a+" "+b+"
"+c);  }

}

class SingleInheritance

{

public static void main(String args[])

{

B obj = new B(); //derived class object

obj.a=10;

obj.b=20;

obj.c=30;
```

```
obj.display();

obj.show();

}

}
```

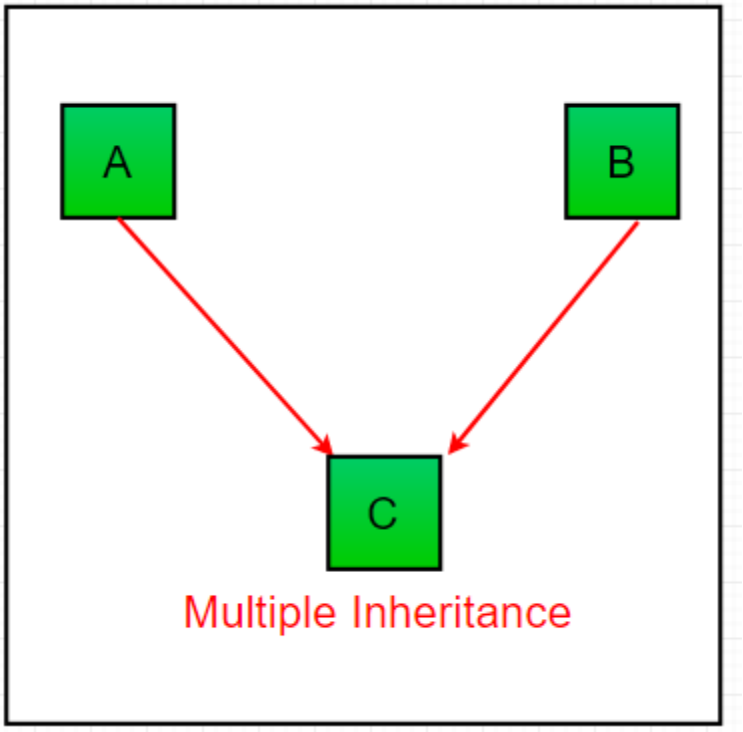
Multiple Inheritance in Java

Defining derived class from numerous base classes is known as ‘Multiple Inheritance’. In this case, there is more than one superclass, and there can be one or more subclasses.

Multiple inheritances are available in object-oriented programming with C++, but it is not available in Java.

Java developers want to use multiple inheritances in some cases. Fortunately, Java developers have interface concepts expecting the developers to achieve multiple inheritances by using multiple interfaces.

A subclass may inherit features from many parent classes under the concept of multiple inheritance. Contrary to popular belief, multiple inheritances are not the same as multi-level inheritance because the newly derived class in multiple inheritances may have more than one superclass. There are no limitations, and this newly derived class is free to inherit the features from the superclasses it has inherited from. Interfaces in Java can be used to achieve multiple inheritances.



Ex: class Myclass implements interface1, interface2,....

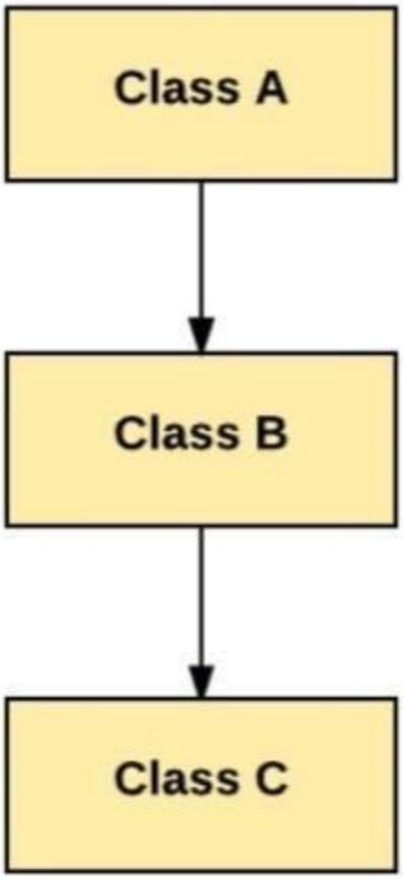
Multi-Level Inheritance in Java

In Multi-Level Inheritance in Java, a class extends to another class that is already extended from another class. For example, if there is a class A that extends class B and class B extends from another class C, then this scenario is known to follow Multi-level Inheritance.

We can take an example of three classes, class Vehicle, class Car, and class SUV. Here, the class Vehicle is the grandfather class. The class Car extends class Vehicle and the class SUV extends class Car.

At least two classes, if not more, are involved in the multi-level inheritance. A subclass that has just been formed becomes the base class for a new class, and one class inherits the features from its parent class.

As the name implies, numerous base classes are involved in multi-level inheritance. As the newly derived class from the parent class becomes the base class for another newly derived class, the inherited features in multilevel inheritance in Java likewise come from several base classes.



Multi-level inheritance:

```
class Electronics {

public Electronics(){

System.out.println("Class Electronics");
```

```
}

public void deviceType() {

System.out.println("Device Type: Electronics");

}

}

class Grinder extends Electronics {

public Grinder() {

System.out.println("Class Grinder");

}

public void category() {

System.out.println("Category - Grinder");
```

```
}

}

class WetGrinder extends Grinder {

public WetGrinder() {

System.out.println("Class WetGrinder");

}

public void grinding_tech() {

System.out.println("Grinding Technology- WetGrinder");

}

}

public class Tester {
```

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

WetGrinder wt= new WetGrinder();

wt.deviceType();

wt.category();

wt.grinding_tech();

}

}
```

Output:

```
Class Electronics

Class Grinder

Class WetGrinder

Device Type: Electronics

Category: Grinder

Grinding Technology: WetGrinder
```

Why Multiple Inheritance is not supported in Java?

Let's consider a case in Inheritance. Consider a class A, class B and class C. Now, let class C extend class A and class B. Now, consider a method read() in both class A and class B. The method read() in class A is different from the method read() in class B. But, while inheritance happens, the compiler has difficulty in deciding on which read() to inherit. So, in order to avoid such kind of ambiguity, multiple inheritance is not supported in Java.

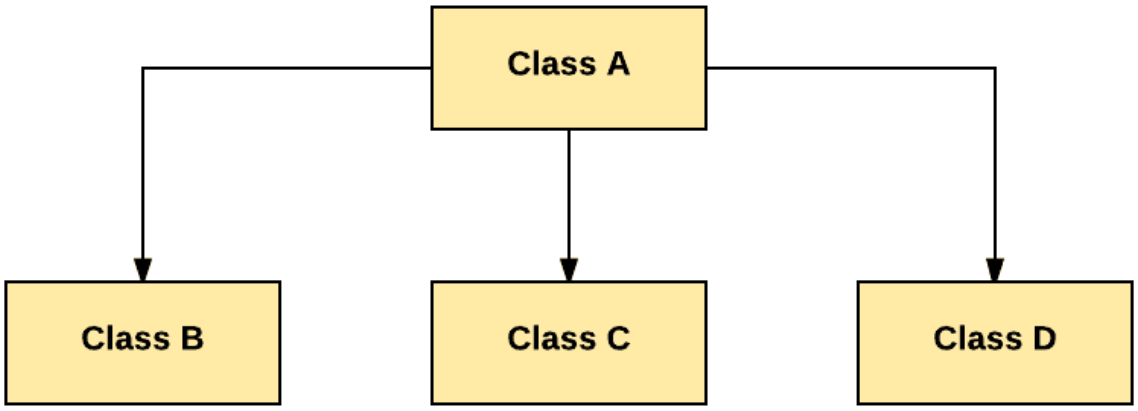
Hierarchical Inheritance in Java

Hierarchical inheritance in java is the sort of inheritance when numerous subclasses derive from a single class.

A mixture of various inheritance types is called hierarchical inheritance.

Due to the fact that numerous classes are descended from a single superclass, it differs from multilevel inheritance. These recently created classes take after this one superclass, inheriting its features, methods, etc. This procedure makes dynamic polymorphism and code reuse possible (method overriding).

For example, consider a parent class Car. Now, consider child classes Audi, BMW and Mercedes. In Hierarchical Inheritance in Java, class Audi, class BMW and class Mercedes, all these three extend class Car.




```
public class ClassH1

{

   public void dispH1()

   {
```

```
        System.out.println("disp() method of ClassH1");

    }

}

public class ClassH2 extends ClassH1

{

   public void dispH2()

    {

        System.out.println("disp() method of ClassH2");

    }

}

public class ClassH3 extends ClassH1
```

```
{

    public void dispH3()

    {

        System.out.println("disp() method of ClassH3");

    }

}

public class ClassH4 extends ClassH1

{

    public void dispH4()

    {

        System.out.println("disp() method of ClassH4");
```

```
    }

}

public class HierarchicalInheritanceTest

{

   public static void main(String args[])

    {

        //Assigning ClassH2 object to ClassH2 reference

        ClassH2 h2 = new ClassH2();

        //call dispH2() method of ClassH2

        h2.dispH2();

        //call dispH1() method of ClassH1
```

```
        h2.dispH1();

        //Assigning ClassH3 object to ClassH3 reference

        ClassH3 h3 = new ClassH3();

        //call dispH3() method of ClassH3

        h3.dispH3();

        //call dispH1() method of ClassH1

        h3.dispH1();

        //Assigning ClassH4 object to ClassH4 reference
```

```
        ClassH4 h4 = new ClassH4();

        //call dispH4() method of ClassH4

        h4.dispH4();

        //call dispH1() method of ClassH1

        h4.dispH1();

    }

}
```

Output:

```
disp() method of ClassH2
disp() method of ClassH1
disp() method of ClassH3
disp() method of ClassH1
disp() method of ClassH4
disp() method of ClassH1
```

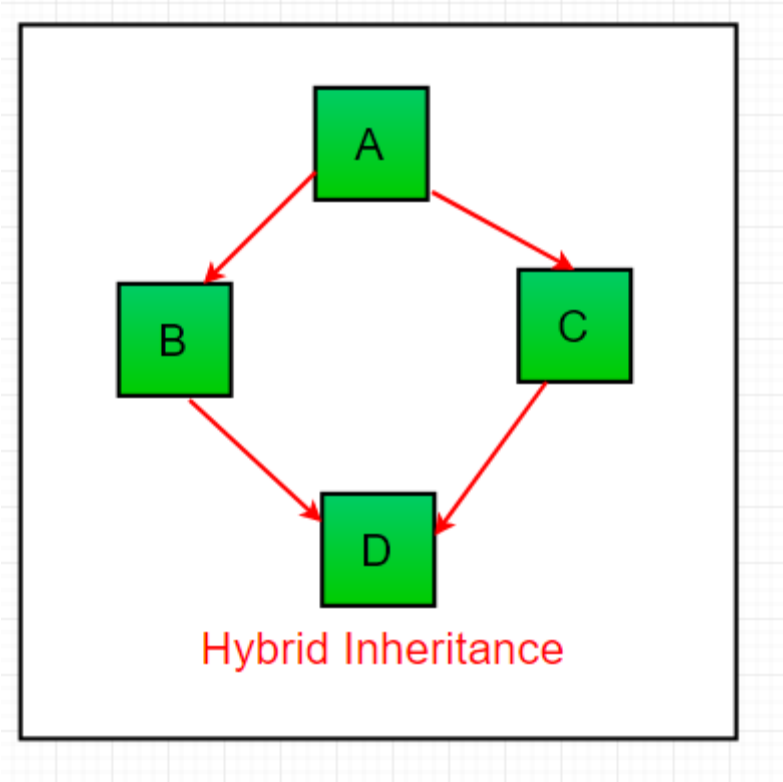
Hybrid Inheritance in Java

Hybrid Inheritance in Java is a combination of inheritance. In this type of Inheritance, more than one kind of inheritance is observed. For example, if we have class A and class B that extend class C and then there is another class D that extends class A, then this type of Inheritance is known as Hybrid Inheritance.

Why? Because we clearly observe that there is two kinds of inheritance here- Hierarchical and Single Inheritance.

A hybrid inheritance combines more than two inheritance types, such as multiple and single. Interfaces are the sole means through which it is possible because Java does not enable multiple inheritance. In essence, it combines straightforward, numerous, and hierarchical inheritances.

In the diagram shown below, we see another example of Hybrid Inheritance.



Inheritance Program in Java

1. If we want to call methods and variables using the Parent class object, you will get an error.

Inheritance Example in Java-1:

```
class Parent

{
```

```
public void M1()

{

System.out.println("Parent Class Method");

}

}

class Child extends Parent

{

public void M2()

{

System.out.println("Child Class Method");

}
```

```
}

class Inh_In_Java

{

public static void main(String[] args)

{

 Parent p=new Parent();

 p.M1();

 p.M2(); // error-wrong way to call derived class
method

}

}
```

2. Creating objects will be very important

Parent p=new Child(); // will not work

Inheritance Example in Java-2:

```
class Parent

{

public void M1()

{

System.out.println("Parent Class Method");

}

}

class Child extends Parent
```

```
{

public void M2()

{

System.out.println("Child Class Method");

}

}

class Inh_In_Java

{

public static void main(String[] args)

{

 Parent p=new Child();
```

```
 p.M1();

 p.M2();

}

}
```

3. Child p=new Parent();

This implementation will not work because of incompatible types: It is not possible to convert a Parent to a Child

Inheritance Example in Java-3:

```
class Parent

{

public void M1()
```

```
{

System.out.println("Parent Class Method");

}

}

class Child extends Parent

{

public void M2()

{

System.out.println("Child Class Method");

}

}
```

```
class Inh_In_Java

{

public static void main(String[] args)

{

Child p=new Parent();

p.M1();

p.M2();

}

}
```

4. From the above three examples, we understand that inheritance will be useful when derived class objects call base class(parent class or superclass) methods and variables. It will not throw an error.

Inheritance Example in Java-4:

```
class Parent

{

public void M1()

{

System.out.println("Parent Class Method");

}

}

class Child extends Parent

{

public void M2()
```

```
{

System.out.println("Child Class Method");

}

}

class Inh_In_Java

{

public static void main(String[] args)

{

Child p=new Child();

p.M1();

p.M2();
```

```
}
```

```
}
```

Method Overriding in Java

If the child class has the same method in its implementation as that of its parent's class, then the concept of method overriding comes into play.

In method overriding, the child class has the same method as that of the parent class. The main use of this is to achieve runtime polymorphism.

Method Overriding is used to provide specific implementation of a particular method which was provided by the parent's class.

Rules for Method overriding are:

- Methods must share the same name in child and parent class.
- It must have the same parameter as in the superclass.
- There must be an IS-A type of Inheritance.

IS-A Relationship in java

A combination of two or more classes in java is known as a relationship.

What is the use case of inheritance? Well, the answer is that wherever we come across an IS-A relationship between objects, we can use inheritance.

Existing Class	Derived Class
Shape	Square
Programming Language	Java
Vehicle	Car

In Java, we have two types of relationships:

1. **Is-A relationship**
2. **Has-A relationship**

Is-A relationship

IS-A Relationship is completed related to inheritance. For example — a carrot is a vegetable; a fan is a device.

This relationship can be achieved by:

- Usingextend Keyword
- To avoid code redundancy.

Super keyword in Java

Super keyword usage in inheritance, always refers to its immediate as an object.

There are three usages of super keyword in Java:

1. We can invoke the superclass variables.

2. We can invoke the superclass methods.

3. We can invoke the superclass constructor.

Example for Super Keyword in Java:

```
class Superclass

{

int i =20;

void display()

{

System.out.println("Superclass display method");

}

}
```

```
class Subclass extends Superclass

{

int i = 100;

void display()

{

super.display();

System.out.println("Subclass display method");

System.out.println(" i value ="+i);

System.out.println("superclass i value ="+super.i);

}

}
```

```
class SuperUse

{

public static void main(String args[])

{

Subclass obj = new Subclass();

obj.display();

}

}
```

Protected member in Inheritance

A class's private members are inaccessible from outside the class. The private members are only directly accessible to methods of that class. However, as was previously said, it could occasionally be required for a

subclass to have access to a superclass's private member. Anybody can access a private member if you make it public. In order to block direct access to a member of a superclass outside the class while yet allowing it to be used in a subclass, you must designate that member protected.

If you liked the article, show your support with a ***clap*** 👏 *and* ***follow*** *me! Feel free to* ***highlight*** *your favorite parts too.* ***Your engagement keeps me inspired!***