
Chapter 1

Architecture of the .NET Framework

The .NET Framework development platform introduces many new concepts, technologies, and terms. The goal of this chapter is to give an overview of the .NET Framework: to show how it is architected, to introduce some of the new technologies, and to define many of the new terms. I'll also take you through the process of building your source code into an application or a set of redistributable components (types), and then explain how these components execute.

Compiling Source Code into Managed Modules

OK, so you've decided to use the .NET Framework as your development platform. Great! Your first step is to determine what type of application or component you intend to build. Let's just assume that you've handled this minor detail, everything is designed, the specifications are written, and you're ready to start development.

Next, you must decide what programming language to use. This is usually a difficult task because different languages offer different capabilities. For example, in unmanaged C/C++, you have pretty low-level control of the system. You can manage memory exactly the way you want to, create threads easily if you need to, and so on. Visual Basic 6, on the other hand, allows you to build UI applications very rapidly and allows the easy control of COM objects and databases.

If you use the .NET Framework, your code targets the common language runtime (CLR), which affects your decision about a programming language. The common language runtime is just what its name says it is: A runtime that is usable by different and varied programming languages. The features of the CLR are available to any and all programming languages that target it-period. If the runtime uses exceptions to report errors, then all languages get errors reported via exceptions. If the runtime allows you to create a thread, then any language can create a thread.

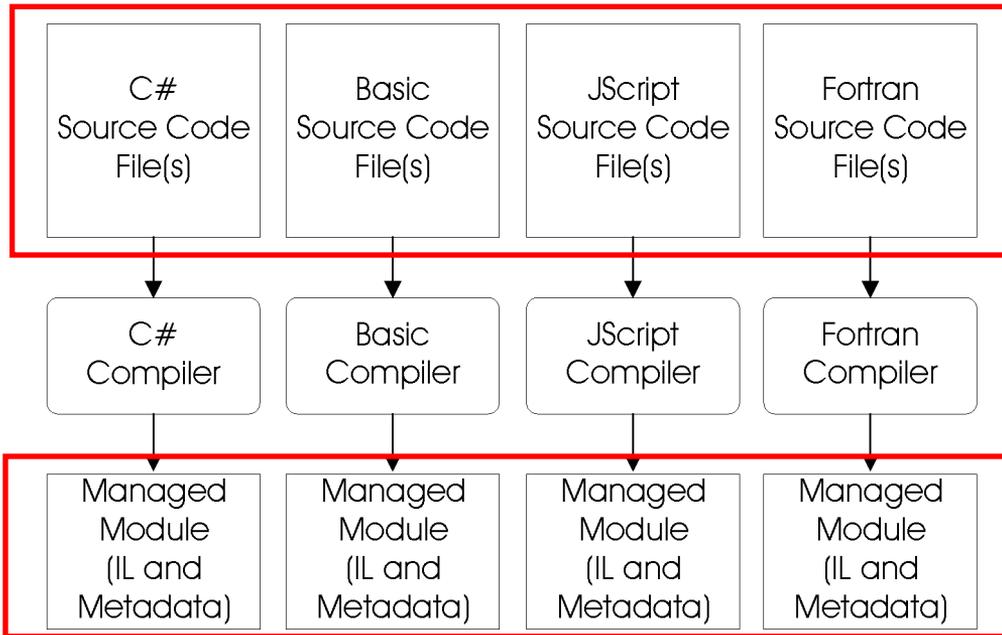
In fact, at runtime, the CLR has no idea which programming language the developer used for the source code. This means that you should choose whatever programming language allows you to express your intentions most easily. You may develop your code in any programming language you desire as long as the compiler you use to compile your code targets the CLR.

So if what I say is true, then what is the advantage of using one programming language over another? Well, I think of compilers as syntax checkers and "correct code" analyzers. They examine your source code, ensure that whatever you've written makes some sense, and then output code that describes your intention. Simply put, different programming languages allow you to develop using different syntax. Don't underestimate the value of this. For mathematical or financial applications, expressing your intentions using APL syntax can save many days of development time when compared to expressing the same intention using Perl syntax, for example.

Microsoft is creating several language compilers that target the runtime: C++ with managed extensions, C# (pronounced "C sharp"), Visual Basic.NET, JScript, Java, and an intermediate language (IL) Assembler. In addition to Microsoft, there are several other companies creating compilers that produce code that targets the CLR. At this writing, I am aware of compilers for Alice, APL, COBOL, Component Pascal, Eiffel, Fortran, Haskell, Mercury, ML, Mondrian, Oberon, Perl, Python, RPG, Scheme, and Smalltalk.

The figure on the next page shows the process of compiling source code files:

Compiling Source Code into Managed Modules



As the figure shows, you can create source code files using any programming language that supports the CLR. Then, you use the corresponding compiler to check syntax and analyze the source code. Regardless of which compiler you use, the result is a *managed module*. A managed module is a standard Windows portable executable (PE) file that requires the CLR to execute. In the future, other operating systems may use the PE file format as well.

A Managed Module is composed of the following parts:

Part	Description
PE header	This is the standard Windows PE file header, which is similar to the Common Object File Format (COFF) header. The PE header indicates the type of file--GUI, CUI, or DLL—and also has a timestamp indicating when the file was built. For modules that contain only IL code (see below, Intermediate Language Code), the bulk of the information in the PE header is ignored. For modules that contain native CPU code, this header contains information about the native CPU code.
CLR header	This header contains the information (interpreted by the CLR and utilities) that makes this a managed module. It includes the version of the CLR required, some flags, the MethodDef metadata token of the managed module's entry point method (Main method), and the location/size of the module's metadata, resources, strong name, some flags, and other less interesting stuff.
Metadata	Every managed module contains metadata tables, of which there are 2 main types: those that describe the types and members <i>defined in your source code</i> , and those that describe the types and members <i>referenced by your source code</i> .

Part	Description
Intermediate Language (IL) Code	This is the code that was produced by the compiler as it compiled the source code. IL is later compiled by the CLR into native CPU instructions.

Most compilers of the past produced code targeted to a specific CPU architecture, such as x86, IA64, Alpha, or PowerPC. All CLR-compliant compilers produce intermediate language (IL) code instead. IL code is sometimes referred to as managed code, because its lifetime and execution are managed by the CLR. IL code is discussed later in this chapter.

In addition to emitting IL, every compiler targeting the CLR is required to emit full metadata into every managed module. In brief, metadata is simply a set of data tables that describe what is defined in the module, such as types and their members. In addition, metadata also has tables indicating what the managed module references, such as imported types and their members. Metadata is a superset of older technologies such as type libraries and IDL files. The important thing to note is that CLR metadata is far more complete than its predecessors. And, unlike type libraries and IDL, metadata is always associated with the file that contains the IL code. In fact, the metadata is always embedded in the same EXE/DLL as the code, making it impossible to separate the two. Since the metadata and code are produced by the compiler at the same time and are bound into the resulting managed module, the metadata and the IL code it describes are never out of sync with one another.

Metadata has many uses. Here are some of them:

- Metadata removes the need for header and library files when compiling, because all the information about the referenced types/members is contained in one file along with the IL that implements those type/members. Compilers can read metadata directly from managed modules.
- Visual Studio uses metadata to help you write code. Its IntelliSense feature parses metadata to tell you what methods a type offers and what parameters that method expects.
- The CLR code verification process uses metadata to ensure that your code performs only “safe” operations. Verification is discussed shortly.
- Metadata allows an object’s fields to be serialized into a memory block, remoted to another machine, and then deserialized, recreating the object and its state on the remote machine.
- Metadata allows the garbage collector to track the lifetime of objects. For any object, the garbage collector can determine the type of the object, and from the metadata it knows which fields within that object refer to other objects.

The next chapter, “Building, Packaging, Deploying, and Administering Applications and Types,” will describe metadata in much more detail. And a little later in this chapter, we’ll explore intermediate language in more detail.

Four of the compilers that Microsoft offers—C#, Visual Basic, JScript, and the IL Assembler—always produce managed modules, which require the CLR to execute. That is, end users must have the CLR installed on their machines in order to execute any managed modules. This situation is similar to the one that end users face with MFC or VB 6 applications: they must have the MFC or VB DLLs installed in order to run them.

The Microsoft C++ compiler, by default, builds unmanaged modules: the EXE or DLL files with which we are all familiar. These modules do not require the CLR in order to execute. However, by specifying a new command-line switch, the C++ compiler can produce managed modules that do require the CLR to execute. Of all the Microsoft compilers mentioned, C++ is unique in that it is the only language that allows the developer to write both managed and

Applied .NET Framework Programming

unmanaged code and have it emitted into a single managed module. This can be a great feature because it allows developers to write the bulk of their applications in managed code (for type-safety and component interoperability) but continue to access their existing unmanaged C++ code.

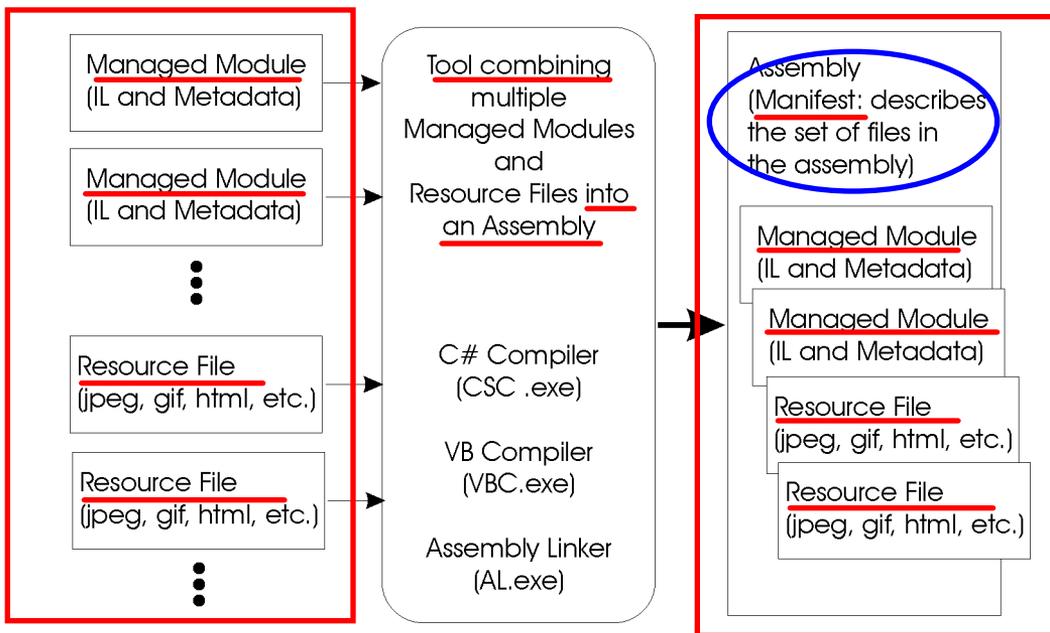
Combining Managed Modules into Assemblies

The CLR doesn't actually work with modules; it works with assemblies. An assembly is an abstract concept, which can be difficult to grasp at first. First, an assembly is a logical grouping of one or more managed modules or resource files. Second, an assembly is the smallest unit of reuse, security, and versioning. Depending on the choices you make with your compilers or tools, you can produce a single-file assembly or you can produce a multi-file assembly.

Chapter 2, "Building, Packaging, Deploying, and Administering Applications and Types," discusses assemblies in great detail, so I don't want to spend a lot of time on it here. All I want to do now is make you aware that there is this extra conceptual notion that offers a way to treat a group of files as a single entity.

The figure below should help explain what assemblies are about:

Combining Managed Modules into Assemblies



In this figure, we are passing the file names of some managed modules and resource (or data) files to a tool. This tool produces a single PE file that represents the logical grouping of files. This PE file contains a block of data called the manifest, which is simply another set of metadata tables. These tables describe the assembly: the files that make it up, the publicly exported types implemented by the files in the assembly, and the resource or data files that are associated with it.

By default, compilers actually do the work of turning the emitted managed module into an assembly. That is, the C# compiler emits a managed module that contains a manifest. The manifest indicates that the assembly consists of just the

one file. So for projects that have just one managed module and no resource files, the assembly will be the managed module, and you don't have any additional steps to perform during your build process. If you wish to group a set of files into an assembly, then you will have to be aware of more tools (such as the assembly linker, AL.exe) and their command-line options. These tools and options are explained in the next chapter.

An assembly allows you to decouple the logical and physical notions of a reusable, deployable, versionable component. The way in which you partition your code and resources into different files is completely up to you. For example, you could put rarely used types or resources in separate files that are part of an assembly. The separate files could be downloaded from the web as needed. If the files are never needed, they're never downloaded, saving disk space and reducing installation time. An assembly lets you break up the physical deployment of the files but still treat them as a single collection.

The modules in an assembly also include information, including version numbers, about referenced assemblies. This information makes an assembly *self-describing*. In other words, the CLR knows everything that an assembly needs in order to execute. No additional information is required in the registry or in the Active Directory, so deploying assemblies is much easier than deploying unmanaged components.

Loading the Common Language Runtime

Each assembly that you build can either be an executable application or a DLL containing a set of types (components) for use by an executable application. Of course, the CLR is responsible for managing the execution of code contained within these assemblies. This means that the .NET Framework must be installed on the host machine. Microsoft has created a redistribution package that you can freely ship to install the .NET Framework on your customer's machines. Future versions of Windows will include the .NET Framework, at which point you will no longer need to ship it with your assemblies.

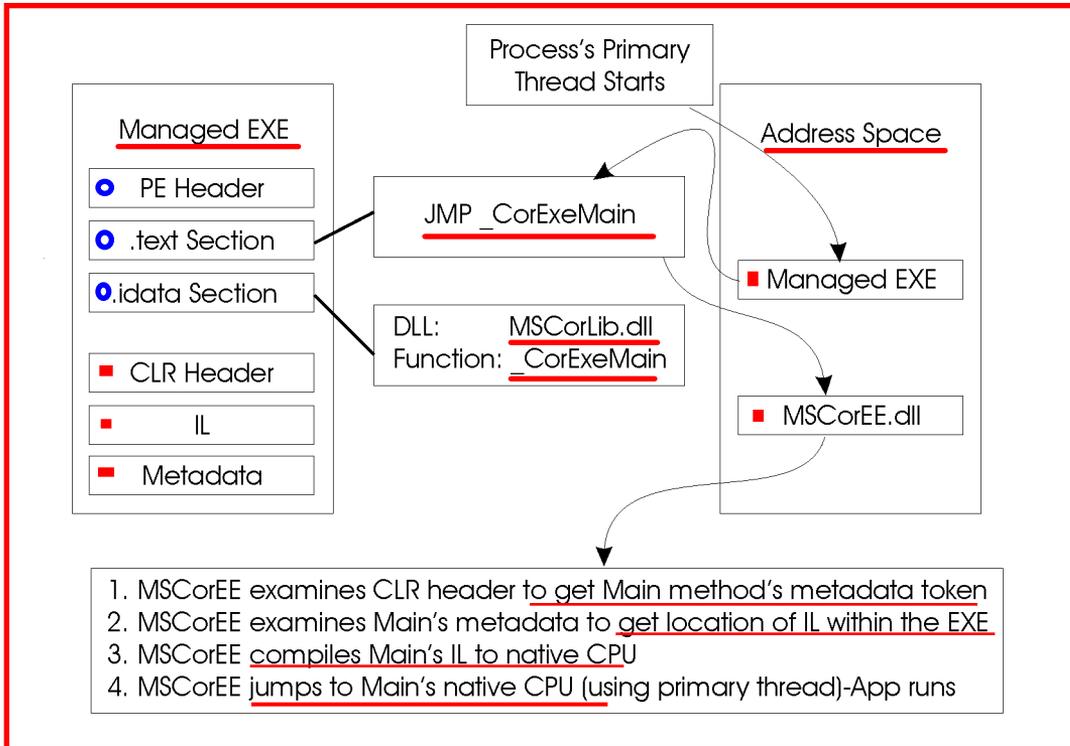
You can tell whether the .NET Framework has been installed by looking for the `MSCorEE.dll` file in the `%windir%\system32` directory. The existence of this file tells you that the .NET Framework is installed. However, several versions of the .NET Framework may be installed on a single machine simultaneously. If you want to determine exactly which versions of the .NET Framework are installed, examine the subkeys under the following registry key: `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\NETFramework\policy`.

When you build an EXE assembly, the compiler/linker emits some special information into the resulting assembly's PE File header and the file's .text section. When the EXE file is invoked, this special information causes the CLR to load and initialize. Then the CLR locates the entry point method for the application and lets the application start executing.

Similarly, if an unmanaged application calls `LoadLibrary` to load a managed assembly, the entry point function for the DLL knows to load the CLR in order to process the code contained within the assembly.

For the most part, you do not need to know about or understand how the CLR actually gets loaded. For the curious, however, I will explain how a managed EXE or DLL starts the CLR. If you're not interested in this, feel free to skip to the next section.

The figure on the next page summarizes how a managed EXE loads and initializes the CLR.



When the compiler/linker creates an executable assembly, the following 6-byte x86 stub function is emitted into the .text section of the PE file:

```
JMP _CorExeMain
```

The **_CorExeMain** function is imported from the Microsoft MSCorEE.dll dynamic-link library, and therefore MSCorEE.dll is referenced in the import (.idata) section of the assembly file. (MSCorEE.dll stands for Microsoft Component Object Runtime Execution Engine.) When the managed EXE file is invoked, Windows treats it just like any normal (unmanaged) EXE file: the Windows loader loads the file and examines the .idata section to see that MSCorEE.dll should be loaded into the process's address space. Then, the loader obtains the address of the **_CorExeMain** function inside MSCorEE.dll and fixes up the stub function's **JMP** instruction in the managed EXE file.

The primary thread for the process begins executing this x86 stub function, which immediately jumps to **_CorExeMain** in MSCorEE.dll. **_CorExeMain** initializes the CLR and then looks at the CLR header for the executable assembly to determine what managed entry point method should execute. The IL code for the method is then compiled into native CPU instructions, after which the CLR jumps to the native code (using the process's primary thread). At this point, the managed application code is running.

The situation is similar for a managed DLL. When building a managed DLL, the compiler/linker emits a similar 6-byte x86 stub function for a DLL assembly in the .text section of the PE file:

```
JMP _CorDllMain
```

The `_CorDllMain` function is also imported from the `MSCorEE.dll`, causing the `.idata` section for the DLL to reference `MSCorEE.dll`. When Windows loads the DLL, it automatically loads `MSCorEE.dll` (if it isn't already loaded), obtains the address of the `_CorDllMain` function, and fixes up the 6 byte `x86 JMP` stub in the managed DLL. The thread that called `LoadLibrary` to load the managed DLL then jumps to the `x86` stub in the managed DLL assembly, which immediately jumps to the `_CorDllMain` in `MSCorEE.dll`. `_CorDllMain` initializes the CLR (if it hasn't already been initialized for the process) and then returns so that the application can continue executing as normal.

These 6-byte `x86` stub functions are required to run managed assemblies on Windows 98, Windows 98SE, Windows ME, Windows NT 4, and Windows 2000 because all these operating systems shipped long before the CLR became available. Note that the 6-byte stub function is specifically for `x86` machines. This stub does not work properly if the CLR is ported to run on other CPU architectures. Because Windows XP and the Windows .NET Servers support both the `x86` and the IA64 CPU architectures, the loader for Windows XP and the Windows .NET Servers was modified to look specifically for managed assemblies.

On Windows XP and the Windows .NET Servers, when a managed assembly is invoked (typically via `CreateProcess` or `LoadLibrary`), the OS loader detects that the file contains managed code. It does this by examining directory entry 14 in the PE file header (see `IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR` in `WinNT.h`). If this directory entry exists and is not 0, then the loader ignores the file's import (`.idata`) section entirely and knows to automatically load `MSCorEE.dll` into the address space for the process. Once loaded, the OS loader makes the process thread jump directly to the correct function in `MSCorEE.dll`. The 6-byte `x86` stub functions are ignored on machines running Windows XP and the Windows .NET Servers.

One last note about managed PE files: Managed PE files always use the 32-bit PE file format; they do not use the newer 64-bit PE file format. On 64-bit Windows systems, the OS loader detects the managed 32-bit PE file and automatically knows to create a 64-bit address space.

Executing the Code in Your Assembly

As mentioned earlier, managed modules contain both metadata and intermediate language (IL) code. IL is a CPU-independent machine language created by Microsoft after consultation with several external commercial and academic language/compiler writers. IL is much higher-level than most CPU machine languages. IL 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 throw and catch exceptions for error handling. You can think of IL as an object-oriented machine language.

Usually, developers prefer to program in a high-level language, such as C# or Visual Basic.NET. The compilers for all these high-level languages produce IL. However, like any other machine language, IL can be written in assembly language and Microsoft does provide an IL assembler, `ILAsm.exe`. Microsoft also provides an IL disassembler, `ILDasm.exe`.

Some people are concerned that IL does not offer enough intellectual property protection for their algorithms. In other words, you could build a managed module and someone else could use a tool, like an IL disassembler, to reverse engineer exactly what your application code does.

Yes, it's true that IL code is higher-level than most other assembly languages and, in general, reverse engineering IL code is relatively simple. However, when you implement a web service or web form

Applied .NET Framework Programming

application, your managed module resides on your server, inaccessible to anyone outside your company. Outsiders cannot use any tool to see the IL code, so your intellectual property is completely safe.

If you are concerned about any of your managed modules that you do distribute, then you can use the Microsoft obfuscator utility (OB.exe), which is downloadable from <http://www.GotDotNet.com>. This utility “scrambles” the names of all the private symbols in your managed module metadata. It will be difficult for someone to “unscramble” the names and understand the purpose of each method. Note that the Microsoft obfuscator scrambles metadata names only and does not scramble the IL code in anyway since the CLR must be able to process unscrambled IL.

If you don't feel that the obfuscator offers the kind of intellectual property protection that you desire, then you can consider implementing your more-sensitive algorithms in some unmanaged module, which will contain native CPU instructions instead of IL and metadata. Then, you can use the CLR's interoperability features to communicate between the managed and unmanaged portions of your application. Of course, this assumes that you're not worried about people reverse engineering the native CPU instructions in your unmanaged code.

Any high-level language will most likely expose only a subset of the facilities offered by the CLR. IL assembly language, however, gives a developer access to all the facilities of the CLR. So if your programming language of choice hides a CLR feature that you really wish to take advantage of, you can write that portion of your code in IL assembly or in another programming language that exposes the CLR feature you seek.

The only way for you to know what facilities are offered by the runtime is to read documentation specific to the CLR itself. In this book, I try to concentrate on CLR features and how they are exposed or not exposed by the C# language. I suspect that most other books and articles will present the CLR via a particular language and that most developers will come to believe that the CLR offers only those features that the developer's chosen language exposes. As long as your language allows you to accomplish what you're trying to do, this blurred perspective is not a bad thing.

Personally, I feel that an ability to switch programming languages easily with rich integration between languages is an awesome feature of the CLR. I also believe that this is a feature that will, unfortunately, often be overlooked by developers.

Programming languages like C# and Visual Basic.NET are excellent languages for doing I/O operations. APL is an awesome language for doing advanced engineering or financial calculations. Through the CLR, you can write the I/O portions of your application using C# and then write the engineering calculations using APL. The CLR offers a level of integration between these languages that is unprecedented, and really makes mixed-language programming worthy of consideration for many development projects.

An important thing to note about IL is that it is not tied to any specific CPU platform. This means that a managed module containing IL can run on any CPU platform as long as the operating system running on that CPU platform hosts a version of the CLR. Although the initial release of the CLR will run only on 32-bit Windows platforms, developing an application using managed IL sets up a developer to be more independent of the underlying CPU architecture.

In October 2000, Microsoft (along with Intel and Hewlett-Packard as co-sponsors) proposed a large subset of the .NET Frameworks to the ECMA (the European Computer Manufacturer's Association) for the purpose of standardization. The ECMA accepted this proposal and created a technical committee (TC39) to oversee the standardization process. The technical committee is charged with the following duties:

- **Technical Group 1:** Develop a dynamic scripting language standard (ECMAScript). Microsoft's implementation of ECMAScript is JScript.
- **Technical Group 2:** Develop a standardized version of the C# programming language.
- **Technical Group 3:** Develop a common language infrastructure (CLI) based on a subset of the functionality offered by the .NET Framework's CLR and class library. Specifically, the CLI defines a file format, a common type system, an extensible metadata system, an intermediate language (IL), and access to the underlying platform (P/Invoke). In addition, the CLI also defines a factorable (to allow for small hardware devices) base class library designed for use by multiple programming languages.

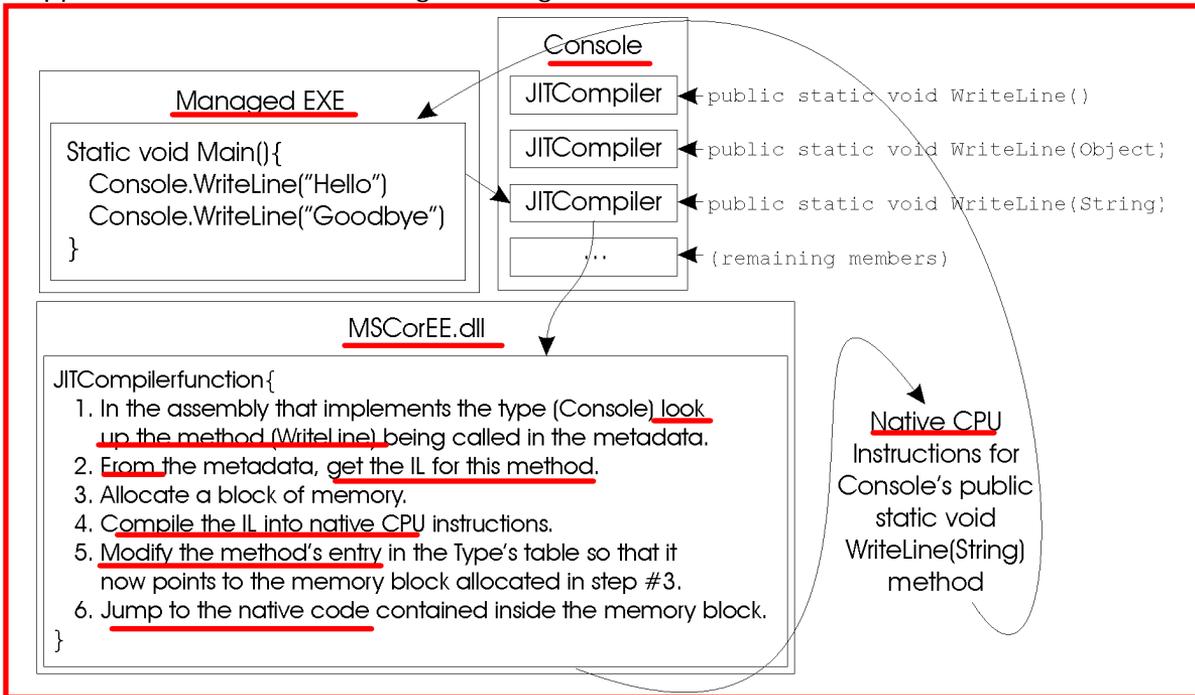
Once the standardization is completed, these standards will be contributed to ISO/IEC JTC 1 (Information Technology). At this time, the technical committee will investigate further directions for CLI, C#, and ECMAScript, as well as entertain proposals for any complementary or additional technology. For more information about ECMA, please see <http://www.ECMA.ch> and <http://MSDN.Microsoft.com/Net/ECMA>.

With the standardization of the CLI, C#, and ECMAScript, Microsoft won't "own" any of these technologies. Microsoft will simply be one company of many (hopefully) that are producing implementations of these technologies. Certainly Microsoft hopes that their implementation will be the best in terms of performance and customer-demand-driven features. This is what will help sales of Windows, since the Microsoft "best of breed" implementation will only run on Windows. However, other companies are free to implement these standards and compete against Microsoft.

Of course, IL instructions cannot be executed *directly* by today's CPUs (although this may change someday). In order to execute a method, its IL code must first be converted to native CPU instructions. To make this conversion, the CLR provides a JIT (just-in-time) compiler.

The figure below shows what happens the first time a method is called.

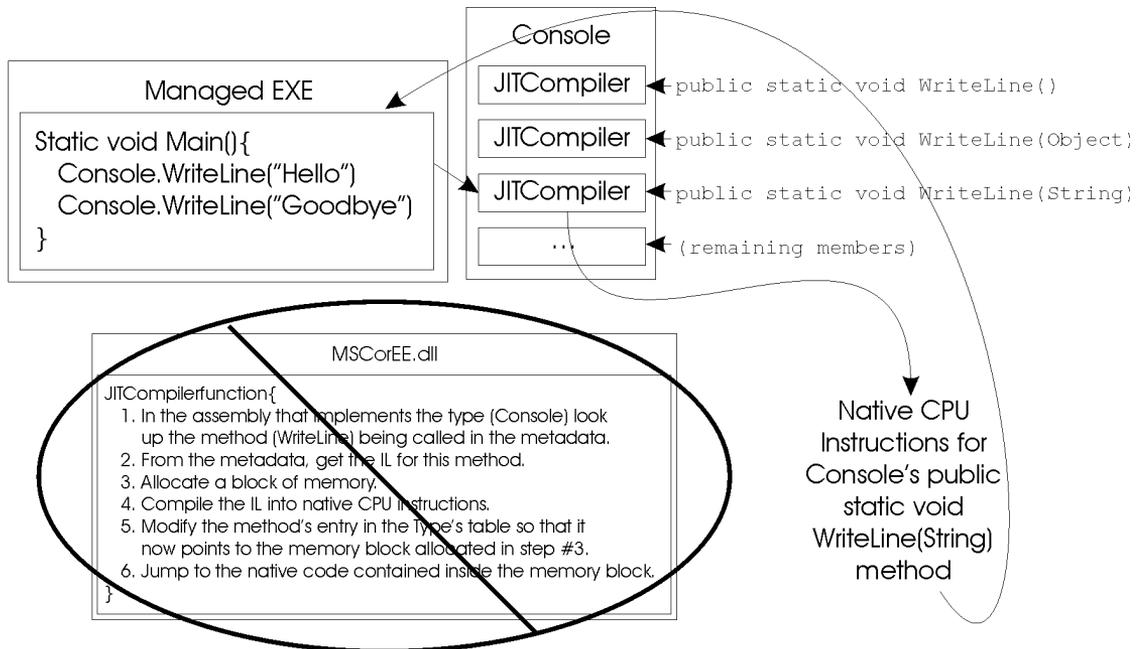
Applied .NET Framework Programming



- Just before the `Main` method executes, the CLR detects all the types that are referenced by the code in `Main`. This causes the CLR to allocate an internal data structure that is used to manage access to each referenced type. In the figure above, the `Main` method refers to a single type, `Console`, causing the CLR to allocate a single internal structure.
- This internal data structure contains an entry for each method defined by the type. Each entry holds the address where the method implementation can be found. When initializing this structure, the CLR sets each entry to a function contained inside the CLR itself. I call this function `JITCompiler`.
- When `Main` makes its first call to `WriteLine`, the `JITCompiler` function is called. The `JITCompiler` function is responsible for compiling a method's IL code into native CPU instructions. Since the IL is being compiled "just in time", this component of the CLR is frequently referred to as a *Jitter* or *JIT Compiler*.
- When called, the `JITCompiler` function knows what method is being called and what type defines this method.
- The `JITCompiler` function then searches the defining assembly's metadata for the called method's IL. `JITCompiler` verifies and compiles the IL code into native CPU instructions, which are then saved in a dynamically allocated block of memory. `JITCompiler` goes back to the type's internal data structure and replaces the address of the called method with the address of the block of memory containing the native CPU instructions.
- Finally, `JITCompiler` jumps to the code in the memory block. This code is the implementation of the `WriteLine` method (the version that takes a `String` parameter). When this code returns, execution resumes in `Main`, as normal.

Suppose now that `Main` calls `WriteLine` a second time. This time, because the code for `WriteLine` has already been verified and compiled, the call goes directly to the native code in the memory block, skipping the `JITCompiler` function entirely. After the `WriteLine` method executes it returns to `Main`. The figure below shows what the situation looks like when `WriteLine` is called the second time:

Architecture of the .NET Framework



The important thing to note is that the process incurs a performance hit only the first time a method is called. All subsequent calls to the method execute at the full speed of the native code: Verification and compilation to native code is not performed again.

Note that the JIT compiler stores the native CPU instructions in dynamic memory: The compiled code is discarded when the application terminates. So if you run the application in the future or if you run two instances of the application simultaneously (in two different operating system processes), then the JIT compiler will have to compile the IL to native instructions again.

For most applications, the performance hit incurred by JIT compilation is not significant. Most applications tend to call the same methods over and over again, so these methods will take the performance hit only once while the application executes. Also, a process normally spends more time inside the method than it spends calling the method.

Be aware that the JIT compiler optimizes the native code just like the back end of an unmanaged C++ compiler. Again, the optimization may take time, but the code will execute with much better performance than unoptimized code.

Developers coming from an unmanaged C or C++ background are 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 our source code, doing as much work as possible in producing IL. But then, in order to actually execute the code, the IL itself must be compiled into native CPU instructions at run time, requiring that more memory be allocated, and requiring addition CPU time to do the work.

Believe me, I approached the CLR from a C/C++ background myself, so I was quite skeptical and concerned about this additional overhead. The truth is that this second compilation stage that occurs at runtime does hurt performance and it does allocate dynamic memory. However, Microsoft has done a lot of performance work to keep this additional overhead to a minimum.

Applied .NET Framework Programming

If you too are skeptical, then you should certainly build some applications to test the performance for yourself. In addition, you should run some non-trivial managed applications produced by Microsoft or others and measure their performance. I think you'll be surprised at how good the performance actually is.

In fact, hard as this might be to believe, many people (including me) think that managed applications could actually out-perform unmanaged applications. For example, when the JIT compiler compiles the IL code into native code at runtime, the compiler knows more about the execution environment than an unmanaged compiler would know. Here are some ways that managed code could out-perform unmanaged code:

- A JIT compiler could detect that the application is running on a Pentium 4 and produce native code that takes advantage of any special instructions offered by the Pentium 4. Usually, unmanaged applications are compiled for the lowest common denominator CPU and avoid using special instructions that would give the application a performance boost on newer CPUs.
- A JIT compiler could detect that a certain test is always false on the current host machine. For example, a method with code like this:

```
if (numberOfCPUs > 1) {  
    ...  
}
```

could cause the JIT compiler to not generate any CPU instructions for the above code if the host machine has only 1 CPU. In this case, the native code has been fine-tuned for the host machine: the code is smaller and executes faster.

- The CLR could profile the code execution and recompile the IL into native code while the application runs. The recompiled code could be reorganized to reduce incorrect branch predictions depending on the observed execution patterns.

For these and many other reasons, you should expect future accomplishments with managed code to execute better than today's unmanaged code. As I said, the performance today is quite good for most applications, and it promises to improve as time goes on.

If your experiments show that the CLR's JIT compiler does not offer your application the kind of performance it requires, then you might want to take advantage of the NGen.exe tool that ships with the .NET Framework SDK. This tool compiles all the IL code for an assembly into native code and saves the resulting native code to a file on disk. At runtime, when an assembly is loaded, the CLR automatically checks to see whether a precompiled version of the assembly also exists, and if it does, the CLR loads the precompiled code so that no compilation at runtime is required.

IL and Verification

IL is stack-based, which means that all its instructions push operands onto an execution stack and pop results off the stack. Accordingly, IL offers no instructions to manipulate registers. Compiler developers can easily produce IL code: They don't have to think about managing registers, and there are fewer IL instructions (since none exist for manipulating registers).

And here's another simplification: IL instructions are typeless. For example, IL offers an add instruction that adds the last two operands pushed on the stack. IL does not offer a 32-bit add instruction and a 64-bit add instruction. When the add instruction executes, it determines the types of the operands on the stack and performs the appropriate operation.

In my opinion, the biggest benefit of IL is not that it abstracts away the underlying CPU. The biggest benefit is application robustness. (While compiling IL into native CPU instructions, the CLR performs a process called *verification*. Verification examines the high-level IL code and ensures that everything it does is “safe.”) For example, verification checks that no memory is read from without having previously been written to, that every method is called with the correct number of parameters, that each parameter is of the correct type, that every method’s return value is used properly, that every method has a return statement, and so on.

The metadata for each managed module includes all the method and type information used by the verification process. If the IL code is determined to be “unsafe,” then a **System.Security.VerifierException** exception is thrown, preventing the method from executing.

By default, Microsoft’s C# and Visual Basic.NET compilers produce “safe” code. (Safe code is code that is verifiably safe.) However, using C#’s **unsafe** keyword or using other languages (such as C++ with Managed Extensions or IL Assembly language), you can produce code that cannot be verifiably safe. That is, the code might, in fact, be safe, but the verification is unable to prove it.

To ensure that all methods in your managed module contain verifiably safe IL, you can use the **PEVerify.exe** utility that ships with the .NET Framework SDK. When Microsoft developers test their C# and Visual Basic.NET compilers, they run the resulting module through PEVerify to ensure that the compiler always produces verifiably safe code. If PEVerify detects unsafe code, then Microsoft fixes the compiler.

You may want to consider running PEVerify on your own modules before you package and ship them. If PEVerify detects a problem, then there is a bug in the compiler and you should report this to Microsoft (or whatever company produces the compiler you’re using). If PEVerify doesn’t detect any unverifiable code, then you know that your code will run without throwing a **VerifierException** on the end-user’s machine.

Note that an administrator can elect to turn verification off (using the “.NET Management” Microsoft Management Console Snap-In). With verification off, the JIT compiler will compile unverifiable IL into native CPU instructions; however, the administrator is taking full responsibility for the code’s behavior.

In Windows, each process has its own virtual address space. Separate address spaces are necessary because Windows can’t trust the application code. It is entirely possible (and unfortunately, all too common) that an application will read from or write to an invalid memory address. Placing each Windows process in a separate address space enhances robustness: One process cannot adversely affect another process.

However, by verifying the managed code, we know that the code does not improperly access memory and cannot adversely affect another application’s code. This means that we can run multiple managed applications in a single Windows virtual address space.

Because Windows processes require a lot of operating system resources, launching many processes can hurt performance and limit available OS resources. Running multiple applications in a single OS process reduces the number of processes, which can improve performance, require fewer resources, and offer equivalent robustness. This is another benefit of managed code as compared to unmanaged code.

The CLR does, in fact, offer the ability to execute multiple managed applications in a single OS process. Each managed application is called an *AppDomain*. By default, every managed EXE will run in its own separate address space that has

Applied .NET Framework Programming

just the one AppDomain. However, a process hosting the CLR (such as IIS or a future version of SQL Server) can decide to run AppDomains in a single OS process.

The .NET Framework Class Library

Included with the .NET Framework is a set of Framework Class Library (FCL) assemblies that contains several thousand type definitions, where each type exposes some functionality. All in all, the CLR and the FCL allow developers to build the following kinds of applications:

- **Web Services.** Components that can be accessed over the Internet very easily. Web services are, of course, the main thrust of Microsoft's .NET initiative.
- **Web Forms.** HTML-based applications (web sites). Typically, web form applications make database queries and web service calls, combine and filter the returned information, and then present that information in a browser using a rich HTML-based UI. Web forms provide a Visual Basic 6- and InterDev-like development environment for web applications written in any CLR language.
- **Windows Forms.** Rich Windows GUI applications. Instead of using a web form to create your application's UI, you can use the more powerful, higher-performance functionality offered by the Windows desktop. Windows form applications can take advantage of controls, menus, mouse and keyboard events, and can talk directly to the underlying operating system. Like web form applications, Windows form applications make database queries and call web services. Windows Forms provides a Visual Basic 6-like development environment for GUI applications written in any CLR language.
- **Windows Console Applications.** For applications with very simple UI demands, a console application provides a quick and easy solution. Compilers, utilities, and tools are typically implemented as console applications.
- **Windows Services.** Yes, it is possible to build service applications controllable via the Windows Service Control Manager (SCM) using the .NET Framework.
- **Component Library.** Of course, the .NET Framework allows you to build stand-alone components (types) that may be easily incorporated into any of the above mentioned application types.

Since the FCL contains literally thousands of types, a set of related types is presented to the developer within a single *namespace*. For example, the System namespace (which you should become most familiar with) contains the Object base type, from which all other types ultimately derive. In addition, the System namespace contains types for integers, characters, strings, exception handling, and console I/O, as well as a bunch of utility types that convert safely between data types, format data types, generate random numbers, and perform various math functions. All applications use types from the System namespace.

To access any platform feature, you need to know which namespace contains the types that expose the facility you're after. If you want to customize the behavior of any type, you can simply derive your own type from the desired FCL type. The .NET Framework relies on the object-oriented nature of the platform to present a consistent programming paradigm to software developers. It also enables developers to create their own namespaces containing their own types, which merge seamlessly into the programming paradigm. Compared to Win32 programming paradigms, this greatly simplifies software development.

Most of the namespaces in the FCL present types that you can use for any kind of application. The table below lists some of the more general namespaces, with a brief description of what the types in that namespace are used for:

Namespace	Purpose of Types
System	All the basic types used by every application.
System.Collections	Managing collections of objects. Includes the popular collection types such as Stacks, Queues, Hashtables, and so on.
System.Diagnostics	Instrumenting and debugging your application.
System.Drawing	Manipulating 2D graphics. Typically used for Windows Forms applications and for creating images that are to appear in a web form.
System.EnterpriseServices	Managing transactions, queued components, object pooling, just-in-time activation, security, and other features to make the use of managed code more efficient on the server.
System.Globalization	National Language Support (NLS), such as string compares, formatting, and calendars.
System.IO	Doing stream I/O, walking directories and files.
System.Management	Managing other computers in the enterprise via WMI.
System.Net	Network communications.
System.Reflection	Inspecting metadata and late binding to types and their members.
System.Resources	Manipulating external data resources.
System.Runtime.InteropServices	Enabling managed code to access unmanaged OS platform facilities, such as COM components and functions in Win32 DLLs.
System.Runtime.Remoting	Accessing types remotely.
System.Runtime.Serialization	Enabling instances of objects to be persisted and regenerated from a stream.
System.Security	Protecting data and resources.
System.Text	Working with text in different encodings, like ASCII or Unicode.
System.Threading	Performing asynchronous operations and synchronizing access to resources.
System.Xml	Processing XML schemas and data.

This book is about the CLR and about the general types that interact closely with the CLR (which would include most of the namespaces listed above). This means that the content of this book is applicable to all .NET Framework programmers regardless of the type of application they're building.

You should be aware, however, that in addition to supplying the more general namespaces, the FCL offers namespaces whose types are used for building specific application types. The table below lists some of the application-specific namespaces:

Namespace	Purpose of Types
System.Web.Services	Building web services.
System.Web.UI	Building web forms.
System.Windows.Forms	Building Windows GUI applications.
System.ServiceProcess	Building a Windows service controllable by the Service Control Manager.

I expect many good books will be published that explain how to build specific application types (such as Windows services, web forms, or Windows forms). These books will give you an excellent start at building your application. I tend to think of these application-specific books as helping you learn from the top down because they concentrate on the application type and not on the development platform. It is my intent that this book offer information that will help you learn from the bottom up. The two types of books should complement each other: After reading this book and an

application-specific book, you should be able to easily and proficiently build any kind of .NET Framework application you desire.

The Common Type System

By now, it should be obvious to you that the CLR is all about types. Types expose functionality to your applications and components. Types are the mechanism by which code written in one programming language can talk to code written in a different programming language. Because types are at the root of the CLR, Microsoft created a formal specification – the common type system (CTS)—that describes how types are defined and behave.

The CTS specification states that a type may contain zero or more members. In Chapter 7, “Type Member Accessibility,” I’ll discuss all these members in great detail. For now, I just want to give you a brief introduction to them:

- **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, often 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.
- **Property.** To the caller, this member looks like a field. But to the type implementer, this member looks like a method (or two). Properties allow an implementer to validate input parameters and object state before accessing the value and to calculate a value only when necessary; they also allow a user of the type to have simplified syntax. Finally, properties also allow you to create read-only or write-only “fields.”
- **Event.** 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. For example, marking a type as **public** exports the type, making it visible and accessible to any assembly. On the other hand, marking a type as **assembly** (called **internal** in C#) makes the type visible and accessible to code within the same assembly only. Thus, the CTS establishes the rules by which assemblies form a boundary of visibility for a type, and the 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. The following list shows the valid options for controlling access to a method or field:

- **Private.** Callable only by other methods in the same class type.
- **Family.** Callable by derived types, regardless of whether they are within the same assembly. Note that many languages (like C++ and C#) refer to **family** as **protected**.
- **Family and Assembly.** Callable by derived types, but only if the derived type is defined in the same assembly.
- **Assembly.** Callable by any code in the same assembly. Note that many languages refer to **assembly** as **internal**.
- **Family or Assembly.** Callable by derived types in any assembly and by any types in the same assembly. Note that C# refers to **family or assembly** as **protected internal**.

- **Public.** Callable by any code in any assembly.

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 modern programming languages. In fact, you won't even need to learn the CTS rules, per se, because the language you use exposes its own language syntax and type rules in the same way you are familiar with today; it maps the language-specific syntax into the "language" of the CLR when it emits the managed module.

When I first started working with the CLR, I soon realized that it is best to think of the language and the behavior of your code as two separate and distinct things. Using C++, you can define your own types with their own members. Of course, you could have used C# or Visual Basic.NET to define the same type with the same members. Sure, the syntax you use for defining this type is different depending on the language you choose, but the behavior of the type will be absolutely identical regardless of the language because the CLR—by means of the CTS—defines the behavior of the type.

To help make this clear, let me give you an example: The CTS supports single inheritance only. The C++ language supports types that inherit from multiple base types; nevertheless, the CTS cannot accept and operate on any such type. To help you, the Visual C++ compiler reports an error if it detects that you're attempting to create managed code that includes a type inherited from multiple base types.

Here's another CTS rule: All types must (ultimately) inherit from a predefined type, **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 types and therefore guarantees every type instance has a minimum set of behaviors. Specifically, the **System.Object** type allows you to:

- compare two instances for equality
- obtain a hash code for the instance
- query the true type of an instance
- perform a shallow (bitwise) copy of the instance
- obtain a string representation of the instance's object's current state

The Common Language Specification

COM allows objects created in different languages to communicate with one another. The CLR goes further. It integrates all languages to let objects created in one language be treated as equal citizens by code written in a completely different language. To make this possible, the CLR defines a standard behavior for types, embeds self-describing type information (metadata), and provides a common execution environment.

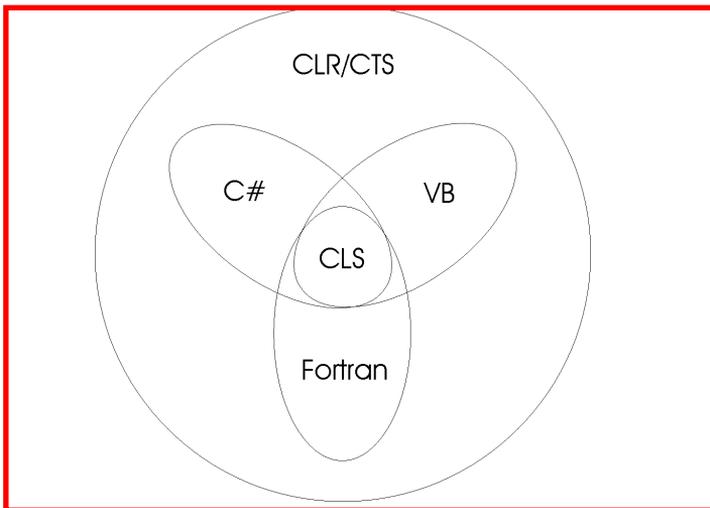
Language integration is a fantastic goal, of course, but the truth of the matter is that programming languages are very different from one another. For example, some languages lack features commonly used in other languages:

- case-sensitivity
- unsigned integers
- operator overloading
- methods that support a variable number of parameters

Applied .NET Framework Programming

If you intend to create types that are easily accessible from other programming languages, then it is important that you use only 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 details for compiler vendors the minimum set of features that their compilers must support if they are to target the runtime.

Note that the CLR/CTS supports a lot more features than the subset defined by the common language specification, so if you don't care about language interoperability, you can develop very rich types limited only by the capabilities of the language. Specifically, the CTS defines rules to which externally visible types and methods must adhere if they are to be accessible from any CLR-compliant programming language. Note that the CLS rules do not apply to code that is only accessible within the defining assembly. The figure below summarizes the way in which language features overlap with the CLS, within the wider context of the CLR/CTS.



As this figure shows, the CLR/CTS offers a broadly inclusive set of features. Some languages expose a large subset of the CLR/CTS; in fact, a programmer willing to write in IL assembly language is able to use all the features offered by the CLR/CTS. Most other languages—such as C#, VB, and Fortran—expose a subset of the CLR/CTS features to the programmer. The CLS defines a minimum set of features that all languages must support.

If you are designing a type in one language and you expect that type to be used by another language, then you should not take advantage of any features that are outside of the CLS. Doing so means that your type members might not be accessible by programmers writing code in other programming languages.

In the code below, a CLS-compliant type is being defined in C#. However, the type has a few non-CLS-compliant constructs that cause the C# compiler to complain about the code:

```
using System;

// Tell compiler to check for CLS Compliance
[assembly:CLSCompliant(true)]

// Errors appear because the class is public
public class App {

    // Error: Return type of 'App.Abc()' is not CLS-compliant
```

```

public UInt32 Abc() { return 0; }

// Error: Identifier 'App.abc()' differing
// only in case is not CLS-compliant
public void abc() { }

// No error: Method is private
private UInt32 ABC() { return 0; }
}

```

In the code above, the `[assembly:CLSCompliant(true)]` attribute is applied to the assembly. This attribute tells the compiler to ensure that any publicly exposed type has no construct that would prevent the type from being accessed from any other programming language. When the code above is compiled, the C# compiler emits two errors. The first error is reported because the method `Abc` returns an unsigned integer; Visual Basic.NET and some other languages cannot manipulate unsigned integer values. The second error arises because this type exposes two public methods which differ only by case: `Abc` and `abc`. Visual Basic.NET and some other languages cannot call both of these methods.

Note that if you were to delete `public` from in front of `class App` and recompile, both errors would go away. The reason is that the `App` type would default to `internal` and would therefore no longer be exposed outside the assembly. For a complete list of CLS rules, refer to the "Cross-Language Interoperability" section in the .NET Framework SDK documentation.

Let me distill the CLS rules to something very simple. In the CLR, every member of a type is either a field (data) or a method (behavior). This means that every programming language must be able to access fields and call methods. Certain fields and certain methods are used in special and common ways. To make coding these common programming patterns easier, languages typically offer additional abstractions. For example, languages expose concepts such as enums, arrays, properties, indexers, delegates, events, constructors, destructors, operator overloads, and conversion operators. When compilers come across any of these constructs in your source code, the compiler must translate them into fields and methods so that the CLR (and any other programming language) can access the construct.

Consider the following type definition that contains a constructor, a destructor, some overloaded operators, a property, an indexer, and an event. Note that additional code is introduced to make the code compile; the code does not illustrate the correct way to implement a type.

```

using System;

class Test {
    // Constructor
    public Test() {}

    // Destructor
    ~Test() {}

    // Operator overload
    public static Boolean operator == (Test t1, Test t2) {
        return true;
    }
    public static Boolean operator != (Test t1, Test t2) {
        return false;
    }
}

```

Applied .NET Framework Programming

```
// An operator overload
public static Test operator + (Test t1, Test t2) { return null; }

// A Property
public String AProperty {
    get { return null; }
    set { }
}

// An Indexer
public String this[Int32 x] {
    get { return null; }
    set { }
}

// An Event
event EventHandler AnEvent;
}
```

When the compiler compiles this code, the result is a type that has a number of fields and methods defined in it. You can easily see this using the IL disassembler tool (ILDasm.exe) provided with the .NET Framework SDK to examine the resulting managed module:



The table on the next page shows how the programming language constructs got mapped to the equivalent CLR fields and methods:

Type Member	Member Type	Equivalent Programming Language Construct
AnEvent	Field	Event. The name of the field is AnEvent and its type is System.EventHandler
.ctor	Method	Constructor
Finalize	Method	Destructor
add_AnEvent	Method	Event add accessor method
get_AProperty	Method	Property get accessor method
get_Item	Method	Indexer get accessor method
op_Addition	Method	+ operator
op_Equality	Method	== operator
op_Inequality	Method	!= operator
remove_AnEvent	Method	Event remove accessor method
set_AProperty	Method	Property set accessor method
set_Item	Method	Indexer set accessor method

The additional nodes under the **Test** type that are not mentioned in the table above—.class, .custom, AnEvent, AProperty, and Item—identify additional metadata about the type. These nodes do not map to fields or methods; they just offer some additional information about the type that the CLR, programming languages, or tools can get access to. For example, a tool can see that the **Test** type offers an event, called **AnEvent**, which is exposed via the two methods (**add_AnEvent** and **remove_AnEvent**).

Interoperability with Unmanaged Code

The .NET Framework offers a ton of advantages over other development platforms. However, very few companies can afford to redesign and re-implement all their existing code. Microsoft realizes this and has built the CLR so that it offers mechanisms that allow an application to consist of both managed and unmanaged parts. Specifically, the CLR supports three interoperability scenarios:

- **Managed code can call an unmanaged function in a DLL.** Managed code can easily call functions contained in DLLs using a mechanism called P/Invoke (for Platform Invoke). After all, many of the types defined in the FCL internally call functions exported from Kernel32.dll, User32.dll, and so on. Many programming languages will expose a mechanism that makes it easy for managed code to call out to unmanaged functions contained in DLLs.
Example: A C# or VB application can call the CreateSemaphore function exported from Kernel32.dll.
- **Managed code can use an existing COM component (server).** Many companies have already implemented a number of unmanaged COM components. Using the type library from these components, a managed assembly can be created that describes the COM component. Managed code can access the type in the managed assembly just like any other managed type. See the TlbImp.exe tool that ships with the .NET Framework SDK for more information. If you do not have a type library or you want to have more control over what TlbImp.exe produces, you can manually build a type in source code that the CLR can use for doing the proper interop.
Example: Using DirectX COM components from a C# or VB application.

Applied .NET Framework Programming

- **Unmanaged code can use a managed type (server).** A lot of existing unmanaged code requires that you supply a COM component for the code to work correctly. You can implement these components easily using managed code, avoiding all the code that has to do with reference counting and interfaces. See the TlbExp.exe and RegAsm.exe tools that ship with the .NET Framework SDK for more information. Example: Creating an ActiveX control or a shell extension in C# or VB.

In addition to the above, the Microsoft Visual C++ compiler (version 13) supports a new /clr command-line switch. This switch tells the compiler to emit IL code instead of native x86 instructions. If you have a large amount of existing C++ code, you can recompile the code using this new compiler switch. The new code will require the CLR to execute, and you can modify the code over time to take advantage of the CLR-specific features.

Note that the /clr switch cannot compile to IL any method that does one of the following:

- contains inline assembly language (via the `__asm` keyword),
- accepts a variable number of arguments,
- calls `setjmp`,
- contains intrinsic routines (such as `__enable`, `__disable`, `_ReturnAddress`, and `_AddressOfReturnAddress`).

For a complete list of the constructs that the C++ compiler cannot compile into IL, see the documentation for the Visual C++ compiler. When the compiler can't compile the method into IL, it compiles the method into x86 so that the application still runs.

Note further that although the IL code produced is managed, the data is not managed. That is, data objects are not allocated from the managed heap, and they are not garbage collected. In fact, the data types do not have metadata produced for them and the type method names are mangled.

The C code below calls the standard C runtime library `printf` function and also calls the `WriteLine` method in `System.Console`. Note that the `System.Console` type is defined in the Framework Class Library. Accordingly, C/C++ code can use libraries available to C/C++ as well as managed types.

```
#include <stdio.h>           // For printf

#using <microsoft.dll>      // For managed types defined in this assembly
using namespace System;    // Easily access System namespace types

// Implement a normal C/C++ main function
void main () {

    // Call the C-Runtime library's printf function
    printf("Displayed by printf.\r\n");

    // Call System.Console's WriteLine method
    Console::WriteLine("Displayed by Console::WriteLine.");
}
```

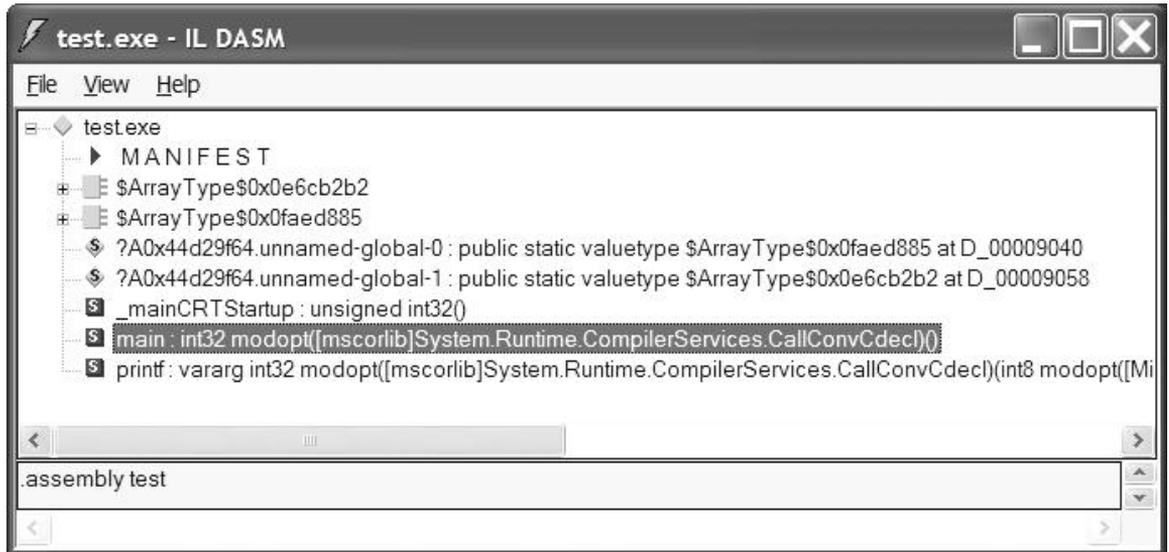
Compiling this code couldn't be easier. If the code above were placed in a `Test.cpp` file, you'd compile it by executing the following line at the command prompt:

```
cl /clr Test.cpp
```

The result is a Test.exe assembly file. If you run Test.exe, you'll see the following output:

```
C:\>Test
Displayed by printf.
Displayed by Console::WriteLine.
```

If you use ILDasm to examine this file, you'll see the following:



Here, ILDasm shows all the global functions and global fields defined within the assembly. Obviously, the compiler has generated a lot of stuff automatically. If you double-click the **main** method, ILDasm will show you the IL code:

```
.method public static int32
    modopt([mscorlib]System.Runtime.CompilerServices.CallConvCdecl)
    main() cil managed
{
    .vtable 1 : 1
    // Code size      28 (0x1c)
    .maxstack 1
    IL_0000: ldsflda    valuetype
                $ArrayType$0x0faed885 '?A0x44d29f64.unnamed-global-0'
    IL_0005: call      vararg int32
                modopt([mscorlib]System.Runtime.CompilerServices.CallConvCdecl)
                printf(int8
                modopt([Microsoft.VisualBasic]Microsoft.VisualBasic.NoSignSpecifiedModifier)
                modopt([Microsoft.VisualBasic]Microsoft.VisualBasic.IsConstModifier)*)
    IL_000a: pop
    IL_000b: ldsflda    valuetype
                $ArrayType$0x0e6cb2b2 '?A0x44d29f64.unnamed-global-1'
    IL_0010: newobj
                instance void [mscorlib]System.String::.ctor(int8*)
    IL_0015: call      void [mscorlib]System.Console::WriteLine(string)
    IL_001a: ldc.i4.0
```

Applied .NET Framework Programming

```
    IL_001b: ret  
} // end of method 'Global Functions'::main
```

What we see here isn't pretty because the compiler generates a lot of special code to make all this work. You can see, however, that the IL above makes calls to both **printf** and the **WriteLine** method in **Console**.

