[0950] EXTRA: Program to Interfaces, Not Implementations

One of the most valuable "best practices" in the world of OO design appears to have been ignored by the designers of the C# collection classes - specifically the guideline that you should **Program to interfaces, not implementations**.

How Java Does It

In the world of Java, following the best practice is easy to do. For instance, in Java, "List" is an interface that is implemented by either the concrete "ArrayList" class or the concrete "LinkedList" class (or possibly by a concrete class you created). That means Java programmers can declare a List collection like this:

```
List<String> myBooks = new ArrayList<>();
```

or this,

```
List<String> myBooks = new LinkedList<>();
```

depending on how they plan to use the list in their program. If they are going to insert and delete lots of books from the list, they should choose the LinkedList class for the implementation. If they aren't planning on doing that, then ArrayList is probably a better choice. The point is, **the rest of the Java code doesn't care which implementation class was selected**. The rest of the program is written using the List interface and is automatically independent of the implementation class choice. In fact - and this is why this is so important - later the Java programmer could change their mind and change the implementation class to something else (LinkedList) if necessary AND NONE OF THE OTHER CODE WOULD HAVE TO CHANGE!

That's one of the key advantages of programming to the interface (List) instead of the implementation (ArrayList).

(BTW - If you ever see "ArrayList<String> foo = new ArrayList<>();" in Java, you can be sure that the programmer doesn't fully understand collections and interfaces.)

C# Generic Collections Do Not Follow This Principle

While C# does have interfaces for its collection classes, most C# programmers do not program to them and instead program directly to the collection's implementation class. There are several factors that lead C# programmers to this behavior:

- Most C# programmers use the List collection which (currently) only has one implementation class ("List"). Because there is only one concrete class, the distinction between using "IList" and "List"

seems unnecessary.

- For some reason, C# doesn't consider a "LinkedList" to be a "List." That means that, if you really want to duplicate the Java example above, you'd need to use the "IEnumerable" interface instead of the "IList" interface which just sounds... wrong.
- Because C# doesn't have the <> operator, when C# programmers declare a generic collection object, they usually use "var" as the object's type. Among other problems, "var" guarantees that you are programming to the implementation instead of the interface!

So C# programmers write the following code:

```
var myBooks = new List<string>();
```

If they ever needed to change myBooks into a LinkedList (for example), there is a very good chance the code would break. (Specifically when they passed the LinkedList version of myBooks to a method that was expecting a List object.)

Bottom Line

Despite all of these reservations, I see hundreds of C# programs using the "var" approach to declaring generic collections. So be it. Just realize that your code is less flexible than it could/should be.