Inheritance(IS-A)

Why multiple inheritance is not supported in java?

- To reduce the complexity and simplify the language, multiple inheritances is not supported in java.
- Consider a scenario where **A**, **B**, and **C** are three classes. The **C** class inherits the **A** and **B** classes. If **A** and **B** classes have the same method and you call it from child class object, there will be ambiguity to call a method of **A** or **B** class.
- Since compile-time errors are better than runtime errors, java renders compile-time error if you inherit 2 classes. So whether you have the same method or different, there will be a compile-time error now.

```
class A{
    void msg(){System.out.println("Hello");}
}
class B{
    void msg(){System.out.println("Welcome");}
}
class C extends A,B{//suppose if it were

     Public Static void main(String args[]){
          C obj=new C();
```

```
          obj.msg();//Now which msg() method would be invoked?
     }
}

 Compile Time Error
```

Java IS-A type of Relationship

- IS-A is a way of saying: This object is a type of that object. Let us see how the extends keyword is used to achieve inheritance.

```
public class SolarSystem {
}
public class Earth extends SolarSystem {
}
public class Mars extends SolarSystem {
}
public class Moon extends Earth {
}
```

Now, based on the above example, in Object-Oriented terms, the following are true:-

- SolarSystem is the superclass of Earth class.
- SolarSystem is the superclass of Mars class.
- Earth and Mars are subclasses of SolarSystem class.

- Moon is the subclass of both Earth and SolarSystem classes.

```
class SolarSystem {
}
class Earth extends SolarSystem {
}
class Mars extends SolarSystem {
}
public class Moon extends Earth {
    public static void main(String args[])
    {
        SolarSystem s = new SolarSystem();
        Earth e = new Earth();
        Mars m = new Mars();

        System.out.println(s instanceof SolarSystem);
        System.out.println(e instanceof Earth);
        System.out.println(m instanceof SolarSystem);
    }
}

Output:
true
```

```
true
true
```

What Can Be Done in a Subclass?

In sub-classes we can inherit members as is, replace them, hide them, or supplement them with new members:

- The inherited fields can be used directly, just like any other fields.
- We can declare new fields in the subclass that are not in the superclass.
- The inherited methods can be used directly as they are.
- We can write a new *instance* method in the subclass that has the same signature as the one in the superclass, thus **overriding** it (as in the example above, *toString()* method is overridden).
- We can write a new *static* method in the subclass that has the same signature as the one in the superclass, thus **hiding** it.
- We can declare new methods in the subclass that are not in the superclass.
- We can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword **super**.

Advantages Of Inheritance in Java:

1. Code Reusability: Inheritance allows for code reuse and reduces the amount of code that needs to be written. The subclass can reuse the properties and methods of the superclass, reducing duplication of code.
2. Abstraction: Inheritance allows for the creation of abstract classes that define a common interface for a group of related classes. This promotes abstraction and encapsulation, making the code easier to maintain and extend.

3. Class Hierarchy: Inheritance allows for the creation of a class hierarchy, which can be used to model real-world objects and their relationships.
4. Polymorphism: Inheritance allows for polymorphism, which is the ability of an object to take on multiple forms. Subclasses can override the methods of the superclass, which allows them to change their behavior in different ways.

Disadvantages of Inheritance in Java:

1. Complexity: Inheritance can make the code more complex and harder to understand. This is especially true if the inheritance hierarchy is deep or if multiple inheritances is used.
2. Tight Coupling: Inheritance creates a tight coupling between the superclass and subclass, making it difficult to make changes to the superclass without affecting the subclass.

Conclusion

Let us check some important points from the article are mentioned below:

- **Default superclass**: Except **Object** class, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of the Object class.
- **Superclass can only be one:** A superclass can have any number of subclasses. But a subclass can have only **one** superclass. This is because Java does not support multiple inheritances with classes. Although with interfaces, multiple inheritances are supported by Java.
- **Inheriting Constructors:** A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

- **Private member inheritance:** A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods(like getters and setters) for accessing its private fields, these can also be used by the subclass.

Inheritance and Constructors in Java

- Constructors in Java are used to initialize the values of the attributes of the object.
- We already have a default constructor that is called automatically if no constructor is found in the code. But if we make any constructor say parameterized constructor in order to initialize some attributes then it must write down the default constructor because it now will be no more automatically called.
- `Note:` *In Java, constructor of the base class with no argument gets automatically called in the derived class constructor.*

▼ Example 1 code:

```
// Java Program to Illustrate
// Invocation of Constructor
// Calling Without Usage of
// super Keyword

// Class 1
// Super class
class Base {

    // Constructor of super class
    Base()
    {
```

```
        // Print statement
        System.out.println(
            "Base Class Constructor Called ");
    }
}

// Class 2
// Sub class
class Derived extends Base {

    // Constructor of sub class
    Derived()
    {

        // Print statement
        System.out.println(
            "Derived Class Constructor Called ");
    }
}

// Class 3
// Main class
class GFG {

    // Main driver method
```

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

        // Creating an object of sub class
        // inside main() method
        Derived d = new Derived();

        // Note: Here first super class constructor will be
        // called there after derived(sub class) constructor
        // will be called
    }
}

output:
Base Class Constructor Called
Derived Class Constructor Called
```

▼ Example 2 code:

```
// Java Program to Illustrate Invocation
// of Constructor Calling With Usage
// of super Keyword

// Class 1
// Super class
```

```
class Base {
    int x;

    // Constructor of super class
    Base(int _x) { x = _x; }
}

// Class 2
// Sub class
class Derived extends Base {

    int y;

    // Constructor of sub class
    Derived(int _x, int _y)
    {

        // super keyword refers to super class
        super(_x);
        y = _y;
    }

    // Method of sub class
    void Display()
    {
```

```
        // Print statement
        System.out.println("x = " + x + ", y = " + y);
    }
}

// Class 3
// Main class
public class GFG {

    // Main driver method
    public static void main(String[] args)
    {

        // Creating object of sub class
        // inside main() method
        Derived d = new Derived(10, 20);

        // Invoking method inside main() method
        d.Display();
    }
}

output:
x = 10, y = 20
```

Java and Multiple Inheritance

- Multiple Inheritance is a feature of an object-oriented concept, where a class can inherit properties of more than one parent class. The problem occurs when there exist methods with the same signature in both the superclasses and subclass. On calling the method, the compiler cannot determine which class method to be called and even on calling which class method gets the priority.
- `Note:` Java doesn't support Multiple Inheritance

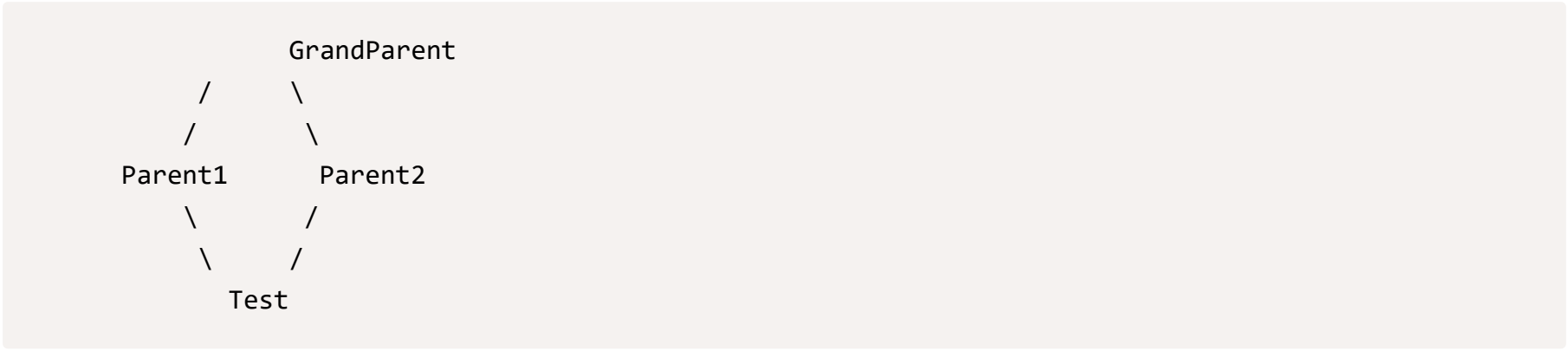
▼ Example 1 code:

```
// Java Program to Illustrate Unsupportance of
// Multiple Inheritance

// Importing input output classes
import java.io.*;

// Class 1
// First Parent class
class Parent1 {

// Method inside first parent class
void fun() {

    // Print statement if this method is called
    System.out.println("Parent1");
```

```
}
}

// Class 2
// Second Parent Class
class Parent2 {

// Method inside first parent class
void fun() {

    // Print statement if this method is called
    System.out.println("Parent2");
}
}

// Class 3
// Trying to be child of both the classes
class Test extends Parent1, Parent2 {

// Main driver method
public static void main(String args[]) {

    // Creating object of class in main() method
    Test t = new Test();
```

```
    // Trying to call above functions of class where
    // Error is thrown as this class is inheriting
    // multiple classes
    t.fun();
}
}

Output: Compilation error is thrown
```

▼ Example 2 code:




```
// Java Program to Illustrate Unsupportance of
// Multiple Inheritance
// Diamond Problem Similar Scenario

// Importing input output classes
```

```
import java.io.*;

// Class 1
// A Grand parent class in diamond
class GrandParent {

void fun() {

    // Print statement to be executed when this method is called
    System.out.println("Grandparent");
}
}

// Class 2
// First Parent class
class Parent1 extends GrandParent {
void fun() {

    // Print statement to be executed when this method is called
    System.out.println("Parent1");
}
}

// Class 3
// Second Parent Class
```

```
class Parent2 extends GrandParent {
void fun() {

    // Print statement to be executed when this method is called
    System.out.println("Parent2");
}
}

// Class 4
// Inheriting from multiple classes
class Test extends Parent1, Parent2 {

// Main driver method
public static void main(String args[]) {

    // Creating object of this class i main() method
    Test t = new Test();

    // Now calling fun() method from its parent classes
    // which will throw compilation error
    t.fun();
}
}
```

How are the above problems handled for Default Methods and Interfaces?

- Java 8 supports default methods where interfaces can provide a default implementation of methods. And a class can implement two or more interfaces. In case both the implemented interfaces contain default methods with the same method signature, the implementing class should explicitly specify which default method is to be used in some method excluding the main() of implementing class using super keyword, or it should override the default method in the implementing class, or it should specify which default method is to be used in the default overridden method of the implementing class.

▼ Example 3 code:

```
// Java program to demonstrate Multiple Inheritance
// through default methods

// Interface 1
interface PI1 {

    // Default method
    default void show()
    {

        // Print statement if method is called
        // from interface 1
        System.out.println("Default PI1");
    }
}
```

```
// Interface 2
interface PI2 {

    // Default method
    default void show()
    {

        // Print statement if method is called
        // from interface 2
        System.out.println("Default PI2");
    }
}

// Main class
// Implementation class code
class TestClass implements PI1, PI2 {

    // Overriding default show method
    @Override
    public void show()
    {

        // Using super keyword to call the show
        // method of PI1 interface
        PI1.super.show();//Should not be used directly in the main method;
```

```
        // Using super keyword to call the show
        // method of PI2 interface
        PI2.super.show();//Should not be used directly in the main method;
    }

    //Method for only executing the show() of PI1
    public void showOfPI1() {
        PI1.super.show();//Should not be used directly in the main method;
    }

    //Method for only executing the show() of PI2
    public void showOfPI2() {
        PI2.super.show(); //Should not be used directly in the main method;
    }

    // Mai driver method
    public static void main(String args[])
    {

        // Creating object of this class in main() method
        TestClass d = new TestClass();
        d.show();
        System.out.println("Now Executing showOfPI1() showOfPI2()");
        d.showOfPI1();
```

```
        d.showOfPI2();
    }
}

output:
Default PI1
Default PI2
Now Executing showOfPI1() showOfPI2()
Default PI1
Default PI2
```

▼ Example 4 code:

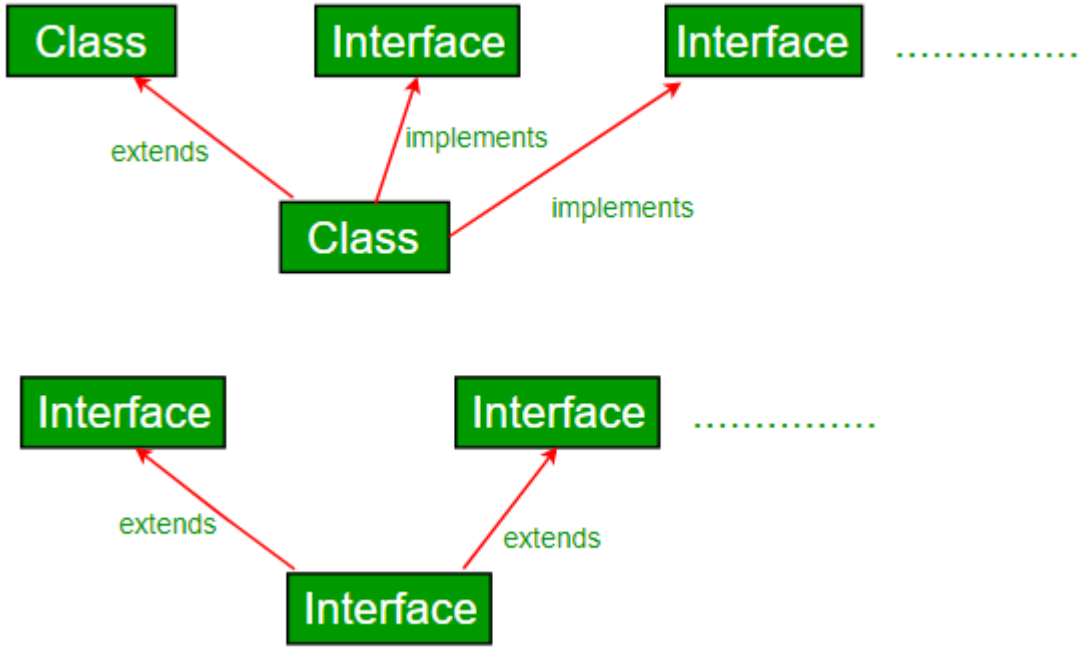
```
// Java program to demonstrate How Diamond Problem
// Is Handled in case of Default Methods

// Interface 1
interface GPI {

    // Default method
    default void show()
    {

        // Print statement
        System.out.println("Default GPI");
```

```
    }
}

// Interface 2
// Extending the above interface
interface PI1 extends GPI {
}

// Interface 3
// Extending the above interface
interface PI2 extends GPI {
}

// Main class
// Implementation class code
class TestClass implements PI1, PI2 {

    // Main driver method
    public static void main(String args[])
    {

        // Creating object of this class
        // in main() method
        TestClass d = new TestClass();
```

```
        // Now calling the function defined in interface 1
        // from whom Interface 2and 3 are deriving
        d.show();
    }
}

Output:
Default GPI
```

Interfaces and Inheritance in Java

- A class can extend another class and can implement one and more than one **Java interface**. Also, this topic has a major influence on the concept of **Java and Multiple Inheritance**.
- `Note:` *This Hierarchy will be followed in the same way, we cannot reverse the hierarchy while inheritance in Java. This means that we cannot implement a class from the interface because* ***Interface-to-Class inheritance is not allowed,*** *and it goes against the fundamental principles of class-based inheritance. This will also breach the relationship known as the Is-A relationship, where a subclass is a more specialized version of its superclass.*



▼ Example 1 code:

```
// Java program to demonstrate that a class can
// implement multiple interfaces
import java.io.*;

interface intfA {
    void m1();
}
```

```
interface intfB {
    void m2();
}

// class implements both interfaces
// and provides implementation to the method.
class sample implements intfA, intfB {
    @Override public void m1()
    {
        System.out.println("Welcome: inside the method m1");
    }

    @Override public void m2()
    {
        System.out.println("Welcome: inside the method m2");
    }
}

class GFG {
    public static void main(String[] args)
    {
        sample ob1 = new sample();

        // calling the method implemented
```

```
        // within the class.
        ob1.m1();
        ob1.m2();
    }
}

Output:
Welcome: inside the method m1
Welcome: inside the method m2
```