



4 Essential C# Tips

Here are the author's favorite tips on using the C# language more effectively.

by *Bill Wagner*



Posted April 22, 2002

1. Program to Interfaces Whenever Possible

The .NET Framework contains both classes and interfaces. When you write routines, you will find that you probably know which .NET class you're using. However, your code will be more robust and more reusable if you program using any supported interfaces instead of the class you happen to be working with at the time. Consider this code:

```
private void LoadList (object [] items,
    ListBox l) {
    for (int i = 0; i < items.Length;i++)
        l.Items.Add (items[i].ToString ());
}
```

This function loads a ListBox from an array of any kind of objects. The code is limited to an array only. Suppose that later you find that objects are stored in a database, or in some other collection. You need to modify the routine to use the different collection type. However, had you written the routine using the ICollection interface, it would work on any type that implements the ICollection interface:

```
private void LoadList (ICollection items,
    ListBox l) {
    foreach (object o in items)
        l.Items.Add (o.ToString ());
}
```

The ICollection interface is implemented by arrays, and all the collections in the System.Collection. In addition, multidimensional arrays support the ICollection interface. If that's not enough, the database .NET classes support the ICollection interface as well. The function written using the interface can be reused many more ways without any modification.

2. Use Properties Instead of Raw Data

With the addition of properties as language elements, there is absolutely no reason to declare data elements with any access level greater than private. Because client code will view properties as data elements, you don't even lose the convenience of working with simple data elements in classes. In addition, using properties gives you more flexibility and more capabilities. Properties provide better encapsulation of your data elements. Properties let you make use of lazy evaluation to return data. Lazy evaluation means that you can calculate the data value only when it is requested from a client, rather than keep it around all the time.

Finally, properties can be virtual. They can even be abstract. You can also declare properties in interfaces.

There is yet another maintenance reason: Even though they are accessed the same way, if you change a data element to a property, client code that had been compiled using the data element will no longer work with the version using the property. In fact, you can even use properties in Web services for those values you want to serialize:

```
private int TheMonth = 0;
```

```
[XmlAttribute ("Month")]
public int Month {
    get {
        return TheMonth;
    }
    set {
        TheMonth = value;
    }
}
```

Simply put, properties let you make all your data elements private.

3. Use Delegates for Producer/Consumer Idiom

When you create a class that implements the producer idiom, use a delegate to notify consumers. This will be a more flexible way to implement this idiom than interfaces. Delegates are multicast, so you can support multiple consumers without creating extra code. Also, you lower the coupling between classes by using the delegate model rather than a full interface model. This simple class processes keyboard input and publishes the input to any registered listeners:

```
public class KeyboardProcessor
{
    private OnGetLine theFunc = null;

    public OnGetLine OnGetLineCallback {
        get {
            return theFunc;
        }
        set {
            theFunc = value;
        }
    }

    public void Run () {
        // Read input.
        // If there is any listeners, publish:
        string s;
        do {
            s = Console.ReadLine ();
            if (s.Length == 0)
                break;
            if (theFunc != null) {
                System.Delegate [] funcs =
                    theFunc.GetInvocationList();
                foreach (OnGetLine f in funcs) {
                    try {
                        f (s);
                    } catch (Exception e) {
                        Console.WriteLine
                            ("Caught Exception: {0}",
                             e.Message);
                    }
                }
            }
        } while (true);
    }
}
```

Any number of listeners can be registered with this producer, and those listeners only need to provide one specific function: the delegate.

4. Pay Attention to Initialization Order

The C# language adds the concept of initializers on member variable declarations. These initializers get executed before the body of the constructor gets executed. In fact, variable initializers get executed before the base class's constructor gets executed.

For this reason, make sure any variable initializers do not make use of base class data; the base class

has not yet been constructed.

Got Questions?

Send questions to wwagner@SRTsolutions.com and I will get the answers on the site.

About the Author

Bill Wagner is [SRT Solutions'](#) Windows technology expert. He is a contributing editor for [Visual Studio Magazine](#) and the author of [C# Core Language Little Black Book](#), an advanced reference for C# developers. Bill has had a lead design role in many projects in his 16 years of software development. He has designed software for engineering and business applications, for desktop and Web environments. He has an extensive background in 2-D and 3-D graphics and multimedia software, including developing the video playback engine used for "The Lion King Animated Storybook." Reach him at wwagner@SRTSolutions.com.



[Close Window](#)

© Copyright 2001-2004 **Fawcette Technical Publications** | [Privacy Policy](#) | [Contact Us](#)