CHAPTER 1

C# Preview

Since this is a book for experienced object-oriented developers, I assume that you already have some familiarity with the .NET runtime. *Essential .NET Volume 1: The Common Language Runtime* by Don Box (Boston, MA: Addison-Wesley, 2002) is an excellent book specifically covering the .NET runtime. Additionally, it's important to look at some of the similarities and differences between C# and C++, and then go through an elementary "Hello World!" example for good measure. If you already have experience building .NET applications, you may want to skip this chapter. However, you may want to read the section "Overview of What's New in C# 3.0."

Differences Between C# and C++

C# is a strongly typed object-oriented language whose code visually resembles C++ (and Java). This decision by the C# language designers allows C++ developers to easily leverage their knowledge to quickly become productive in C#. C# syntax differs from C++ in some ways, but most of the differences between these languages are semantic and behavioral, stemming from differences in the runtime environments in which they execute.

C#

C# source code compiles into *managed code*. Managed code, as you may already know, is an intermediate language (IL) because it is halfway between the high-level language (C#) and the lowest-level language (assembly/machine code). At run time, the Common Language Runtime (CLR) compiles the code on the fly by using Just In Time (JIT) compiling. As with just about anything in engineering, this technique comes with its pros and cons. It may seem that an obvious con is the inefficiency of compiling the code at run time. This process is different from interpreting, which is typically used by scripting languages such as Perl and JScript. The JIT compiler doesn't compile a function or method each and every time it's called; it does so only the first time, and when it does, it produces machine code native to the platform on which it's running. An obvious pro of JIT compiling is that the working set of the application is reduced, because the memory footprint of intermediate code is smaller. During the execution of the application, only the needed code is JIT-compiled. If your application contains printing code, for example, that code is not needed if the user never prints a document, and therefore the JIT compiler never compiles it. Moreover, the CLR can optimize the program's execution on the fly at run time. For example, the CLR may determine a way to reduce page faults in the memory manager by rearranging compiled code in memory, and it could do all this at run time. Once you weigh all the pros together, you find that they outweigh the cons for most applications.

Note Actually, you can choose to code your programs in raw IL while building them with the IL Assembler (ILASM). However, it will likely be an inefficient use of your time. High-level languages can nearly always provide any capability that you can achieve with raw IL code.

C++

Unlike C#, C++ code traditionally compiles into *native code*. Native code is the machine code that's native to the processor for which the program was compiled. For the sake of discussion, assume that we're talking about natively compiled C++ code rather than managed C++ which can be achieved by using C++/CLI. If you want your native C++ application to run on different platforms, such as on both a 32-bit platform and a 64-bit platform, you must compile it separately for each. The native binary output is generally not compatible across platforms.

IL, on the other hand, is compatible across platforms, because it, along with the Common Language Infrastructure (CLI) upon which the CLR is built, is a defined international standard.[1] This standard is rapidly gaining traction and being implemented beyond the Microsoft Windows platform.

Note I recommend you check out the work the Mono team has accomplished toward creating alternate, open source Virtual Execution Systems (VESs) on other platforms.[2]

Included in the CLI standard is the Portable Executable (PE) file format for managed modules. Therefore, you can actually compile a C# program on a Windows platform and execute the output on both Windows and Linux without having to recompile, because even the file format is standardized.[3] This degree of portability is extremely convenient and was in the hearts and minds of the COM/DCOM designers back in the day, but for various reasons, it failed to succeed across disparate platforms at this level.[4] One of the major reasons for that failure is that COM lacked a sufficiently expressive and extensible mechanism for describing types and their dependencies. The CLI specification solves this nicely by introducing metadata, which I'll describe in Chapter 2.

CLR Garbage Collection

One of the key facilities in the CLR is the *garbage collector* (GC). The GC frees you from the burden of handling memory allocation and deallocation, which is where many software errors can occur. However, the GC doesn't remove all resource-handling burdens from your plate, as you'll see in

---

1. You can find the CLI standard document `Ecma-335` at `www.ecma-international.org`. Additionally, `Ecma-334` is the standard document for the C# language.
2. You can find the Mono project on the Internet at `www.mono-project.com`.
3. Of course, the target platform must also have all of the dependent libraries installed. This is quickly becoming a reality, given the breadth of the .NET Standard Library. For example, check out `www.go-mono.com/docs/` to see how much coverage the Mono project libraries have.
4. For all the gory details, I recommend reading Don Box and Chris Sells' *Essential .NET, Volume I: The Common Language Runtime* (Boston, MA: Addison-Wesley Professional, 2002). (The title leads one to believe that Volume II is due out any time now, so let's hope it's not titled in the same vein as Mel Brooks' *History of the World: Part I.*)

Chapter 4. For example, a file handle is a resource that must be freed when the consumer is finished with it, just as memory must be freed in the same way. The GC handles only memory resources directly. To handle resources other than memory, such as database connections and file handles, you can use a finalizer (as I'll show you in Chapter 13) to free your resources when the GC notifies you that your object is being destroyed. However, an even better way is to use the Disposable pattern for this task, which I'll demonstrate in Chapters 4 and 13.

■ **Note** The CLR references all objects of *reference type* indirectly, similar to the way you use pointers and references in C++, except without the pointer syntax. When you declare a variable of a reference type in C#, you actually reserve a storage location that has a type associated with it, either on the heap or on the stack, which stores the reference to the object. So when you copy an object reference in one variable into another variable, you end up with two variables referencing the same object. All reference type instances live on the managed heap. The CLR manages the location of these objects, and if it needs to move them around, it updates all the outstanding references to the moved objects to point to the new location. Also, *value types* exist in the CLR, and instances of them live on the stack or as a field of an object on the managed heap. Their usage comes with many restrictions and nuances. You normally use them when you need a lightweight structure to manage some related data. Value types are also useful when modeling an immutable chunk of data. I cover this topic in much more detail in Chapter 4.

C# allows you to develop applications rapidly while dealing with fewer mundane details than in a C++ environment. At the same time, C# provides a language that feels familiar to either C++ or Java developers.

Example of a C# Program

Let's take a close look at a C# program. Consider the venerable "Hello World!" program that everyone knows and loves. A console version of it looks like this in C#:

```
class EntryPoint {
    static void Main() {
        System.Console.WriteLine( "Hello World!" );
    }
}
```

Note the structure of this C# program. It declares a type (a class named `EntryPoint`) and a member of that type (a method named `Main`). This differs from C++, where you declare a type in a header and define it in a separate compilation unit, usually a `.cpp` file. Also, metadata (which describes all of the types in a module and is generated transparently by the C# compiler) removes the need for the forward declarations and inclusions as required in C++. In fact, forward declarations don't even exist in C#.

C++ programmers will find the static `Main` method familiar, except for the fact that its name begins with a capital letter. Every program requires an entry point, and in the case of C#, it is the static `Main` method. There are some further differences. For example, the `Main` method is declared within a class (in this case, named `EntryPoint`). In C#, you must declare all methods within a type definition. There is no such thing as a static, free function as there is in C++. The return type for the `Main` method may be either of type `int` or `void`, depending on your needs. In my example, `Main` has no parameters, but if you need access to the command-line parameters, your `Main` method can declare a parameter (an array of strings) to access them.

■**Note** If your application contains multiple types with a static `Main` method, you can select which one to use via the `/main` compiler switch.

You may notice that the call to `WriteLine` seems verbose. I had to qualify the method name with the class name `Console`, and I also had to specify the namespace that the `Console` class lives in (in this case, `System`). .NET (and therefore C#) supports namespaces to avoid name collisions in the vast global namespace. However, instead of having to type the fully qualified name, including the namespace, every time, C# provides the `using` directive, which is analogous to Java's `import` and C++'s `using namespace`. So you could rewrite the previous program slightly, as Listing 1-1 shows.

Listing 1-1. *hello_world.cs*

```
using System;

class EntryPoint {
    static void Main() {
        Console.WriteLine( "Hello World!" );
    }
}
```

With the `using System;` directive, you can omit the `System` namespace when calling `Console.WriteLine`.

To compile this example, execute the following command from a Windows command prompt:

```
csc.exe /r:mscorlib.dll /target:exe hello_world.cs
```

Let's take a look at exactly what this command line does:

- `csc.exe` is the Microsoft C# compiler.
- The `/r` option specifies the assembly dependencies this program has. Assemblies are similar in concept to DLLs in the native world. `mscorlib.dll` is where the `System.Console` object is defined. In reality, you don't need to reference the `mscorlib` assembly because the compiler will reference it automatically, unless you use the `/nostdlib` option.
- The `/target:exe` option tells the compiler that you're building a console application, which is the default if not specified. Your other options here are `/target:winexe` for building a Windows GUI application, `/target:library` for building a DLL assembly with the `.dll` extension, and `/target:module` for generating a DLL with the `.netmodule` extension. `/target:module` generated modules don't contain an assembly manifest, so you must include it later into an assembly using the assembly linker `al.exe`. This provides a way to create multiple-file assemblies.
- `hello_world.cs` is the C# program you're compiling. If multiple C# files exist in the project, you could just list them all at the end of the command line.

Once you execute this command line, it produces `hello_world.exe`, and you can execute it from the command line and see the expected results. If you want, you can rebuild the code with the `/debug` option. Then you may step through the execution inside of a debugger. To give an example of C# platform independence, if you happen to have a Linux OS running and you have the Mono VES installed on it, you can copy this `hello_world.exe` directly over in its binary form and it will run as expected, assuming everything is set up correctly on the Linux box.

Overview of Features Added in C# 2.0

Since its initial release in late 2000, the C# language has evolved considerably. This evolution has likely been accelerated thanks to the wide adoption of C#. With the release of Visual Studio 2005 and the .NET Framework 2.0, the C# compiler supported the C# 2.0 enhancements to the language. This was great news, since C# 2.0 included some handy features that provided a more natural programming experience as well as greater efficiency. This section provides an overview of what those features are and what chapters of the book contain more detailed information.

Arguably, the meatiest addition to C# 2.0 was support for generics. The syntax is similar to C++ templates, but the main difference is that constructed types created from .NET generics are dynamic in nature—that is, they are bound and constructed at run time. This differs from C++ concrete types created from templates, which are static in the sense that they are bound and created at compile time.[5] Generics are most useful when used with container types such as vectors, lists, and hash tables, where they provide the greatest efficiency gains. Generics can treat the types that they contain specifically by their type, rather than by using the base type of all objects, `System.Object`. I cover generics in Chapter 11, and I cover collections in Chapter 9.

C# 2.0 added support for anonymous methods. An anonymous method is sometimes referred to as a *lambda function*, which comes from functional programming disciplines. C# anonymous methods are extremely useful with delegates and events. Delegates and events are constructs used to register callback methods that are called when triggered. Normally, you wire them up to a defined method somewhere. But with anonymous methods, you can define the delegate's or event's code inline, at the point where the delegate or event is set up. This is handy if your delegate merely needs to perform some small amount of work for which an entire method definition would be overkill. What's even better is that the anonymous method body has access to all variables that are in scope at the point it is defined.[6] I cover anonymous methods in Chapter 10. Lambda expressions, which are new to C# 3.0, supersede anonymous methods and make for more readable code.

C# 2.0 added support for iterators. Anyone familiar with the C++ Standard Template Library (STL) knows about iterators and their usefulness. In C#, you typically use the `foreach` statement to iterate over an object that behaves as a collection. That collection object must implement the `IEnumerable` interface, which includes the `GetEnumerator` method. Implementing the `GetEnumerator` method on container types is typically very tedious. However, when using C# iterators, implementing the `GetEnumerator` method is a snap. You can find more information regarding iterators in Chapter 9.

Finally, C# 2.0 added support for partial types. Prior to C# 2.0, you had to define each C# class entirely in one file (also called a *compilation unit*). This requirement was relaxed with the support for partial types. This was great news for those who rely upon code generators to provide skeleton code. For example, you can use the Visual Studio wizards to generate such useful things as `System.Data.DataSet` derived types for accessing data in a database. Prior to C# 2.0, it was problematic if you needed to make modifications to the generated code. You either had to derive from or contain the generated type in a new type while specializing its implementation, or you had to edit the generated code. Editing the generated code was risky because you normally lost those changes when the wizard was forced to regenerate the type for some reason. Partial types solve this problem, because now you can augment the generated code in a separate file so that your changes aren't lost when the wizard regenerates the code. For a great example of how partial types are used, look at the code automatically generated when you create a Windows Forms application using Visual Studio. You can find more information regarding partial types in Chapter 4.

---

5. Using C++/CLI, standardized in Ecma-372 and first made available with Visual Studio 2005, you can use generics and templates together.

6. This is referred to as either a *closure* or a *variable capture*.

Overview of What's New in C# 3.0

C# 3.0 includes some great new features. Most of the new features are stepping stones designed to support Language Integrated Query (LINQ). Nevertheless, all of them are extremely useful when used individually outside of the context of LINQ. Many of them allow programmers to employ functional programming techniques more easily.

C# now supports implicitly typed local variables by making use of a new keyword `var`. It's important to note that these variables are not typeless; rather, their type is inferred at compile time. You can read more about them in Chapter 3.

Have you ever wanted to create a simple type to hold some related data but been annoyed at having to create an entire new class? In many cases, the new support for anonymous types helps relieve you of this burden. Using anonymous types, you can define and instantiate a type all in one compound statement. I cover anonymous types in Chapter 4.

Auto-implemented properties are another helpful new feature to save us some typing and reduce the potential to introduce bugs. How many times have you simply declared a class to hold a few pieces of data and been annoyed with the amount of typing required to create property accessors for that data? After all, doing so follows good encapsulation practices. Thankfully, auto-implemented properties greatly reduce the amount of typing necessary to define properties on types. You can read more about them in Chapter 4.

While we're on the subject of conveniences, C# 3.0 also introduces two new features that help when instantiating and initializing object instances. Using object and collection initializers, you can instantiate and initialize either an object or a collection in one compound statement. I cover object initializers in Chapter 4 and collection initializers in Chapter 9.

C# 2.0 introduced partial class definitions to facilitate using code generators. C# 3.0 adds to that by introducing partial methods. Using partial methods, a code generator can declare a method signature, and the consumer of that generated code, the one that creates the rest of the partial class definition, can choose to implement it or not. You can read more about partial methods in Chapter 4.

Extension methods are one of the most exciting new features. Taken from the surface view, they are merely static methods that can be called as if they were instance methods. They do not get any special access into the instance they are operating on, so in that respect, they are just like static methods. However, the syntax they foster allows us to program in a more functional manner, usually resulting in clearer and more readable code. I devote the entire Chapter 14 to extension methods and what you can do with them.

Probably more compelling than extension methods is support for lambda expressions. Lambda expressions supersede support for anonymous methods. That is, if lambda expressions had existed in C# 2.0, there would have been no need for anonymous methods at all. However, lambda expressions offer much more than anonymous methods as they can be converted into both delegates and expression trees. Lambda expressions are covered in Chapter 15.

The granddaddy of all new C# 3.0 features has to be LINQ, which builds upon all of the new features, especially extension methods, lambda expressions, and anonymous types. It also adds some new language keywords to allow us to code intuitive query statements, thus seamlessly bridging the gap between the object-oriented world and the data world. You can use LINQ to access data from multiple sources. Visual Studio provides the capability to use LINQ on native object collections, SQL data stores, and XML. Support for many other data sources is coming soon from both Microsoft and third parties. For example, you'll be able to use LINQ to connect to Windows Management Instrumentation (WMI), the Document Object Model (DOM), and the Web. Additionally, there are implementations in the works to use LINQ against popular web sites such as Google and Flickr. Chapter 16 is devoted to LINQ.

Summary

In this chapter, I've touched upon the high-level characteristics of programs written in C#. That is, all code is compiled into IL rather than the native instructions for a specific platform. Additionally, the CLR implements a GC to manage raw memory allocation and deallocation, freeing you from having to worry about one of the most common errors in software development: improper memory management. However, as with most engineering trade-offs, there are other aspects (read: complications) of memory and resource management that the GC can introduce in certain situations.

Using the venerable "Hello World!" example, I was able to quickly show the usefulness of namespaces as well as the fact that C# is devoid of any kind of inclusion syntax as available in C++. Instead, all other external types are brought into the compilation unit via metadata, which is a rich description format of the types contained within an assembly. Therefore, the metadata and the compiled types are always contained in one neat package.

Generics open up such a huge area of development that you'll probably still be learning handy tricks of applying them over the next several years. Some of those tricks can be borrowed from the C++ template world, but not all of them, since the two concepts are fundamentally different. Iterators and anonymous methods offer a concise way of expressing common idioms such as enumeration and callback methods, while support for partial type declarations within C# makes it easier to work with tool-generated code.

C# 3.0 offers many new and exciting features that allow one to employ functional programming techniques very easily with little overhead. Some of the new features add convenience to programming in C#. LINQ provides a seamless mechanism to bridge to the data storage world from the object-oriented world.

In the next chapter, I'll briefly cover more details regarding the JIT compilation process. Additionally, I'll dig into assemblies and their contained metadata a bit more. Assemblies are the basic building blocks of C# applications, analogous to DLLs in the native Windows world.