

1. Interface or Abstract Class?

Interfaces, like classes, define a set of properties, methods, and events. But unlike classes, interfaces do not provide implementation. They are implemented by classes, and defined as separate entities from classes. Even though class inheritance allows your classes to inherit implementation from a base class, it also forces you to make most of your design decisions when the class is first published.

Abstract classes are useful when creating components because they allow you specify an invariant level of functionality in some methods, but leave the implementation of other methods until a specific implementation of that class is needed. They also version well, because if additional functionality is needed in derived classes, it can be added to the base class without breaking code.

Interfaces vs. Abstract Classes

Feature	Interface	Abstract class
Multiple inheritance	A class may implement several interfaces.	A class may extend only one abstract class.
Default implementation	An interface cannot provide any code at all, much less default code.	An abstract class can provide complete code, default code, and/or just stubs that have to be overridden.
Constants	Static final constants only, can use them without qualification in classes that implement the interface. On the other paw, these unqualified names pollute the namespace. You can use them and it is not obvious where they are coming from since the qualification is optional.	Both instance and static constants are possible. Both static and instance intialiser code are also possible to compute the constants.
Third party convenience	An interface implementation may be added to any existing third party class.	A third party class must be rewritten to extend only from the abstract class.
is-a vs -able or can-do	Interfaces are often used to describe the peripheral abilities of a class, not its central identity, e.g. an Automobile class might implement the Recyclable interface, which could apply to many otherwise totally unrelated objects.	An abstract class defines the core identity of its descendants. If you defined a Dog abstract class then Damamation descendants are Dogs, they are not merely dogable. Implemented interfaces enumerate the general things a class can do, not the things a class is.
Plug-in	You can write a new replacement module for an interface that contains not one stick of code in common with the existing implementations. When you implement the interface, you start from scratch without any default implementation. You have to obtain your tools from other classes; nothing comes with the interface other than a few constants. This gives you freedom to implement a radically different internal design.	You must use the abstract class as-is for the code base, with all its attendant baggage, good or bad. The abstract class author has imposed structure on you. Depending on the cleverness of the author of the abstract class, this may be good or bad. Another issue that's important is what I call "heterogeneous vs. homogeneous." If implementors/subclasses are homogeneous, tend towards an abstract base class. If they are heterogeneous, use an interface. (Now all I have to do is come up with a good definition of hetero/homogeneous in this context.) If the various objects are all of-a-kind, and share a common state and behavior, then tend towards a common base class. If all they share is a set of method signatures, then tend towards an interface.
Homogeneity	If all the various implementations share is the method signatures, then an interface works best.	If the various implementations are all of a kind and share a common status and behavior, usually an abstract class works best.
Maintenance	If your client code talks only in terms of an interface, you can easily change the concrete implementation behind it,	Just like an interface, if your client code talks only in terms of an abstract class, you can easily change the concrete

	using a factory method.	implementation behind it, using a factory method.
Speed	Slow, requires extra indirection to find the corresponding method in the actual class. Modern JVMs are discovering ways to reduce this speed penalty.	Fast
Terseness	The constant declarations in an interface are all presumed public static final, so you may leave that part out. You can't call any methods to compute the initial values of your constants. You need not declare individual methods of an interface abstract. They are all presumed so.	You can put shared code into an abstract class, where you cannot into an interface. If interfaces want to share code, you will have to write other bubblegum to arrange that. You may use methods to compute the initial values of your constants and variables, both instance and static. You must declare all the individual methods of an abstract class abstract.
Adding functionality	If you add a new method to an interface, you must track down all implementations of that interface in the universe and provide them with a concrete implementation of that method.	If you add a new method to an abstract class, you have the option of providing a default implementation of it. Then all existing code will continue to work without change.