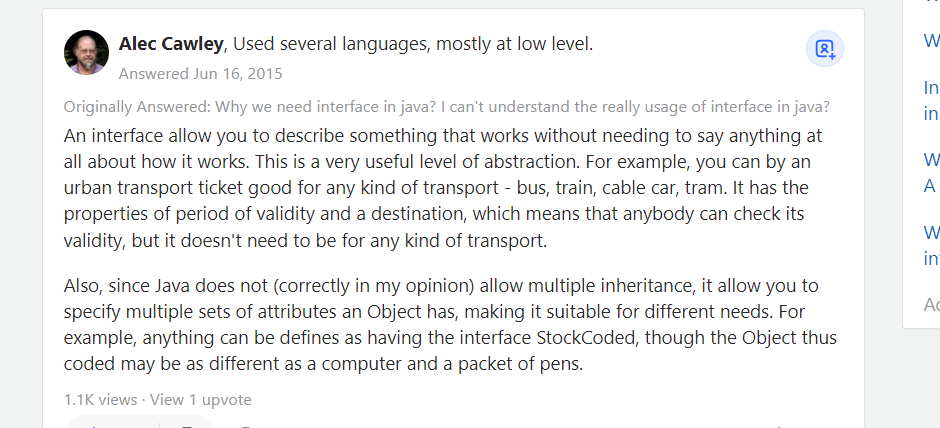
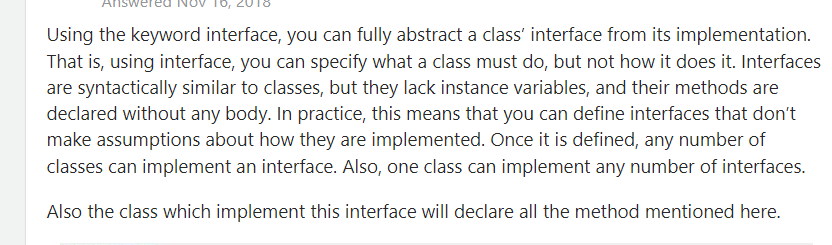
Why are interfaces used?





Why use interfaces for programming in Java and C#?

People often give the answer "because it promises a certain set of behaviours or methods" and then don't offer a good reason why this is important. Let me provide a concrete example:

One of the goals of modern object oriented software development is to achieve "loose coupling" and to remove dependencies between different classes. The benefit of this is to prevent changes in one part of a system affecting other parts which minimises the amount of new code that needs to be written when new functionality is added or updated. Interfaces provide a fantastic tool to achieve this.

Let's imagine you have 3 classes, A, B and C. Class A wants to perform a certain action, which may require either class B or class C.

Let's say class A is a class that monitors and logs system events, and B is a class which implements logging to a file, while C is a class which implements logging/alerting by email.

You could implement class A to make the choice of whether to use B or C to handle the logging/alerting process. It would need a conditional structure which chooses to instantiate either class B if it wants to log an event to file, or to instantiate class C if, let's say, a major error occurs and it wants to send an email alert to the administrator.

But let's say you now want to add a fourth class, D, which handles logging via SMS message, and a fifth class, E, which makes an automated phone call to report super-critical errors. Every time you add an extra option, you need to make another change to class A's conditional statement. This creates a dependency because class A's implementation is entirely dependent on how many types of logging/alerting there are, and keeps needing to be updated every time the system adds functionality elsewhere. This is a fairly simple and trivial example on its own, but if you multiply this kind of design across a large and complex system, these dependencies quickly add up and make maintaining code much more difficult because you're constantly having to update multiple classes all the time.

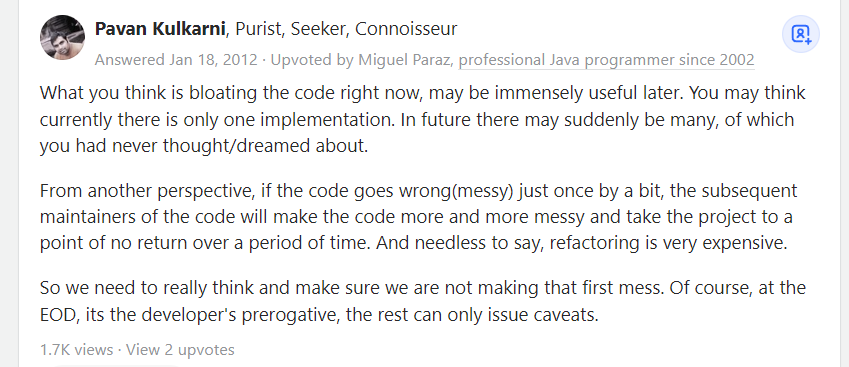
To solve this problem, we can use a technique called Inversion of Control (IoC) - the only thing that Class A needs to know is that it needs to call a method in another class which implements logging/alerting of system events - it doesn't know how - it just needs to provide the log message and the other class is automatically of the right type to handle the type of event and log to file, send email alert etc. It doesn't make the decision about which class to use.

At runtime, when a system event occurs, the correct object (an instantiation of B, C, D or E etc.) is created depending on the type of event, perhaps by the code which raises the event or perhaps by another specialist class, and is passed (or we often say "injected") into Class A - this could be done by setting a property, or passed as an argument to a Class A constructor or method.

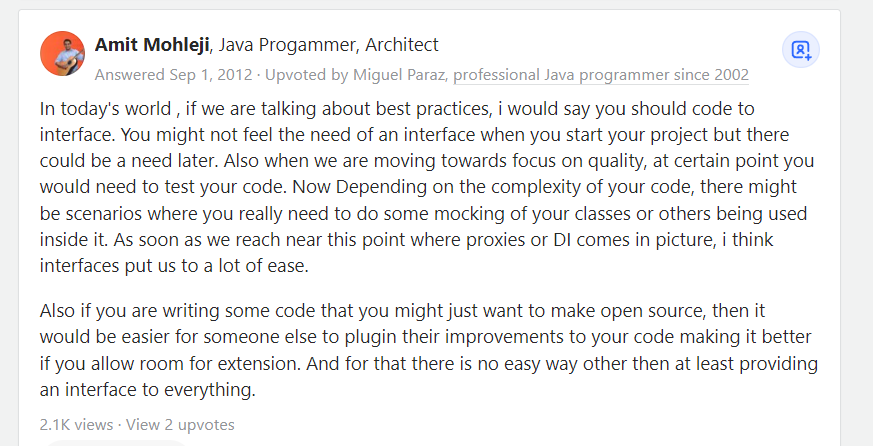
Class A simply calls one or more methods in Class B, C, D or E which handle the logging/alerting process and passes them the log message. But how does Class A know which methods in B, C, D or E to call - they all do different things? Because they are all WRITTEN TO THE SAME INTERFACE. Class A only needs to know that B, C, D, E etc. implement the "ILogEvent" interface in order to know exactly what methods are available to it. Each individual class (A, B, C...) may have other methods to handle their specific functions, but they also share a common set of functionality which Class A uses to interface with them.

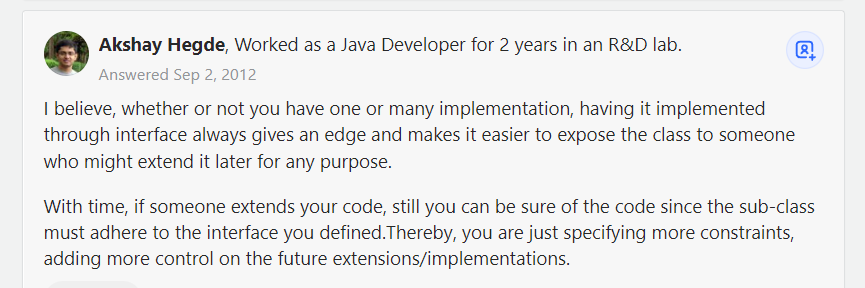
Class A never needs to know what object it is going to be given. It just needs to know that its class is an implementation of a particular interface. This means that years in the future, a totally separate developer could come along, and　add, for example, a "log by radio broadcast" Class F, and providing it implements the IUpdateLog interface, Class A won't even know the change has been made - it doesn't need further maintenance. It has been de-coupled. The dependency on another part of the system has been removed.

This is perhaps a rather long winded and singular example of using interfaces (there are many others), but I know that years ago when I was struggling to figure out why interfaces were important, it helped me a lot to see a specific case where they were hugely beneficial. And the example someone explained to me, was pretty similar to the above.

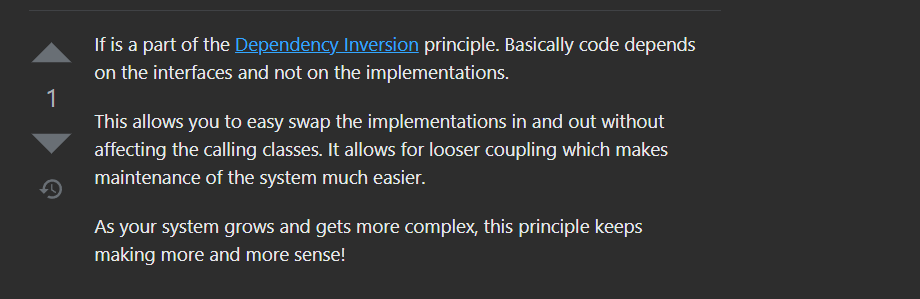


Hence it’s best practice to use an interface for separate classes



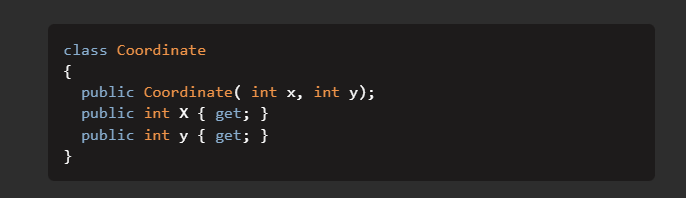


So having multiple interfaces for multiple classes in a project is actually best practice, do not hesitate to do this



So having more interfaces means our code is less dependent on classes, meaning the implementation of methods rather on the interfaces which only defines the methods

But Beware



For classes like this where there is no implementation of any method or a very little implementation creating an interface will be useless and expensive

So, for heavy and big classes having more methods, creating interfaces is the best practice