

Abstract Classes versus Interfaces

C# (and other languages) polymorphism explained to young padawans.

Until very recently, whenever I was interviewing a candidate for a developer job, one of my favorite questions was: “What are the similarities and differences between an abstract class and an interface?”

Not that I expected a single perfect, universal and documented answer but I found that answers to this question are quite revealing of one’s level of understanding of OOP and C#. Alas, so many interviewees don’t reveal that much... ☹

So, today I finally figured that, rather than trying to pin down the poor guys, I’d better put some time into explaining what polymorphism exactly is and how it translates into C# abstract classes or interfaces, and when or why you’ll want to choose one or the other. And, for my interviews, I’ll just have to devise some other way to sort out the candidates...

Also it’s an opportunity to share my views with the community and, hopefully, get some valuable comments (either on OOP or...interview techniques!).

Why are you talking about Polymorphism, and what exactly is that?

The barbarian name simply hides one of the three pillars¹ of Object Oriented Programming (OOP); it covers the ability to use different types of objects without knowing the specifics of their incarnation but by addressing a known shared definition of methods and controls.

In real life, polymorphism allows you to rent a car: you don’t need to know the precise make and model of the car; as long as it behaves like any other car, you know you’ll be able to drive it.

For a programming language to implement polymorphism you need a way to specify a set of methods and/or properties that you can manipulate without knowing the details of their implementations; in other words you need an abstraction mechanism.

For instance, designing a drawing software, you could define a concept of “graphical element” that has coordinates, a size (bounding rect width and height), that can be drawn on a surface and whatever other knobs you need to manipulate it.

Polymorphism in C# (and other languages...)

C-Sharp, like many other languages, lets you define this in two ways: through inheritance or by declaring interfaces.

In our example, you could define a base class *GraphicalElement* that has all the methods and properties you need.

Marking these methods as *virtual* allows descendants to override the ancestor’s implementation and provide their own code (e.g. drawing method for *Rectangle* class will be different from *Circle*’s *Draw* implementation).

¹ The two other pillars being : encapsulation and inheritance

Inheritance will allow you to treat any instance of `Rectangle` or `Circle` as `GraphicalElement`, ignoring their specifics. That's polymorphism.

However, it would be rather pointless in our case to implement the `GraphicalElement` base `Draw` method as we don't know what to draw. Moreover, you wouldn't want to instantiate a `GraphicalElement`; you want to instantiate `Rectangles` or `Circles`.

That's because `GraphicalElement` has no physical reality, it's a concept, an Abstraction.

That's where *abstract* classes appear. Formally, an abstract class is a class that has at least one method (or property) that's abstract, i.e. that has no implementation.

What's with that: a class with methods or properties that don't have an implementation? How can we use it?

That's the point: you can't use it directly. An abstract class cannot be instantiated; you have to inherit it and instantiate descendant, non-abstract, classes. A non-abstract descendant implies that it provides the missing implementations, so that the class can be used.

In our case, if we mark the `GraphicalElement.Draw` method as abstract, the `GraphicalElement` cannot be instantiated directly anymore and both `Circle` and `Rectangle` implementations must provide overridden definitions of the `Draw` method. That's another advantage over simple inheritance. If the ancestor method were just *virtual*, we could override them; here we must.

So, an abstract class is a class that has at least one of its methods that's simply defined, not implemented.

If all methods/properties of the class are abstract, it is said to be a "pure abstract" class ("pure virtual" in C++).

Ok, I get it with the abstract classes, so what about interfaces?

Take a moment to consider a potentially pure abstract class:

```
public abstract class GraphicalElement{
    public abstract void Draw(Graphics canvas);
    public abstract Point Location{get; set;}
    public abstract Size Size{get; set;}
}
```

What's in it: a set of methods and/or properties and... an ancestor. Since every class inherits at least from the `Object` class, directly or indirectly, and `Object` is not a pure abstract class, you in fact always have at least some code base in you class.

So, here, our `GraphicalElements` also have a `ToString()`, `Equals()` and `GetType()` methods, with defined implementations.

An interface, however, is the simplest expression of a programming abstraction: it is a list of methods and/or properties that should be exposed by a class which claims to implement the interface, and only that list. It carries no code and no assumptions about the way it will be implemented.

```
public interface IGraphicalElement{
    void Draw(Graphics canvas);
    Point Location{get; set;}
    Size Size{get; set;}
}
```

The interface can be defined and used without any implementation being known.

If you want to implement our abstract GraphicalElement, you need to inherit GraphicalElement and, indirectly, whatever ancestor class GraphicalElement was based on. If the ancestor class is changed, it impacts your implementation.

The interface on the other hand allows purely functional programming. The implementor is free to choose the base class.

So, what's the answer to your interview question?

Well, I think I have already answered that.

Interfaces and abstract classes both are means of defining a set of methods or properties that different classes will implement, without providing a code base. That's for their similarities. The difference lies in the fact that an abstract class allows you to mix a code base plus abstract methods, whereas an interface is purely abstract. On the other hand, using interfaces for polymorphism lets you free to decide which base class you'll inherit.

Also because of the freedom regarding interfaces implementation, a class can implement multiple interfaces whereas you can only inherit one abstract class.

So choosing between interfaces or abstract classes merely depends on the level of freedom you need/want versus how polymorphic your classes are.

If you need the behavior to be implemented in a large variety of classes, you might be more comfortable with interfaces. On the contrary, if you are working in a closed domain, you might need your different implementations to share more than a contract but also inherit a common set of pre-implemented methods. That's where abstract classes might come handy.

Of course, there's always the option of combining both:

```
public interface IGraphicalElements{
    // ...
}

public abstract class BaseGraphicalElement: IGraphicalElements{
    // some methods from IGraphicalElements are implemented,
    // others are declared abstract
    // ...
}
```

Quick word: C-Sharp particularities

So far, all I've said is rather generic. It is true also of many other languages; most OO languages that have a single-inheritance model work this way, even if keywords or syntax expressions may vary.

C# has some specifics though.

For instance C-Sharp requires that a class which bears abstract methods be marked with the *abstract* keyword, but that's only a syntactic constraint; other languages might infer it from the presence of abstract methods.

Also, you cannot declare a static interface (some languages call it a "class interface") nor static abstract methods or property accessors. So if you need static polymorphism, you'll have to resort to virtual static methods on a base class.

Last, it seems obvious, but for some languages it isn't: abstract methods are implicitly virtual methods (otherwise, how would you override them?).

Conclusion and summary

Both Abstract classes and Interfaces are expressions of polymorphism. The similarities and differences are expressed in the table below:

	→ growing abstraction →		
	Virtual methods	Abstract classes	Interfaces
Polymorphism <i>Multiple implementations of a shared set of methods</i>	☺	☺	☺
Abstraction <i>Definition of methods with no implementation</i>		☺	☺
Partial polymorphism <i>Can provide some code base, which implementers inherit</i>	☺	☺	
Multiple inheritance <i>Same class implements multiple abstractions</i>			☺
Metaclass polymorphism <i>Multiple implementations of static methods or properties</i>	☺		