# Why do interfaces extend Object, according to the class file format?



Why does the JVM specification state that interfaces must have a `super_class` of `java/lang/Object`, even though interfaces do not extend `java/lang/Object`?

I'm specifically referring to §4.1 of the JVM spec, where it says:

> For an interface, the value of the super_class item must always be a valid index into the constant_pool table. The constant_pool entry at that index must be a CONSTANT_Class_info structure representing the class Object.

yet in §9.2 of the JLS, it says that interfaces do not extend Object. Instead a implicitly created abstract method is declared which matches each public method in the Object class:

> 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 t corresponding 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 jvm jls

asked Apr 27 '13 at 17:39

Phil K

**2,086** 16 39

## 2 Answers

As mentioned in §9.2 :

> 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 t corresponding 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.

Hence , we see that , Although an interface having no direct superinterface doesn't explicitly extends `Object` but still it has a link with `Object` class internally as it is used by the compiler to insert abstract methods with same signature and return type and throws clause as that of public methods in `Object` class, within the interface. That's why For an interface, the value of the super_class item must always be a valid index into the constant_pool table. The constant_pool entry at that index must be a CONSTANT_Class_info structure representing the class **Object**. This is the reason that an interface reference variable can successfully call public instance methods for example `toString()` method of `Object` . For example, consider the code given below:

```
interface MyInterface
{}
public class InterfaceTest implements MyInterface
{
    public static void main(String[] args)
    {
        MyInterface mInterface = new InterfaceTest();
        System.out.println(mInterface.toString());//Compiles successfully.
Although toString() is not declared within MyInterface
    }
}
```

The above code compiles successfully even though `toString()` method (Which is the method of `Object` ) is not declared within `MyInterface` . Above code is providing following output on my System:

```
InterfaceTest@1ba34f2
```

The output may vary from system to system..

edited Apr 27 '13 at 18:22

answered Apr 27 '13 at 18:02

Vishal K

**11.5k** 1 16 34

What you see in the JVM spec is basically the concrete implementation of the behavior specified by the JLS - just like classes implement interfaces and have implementation details.

answered Apr 27 '13 at 18:22

Michael Borgwardt
**284k** 59 414 652