Stephen Downey

Professor Mueller

Full-Stack Developer Bootcamp

1 March 2021

Contrasting Classes, Abstract Classes, and Interfaces

Java’s object oriented programming hierarchy includes classes, abstract classes, and interfaces. Most programming takes place in concrete classes, but abstract classes and interfaces can add organization and direction to a class. A program’s hierarchy refers to the tree of classes and interfaces that are extended and implemented in our main program. For example, if a class does not explicitly extend another class, then it automatically extends the base class Object, and inherits all of that class’ methods. Classes, abstract classes, and interfaces have many technical differences though which makes each one more or less useful in different situations. Here are some of the differences between the three and how those differences affect how each one is used.

First of all, there are quite a few technical differences between classes, abstract classes, and interfaces, more than this essay will cover. One of the biggest differences though is that concrete classes must define the behavior of their methods. That means, if a class has a method with a return type, that method has to return something. An interface is the opposite because all of the methods in an interface are implicitly abstract, meaning that the behavior does not have to be defined and the method doesn’t have to return anything. The abstract class can do either. It can have both abstract and concrete methods. Also, classes and abstract classes can have member variables while interfaces only have methods. Another big difference is that only one class or abstract class can be extended into another class at a time. This differs from interfaces in that a class can implement as many interfaces as it needs. A couple more differences include that interfaces and classes can be instantiated into objects, but an abstract class cannot. In fact, this fact and the fact that abstract classes can have abstract methods seem to be the defining differences between classes and interfaces. Finally, an interesting difference between the three is that a developer can add new methods to a class or abstract class, that has already been extended into other classes, without breaking any of the subclasses. This isn’t true with interfaces. If a developer adds any new methods to an interface without adding that method to all of the subclasses as well, or adding some default behaviour for the method inside the interface, then all of the subclasses will become unusable. That’s a good list of differences, and it certainly isn’t all of them. What’s important to note is that these differences will affect how each one is used.

Here’s three differences in how classes, abstract classes, and interfaces are used based on the differences listed above. First of all, the abstract class cannot be instantiated into an object, and that’s the main reason a developer would use that structure instead of a concrete class. This might be the case if the class is high in the hierarchy of organization and thus a more general or abstract concept that shouldn’t be used as an object in a class. Then a developer might make the class abstract. A developer might also consider making an abstract class because it can hold abstract methods, but an interface can also do this. In fact, using abstract classes for abstract methods can be limiting because only one class can be inherited in another class versus the ability to implement many interfaces. Because of this, it’s likely that classes and abstract classes are used more often in cases where the abstraction is closely related to the core purpose of the software that’s being developed. For more tangential relationships an interface might be more appropriate because it gives the developer access to the package without using the one available spot to extend a class (LearnerLearner 4). In this case, the developer would be using the base class for organization, hierarchy, and code reduction, and using the interface to help define a desired set of behaviors. For this reason, it could be useful to create interfaces with a class or abstract class that defines its behavior. This way, the interface can be easily used by extending the class, but can also be accessed directly from the interface for projects that are not as closely related to that base class. This can be seen in the AbstractList class which is, “a skeletal implementation of the List interface to minimize the effort required to implement this interface backed by a "random access" data store (such as an array)" (AbstractList). Finally, a developer may be driven in their choice to use a class, abstract class, or interface based on the different levels of support for abstract methods. On one hand, using a base class can save a developer from repeating code because all subclasses will inherit the behaviour of the base class methods. An example of this from class is the program Professor Mueller demonstrated with the developer, team lead, and architect. The team lead and architect could do anything the developer could, so they inherited that base class and its methods, thus saving the developer the need to code them again. On the other hand, for the same reason, an interface may be more desirable if the subclasses will have quite different behaviors.

This isn’t all the differences between classes, abstract classes, and interfaces either technically or in regards to their use. There are a few more that could be listed under each of those categories. However, this essay aims to show that there isn’t just one of these structures that is the best to use all the time. Each one has its own specific use based on its unique abilities and limitations.

Works Cited

*AbstractList (Java Platform SE 8 )*, 7 Jan. 2021, docs.oracle.com/javase/8/docs/api/java/util/AbstractList.html.

LearnerLearner 4, et al. “Abstract Class vs Interface in Java.” *Stack Overflow*, 1 Feb. 1961, stackoverflow.com/questions/10040069/abstract-class-vs-interface-in-java.