Quick Byte: Extension Methods

A quick intro to Extension Methods by JeremyBytes.com

Overview

Extension methods allow you to add functionality to existing types by adding new methods without deriving a new subtype. If you use LINQ, you will make use of extension methods. And if you use ASP.NET MVC, it is very likely that you will create your own. So, let's see how they work.

Creating an Extension Method

An extension method is created by declaring a `public static` class with a `public static` method. You can download the sample code here: http://www.jeremybytes.com/Downloads.aspx. Here's an example (from JeremyBytes.Extensions project, JBExtensions.cs file):

```
public static class JBExtensions

{
    public static string ToDelimitedString<T>(
        this IEnumerable<T> input, string delimiter)
    {
        var output = new StringBuilder();

        foreach (var itm in input)
        {
            if (output.Length > 0)
                output.Append(delimiter);
            output.Append(itm.ToString());
        }

        return output.ToString();
    }
}
```

This method is designed to take an `IEnumerable<T>` and a string `delimiter`, and return a single `string` which is a delimited list of the items. The definition looks like a pretty standard static method; the only difference is the `this` keyword before the first parameter.

Using an Extension Method

Now let's take a look at how this method can be used. First, the standard way of using an extension method (from JeremyBytes.UI project, MainPage.xaml.cs file):

```
private void MonthButton_Click(object sender, RoutedEventArgs e)
{
    List<string> months = Months.GetMonths();
    MonthBox.Text = JBExtensions.ToDelimitedString(months, ", ");
}
```

Using the static class `JBExtensions`, we call the `ToDelimitedString` method with 2 parameters. `List<T>` implements `IEnumerable<T>` which is why we can pass the `months` object as the first parameter. This is how we would normally call a static method.

But since we included the `this` keyword before the first parameter in our definition, we can use this method as if it were an extension of our `months` object. Here's the updated code:

```
private void MonthButton_Click(object sender, RoutedEventArgs e)
{
    List<string> months = Months.GetMonths();
    MonthBox.Text = months.ToDelimitedString(", ");
}
```

Notice that instead of referencing the static class `JBExtensions`, we are simply using the `months` variable which implements `IEnumerable<T>`. From here, we can call the `ToDelimitedString` method directly, as if it were a member of the `months` object. Additional parameters (such as our delimiter) are passed normally.

What we have essentially done here is extend the `IEnumerable<T>` behavior without creating a derivative type (subtype). The extension will work with any object that implements `IEnumerable<T>` (even if it was created by someone else in a different assembly). The sample code shows this with another class that is a `List<Person>`:

```
private void PersonButton_Click(object sender, RoutedEventArgs e)
{
    List<Person> people = People.GetPeople();
    PersonBox.Text = people.ToDelimitedString(" | ");
}
```

Guidelines

To create and use extension methods, you must follow these guidelines:

- Extension methods must be `public static` methods in a `public static` class. The class name itself is unimportant.
- Extension methods are declared by including the `this` keyword in front of the first parameter. The `this` keyword can only be used with the first parameter.
- Extension methods are used by including the namespace of the `public static` class in the scope where the methods are to be used. This means that extension methods can be collected in a shared library that is used across projects, if desired.

To show these features, the sample project is broken up into multiple assemblies and namespaces. The `Months` class, `People` class, and `JBExtensions` class all exist in their own namespaces in separate assemblies. The UI project references these assemblies and has the namespaces in the `using` statements in the MainPage.xaml.cs file.

If you remove the `using JeremyBytes.Extensions;` statement (or comment it out), then you can see that the UI project will no longer compile. Add it back in, and you'll see that you also get full IntelliSense for the extension method on the extended type.

Real World Usage

LINQ is implemented with a number of extension methods. Take a look at the Help documentation for `IEnumerable<T>` and you'll see several dozen extension methods including things like Where(), OrderBy(), Average(), and Count(). To use these methods in your own code, you just need to include the System.Linq namespace in your project.

As far as creating your own extension methods, you are likely to do this if you use ASP.NET MVC. There is the HtmlHelper class which is very useful in crafting your UI. If you want to do custom data formatting, you simply add your own extension method(s) to the HtmlHelper class and then use them in your View.

Wrap Up

Extension methods are a very powerful tool, allowing you to extend already existing types that you may or may not have direct access to. LINQ uses extension methods quite liberally – in fact, extension methods were created specifically to support LINQ with .NET 3.5. So even if you do not create your own extension methods, understanding how to use them is a step forward in making your code readable and maintainable.

Happy Coding!