Join the discussion @ p2p.wrox.com

BUILD WINDOWS 8 STYLE APPS WITH C#

Professional C# 2012 and .NET 4.5

Christian Nagel, Bill Evjen, Jay Glynn, Karli Watson, Morgan Skinner

CONTENTS

Part I

The C# Language

Chapter 1

.NET Architecture

WHAT’S IN THIS CHAPTER?

- Compiling and running code that targets .NET
- Advantages of Microsoft Intermediate Language (MSIL)
- Value and reference types
- Data typing
- Understanding error handling and attributes
- Assemblies, .NET base classes, and namespaces

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

There are no code downloads for this chapter.

THE RELATIONSHIP OF C# TO .NET

This book emphasizes that the C# language must be considered in parallel with the .NET Framework, rather than viewed in isolation. The C# compiler specifically targets .NET, which means that all code written in C# always runs within the .NET Framework. This has two important consequences for the C# language:

1. The architecture and methodologies of C# reflect the underlying methodologies of .NET.
2. In many cases, specific language features of C# actually depend on features of .NET or of the .NET base

classes.

Because of this dependence, you must gain some understanding of the architecture and methodology of .NET before you begin C# programming, which is the purpose of this chapter.

C# is a programming language newly designed for .NET. and is significant in two respects:

- It is specifically designed and targeted for use with Microsoft’s .NET Framework (a feature-rich platform for the development, deployment, and execution of distributed applications).
- It is a language based on the modern object-oriented design methodology, and when designing it Microsoft learned from the experience of all the other similar languages that have been around since object-oriented principles came to prominence 20 years ago.

C# is a language in its own right. Although it is designed to generate code that targets the .NET environment, it is not part of .NET. Some features are supported by .NET but not by C#, and you might be surprised to learn that some features of the C# language are not supported by .NET (for example, some instances of operator overloading).

However, because the C# language is intended for use with .NET, you must understand this Framework if you want to develop applications in C# effectively. Therefore, this chapter takes some time to peek underneath the surface of .NET.

THE COMMON LANGUAGE RUNTIME

Central to the .NET Framework is its runtime execution environment, known as the *Common Language Runtime*

(**CLR**) or the *.NET runtime*. Code running under the control of the CLR is often termed *managed code*.

However, before it can be executed by the CLR, any source code that you develop (in C# or some other language) needs to be compiled. Compilation occurs in two steps in .NET:

1. Compilation of source code to Microsoft Intermediate Language (IL).

2. Compilation of IL to platform-specific code by the CLR.

This two-stage compilation process is important because the existence of the Microsoft Intermediate Language is the key to providing many of the benefits of .NET.

IL shares with Java byte code the idea that it is a low-level language with a simple syntax (based on numeric codes rather than text), which can be quickly translated into native machine code. Having this well-defined universal syntax for code has significant advantages: platform independence, performance improvement, and language interoperability.

Platform Independence

First, platform independence means that the same file containing byte code instructions can be placed on any platform; at runtime, the final stage of compilation can then be easily accomplished so that the code can run on that particular platform. In other words, by compiling to IL you obtain platform independence for .NET in much the same way as compiling to Java byte code gives Java platform independence.

The platform independence of .NET is only theoretical at present because, at the time of writing, a complete implementation of .NET is available only for Windows. However, a partial, cross-platform implementation is available (see, for example, the Mono project, an effort to create an open source implementation of .NET, at www.go-mono.com).

Performance Improvement

Although previously compared to Java, IL is actually a bit more ambitious than Java byte code. IL is always *Just-in-Time* compiled (known as **JIT** compilation), whereas Java byte code was often interpreted. One of the disadvantages of Java was that, on execution, the process to translate from Java byte code to native executable resulted in a loss of performance (with the exception of more recent cases in which Java is JIT compiled on certain platforms).

Instead of compiling the entire application at one time (which could lead to a slow startup time), the JIT compiler simply compiles each portion of code as it is called (just in time). When code has been compiled once, the resultant native executable is stored until the application exits so that it does not need to be recompiled the next time that portion of code is run. Microsoft argues that this process is more efficient than compiling the entire application code at the start because of the likelihood that large portions of any application code will not actually be executed in any given run. Using the JIT compiler, such code can never be compiled.

This explains why you can expect that execution of managed IL code will be almost as fast as executing native machine code. What it does not explain is why Microsoft expects that you get a performance *improvement*. The reason given for this is that because the final stage of compilation takes place at runtime, the JIT compiler knows exactly what processor type the program runs on. This means that it can optimize the final executable code to take advantage of any features or particular machine code instructions offered by that particular processor.

Traditional compilers optimize the code, but they can perform optimizations that are only independent of the particular processor that the code runs on. This is because traditional compilers compile to native executable code before the software is shipped. This means that the compiler does not know what type of processor the code runs on beyond basic generalities, such as that it is an x86-compatible processor or an Alpha processor.

Language Interoperability

The use of IL not only enables platform independence, but it also facilitates *language interoperability*. Simply put, you can compile to IL from one language, and this compiled code should then be interoperable with code that has been compiled to IL from another language.

You are probably now wondering which languages aside from C# are interoperable with .NET. The following sections briefly discuss how some of the other common languages fit into .NET.

Visual Basic 2012

Visual Basic .NET 2002 underwent a complete revamp from Visual Basic 6 to bring it up to date with the first version of the .NET Framework. The Visual Basic language had dramatically evolved from VB6, which this meant that VB6 was not a suitable language to run .NET programs. For example, VB6 is heavily integrated into Component Object Model (COM) and works by exposing only event handlers as source code to the developer — most of the background code is not available as source code. Not only that, it does not support implementation inheritance, and the standard data types that Visual Basic 6 uses are incompatible with .NET.

Visual Basic 6 was upgraded to Visual Basic .NET in 2002, and the changes that were made to the language are so extensive you might as well regard Visual Basic as a new language. Existing Visual Basic 6 code does not compile to the present Visual Basic 2012 code (or to Visual Basic .NET 2002, 2003, 2005, 2008, and 2010 for that matter). Converting a Visual Basic 6 program to Visual Basic 2012 requires extensive changes to the code. However, Visual Studio 2012 (the upgrade of Visual Studio for use with .NET) can do most of the changes for you. If you attempt to read a Visual Basic 6 project into Visual Studio 2012, it can upgrade the project for you, which means that it can rewrite the Visual Basic 6 source code into Visual Basic 2012 source code. Although this means that the work involved for you is heavily reduced, you need to check through the new Visual Basic 2012 code to make sure that the project still works as intended because the conversion is not perfect.

One side effect of this language upgrade is that it is no longer possible to compile Visual Basic 2012 to native executable code. Visual Basic 2012 compiles only to IL, just as C# does. If you need to continue coding in Visual Basic 6, you can do so, but the executable code produced completely ignores the .NET Framework, and you need to keep Visual Studio 6 installed if you want to continue to work in this developer environment.

Visual C++ 2012

Visual C++ 6 already had a large number of Microsoft-specific extensions on Windows. With Visual C++ .NET, extensions have been added to support the .NET Framework. This means that existing C++ source code will continue to compile to native executable code without modification. It also means, however, that it will run independently of the .NET runtime. If you want your C++

code to run within the .NET Framework, you can simply add the following line to the beginning of your code:

```
#using <mscorlib.dll>
```

You can also pass the flag `/clr` to the compiler, which then assumes that you want to compile to managed code and will hence emit IL instead of native machine code. The interesting thing about C++ is that when you compile to managed code, the compiler can emit IL that contains an embedded native executable. This means that you can mix managed types and unmanaged types in your C++ code. Thus, the managed C++ code

```
class MyClass
{
```

defines a plain C++ class, whereas the code

```
ref class MyClass
{
```

gives you a managed class, just as if you had written the class in C# or Visual Basic 2012. The advantage to use managed C++ over C# code is that you can call unmanaged C++ classes from managed C++ code without resorting to COM interop.

The compiler raises an error if you attempt to use features not supported by .NET on managed types (for example, templates or multiple inheritances of classes). You can also find that you need to use nonstandard C++ features when using managed classes.

Writing C++ programs that uses .NET gives you different variants of interop scenarios. With the compiler setting /clr for Common Language Runtime Support, you can completely mix all native and managed C++ features. Other options such as /clr:safe and /clr:pure restrict the use of native C++ pointers and thus enable writing safe code like with C# and Visual Basic.

Visual C++ 2012 enables you to create programs for the Windows Runtime (WinRT) with Windows 8. This way C++

does not use managed code but instead accesses the WinRT natively.

COM and COM+

Technically speaking, COM and COM+ are not technologies targeted at .NET — components based on them cannot be compiled into IL. (Although you can do so to some degree using managed C++ if the original COM component were written in C++). However, COM+ remains an important tool because its features are not duplicated in .NET. Also, COM components can still work — and .NET incorporates COM interoperability features that make it possible for managed code to call up COM components and vice versa (discussed in Chapter 23, “Interop”). In general, you will probably find it more convenient for most purposes to code new components as .NET components so that you can take advantage of the .NET base classes and the other benefits of running as managed code.

Windows Runtime

Windows 8 offers a new runtime used by the new applications. You can use this runtime from Visual Basic, C#, C++, and JavaScript. When using the runtime with these different environments, it looks different. Using it from C# it looks like classes from the .NET Framework. Using it from JavaScript it looks like what JavaScript developers are used to with JavaScript libraries. And using it from C++, methods looks like the Standard C++ Library. This is done by using language projection. The Windows Runtime and how it looks like from C# is discussed in Chapter 31, “Windows Runtime.”

A CLOSER LOOK AT INTERMEDIATE LANGUAGE

From what you learned in the previous section, Microsoft Intermediate Language obviously plays a fundamental role in the .NET Framework. It makes sense now to take a closer look at the main features of IL because any language that targets .NET logically needs to support these characteristics.

Here are the important features of IL:

- Object orientation and the use of interfaces
- Strong distinction between value and reference types
- Strong data typing
- Error handling using exceptions
- Use of attributes

The following sections explore each of these features.

Support for Object Orientation and Interfaces

The language independence of .NET does have some practical limitations. IL is inevitably going to implement some particular programming methodology, which means that languages targeting it need to be compatible with that methodology. The particular route that Microsoft has chosen to follow for IL is that of classic object-oriented programming, with single implementation inheritance of classes.

In addition to classic object-oriented programming, IL also brings in the idea of interfaces, which saw their first implementation under Windows with COM. Interfaces built using .NET produce interfaces that are not the same as COM interfaces. They do not need to support any of the COM infrastructure. (For example, they are not derived from

`IUnknown` and do not have associated globally unique identifiers, more commonly known as GUIDs.) However, they do share with COM interfaces the idea that they provide a contract, and classes that implement a given interface must provide implementations of the methods and properties specified by that interface.

You have now seen that working with .NET means compiling to IL, and that in turn means that you need to use traditional object-oriented methodologies. However, that alone is not sufficient to give you language interoperability. After all, C++ and Java both use the same object-oriented paradigms but are still not regarded as interoperable. You need to look a little more closely at the concept of language interoperability.

So what exactly is language interoperability?

After all, COM enabled components written in different languages to work together in the sense of calling each other's methods. What was inadequate about that? COM, by virtue of being a binary standard, did enable components to instantiate other components and call methods or properties against them, without worrying about the language in which the respective components were written. To achieve this, however, each object had to be instantiated through the COM runtime and accessed through an interface. Depending on the threading models of the relative components, there may have been large performance losses associated with marshaling data between apartments or running components or both on different threads. In the extreme case of components hosted as an executable rather than DLL files, separate processes would need to be created to run them. The emphasis was very much that components could talk to each other but only via the COM runtime. In no way with COM did components written in different languages directly communicate with each other, or instantiate instances of each other — it was always done