# Multiple Inheritance in Java

***by Alan Oursland***


Comment on this article



If you are familiar with C++, you know that multiple inheritance allows an object to inherit the functionality of two or more classes. It can be a very powerful tool, but is not without its drawbacks. Base classes with identical function names, or even worse base classes with common base classes, can create headaches for even the most experienced developers.

Java gets rid of the headaches associated with multiple inheritance by not even supporting multiple inheritance. It can be argued that multiple inheritance isn't really necessary even in C++. Any time you think you need multiple inheritance it can usually be replaced by a container or delegate pattern. The title of this article may confuse you since Java doesn't actually support multiple inheritance. I'm going to show you some techniques that can be used to get the advantages of multiple inheritance without some of the drawbacks.

In order to accomplish this goal, we first need to know the drawbacks and advantages of multiple inheritance. Multiple inheritance allows us to give classes with different bases new common functionality. It allows us to add that functionality automatically while maintaining the code for the functionality in one place. On the other hand, multiple inheritance can cause a lot of confusion when two base classes implement methods with the same name. This results in two implementations for the same function identifier in the derived class. Java fixes the problem by only allowing classes in Java to inherit functionality from one class. This has the unfortunate side effect of removing all of the benefits you get from multiple inheritance. This article will show you how you can use interfaces and a delegate model to reclaim some of those benefits.

Classes in Java can only extend one class. However, classes can implement as many interfaces as they want. Let's look briefly at what this means. Extending a class means the derived class inherits all of the data members, functions, and function definitions of the base class. The derived class can add to or completely replace implementations of the base class functions. The derived class will also show up as an instance of the base class when you use the operator "instanceof". Implementing an interface means the derived class must provide its own implementation for each function in the interface. There is no base functionality. If a class implements two interfaces that both happen to have identical functions, the derived class will only have one implementation of that function. While somewhat limiting, this removes a lot of the headaches associated with multiple inheritance. It is impossible for a class to inherit two sets of functionality for the same function name. If it happens to inherit the same function name from two different interfaces, the class is forced to resolve the conflict and callers to the class do not have to figure out which implementation to call.

Look at the version 001 of the Java files included with this article (see link at end of article). This program draws a line on the screen in Cartesian coordinates where the origin is in the lower left hand corner of the screen, the x-axis is positive to the right, and the y-axis is positive to the top. Screen coordinates on the other hand have the origin at the upper left corner of the screen with the y-axis positive to the bottom. There are a few things that you should note about this program. First, both classes implement the functions called mapWorldToScreen and mapScreenToWorld. These functions convert screen coordinates to Cartesian coordinates and vice versa. If you want to change how your space is mapped (perhaps you would like to change the origin of the Cartesian space), you will need to make changes to both files. Second, CLine has to know that its parent is a CCartesianPlane object, and CCartesianPlane expects CLine as its child. This causes CLine to be strongly coupled to CCartesianPlane. If we wanted to add a new shape, it would be nearly impossible at this point. While not really dealing with multiple inheritance, strong coupling is something you generally want to avoid because it reduces the flexibility and reusability of your code.

Version 002 takes a first step towards multiple inheritance by defining an interface that both classes can support to map points between the screen and Cartesian spaces. We now have two disjoint classes with similar functionality that can be referenced the same way. The cast to CCartesianPlane in CLine has been changed to a cast to ICoordinateSpace. It is worth noting here that casting an object like this is considered to be weak object oriented program. It could be argued that the ICoordinateSpace interface should support the concept of a parent instead of using the Component's parent. When we are dealing with interfaces, I usually break this rule of OOP. My feeling is that the class knows what interfaces it uses and can act accordingly if a member variable does not support that interface.

The implementations of ICoordinateSpace in CCartesianPlane and CLine still duplicate a lot of code. Version 003 puts all of this duplicated code into a new class called CCoordinateSpaceImplBase. This new class provides a base implementation of ICoordinateSpace. CLine and CCartesianPlane now contain an instance of CCoordinateSpaceImplBase. The implementations of ICoordinateSpace are now routed to this data

member. Had the classes been derived from CCoordinateSpaceImplBase, these calls would have looked something like "super.mapScreenToWorld(p)". CCartesianPlane still provides its own implementation of mapScreenToWorld and mapWorldToScreen. In a sense, it is overriding the functionality of its 'base class' CCoordinateSpaceImplBase.

The base functionality for ICoordinateSpace is now contained in a single class. If we want to change how our coordinate space works, all we need to do is change this one file. We have now recovered two of the benefits of multiple inheritance. The third benefit, automatically inheriting default behavior, will not be possible with this method. However, if you compare the code in CLine and CCartesianPlane, you will notice that it is almost identical. You could copy this code directly into a new class to obtain the new results. So, while the inheritance isn't quite automatic, it is extremely easy to add.

Although the examples themselves aren't very exciting, it could be jazzed up with new shapes or perhaps even transforms to rotate and scale the coordinate spaces. What is exciting is your ability to approximate multiple inheritance in Java. I hope you find this technique as useful as I have.

multiple_inher.zip contains code examples described by this article.
Get multiple_inher.zip (16K).

| Developed Under: |
| --- |
| JDK 1.3 |