The Future of .NET Languages

By Damon Armstrong , published on 09 Jan 2009 Comments (1) PDF

The Future of .NET Languages

At this year's Professional Developers Conference (PDC 2008), Microsoft® was quite forthcoming with specifics about the future for developers on the Microsoft .NET® platform. In this article, we will take a quick look at some of the upcoming features in the .NET 4.0 Framework, the new Microsoft F#® language, and discuss the long-term vision Microsoft has for the .NET Platform in the years to come.

Co-Evolution for VB.NET and C#

One of the most prominent messages coming from Microsoft right now is geared towards Microsoft Visual Basic® .NET developers. VB.NET and Microsoft Visual C#® are both built on top of the Common Language Runtime (CLR), which means they both compile down into the same Common Intermediate Language (CIL). Since they both compile down to the same code, there should be no intrinsic benefit of one language over another. However, both languages are maintained by separate teams at Microsoft, and over the years this separation has led to a variety of language-specific features in both C# and VB.NET as the teams focus on different areas with their respective products. Many VB.NET developers feel that the most exciting new features appear in C# first and are only later introduced into VB.NET. Naturally, this has generated a bit of animosity in the VB.NET community.

Co-evolution is a promise from Microsoft that recognises VB.NET and C# as equally important languages, and guarantees that as new language features evolve, those features will be incorporated into both languages simultaneously. No longer will you need to second guess your decision to go with a particular language for want of a particular feature, and the debate between which language is "better" will be reduced back down to syntactic preference. And while C# developers will probably have a lingering superiority complex with which VB.NET developers will have to contend, all VB.NET developers need to do is remind the C# devs that it's all the same under the covers.

Introducing the Dynamic Language Runtime

Microsoft is acutely aware that the .NET Framework is not the only choice for building applications. All you have to do is take a quick glance around the development sphere and you'll find a number of language options, and that number is only expected to rise as domain-specific languages emerge. People are spending time and energy writing useful components in these languages, so the question is, how can you use a component written in another language without having to rewrite it in .NET?

Perhaps the most exciting new feature in the upcoming .NET 4.0 release is the Dynamic Language Runtime (DLR). In as much as the Common Language Runtime (CLR) provides a common platform for statically typed languages like VB.NET and C#, the Dynamic Language Runtime provides a common platform for dynamically typed languages like JavaScript, Ruby, Python, and even legacy COM components. It represents a major leap forward in language interoperability for the .NET Framework, providing an abstraction of language operations, shared memory space to avoid marshalling data back and forth between processes, a common set of language features like garbage collection, and the plumbing to convert different representations of data from one language to another.

At a high level, you can think of the Dynamic Language Runtime as having three layers (see figure 1 below):

- .NET Language Integration
- DLR Core Components
- Language Binders

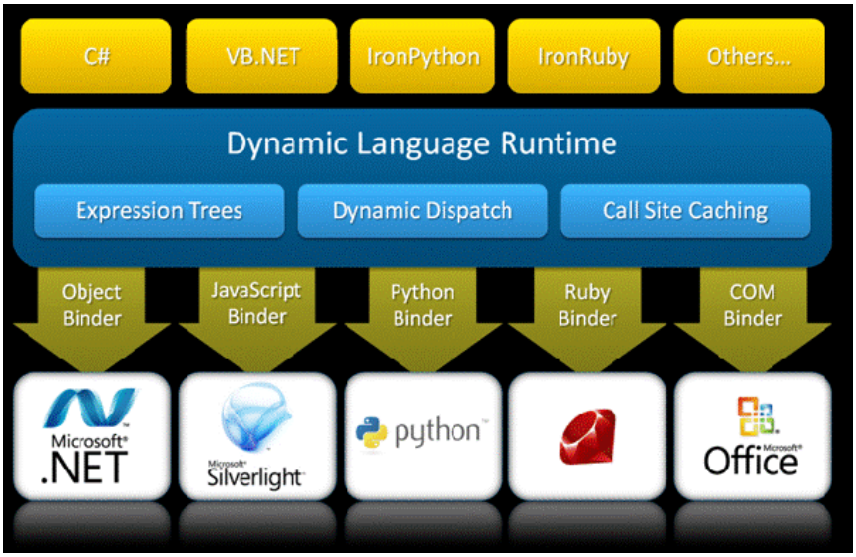

Figure 1. Dynamic Language Runtime (Slide courtesy of Anders Hejlsberg and Jim Hugunin's presentation at PDC 2008.)

The first layer, .NET Language Integration, simply represents the need for .NET languages to have a notion of what

the DLR is and how to use it. For the most part, you won't even notice this aspect of the DLR because most of the .NET languages had a natural integration point. IronRuby and IronPython are both dynamically typed languages, so the DLR fit right in. VB.NET has always supported the notion of late binding on the **Object** type, so the DLR incorporated nicely into late binding resolution. C#, however, has no notion of late binding and needed an additional static type for dynamic language support. It's called the **dynamic** type, and we'll talk about it in more detail a bit later.

The second layer is the Dynamic Language Runtime itself, which consists of three core components: Expression Trees, Dynamic Dispatch, and Call Site Caching. An Expression Tree is a representation of code in the form of a tree, which helps abstract languages into a consistent format on which the DLR can operate. Once dynamic code is in a tree representation, the DLR can look at the tree and generate CLR code from that tree for actual execution. Parsing dynamic code into an expression tree and then building the CLR is an expensive operation, so the DLR employs a performance technique known as Call Site Caching to avoid having to "recompile" the dynamic code each time it's called. Dynamic Dispatch ensures that appropriate Language Binders are used for dynamic invocations.

Language Binders, which make up the third layer, are language-specific implementations of certain operations the Dynamic Language Runtime needs to understand about each language that wishes to participate in the DLR.

Of course, the DLR is far more detailed than this brief overview can provide. For more information on all of its intricacies, please watch Jim Hugunin's PDC talk on Dynamic Languages in .NET.

New Language Features in .NET 4.0

Each iteration of the .NET Framework brings more features into the language set with the intention of making.NET more powerful and easier to use, and the .NET 4.0 release continues the trend. As we look through the individual features, you should begin to see how Microsoft is incorporating language features from C# to VB.NET and vice-versa in pursuit of their promise of co-evolution.

Dynamic Lookup (New to C#)

As mentioned earlier, C# added a new static type named dynamic. Although there is a lot that goes into making this new type work, there's not a lot that goes into using it. You can think of the dynamic type as an object that supports late binding.

```
dynamic Car = GetCar();               //Get a reference to the dynamic object
Car.Model = "Mondeo";                 //Assign a value to a property (also wo
Car.StartEngine();                    //Calling a method
Car.Accelerate(100);                  //Call a methods with parameters
dynamic Person = Car["Driver"];       //Getting with an indexer
Car["Passenger"] = CreatePassenger(); //Setting with an indexer
```

At compile time, calls to fields, properties, and methods of dynamic objects are essentially ignored. In other words, you won't get a compiler error about a member not being available. Since that information is only available at runtime, .NET knows to use the DLR to resolve the dynamic members. Now, C# is a statically typed language for a reason, and that reason is performance. The new dynamic type is not a licence to forgo static typing altogether; it's just another tool in case you need to interact with dynamic types. Also remember that VB.NET already supports Dynamic Lookups.

Named and Optional Parameters (New to C#)

Named and optional parameters have been around in VB.NET for quite some time, and they are finally being incorporated into C#. As the name suggests, optional parameters are parameters that you can opt to pass into a method or constructor. If you opt not to pass a parameter, the method simply uses the default value defined for the parameter. To make a method parameter optional in C#, you just give it a default value:

```
public void CreateBook(string title="No Title", int pageCount=0, string isbn
{
    this.Title = title;
    this.PageCount = pageCount;
    this.ISBN = isbn;
}
```

You can call the CreateBook method defined above in the following ways:

```
CreateBook();
CreateBook("Some Book Title");
CreateBook("Some Book Title", 600);
CreateBook("Some Book Title", 600, "5-55555-555-5");
```

Notice that optional parameters are dependent on position. Title must appear as the first parameter, page count as the second, and ISBN as the third. If you wanted to call CreateBook and only pass in the ISBN number, then you have two options. The first is to create an overloaded method that accepts ISBN as a parameter, which is the time tested and tedious option. The second is to use named parameters, which is much more succinct. Named parameters allow you to pass parameters into a method in any order you want, as long as you provide the name of the parameter. So you can call the method in the following ways:

```
CreateBook(isbn: "5-55555-5555-5");
CreateBook("Book Title", isbn: "5-55555-5555-5");
CreateBook(isbn: "5-55555-5555-5", title: "Book Title", pageCount: 600);
```

Please note that you can start with positional parameters, and then start using named parameters as shown in the second Create Book method call above, but once you start using Named Parameters you have to keep using them.

Anonymous Method Support (New to VB.NET)

Another long sought feature being introduced into VB.NET is the Inline or Anonymous Method. Anonymous Method is an appropriate name because it allows you to define Subs and Functions without needing to add a top-level method to your class, which keeps the method hidden (i.e. anonymous). Anonymous Methods also have access to all of the variables available to the code block in which the inline method resides, so you don't even have to bother defining method parameters to get data into and out of an anonymous method. You can define an anonymous method anywhere you would normally use the AddressOf keyword to point to a method, so the most prominent use is likely going to be on event handles, as demonstrated in the following example:

```
Dim MyTimer As New System.Timers.Timer(1000)
Dim Seconds As Integer = 0

AddHandler MyTimer.Elapsed,
    Sub()
        Seconds += 1
        Console.WriteLine(Seconds.ToString() & " seconds have elapsed")
    End Sub

MyTimer.Start()
Console.WriteLine("Press any key to exit")
Console.ReadLine()
```

Notice that the event handler for the Timer's Elapsed event is written inline, and the method has access to the Seconds variable defined outside of the inline sub. You can also define inline functions:

```
Dim f = Function(a As Integer, b As Integer)
            Return a + b
        End Function

Dim x = 10
Dim y = 20
Dim z = f(x, y)
```

Inline functions are great if a function makes sense in the context of a code block, but isn't really reusable enough to make it a class-level function.

Co-variance and Contra-variance (New to C# and VB.NET)

One of the most annoying issues with generics has been resolved in .NET 4.0. Previously, if you had an object that supports `IEnumerable<String>` and you wanted to pass it into a method requiring an `IEnumerable<object>` parameter, you just couldn't do it. You would have to make a new object that supports `IEnumerable<object>`, populate it with the strings from your IEnumerable instance, and then pass it into the method. We all know that a string is more specific than an object, so in our minds a `List<string>` should support the `IEnumerable<string>` interface and the `IEnumerable<object>` interface as well. Regardless, the compiler would have nothing of this. In .NET 4.0, the issue has been resolved because generics now support Co-variance and Contra-variance.

Co-variance and Contra-variance are really about type safety and performance. Without getting into too much detail, Co-variance means that an object can be treated as less derived, and is represented by decorating a generic type parameter with the out keyword. Co-variant types are restricted to output positions only, meaning they can appear as results of a method call or results of a property accessor. These are the only places where a co-variant type is "safe", or rather the only place where they can appear without having to inject additional type checks during compilation. In .NET 4.0, the `IEnumerable<T>` interface is now `IEnumerable<out T>` because IEnumerable is co-variant. This means that the following example is perfectly valid:

```
IEnumerable<string> strings = GetStrings();
IEnumerable<object> objects = strings;
```

Contra-variance means that an object can be treated as more derived, and is represented by decorating a generic type parameter with the in keyword. Contra-variant types are restricted to input positions, meaning they can only appear as method parameters or write-only properties. In .NET 4.0, the `IComparer<T>` inteface is now `IComparer<in T>` because IComparer is contra-variant. It's a difficult concept to wrap your head around, but know that in the end it has alleviated some of issues with casting from one generic type to another.

Dynamic Import (New to C#)

Almost all of the components exposed through COM APIs are variant data types, which have traditionally been represented in C# by the object data type. Previously, C# had no way to deal with dynamic types, so working with these objects quickly became an exercise in casting just about everything. Now that C# supports dynamic types, you have the option of importing COM components as dynamic objects, allowing you to set properties and make method calls without having to explicitly cast the object.

Omitting Ref Parameters (New to C#)

Another by-product of COM APIs is that a great number of method parameters have to be passed by reference. Most of the time you really just want to pass a value into the method and you don't really care what comes out. Nonetheless, you still have to create a number of temporary values to hold the result. The tedium of this task could easily be passed to an intern to do claiming its "real world experience". In C# 4.0, you can now pass parameters by value into COM method calls and the compiler will automatically generate temporary variables for you, which saves a great deal of time, and saves the intern for other "real world" tasks.

Compiling without Primary Interop Assemblies (New to C# and VB.NET)

Primary Interop Assemblies (PIA) are vendor-provided assemblies that act as a go between for COM components and the .NET Framework, one of the most widely known being the Microsoft Office Primary Interop Assemblies. Any time you deploy an assembly that references a PIA, you have to remember to deploy the PIA along with it or provide

instructions for how to go about getting it on the system. A new feature for both C# and VB.NET allows you to embed a Primary Interop Assembly directly into your assembly, which makes deployment extremely simple. PIAs also tend to be relatively large, so including the whole thing could easily bloat your assembly. Fortunately, the compiler is optimised to embed only those parts of the PIA that you actually use, which can reduce the PIA footprint if you are only using a subset of what it contains.

Implicit Line Continuation (New to VB.NET)

When you look at C# code it's easy to determine where a statement ends because it terminates with a semicolon. VB also has a statement terminator, but it's the carriage return; each code statement is assumed to be on a single line. Any time you break that norm in VB.NET, you've always had to use an underscore at the end of the line to inform the compiler that the statement continues on the next line. If you work in VB.NET then you know it's a hassle to type and it makes for some ugly code:

```
Dim text As String = "Wouldn&#39;t it be nice" & _
            "If you didn&#39;t have to" & _
            "put an underscore to" & _
            "continue to the next line?"
```

Alas, it is now a thing of the past. VB.NET now supports implicit line continuation. When the compiler encounters half of a statement on a single line, it's smart enough to check the next line to see if the rest of the statement is present.

```
Dim text As String = "Wouldn&#39;t it be nice" &
            "If you didn&#39;t have to" &
            "put an underscore to" &
            "continue to the next line?" &
            "Sweet! Now you can!"
```

If you are feeling nostalgic you can still use the explicit line continuation character, because it's still around. And you may need it because the VB.NET team has identified a few scenarios where the compiler has trouble determining whether a line should continue or not. Rest assured that those scenarios should be rare, and I'm sure the compiler will let you know about it if it does occur.

Simplified Property Syntax (New to VB.NET)

Another C# feature to make its way into VB.NET in this release is simplified property syntax. Property definitions that once looked like this:

```
'Field
Private _name As String

'Property
Public Property Name() As String
    Get
        Return _name
    End Get
    Set(ByVal value As String)
        _name = value
    End Set
End Property
```

Can now be reduced to:

```
Public Property Name() as String
```

This reduces the total lines of code you are responsible for writing from 9 down to 1. If you do opt for this syntax, note that you will not have access to the field which stores the value for the property, which might be an issue if you need to pass the property value by reference into a method. If this happens, you can always revert back to the expanded syntax or just use a temporary variable.

Array Type Inference and Jagged Arrays (New to VB.NET)

VB.NET now sports array type inference and a syntax for jagged arrays definition. This means you don't have to explicitly declare the type of an array if you are initialising it with values because the compiler will figure it out. For example,

```
Dim Numbers = {1, 1, 2, 3, 5, 8, 13, 21, 34}
```

When you look at this array you can quickly determine that it holds integers, and the compiler is now smart enough to make that determination as well.

```
Dim Numbers = {1, 1, 2, 3, 5.0, 8, 13, 21, 34}
```

When the compiler sees the previous example, it notices that 5.0 is not an integer, so the array becomes an array of doubles. Type inference also works with matrices:

```
Dim Names = {{"Sarah", "Jane", "Mary", "Susan", "Amanda"},
             {"Bob", "Joe", "Dustin", "Richard", "Nick"}}
```

The compiler can infer that string(,) is the appropriate type for the example above. You do run into a bit of issue with jagged arrays. You can think of a two dimensional matrix as having rows and columns and a matrix always has the

same number of columns in each row. A jagged array can have a variable number of columns on each row, so it's a bit different than a matrix. You would think you could define a jagged array such as:

```
Dim Names = {"Sarah", "Jane", "Mary", "Susan", "Amanda"},
            "Bob", "Nick"}
```

But it throws a compiler error stating that the "Array initialiser is missing 3 elements" because it assumes everything is a matrix. If you want to define a jagged array, you just need to surround your "rows" with braces:

```
Dim Names = {{"Sarah", "Jane", "Mary", "Susan", "Amanda"},
             {"Bob", "Nick"}}
```

And the compiler now infers that the type is string()(), which is appropriate for a jagged array.

From Keyword (New to VB.NET)

While we're on the topic of initialisation, let's talk about the new From keyword in VB.NET. When you create a dictionary, or a list, or really any object that is responsible for holding a collection of objects, you normally create the collection object and then you populate it with a bunch of items. Instead of having to repeatedly call the Add method to populate a list, you can now use From, which essentially calls the Add method on the object for you. So, instead of having to type out all of this:

```
Dim Colors As New List(Of String)
Colors.Add("Red")
Colors.Add("Green")
Colors.Add("Blue")
```

It can now be reduced down to this:

```
Dim Colors As New List(Of String) From {"Red", "Green", "Blue"}
```

Make no mistake, it is really calling the Add method on the object, which means that it works with any object that has an Add method. In fact, you can even use extension methods to create an Add method or overload an Add method and the From keyword will use it if the method signature matches the incoming parameters. In the previous example, the List object has an add method that accepts a single parameter, the string you want added to a list. If you have an Add method with multiple parameters, you just pass in the parameters using syntax much like defining a matrix. Here is an example that shows how to use the Add method with a Dictionary object:

```
Dim Colors2 As New Dictionary(Of String, String) From {
     {"Red", "FF0000"},
     {"Green", "00FF00"},
     {"Blue", "0000FF"}}
```

Since the Add method on a Dictionary contains two parameters, the key and the value, we have to pass in a set of two parameters for each item in the From statement. Remember, however, to take readability into account when using the From keyword. You may even want to think about reverting back to the Add method if it makes sense in a particular situation.

Functional Programming, Parallelism and beyond

Functional Programming with F#

F# is a succinct, high performance, type-inferred, functional language built on top of the .NET Framework. Microsoft has a solid base of imperative programming languages with VB.NET and C#, but there is a trend in computing that tends to be moving towards a more declarative style of programming. What's the difference? In an imperative language you write code that tells the compiler exactly how to do something, whereas in a declarative language you write code that says what you want to do, but leave the "how" part up to the compiler. Now, the ultimate declarative language would allow you to write something like "Morph the screen into something cool" and then compile your thoughts into a wicked screen saver or some such, but we're not there just yet. F# offers developers the opportunity to explore declarative concepts and offer a useful language to customers whose thinking is geared more towards functional development.

Another thought that probably comes to mind, is what does this mean for me? VB.NET and C# have already introduced some declarative concepts like LINQ. , In the years to come you're going to see more declarative features introduced into both languages. You may want to take a look at F# to familiarise yourself with declarative concepts so they're easier to absorb when they suddenly show up in the languages you use. For a deeper look at F#, check out the following resources:

- Microsoft F# Developers Center
- F# Hompage (Microsoft Research)
- An Introduction to Microsoft F# (PDC Video)

Parallelism

Processor speeds have started to level off, which means we can no longer rely on the latest and greatest chip to give our code a performance boost. Chipmakers have, however, continued the trend of putting more and more transistors on their chips, so the number of processors they can pack on a chip will continue to rise for the foreseeable future. But having all of the processors in the world is useless unless you have a way to take advantage of them all. To do that, developers need a way to incorporate parallel computing concepts into their applications.

Here's an example to accentuate the point. Let's say you have two applications running on a single processor. Assuming they have the same priority, they should share the processor equally, so both applications end up with about 50 percent of the processor and run at half speed. Now let's say you add another processor which allows each

application to run on its own dedicated processor. It should result in an instant performance gain because they can both run at 100 percent without interfering with one another. But what happens if you take it a step further and add two more processors beyond that? Would it increase performance yet again? Unfortunately, the answer is likely no. You'll just end up with two idle processors.

Parallel computing is about breaking down large tasks into smaller tasks that can be processed individually. When you break down a task in this way, you can divide up the work between whatever resources you have at your disposal. If you have one processor and sixteen tasks, then you run all sixteen tasks on the same processor. If you have sixteen processors, then you can run one task on each processor and get work done much more quickly. In this scenario, adding more processors really can speed up an application. Microsoft is committed to making parallel computing capabilities more accessible to .NET developers by offering a development model that offloads the complexities of high performance computing. Developers are then free to focus on solving business needs, knowing that their solutions are built on a framework that scales along with the hardware.

Microsoft is laying a great foundation for parallel computing with the Parallel Extensions to the .NET Framework, which are included in .NET 4.0. It introduces the concept of tasks, which uses an enhanced thread pool to achieve some extremely compelling performance benefits over threads. It also provides the Parallel class, which contains static methods that help parallelise for loops, for each loops, and can even execute individual method calls in parallel. And it even includes Parallel LINQ, or PLINQ, which introduces syntax for running LINQ queries in parallel.

For a more in depth look at Parallelism, you can always check out these links:

- Parallel Programming for Managed Developers with Visual Studio 2010
- How to Use the Static Parallel Class
- Running Queries on Multi-Core Processors
- Taking Parallelism Mainstream

Looking at the Future of .NET

People always like to know what the future holds because our expectation of the future is what guides our actions in the present. And right now, it looks like .NET is positioned to give developers a leg up in the development arena. Languages are getting easier to use, the DLR provides a mechanism for talking to just about any language out there, and the complexities of parallel computing are being reigned in. In all, the future is looking even better for .NET developers.

Damon Armstrong is a senior architect with Cogent Company in Dallas, Texas, and author of 'Pro ASP.NET 2.0 Website Programming'. He specializes in Microsoft technologies with a focus on Web application design using ASP.NET. When not staying up all night coding, he can be found playing disc golf, softball, working on something for Carrollton Young Life, or recovering from coding all night.

Related tags

.net, .net 4.0, dlr, dynamic language, functional, language features, parallel