Language and Dynamic Changes

There are some welcome changes to C# and VB.NET, and major enhancements to the Common Language Runtime (CLR) and Base Class Library (BCL) in .NET 4.0. I have separated these changes into two chapters: language (this chapter) and CLR and BCL (Chapter 4), although there is of course some overlap.

In this chapter I will be covering the following:

- Future co-evolution of VB and C#
- Changes to C# and VB
- Improved COM interoperability
- Variance
- Dynamic code
- F#

Future Co-evolution of VB and C#

All .NET languages compile to the same IL code, so there is really no reason a feature should be present in one language and not another. Traditionally, C# got most of the new features, but not anymore. Microsoft has stated that it will now aim to ensure that VB and C# contain the same functionality. VB.NET and C# will be kept in sync like an elastic band: when a new feature is introduced in one language, it will be brought into the other language in the next release pinging the two into line. There will of course continue to be style differences between the languages, so don’t expect to see semi-colons appearing in VB because that would be very silly.

This release tries to sync the two languages, fixes an old constraint, and introduces some great dynamic functionality. No longer will VB.NET developers miss out on anonymous methods and C# developers will now benefit from named and optional parameters.

C# Enhancements

C# 2010 introduces two useful features that have been present in VB for a long time: optional and named parameters (technically two separate features but often found together).
Named and Optional Parameters

Named parameters allow you to pass parameters into a function in any order and are near essential when using C#'s other new feature: optional parameters. To use a named parameter, simply specify the parameter name followed by a colon and then the value you are passing into a function. The following code illustrates passing the value 1 to a method's Copies parameter, COLOR to the ColorMode parameter, and readme.txt to DocumentName:

```csharp
Print(Copies:1,ColorMode:"COLOR",DocumentName:"readme.txt");
```

```csharp
static void Print(string ColorMode, string DocumentName, int Copies)
{
    ...
}
```

Optional parameters are created in C# by specifying a default value and must appear after required parameters:

```csharp
static void Print(int Copies=1, string ColorMode="Color", string DocumentName="") {...}
```

This method can then be called in a multitude of ways, some of which are shown here:

```
Print(1);
Print(1, "Color");
Print(1, "Color", "My doc");
Print(Copies: 1);
Print(ColorMode: "Color");
Print(DocumentName: "myDoc.txt");
Print(Copies: 1, ColorMode: "Color");
Print(Copies: 1, ColorMode: "Color", DocumentName: "myDoc.txt");
```

Optional parameters can make your code more readable and easier to maintain, and can reduce the amount of typing you have to do. They also can make it easier to work with COM objects (see the following section). For example, if we were creating a Print() method that accepts a number of different parameters, we no longer have to overload it with a number of methods, such as:

```
public void Print(string DocumentName)
{
    Print(DocumentName, 1, "COLOR");
}
```

```
public void Print(string DocumentName, int Copies)
{
    Print(DocumentName, Copies, "COLOR");
}
```

```
public void Print(string DocumentName, int Copies, string ColorMode)
{}
```

Optional parameters allow us to refine this as:

```
public void Print(string DocumentName, int Copies=1, string ColorMode="COLOR")
{...}
```