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

**interface** 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.

**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();      } |

An interface is similar to a class in the following ways −

* An interface can contain any number of methods.
* An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.
* The byte code of an interface appears in a **.class** file.
* Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

However, an interface is different from a class in several ways, including −

* You cannot instantiate an interface.
* An interface does not contain any constructors.
* All of the methods in an interface are abstract.
* An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
* An interface is not extended by a class; it is implemented by a class.
* An interface can extend multiple interfaces.

When overriding methods defined in interfaces, there are several rules to be followed −

* Checked exceptions should not be declared on implementation methods other than the ones declared by the interface method or subclasses of those declared by the interface method.
* The signature of the interface method and the same return type or subtype should be maintained when overriding the methods.
* An implementation class itself can be abstract and if so, interface methods need not be implemented.

When implementation interfaces, there are several rules −

* A class can implement more than one interface at a time.
* A class can extend only one class, but implement many interfaces.
* An interface can extend another interface, in a similar way as a class can extend another class.

Extending Interfaces

An interface can extend another interface in the same way that a class can extend another class. The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.

The following Sports interface is extended by Hockey and Football interfaces.

## Extending Multiple Interfaces

A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface.

The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.

For example, if the Hockey interface extended both Sports and Event, it would be declared as −

### **Example**

public interface Hockey extends Sports, Event

## Tagging Interfaces

The most common use of extending interfaces occurs when the parent interface does not contain any methods. For example, the MouseListener interface in the java.awt.event package extended java.util.EventListener, which is defined as −

### **Example**

package java.util;

public interface EventListener

{}

An interface with no methods in it is referred to as a **tagging** interface. There are two basic design purposes of tagging interfaces −

**Creates a common parent** − As with the EventListener interface, which is extended by dozens of other interfaces in the Java API, you can use a tagging interface to create a common parent among a group of interfaces. For example, when an interface extends EventListener, the JVM knows that this particular interface is going to be used in an event delegation scenario.

**Adds a data type to a class** − This situation is where the term, tagging comes from. A class that implements a tagging interface does not need to define any methods (since the interface does not have any), but the class becomes an interface type through polymorphism.