Justification for Names and Optional Parameters

By **Bill Wagner**

March 2012

Many developers ask why named and optional parameters were not added earlier to the C# language. As other languages have shown, these are useful features. They are particularly useful when you might have a very large number of parameters on a method and that method has reasonable defaults for many of those parameters. The COM based Office APIs are the obvious example.

The C# language did not add this feature until version 4 because of other more pressing requests. In addition, named and optional parameters, as useful as they are, can lead to many different problems. Those problems become even more pronounced as you introduce the interaction between optional parameters and method overloads. You may have heard C# team members describe that each new language feature starts with negative points, and must be compelling enough to earn its way to a large positive value. In this article, I'll discuss many of the ways where named and optional parameters constrain future component releases and complicate code understanding. You'll learn how to best avoid these pitfalls in your own code and a little about why this feature was added as late as it was in C#'s evolution.

Method Resolution Rules

This method signature:

```
public static IEnumerable<Record> ApplyFilters(
   NameFilter nameFilter = default(NameFilter),
   CityFilter cityFilter = default(CityFilter),
   AgeFilter ageFilter = default(AgeFilter)
   )
```

Looks like it is emulating multiple overloaded versions. You may think it is a simplified way to create many overloads for a method. In your mind you may be thinking that you've created all these methods for your users:

```
public static IEnumerable<Record> ApplyFilters(
   // no parms means no filter.
   )
public static IEnumerable<Record> ApplyFilters(
   NameFilter nameFilter = default(NameFilter)
   )
public static IEnumerable<Record> ApplyFilters(
   CityFilter cityFilter = default(CityFilter)
```

```
    )
public static IEnumerable<Record> ApplyFilters(
    AgeFilter ageFilter = default(AgeFilter)
    )
public static IEnumerable<Record> ApplyFilters(
    NameFilter nameFilter = default(NameFilter),
    CityFilter cityFilter = default(CityFilter),
    )
public static IEnumerable<Record> ApplyFilters(
    NameFilter nameFilter = default(NameFilter),
    AgeFilter ageFilter = default(AgeFilter)
    )
public static IEnumerable<Record> ApplyFilters(
    CityFilter cityFilter = default(CityFilter),
    AgeFilter ageFilter = default(AgeFilter)
    )
public static IEnumerable<Record> ApplyFilters(
    NameFilter nameFilter = default(NameFilter),
    CityFilter cityFilter = default(CityFilter),
    AgeFilter ageFilter = default(AgeFilter)
    )
```

While it may appear that way, that's not what happens. The compiler instead realizes that the ApplyFilters method has optional parameters and adds the default values for those optional parameters to the method call. That's why a situation like this:

```
public static IEnumerable<Record> ApplyFilters(
    NameFilter nameFilter
    )
public static IEnumerable<Record> ApplyFilters(
    NameFilter nameFilter = default(NameFilter),
    CityFilter cityFilter = default(CityFilter),
    AgeFilter ageFilter = default(AgeFilter)
    )
// sample call:
ApplyFilters(new NameFilter());
```

does not create an ambiguity. The language rules on method resolution indicate that an exact match for the supplied parameter list is a better match than a method that uses optional parameters. Therefore, the first method is the better match. So far, that's fairly simple. Most developers know that rather well. But we've only scratched the surface. Now, consider these two methods in a class, and the following call to one of those methods:

```
// assume NameFilter, CityFilter, and AgeFilter all derive from
// an abstract Filter class.
public static IEnumerable<Record> ApplyFilters(
    NameFilter nameFilter = default(NameFilter),
    CityFilter cityFilter = default(CityFilter),
    AgeFilter ageFilter = default(AgeFilter)
    )
{
    Console.WriteLine("First version");
    return null;
}

public static IEnumerable<Record> ApplyFilters(
    Filter filter
    )
{
    Console.WriteLine("second version");
    return null;
}

// sample call:
ApplyFilters(new NameFilter());
ApplyFilters(new CityFilter());
```

Now, the parameters in the calls do not exactly match the parameters types of the formal argument in the second overload. In the first call, the parameter that is supplied does exactly match the first formal parameter for the first overload. Now which method should the compiler pick? The first call resolves to the first method; the second call resolves to the second method. This rule is explained in section 7.5.3.2 of the C# Specification (4th edition). The language specifies that for purposes of overload resolution, any optional parameters for which no value has been specified are removed from the signature. For the purpose of choosing the better method, the compiler must imagine the two overloads have the following signatures:

```
public static IEnumerable<Record> ApplyFilters(
    NameFilter nameFilter = default(NameFilter),
    )
public static IEnumerable<Record> ApplyFilters(
    Filter filter
    )
```

Between those two methods, the first method is a better match for the first method call. This decision makes it a little easier for a developer to use the methods with optional parameters. If the parameters supplied are a better match for a method with optional parameters, that method is chosen. If the language specified that the number of parameters as the overriding rule, optional parameters would be less useful. Callers would need to explicitly specify each parameter in order to specify that method. As designed, the method with optional parameters is the better match if the parameters supplied are better matches.

If you want to force the compiler to call the other method you specify the parameter by its name:

```
// NameFilter, CityFilter, and AgeFilter all derive from
// an abstract Filter class.
public static IEnumerable<Record> ApplyFilters(
   NameFilter nameFilter = default(NameFilter),
   CityFilter cityFilter = default(CityFilter),
   AgeFilter ageFilter = default(AgeFilter)
   )
public static IEnumerable<Record> ApplyFilters(
   Filter filter
   )

ApplyFilters(filter: new NameFilter());
ApplyFilters(cityFilter: new CityFilter());
```

This works because of another point in Section 7.5.3.2 of the C# Language Specification:

The parameters are reordered so that they occur at the same position as the corresponding argument in the argument list.

That means the second call must be transformed into this:

```
ApplyFilters(default(NameFilter), cityFilter: new CityFilter());
```

That operation puts the parameters in the proper order for the argument list. Once that operation occurs, you can easily see which method is the better method. The logic behind this choice is that the eveloper explicitly named the parameter, and clearly made a choice that a particular method should be called.

Using named and optional parameters gets more complicated as you mix it with a class hierarchy and virtual methods. Consider this hierarchy:

```
public abstract class Animal
{
   public abstract void Feed(string food = "chow");
}

public class Cat : Animal
{
   public override void Feed(string catfood = "cat chow")
   {
      Console.WriteLine(catfood);
   }
}

public class Dog : Animal
{
   public override void Feed(string dogfood = "dog chow")
```

```
    {
        Console.WriteLine(dogfood);
    }
}
```

Consider this set of example calls:

```
var d = new Dog();
d.Feed();
var c = new Cat();
c.Feed();
Animal thing = new Dog();
thing.Feed();
```

d.Feed() feeds “dog chow” to the dog. c.Feed() feeds “cat chow” to the cat. thing.Feed() feeds “chow” to this second dog. Let’s examine why that is.

A quick examination of the code should explain that it is because of the static (or compile time type) of the variable ‘thing’. Thing is an Animal, and therefore the value of the optional parameter for the Feed() method comes from the method declaration in the Animal class. That’s true even though the Animal class, and the Feed method are abstract.

Now, let’s complicate this one more time. Add a new overload of Feed to the dog class:

```
public void Feed(string dogfood = "dog chow", bool moist = false)
{
    Console.WriteLine("{0} {1}", moist ? "moist" : "dry", dogfood);
}
```

Now what happens? You may find it surprising that the compiler chooses the method with the extra optional parameter. This is because it is consistent with the other rules for method overload resolution. An application candidate from a more derived class is always considered better than an applicable candidate from a less derived class. That includes applying any optional parameters. That makes it consistent with the other method overload resolution rules in the language. The other Feed() method, the virtual method, is actually declared in the Animal class, not the Dog class. That means only the second Feed method is declared in the derived class, so that method wins. This is described in Section 7.6.5.1 of the Language Specification.

There are still several other rules that must be examined before we even consider the ramifications of upgrading a component with these methods declared. Now, let’s consider a virtual method with some optional parameters. Consider this change to the Cat class:

```
public class Cat : Animal
{
    public override void Feed(string catfood) // parameter is not optional
    {
        Console.WriteLine(catfood);
    }
}
// usage:
var c = new Cat();
c.Feed();
```

This call to c.Feed() does not compile! The variable through which you're calling the method does not include the default value for any of the parameters. Therefore, the compiler does not consider that it could add those parameters to the list of formal parameters. However, if the variable is cast to the Animal class, the call works:

```
var c = new Cat();
((Animal)c).Feed();
```

Of course, the version of Feed in the Cat class is called because it is a virtual method.

You should re-declare any of the optional parameters on each override of a virtual method to avoid confusing client code this way. When you do that, it's critically important that you ensure the default value of that parameter is the same for each override. Otherwise, these two calls could use different values for the food parameter.

The rules that cover default parameters in interface methods are very similar to the rules for base class methods. That carries all the way through: If you create an overloaded method with the same name as a method in an interface, the overloaded method will be preferred to the interface method, but only if you've implemented the interface method explicitly.

Releasing new Versions

Now that we've covered the simple case, let's examine what happens when new versions of components are delivered. Remember that one of C#'s original goals was to be a 'component oriented language', meaning that well-written C# assemblies should be safely upgradeable on an individual basis. These new features in the C# language have increased the potential for breaking changes when you deliver upgraded components. With each of these changes, code behavior may change at compile time or runtime. Compile time breaks will show up only when developers using the component build an updated version. Runtime breaks will show up when users have the new component installed and run the application. Some changes can cause both.

Let's start with the obvious: changing the name of any parameter on a public, or protected method is a compile time breaking change. What many C# developers don't know is that this has been a breaking change for some time. Even though C# only added support for named and optional parameters in C# 4, other languages on the .NET Framework, have had this features for some time (most obviously, Visual Basic). Anyone using your component from one those languages would be affected by your changes earlier.

Next, of course, changing the values of default parameters creates a breaking change. This change could cause breaks at either compile time, or runtime, depending on how you code the change. The default value of an optional parameter is inserted by the compiler at the callsite. At runtime, any methods that were compiled with the previous version of the component will continue to have the previous value inserted at the callsite. However, when you recompile any caller, the default value will change. It's really impossible to avoid this as a breaking change, you must pick whether the break is discovered at compile time, or runtime. Method calls compiled before the update will continue to use the previous default value; method calls recompiled with the new version of the component will use the new default value. How your method interprets the old and new values of the optional parameter determines which version has changed behavior.

All those issues I brought up earlier regarding method overloads, methods declared in base classes, and methods declared in interface methods all apply here as well. Modifying the default values of any parameter or the number of default parameters will affect the better method choice made by the compiler. That won't have any runtime breaking changes, but could potentially introduce any number of breaking compile time changes. Of course, calling a different method could change the runtime behavior as well.

Caution, not Prohibition

All of this is not to say that you should never define optional parameters, or use named and optional parameters in your calling code. After all, they are part of the language. You should use the features where they make sense. Most importantly, you must keep some of the pitfalls in mind so that you avoid them. If you find that you are sprinkling many optional parameters in your methods, maybe that's a sign that you should work a little harder on your API design. I recommend using the optional parameters sparingly. Ask yourself if the default value may change over time, or if the method may require new overloads over time. Most importantly, avoid optional parameters on virtual methods, or interface method definitions.