Introducing the .NET Framework and Visual Studio

Business application programming has evolved from a two-tier, tightly coupled model into a multitiered, loosely coupled model, often involving data transfer over the Internet or a corporate intranet. In an effort to allow programmers to be more productive and deal with the complexities of this type of model, Microsoft developed the .NET Framework. To effectively program in C#, you need to understand the underlying framework upon which it is built.

After reading this chapter, you should be familiar with the following:

- the .NET Framework
- the features of the Common Language Runtime (CLR)
- how the just-in-time (JIT) compiler works
- the .NET Framework base class library
- namespaces and assemblies
- the features of the Visual Studio integrated development environment

Introducing the .NET Framework

The .NET Framework is a collection of fundamental classes designed to provide the common services needed to run applications. Let’s look at the goals of the .NET Framework and then review its components.

Goals of the .NET Framework

Microsoft designed the .NET Framework with certain goals in mind. The following sections examine these goals and how the .NET Framework achieves them.

Support of Industry Standards

Microsoft wanted the .NET Framework to be based on industry standards and practices. As a result, the framework relies heavily on industry standards such as the Extensible Markup Language (XML), HTML 5, and OData. Microsoft
has also submitted a Common Language Infrastructure (CLI) Working Document to the European Computer Manufacturers Association (ECMA), which oversees many of the common standards in the computer industry.

The CLI is a set of specifications needed to create compilers that conform to the .NET Framework. Third-party vendors can use these specifications to create .NET-compliant language compilers; for example, Interactive Software Engineering (ISE) has created a .NET compiler for Eiffel. Third-party vendors can also create a CLR that will allow .NET-compliant languages to run on different platforms. One example, Mono is an open source, cross platform implementation of the CLR that gives C# applications the ability to run on the Linux platform.

**Extensibility**

To create a highly productive environment in which to program, Microsoft realized the .NET Framework had to be extensible. As a result, Microsoft exposed the framework class hierarchy to developers. Through inheritance and interfaces, you can easily access and extend the functionality of these classes. For example, you could create a button control class that not only inherits its base functionality from the button class exposed by the .NET Framework, but also extends the base functionality in the unique way required by your application.

Microsoft has also made it much easier to work with the underlying operating system. By repackaging and implementing the Windows operating system application programming interface (API) functions in a class-based hierarchy, Microsoft has made it more intuitive and easier for OOP programmers to work with the functionality exposed by the underlying operating system.

**Unified Programming Models**

Another important goal Microsoft incorporated into the .NET Framework was cross-language independence and integration. To achieve this goal, all languages that support the Common Language Specification (CLS) compile into the same intermediate language, support the same set of basic data types, and expose the same set of code-accessibility methods. As a result, not only can classes developed in the different CLS-compliant languages communicate seamlessly with one another, but you can also implement OOP constructs across languages. For example, you could develop a class written in C# that inherits from a class written using Visual Basic (VB). Microsoft has developed several languages that support the .NET Framework. Along with C#, the languages are VB.NET, managed C++, JScript, and F#. In addition to these languages, many third-party vendors have developed versions of other popular languages designed to run under the .NET Framework, such as Pascal and Python.

**Easier Deployment**

Microsoft needed a way to simplify application deployment. Before the development of the .NET Framework, when components were deployed, component information had to be recorded in the system registry. Many of these components, especially system components, were used by several different client applications. When a client application made a call to the component, the registry was searched to determine the metadata needed to work with the component. If a newer version of the component was deployed, it replaced the registry information of the old component. Often, the new components were incompatible with the old version and caused existing clients to fail. You have probably experienced this problem after installing a service pack that ended up causing more problems than it fixed!

The .NET Framework combats this problem by storing the metadata for working with the component in a file called a manifest, which is packaged in the assembly containing the component code. An assembly is a package containing the code, resources, and metadata needed to run an application. By default, an assembly is marked as private and placed in the same directory as the client assembly. This ensures that the component assembly is not inadvertently replaced or modified and also allows for a simpler deployment because there is no need to work with the registry. If a component needs to be shared, its assembly is deployed to a special directory referred to as the Global Assembly Cache (GAC). The manifest of the assembly contains versioning information, so newer versions of the