Use Optional Parameters to Minimize Method Overloads

By Bill Wagner
March 2010

C# now has support for named parameters at the call site. That means the names of formal parameters are now part of the public interface for your type. Changing the name of a public parameter could break calling code. That means you should avoid using named parameters in many situations, and also you should avoid changing the names of the formal parameters on public, or protected methods.

Of course, no language designer adds features just to make your life difficult. Named parameters were added for a reason, and they have positive uses. Named parameters work with optional parameters to limit the noisiness around many APIs, especially COM APIs for Microsoft Office. This small snippet of code creates a Word document and inserts a small amount of text, using the classic COM methods:

```
var wasted = Type.Missing;

var wordApp = new

   Microsoft.Office.Interop.Word.Application();

wordApp.Visible = true;

Documents docs = wordApp.Documents;



Document doc = docs.Add(ref wasted,   ref wasted, ref wasted, ref wasted);



Range range = doc.Range(0, 0);



range.InsertAfter("Testing, testing, testing. . .");
```

This small, and arguably useless, snippet uses the Type.Missing object four times. Any Office Interop application will use a much larger number of Type.Missing objects in the application. Those instances clutter up your application and hide the actual logic of the software you're building.

That extra noise was the primary driver behind adding optional and named parameters in the C# language. Optional parameters means that these Office APIs can create default values for all those locations where Type.Missing would be used. That simplifies even this small snippet:

```
var wordApp = new

   Microsoft.Office.Interop.Word.Application();

wordApp.Visible = true;

Documents docs = wordApp.Documents;



Document doc = docs.Add();



Range range = doc.Range(0, 0);



range.InsertAfter("Testing, testing, testing. . .");
```

Even this small change increases the readability of this snippet. Of course, you may not always want to use all the defaults. And yet, you still don't want to add all the Type.Missing parameters in the middle. Suppose you wanted to create a new Web page instead of new Word document. That's the last parameter of four in the Add() method. Using named parameters, you can specify just that last parameter:

```
var wordApp = new

   Microsoft.Office.Interop.Word.Application();

wordApp.Visible = true;

Documents docs = wordApp.Documents;



object docType = WdNewDocumentType.wdNewWebPage;
```

```
Document doc = docs.Add(DocumentType : ref docType);


Range range = doc.Range(0, 0);


range.InsertAfter("Testing, testing, testing. . .");
```

Named parameters mean that in any API with default parameters, you only need to specify those parameters you intend to use. It's simpler than multiple overloads. In fact, with four different parameters, you would need to create 15 different overloads of the Add() method to achieve the same level of flexibility that named and optional parameters provide. Remember that some of the Office APIs have as many as 16 parameters, and optional and named parameters are a big help.

I left the ref decorator in the parameter list, but another change in C# 4.0 makes that optional in COM scenarios. That's because COM, in general, passes objects by reference, so almost all parameters are passed by reference, even if they aren't modified by the called method. In fact, the Range() call passes the values (0,0) by reference. I did not include the ref modifier there, because that would be clearly misleading. In fact, in most production code, I would not include the ref modifier on the call to Add() either. I did above so that you could see the actual API signature.

Of course, just because the justification for named and optional parameters was COM and the Office APIs, that doesn't mean you should limit their use to Office interop applications. In fact, you can't. Developers calling your API can decorate calling locations using named parameters whether you want them to or not.

This method:

```
private void SetName(string lastName, string firstName)

{

   // elided

}
```

Can be called using named parameters to avoid any confusion on the order:

```
SetName(lastName: "Wagner", firstName: "Bill");
```

Annotating the names of the parameters ensures that people reading this code later won't wonder if the parameters are in the right order or not. Developers will use named parameters whenever adding the names will increase the clarity of the code someone is trying to read. Anytime you use methods that contain multiple parameters of the same type, naming the parameters at the callsite will make your code more readable.

Changing parameter names manifests itself in an interesting way as a breaking change. The parameter names are stored in the MSIL only at the callsite, not at the calling site. You can change parameter names and release the component without breaking any users of your component in the field. The developers who use your component will see a breaking change when they go to compile against the updated version, but any earlier client assemblies will continue to run correctly. So at least you won't break existing applications in the field. The developers who use your work will still be upset, but they won't blame you for problems in the field. For example, suppose you modify SetName() by changing the parameter names:

public void SetName(string Last, string First)

You could compile and release this assembly as a patch into the field. Any assemblies that called this method would continue to run, even if they contain calls to SetName that specify named parameters. However, when client developers went to build updates to their assemblies, any code like this would no longer compile:

SetName(lastName: "Wagner", firstName: "Bill");

The parameter names have changed.

Changing the default value also requires callers to recompile in order to pick up those changes. If you compile your assembly and release it as a patch, all existing callers would continue to use the previous default parameter.

Of course, you don't want to upset the developers who use your components either. For that reason, you must consider the names of your parameters as part of the public interface to your component. Changing the names of parameters will break client code at compile time.

In addition, adding parameters (even if they have default values) will break at runtime. Optional parameters are implemented in a similar fashion to named parameters. The callsite will contain annotations in the MSIL that reflect the existence of default values, and what those default values are. The calling site substitutes those values for any optional parameters the caller did not explicitly specify.

Therefore, adding parameters, even if they are optional parameters, is a breaking change at runtime. If they have default values, it's not a breaking change at compile time.

Now, after that explanation, the guidance should be clearer. For your initial release, use optional and named parameters to create whatever combination of overloads your users may want to use. However, once you start creating future releases, you must create overloads for additional parameters. That way, existing client applications will still function. Furthermore, in any future release, avoid changing parameter names. They are now part of your public interface.