Java – Interfaces

An interface is a reference type in Java. It is similar to class. It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.

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.

An **interface in java** is a blueprint of a class. It has static constants and abstract methods.

The interface in java is **a mechanism to achieve abstraction**. There can be only abstract methods in the java interface not method body. It is used to achieve abstraction and multiple inheritance in Java.

Java Interface also **represents IS-A relationship**.

It cannot be instantiated just like abstract class.

Writing an interface is similar to writing a class. But a class describes the attributes and behaviors of an object. And an interface contains behaviors that a class implements.

Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

**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.

## Why use Java interface?

There are mainly three reasons to use interface. They are given below.

* It is used to achieve abstraction.
* By interface, we can support the functionality of multiple inheritance.
* It can be used to achieve loose coupling.

Declaring Interfaces

The **interface** keyword is used to declare an interface. Here is a simple example to declare an interface −

**Example**

Following is an example of an interface −

/\* File name : NameOfInterface.java \*/

import java.lang.\*;

// Any number of import statements

public interface NameOfInterface {

// Any number of final, static fields

// Any number of abstract method declarations\

}

Interfaces have the following properties −

* An interface is implicitly abstract. You do not need to use the **abstract** keyword while declaring an interface.
* Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
* Methods in an interface are implicitly public.

Example

/\* File name : Animal.java \*/

interface Animal {

public void eat();

public void travel();

}

Implementing Interfaces

When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.

A class uses the **implements** keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

Example

/\* File name : MammalInt.java \*/

public class MammalInt implements Animal {

public void eat() {

System.out.println("Mammal eats");

}

public void travel() {

System.out.println("Mammal travels");

}

public int noOfLegs() {

return 0;

}

public static void main(String args[]) {

MammalInt m = new MammalInt();

m.eat();

m.travel();

}

}

This will produce the following result −

Output

Mammal eats

Mammal travels

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.

Example

// Filename: Sports.java

public interface Sports {

public void setHomeTeam(String name);

public void setVisitingTeam(String name);

}

// Filename: Football.java

public interface Football extends Sports {

public void homeTeamScored(int points);

public void visitingTeamScored(int points);

public void endOfQuarter(int quarter);

}

// Filename: Hockey.java

public interface Hockey extends Sports {

public void homeGoalScored();

public void visitingGoalScored();

public void endOfPeriod(int period);

public void overtimePeriod(int ot);

}

The Hockey interface has four methods, but it inherits two from Sports; thus, a class that implements Hockey needs to implement all six methods. Similarly, a class that implements Football needs to define the three methods from Football and the two methods from Sports.

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.

## Java 8 Interface Improvement

Since Java 8, interface can have default and static methods which is discussed later.

## Internal addition by compiler

#### The java compiler adds public and abstract keywords before the interface method. More, it adds public, static and final keywords before data members.

In other words, Interface fields are public, static and final by default, and methods are public and abstract.

interface in java

**Interface in Java**

* An abstract class can contain both concrete methods and abstract methods or sometimes it may contain no abstract method, so that using abstract classes as reusable components will be a problem. To maintain the pure reusable

Components then we take the help of interfaces.

* An interface is one kind of class which contains only a group of abstract methods which can be used as a reusable component.
* An interface can be considered as a business agreement.
* To create the interfaces we use java keyword called "interface"

**Syntax:**

*interface InterfaceName{*

*//members;*

*}*

**Eg:**

*interface Calc{*

*int a=12;*

*void add();*

*void sub();*

*}*

**Note:**

Whenever any java program is compiled then java compiler will generate .class file for every class available in the program and also create the .class file for every interface available in the program.

If we compile the above program then compiler will generate .class file like Calc.class.

**Displaying the profile of Calc.class**

> **javap Calc**

*interface Calc{*

*public static final int a;*

*public abstract void add();*

*public abstract void sub();*

*}*

**Note:**

All the variables declared in interface are public static final by default whether we specify or not

All the methods declared in interface are public abstract by default whether we specify or not

**public:**

All the variables and methods declared in interface are public so that they can be accessible from any where.

**static:**

Variables declared in itnerface are bydefault static so that they can be accessible directly by using the interface name.

**final:**

The variables declared in interface are bydefault final it means they are constant whose value cannot be changed.

**Abstract:**

All the methods declared in interface are abstract because they dont contain any method body.

**Note:**

1. For an interface we can not create any object directly

2. But we can create a reference for interface

If we want to create an object for any interface then we must create a class that inherit the particular interface and provide the implementation for all the abstract methods of that interface.

The members of interface can be inherited into any class by using a keyword called "implements " this concept is also considered as IS-A relation.

**Syntax:**

*class classname implements interfacename{*

*// members*

*}*

**Rules**

1. Once an interface is implemented by any class then that class must provide the implementation for all the methods available in the particular interface.This class is also called as implementation class or child class.
2. For example if our class is not providing implementation for at least 1 method then our class must be declared as abstract.

3. We cannot create an object for abstract class or interface but we can create an object only for implementation class.

Once an interface is created then any number of classes can inherit that interface

**//program to demo on interfaces**

*interface Vehicle{*

*void travelling();*

*}*

*class Bus implements Vehicle{*

*public void travelling(){*

*System.out.println("Travelling on the dividers");*

*}*

*}*

*class TrackBus implements Vehicle{*

*public void travelling(){*

*System.out.println("Travelling on the roads");*

*}*

*}*

*class AirBus implements Vehicle{*

*public void travelling(){*

*System.out.println("Travelling on the rivers");*

*}*

*}*

*class Interface1{*

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

*Bus b = new Bus();*

*b.travelling();*

*TrackBus tb = new TrackBus();*

*tb.travelling();*

*AirBus ab = new AirBus();*

*ab.travelling();*

*}*

*}*

Variables in interface are final so while declaring the variable in interface specifying the initial value is mandatory

Variables in interface are public static so that we can access directly in side the implementation classes but if we want to use in any other class we can access by using InterfaceName directly.

**Eg:**

*interface Shape{*

*double PI=3.14;*

*void area();*

*}*

*class Circle implements Shape{*

*int r;*

*Circle(int r){*

*this.r=r;*

*}*

*public void area(){*

*System.out.println("Area of Circle:"+PI\*r\*r);*

*}*

*}*

*class Sphere implements Shape{*

*int r;*

*Sphere(int r){*

*this.r=r;*

*}*

*public void area(){*

*System.out.println("Area of Sphere:"+4/3.0\*PI\*r\*r\*r);*

*}*

*}*

*class Interface2{*

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

*Shape s;*

*s = new Circle(5);*

*s.area();*

*s = new Sphere(4);*

*s.area();*

*System.out.println(Shape.PI);*

*}*

*}*

Multiple inheritance is not possible in java using classes because in java a class cannot extend more than 1 class.

But we can achieve this multiple inheritance by taking the help of interfaces because in java a class can implement any number of interfaces.

*interface Calc1{*

*void add(int x,int y);*

*void sub(int x,int y);*

*void mul(int x,int y);*

*}*

*interface Calc2{*

*void sub(int x,int y);*

*}*

*interface Calc3{*

*void mul(int x,int y);*

*}*

*interface Calc4{*

*void div(int x,int y);*

*}*

*class Demo{*

*}*

*class Calculator implements Calc1,Calc2,Calc3,Calc4 {*

*public void add(int a,int b){*

*System.out.println("Addition: "+(a+b));*

*}*

*public void sub(int a,int b){*

*System.out.println("Subtraction: "+(a-b));*

*}*

*public void mul(int a,int b){*

*System.out.println("Multiplication: "+(a\*b));*

*}*

*public void div(int a,int b){*

*System.out.println("Division: "+(a/b));*

*}*

*}*

*class Interface3{*

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

*Calc1 c = new Calculator();*

*c.add(10,20);*

*c.sub(10,2);*

*}*

*}*

**Rules**





|  |  |
| --- | --- |
| **abstract classes** | **interfaces** |
| 1.abstra class contains both abstract methods concrete methods | 1. but interface contains only abstract methods |
| 2.writing abstract keyword mandatory for abstract methods | 2. writing abstract keyword optional |
| 3. It contains both public & non public members | 3. It contains only public members |
| 4. It contains both static & non-static methods | 4.It contains only instance(nonstatic)methods |
| 5.It contains both static & non-static variables | 5. It contains only static variables |
| 6.It contains both final &non-final variables | 6. It contains only final variables |
| 7. constructor concept is there for abstract classes | 7. But we dont have any constructor concept for interfaces |
| 8.Super most class is Object by default. | 8.But we dont have any super most interface concept but we can extend and create Parent interfaces |
| 9.absrtact class canbe inheirted using extends keyword | 9. But interfaces canbe inheirted using implements keyword |
| 10.we can write main() method | 10. we cannot write main() method |

**Marked interface or tagged interface**

We can also declare an interface without any abstract methods which is called Marked interface or tagged interface

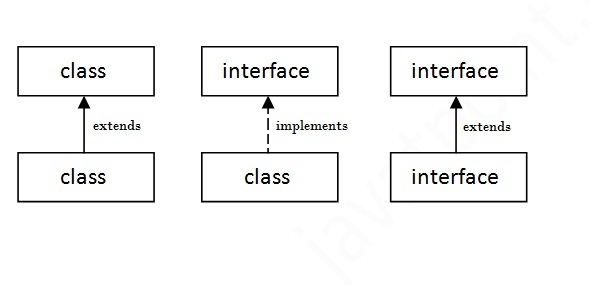
The main advantage of Marked interfaces is giving an instruction to JVM to perform a special task.

**Eg:**

Cloneable, Serializable, EventListener,...

#### Understanding relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface but a **class implements an interface**.



## Java Interface Example

In this example, Printable interface has only one method, its implementation is provided in the A class.

1. interface printable{
2. void print();
3. }
4. class A6 implements printable{
5. public void print(){System.out.println("Hello");}
7. public static void main(String args[]){
8. A6 obj = new A6();
9. obj.print();
10. }
11. }



[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=A6)

Output:

Hello

## Java Interface Example: Drawable

In this example, Drawable interface has only one method. Its implementation is provided by Rectangle and Circle classes. In real scenario, interface is defined by someone but implementation is provided by different implementation providers. And, it is used by someone else. The implementation part is hidden by the user which uses the interface.

File: TestInterface1.java

1. //Interface declaration: by first user
2. interface Drawable{
3. void draw();
4. }
5. //Implementation: by second user
6. class Rectangle implements Drawable{
7. public void draw(){System.out.println("drawing rectangle");}
8. }
9. class Circle implements Drawable{
10. public void draw(){System.out.println("drawing circle");}
11. }
12. //Using interface: by third user
13. class TestInterface1{
14. public static void main(String args[]){
15. Drawable d=new Circle();//In real scenario, object is provided by method e.g. getDrawable()
16. d.draw();
17. }}



[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestInterface1)

Output:

drawing circle

## Java Interface Example: Bank

Let's see another example of java interface which provides the implementation of Bank interface.

File: TestInterface2.java

1. interface Bank{
2. float rateOfInterest();
3. }
4. class SBI implements Bank{
5. public float rateOfInterest(){return 9.15f;}
6. }
7. class PNB implements Bank{
8. public float rateOfInterest(){return 9.7f;}
9. }
10. class TestInterface2{
11. public static void main(String[] args){
12. Bank b=new SBI();
13. System.out.println("ROI: "+b.rateOfInterest());
14. }}



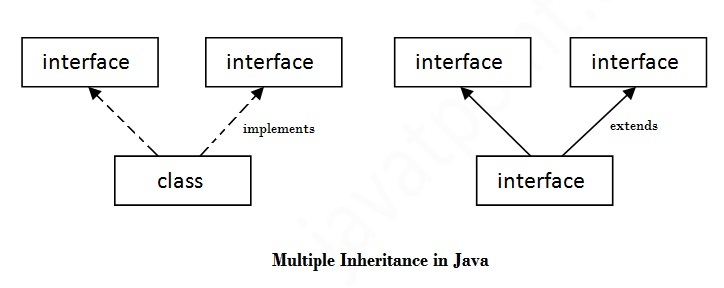
[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestInterface2)

Output:

ROI: 9.15

## Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.



1. interface Printable{
2. void print();
3. }
4. interface Showable{
5. void show();
6. }
7. class A7 implements Printable,Showable{
8. public void print(){System.out.println("Hello");}
9. public void show(){System.out.println("Welcome");}
11. public static void main(String args[]){
12. A7 obj = new A7();
13. obj.print();
14. obj.show();
15. }
16. }



[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=A7)

Output:Hello

Welcome

## Q) Multiple inheritance is not supported through class in java but it is possible by interface, why?

As we have explained in the inheritance chapter, multiple inheritance is not supported in case of class because of ambiguity. But it is supported in case of interface because there is no ambiguity as implementation is provided by the implementation class. For example:

1. interface Printable{
2. void print();
3. }
4. interface Showable{
5. void print();
6. }
8. class TestTnterface3 implements Printable, Showable{
9. public void print(){System.out.println("Hello");}
10. public static void main(String args[]){
11. TestTnterface1 obj = new TestTnterface1();
12. obj.print();
13. }
14. }



[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=testInterface3)

Output:

Hello

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestTnterface1, so there is no ambiguity.

## Interface inheritance

A class implements interface but one interface extends another interface .

1. interface Printable{
2. void print();
3. }
4. interface Showable extends Printable{
5. void show();
6. }
7. class TestInterface4 implements Showable{
8. public void print(){System.out.println("Hello");}
9. public void show(){System.out.println("Welcome");}
11. public static void main(String args[]){
12. TestInterface4 obj = new TestInterface4();
13. obj.print();
14. obj.show();
15. }
16. }



[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestInterface4)

Output:

Hello

Welcome

## Java 8 Default Method in Interface

Since Java 8, we can have method body in interface. But we need to make it default method. Let's see an example:

File: TestInterfaceDefault.java

1. interface Drawable{
2. void draw();
3. default void msg(){System.out.println("default method");}
4. }
5. class Rectangle implements Drawable{
6. public void draw(){System.out.println("drawing rectangle");}
7. }
8. class TestInterfaceDefault{
9. public static void main(String args[]){
10. Drawable d=new Rectangle();
11. d.draw();
12. d.msg();
13. }}



[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestInterfaceDefault)

Output:

drawing rectangle

default method

## Java 8 Static Method in Interface

Since Java 8, we can have static method in interface. Let's see an example:

File: TestInterfaceStatic.java

1. interface Drawable{
2. void draw();
3. static int cube(int x){return x\*x\*x;}
4. }
5. class Rectangle implements Drawable{
6. public void draw(){System.out.println("drawing rectangle");}
7. }
9. class TestInterfaceStatic{
10. public static void main(String args[]){
11. Drawable d=new Rectangle();
12. d.draw();
13. System.out.println(Drawable.cube(3));
14. }}



[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestInterfaceStatic)

Output:

drawing rectangle

27

## Q) What is marker or tagged interface?

An interface that have no member is known as marker or tagged interface. For example: Serializable, Cloneable, Remote etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

1. //How Serializable interface is written?
2. public interface Serializable{
3. }



#### Nested Interface in Java

Note: An interface can have another interface i.e. known as nested interface. We will learn it in detail in the nested classes chapter. For example:

1. interface printable{
2. void print();
3. interface MessagePrintable{
4. void msg();
5. }
6. }

## Interfaces and Inheritance

It is possible for a Java interface to inherit from another Java interface, just like classes can inherit from other classes. You specify inheritance using the extends keyword. Here is a simple interface inheritance example:

public interface MySuperInterface {

public void saiHello();

}

public interface MySubInterface **extends MySuperInterface** {

public void sayGoodbye();

}

The interface MySubInterface extends the interface MySuperInterface. That means, that the MySubInterface inherits all field and methods from MySuperInterface. That then means, that if a class implements MySubInterface, that class has to implement all methods defined in both MySubInterface and MySuperInterface.

It is possible to define methods in a subinterface with the same signature (name + parameters) as methods defined in a superinterface, should you find that desirable in your design, somehow.

Unlike classes, interfaces can actually inherit from multiple superinterfaces. You specify that by listing the names of all interfaces to inherit from, separated by comma. A class implementing an interface which inherits from multiple interfaces must implement all methods from the interface and its superinterfaces.

Here is an example of a Java interface that inherits from multiple interfaces:

public interface MySubInterface extends

SuperInterface1, SuperInterface2 {

public void sayItAll();

}

As when implementing multiple interfaces, there are no rules for how you handle the situation when multiple superinterfaces have methods with the same signature (name + parameters).

### Inheritance and Default Methods

Interface default methods add a bit complexity to the rules of interface inheritance. While it is normally possible for a class to implement multiple interfaces even if the interfaces contain methods with the same signature, this is not possible if one or more of these methods are default methods. In other words, if two interfaces contain the same method signature (name + parameters) and one of the interfaces declare this method as a default method, a class cannot automatically implement both interfaces.

The situation is the same if an interface extends (inherits from) multiple interfaces, and one or more of these interfaces contain methods with the same signature, and one of the superinterfaces declare the overlapping method as a default method.

In both of the above situations the Java compiler requires that the class implementing the interface(s) explicitly implements the method which causes the problem. That way there is no doubt about which implementation the class will have. The implementation in the class takes precedence over any default implementations.

## Interfaces and Polymorphism

Java interfaces are a way to achieve polymorphism. Polymorphism is a concept that takes some practice and thought to master. Basically, polymorphism means that an instance of an class (an object) can be used as if it were of different types. Here, a type means either a class or an interface.

Look at this simple class diagram:

|  |
| --- |
| Two parallel class hierarchies used in the same application. |
| **Two parallel class hierarchies used in the same application.** |

The classes above are all parts of a model representing different types of vehicles and drivers, with fields and methods. That is the responsibility of these classes - to model these entities from real life.

Now imagine you needed to be able to store these objects in a database, and also serialize them to XML, JSON, or other formats. You want that implemented using a single method for each operation, available on each Car, Truck or Vehicle object. A store() method, a serializeToXML() method and a serializeToJSON() method.

Please forget for a while, that implementing this functionality as methods directly on the objects may lead to a messy class hierarchy. Just imagine that this is how you want the operations implemented.

Where in the above diagram would you put these three methods, so they are accessible on all classes?

One way to solve this problem would be to create a common superclass for the Vehicle and Driver class, which has the storage and serialization methods. However, this would result in a conceptual mess. The class hierarchy would no longer model vehicles and drivers, but also be tied to the storage and serialization mechanisms used in your application.

A better solution would be to create some interfaces with the storage and serialization methods on, and let the classes implement these interfaces. Here are examples of such interfaces:

public interface Storable {

public void store();

}

public interface Serializable {

public void serializeToXML(Writer writer);

public void serializeToJSON(Writer writer);

}

When each class implements these two interfaces and their methods, you can access the methods of these interfaces by casting the objects to instances of the interface types. You don't need to know exactly what class a given object is of, as long as you know what interface it implements. Here is an example:

Car car = new Car();

Storable storable = (Storable) car;

storable.store();

Serializable serializable = (Serializable) car;

serializable.serializeToXML (new FileWriter("car.xml"));

serializable.serializeToJSON(new FileWriter("car.json"));

As you can probably imagine by now, interfaces provide a cleaner way of implementing cross cutting functionality in classes than inheritance.

### Important Points about Interface in Java

1. interface is the code that is used to create an interface in java.
2. We can’t instantiate an interface in java.
3. Interface provides absolute abstraction, in last post we learned about [abstract classes in java](http://www.journaldev.com/1582/abstract-class-in-java) to provide abstraction but abstract classes can have method implementations but interface can’t.
4. Interfaces can’t have constructors because we can’t instantiate them and interfaces can’t have a method with body.
5. By default any attribute of interface is **public**, **static** and **final**, so we don’t need to provide access modifiers to the attributes but if we do, compiler doesn’t complain about it either.
6. By default interface methods are implicitly **abstract** and **public**, it makes total sense because the method don’t have body and so that subclasses can provide the method implementation.
7. An interface can’t extend any class but it can extend another interface. public interface Shape extends Cloneable{} is an example of an interface extending another interface. Actually java provides multiple inheritance in interfaces, what is means is that an interface can extend multiple interfaces.
8. implements keyword is used by classes to implement an interface.
9. A class implementing an interface must provide implementation for all of its method unless it’s an abstract class. For example, we can implement above interface in abstract class like this:

ShapeAbs.java

package com.journaldev.design;

public abstract class ShapeAbs implements Shape {

@Override

public double getArea() {

// TODO Auto-generated method stub

return 0;

}

}

1. We should always try to write programs in terms of interfaces rather than implementations so that we know beforehand that implementation classes will always provide the implementation and in future if any better implementation arrives, we can switch to that easily.

Java interfaces are different from classes, and it's important to know how to use their special properties in your programs. This article introduces the difference between classes and interfaces, then guides you through short examples demonstrating how to declare, implement, and extend Java interfaces. I also demonstrate how the interface has evolved in Java 8, with the addition of default and static methods. These additions make interfaces more useful to experienced developers, but they also blur the lines between classes and interfaces, making interface programming even more confusing to Java beginners.

What is an interface?

An interface is a point where two systems meet and interact. For example, a vending machine interface is a mechanism that allows users to select an item, pay for it, and receive the desired food or drink. From a programming perspective, an interface sits between software components. Consider that a method header (method name, parameter list, and so on) interface sits between external code that calls the method and the code within the method that will be executed as a result of the call. Here is an example:

System.out.println(average(10, 15));

double average(double x, double y) // interface between average(10, 15) call and return (x + y) / 2;

{

return (x + y) / 2;

}

What's often confusing to Java beginners is that classes also have interfaces. As I explained in [Java 101: Classes and objects in Java](http://www.javaworld.com/article/2979739/learn-java/java-101-classes-and-objects-in-java.html), the interface is the part of the class that is accessible to code located outside of it. A class's interface consists of some combination of methods, fields, constructors, and other entities. Consider Listing 1.

Listing 1. Declaring an Account class

class Account

{

private String name;

private long amount;

Account(String name, long amount)

{

this.name = name;

setAmount(amount);

}

void deposit(long amount)

{

this.amount += amount;

}

String getName()

{

return name;

}

long getAmount()

{

return amount;

}

void setAmount(long amount)

{

this.amount = amount;

}

}

The Account(String name, long amount) constructor and the void deposit(long amount), String getName(), long getAmount(), and void setAmount(long amount) methods form the Account class's interface: they are accessible to external code. The private String name; and private long amount; fields are inaccessible.

The code that supports a method's or a class's interface (such as a class's private fields) is known as implementation code. Implementation code should be hidden from external code so that it can be changed to meet evolving requirements.

Exposed implementation code can lead to unwanted interdependencies between software components. For example, method code might come to rely on external variables, or a class's users could become dependent on fields that should have been hidden. This coupling might not be an issue for the early iterations of the software, but it can lead to problems when an implementation must evolve.

Java developers use the interface language feature to abstract class interfaces, thus decoupling classes from their users. By focusing on Java interfaces instead of classes, you can minimize the number of references to class names in your source code. This facilitates changing from one class to another (perhaps to improve performance) as your software matures. Here is an example:

List names = new ArrayList<>()

void print(List names)

{

// ...

}

This simple program declares and initializes a names field that stores a list of string names. The program also declares a print() method for printing out the contents of a list of strings, perhaps one string per line. For brevity, I've omitted the method's implementation.

List is a Java interface that describes a sequential collection of objects. ArrayList is a class that describes an array-based implementation of the List Java interface. A new instance of the ArrayList class is obtained and assigned to List variable names. (List and ArrayList are stored in the standard class library's java.util package.)

Angle brackets and generics

The angle brackets (< and >) are part of Java's generics feature set. They indicate that names describes a list of strings (only strings can be stored in the list). I'll introduce generics in a future Java 101 article.

When client code interacts with names, it will invoke those methods that are declared by List, and which are implemented by ArrayList. The client code will not interact directly with ArrayList. As a result, the client code will not break when a different implementation class, such as LinkedList, is required:

List names = new LinkedList<>()

// ...

void print(List names)

{

// ...

}

Because the print() method parameter type is List, this method's implementation doesn't have to change. However, if the type had been ArrayList, the type would have to be changed to LinkedList. If both classes were to declare their own unique methods, you might need to significantly change print()'s implementation.

Decoupling List from ArrayList and LinkedList lets you write code that's immune to class-implementation changes. By using Java interfaces, you can avoid problems that could arise from relying on implementation classes. This decoupling is the main reason for using Java interfaces.

Interfaces vs Java interfaces

Earlier, I distinguished between interface (a generic term) and Java interface (Java's interface language feature) to avoid confusion. Because the remainder of the article focuses on Java interfaces, I'll drop "Java" and just specify "interfaces."

Interface declaration

You declare an interface by adhering to a class-like syntax that consists of a header followed by a body. At minimum, the header consists of keyword interface followed by a name that identifies the interface. The body starts with an open-brace character and ends with a close-brace. Between these delimiters are constant and method header declarations:

interface identifier

{

// interface body

}

By convention, the first letter of an interface's name is uppercased and subsequent characters are lowercased (for example, Drawable). If a name consists of multiple words, the first letter of each word is uppercased (such as DrawableAndFillable). This naming convention is known as [CamelCasing](https://en.wikipedia.org/wiki/CamelCase).

Listing 2 declares an interface named Drawable.

Listing 2. Declaring a Drawable interface

interface Drawable

{

int RED = 1;

int GREEN = 2;

int BLUE = 3;

int BLACK = 4;

int WHITE = 5;

void draw(int color);

}

Interface names in Java's standard class library

Many interface names in Java's standard class library end with the able suffix. Examples include Callable, Cloneable, Comparable, Formattable, Iterable, Runnable, Serializable, and Transferable. The suffix isn't mandatory, however; the standard class library includes the interfaces CharSequence, ClipboardOwner, Collection, Executor, Future, Iterator, List, Map and many others.

Drawable declares five fields that identify color constants. This interface also declares the header for a draw() method that must be called with one of these constants to specify the color used to draw an outline. (Using integer constants isn't a good idea because any integer value could be passed to draw(). However, they suffice in a simple example.)

Field and method header defaults

Fields that are declared in an interface are implicitly public final static. An interface's method headers are implicitly public abstract.

Drawable identifies a reference type that specifies what to do (draw something) but not how to do it. Implementation details are consigned to classes that implement this interface. Instances of such classes are known as drawables because they know how to draw themselves.

Marker and tagging interfaces

An interface with an empty body is known as a marker interface or a tagging interface. The interface exists only to associate metadata with a class. For example, Cloneable (see [Inheritance in Java, Part 2](http://www.javaworld.com/article/2987584/core-java/java-101-inheritance-in-java-part-2.html)) implies that instances of its implementing class can be shallowly cloned. When Object's clone() method detects (via [runtime type identification](http://www.javaworld.com/article/3033445/learn-java/java-101-polymorphism-in-java.html)) that the calling instance's class implements Cloneable, it shallowly clones the object.

Implementing interfaces

A class implements an interface by appending Java's implements keyword followed by a comma-separated list of interface names to the class header, and by coding each interface method in the class. Listing 3 presents a class that implements Listing 2's Drawable interface.

Listing 3. Circle implementing the Drawable interface

class Circle implements Drawable

{

private double x, y, radius;

Circle(double x, double y, double radius)

{

this.x = x;

this.y = y;

this.radius = radius;

}

@Override

public void draw(int color)

{

System.out.println("Circle drawn at (" + x + ", " + y +

"), with radius " + radius + ", and color " + color);

}

double getRadius()

{

return radius;

}

double getX()

{

return x;

}

double getY()

{

return y;

}

}

Listing 3's Circle class describes a circle as a center point and a radius. As well as providing a constructor and suitable getter methods, Circle implements the Drawable interface by appending implements Drawable to the Circle header, and by overriding (as indicated by the @Override annotation) Drawable's draw() method header.

Interface method header overriding caution

When you implement an interface method (by overriding the interface's method header), remember that all of the methods whose headers are declared in the interface are implicitly declared public. If you forget to include public in the implemented method's declaration, the compiler will report an error informing you that you're attempting to assign weaker access to the implemented method.

Listing 4 presents a second example: a Rectangle class that also implements Drawable.

Listing 4. Implementing the Drawable interface in a Rectangle context

class Rectangle implements Drawable

{

private double x1, y1, x2, y2;

Rectangle(double x1, double y1, double x2, double y2)

{

this.x1 = x1;

this.y1 = y1;

this.x2 = x2;

this.y2 = y2;

}

@Override

public void draw(int color)

{

System.out.println("Rectangle drawn with upper-left corner at (" + x1 +

", " + y1 + ") and lower-right corner at (" + x2 +

", " + y2 + "), and color " + color);

}

double getX1()

{

return x1;

}

double getX2()

{

return x2;

}

double getY1()

{

return y1;

}

double getY2()

{

return y2;

}

}

Listing 4's Rectangle class describes a rectangle as a pair of points denoting the upper-left and lower-right corners of this shape. As with Circle, Rectangle provides a constructor and suitable getter methods, and also implements the Drawable interface.

Overriding interface method headers

The compiler reports an error when you attempt to compile a non-abstract class that includes an implements interface clause but doesn't override all of the interface's method headers.

An interface type's data values are the objects whose classes implement the interface and whose behaviors are those specified by the interface's method headers. This fact implies that you can assign an object's reference to a variable of the interface type, provided that the object's class implements the interface. Listing 5 demonstrates.

Listing 5. Aliasing Circle and Rectangle objects as Drawables

class Draw

{

public static void main(String[] args)

{

Drawable[] drawables = new Drawable[] { new Circle(10, 20, 15),

new Circle(30, 20, 10),

new Rectangle(5, 8, 8, 9) };

for (int i = 0; i < drawables.length; i++)

drawables[i].draw(Drawable.RED);

}

}

Because Circle and Rectangle implement Drawable, Circle and Rectangle objects have Drawable type in addition to their class types. Therefore, it's legal to store each object's reference in an array of Drawables. A loop iterates over this array, invoking each Drawable object's draw() method to draw a circle or a rectangle.

Assuming that Listing 2 is stored in a Drawable.java source file, which is in the same directory as the Circle.java, Rectangle.java, and Draw.java source files (which respectively store Listing 3, Listing 4, and Listing 5), compile these source files via either of the following command lines:

javac Draw.java

javac \*.java

Run the Draw application as follows:

java Draw

You should observe the following output:

Circle drawn at (10.0, 20.0), with radius 15.0, and color 1

Circle drawn at (30.0, 20.0), with radius 10.0, and color 1

Rectangle drawn with upper-left corner at (5.0, 8.0) and lower-right corner at (8.0, 9.0), and color 1

Note that you could also generate the same output by specifying the following main() method:

public static void main(String[] args)

{

Circle c = new Circle(10, 20, 15);

c.draw(Drawable.RED);

c = new Circle(30, 20, 10);

c.draw(Drawable.RED);

Rectangle r = new Rectangle(5, 8, 8, 9);

r.draw(Drawable.RED);

}

As you can see, it's tedious to repeatedly invoke each object's draw() method. Furthermore, doing so adds extra bytecode to Draw's class file. By thinking of Circle and Rectangle as Drawables, you can leverage an array and a simple loop to simplify the code. This is an additional benefit from designing code to prefer interfaces over classes.

Implementing multiple interfaces

Earlier, I mentioned that a class can implement multiple interfaces. Each interface's name is specified as part of a comma-separated list of names that follows the implements keyword. Listing 6 presents a simple example where class C implements interfaces A and B.

Listing 6. Implementing multiple interfaces

interface A

{

// appropriate constants and/or method headers

}

interface B

{

// appropriate constants and/or method headers

}

class C implements A, B

{

// override A's and B's method headers

}

Beware of the potential for name collisions when implementing multiple interfaces. This occurs when the same constant name appears in each interface, possibly with different type and/or other information, and is accessed in the class. When a name collision occurs, the compiler will report an error, which is demonstrated in Listing 7.

Listing 7. Demonstrating colliding constants

interface A

{

int CONSTANT = 2;

void method();

}

interface B

{

int CONSTANT = 3;

int method(int x);

}

class C implements A, B

{

int x = CONSTANT;

@Override

public void method()

{

}

@Override

public int method(int x)

{

return x;

}

}

Here, class C is inheriting two different constants named CONSTANT that are initialized to two different values. The Java compiler cannot determine which constant should be inherited by C (the same problem would occur if each constant was assigned the same value) and reports the following error message:

C.java:15: error: reference to CONSTANT is ambiguous

int x = CONSTANT;

^

both variable CONSTANT in A and variable CONSTANT in B match

1 error

Extending interfaces

A class that implements an interface reveals interface inheritance. The class inherits the interface's constants and method headers, which it overrides. For example, each of Circle and Rectangle inherits Drawable's five integer constants and draw() method header.

Interface inheritance is also demonstrated when an interface extends another interface. Just as a subclass can extend a superclass via reserved word extends, you can use this reserved word to have a subinterface extend a superinterface. Listing 8 demonstrates.

Listing 8. Declaring a Fillable subinterface that extends the Drawable superinterface

interface Fillable extends Drawable

{

void fill(int color);

}

Fillable extends Drawable, inheriting its color constants and draw() method header. Fillable also declares the header for a fill() method that must be called with one of these constants to specify the color used to fill an interior. (Fillable extends Drawable to support drawing an outline as well as filling an interior.)

You could retrofit the previous Circle and Rectangle classes to support Fillable by performing the following steps:

Change implements Drawable to implements Fillable. There is no need to specify either implements Drawable, Fillable or implements Fillable, Drawable because Fillable includes all of Drawable by extension.

Override the fill() method header in the same manner as overriding the draw() method header.

Listing 9 presents an equivalent Fill application that demonstrates the Fill interface.

Listing 9. Aliasing Circle and Rectangle objects as Fillables

class Fill

{

public static void main(String[] args)

{

Fillable[] fillables = new Fillable[] { new Circle(10, 20, 15),

new Circle(30, 20, 10),

new Rectangle(5, 8, 8, 9) };

for (int i = 0; i < fillables.length; i++)

{

fillables[i].draw(Drawable.RED);

fillables[i].fill(Fillable.BLACK);

}

}

}

Circle and Rectangle implement Fillable, giving Circle and Rectangle objects a Fillable type in addition to their class types. Therefore, it's legal to store each object's reference in an array of Fillables. A loop iterates over this array, invoking each Fillable's inherited draw() and non-inherited fill() methods to draw and fill a circle or a rectangle.

If Listing 2 is stored in Drawable.java, which is in the same directory as Circle.java, Rectangle.java, Fillable.java, and Fill.java (respectively storing Listing 3 and Listing 4, with updates, Listing 8, and a source file that stores Listing 9) you can compile these source files using either of the following command lines:

javac Fill.java

javac \*.java

Run the Fill application as follows:

java Fill

You should observe the following output:

Circle drawn at (10.0, 20.0), with radius 15.0, and color 1

Circle filled at (10.0, 20.0), with radius 15.0, and color 4

Circle drawn at (30.0, 20.0), with radius 10.0, and color 1

Circle filled at (30.0, 20.0), with radius 10.0, and color 4

Rectangle drawn with upper-left corner at (5.0, 8.0) and lower-right corner at (8.0, 9.0), and color 1

Rectangle filled with upper-left corner at (5.0, 8.0) and lower-right corner at (8.0, 9.0), and color 4

You can upcast the interface type of an object from a subinterface to a superinterface because a subinterface is a kind of superinterface. For example, you could assign a Fillable reference to a Drawable variable and then invoke Drawable's draw() method on the variable:

Drawable d = fillables[0];

d.draw(Drawable.GREEN);

Extending multiple interfaces

As with interface implementation, you can extend multiple interfaces. Each interface's name is specified as part of a comma-separated list of names that follows the extends keyword. Listing 10 presents a simple example where interface C extends interfaces A and B.

Listing 10. Extending multiple interfaces

interface A

{

// appropriate constants and/or method headers

}

interface B

{

// appropriate constants and/or method headers

}

interface C extends A, B

{

// appropriate constants and/or method headers

}

Beware of the potential for name collisions when extending multiple interfaces. This occurs when the same constant name appears in each superinterface, possibly with different type and/or other information, and is accessed in the subinterface. When a name collision occurs, the compiler will report an error, which is demonstrated in Listing 11.

Listing 11. Demonstrating colliding constants

interface A

{

int CONSTANT = 2;

void method();

}

interface B

{

int CONSTANT = 3;

int method(int x);

}

interface C extends A, B

{

int CONSTANT2 = CONSTANT;

}

Here, interface C is inheriting two different constants named CONSTANT that are initialized to two different values. The Java compiler cannot determine which constant should be inherited by C (the same problem would occur if each constant was assigned the same value) and reports the following error message:

C.java:15: error: reference to CONSTANT is ambiguous

int CONSTANT2 = CONSTANT;

^

both variable CONSTANT in A and variable CONSTANT in B match

1 error

Interfaces

Interfaces in java are very much similar to abstract classes but interfaces contain only abstract methods (you can refer to them as only ideas). Abstract classes may contain both abstract methods as well as concrete methods. But interfaces must contain only abstract methods. Concrete methods are not allowed in interfaces. Therefore, Interfaces show 100% abstractness.

Let’s discuss some of the points regarding Interfaces.

* Interfaces are declared with keyword ‘**interface**‘ and interfaces are implemented by the class using ‘**implements**‘ keyword.

[?](http://javaconceptoftheday.com/interfaces-in-java/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | interface InterfaceClass  {      //Some Abstract methods  }    class AnyClass implements InterfaceClass  {      //Use 'implements' while implementing Interfaces      //Don't use 'extends'  } |

* Interfaces should contain only abstract methods. Interfaces should not contain a single concrete method.

[?](http://javaconceptoftheday.com/interfaces-in-java/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | interface InterfaceClass  {      abstract void abstractMethodOne();  //abstract method        abstract void abstractMethodTwo();  //abstract method        void concreteMethod()      {          //Compile Time Error.          //Concrete Methods are not allowed in interface      }  } |

* Interface can have two types of members.  **1) Fields     2) Abstract Methods.**

[?](http://javaconceptoftheday.com/interfaces-in-java/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | interface InterfaceClass  {      int i = 0;      //Field        abstract void abstractMethodOne();  //abstract method        abstract void abstractMethodTwo();  //abstract method  } |

* By default, Every field of an interface is public, static and final (we will discuss about final keyword Later). You can’t use any other modifiers other than these three for a field of an interface.

[?](http://javaconceptoftheday.com/interfaces-in-java/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | interface InterfaceClass  {      int i = 0;      //By default, field is public, static and final        //Following statements give compile time errors        private double d = 10;      protected long l = 15;        //You can't use any other modifiers other than public, static and final  } |

* You can’t change the value of a field once they are initialized. Because they are static and final. Therefore, sometimes fields are called as **Constants**. (We will discuss this feature in detail while covering ‘final’ keyword)

[?](http://javaconceptoftheday.com/interfaces-in-java/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | interface InterfaceClass  {      int i = 0;  }    class AnyClass implements InterfaceClass  {      void methodOne()      {          //Following statement gives compile time error.            InterfaceClass.i = 10;            //final field can not be re-assigned      }  } |

* By default, All methods of an interface are public and abstract.

[?](http://javaconceptoftheday.com/interfaces-in-java/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | interface InterfaceClass  {      void abstractMethodOne();  //Abstract method        void abstractMethodTwo();  //Abstract Method            //No need to use abstract keyword,          //by default methods are public and abstract  } |

* Like classes, for every interface .class file will be generated after compilation.
* While implementing any interface methods inside a class, that method must be declared as public. Because, according to [method overriding](http://javaconceptoftheday.com/method-overriding-java/) rule, you can’t reduce visibility of super class method. By default, every member of an interface is public and while implementing you should not reduce this visibility.

[?](http://javaconceptoftheday.com/interfaces-in-java/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | interface InterfaceClass  {      void methodOne();  }    class AnyClass implements InterfaceClass  {      void methodOne()      {          //It gives compile time error.          //Interface methods must be implemented as public      }  } |

* By default, Interface itself is not public but by default interface itself is abstract like below,

[?](http://javaconceptoftheday.com/interfaces-in-java/)

|  |  |
| --- | --- |
| 1  2  3  4  5 | abstract interface InterfaceClass  {      //By default interface is abstract      //No need to use abstract keyword  } |

* [SIB](http://javaconceptoftheday.com/static-members-java/) – Static Initialization Block and [IIB](http://javaconceptoftheday.com/instance-initialization-block-in-java/) – Instance Initialization Block are not allowed in interfaces.

[?](http://javaconceptoftheday.com/interfaces-in-java/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | interface InterfaceClassOne  {      static      {          //compile time error          //SIB's are not allowed in interfaces      }        {          //Here also compile time error.          //IIB's are not allowed in interfaces      }        void methodOne();  //abstract method  } |

* As we all know that, any class in java can not extend more than one class. But class can implement more than one interfaces. This is how **multiple inheritance** is implemented in java.

[?](http://javaconceptoftheday.com/interfaces-in-java/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | interface InterfaceClassOne  {      void methodOne();  }    interface InterfaceClassTwo  {      void methodTwo();  }    class AnyClass implements InterfaceClassOne, InterfaceClassTwo  {      public void methodOne()      {          //method of first interface is implemented      }        //method of Second interface must also be implemented.      //Otherwise, you have to declare this class as abstract.        public void methodTwo()      {          //Now, method of Second interface is also implemented.          //No need to declare this class as abstract      }  } |

## Does an interface extend Object class in java.?

You may have come across this question while reading about interfaces in java. You may also know that only classes in java are inherited from java.lang.Object class. Interfaces in java don’t inherit from Object class. They don’t have default parent like classes in java. But, following two cases may surprise you.

**Case 1 :**

If an interface does not extend Object class, then why we can call methods of Object class on interface variable like below.

[?](http://javaconceptoftheday.com/interface-extend-object-class-java/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | interface A  {    }    class InterfaceAndObjectClass  {      public static void main(String[] args)      {          A a = null;            a.equals(null);            a.hashCode();            a.toString();      }  } |

**Case 2 :**

If an interface does not extend Object class, then why the methods of Object class are visible in interface.?

[?](http://javaconceptoftheday.com/interface-extend-object-class-java/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | interface A  {      @Override      public boolean equals(Object obj);        @Override      public int hashCode();        @Override      public String toString();  } |

**Explanation :**

This is because, for every public method in Object class, there is an implicit abstract and public method declared in every interface which does not have direct super interfaces. This is the standard Java Language Specification which states like this,

“If an interface has no direct superinterfaces, then the interface implicitly declares a public abstract member method m with signature s, return type r, and throws clause tcorresponding to each public instance method m with signature s, return type r, and throws clause t declared in Object, unless a method with the same signature, same return type, and a compatible throws clause is explicitly declared by the interface.”

## Java Interview Questions On Interfaces :

**1) Can interfaces have constructors, SIB and IIB?**

No. Interfaces can’t have constructors, SIB and IIB. They show 100% abstractness.

**2) Can we re-assign a value to a field of interfaces?**

No. The fields of interfaces are static and final by default. They are just like constants. You can’t change their value once they got.

**3) Can we declare an Interface with “abstract” keyword?**

Yes, we can declare an interface with “abstract” keyword. But, there is no need to write like that. All interfaces in java are abstract by default.

**4) For every Interface in java, .class file will be generated after compilation. True or false?**

True. .class file will be generated for every interface after compilation.

**5) Can we override an interface method with visibility other than public?**

No. While overriding any interface methods, we should use public only. Because, all interface methods are public by default and you should not reduce the visibility while overriding them.

**6) Can interfaces become local members of the methods?**

No. You can’t define interfaces as local members of methods like local inner classes. They can be a part of top level class or interface.

**7) Can an interface extend a class?**

No, a class can not become super interface to any interface. Super interface must be an interface. That means, interfaces don’t extend classes but can extend other interfaces.

**8) Like classes, does interfaces also extend Object class by default?**

No. Interfaces don’t extend Object class.

**9) Can interfaces have static methods?**

No. Interfaces can’t have static methods.

**10) Can an interface have a class or another interface as it’s members?**

Yes. Interfaces can have classes or interfaces as their members.

[Java Practice Coding Questions On Interfaces](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/)

**1) For every interface written in a java file, .class file will be generated after compilation? True or False?**

[**View Answer**](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/#collapse1)

**Answer :**  
True. For every interface written in a java file, .class file will be generated after compilation.

**2) Can you identify the error in the below code?**

[?](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/)

|  |  |
| --- | --- |
| 1  2  3  4 | interface A  {      private int i;  } |

[**View Answer**](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/#collapse2)

**Answer :**  
Illegal modifier for field i. Only public, static and final are allowed.

**3) What will be the output of the following program?**

[?](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | interface A  {      void myMethod();  }    class B  {      public void myMethod()      {          System.out.println("My Method");      }  }    class C extends B implements A  {    }    class MainClass  {      public static void main(String[] args)      {          A a = new C();            a.myMethod();      }  } |

[**View Answer**](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/#collapse3)

**Answer :**  
My Method

**4) Can a class implement more than one interfaces?**

[**View Answer**](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/#collapse4)

**Answer :**  
Yes, a class can implement more than one interfaces.

**5) Why the below code is showing compile time error?**

[?](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | interface X  {      void methodX();  }    class Y implements X  {      void methodX()      {          System.out.println("Method X");      }  } |

[**View Answer**](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/#collapse5)

**Answer :**  
Interface methods must be implemented as public. Because, interface methods are public by default and you should not reduce the visibility of any methods while overriding.

**6) Does below code compile successfully? If not, why?**

[?](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | interface A  {      int i = 111;  }    class B implements A  {      void methodB()      {          i = 222;      }  } |

[**View Answer**](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/#collapse6)

**Answer :**  
No, because interface fields are static and final by default and you can’t change their value once they are initialized. In the above code, methodB() is changing value of interface field A.i. It shows compile time error.

**7) Is the following code written correctly?**

|  |
| --- |
| class A  {      //Class A  }    interface B extends A  {      //Interface B extending Class A  } |

[**View Answer**](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/#collapse7)

**Answer :**  
No. An interface can extend another interface not the class.

**8) What will be the output of the following program?**

[?](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | interface P  {      String p = "PPPP";        String methodP();  }    interface Q extends P  {      String q = "QQQQ";        String methodQ();  }    class R implements P, Q  {      public String methodP()      {          return q+p;      }        public String methodQ()      {          return p+q;      }  }    public class MainClass  {      public static void main(String[] args)      {          R r = new R();            System.out.println(r.methodP());            System.out.println(r.methodQ());      }  } |

[**View Answer**](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/#collapse8)

**Answer :**  
QQQQPPPP  
PPPPQQQQ

**9) Can interfaces have constructors?**

[**View Answer**](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/#collapse9)

**Answer :**  
No. Interfaces can’t have constructors.

**10) Is the below program written correctly? If yes, what will be the output?**

[?](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | class A implements B  {      public int methodB(int i)      {          return i =+ i \* i;      }  }    interface B  {      int methodB(int i);  }    public class MainClass  {      public static void main(String[] args)      {          B b = new A();            System.out.println(b.methodB(2));      }  } |

[**View Answer**](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/#collapse10)

**Answer :**  
Yes, program is written correctly. Output will be,  
4

**11) Can you find out the errors in the following code?**

[?](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | interface A  {      {          System.out.println("Interface A");      }        static      {          System.out.println("Interface A");      }  } |

[**View Answer**](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/#collapse11)

**Answer :**  
Interfaces can’t have initializers.

**12) How do you access interface field ‘i’ in the below code?**

[?](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class P  {      interface Q      {          int i = 111;      }  } |

[**View Answer**](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/#collapse12)

**Answer :**  
P.Q.i

**13) Like classes in java, Interfaces also extend java.lang.Object class by default. True OR False?**

[**View Answer**](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/2/#collapse13)

**Answer :**  
False. Interfaces don’t extend Object class.

**14) Does below program compile successfully?**

[?](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/2/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | interface ABC  {      public void methodOne();        public void methodTwo();  }    interface PQR extends ABC  {      public void methodOne();        public void methodTwo();  } |

[**View Answer**](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/2/#collapse14)

**Answer :**  
Yes, program compiles successfully.

**15) Can interfaces have static methods?**

[**View Answer**](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/2/#collapse15)

**Answer :**  
Yes, from Java 8, interfaces can have static methods.

**16) Is the following program written correctly? If yes, what will be the output?**

[?](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/2/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | interface ABC  {      void methodOne();  }    interface PQR extends ABC  {      void methodTwo();  }    abstract class XYZ implements PQR  {      public void methodOne()      {          methodTwo();      }  }    class MNO extends XYZ  {      public void methodTwo()      {          methodOne();      }  }    public class MainClass  {      public static void main(String[] args)      {          ABC abc = new MNO();            abc.methodOne();      }  } |

[**View Answer**](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/2/#collapse16)

**Answer :**  
Yes, program is written is correctly. But, it will throw StackOverflowError at run time. Because, methodOne() and methodTwo() are cyclicly called.

**17) What will be the output of the following program?**

[?](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/2/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | interface X  {      char c = 'A';        char methodX();  }    class Y implements X  {      {          System.out.println(c);      }        public char methodX()      {          char c = this.c;            return ++c;      }  }    public class MainClass  {      public static void main(String[] args)      {          Y y = new Y();            System.out.println(y.methodX());            System.out.println(y.c);            System.out.println(X.c);      }  } |

[**View Answer**](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/2/#collapse17)

**Answer :**  
A  
B  
A  
A

**18) Can you identify the error in the below code?**

[?](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/2/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | interface A  {      void methodA();  }    class B implements A  {      public void methodA()      {          interface C          {              int i = 123;          }      }  } |

[**View Answer**](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/2/#collapse18)

**Answer :**  
Interfaces can’t be local members of a method.

**19) Can we declare an interface as ‘abstract’?**

[**View Answer**](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/2/#collapse19)

**Answer :**  
Yes, interfaces can be declared as ‘abstract’. But, there is no need to declare like that because interfaces are ‘abstract’ by default.

**20) What will be the output of the following program?**

[?](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/2/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | interface One  {      String s = "FINAL";        String methodONE();  }    interface Two  {      String methodONE();  }    abstract class Three  {      String s = "NOT FINAL";        public abstract String methodONE();  }    class Four extends Three implements One, Two  {      public String methodONE()      {          String s = super.s + One.s;            return s;      }  }    public class MainClass  {      public static void main(String[] args)      {          Four four = new Four();            System.out.println(four.methodONE());            One one = four;            System.out.println(one.s);      }  } |

[**View Answer**](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/2/#collapse20)

**Answer :**  
NOT FINALFINAL  
FINAL

**21) What will be the output of the below program?**

[?](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/2/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | interface X  {      void method();  }    class Y  {      public void method()      {          System.out.println("CLASS Y");      }  }    class Z extends Y implements X  {    }    public class MainClass  {      public static void main(String[] args)      {          X x = new Z();            x.method();      }  } |

[**View Answer**](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/2/#collapse21)

**Answer :**  
CLASS Y

**22) Can interfaces have methods other than abstract?**

[**View Answer**](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/2/#collapse22)

**Answer :**  
Yes, from Java 8, interfaces can have static methods and default methods other than abstract methods.

**23) What will be the output of the following program?**

[?](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/2/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56 | interface A  {      int methodA();  }    interface B  {      int methodB();  }    interface C  {      int methodC();  }    class D implements A, B, C  {      int i = 999+111;        public int methodA()      {          i =+ i / i;            return i;      }        public int methodB()      {          i =- i \* i;            return i;      }        public int methodC()      {          i = ++i - --i;            return i;      }  }    public class MainClass  {      public static void main(String[] args)      {          D d = new D();            System.out.println(d.i);            System.out.println(d.methodA());            System.out.println(d.methodB());            System.out.println(d.methodC());      }  } |

[**View Answer**](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/2/#collapse23)

**Answer :**  
1110  
1  
-1  
1

**24) How do you print the value of field ‘i’ of interface ‘OneTwoThree’ in the below example and what will be the it’s value?**

[?](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/2/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | interface One  {      int i = 222;        interface OneTwo      {          int i = One.i+One.i;            interface OneTwoThree          {              int i = OneTwo.i + OneTwo.i;          }      }  } |

[**View Answer**](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/2/#collapse24)

**Answer :**  
Printing ‘i’ value —> System.out.println(One.OneTwo.OneTwoThree.i)  
Value of One.OneTwo.OneTwoThree.i will be 888.

**25) All members of interface are public by default. True or false?**

[**View Answer**](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/3/#collapse25)

**Answer :**  
True.

**26) What will be the output of the following program?**

[?](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/3/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54 | interface A  {      String A = "AAA";        String methodA();  }    interface B  {      String B = "BBB";        String methodB();  }    class C implements A, B  {      public String methodA()      {          return A+B;      }        public String methodB()      {          return B+A;      }  }    class D extends C implements A, B  {      String D = "DDD";        public String methodA()      {          return D+methodB();      }  }    public class MainClass  {      public static void main(String[] args)      {          C c = new C();            System.out.println(c.methodA());            System.out.println(c.methodB());            c = new D();            System.out.println(c.methodA());            System.out.println(c.methodB());      }  } |

[**View Answer**](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/3/#collapse26)

**Answer :**  
AAABBB  
BBBAAA  
DDDBBBAAA  
BBBAAA

**27) Is the below program written correctly? If yes, what will be the output?**

[?](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/3/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47 | interface X  {      void methodX();        interface Y      {          void methodY();      }  }    class Z implements X, X.Y  {      {          methodX();            System.out.println(1);      }        public void methodX()      {          methodY();            System.out.println(2);      }        public void methodY()      {          System.out.println(3);      }  }      public class MainClass  {      public static void main(String[] args)      {          Z z = new Z();            z.methodX();            z.methodY();            X x = z;            x.methodX();      }  } |

[**View Answer**](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/3/#collapse27)

**Answer :**  
Yes, program is correct. Output will be,  
3  
2  
1  
3  
2  
3  
3  
2

**28) Can you identify the error in the below code?**

[?](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/3/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class A implements A.B  {      static interface B      {          void methodB();      }  } |

[**View Answer**](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/3/#collapse28)

**Answer :**  
Cycle detected. Any class can not extend itself or it’s member types.

**29) Interfaces are abstract and public by default. True or False?**

[**View Answer**](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/3/#collapse29)

**Answer :**  
False. Interfaces are abstract by default but not public.

**30) Can you identify the error in the below code?**

|  |
| --- |
| interface X  {      void methodX();  }    interface Y extends X  {      void methodY();  }    class Z implements Y  {      public void methodY()      {          System.out.println("Method Y");      }  } |

[**View Answer**](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/3/#collapse30)

**Answer :**  
Class Z must implement methodX() also.

**31) Can we define interfaces as generic?**

[**View Answer**](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/3/#collapse31)

**Answer :**  
Yes, we can define generic interfaces.

**32) What will be the output of the following program?**

[?](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/3/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | abstract class A  {      abstract void myMethod(Number N);  }    interface B  {      abstract void myMethod(Object O);  }    class C extends A implements B  {      void myMethod(Number N)      {          System.out.println("Number");      }        public void myMethod(Object O)      {          System.out.println("Object");      }  }    public class MainClass  {      public static void main(String[] args)      {          A a = new C();            a.myMethod(new Integer(121));            B b = new C();            b.myMethod(new Integer(121));            C c = new C();            c.myMethod(new Integer(121));      }  } |

[**View Answer**](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/3/#collapse32)

**Answer :**  
Number  
Object  
Number

**33) Is the below program written correctly? If yes, what will be the output?**

[?](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/3/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | interface I  {      class C      {          int i;            public C(int i)          {              this.i = ++i;          }            int methodC()          {              return ++i;          }      }  }    public class MainClass  {      public static void main(String[] args)      {          I.C c = new I.C(000);            System.out.println(c.methodC());      }  } |

[**View Answer**](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/3/#collapse33)

**Answer :**  
Yes, program is written correctly. Output will be,  
2

**34) What will be the output of the following program?**

[?](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/3/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | class A { }    class B extends A { }    class C extends B { }    interface ABC  {      void method(A a);  }    interface PQR  {      void method(B b);  }    class M implements ABC, PQR  {      public void method(A a)      {          System.out.println(2);      }        public void method(B b)      {          System.out.println(3);      }  }    public class MainClass  {      public static void main(String[] args)      {          M m = new M();            m.method(new A());            m.method(new B());            m.method(new C());      }  } |

[**View Answer**](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/3/#collapse34)

**Answer :**  
2  
3  
3

**35) Can you identify the errors in the below code?**

[?](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/3/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | interface I  {      class C implements I      {          public void methodI(int i)          {              System.out.println(i);          }      }        void methodI(int i);  } |

[**View Answer**](http://javaconceptoftheday.com/java-practice-coding-questions-on-interfaces/3/#collapse35)

**Answer :**  
No errors.