.NET FRAMEWORK

By Jeffery Richter

Summary

The Microsoft .NET Framework is a new platform for building integrated, service-oriented applications to meet the needs of today's Internet businesses; apps that gather information from, and interact with, a wide variety of sources, regardless of the platforms or languages in use. This article illustrates how the .NET Framework and the common language runtime can deliver write-once, compile-once, run-anywhere application development. Microsoft ***Intermediate Language and JIT compiler***, which make this reuse possible, are described as well as managed components, assemblies, and the ***Common Type System (CTS)***.

I'm finding that there are many things I'd like to do on the Internet that are still not possible today. For example, not only do I want to find restaurants in my area that serve a specific type of cuisine, I'd like to know if a particular restaurant has seating available at 7 pm tonight. If I had my own business, I might like to know which vendor has a particular item in stock. If several vendors can supply the item, I would want to know which vendor has the best price and which one can deliver the item faster. Services like these don't yet exist because there are no standards in place for integrating all this proprietary information. After all, vendors have their own way of describing items they sell. Additionally, it is difficult to develop the code necessary to integrate such services.

It should be no surprise that Microsoft recognizes the need to integrate services like these. To that end, Microsoft is embracing the XML standard for describing data, and is providing a new development platform infrastructure that I believe will help developers create and deploy distributed applications on Internet time.
This platform is called the Microsoft® .NET Framework (.NET).

.NET will allow developers to take better advantage of technologies than any earlier Microsoft platform. Specifically, .NET will really deliver on code reuse, code specialization, resource management, multilanguage development, security, deployment, and administration. While designing this new platform, Microsoft also improved some features of the current Windows® platform. I will address many of these improvements in this article.

Common Language Runtime and Class Library

.NET represents a whole new way of developing software. You'll notice that I didn't say developing software for Windows. That's because .NET is not inextricably tied to the Windows operating system. In this article, I'll discuss the .NET architecture and describe how this new platform offers the type of efficiencies I mentioned earlier.

At the heart of the .NET platform is a common language runtime engine and a base framework. All programmers are familiar with these concepts. I'm sure many of you have at least dabbled with the C runtime library, the standard template library, the MFC library, the Active Template Library, the Visual Basic® runtime library, or the Java virtual machine. In fact, the Windows operating system itself can be thought of as a runtime engine and library. Runtime engines and libraries offer services to applications, and programmers love them because they save time and facilitate code reuse.

The .NET base framework will allow developers to access the features of the common language runtime and also will offer many high-level services so that developers don't have to code the same services repeatedly. But more importantly, the .NET common language runtime engine sitting under this library will provide the technologies to support rapid software development. The following lists a small sampling of features to be provided by the .NET common language runtime engine.

Consistent Programming Model

All application services are offered via a common object-oriented programming model, unlike today where some OS facilities are accessed via DLL functions and other facilities are accessed via COM objects. Simplified programming model .NET seeks to greatly simplify the plumbing and arcane constructs required by Win32® and COM. Specifically, developers no longer need to gain an understanding of the registry, GUIDs, IUnknown, AddRef, Release, HRESULTS, and so on. It is important to note that .NET doesn't just abstract these concepts away from the developer; in the new .NET platform, these concepts simply do not exist at all.

Run Once, Run Always

All developers are familiar with DLL Hell. Since installing components for a new application can overwrite components of an old application, the old app can exhibit strange behavior or stop functioning altogether. The .NET architecture now separates application components so that an app always loads the components with which it was built and tested. If the application runs after installation, then the application should always run. This marks the end of DLL Hell.

Execute On Many Platforms

Today, there are many different flavors of Windows: Windows 95, Windows 98, Windows 98 SE, Windows Me, Windows NT® 4.0, Windows 2000 (with various service packs), Windows CE, and soon a 64-bit version of Windows 2000. Most of these systems run on x86 CPUs, but Windows CE and 64-bit Windows run on non-x86 CPUs. Once written and built, a managed .NET application (that consists entirely of managed code, as I'll explain shortly) can execute on any platform that supports the .NET common language runtime. It is even possible that a version of the common language runtime could be built for platforms other than Windows in the future. Users will immediately appreciate the value of this broad execution model when they need to support multiple computing hardware configurations or operating systems.

Language Integration

COM allows different programming languages to interoperate with one another. .NET allows languages to be integrated with one another. For example, it is possible to create a class in C++ that derives from a class implemented in Visual Basic. .NET can enable this because it defines and provides a type system common to all .NET languages. The Microsoft Common Language describes what compiler implementors must do in order for their languages to integrate well with other languages. Microsoft is providing several compilers that produce code targeting the .NET common language runtime: C++ with managed extensions, C# (pronounced "C sharp"), Visual Basic (which now subsumes VBScript and Visual Basic for Applications), and JScript®. In addition, companies other than Microsoft are producing compilers for languages that also target the .NET common language runtime.

Code Reuse

Using the mechanisms I just described, you can create your own classes that offer services to third-party applications. This, of course, makes it extremely simple to reuse code and broadens the market for component vendors. Automatic resource management As you know, programming requires great skill and discipline. This is especially true when it comes to managing resources such as files, memory, screen space, network connections, database resources, and so on.

One of the most common bugs occurs when an application neglects to free one of these resources, causing that application or others to perform improperly at some unpredictable time. The .NET common language runtime automatically tracks resource usage, guaranteeing that your application never leaks resources. In fact, there is no way to explicitly free a resource. In a future article, I'll explain exactly how this works.

Type Safety

The .NET common language runtime can verify that all your code is type safe. Type safety ensures that allocated objects are always accessed in compatible ways. Hence, if a method input parameter is declared as accepting a 4-byte value, the common language runtime will detect and trap attempts to access the parameter as an 8-byte value. Similarly, if an object occupies 10 bytes in memory, the application can't coerce this into a form that will allow more than 10 bytes to be read.

Type safety also means that execution flow will only transfer to well-known locations (namely, method entry points). There is no way to construct an arbitrary reference to a memory location and cause code at that location to begin execution. Together, these eliminate many common programming errors and classic system attacks such as the exploitation of buffer overruns.

Rich Debugging Support

Because the .NET common language runtime is used for many languages, it is now much easier to implement portions of your application using the language that's best suited for it. The .NET common language runtime fully supports debugging applications that cross language boundaries. The runtime also provides built-in stack-walking facilities, making it much easier to locate bugs and errors.

Consistent Error-Handling

One of the most aggravating aspects of programming in Windows is the inconsistent ways errors are reported. Some functions return Win32 error codes, some return HRESULTS, and some raise exceptions. In .NET, all errors are reported via exceptions—period. Exceptions allow the developer to isolate the error-handling code from the code required to get the work done. This greatly simplifies writing, reading, and maintaining code. In addition, exceptions work across module and language boundaries as well.

Deployment

Today, Windows-based applications can be incredibly difficult to install and deploy. There are usually several files, registry settings, and shortcuts that need to be created. In addition, completely uninstalling an application is nearly impossible. With Windows 2000, Microsoft introduced a new installation engine that helps with all of these issues, but it is still possible that a company authoring a Microsoft Installer Package may fail to do everything correctly. .NET seeks to make these issues ancient history. .NET components are not referenced in the registry. In fact, installing most .NET-based applications will require no more than
copying the files to a directory, and uninstalling an application will be as easy as deleting those files.

Security

Traditional OS security provides isolation and access control based on user accounts. This has proven to be a useful model, but at its core it assumes that all code is equally trustworthy. This assumption was justified when all code was installed from physical media (such as CD-ROM) or trusted corporate servers. But with the increasing reliance on mobile code such as Web scripts, Internet application downloads, and e-mail attachments, there is a need for more granular control of application behavior.

The .NET Code Access Security model will deliver this control. Microsoft also recognizes that many apps need to enforce behaviors based on a concept of roles as opposed to individual accounts. Microsoft initially delivered support for this concept with Microsoft Transaction Server (MTS), and provided further enhancements with COM+ 1.0. In .NET, Microsoft supports the deployment of

application-defined roles and access control based on these roles. These mechanisms are extended in ways that are appropriate for the Internet and heterogeneous environments.

As you might imagine, addressing all these issues requires an architecture consisting of many pieces. To create a clearer picture of the .NET Framework, I'll walk through the process of developing and deploying an application. The first step is to determine what type of application I intend to build and what languages I will use to develop it. For this discussion, let's assume that I already know the application type, that the application is designed, the specs are written, and I'm ready to start development.

The Right Language for the Job

My application consists of several different subcomponents. To make development easier, I realize that certain programming languages are best tailored to handle specific tasks. In addition, some of the developers on my team prefer working in C++, while others prefer Pascal. In .NET, the language you select for any given task is totally up to you; it is simply a matter of personal choice.

As far as the .NET common language runtime is concerned, all languages are equal. Now it's true that some languages offer certain features that others don't, but these are language features, not common language runtime features. For example, the .NET common language runtime supports the creation and manipulation of threads. Since this is a common language runtime feature, any language that targets the common language runtime will be able to create and manipulate threads.

On the other hand, Visual Basic is specifically designed to prevent programmers from shooting themselves in the foot. One way that Visual Basic does this is by treating all variables, function names, and so on as case-insensitive symbols. This feature of Visual Basic is purely a language issue and has no bearing on the Visual Basic runtime at all.

Microsoft is planning to create four language compilers that target the common language runtime: C++ with managed extensions, C#, Visual Basic, and JScript. By default, the Microsoft Visual C++® compiler builds an executable or DLL that does not target the common language runtime; specifying a new command-line switch will make the Visual C++ compiler target it. The term "managed code" describes code that requires the common language runtime. Unmanaged code describes code that does not require the common language runtime engine.

Of the four Microsoft compilers mentioned, Visual C++ is the only one capable of producing unmanaged code. Since the other compilers (C#, Visual Basic, and JScript) can only produce managed code, the code written in these languages absolutely requires the common language runtime engine in order to execute. In addition to Microsoft, several companies are producing compilers that generate managed code. I am aware of compilers for APL, CAML, Cobol, Haskell, Mercury, ML, Oberon, Oz, Pascal, Perl, Python, Scheme, and Smalltalk. In fact, Rational is planning to create a Java-language compiler that targets the .NET common language runtime.

Assemblies (Managed Components)

After I create my source code files, I run them through the respective compilers, and the result is an EXE or DLL. These EXE or DLL files are very similar to the ***Portable Executable (PE)*** files that you've become familiar with over the years—in fact, they are valid PE files—with just a few differences.

One difference is that the code in managed PE files generated by the language compilers is not x86 machine code or machine code targeted to any specific CPU platform. Instead, the code generated by

the compiler is a ***Microsoft intermediate language (MSIL)***, which will be discussed in more detail later.

Another difference is that the file contains metadata emitted by the compiler that is used by the common language runtime to locate and load class types in the file, lay out object instances in memory, resolve method invocations and field references, translate MSIL to native code, enforce security, and perform a whole slew of additional features.

Yet another difference is that the components you are producing are not just EXEs or DLLs. Rather, the unit of reuse and deployment in .NET is an assembly. Depending on the choices you make in your compiler or tool, you might be producing a single-file assembly or you might be producing a managed code module that is intended to be deployed as a part of a multi-file assembly. Think of an assembly as a logical EXE or DLL. In fact, because in the single-file assembly case the assembly carries an .exe or .dll extension, you might wonder why a new concept was introduced.

The point, very precisely, is that by decoupling the logical and physical notions of a reusable component, you get to partition your code into multiple files as an implementation choice (for example, to get benefits of incremental and on-demand download), while still maintaining the collection as a single unit of versioning and deployment.

From an assembly consumer's perspective (the external view), an assembly is a named and versioned collection of exported types and resources. From an assembly developer's perspective (the internal view), an assembly is a collection of one or more files—from a single PE file to a collection of PE files, resource files, HTML pages, GIFs, and so on—that implement types and resources.

The components of an assembly are described in a manifest. A manifest is a block of data that enumerates the assembly's files, and controls what types and resources are exposed outside of the assembly. For example, some types may be reserved for use only within the assembly, while others may be exported to consumers of the assembly. The manifest also governs how references to these types and resources are mapped onto the files that contain their declarations and implementations, and enumerates other assemblies on which this one is dependent. The existence of a manifest provides a level of indirection between consumers of the assembly and the implementation details of the assembly and makes assemblies self-describing.

Every module loaded at runtime is loaded in the context of an assembly. Every type that is loaded is scoped in an assembly scope—that is, part of the identity of a type is its assembly identity. A type Foo loaded in the scope of one assembly is not the same type Foo loaded in the scope of another assembly even if a hash of their declarations and implementations are exactly the same.

As a developer, you will make explicit choices at development time. If you import a type from another assembly, your own file's metadata will contain references to that assembly—the compiler will put them there. If you import a type from an individual module within a multi-file assembly, your own file will contain an explicit dependency on that module—again, the compiler will put it there. In the latter case, .NET will honor and enforce assembly boundaries at runtime, making sure that you and that module are, in fact, built as part of the same assembly.

The Microsoft languages and compilers make it easy to tackle these decisions by providing reasonable defaults for the typical case and by providing directives and tools—such as the Assembly Linker utility, AL.exe, that aid in building multi-file assemblies.

Assemblies are the basis for the deployment and versioning features of .NET. Assemblies allow developers and administrators to express strict version dependencies between pieces of an application and, because they are self-describing, help enable the notion of zero-impact install. Assemblies also play a role in the .NET security system in that the assembly is the unit at which permissions are requested and granted. To properly enforce the versioning, deployment, and security features in .NET, the assembly acts as the resolution scope that is used to resolve references to types. .NET honors the rules expressed in the assembly manifest for resolving types and resources to their implementation files within an assembly and for binding to dependent assemblies.

System Services

Included with .NET will be a base framework assembly that contains several class definitions, where each class exposes some feature of the underlying platform. Since Microsoft defines literally hundreds of classes in the base library, the library is divided into namespaces that group related classes together. For example, the System namespace (which you should become most familiar with) contains the base class, Object, which all other classes ultimately derive from (I'll mention more about this later). In addition, the System namespace contains classes for exception handling, garbage collection, console I/O, as well as a bunch of utility classes that convert safely between data types, format data types, generate random numbers, and perform various math functions.

Figure 1 Namespaces in the Base Framework

Namespace	Description	Example Classes
System	Contains all the basic types used by every application	Object, Buffer, Byte, Char, Array, Int32, Exception, GC, String
System.Collections	Contains types for managing collections of Objects	ArrayList, BitArray, Dictionary, Hashtable, Queue, SortedList, Stack
System.Data	Contains basic database management types	DataBinding, DataRelation, DataRow, DataSet, DataTable, DataSource
System.Globalization	Contains types for National Language Support (NLS), String compares, and Calendars	Calendar, CultureInfo, JulianCalendar, NumberFormatInfo, NumberStyles, RegionInfo
System.IO	Contains types that allow synchronous and asynchronous reading and writing to data streams	ByteStream, File, FileStream, MemoryStream,Path, StreamReader, StreamWriter
System.Net	Contains types that allow for network communications	WebRequest, WebResponse, TcpClient, TcpListener, UdpClient, Sockets
System.Reflection	Contains types that allow the inspection of metadata	Assembly, ConstructorInfo, FieldInfo, MemberInfo, MethodInfo, Module, ParameterInfo
System.Runtime.Remoting	Contains types that allow for managed remote objects	ChannelServices, RemotingServices, IMessage, ImessageSink
System.Security	Contains types that enable security features	Permissions, Policy, Principal, Util, Cryptography
System.Web.UI.WebControls	Contains types that enable rich user interface controls for Web-based applications	AdRotator, BorderStyle, DataGrid, HyperLink, ListBox, Panel, RadioButton, Table
System.WinForms	Contains types that enable rich user interface controls for desktop	Button, CheckBox, DataGrid, FileDialog, Form, ListBox,

	applications	MainMenu, MonthCalendar, NewFontDialog, RichEdit, ToolBarTreeView

Figure 1 shows a small sampling of some of the other namespaces included in the base framework. Most of the namespaces listed in Figure 1 show "lower-level" classes. However, the base framework also includes namespaces for building rich user interface applications. For example, the System.Web.UI.WebControls and System.WinForms namespaces include classes that allow for very rich user interfaces in both Web-based and non-Web-based applications. .NET and the base framework allow developers to build the following application types: Web Services, Win32-based GUI applications, Win32 CUI-based applications, services (controlled by the Service Control Manager), compilers and tools, personal-tier applications, and components.

To access any of the platform's features, you need to work with these namespaces and their defined classes. If you want to customize the behavior of any class, you can simply derive your own class from the desired base library class.

.NET can present a consistent programming paradigm to software developers because of the object-oriented nature of the platform. Developers can create their own namespaces containing their own classes. These merge seamlessly into the programming paradigm, greatly simplifying software development when compared to classic Windows-based programming paradigms.

Intermediate Language, the Common Language Runtime, and JIT Compilers

As stated earlier, compiling your source code causes the compiler to emit MSIL. MSIL is a CPU-independent intermediate language created by Microsoft after consultation with several external commercial and academic language/compiler writers. MSIL is much higher level than most CPU machine languages. It understands object types and has instructions that create and initialize objects, call virtual methods on objects, and manipulate array elements directly. It even has instructions that raise and catch exceptions for error handling.

Like any other machine language, MSIL can be written in assembly language. Microsoft provides an MSIL assembler, ILAsm.exe, as well as an MSIL disassembler, ILDasm.exe. The important thing to note about MSIL is that it is not tied to any specific CPU platform. This means that a PE file containing MSIL can run on any CPU platform as long as the operating system running on that CPU platform hosts the .NET common language runtime engine. Initially, Microsoft plans to offer the .NET common language runtime engine on the various flavors of Windows 95, Windows 98, Windows NT 4.0, Windows 2000 (both 32-bit and 64-bit), and Windows CE.

Another benefit of MSIL is that it provides a hardware abstraction layer. While the initial release of the common language runtime is planned to run only on 32-bit Windows platforms, developing an application using managed MSIL allows the application to be more independent of the underlying operating system. For example, MSIL will help make code portable to 64-bit versions of Windows when the common language runtime is available on that platform.

Finally, MSIL is a very important aspect of the .NET security story since verification is capable of examining the code's intention regardless of what high-level language was used to generate the code. Today, when you build a managed executable, the module imports a function (called _CorExeMain) from the .NET common language runtime (MSCorEE.dll). When the user invokes the executable, the operating system loader loads MSCorEE.dll and jumps to an unmanaged entry point inside the executable module.

The code in the executable module is a small stub (6 bytes on x86) that simply jumps to the _CorExeMain function contained inside MSCorEE.dll. Once the common language runtime engine has initialized, the engine locates the executable module's managed entry point and then starts executing the executable module's managed MSIL code. In the future, operating system loaders will be modified to look specifically for modules containing managed MSIL code. When the OS loader detects a module with managed code, the unmanaged stub entry point will no longer be necessary and the common language runtime will simply be able to jump to the executable's managed entry point.

Of course, the MSIL instructions cannot be executed directly by today's host CPUs. So, the common language runtime engine must first compile the managed MSIL instructions into native CPU instructions. This process is shown in Figure 2. (Note that other parts of the figure are described in Part 2 of this article.)

Figure 2 Steps in Compiling Source Code

The need to compile MSIL into native code raises several questions. For example, should the common language runtime convert all of the file's MSIL code to CPU instructions at load time? The advantage of doing this is that the program runs very fast. However, the disadvantage is that compiling the MSIL requires time, and this would significantly prolong the application's initialization

time. In addition, very rarely does a user cause an application to execute all of its code. If the common language runtime compiles all of the MSIL to CPU instructions, it is likely that a lot of time and memory would be wasted.

The consensus is that it would be much more efficient to have the common language runtime compile the MSIL instructions as functions are being called. When the common language runtime loads a class type, it connects stub code to each method. When a method is called, the stub code directs program execution to the component of the common language runtime engine that is responsible for compiling the method's MSIL into native code. Since the MSIL is being compiled just-in-time (JIT), this component of the runtime is frequently referred to as a JIT compiler or JITter. Once the JIT compiler has compiled the MSIL, the method's stub is replaced with the address of the compiled code. Whenever this method is called in the future, the native code will just execute and the JIT compiler will not have to be involved in the process. As you can imagine, this boosts performance considerably.

The common language runtime is slated to ship with two JIT compilers, the normal compiler and an economy compiler. The normal compiler examines a method's MSIL and optimizes the resulting native code just like the back end of a normal, unmanaged C/C++ compiler. The economy JIT compiler is typically used on machines where the cost of using memory and CPU cycles is high (such as many Windows CE-powered devices). The economy compiler simply replaces each MSIL instruction with its native code counterpart. As you can imagine, the economy compiler compiles code much faster than the normal compiler; however, the native code produced by the economy compiler is significantly less efficient. Even so, the economy compiler will still produce code that executes much faster than interpreted code.

When the common language runtime is ported to a new CPU platform, Microsoft first creates the economy JITter for that platform. Since the economy JITter is relatively easy to implement, this allows .NET applications to run on the new CPU platform in a very short period of time. Once the economy JITter is complete, Microsoft then focuses its efforts on the normal JITter to improve application performance.

When Microsoft ships the .NET common language runtime, the normal JITter will be the default on most machines. However, on machines with small footprints, like Windows CE-powered devices, the economy JITter will be the default because it requires less memory to run.

In addition, the economy JITter supports code pitching. Code pitching is the ability for the common language runtime to discard a method's native code block, freeing up memory used by methods that haven't been executed in a while. Of course, when the common language runtime pitches a block of code, it replaces the method with a stub so that the JITter can regenerate the native code the next time the method is called.

For those of you who are used to developing in low-level languages like C or C++, you're probably thinking about the performance ramifications of all this. After all, unmanaged code is compiled for a specific CPU platform, and when invoked the code can simply execute. In this managed environment, compiling the code is accomplished in two phases. First, the compiler passes over the source code, doing as much work as possible in producing MSIL. But then, in order to actually execute the code, the MSIL itself must be compiled into native CPU instructions at runtime, requiring that more memory and additional CPU time be allocated to do the work.

Believe me, since I approached the runtime from a C/C++ background myself, I was quite skeptical and concerned about this additional overhead. The truth is, managed code does not execute as fast

and does have more overhead than unmanaged code. However, Microsoft has done a lot of performance work to address these issues. In addition, I've spoken to many developers at Microsoft who truly believe that in the future managed code will actually offer better performance than unmanaged code. Here's why: when the JITter compiles the MSIL code at runtime, it knows more about the execution environment than the compiler knows. For example, the JITter can detect that the host CPU is a Pentium III and generate CPU instructions that take advantage of any performance enhancements that Intel has made to the Pentium III over the Pentium II or Pentium.

In addition, the JITter may generate code that uses CPU registers that a compiler would normally avoid using. Or, a JITter may be able to detect that a variable always contains a specific value and can generate small, fast code that works solely because the JITter can make a runtime assumption. Furthermore, memory allocations are significantly faster than allocating memory via the Win32 HeapAlloc function. I will address this issue more fully in a future article.

Microsoft plans to offer a tool, tentatively called PreJit.exe, that can compile an entire assembly to native code and save the result on disk. When the assembly is loaded the next time, this saved version is loaded and the application starts up faster as a result. Because this tool takes advantage of information about other assemblies that have already been loaded when PreJit is run, it is best used in a warm-up mode. You turn PreJit on, and as assemblies are loaded, they are compiled and saved. After your application has run for a sufficient length of time, you turn off PreJit, and from then on the application will start up more quickly. Microsoft uses a variation on this same technique to make the key assemblies (such as the base framework) that will ship with .NET start faster.

The Common Type System

By now you know that developing an application for the .NET Framework requires the base library classes to access platform features and services. You also know how to compile source code and how to access the mechanisms required to get the code to execute. In this section, I'll discuss how to extend the base framework by defining your own class objects.

The formal specification of the type system implemented by the common language runtime is called the ***Common Type System (CTS***). The CTS specifies how object classes (called types) are defined. For example, the CTS allows a class type to contain zero or more members. The following is the list of possible members: Field A data variable that is part of the object's state. Fields are identified by their name and type.

Method - A function that performs an operation on the object, usually changing the object's state. Methods have a name, signature, and modifiers. The signature specifies the calling convention, number of parameters (and their sequence), the types of the parameters, and the type of value returned by the method. The modifiers can include custom attributes, whether the method is public, private, static, and so on.

Property - To the caller, this member looks like a field. But to the class implementor, this member looks like a method. Properties allow an implementor to calculate a value only when necessary and allow a class user to have simplified syntax. Properties also allow you to create read-only or write-only "fields."
Event - Events provide a notification mechanism between an object and other interested objects. For example, a button could offer an event that notifies other objects when the button is clicked.

The CTS also specifies the rules for type visibility and for access to the members of a type. Types are either visible outside of the assembly, to clients of the assembly, or they are visible only to code within

the same assembly. Marking a type as public enables it to be visible (exported) outside of the assembly. Thus, the CTS establishes the rules by which assemblies form a boundary of visibility for a type and its methods, and the common language runtime enforces the visibility rules. Regardless of whether a type is visible to a caller, the type gets to control whether the caller has access to its members.

Figure 3 Controlling Access

Access Type	Description
Public	The method is callable by any code in any assembly.
Private	The method is callable only by other methods in the same class type.
Family	The method is callable by derived types, regardless of whether they are within the same assembly.
Assembly	The method is callable by any code in the same assembly.
Family and Assembly	The method is callable by derived types only if the derived type is defined in the same assembly.
Family or Assembly	The method is callable by derived types in any assembly. The method is also callable by any types in the same assembly.

Figure 3 shows the valid options for controlling access to a method or field. In addition, the CTS defines the rules governing type inheritance, virtual functions, object lifetime, and so on. These rules have been designed to accommodate the semantics expressible in the languages you use today. In fact, you won't even need to learn the CTS rules per se, as the language you choose will expose its own language syntax and type rules in the same way you are familiar with, and will map this language-specific syntax into the syntax of the common language runtime when it emits the PE file.

When I first started working with the common language runtime, I realized that it is best to think of the language and the behavior of your code as two separate and distinct things. Using Visual C++, you can define your own classes with their own members. Of course, you could have used C# or Visual Basic to define the same class with the same members. Sure, the syntax you use for defining this class will differ depending on the language you choose, but the behavior of the class will be absolutely identical because the common language runtime CTS defines the behavior of the class object.

To help make this clear, let me give you an example. The CTS supports single inheritance only. So while the C++ language supports classes that inherit from multiple base classes, the CTS cannot accept and operate on any such class. To help the developer, the Visual C++ compiler will report an error if it detects that you're attempting to create managed code that includes a class inherited from multiple base classes.
Here's another CTS rule: all class types must (ultimately) inherit from a predefined class type called System.Object. As you can see, Object is the name of a type defined in the System namespace. This Object is the root of all other class types and therefore guarantees every class type has a minimum set of behaviors. Specifically, the System.Object type allows two objects to be compared for equality, allows you to uniquely identify an object via a hash code, allows you to query the object's true class type, allows you to perform a shallow (bitwise) copy of the object, and allows you to obtain a string representation of the object's current state.

The Common Language Specification

COM allows objects created in different languages to communicate with one another. In contrast, the .NET common language runtime integrates all languages and allows objects created in one language to be treated as equal citizens by code written in a completely different language. The common

language runtime makes this possible due to its standard set of types, self-describing type information (metadata), and common execution environment.

While language integration is a fantastic goal, the truth of the matter is that programming languages are very different from one another. For example, not all languages let you treat symbols with case-sensitivity, provide unsigned integers, offer operator overloading, unions, or even have methods that support a variable number of parameters, to name a few differences.

If you intend to create .NET types that are easily accessible from other programming languages, it is important to use features of your programming language that are guaranteed to be available in all other languages. To help you with this, Microsoft has defined a Common Language Specification (CLS) that informs compiler vendors of the minimum set of features that their compilers must support to target the common language runtime. Several vendors are already planning .NET-compatible versions of their compilers. Note that the CTS supports many more features than the subset defined by the common language specification, so if you don't care about interlanguage operability you can develop very rich class
types that are limited only by the language's feature set.

Metadata

Last month I mentioned that the compiler's job is to process your source code and produce the corresponding MSIL. In addition, the compiler is responsible for embedding metadata into every .NET-compliant EXE and DLL. In brief, .NET metadata is simply a collection of information persisted in binary form in a Portable Executable (PE) file that is a specification of the types declared and methods implemented in the file or assembly. Metadata is a superset of older technologies such as type libraries and IDL files. The important thing to note is that metadata is far more complete than its predecessors and is always associated with the file that contains the code. In fact, the metadata is always embedded in the same EXE and DLL as the code, making it impossible to separate the two.

All .NET-compliant compilers are required to emit full metadata information about every class type in the compiled source code module. This metadata contains a declaration for every type and a declaration, including names and types, for all of its members (methods, fields, properties, and events). For every implemented method, the metadata contains information that the loader uses to locate the method body.

Now I'll run through a few examples demonstrating the power of metadata. Say you're using Visual Studio® to edit your source code. In your code, you want to call a method on some class object. Because it can parse a type's metadata, Visual Studio can easily show you all the members of the type. In addition, if you call a method Visual Studio can show you the exact types required for the method's parameters and verify that your code uses the method's return type properly. It is also possible for the creator of a class type to associate help text and comments with a method or parameter in the metadata. As you can see, metadata helps you develop code faster and more accurately.

Tools like Visual Studio .NET retrieve a file's metadata information using a technology called reflection. There is a set of classes for enumerating the types in a file as well as the type's members. See the System.Reflection namespace for more information.

One of my favorite .NET tools is the MSIL disassembler, ILDasm.exe.

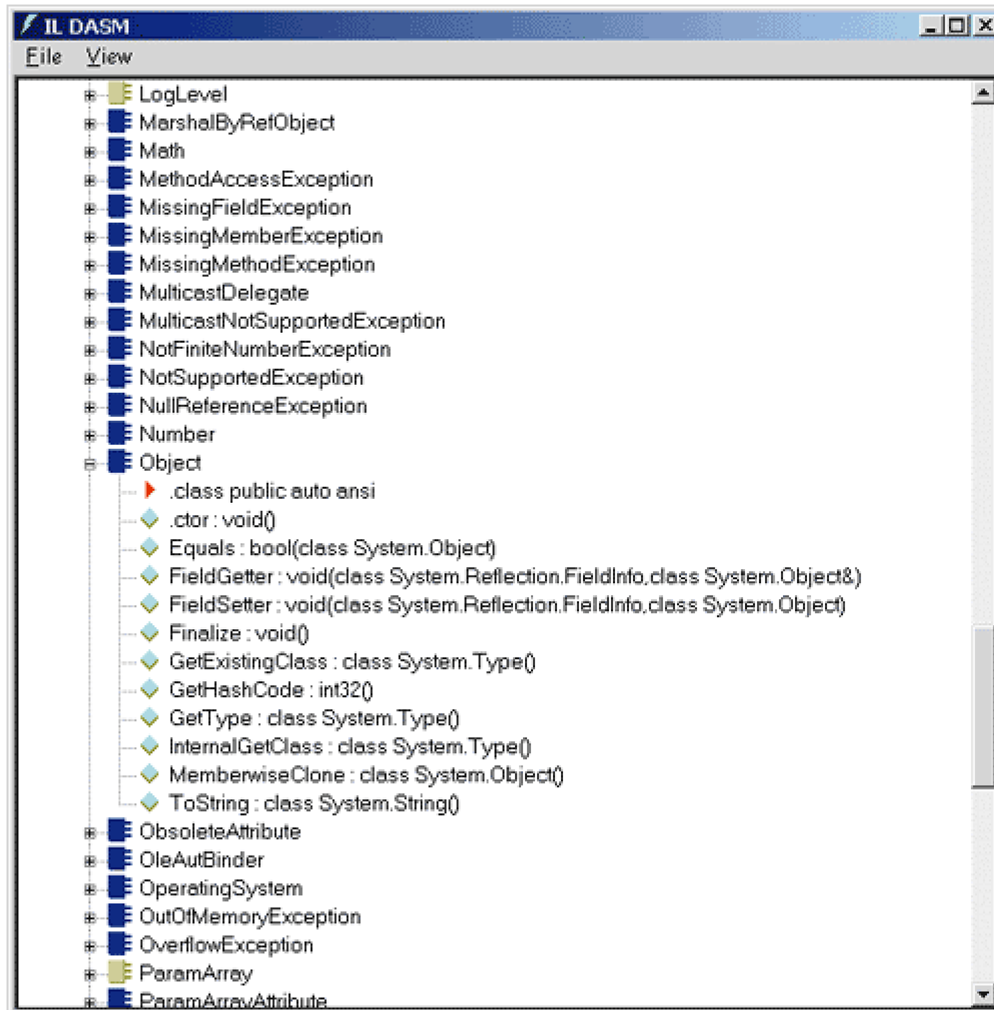


Figure 2 GetType

The figure above shows what happens when you run ILDasm on a portion of the base framework contained in MSCorLib.dll. In the window, ILDasm parses the file's metadata and displays the namespaces, types, interfaces, and members in a tree-like hierarchy. Double-clicking on any method causes another window to pop up; this shows the actual MSIL code that is generated by the compiler.

```
.method public instance class System.Type GetType() il managed
{
   // Code size     20 (0x14)
   .maxstack 2
   .locals (class System.Type V_0)
   IL_0000:   ldarg.0
   IL_0001:   call instance class System.Type
System.Object::GetExistingClass()
   IL_0006:   stloc.0
   IL_0007:   ldloc.0
   IL_0008:   ldnull
```

```
   IL_0009:   bne.un.s IL_00012
   IL_000b:   ldarg.0
   IL_000c:   call instance class System.Type
System.Object::InternalGetClass()
   IL_0011:   stloc.0
   IL_0012:   ldloc.0
   IL_0013:   ret
}  // end of method 'Object::GetType'
```

The figure above illustrates the MSIL code for the System.Object's GetType method. I am constantly using this tool in order to fully understand how .NET works. Note that the compiler emits types, fields, methods, properties, and event names into the metadata. However, the compiler only emits names given to values, local variables, and parameters into a module's program database (PDB) file for use when debugging. Since this debug information is not emitted into a module's metadata, it is much harder for someone to reverse-engineer MSIL code into source code.

Finally, metadata is the technology that enables remote method calls in the .NET Framework. To make a remote method call, the common language runtime must allocate a block of memory and lay the method's parameter data into it using a process called serialization. The common language runtime uses the metadata to determine how large a memory block to allocate and how to marshal the parameter data into the block. The block of data is then sent over a stream to the remote machine, where it is deserialized. Again, the metadata provides the template for the memory block. Then the method is called and the method's return value is remoted back to the originating machine.

Assemblies and Application Deployment

Over the years, the packaging and deployment of Windows®-based applications has gotten quite complex. Applications typically require several types of files (EXEs, DLLs, and data files) to run, contributing to the familiar problem in which a new installation causes an existing application to break because of some incompatibility between shared files. In addition, many applications—especially COM apps—require several registry settings to help the system locate these files. These application deployment and administration hassles are one of the pet peeves of customers that Microsoft is addressing very seriously.

Key to application deployment and versioning in .NET is the introduction of assemblies. As I described in Part 1 last month, assemblies are collections of behaviors designed to work together, built into single or multiple files. Depending on the application model you use (for example, Web server application or desktop application), the specifics of deployment may vary somewhat. But in general, deploying an application doesn't necessarily involve any more than copying a collection of one or more assemblies (one
containing your application entry point) to a specific directory that becomes your application directory. There are three reasons you don't need to register components.

First, every assembly is self-describing through metadata. Second, every reference to a type is scoped by an assembly reference (unlike COM, where all references to types are effectively machine-wide). Third, .NET uses a set of well-defined heuristics to locate referenced assemblies. Let's take a more detailed look at assembly identity and references and how they contribute to versioning. An assembly's identity attributes consist of:

A textual simple name

For persisted assemblies, the resolution heuristics used by the common language runtime expect that this name is the same as the name of the file in which the assembly manifest is deployed (excluding file extension). It is carried in metadata so that caching mechanisms may rename the file and still carry the original assembly identity.

A compatibility version number

Each assembly contains a version number in the form of incompat.compat.hotfix. A change in the incompat portion of the version number means that the new version is not compatible with prior versions. A change in the compat portion of the version number means that this is intended to be compatible with prior versions that share the same incompat version number. A change in the hotfix portion of the version number means that, yes, you've made a change, but it should be considered essentially the same bits as the prior version. The .NET common language runtime then uses this information in making binding decisions.

A cultural locale

Every assembly is either the default locale or is localized for a single locale. A development tool may be able to generate reasonable defaults for all of these elements, but most expose a way for the author to supply this information explicitly. The simple name (like My401kApp) is not guaranteed to be unique—there is nothing in the tools or in the common language runtime to prevent name collisions. There is also nothing to prevent a simple-named assembly from being modified in unintended or uncontrolled ways. This approach works well for private assemblies (intended only for use by the application they are deployed with), but for published assemblies (intended to be shared by multiple applications) a stronger notion of identity is required.

The common language runtime addresses the problem of name uniqueness and code integrity using standard public-key cryptographic techniques. The approach is to use the public-key value as a unique prefix to the simple, developer-generated name. To ensure that only the holder of the public-private key pair can publish something with this name, the private key is used to generate a digital signature for the file. This signature can later be verified using the public-key value and has the desirable side effect of sealing the executable file. Any modifications to the executable file after signing will prevent the signature from being verified. An assembly that is digitally signed in this manner is said to have a shared name that is suitable for publishing. Shared names satisfy the following requirements:

- Guarantee name uniqueness by relying on unique key-pairs.

- Prevent others from taking over your namespace. Because you are the sole owner of your private key, no one can generate the same identity that you can. The 401k assembly that is generated with my private key has a different identity than the 401k assembly that is generated with your private key. This is particularly important when releasing subsequent versions of your assembly.

- Provide a secure notion of identity. If the common language runtime verification checks pass, you can be guaranteed that the assembly comes from the person you thought it did (assuming his private key wasn't compromised, of course). Note, however, that shared names in and of themselves only tell you that the publisher had control of the private-key, not the identity of the publisher. Microsoft Authenticode® technology provides a compatible way to establish the publisher's name if you want to use this to make trust decisions.

Shared (or published) names work by giving the assembly developer a public-private key pair. The private key is closely guarded, while the public key is published in the manifest. .NET provides a set of

APIs that compilers and "post-link" tools can use to generate shared names and manage the key pairs associated with the names. When the assembly's manifest is built, it includes references to all the files that make up the assembly. Each file listed in the manifest is hashed, and this hash value is stored along with the file's name. Once the file containing the manifest is built, that file's entire contents are hashed. This hash value is signed with the publisher's private key. The resulting RSA digital signature is stored in a reserved section of the file (that was not included in the hash). The file is now ready to be distributed.

When the file is installed on a user's machine, the system verifies the shared-name digital signature using the public key. If the signature fails to verify, then the file's contents have been tampered with and cannot be trusted. Note that the system only detects this if the file containing the manifest has been altered at install time. The detection of another file being altered occurs when the assembly is accessed at runtime.

At runtime, when a reference to an assembly is resolved, the file containing the manifest is located. At this time, the public key of the referenced assembly is compared with the public key that the referencing assembly has on record. If these keys don't match, then a TypeLoadException is raised. This also ensures that the referencing assembly was actually authored by the holder of the private key.

Finally, when other files from this assembly are loaded, the system hashes the file and compares the resulting hash value with the hash value on record for the file (saved in the referenced assembly's manifest). If the values do not match, the file's contents have been tampered with and cannot be trusted. The one drawback to this approach is that the assembly file is hashed at runtime, causing a runtime performance hit.

Earlier, I mentioned that every reference to a type is scoped by an assembly reference and that this information is carried in the metadata with each file. Another way to look at this is that every file carries information (through these references) about the components (assemblies) and their versions that it was built against. An assembly reference contains the simple name, the compatibility version number, and the locale/culture of the referenced assembly. If the referenced assembly has a shared name, the reference also carries the public key of the referenced assembly. (Actually, to save space, the referencing assembly may hash the public key and persist the hashed form, but that's purely an optimization.) Note that this is just a record of the assembly reference at the time the code was compiled. By default, .NET will allow a reference to succeed as long as a compatible version is located at runtime—it doesn't have to be the exact version requested. Using a configuration file, an administrator can tailor the exact rules governing the version bindings.

Because the identity of a type includes the identity of the assembly in whose context it was loaded, multiple versions of a single type may be loaded at one time. In fact, the common language runtime delivers a range of technologies that enable it to provide application isolation. Among other things, this means that a version of a type used by one application does not affect the version of a type used by another. If you are careful in how you access resources, components can live side-by-side in .NET. As a matter of fact, so is the common language runtime itself, but I'll leave that for another discussion.

Now that you've seen the difference between private assemblies (those without shared names) and published assemblies (those with shared names), I'd like to say a little more about deployment. Your application directory may contain all of the assemblies that the application needs to run. In that case, the common language runtime will use exactly what you deployed there because it honors the application directory first. Although this is the only place that the runtime will look for application

private assemblies, you don't have to redeploy published assemblies if you don't want to. You may rely on them being already installed on the machine or you can supply a configuration file that tells the common language runtime where they might be found—locally, on your intranet, or across the Internet. Aha, you say, what about the download scenario? What about security?

Verification and Security

Security is integral to the design of .NET. It starts as soon as a class is loaded in the form of verification checks. Then it extends to the controlling code's access to resources via code access security. It provides mechanisms for determining roles and identity information and reaches across context, process, and machine boundaries so that you can be certain that your data are not compromised in remoting scenarios. It also digs deep into the common language run-time architecture to ensure that application isolation boundaries are honored. These mechanisms work with and extend the security mechanisms commonly found in today's operating systems. Overall, .NET Framework security spans the following areas:

Type Safety

Type-safe programs reference only memory that has been allocated for their use and access objects only through their exposed interfaces. From a security standpoint, referencing only designated memory allows multiple objects to safely share a single address space. Accessing objects only through their exposed interfaces guarantees that security checks associated with specific interfaces are not circumvented. The common language runtime ensures type safety by combining strong typing in the metadata (parameters, members and array elements, return values of methods, and static values) with strong typing in the MSIL (local variables and stack slots). This provides a basis for automatically verifying the type safety of programs written in MSIL, independent of the language that produced them. By default, all code loaded by the .NET common language runtime requires verification before it is allowed to run. It is recognized that not all safe code may be verifiable with existing technology. You can bypass verification by explicitly trusting an application using administratively controlled policies.

Code Identity

There are only two ways for code to become executable: through the class loader or through interoperability services (more about this later). Both of these are services provided by the common language runtime and are part of the security perimeter. For example, the class loader maintains information about the source of every implementation that it loads. Therefore, the class loader is able to reliably provide some of the evidence upon which you can base code identity. Evidence can include information such as which Internet zone and site the code originated from, its shared name, and its publisher's identity. Using this information, security policy can control the privileges granted to a specific application or collection of related applications.

Code Access Security

This mechanism provides policy-based enforcement over privileges granted to executing assemblies. These privileges can control access to system resources and other code based on the notion of permissions. .NET provides a fairly broad set of permissions related to common resources and code evidence-based identity. Tools or application developers can extend these by adding new permission types to meet specific needs. Permissions are logically grouped into permission sets that are granted to assemblies by the .NET security system. These granted permissions define what the code is allowed to do. They are determined based on code evidence. The evidence is used to determine a set of policy code-groups for a given assembly, which is used by the security system to determine the permission set to grant. Permissions are typically demanded by trusted code such as a class that encapsulates access to the local file system. A demand causes the security system to check that the

calling assemblies have the required permission. Most applications will simply make calls to such trusted resource-managing code and will not need to incorporate any security-specific code. Resource Permissions These permissions are associated with information or communications resources within the system.

Examples include files, display windows, clipboard, network channels, application private storage, and the ability to make calls to unmanaged APIs. (Application private storage is provided by IsolatedStorage, a new set of types and methods supported by the .NET platform.) These types of permissions can be used to determine if code has rights to access specific resources at either load time or runtime (see Declarative Security, discussed shortly). Consider the case of an application that wants to read and write a file in C:\Temp. To open a file in this directory, a managed application would use the base framework's System.IO.File class. This class contains a static method called Open. Internally, the Open method takes the path name of the file that the caller wants to open, and asks the common language runtime engine if access to the specified path name is allowed. This is called demanding access. In this case, the Open method demands that the calling code be granted a FileIO permission with read and write access to C:\Temp*.* before it will attempt to open the file.

When a method demands access, the common language runtime walks up the call stack and checks to see if all assemblies in the call chain have the required permission. If any code in the call chain lacks the required permission, then the demand fails and a security exception is thrown; otherwise, the demand succeeds. To complete the File.Open operation, the File class needs to call the OS file system APIs. This requires the new UnmanagedCode permission, which is typically granted to highly trusted resource-managing classes. The system will check that the File class has this permission before allowing the call to unmanaged code. You should note that the other semi-trusted assemblies in the call chain aren't expected to have been granted this permission. To allow the UnmanagedCode permission demand to succeed, highly trusted code is allowed to assert its right to use a permission. This causes the permission-check stack walk to terminate at the assembly that asserted the permission.

Identity Permissions

These permissions are based on evidence associated with a given assembly (Site, Zone, Shared Name, and Publisher). They allow you to control access to class methods based on this information. Identity permissions operate in the same manner as resource permissions. Code demands that calling code have a given identity attribute. If the callers have the demanded identity attribute, the call succeeds; otherwise a security exception is thrown.

Declarative Security

This powerful mechanism allows you to insert code access security checks into your code by annotating classes, fields, or methods. The declarations are encoded in the assembly metadata and are enforced by the .NET security system. Using these declarations, you can request checks to be performed at either load time or runtime. Though they are more limited, load-time checks are more efficient in that they are only checked once during execution, as opposed to checks on each call to a method. They only check the permissions (resource or identity) of the immediate caller, so you need to carefully assess whether this is adequate for a given situation. Typically, load-time checks are used to limit access to specific classes. For example, you can use a load-time identity check to ensure that only code from a specific publisher or with a given shared name public-key can be the immediate caller to a class. You can also use these checks to limit the ability to subclass or override virtual methods.

Declarative checks at runtime are a simple means to indicate permission demands and asserts for a given type. As mentioned earlier, demands cause a full stack walk to check the permissions of all code in the call chain. They are appropriate when the requested action and permission attributes are static; if you always demand read permission to C:\Temp, it is easy to express this declaratively. But if you need to dynamically create the check to account for different access modes and directory/file names, then imperative checks (described next) are needed.

Imperative Security

These checks are implemented as code within a method by the application developer. They are enforced in exactly the same manner as declarative runtime checks. The benefit of using them is that you can support dynamic determination of specific permissions that are required in a given execution context. This is typically important for permissions such as FileIO, IsolatedStorage, UI, and so forth, that support multiple
access modes and subsetting of the resources being accessed.

Policy-based Security

Enabling effective use of code access security requires a policy-based system. Policy allows you to make specific statements about what the code should be allowed to do based on its point of origin and other identifying information. Using this policy, appropriate privileges are automatically granted to the code without any required user interaction. .NET is slated to ship with a default policy that reflects the different levels of trust that users typically place on code on their local system, code from known trusted servers, and code from untrusted Internet sites. The user may refine this policy as they see fit. Role-based Security It is fairly common for developers to make authorization decisions based on the identity or roles associated with the execution context. You often see this in financial or business systems as a means of policy enforcement. For example, you may impose limits on the value of a transaction, depending on the role of the user making the request. Clerks may be able to process transactions up to some set value, supervisors to some higher limit, and VPs to an even higher limit. .NET provides services to allow applications to easily incorporate such logic in a way that is platform-independent and scalable to the Internet. The .NET mechanisms are built around the notion of identities and principals. Identities encapsulate entity naming that the application understands. This can map to underlying OS notions of accounts, but can also be application-defined. The corresponding principal encapsulates the identity, along with corresponding role information. Again, this may or may not map to an operating system notion such as groups.

In operation, a trusted authentication provider will typically compute the principal information based on credentials associated with a given request (HTTP Put, remote procedure call, and so on). The principal is then attached to the execution context, making it available to application code to check the role or identity
information prior to taking a given action.

Remoting Security

.NET provides support for remote object invocation that can span AppDomains, processes, or machines. (An AppDomain is a logical application; multiple AppDomains may run in a single Win32® process.) When crossing machine boundaries, security issues such as authentication, authorization, confidentiality, and integrity become critical. .NET will provide support for these mechanisms in a way that's compatible with existing network protocols and will integrate tightly into the remoting infrastructure. This will make it simple for applications to incorporate these security features.

Authentication mechanisms will be suitable for identification of users as well as applications or business entities. A key-based identity infrastructure will be provided for managing such identities and

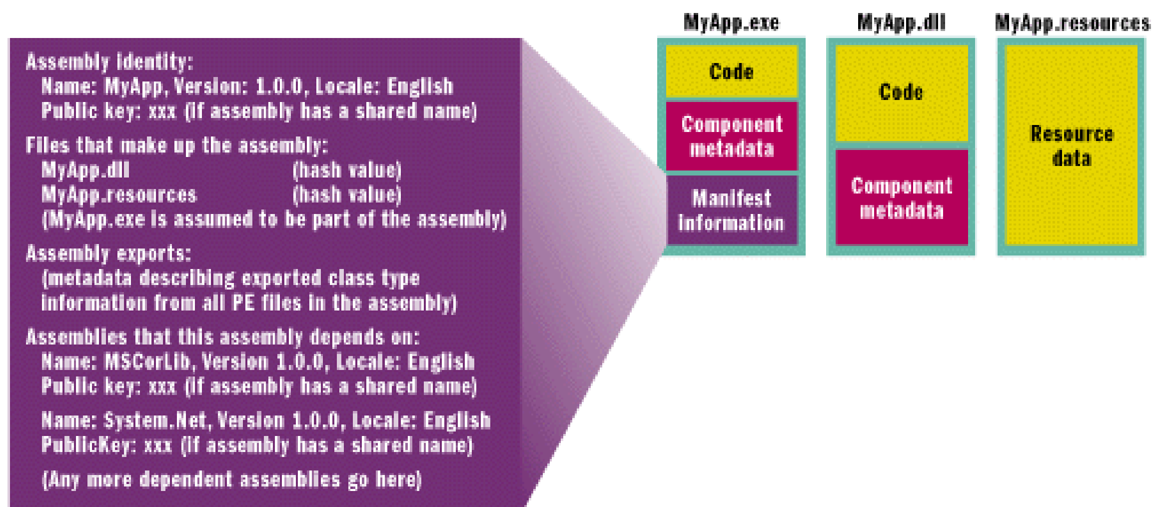
integrating with key-based authentication protocols. Confidentiality and integrity services will take advantage of cryptographic techniques. It will be possible to use point-to-point mechanisms such as Secure Sockets Layer (SSL) and IP Security Protocol (IPSec). Application layer encryption, integrity, and digital signature technologies at the messaging level will also be supported in the .NET Framework.

Cryptography

.NET will provide a set of cryptographic objects supporting well-known algorithms and common uses (hashing, encryption, and digital signatures). These objects are designed in a manner that facilitates the incorporation of these basic capabilities into more complex operations, such as signing and encrypting a document. Cryptographic objects are used by .NET to support internal services, but are also available to developers who need cryptographic support.

Putting it all Together

Now that you've been introduced to the key concepts of the .NET common language runtime, let's actually deploy and run the code.Say I'm building an application that's a single assembly containing multiple files, as shown in Figure 3.



The assembly manifest is embedded in MyApp.exe. It shows the three files that make up the assembly, which types are defined in the assembly's files (MyApp.exe and MyApp.dll), and what other assemblies the MyApp assembly requires to execute. In this example, MyApp requires the .NET base framework assembly (version 1.0, English). Let's say that my assembly has been copied to a user's machine and resides on their local hard disk. One of the assemblies that my application references has been installed into the global assembly cache on that user's machine (described in a little more detail later), so that any application can just find it there. The other assembly is not deployed with my application, but on a server on my company's intranet.

The user double-clicks on my EXE. Since the PE file is a .NET file, the common language runtime initializes and establishes a "domain" for the application. The assembly is then loaded into the domain, essentially just reading in the assembly manifest. (Note that the common language runtime does not verify that all of its files are present or that referenced assemblies are available. Downloading an assembly really just downloads the file containing the manifest.)

Next, the entry point for the application is loaded, verified, JITted (compiled with the JITCompiler), and the code begins to execute. As the code executes, it may make references to other types. These references will be in one of three forms: a reference to a type in the same file, a reference to a type in another file in the same assembly, or a reference to a type in a dependent assembly. If it is a reference to a type in the same file, the reference is early bound and the type is loaded out of the file directly. If it's a reference to a type in another file in the same assembly, the common language runtime makes sure that the file being referenced is, in fact, in the file list in the current assembly's manifest and then looks in the application's directory for the file to load. Finally, if it's a reference to a type in a dependent assembly, the common language runtime resolves the reference using the following heuristics:

1. The common language runtime passes the assembly reference to the assembly resolver. The assembly resolver will pick up from the application's directory any configuration information that might affect the binding rules, such as version binding, and the result may be a modified form of the persisted assembly reference.

2. If the reference does not have a shared name (that is, does not include a public key and is thus considered a private assembly): a.The assembly resolver looks in the specified application directory for a file with the same name as the assembly. If found, the assembly resolver makes sure that all of the attributes passed with the reference match those in the located manifest. (Actually, the heuristics are a little more complex than this, involving looking for naming patterns in the subdirectories as well.) b.If successful, the assembly resolver returns a pointer to the assembly manifest at its persisted file location. c.If not successful, a TypeLoadException is raised.

3. If the reference has a shared name (that is, has been published with a public key, as in my example): a.The assembly resolver looks in the specified application directory for a file with the same name as the assembly. If found, the assembly resolver makes sure that all of the attributes passed with the reference match those in the located manifest, including using the public key to check the manifest file signature. The assembly resolver first looks in the application directory to give the application an opportunity to deploy a preferred version of the assembly, even if there might be another version already loaded in the global assembly cache. b.If successful, the assembly resolver returns a pointer to the assembly manifest at its persisted file location. c.If the file is not found in the application directory, the assembly resolver looks in its global cache to see if an appropriate version of the assembly has already been installed. If that is the case, the assembly resolver simply returns a pointer to the assembly manifest in the global assembly cache. d.If the file is not found in the specified application directory or in the global assembly cache, the assembly resolver uses a search path provided in the configuration information for the application. If the assembly was located off the machine, it is downloaded into a transient assembly cache. e.If the file is not found in any of these locations, a TypeLoadException is raised.

The assembly cache is a directory normally residing in the \WinNT\Assembly directory. When assemblies are installed on the machine, they can get merged into the assembly cache. The assembly cache consists of two logically separate caches: the global assembly cache and the transient assembly cache. When using MSI files to distribute an assembly, the Windows installer package (the MSI file) can be authored so that assemblies are automatically added to the global assembly cache. When assemblies are downloaded using Microsoft Internet Explorer, the assembly is installed in the transient assembly cache. Keeping these assemblies logically separate prevents a downloaded module from adversely affecting an installed application. Note that the assembly cache can hold multiple versions of an assembly. For example, the assembly cache can contain Versions 1

and 2 of MyAssem.dll. If an application was built and tested using Version 1 of MyAssem.dll, then the common language runtime will load Version 1 of MyAssem.dll for that application even though a later version of the assembly exists in the cache. An administrator can change this version policy, but doing so is not recommended because the strict version policy resolves the classic DLL Hell problems. The application should behave as it always has because the code that it is executing is the same code that it was built and tested with. This also means that Version 2 of MyAssem.dll doesn't have to maintain backward compatibility with Version 1.

Interoperability with Unmanaged Code

The managed code environment provided by the common language runtime hosts a ton of features that make software development quite a pleasure. However, the operating system on which the common language runtime executes also offers a number of features, and it would be a terrible misfortune if managed applications were prevented from using the native operating system's services or other unmanaged code. To this end, the common language runtime offers a System.Runtime.InteropServices namespace. This namespace contains a set of types that allow managed applications to access unmanaged code. Specifically, there are three supported interop scenarios: managed code calling unmanaged DLL functions, managed code instantiating and calling interface methods of COM servers, and unmanaged code instantiating and calling methods on .NET servers. To call an unmanaged DLL function, you just need to tell the common language runtime the name of the function you want to call (such as MessageBoxA), the name of the DLL containing the function (such as User32.dll), and how to marshal the function's parameters (for example, which parameters are input, output or input/output parameters).

To work with a COM server, a .NET wrapper must be created so that .NET clients think that they are calling .NET code; the underlying details are handled seamlessly by the common language runtime. The TlbImp.exe utility parses a COM type library and produces a managed DLL whose metadata describes the methods that wrap the COM server's interface methods. A managed application can now create instances of the object and use it as though it were a native managed type. For each COM server, the runtime generates a runtime-callable wrapper. This wrapper acts as a proxy between the managed and unmanaged code, handling all administrative tasks such as marshaling, AddRef, Release, and so on.

For an unmanaged application to instantiate and use a managed .NET server, the .NET object must be added to the system's registry (since that's how COM expects to locate COM servers). The register assembly utility, RegAsm.exe, generates a GUID for the .NET server and updates the registry. If the unmanaged code wants a type library, one can be generated using the TlbExp.exe utility. An unmanaged application can now create instances of the object and use it as though it were a COM server. For each type of managed .NET server, the common language runtime generates a COM-callable wrapper. This wrapper acts as a proxy between the unmanaged and managed code, handling all administrative tasks such as marshaling, AddRef, Release, and so on. Due to the common language runtime's interoperability features, developers do not have to give up anything in order to gain the benefits offered by the common language runtime. In addition, the common language runtime makes working with COM servers even easier than it would be from unmanaged C/C++ code. Of course, transitioning between managed and unmanaged code has its performance issues, so you should try to reduce these transitions when possible. This may mean porting your COM server to .NET. At least the interoperability features allow you to make this decision on your own time when you're ready.

Just the Beginning

I haven't described the services Microsoft is delivering via its class libraries. In particular, Microsoft is creating a new database access model, called ADO+, that exposes database functionality via .NET types contained in the System.Data namespace. Microsoft is also creating a class library, called WinForms, for developing standalone apps with rich user interfaces. You can find these types in the System.WinForms namespace. Finally, Microsoft is providing a new architecture called Active Server Pages+ (ASP+) for producing Web applications. This architecture includes a class library of types that render complex visuals as HTML. The architecture also manages all client/server state information, making rapid Web application development a breeze. You can learn more about these types by examining the System.Web namespace.

Users might not appreciate the common language runtime and its capabilities, but they will certainly notice the quality and features of the applications that utilize it. In addition, developers will certainly appreciate how the common language runtime allows applications to be built and deployed more rapidly and with less administration than Windows has ever allowed in the past.