<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.