Abstract Classes vs. Interface

Source http://www.javaworld.com/javaworld/javaqa/2001-04/03-qa-0420-abstract.html

Abstract classes vs. interfaces

In Java, under what circumstances would you use abstract classes instead of interfaces? When you declare a method as abstract, can other nonabstract methods access it? In general, could you explain what abstract classes are and when you might use them?

Those are all excellent questions: the kind that everyone should ask as they begin to dig deeper into the Java language and object-oriented programming in general

Yes, other nonabstract methods can access a method that you declare as abstract.

But first, let's look at when to use normal class definitions and when to use interfaces. Then I'll tackle abstract classes.

Class vs. interface

Some say you should define all classes in terms of interfaces, but I think recommendation seems a bit extreme. I use interfaces when I see that something in my design will change frequently.

For example, the Strategy pattern lets you swap new algorithms and processes into your program without altering the objects that use them. A media player might know how to play CDs, MP3s, and wav files. Of course, you don't want to hardcode those playback algorithms into the player; that will make it difficult to add a new format like AVI. Furthermore, your code will be littered with useless case statements. And to add insult to injury, you will need to update those case statements each time you add a new algorithm. All in all, this is not a very object-oriented way to program.

With the Strategy pattern, you can simply encapsulate the algorithm behind an object. If you do that, you can provide new media plug-ins at any time. Let's call the plug-in class `MediaStrategy`. That object would have one method: `playStream(Stream s)`. So to add a new algorithm, we simply extend our algorithm class. Now, when the program encounters the new

media type, it simply delegates the playing of the stream to our media strategy. Of course, you'll need some plumbing to properly instantiate the algorithm strategies you will need.
This is an excellent place to use an interface. We've used the Strategy pattern, which clearly indicates a place in the design that will change. Thus, you should define the strategy as an interface. You should generally favor interfaces over inheritance when you want an object to have a certain type; in this case, `MediaStrategy`. Relying on inheritance for type identity is dangerous; it locks you into a particular inheritance hierarchy. Java doesn't allow multiple inheritance, so you can't extend something that gives you a useful implementation or more type identity.

Interface vs. abstract class

Choosing interfaces and abstract classes is not an either/or proposition. If you need to change your design, make it an interface. However, you may have abstract classes that provide some default behavior. Abstract classes are excellent candidates inside of application frameworks.

Abstract classes let you define some behaviors; they force your subclasses to provide others. For example, if you have an application framework, an abstract class may provide default services such as event and message handling. Those services allow your application to plug in to your application framework. However, there is some application-specific functionality that only your application can perform. Such functionality might include startup and shutdown tasks, which are often application-dependent. So instead of trying to define that behavior itself, the abstract base class can declare abstract shutdown and startup methods. The base class knows that it needs those methods, but an abstract class lets your class admit that it doesn't know how to perform those actions; it only knows that it must initiate the actions. When it is time to start up, the abstract class can call the startup method. When the base class calls this method, Java calls the method defined by the child class.

Many developers forget that a class that defines an abstract method can call that method as well. Abstract classes are an excellent way to create planned inheritance hierarchies. They're also a good choice for nonleaf classes in class hierarchies.

~~~ End of Article ~~~