**What is the .NET Framework?**

The .NET Framework defines an environment that supports the development and execution of highly distributed, component-based applications. It enables differing computer languages to work together and provides for security, program portability, and a common programming model for the Windows platform. As it relates to C#, the .NET Framework defines two very important entities. The first is the Common Language Runtime (CLR). This is the system that manages the execution of your program. Along with other benefits, the Common Language Runtime is the part of the .NET Framework that enables programs to be portable, supports mixed-language programming, and provides for secure execution.

The second entity is the .NET class library. This library gives your program access to the runtime environment. For example, if you want to perform I/O, such as displaying something on the screen, you will use the .NET class library to do it. If you are new to programming, then the term class may be new. Although it is explained in detail later in this book, for now a brief definition will suffice: a class is an object-oriented construct that helps organize programs. As long as your program restricts itself to the features defined by the .NET class library, your programs can run anywhere that the .NET runtime system is supported. Since C# automatically uses the .NET Framework class library, C# programs are automatically portable to all .NET environments.

**How the Common Language Runtime Works**

The Common Language Runtime manages the execution of .NET code. Here is how it works: When you compile a C# program, the output of the compiler is not executable code. Instead, it is a file that contains a special type of pseudocode called Microsoft Intermediate Language (MSIL). MSIL defines a set of portable instructions that are independent of any specific CPU. In essence, MSIL defines a portable assembly language. One other point: although MSIL is similar in concept to Java’s bytecode, the two are not the same.

It is the job of the CLR to translate the intermediate code into executable code when a program is run. Thus, any program compiled to MSIL can be run in any environment for which the CLR is implemented. This is part of how the .NET Framework achieves portability.

Microsoft Intermediate Language is turned into executable code using a JIT compiler. “JIT” stands for “Just-In-Time.” The process works like this: When a .NET program is executed, the CLR activates the JIT compiler. The JIT compiler converts MSIL into native code on demand as each part of your program is needed. Thus, your C# program actually executes as native code even though it is initially compiled into MSIL. This means that your program runs nearly as fast as it would if it had been compiled to native code in the first place, but it gains the portability benefits of MSIL. Also, during compilation, code verification takes place to ensure type safety (unless a security policy has been established that avoids this step).

In addition to MSIL, one other thing is output when you compile a C# program: metadata. Metadata describes the data used by your program and enables your code to interact easily with other code. The metadata is contained in the same file as the MSIL.

An application stays in MSIL form until it's executed, at which time it is just-in-time (JIT) compiled into native code. **Figure A** illustrates this process.

|  |
| --- |
| **Figure A** |
| Description: http://www.techrepublic.com/i/tr/cms/contentPics/u00320020405adm01_01.gif |
| .NET's compilation process, from source to native instructions |

JIT compilation occurs at the assembly level whenever an assembly is first loaded. When a reference to an object is first encountered, the JITer loads a stub for each method that matches that method's declaration. When a method is later invoked, the IL for it is compiled and the stub is replaced with the address of the method's compiled code. This happens each time a method is invoked for the first time, and the resulting native code is cached so that it can be reused the next time the assembly is loaded during that session. Obviously, this execution system results in more required overhead than that for a traditional compiled language, but not as much as you'd think.  
  
That should clear up one common misconception: that .NET applications are interpreted. Another common misconception is that the JIT compiled code is stored on disk and reused for subsequent executions of the same application. While it's possible to do so, as you'll see shortly, this is not the default arrangement. The IL code for an application is recompiled into native code each time that application is run.

A tale of two compilers  
There are, in actuality, two different flavors of JIT compilers (the economy compiler and the normal compiler), and they are not created equal. The economy JITer represents the bare minimum functionality needed to run a .NET application, it directly replaces each MSIL instruction with equivalent native code, doing no optimization, thereby consuming less overhead. It's meant for use on platforms where memory resources are at a premium.  
  
On the other hand, the normal JITer, which is the default runtime configuration, can perform quite a few on-the-fly optimizations to the code it produces. This gives .NET an advantage over a traditional precompiled language, which can't make anything but fairly gross assumptions about the platform its emitted code will be run on. The JITer can adjust to the exact current runtime situation, allowing it to do some things that precompiled languages cannot:

* Utilize and allocate CPU registers more efficiently
* Perform low-level code optimizations when appropriate, such as constant folding, copy propagation, elimination of range checking, elimination of common subexpressions, and method inlining
* Utilize memory more efficiently by monitoring the current demand for physical and virtual memory during execution
* Take advantage of the exact processor model in use by emitting instructions specifically for it

The .NET result, if you'll pardon the pun, is that the extra overhead required for the JITer doesn't exact as much of a performance penalty as you'd expect.  
  
An option for the speed demon in you  
Okay, so MSIL is JIT-compiled every time an application is started. It's common sense, then, to suppose that initial startup times, as well as the first use of noncore functionality, could result in slower-than-optimal performance. What can you do to minimize this hit?  
  
Microsoft provides what is known by the somewhat redundant name of a Pre-JIT compiler (otherwise known as the Native Image Generator, hence the name [Ngen.exe](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cptools/html/cpgrfnativeimagegeneratorngenexe.asp)). On the surface, at least, it offers a remedy for any performance problems. The Pre-JIT compiler is meant to be invoked before runtime, like at install time, and it compiles all MSIL in an assembly into native code. This native code is then stored in a special part of the Global Assembly Cache for later use, bypassing the JIT compilation process altogether.  
  
At first glance, this sure sounds like a winner, particularly for client-side code. But recall that the normal JIT performs lots of on-the-fly optimizations while compiling MSIL. Many of these optimizations, particularly those involving the use of registers and memory, are driven by the current demands made on the system. Compiling assemblies in one large batch prevents these optimizations from being made and therefore may actually result in slower final code. Before you go this route, Microsoft recommends that you do your homework and profile both JIT and Ngen versions of the same assembly on the target platform under conditions approximating those found under normal use.

**Steps in Compiling Source Code**

