<https://www.journaldev.com/2752/java-8-interface-changes-static-method-default-method>

<https://beginnersbook.com/2013/05/java-abstract-class-method/>

<https://beginnersbook.com/2014/01/abstract-method-with-examples-in-java/>

<https://javarevisited.blogspot.com/2013/04/10-abstract-class-and-interface-interview-question-java-answers.html>

: <https://javarevisited.blogspot.com/2013/04/10-abstract-class-and-interface-interview-question-java-answers.html#ixzz6FX82XFX6>

: <https://javarevisited.blogspot.com/2013/04/10-abstract-class-and-interface-interview-question-java-answers.html#ixzz6FX72SzbK>

<https://medium.com/heuristics/interface-vs-abstract-class-vs-concrete-class-196f20c3af9a>

* Abstraction is a process of hiding the implementation details and showing only functionality to the user.
* A method that is declared as abstract and does not have implementation is known as abstract method.
* There are two ways to achieve abstraction in java  
  1- By Abstract class (0 to 100%)  , 2- By Interface (100%).

# Abstract Class in Java with example

A class that is declared using “**abstract**” keyword is known as abstract class. It can have abstract methods(methods without body) as well as concrete methods (regular methods with body). A normal class(non-abstract class) cannot have abstract methods.

An abstract class can not be **instantiated**, which means you are not allowed to create an **object** of it.

**Why abstract method Cannot static?**

An **abstract** class **cannot** have a **static method** because abstraction is done to achieve DYNAMIC BINDING while **static methods** are statically binded to their functionality. A **static method** means behavior not dependent on an instance variable, so no instance/object is required.

**Static Binding:**The binding which can be resolved at compile time by compiler is known as static or early binding. Binding of all the static, private and final methods is done at compile-time .

**Why binding of static, final and private methods is always a static binding?**  
Static binding is better performance wise (no extra overhead is required). Compiler knows that all such methods **cannot be overridden** and will always be accessed by object of local class. Hence compiler doesn’t have any difficulty to determine object of class (local class for sure). That’s the reason binding for such methods is static.

public class NewClass {

public static class superclass {

static void print()

{

System.out.println("print in superclass.");

}

}

public static class subclass extends superclass {

static void print()

{

System.out.println("print in subclass.");

}

}

public static void main(String[] args)

{

superclass A = new superclass();

superclass B = new subclass();

A.print();

B.print();

}

}

**Output**:

print in superclass.

print in superclass.

As you can see, in both cases print method of superclass is called. Lets see how this happens

* We have created one object of subclass and one object of superclass with the reference of the superclass.
* Since the print method of superclass is static, compiler knows that it will not be overridden in subclasses and hence compiler knows during compile time which print method to call and hence no ambiguity.

**Dynamic Binding:**In Dynamic binding compiler doesn’t decide the method to be called. Overriding is a perfect example of dynamic binding. In overriding both parent and child classes have same method . Let’s see by an example.

public class NewClass {

public static class superclass {

void print()

{

System.out.println("print in superclass.");

}

}

public static class subclass extends superclass {

@Override

void print()

{

System.out.println("print in subclass.");

}

}

public static void main(String[] args)

{

superclass A = new superclass();

superclass B = new subclass();

A.print();

B.print();

}

}

print in superclass.

print in subclass.

* Methods are not static in this code.
* During compilation, the compiler has no idea as to which print has to be called since compiler goes only by referencing variable not by type of object and therefore the binding would be delayed to runtime and therefore the corresponding version of print will be called based on type on object.

**Important Points**

* private,  final and static members (methods and variables) use static binding while for virtual methods (In Java methods are virtual by default) binding is done during run time based upon run time object.
* Static binding uses Type information for binding while Dynamic binding uses Objects to resolve binding.
* Overloaded methods are resolved (deciding which method to be called when there are multiple methods with same name) using static binding while overridden methods using dynamic binding, i.e, at run time.

## Why we need an abstract class?

Lets say we have a class Animal that has a method sound() and the subclasses(see [inheritance](https://beginnersbook.com/2013/03/inheritance-in-java/)) of it like Dog, Lion, Horse, Cat etc. Since the animal sound differs from one animal to another, there is no point to implement this method in parent class. This is because every child class must override this method to give its own implementation details, like Lion class will say “Roar” in this method and Dog class will say “Woof”.

So when we know that all the animal child classes will and should override this method, then there is no point to implement this method in parent class. Thus, making this method abstract would be the good choice as by making this method abstract we force all the sub classes to implement this method( otherwise you will get compilation error), also we need not to give any implementation to this method in parent class.

Since the Animal class has an abstract method, you must need to declare this class abstract.

Now each animal must have a sound, by making this method abstract we made it compulsory to the child class to give implementation details to this method. This way we ensures that every animal has a sound.

## Abstract class Example

//abstract parent class

abstract class Animal{

//abstract method

public abstract void sound();

}

//Dog class extends Animal class

public class Dog extends Animal{

public void sound(){

System.out.println("Woof");

}

public static void main(String args[]){

Animal obj = new Dog();

obj.sound();

}

}

Output:

Woof

Hence for such kind of scenarios we generally declare the class as abstract and later **concrete classes** extend these classes and override the methods accordingly and can have their own methods as well.

## Abstract class declaration

An abstract class outlines the methods but not necessarily implements all the methods.

//Declaration using abstract keyword

abstract class A{

//This is abstract method

abstract void myMethod();

//This is concrete method with body

void anotherMethod(){

//Does something

}

}

## Rules

**Note 1:** As we seen in the above example, there are cases when it is difficult or often unnecessary to implement all the methods in parent class. In these cases, we can declare the parent class as abstract, which makes it a special class which is not complete on its own.

A class derived from the abstract class must implement all those methods that are declared as abstract in the parent class.

**Note 2:** Abstract class cannot be instantiated which means you cannot create the object of it. To use this class, you need to create another class that extends this this class and provides the implementation of abstract methods, then you can use the object of that child class to call non-abstract methods of parent class as well as implemented methods(those that were abstract in parent but implemented in child class).

**Note 3:** If a child does not implement all the abstract methods of abstract parent class, then the child class must need to be declared abstract as well.

**Do you know?** Since abstract class allows concrete methods as well, it does not provide 100% abstraction. You can say that it provides partial abstraction. Abstraction is a process where you show only “relevant” data and “hide” unnecessary details of an object from the user.

[Interfaces](https://beginnersbook.com/2013/05/java-interface/) on the other hand are used for 100% abstraction (See more about [abstraction](https://beginnersbook.com/2013/03/oops-in-java-encapsulation-inheritance-polymorphism-abstraction/) here).  
You may also want to read this: [Difference between abstract class and Interface in Java](https://beginnersbook.com/2013/05/abstract-class-vs-interface-in-java/)

### Why can’t we create the object of an abstract class?

Because these classes are incomplete, they have abstract methods that have no body so if java allows you to create object of this class then if someone calls the abstract method using that object then What would happen?There would be no actual implementation of the method to invoke.  
Also because an object is concrete. An abstract class is like a template, so you have to extend it and build on it before you can use it.

### **When do you favor abstract class over interface?**

This is the follow-up of previous interview questions on abstract class and interface. If you know syntactical difference, you can answer this question quite easily, as they are the one, which drives the decision. Since it’s almost impossible to add a new method on a published interface, it’s better to use abstract class, when evolution is concern. Abstract class in Java evolves better than interface. Similarly, if you have too many methods inside interface, you are creating pain for all it’s implementation, consider providing an abstract class for default implementation. This is the pattern followed in Java collection package, you can see AbstractList provides default implementation for List interface.

### **Can abstract class have static methods in Java**?

Yes, abstract class can declare and define [static methods](http://javarevisited.blogspot.com/2011/11/static-keyword-method-variable-java.html), nothing prevents from doing that. But, you must follow guidelines for making a method static in Java, as it’s not welcomed in a object oriented design, because [static methods can not be overridden in Java](http://javarevisited.blogspot.com/2013/03/can-we-overload-and-override-static-method-java.html). It’s very rare, you see static methods inside abstract class, but as I said, if you have very good reason of doing it, then nothing stops you.

### **Can abstract class be final in Java?**

No, abstract class can not be final in Java. Making them final will stop abstract class from being extended, which is the only way to use abstract class

## Can abstract class have constructors in Java?

Yes, abstract class can declare and define constructor in Java. Since you can not create instance of abstract class,  constructor can only be called during [constructor chaining](http://javarevisited.blogspot.com/2012/12/constructor-chaining-in-java-calling-another-constructor.html), i.e. when you create instance of concrete implementation class.

Well, it can still be used to initialize common variables, which are declared inside abstract class, and used by various implementation.

### Example to demonstrate that object creation of abstract class is not allowed

As discussed above, we cannot instantiate an abstract class. This program throws a compilation error.

abstract class AbstractDemo{

public void myMethod(){

System.out.println("Hello");

}

abstract public void anotherMethod();

}

public class Demo extends AbstractDemo{

public void anotherMethod() {

System.out.print("Abstract method");

}

public static void main(String args[])

{

//error: You can't create object of it

AbstractDemo obj = new AbstractDemo();

obj.anotherMethod();

}

}

Output:

Unresolved compilation problem: Cannot instantiate the type AbstractDemo

Note: The class that extends the abstract class, have to implement all the abstract methods of it, else you have to declare that class abstract as well.

## Abstract class vs Concrete class

A class which is not abstract is referred as **Concrete class**. In the above example that we have seen in the beginning of this guide, Animal is a abstract class and Cat, Dog & Lion are concrete classes.

**Key Points:**

1. An abstract class has no use until unless it is extended by some other class.
2. If you declare an **abstract method** in a class then you must declare the class abstract as well. you can’t have abstract method in a concrete class. It’s vice versa is not always true: If a class is not having any abstract method then also it can be marked as abstract.
3. It can have non-abstract method (concrete) as well.

I have covered the rules and examples of abstract methods in a separate tutorial, You can find the guide here: [Abstract method in Java](https://beginnersbook.com/2014/01/abstract-method-with-examples-in-java/)  
For now lets just see some basics and example of abstract method.  
1) Abstract method has no body.  
2) Always end the declaration with a **semicolon**(;).  
3) It must be [overridden](https://beginnersbook.com/2014/01/method-overriding-in-java-with-example/). An abstract class must be extended and in a same way abstract method must be overridden.  
4) A class has to be declared abstract to have abstract methods.

**Note:** The class which is extending abstract class must override all the abstract methods.

## Example of Abstract class and method

abstract class MyClass{

public void disp(){

System.out.println("Concrete method of parent class");

}

abstract public void disp2();

}

class Demo extends MyClass{

/\* Must Override this method while extending

\* MyClas

\*/

public void disp2()

{

System.out.println("overriding abstract method");

}

public static void main(String args[]){

Demo obj = new Demo();

obj.disp2();

}

}

Output:

overriding abstract method

# Abstract method in Java with examples

A method without body (no implementation) is known as abstract method. A method must always be declared in an abstract class, or in other words you can say that if a class has an abstract method, it should be declared abstract as well. In the last tutorial we discussed Abstract class, if you have not yet checked it out read it here: [Abstract class in Java](https://beginnersbook.com/2013/05/java-abstract-class-method/), before reading this guide.

This is how an abstract method looks in java:

public abstract int myMethod(int n1, int n2);

As you see this has no body.

## Rules of Abstract Method

1. Abstract methods don’t have body, they just have method signature as shown above.  
   2. If a class has an abstract method it should be declared abstract, the vice versa is not true, which means an abstract class doesn’t need to have an abstract method compulsory.  
   3. If a regular class extends an abstract class, then the class must have to implement all the abstract methods of abstract parent class or it has to be declared abstract as well.

## Example 1: abstract method in an abstract class

//abstract class

abstract class Sum{

/\* These two are abstract methods, the child class

\* must implement these methods

\*/

public abstract int sumOfTwo(int n1, int n2);

public abstract int sumOfThree(int n1, int n2, int n3);

//Regular method

public void disp(){

System.out.println("Method of class Sum");

}

}

//Regular class extends abstract class

class Demo extends Sum{

/\* If I don't provide the implementation of these two methods, the

\* program will throw compilation error.

\*/

public int sumOfTwo(int num1, int num2){

return num1+num2;

}

public int sumOfThree(int num1, int num2, int num3){

return num1+num2+num3;

}

public static void main(String args[]){

Sum obj = new Demo();

System.out.println(obj.sumOfTwo(3, 7));

System.out.println(obj.sumOfThree(4, 3, 19));

obj.disp();

}

}

Output:

10

26

Method of class Sum

## Example 2: abstract method in interface

All the methods of an [interface](https://beginnersbook.com/2013/05/java-interface/) are public abstract by default. You cannot have concrete (regular methods with body) methods in an interface.

//Interface

interface Multiply{

//abstract methods

public abstract int multiplyTwo(int n1, int n2);

/\* We need not to mention public and abstract in interface

\* as all the methods in interface are

\* public and abstract by default so the compiler will

\* treat this as

\* public abstract multiplyThree(int n1, int n2, int n3);

\*/

int multiplyThree(int n1, int n2, int n3);

/\* Regular (or concrete) methods are not allowed in an interface

\* so if I uncomment this method, you will get compilation error

\* public void disp(){

\* System.out.println("I will give error if u uncomment me");

\* }

\*/

}

class Demo implements Multiply{

public int multiplyTwo(int num1, int num2){

return num1\*num2;

}

public int multiplyThree(int num1, int num2, int num3){

return num1\*num2\*num3;

}

public static void main(String args[]){

Multiply obj = new Demo();

System.out.println(obj.multiplyTwo(3, 7));

System.out.println(obj.multiplyThree(1, 9, 0));

}

}

Output:

21

0

# Interface in Java with example

I**nterface** is just like **Java** Class, but it only has static constants and abstract **method**. **Java** uses **Interface** to **implement** multiple inheritance. A **Java** class can **implement** multiple **Java** Interfaces. All **methods** in an **interface** are implicitly public and abstract.

Starting with Java 8, default and static **methods** may have implementation in the **interface** definition. ... **Interfaces** cannot be instantiated, but rather are implemented.

Along with abstract **methods**, an **interface** may also **contain** constants, default **methods**, static **methods**, and nested **types**. **Method** bodies exist only for default **methods** and static **methods**. Writing an **interface** is similar to writing a class.

 But from Java 8, we can have **default methods** and **static methods** in the interfaces.

**Can interface contain concrete methods ?**

**Interface** is a blueprint for your class that **can** be used to implement a class ( abstract or not); the point is **interface** cannot **have** any **concrete methods**. **Concrete methods** are those **methods** which **have** some code inside them; in one word - implemented.

* 1. **All** variables declared inside **interface** are implicitly **public** static final variables(constants).
  2. **All methods** declared inside Java **Interfaces** are implicitly **public and abstract**, even if you don't use **public** or **abstract** keyword
  3. **Interface** can extend one or more other **interface**. **Interface** cannot implement a class.

**Must know facts about Interface**

* A Java class can implement multiple Java Interfaces. It is necessary that the class must implement all the methods declared in the interfaces.
* Class should override all the abstract methods declared in the interface
* The interface allows sending a message to an object without concerning which classes it belongs.
* Class needs to provide functionality for the methods declared in the interface.
* All methods in an interface are implicitly public and abstract
* An interface cannot be instantiated
* An interface reference can point to objects of its implementing classes
* An interface can extend from one or many interfaces. Class can extend only one class but implement any number of interfaces
* An interface cannot implement another Interface. It has to extend another interface if needed.
* An interface which is declared inside another interface is referred as nested interface
* At the time of declaration, interface variable must be initialized. Otherwise, the compiler will throw an error.
* The class cannot implement two interfaces in java that have methods with same name but different return type.

**Summary:**

* The class which implements the interface needs to provide functionality for the methods declared in the interface
* All methods in an interface are implicitly public and abstract
* An interface cannot be instantiated
* An interface reference can point to objects of its implementing classes
* An interface can extend from one or many interfaces. A class can extend only one class but implement any number of interfaces

**does interface have constructor ?**

The answer is No, **interface** cannot **have constructors**. ... In order to call any method we **need** an object since there is no **need** to **have** object of **interface**, there is no **need** of having **constructor** in **interface** (**Constructor** is being called during creation of object).

interfaceSumInterface{

publicint mymethod(int num1,int num2);

}

publicclassSumClassimplementsSumInterface{

publicint mymethod(int num1,int num2){

int op= num1+num2;

return op;

}

publicstaticvoid main(String args[])

{

SumClass obj=newSumClass();

System.out.println(obj.mymethod(2,3));

}

}

As you can see we have implemented the method mymethod() in the class SumClass that implements the interface. So there is no need to have interface object(interface cannot be instantiated in fact) in order to call the method as the implementation is in the class and class object can call this method.

# **Java and Multiple Inheritance**

Multiple Inheritance is a feature of object oriented concept, where a class can inherit properties of more than one parent class. The problem occurs when there exist methods with same signature in both the super classes 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.

**Why Java doesn’t support Multiple Inheritance?**

Consider the below Java code. It shows error.

|  |
| --- |
| // First Parent class  classParent1  {      voidfun()      {          System.out.println("Parent1");      }  }    // Second Parent Class  classParent2  {      voidfun()      {          System.out.println("Parent2");      }  }    // Error : Test is inheriting from multiple  // classes  classTest extendsParent1, Parent2  {     publicstaticvoidmain(String args[])     {         Test t = newTest();         t.fun();     }  } |

Output :

Compiler Error

From the code, we see that, on calling the method fun() using Test object will cause complications such as whether to call Parent1’s fun() or Parent2’s fun() method.

**1. The Diamond Problem:**

GrandParent

/ \

/ \

Parent1 Parent2

\ /

\ /

Test

|  |
| --- |
| // A Grand parent class in diamond  classGrandParent  {      voidfun()      {          System.out.println("Grandparent");      }  }    // First Parent class  classParent1 extendsGrandParent  {      voidfun()      {          System.out.println("Parent1");      }  }    // Second Parent Class  classParent2 extendsGrandParent  {      voidfun()      {          System.out.println("Parent2");      }  }      // Error : Test is inheriting from multiple  // classes  classTest extendsParent1, Parent2  {     publicstaticvoidmain(String args[])     {         Test t = newTest();         t.fun();     }  } |

From the code, we see that: On calling the method fun() using Test object will cause complications such as whether to call Parent1’s fun() or Child’s fun() method.

Therefore, in order to avoid such complications Java does not support multiple inheritance of classes.

**2. Simplicity –** Multiple inheritance is not supported by Java using classes , handling the complexity that causes due to multiple inheritance is very complex. It creates problem during various operations like casting, constructor chaining etc and the above all reason is that there are very few scenarios on which we actually need multiple inheritance, so better to omit it for keeping the things simple and straightforward.

**How are above problems handled for**[**Default Methods and Interfaces**](https://www.geeksforgeeks.org/default-methods-java/)**?**

Java 8 supports default methods where interfaces can provide default implementation of methods. And a class can implement two or more interfaces. In case both the implemented interfaces contain default methods with same method signature, the implementing class should explicitly specify which default method is to be used or it should override the default method.

|  |
| --- |
| // A simple Java program to demonstrate multiple  // inheritance through default methods.  interfacePI1  {      // default method      defaultvoidshow()      {          System.out.println("Default PI1");      }  }    interfacePI2  {      // Default method      defaultvoidshow()      {          System.out.println("Default PI2");      }  }    // Implementation class code  classTestClass implementsPI1, PI2  {      // Overriding default show method      publicvoidshow()      {          // use super keyword to call the show          // method of PI1 interface          PI1.super.show();            // use super keyword to call the show          // method of PI2 interface          PI2.super.show();      }        publicstaticvoidmain(String args[])      {          TestClass d = newTestClass();          d.show();      }  } |

Output:

Default PI1

Default PI2

If we remove implementation of default method from “TestClass”, we get compiler error. See [this](https://ide.geeksforgeeks.org/MLJ4hg) for a sample run.

If there is a diamond through interfaces, then there is no issue if none of the middle interfaces provide implementation of root interface. If they provide implementation, then implementation can be accessed as above using super keyword.

|  |
| --- |
| // A simple Java program to demonstrate how diamond  // problem is handled in case of default methods    interfaceGPI  {      // default method      defaultvoidshow()      {          System.out.println("Default GPI");      }  }    interfacePI1 extendsGPI { }    interfacePI2 extendsGPI { }    // Implementation class code  classTestClass implementsPI1, PI2  {      publicstaticvoidmain(String args[])      {          TestClass d = newTestClass();          d.show();      }  } |

Output:

Default GPI

**DO we have to implement all methods of an interface?**

Yes, it is mandatory to **implement all** the **methods** in a class that implements an **interface** unless and until that class is an Abstract class. ... This forces **you** to declare your class abstract and, as a result, forces **you** to subclass the class (and **implement** the missing **methods**) before **you** can create any objects.

# **why we can't create an object for abstract class?**

You can not instantiate an abstract class or an interface - you can instantiate one of their subclasses/implementers.

The purpose of interfaces and abstract classes is to describe the behaviour of some concrete class that implements the interface or extends the abstract class.

**why can't we implement method interface in java?**

An interface by definition contains no implementation of anything. It specifies what needs to be implemented.

## Why default method?

For example, if several classes such as A, B, C and D implements an interface XYZInterface then if we add a new method to the XYZInterface, we have to change the code in all the classes(A, B, C and D) that implements this interface. In this example we have only four classes that implements the interface which we want to change but imagine if there are hundreds of classes implementing an interface then it would be almost impossible to change the code in all those classes. This is why in java 8, we have a new concept “default methods”. These methods can be added to any existing interface and we do not need to implement these methods in the implementation classes mandatorily, thus we can add these default methods to existing interfaces without breaking the code.

We can say that concept of default method is introduced in java 8 to add the new methods in the existing interfaces in such a way so that they are backward compatible. Backward compatibility is adding new features without breaking the old code.

**Static methods** in interfaces are similar to the default methods except that we cannot override these methods in the classes that implements these interfaces.

## Java 8 Example: Default method in Interface

The method newMethod() in MyInterface is a default method, which means we need not to implement this method in the implementation class Example. This way we can add the default methods to existing interfaces without bothering about the classes that implements these interfaces.

interface MyInterface{

/\* This is a default method so we need not

\* to implement this method in the implementation

\* classes

\*/

default void newMethod(){

System.out.println("Newly added default method");

}

/\* Already existing public and abstract method

\* We must need to implement this method in

\* implementation classes.

\*/

void existingMethod(String str);

}

public class Example implements MyInterface{

// implementing abstract method

public void existingMethod(String str){

System.out.println("String is: "+str);

}

public static void main(String[] args) {

Example obj = new Example();

//calling the default method of interface

obj.newMethod();

//calling the abstract method of interface

obj.existingMethod("Java 8 is easy to learn");

}

}

## Java 8 Example: Static method in Interface

As mentioned above, the static methods in interface are similar to default method so we need not to implement them in the implementation classes. We can safely add them to the existing interfaces without changing the code in the implementation classes. Since these methods are static, we cannot override them in the implementation classes.

interface MyInterface{

/\* This is a default method so we need not

\* to implement this method in the implementation

\* classes

\*/

default void newMethod(){

System.out.println("Newly added default method");

}

/\* This is a static method. Static method in interface is

\* similar to default method except that we cannot override

\* them in the implementation classes.

\* Similar to default methods, we need to implement these methods

\* in implementation classes so we can safely add them to the

\* existing interfaces.

\*/

static void anotherNewMethod(){

System.out.println("Newly added static method");

}

/\* Already existing public and abstract method

\* We must need to implement this method in

\* implementation classes.

\*/

void existingMethod(String str);

}

public class Example implements MyInterface{

// implementing abstract method

public void existingMethod(String str){

System.out.println("String is: "+str);

}

public static void main(String[] args) {

Example obj = new Example();

//calling the default method of interface

obj.newMethod();

//calling the static method of interface

MyInterface.anotherNewMethod();

//calling the abstract method of interface

obj.existingMethod("Java 8 is easy to learn");

}

}

Output:

Newly added default method

Newly added static method

String is: Java 8 is easy to learn

## Java 8 – Abstract classes vs interfaces

With the introduction of default methods in interfaces, it seems that the [abstract classes](https://beginnersbook.com/2013/05/java-abstract-class-method/) are same as interface in java 8. However this is not entirely true, even though we can now have concrete methods(methods with body) in interfaces just like abstract class, this doesn’t mean that they are same. There are still few differences between them, one of them is that abstract class can have constructor while in interfaces we can’t have constructors.

The purpose of interface is to provide full abstraction, while the purpose of abstract class is to provide partial abstraction. This still holds true. The interface is like a blueprint for your class, with the introduction of default methods you can simply say that we can add additional features in the interfaces without affecting the end user classes.

## Default Method and Multiple Inheritance

The [multiple inheritance](https://beginnersbook.com/2013/05/java-multiple-inheritance/) problem can occur, when we have two interfaces with the default methods of same signature. Let’s take an example.

interface MyInterface{

default void newMethod(){

System.out.println("Newly added default method");

}

void existingMethod(String str);

}

interface MyInterface2{

default void newMethod(){

System.out.println("Newly added default method");

}

void disp(String str);

}

public class Example implements MyInterface, MyInterface2{

// implementing abstract methods

public void existingMethod(String str){

System.out.println("String is: "+str);

}

public void disp(String str){

System.out.println("String is: "+str);

}

public static void main(String[] args) {

Example obj = new Example();

//calling the default method of interface

obj.newMethod();

}

}

Output:

Error: Duplicate default methods named newMethod with the parameters () and () are inherited from the types MyInterface2 and MyInterface

This is because we have the same method in both the interface and the compiler is not sure which method to be invoked.

**How to solve this issue?**  
To solve this problem, we can implement this method in the implementation class like this:

interface MyInterface{

default void newMethod(){

System.out.println("Newly added default method");

}

void existingMethod(String str);

}

interface MyInterface2{

default void newMethod(){

System.out.println("Newly added default method");

}

void disp(String str);

}

public class Example implements MyInterface, MyInterface2{

// implementing abstract methods

public void existingMethod(String str){

System.out.println("String is: "+str);

}

public void disp(String str){

System.out.println("String is: "+str);

}

//Implementation of duplicate default method

public void newMethod(){

System.out.println("Implementation of default method");

}

public static void main(String[] args) {

Example obj = new Example();

//calling the default method of interface

obj.newMethod();

}

}

Output:

Implementation of default method

# **Nested Interface in Java**

We can declare interfaces as member of a class or another interface. Such an interface is called as member interface or nested interface.

**Interface in a class**  
Interfaces (or classes) can have only public and default access specifiers when declared outside any other class (Refer [this](https://www.geeksforgeeks.org/g-fact-81/) for details). This interface declared in a class can either be default, public, protected not private. While implementing the interface, we mention the interface as**c\_name.i\_name** where **c\_name** is the name of the class in which it is nested and **i\_name** is the name of the interface itself.

**Interface in another Interface**  
An interface can be declared inside another interface also. We mention the interface as **i\_name1.i\_name2** where **i\_name1** is the name of the interface in which it is nested and **i\_name2** is the name of the interface to be implemented.

. If we try to change access specifier of interface to anything other than public, we get compiler error. Remember, [interface members can only be public.](https://www.geeksforgeeks.org/g-fact-73/).

illegal combination of modifiers: public and protected

protected interface Yes

Let us have a look at the following code:-

|  |
| --- |
| // Java program to demonstrate working of  // interface inside a class.  import java.util.\*;  class Test  {      interface Yes      {          void show();      }  }    class Testing implements Test.Yes  {      public void show()      {          System.out.println("show method of interface");      }  }    class A  {      public static void main(String[] args)      {          Test.Yes obj;          Testing t = new Testing();          obj=t;          obj.show();      } |