





Get started


# What is the difference between an interface and abstract class?

Ask Question

What exactly is the difference between an interface and abstract class?

oop interface abstract-class

edited Feb 2 '17 at 17:36

 **brice**  
15.7k 5 64 84

asked Dec 16 '09 at 8:15

 **Sarfraz**  
289k 58 454 540

- 73 This is an extremely common interview question. It's surprising since an abstract class is rarely used in solutions compared to other things. Your question has helped me Sarfraz. – Catto Feb 28 '14 at 2:34
- 4 This question might also help to understand the concept of interfaces [stackoverflow.com/q/8531292/1055241](#) – gprathour Jul 7 '14 at 5:28
- 3 I've removed the PHP tag from this question, since almost none of the answer are language specific, and the question itself is not language specific. – brice Feb 2 '17 at 17:41

32 Answers

1 2 next

## Interfaces

An interface is a **contract**: The person writing the interface says, "*hey, I accept things looking that way*", and the person using the interface says "*OK, the class I write looks that way*".

**An interface is an empty shell.** There are only the signatures of the methods, which implies that the methods do not have a body. The interface can't do anything. It's just a pattern.

For example (pseudo code):

```
// I say all motor vehicles should look like this:
interface MotorVehicle
{
    void run();

    int getFuel();
}

// My team mate complies and writes vehicle looking that way
class Car implements MotorVehicle
{
    int fuel;

    void run()
    {
        print("Wrrroooooooooom");
    }

    int getFuel()
    {
        return this.fuel;
    }
}
```

Implementing an interface consumes very little CPU, because it's not a class, just a bunch of names, and therefore there isn't any expensive look-up to do. It's great when it matters, such as in embedded devices.

## Abstract classes

Abstract classes, unlike interfaces, are classes. They are more expensive to use, because there is a look-up to do when you inherit from them.

Abstract classes look a lot like interfaces, but they have something more: You can define a behavior for them. It's more about a person saying, "these classes should look like that, and they have that in common, so fill in the blanks!".

For example:

```
// I say all motor vehicles should look like this:
abstract class MotorVehicle
{
    int fuel;

    // They ALL have fuel, so lets implement this for everybody.
    int getFuel()
    {
        return this.fuel;
    }

    // That can be very different, force them to provide their
    // own implementation.
    abstract void run();
}

// My teammate complies and writes vehicle looking that way
class Car extends MotorVehicle
{
    void run()
    {
        print("Wrrroooooooooom");
    }
}
```

## Implementation

While abstract classes and interfaces are supposed to be different concepts, the implementations make that statement sometimes untrue. Sometimes, they are not even what you think they are.

In Java, this rule is strongly enforced, while in PHP, interfaces are abstract classes with no method declared.

In Python, abstract classes are more a programming trick you can get from the ABC module and is actually using metaclasses, and therefore classes. And interfaces are more related to duck typing in this language and it's a mix between conventions and special methods that call descriptors (the `__method__` methods).

As usual with programming, there is theory, practice, and practice in another language :-)

edited May 16 at 2:56



Dave S

875 1 8 19

answered Dec 16 '09 at 8:37



e-satis

337k 96 260 305


- 5

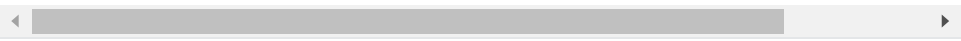
The key point about interfaces is not so much that they say what a class does, but allow objects that can Wizzle to make themselves useful to code that needs a Wizzler. Note that in many cases neither the person who writes the thing that can Wizzle, nor the person who needs a Wizzler, will be the person who writes the interface. – [supercat](#) Mar 27 '13 at 21:28
- 133

I don't think that CPU consumption is the highlight-worthy point on interfaces. – [Dan Lugg](#) Sep 11 '13 at 21:40
- 27

@e-satis With Java 8, you can define default methods in interfaces which is the equivalent of having non-abstract methods in abstract classes. With this addition, I can no longer see the real difference between abstract classes and interface besides the fact that I should use interfaces because classes can implement multiple interfaces but can only inherit one class – [Ogen](#) Oct 31 '15 at 1:34
- 6

I wanted to give you more than one upvote for your explanation... – [Ashish Shukla](#) May 24 '16 at 9:58
- 10

I think the comparison between `interface` and `class` from Head First Java is vivid that A class defines who you are, and an interface tells what roles you could play – [AnnabellChan](#) Jul 11 '16 at 6:43 



Get personalized job matches now



The key technical differences between an [abstract class](#) and an [interface](#) are:

- Abstract classes can have *constants*, *members*, *method stubs* (*methods without a body*) and *defined methods*, whereas interfaces can only have *constants* and *methods stubs*.
- Methods and members of an abstract class can be defined with *any visibility*, whereas all methods of an interface must be defined as `public` (they are defined public by default).

- When inheriting an abstract class, a *concrete* child class *must define the abstract methods*, whereas an abstract class can extend another abstract class and abstract methods from the parent class don't have to be defined.
- Similarly, an interface extending another interface is *not responsible for implementing methods* from the parent interface. This is because interfaces cannot define any implementation.
- A child class can only *extend a single class* (abstract or concrete), whereas an interface can extend or a class can *implement multiple other interfaces*.
- A child class can define abstract methods with the *same or less restrictive visibility*, whereas a class implementing an interface must define the methods with the exact same visibility (public).

edited Nov 20 '16 at 14:14



Peter Mortensen

12.9k ● 19 ● 83 ● 111

answered Dec 16 '09 at 10:11



Justin Johnson

26.3k ● 7 ● 53 ● 84

- 109
- i think this is the best answer because it highlights all of the key differences. an example's not really necessary. – Joshua K Jul 10 '11 at 18:01
- 4
- And normally with classes you can instantiate an object from it unlike the abstract classes which CANNOT be instantiated. – SASM Jul 9 '13 at 20:48
- 3
- "When inheriting an abstract class, the child class must define the abstract methods, whereas an interface can extend another interface and methods don't have to be defined." - This is not true. Just as an interface can extend an interface without defining methods, an abstract class can inherit an abstract class without defining methods. – Nick Mar 10 '14 at 17:21

An Interface contains only the definition / signature of functionality, and if we have some common functionality as well as common signatures, then we need to use an abstract class. By using an abstract class, we can provide behavior as well as functionality both in the same time. Another developer inheriting abstract class can use this functionality easily, as they would only need to fill in the blanks.

oops interface vs abstract class

Interface	Abstract class
Interface support multiple inheritance	Abstract class does not support multiple inheritance
Interface does'n Contains Data Member	Abstract class contains Data Member
Interface does'n contains Cunstructors	Abstract class contains Cunstructors
An interface Contains only incomplete member (signature of member)	An abstract class Contains both incomplete (abstract) and complete member
An interface cannot have access modifiers by default everything is assumed as public	An abstract class can contain access modifiers for the subs, functions, properties
Member of interface can not be Static	Only Complete Member of abstract class can be Static

Taken from:

<http://www.dotnetbull.com/2011/11/difference-between-abstract-class-and.html>

<http://www.dotnetbull.com/2011/11/what-is-abstract-class-in-c-net.html>

<http://www.dotnetbull.com/2011/11/what-is-interface-in-c-net.html>

edited Jun 30 '16 at 17:42



ragingasiancoder

605 ● 4 ● 16

answered Sep 15 '13 at 8:59



Vivek

3,602 ● 2 ● 12 ● 8

- 10
- You need to say what language this applies to ("Abstract class does not support multiple inheritance" is far from being universally true) – Ben Voigt Mar 7 '14 at 4:16
- 1
- Typo it not Cunstructor .. Its Constructor.. – Pra Jazz Jul 18 '14 at 10:57
- 6
- Member of the interface must be static final . Last statement is wrong. – Xar E Ahmer Aug 5 '14 at 8:21
- 1
- What is the targetted programming language here? C#? – Peter Mortensen Nov 20 '16 at 14:26

An explanation can be found here:

<http://www.developer.com/lang/php/article.php/3604111/PHP-5-OOP->


An abstract class is a class that is only partially implemented by the programmer. It may contain one or more abstract methods. An abstract method is simply a function definition that serves to tell the programmer that the method must be implemented in a child class.

An interface is similar to an abstract class; indeed interfaces occupy the same namespace as classes and abstract classes. For that reason, you cannot define an interface with the same name as a class. An interface is a fully abstract class; none of its methods are implemented and instead of a class sub-classing from it, it is said to implement that interface.

Anyway I find this explanation of interfaces somewhat confusing. A more common definition is: *An interface defines a contract that implementing classes must fulfill. An interface definition consists of signatures of public members, without any implementing code.*

edited Dec 16 '09 at 8:24

answered Dec 16 '09 at 8:18

 **Konamiman**  
41.3k ● 14 ● 92 ● 123

- 4 This is the most correct answer, since PHP interfaces differ from other languages in that PHP interfaces ARE abstract classes under the hood, whereas other languages' interfaces are signatures that classes must match. They behave the same as long as there are no errors though. – [Tor Valamo](#) Dec 16 '09 at 8:55
- 1 True, for PHP it's the real best anwser. But it's harder to get from the text blob than from a simple snippet. – [e-satis](#) Dec 16 '09 at 12:15

Some important differences:

In the form of a table:

Interface	Abstract
Java interface are implicitly abstract and cannot have implementations	A Java abstract class can have instance methods that implements a default behavior
Variables declared in a Java interface is by default final	An abstract class may contain non-final variables.
Members of a Java interface are public by default	A Java abstract class can have the usual flavors of class members like private, protected, etc
Java interface should be implemented using keyword “implements”	A Java abstract class should be extended using keyword “extends”
An interface can extend another Java interface only	an abstract class can extend another Java class and implement multiple Java interfaces.
Interface is absolutely abstract and cannot be instantiated	A Java abstract class also cannot be instantiated, but can be invoked if a main() exists.
java interfaces are slow as it requires extra indirection	Comparatively fast


As [stated by Joe from javapapers](#):

- 1.Main difference is methods of a Java interface are implicitly abstract and cannot have implementations. A Java abstract class can have instance methods that implements a default behavior.
- 2.Variables declared in a Java interface is by default final. An abstract class may contain non-final variables.
- 3.Members of a Java interface are public by default. A Java abstract class can have the usual flavors of class members like private, protected, etc..
- 4.Java interface should be implemented using keyword “implements”; A Java abstract class should be extended using keyword “extends”.
- 5.An interface can extend another Java interface only, an abstract class can extend another Java class and implement multiple Java interfaces.
- 6.A Java class can implement multiple interfaces but it can extend only one abstract class.
- 7.Interface is absolutely abstract and cannot be instantiated; A Java abstract class also cannot be instantiated, but can be invoked if a main() exists.
- 8.In comparison with java abstract classes, java interfaces are slow as it requires extra indirection.

edited Oct 14 '13 at 19:53




**Brad Larson** ♦



160k ● 40 ● 360 ● 537

answered May 16 '13 at 5:55



softmage99

612 ● 1 ● 8 ● 12

- 3 I've edited your answer to provide correct attribution. You can't just drop a link at the bottom of your answer. You need to quote all of the language that was copied from another source, as well. Also, if that table was drawn from somewhere, you should clearly indicate where that is from. – Brad Larson ♦ Oct 14 '13 at 19:54
- 2 With Java 8, the differences are less now. Check updated differences here: [journaldev.com/1607/...](http://journaldev.com/1607/...) – Pankaj Jul 15 '14 at 17:28

I don't want to highlight the differences, which have been already said in many answers ( regarding public static final modifiers for variables in interface & support for protected, private methods in abstract classes)

In simple terms, I would like to say:

*interface*: To implement a contract by multiple unrelated objects

*abstract class*: To implement the same or different behaviour among multiple related objects

From the Oracle [documentation](#)

Consider using abstract classes if :

1. You want to share code among several closely related classes.
2. You expect that classes that extend your abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).
3. You want to declare non-static or non-final fields.

Consider using interfaces if :

1. You expect that unrelated classes would implement your interface. For example,many unrelated objects can implement `Serializable` interface.
2. You want to specify the behaviour of a particular data type, but not concerned about who implements its behaviour.
3. You want to take advantage of multiple inheritance of type.

*abstract class establishes "is a" relation with concrete classes. interface provides "has a" capability for classes.*

If you are looking for `Java` as programming language, here are a few more updates:


Java 8 has reduced the gap between `interface` and `abstract` classes to some extent by providing a `default` method feature. *An interface does not have an implementation for a method* is no longer valid now.

Refer to this documentation [page](#) for more details.

Have a look at this SE question for code examples to understand better.

[How should I have explained the difference between an Interface and an Abstract class?](#)


edited May 23 '17 at 12:34



Community ♦

1 ● 1

answered Nov 27 '15 at 12:42



Ravindra babu

26.7k ● 5 ● 142 ● 128

The main point is that:

- **Abstract is object oriented.** It offers the basic data an 'object' should have and/or functions it should be able to do. It is concerned with the object's basic characteristics: what it has and what it can do. Hence objects which inherit from the same abstract class share the basic characteristics (generalization).
- **Interface is functionality oriented.** It defines functionalities an object should have. Regardless what object it is, as long as it can do these functionalities, which are defined in the interface, it's fine. It ignores everything else. An object/class can contain several (groups of) functionalities; hence it is possible for a class to implement multiple interfaces.

edited Jun 30 '16 at 17:48




ragingasiancoder

605 ● 4 ● 16



answered Sep 29 '14 at 9:54


Yusup

616 ● 6 ● 10

When you want to provide polymorphic behaviour in an inheritance hierarchy, use abstract classes.

When you want polymorphic behaviour for classes which are completely unrelated, use an interface.

edited Nov 20 '16 at 14:21

Peter Mortensen

12.9k ● 19 ● 83 ● 111

answered May 28 '12 at 4:42

sculptor

219 ● 2 ● 2

10 now to google polymorphic – Andrew Jan 16 '17 at 22:47

I am constructing a building of 300 floors

The building's blueprint **interface**

- For example, Servlet(I)

Building constructed up to 200 floors - partially completed---**abstract**

- Partial implementation, for example, generic and HTTP servlet

Building construction completed-**concrete**

- Full implementation, for example, own servlet

Interface


- We don't know anything about implementation, just requirements. We can go for an interface.
- Every method is public and abstract by default
- It is a 100% pure abstract class
- If we declare public we cannot declare private and protected
- If we declare abstract we cannot declare final, static, synchronized, strictfp and native
- Every interface has public, static and final
- Serialization and transient is not applicable, because we can't create an instance for in interface
- Non-volatile because it is final
- Every variable is static
- When we declare a variable inside an interface we need to initialize variables while declaring
- Instance and static block not allowed

Abstract

- Partial implementation
- It has an abstract method. An addition, it uses concrete
- No restriction for abstract class method modifiers
- No restriction for abstract class variable modifiers
- We cannot declare other modifiers except abstract
- No restriction to initialize variables


Taken from DurgaJobs Website

edited Apr 1 at 9:35

Chamod Pathirana

87 ● 10

answered Jul 9 '14 at 18:00

Jaichander

578 ● 1 ● 8 ● 21

2 I completely disagree with this view. The blueprint is a completely different concept to 'interface.' Blueprint is more analogous to a static model or design specification for a specific implementation. It is closer to 'class,' as the blueprint can be instantiated multiple times through its constructor, but even this is not close enough as the 'class' also contains the specification for how to construct (the ctor), and the means to do so. Interface as a concept is intended to represent some behavior, such as Heat Up/ Cool Down, that can be applied to range of things, eg: buildings, ovens, etc – Sentinel Sep 18 '17 at 7:48

Let's work on this question again:

The first thing to let you know is that 1/1 and 1\*1 results in the same, but it does not mean that multiplication and division are same. Obviously, they hold some good relationship, but mind you both are different.

I will point out main differences, and the rest have already been explained:

Abstract classes are useful for modeling a class hierarchy. At first glance of any requirement, we are partially clear on what **exactly** is to be built, but we know **what to build**. And so your abstract classes are your base classes.

Interfaces are useful for letting other hierarchy or classes to know that what I am capable of doing. And when you say I am capable of something, you must have that capacity. Interfaces will mark it as compulsory for a class to implement the same functionalities.

edited Nov 20 '16 at 14:18



Peter Mortensen

12.9k ● 19 ● 83 ● 111

answered Apr 11 '12 at 5:18



Dhananjay

2,556 ● 2 ● 16 ● 17

It's pretty simple actually.

You can think of an interface as a class which is only allowed to have abstract methods and nothing else.

So an interface can only "declare" and not define the behavior you want the class to have.

An abstract class allows you to do both declare (using abstract methods) as well as define (using full method implementations) the behavior you want the class to have.

And a regular class only allows you to define, not declare, the behavior/actions you want the class to have.

One last thing,

In Java, you can implement multiple interfaces, but you can only extend one (Abstract Class or Class)...

This means inheritance of defined behavior is restricted to only allow one per class... ie if you wanted a class that encapsulated behavior from Classes A,B&C you would need to do the following: Class A extends B, Class C extends A .. its a bit of a round about way to have multiple inheritance...

Interfaces on the other hand, you could simply do: interface C implements A, B

So in effect Java supports multiple inheritance only in "declared behavior" ie interfaces, and only single inheritance with defined behavior.. unless you do the round about way I described...

Hopefully that makes sense.

edited Mar 17 '17 at 22:31

answered Jul 22 '14 at 23:36



g00dnatur3

818 ● 6 ● 12

The only difference is that one can participate in multiple inheritance and other cannot.

The definition of an interface has changed over time. Do you think an interface just has method declarations only and are just contracts? What about static final variables and what about default definitions after Java 8?

Interfaces were introduced to Java because of [the diamond problem](#) with multiple inheritance and that's what they actually intend to do.

Interfaces are the constructs that were created to get away with the multiple inheritance problem and can have abstract methods, default definitions and static final variables.

See [Why does Java allow static final variables in interfaces when they are only intended to be contracts?](#).

edited Dec 5 '16 at 16:44

answered Aug 25 '14 at 17:53



Vivek Vermani  
1,558 ● 9 ● 33

The comparison of interface vs. abstract class is wrong. There should be two other comparisons instead: 1) **interface vs. class** and 2) **abstract vs. final class**.

## Interface vs Class

**Interface** is a contract between two objects. E.g., I'm a Postman and you're a Package to deliver. I expect you to know your delivery address. When someone gives me a Package, it has to know its delivery address:

```
interface Package {
    String address();
}
```

**Class** is a group of objects that obey the contract. E.g., I'm a box from "Box" group and I obey the contract required by the Postman. At the same time I obey other contracts:

```
class Box implements Package, Property {
    @Override
    String address() {
        return "5th Street, New York, NY";
    }
    @Override
    Human owner() {
        // this method is part of another contract
    }
}
```

## Abstract vs Final

**Abstract class** is a group of incomplete objects. They can't be used, because they miss some parts. E.g., I'm an abstract GPS-aware box - I know how to check my position on the map:

```
abstract class GpsBox implements Package {
    @Override
    public abstract String address();
    protected Coordinates whereAmI() {
        // connect to GPS and return my current position
    }
}
```

This class, if inherited/extended by another class, can be very useful. But by itself - it is useless, since it can't have objects. Abstract classes can be building elements of final classes.

**Final class** is a group of complete objects, which can be used, but can't be modified. They know exactly how to work and what to do. E.g., I'm a Box that always goes to the address specified during its construction:

```
final class DirectBox implements Package {
    private final String to;
    public DirectBox(String addr) {
        this.to = addr;
    }
    @Override
    public String address() {
        return this.to;
    }
}
```

In most languages, like Java or C++, it is possible to have **just a class**, neither abstract nor final. Such a class can be inherited and can be instantiated. I don't think this is strictly in line with object-oriented paradigm, though.

Again, comparing interfaces with abstract classes is not correct.

answered Jul 24 '14 at 21:57



yegor256  
53.1k ● 83 ● 352 ● 491

In short the differences are the following:

Syntactical Differences Between **Interface** and **Abstract Class**:

1. Methods and members of an abstract class can have any visibility. All methods of an **interface** must be **public**. *//Does not hold true from Java 9 anymore*
2. A concrete child class of an **Abstract** Class must define all the abstract methods. An **Abstract** child class can have abstract methods.



An *interface* extending another interface need not provide default implementation for methods inherited from the parent interface.

- 3. A child class can only extend a single class. An **interface** can extend multiple interfaces. A class can implement multiple interfaces.
- 4. A child class can define abstract methods with the same or less restrictive visibility, whereas class implementing an **interface** must define all interface methods as public.
- 5. **Abstract Classes** can have constructors but not **interfaces**.
- 6. Interfaces from Java 9 have private static methods.

In Interfaces now:

public static - supported  
public abstract - supported  
public default - supported  
private static - supported  
private abstract - compile error  
private default - compile error  
private - supported

edited Oct 27 '17 at 5:18

answered Jun 27 '17 at 1:43



Pritam Banerjee

9,651 ● 6 ● 41 ● 59

Interface: Turn ( Turn Left, Turn Right.)

Abstract Class: Wheel.

Class: Steering Wheel, derives from Wheel, exposes Interface Turn

One is for categorizing behavior that can be offered across a diverse range of things, the other is for modelling an ontology of things.

answered Mar 6 '17 at 21:46




Sentinel

2,048 ● 18 ● 29

Not really the answer to the original question, but once you have the answer to the difference between them, you will enter the when-to-use-each dilemma: [When to use interfaces or abstract classes? When to use both?](#)

I've limited knowledge of OOP, but seeing interfaces as an equivalent of an adjective in grammar has worked for me until now (correct me if this method is bogus!). For example, interface names are like attributes or capabilities you can give to a class, and a class can have many of them: `ISerializable`, `ICountable`, `IList`, `ICacheable`, `IHappy`, ...

edited May 23 '17 at 12:02



Community ♦

1 ● 1

answered Dec 16 '09 at 8:48



azkotoki

2,037 ● 1 ● 18 ● 25

If you have some common methods that can be used by multiple classes go for abstract classes. Else if you want the classes to follow some definite blueprint go for interfaces.

Following examples demonstrate this.

Abstract class in Java:

```
abstract class animals
{
    // They all love to eat. So let's implement them for everybody
    void eat()
    {
        System.out.println("Eating...");
    }
    // The make different sounds. They will provide their own implementation
    abstract void sound();
}

class dog extends animals
{
    void sound()
    {
        System.out.println("Woof Woof");
    }
}
```

```
class cat extends animals
{
    void sound()
    {
        System.out.println("Meoww");
    }
}
```

Following is an implementation of interface in Java:

```
interface Shape
{
    void display();
    double area();
}

class Rectangle implements Shape
{
    int length, width;
    Rectangle(int length, int width)
    {
        this.length = length;
        this.width = width;
    }
    @Override
    public void display()
    {
        System.out.println("****\n* *\n* *\n****");
    }
    @Override
    public double area()
    {
        return (double)(length*width);
    }
}

class Circle implements Shape
{
    double pi = 3.14;
    int radius;
    Circle(int radius)
    {
        this.radius = radius;
    }
    @Override
    public void display()
    {
        System.out.println("O"); // :P
    }
    @Override
    public double area()
    {
        return (double)((pi*radius*radius)/2);
    }
}
```

Some Important Key points in a nutshell:

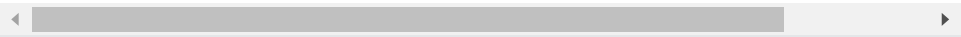
1. The variables declared in Java interface are by default final. Abstract classes can have non-final variables.
2. The variables declared in Java interface are by default static. Abstract classes can have non-static variables.
3. Members of a Java interface are public by default. A Java abstract class can have the usual flavors of class members like private, protected, etc..

answered Oct 3 '17 at 6:25



[Pransh Tiwari](#)

639 ● 3 ● 17



Inheritance is used for two purposes:

- To allow an object to regard parent-type data members and method implementations as its own.
- To allow a reference to an objects of one type to be used by code which expects a reference to supertype object.

In languages/frameworks which support generalized multiple inheritance, there is often little need to classify a type as either being an "interface" or an "abstract class". Popular languages and frameworks, however, will allow a type to regard one other type's data members or method implementations as its own even though they allow a type to be substitutable for an arbitrary number of other types.

Abstract classes may have data members and method implementations, but can only be inherited by classes which don't inherit from any other classes. Interfaces put almost no restrictions on the types which implement them, but cannot include any data members or method implementations.

There are times when it's useful for types to be substitutable for many different things; there are other times when it's useful for objects to regard parent-type data members and method implementations as their own. Making a distinction between interfaces and abstract classes allows each of those abilities to be used in cases where it is most relevant.

answered Sep 3 '13 at 16:09

Key Points:

- Abstract class can have property, Data fields ,Methods (complete / incomplete) both.
- If method or Properties define in abstract keyword that must override in derived class.(its work as a tightly coupled functionality)
- If define abstract keyword for method or properties in abstract class you can not define body of method and get/set value for properties and that must override in derived class.
- Abstract class does not support multiple inheritance.
- Abstract class contains Constructors.
- An abstract class can contain access modifiers for the subs, functions, properties.
- Only Complete Member of abstract class can be Static.
- An interface can inherit from another interface only and cannot inherit from an abstract class, where as an abstract class can inherit from another abstract class or another interface.

Advantage:

- It is a kind of contract that forces all the subclasses to carry on the same hierarchies or standards.
- If various implementations are of the same kind and use common behavior or status then abstract class is better to use.
- If we add a new method to an abstract class then we have the option of providing default implementation and therefore all the existing code might work properly.
- Its allow fast execution than interface.(interface Requires more time to find the actual method in the corresponding classes.)
- It can use for tight and loosely coupling.

find details here...

<http://pradeepatkari.wordpress.com/2014/11/20/interface-and-abstract-class-in-c-oops/>

edited Dec 5 '14 at 10:00

answered Nov 20 '14 at 7:55



Pradeep atkari  
280 ● 1 ● 4 ● 13

The shortest way to sum it up is that an `interface` is:

1. Fully abstract, apart from `default` and `static` methods; while it has definitions (method signatures + implementations) for `default` and `static` methods, it only has declarations (method signatures) for other methods.
2. Subject to laxer rules than classes (a class can implement multiple `interface S`, and an `interface` can inherit from multiple `interface S`). All variables are implicitly constant, whether specified as `public static final` or not. All members are implicitly `public`, whether specified as such or not.
3. Generally used as a guarantee that the implementing class will have the specified features and/or be compatible with any other class which implements the same interface.

Meanwhile, an `abstract` class is:

1. Anywhere from fully abstract to fully implemented, with a tendency to have one or more `abstract` methods. Can contain both declarations and definitions, with declarations marked as `abstract`.
2. A full-fledged class, and subject to the rules that govern other classes (can only inherit from one class), on the condition that it cannot be instantiated (because there's no guarantee that it's fully implemented). Can have non-constant member variables. Can implement member access control, restricting members as `protected`, `private`, or private package (unspecified).
3. Generally used either to provide as much of the implementation as can be shared by multiple subclasses, or to provide as much of the implementation as the programmer is able to supply.

Or, if we want to boil it all down to a single sentence: An `interface` is what the implementing class *has*, but an `abstract` class is what the subclass *is*.

answered Jan 28 '16 at 21:58



Many junior developers make the mistake of thinking of interfaces, abstract and concrete classes as slight variations of the same thing, and choose one of them purely on technical grounds: *Do I need multiple inheritance? Do I need some place to put common methods? Do I need to bother with something other than just a concrete class?* This is wrong, and hidden in these questions is the main problem: **"I"**. When you write code for yourself, by yourself, you rarely think of other present or future developers working on or with your code.

Interfaces and abstract classes, although apparently similar from a technical point of view, have completely different meanings and purposes.

## Summary

1. An interface **defines a contract** that some implementation will fulfill *for you*.
2. An abstract class **provides a default behavior** that *your implementation* can reuse.

## Alternative summary

1. An interface is for defining public APIs
2. An abstract class is for internal use, and for defining SPIs

## On the importance of hiding implementation details

A concrete class does the actual work, in a very specific way. For example, an `ArrayList` uses a contiguous area of memory to store a list of objects in a compact manner which offers fast random access, iteration, and in-place changes, but is terrible at insertions, deletions, and occasionally even additions; meanwhile, a `LinkedList` uses double-linked nodes to store a list of objects, which instead offers fast iteration, in-place changes, and insertion/deletion/addition, but is terrible at random access. These two types of lists are optimized for different use cases, and it matters a lot how you're going to use them. When you're trying to squeeze performance out of a list that you're heavily interacting with, and when picking the type of list is up to you, you should carefully pick which one you're instantiating.

On the other hand, high level users of a list don't really care how it is actually implemented, and they should be insulated from these details. Let's imagine that Java didn't expose the `List` interface, but only had a concrete `List` class that's actually what `LinkedList` is right now. All Java developers would have tailored their code to fit the implementation details: avoid random access, add a cache to speed up access, or just reimplement `ArrayList` on their own, although it would be incompatible with all the other code that actually works with `List` only. That would be terrible... But now imagine that the Java masters actually realize that a linked list is terrible for most actual use cases, and decided to switch over to an array list for their only `List` class available. This would affect the performance of every Java program in the world, and people wouldn't be happy about it. And the main culprit is that implementation details were available, and the developers assumed that those details are a permanent contract that they can rely on. This is why it's important to hide implementation details, and only define an abstract contract. This is the purpose of an interface: define what kind of input a method accepts, and what kind of output is expected, without exposing all the guts that would tempt programmers to tweak their code to fit the internal details that might change with any future update.

An abstract class is in the middle between interfaces and concrete classes. It is supposed to help implementations share common or boring code. For example, `AbstractCollection` provides basic implementations for `isEmpty` based on size is 0, `contains` as iterate and compare, `addAll` as repeated `add`, and so on. This lets implementations focus on the crucial parts that differentiate between them: how to actually store and retrieve data.

## APIs versus SPIs

Interfaces are low-cohesion **gateways** between different parts of code. They allow libraries to exist and evolve without breaking every library user when something changes internally. It's called *Application Programming Interface*, not Application Programming Classes. On a smaller scale, they also allow multiple developers to collaborate successfully on large scale projects, by separating different modules through well documented interfaces.

Abstract classes are high-cohesion **helpers** to be used when implementing an interface, assuming some level of implementation details. Alternatively, abstract classes are used for defining SPIs, Service Provider Interfaces.

The difference between an API and an SPI is subtle, but important: for an API, the focus is on who **uses** it, and for an SPI the focus is on who **implements** it.

Adding methods to an API is easy, all existing users of the API will still compile. Adding methods to an SPI is hard, since every service provider (concrete implementation) will have to implement the new methods. If interfaces are used to define an SPI, a provider will have to release a new version whenever the SPI contract changes. If abstract classes are used instead, new methods could either be defined in terms of existing abstract methods, or as empty `throw not implemented exception` stubs, which will at least allow an older version of a service implementation to still compile and run.

### A note on Java 8 and default methods

Although Java 8 introduced default methods for interfaces, which makes the line between interfaces and abstract classes even blurrier, this wasn't so that implementations can reuse code, but to make it easier to change interfaces that serve both as an API and as an SPI (or are wrongly used for defining SPIs instead of abstract classes).

### Which one to use?

1. Is the *thing* supposed to be **publicly used** by other parts of the code, or by other external code? Add an interface to it to hide the implementation details from the public abstract contract, which is the general behavior of the thing.
2. Is the *thing* something that's supposed to have multiple implementations with a lot of code in common? Make both an interface and an abstract, incomplete implementation.
3. Is there ever going to be only one implementation, and nobody else will use it? Just make it a concrete class.
  1. "ever" is long time, you could play it safe and still add an interface on top of it.

A corollary: the other way around is often wrongly done: when using a *thing*, always try to use the most generic class/interface that you actually need. In other words, don't declare your variables as `ArrayList theList = new ArrayList()`, unless you actually have a very strong dependency on it being an **array** list, and no other type of list would cut it for you. Use `List theList = new ArrayList` instead, or even `Collection theCollection = new ArrayList` if the fact that it's a list, and not any other type of collection doesn't actually matter.

answered Jun 19 '16 at 9:02



**Sergiu Dumitriu**  
9,203 ● 3 ● 24 ● 57

By definition, interfaces cannot have an implementation for any methods, and member variables cannot be initialized.

However, abstract classes can have methods implementated and member variables initialized.

Use abstract classes when you expect changes in your contract, i.e., say in future you might need to add a new method.

In this situation, if you decide to use an interface, when the interface is changed to include interface, your application will break when you dumped the new interface dll.

To read in detail, visit [difference between abstract class and a interface](#)

edited Jun 30 '16 at 16:27



**ragingasiancoder**  
605 ● 4 ● 16

answered Jan 19 '15 at 8:59



**Ranganatha**  
675 ● 5 ● 19

I'd like to add one more difference which makes sense. For example, you have a framework with thousands of lines of code. Now if you want to add a new feature throughout the code using a method `enhanceUI()`, then it's better to add that method in abstract class rather in interface. Because, if you add this method in an interface then you should implement it in all the implemented class but it's not the case if you add the method in abstract class.

answered Dec 15 '16 at 17:47



**Toothless**  
115 ● 3 ● 12

**Differences between abstract class and interface on behalf of real implementation.**



**Interface:** It is a keyword and it is used to define the template or blue print of an object and it forces all the sub classes would follow the same prototype,as for as implementation, all the sub classes are free to implement the functionality as per it's requirement.

Some of other use cases where we should use interface.

Communication between two external objects(Third party integration in our application) done through **Interface** here Interface works as Contract.

**Abstract Class:** Abstract,it is a keyword and when we use this keyword before any class then it becomes abstract class.It is mainly used when we need to define the template as well as some default functionality of an object that is followed by all the sub classes and this way it removes the redundant code and **one more use cases where we can use abstract class**, such as we want no other classes can directly instantiate an object of the class, only derived classes can use the functionality.

**Example of Abstract Class:**

```
public abstract class DesireCar
{

    //It is an abstract method that defines the prototype.
    public abstract void Color();

    // It is a default implementation of a Wheel method as all the desire cars have
    same no. of wheels.
    // and hence no need to define this in all the sub classes in this way it avoids
    duplicasy

    public void Wheel() {

        Console.WriteLine("Car has four wheel");
    }

    **Here is the sub classes:**

    public class DesireCar1 : DesireCar
    {
        public override void Color()
        {
            Console.WriteLine("This is a red color Desire car");
        }
    }

    public class DesireCar2 : DesireCar
    {
        public override void Color()
        {
            Console.WriteLine("This is a red white Desire car");
        }
    }
}
```

**Example Of Interface:**


```
public interface IShape
{
    // Defines the prototype(template)
    void Draw();
}

// All the sub classes follow the same template but implementation can be different

public class Circle : IShape
{
    public void Draw()
    {
        Console.WriteLine("This is a Circle");
    }
}

public class Rectangle : IShape
{
    public void Draw()
    {
        Console.WriteLine("This is a Rectangle");
    }
}
```

[edited Aug 18 '17 at 5:56](#)

answered Aug 16 '17 at 8:43  
 [Sheo Dayal Singh](#)  
428 ● 5 ● 5

You can find clear difference between **interface** and **abstract class**.

**Interface**

- Interface only contains abstract methods.
- Force users to implement all methods when implements the interface.
- Contains only final and static variables.

- Declare using interface keyword.
- All methods of an interface must be defined as public.
- An interface can extend or a class can implement multiple other interfaces.

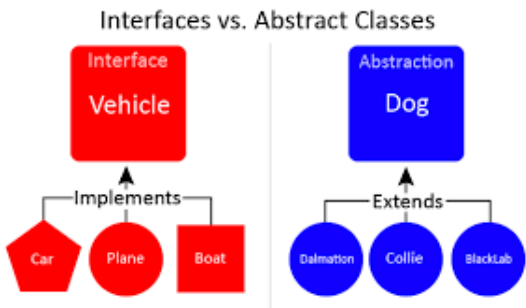
Abstract class

- Abstract class contains abstract and non-abstract methods.
- Does not force users to implement all methods when inherited the abstract class.
- Contains all kinds of variables including primitive and non-primitive
- Declare using abstract keyword.
- Methods and members of an abstract class can be defined with any visibility.
- A child class can only extend a single class (abstract or concrete).

answered Aug 24 '17 at 14:05




**Rahul Chauhan**  
367 ● 2 ● 9



Here is a very basic understanding over interface vs abstract class.

answered Jan 22 '16 at 12:24



**Santosh**  
297 ● 3 ● 13

- 10 How does this explain the difference? Why couldn't Car/Plane/Boat *extend* an abstract Vehicle class? – [aioobe](#) Aug 30 '16 at 6:58
- 3 I don't understand, I even don't agree. It contradicts the answer with maximum votes – [whiteletters in blankpapers](#) Apr 22 '17 at 11:04 ✎
- 1 How can this answer get 6 upvotes? – [Pritam Banerjee](#) Sep 17 '17 at 6:58


An abstract class is a class whose object cannot be created or a class which cannot be instantiated. An abstract method makes a class abstract. An abstract class needs to be inherited in order to override the methods that are declared in the abstract class. No restriction on access specifiers. An abstract class can have constructor and other concrete(non abstract methods ) methods in them but interface cannot have.

An interface is a blueprint/template of methods.(eg. A house on a paper is given(interface house) and different architects will use their ideas to build it(the classes of architects implementing the house interface) . It is a collection of abstract methods , default methods , static methods , final variables and nested classes. All members will be either final or public , protected and private access specifiers are not allowed.No object creation is allowed. A class has to be made in order to use the implementing interface and also to override the abstract method declared in the interface. An interface is a good example of loose coupling(dynamic polymorphism/dynamic binding) An interface implements polymorphism and abstraction.It tells what to do but how to do is defined by the implementing class. For Eg. There's a car company and it wants that some features to be same for all the car it is manufacturing so for that the company would be making an interface vehicle which will have those features and different classes of car(like Maruti Suzkhi , Maruti 800) will override those features(functions).

Why interface when we already have abstract class? Java supports only multilevel and hierarchal inheritance but with the help of interface we can implement multiple inheritance.

edited Sep 20 '16 at 19:19

answered Sep 20 '16 at 19:12



**Tutu Kumari**  
109 ● 1 ● 3

To give a simple but clear answer, it helps to set the context : you use both

when you do not want to provide full implementations.

The main difference then is an interface has no implementation at all (only methods without a body) while abstract classes can have members and methods with a body as well, i.e. can be partially implemented.

answered Jun 28 '17 at 10:49



user3775501

57 2 9

In an interface all methods must be only definitions, not single one should be implemented.

But in an abstract class there must an abstract method with only definition, but other methods can be also in the abstract class with implementation...

edited Nov 20 '16 at 14:25



Peter Mortensen

12.9k 19 83 111

answered Aug 25 '13 at 21:56



Shamim Ahmed

716 2 15 26

I read a simple yet effective explanation of Abstract class and Interface on [php.net](#)

Which is as follows.

An Interface is like a protocol. It doesn't designate the behavior of the object; it designates how your code tells that object to act. An interface would be like the English Language: defining an interface defines how your code communicates with any object implementing that interface.

An interface is always an agreement or a promise. When a class says "I implement interface Y", it is saying "I promise to have the same public methods that any object with interface Y has".

On the other hand, an Abstract Class is like a partially built class. It is much like a document with blanks to fill in. It might be using English, but that isn't as important as the fact that some of the document is already written.

An abstract class is the foundation for another object. When a class says "I extend abstract class Y", it is saying "I use some methods or properties already defined in this other class named Y".

So, consider the following PHP:

```
<?php
class X implements Y { } // this is saying that "X" agrees to speak language
your code.

class X extends Y { } // this is saying that "X" is going to complete the p
"Y".
?>
```

You would have your class implement a particular interface if you were distributing a class to be used by other people. The interface is an agreement to have a specific set of public methods for your class.

You would have your class extend an abstract class if you (or someone else) wrote a class that already had some methods written that you want to use in your new class.

These concepts, while easy to confuse, are specifically different and distinct. For all intents and purposes, if you're the only user of any of your classes, you don't need to implement interfaces.

answered May 23 at 9:19



Vatsal Shah

339 2 14

protected by Stefano Borini Mar 27 '13 at 18:20

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now requires 10 reputation on this site (the association bonus does not count).

Would you like to answer one of these unanswered questions instead?