.NET Framework 4

**Compiling MSIL to Native Code**

[This documentation is for preview only, and is subject to change in later releases. Blank topics are included as placeholders.]

Before you can run Microsoft intermediate language (MSIL), it must be compiled against the common language runtime to native code for the target machine architecture. The .NET Framework provides two ways to perform this conversion:

* A .NET Framework just-in-time (JIT) compiler.
* The .NET Framework [Ngen.exe (Native Image Generator)](http://msdn.microsoft.com/en-us/library/6t9t5wcf.aspx).

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifCompilation by the Just-in-time Compiler

JIT compilation converts MSIL to native code on demand at application run time, when the contents of an assembly are loaded and executed. Because the common language runtime supplies a JIT compiler for each supported CPU architecture, developers can build a set of MSIL assemblies that can be JIT-compiled and run on different computers with different machine architectures. However, your managed code will run only on a specific operating system if it calls platform-specific native APIs, or a platform-specific class library.

JIT compilation takes into account the fact that some code might never get called during execution. Rather than using time and memory to convert all the MSIL in a portable executable (PE) file to native code, it converts the MSIL as needed during execution and stores the resulting native code in memory so that it is accessible for subsequent calls in the context of that process. The loader creates and attaches a stub to each method in a type when the type is loaded and initialized. When a method is called for the first time, the stub passes control to the JIT compiler, which converts the MSIL for that method into native code and modifies the stub to point directly to the generated native code. Subsequent calls to the JIT-compiled method therefore proceed directly to the native code.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifInstall-Time Code Generation Using NGen.exe

Because the JIT compiler converts an assembly's MSIL to native code when individual methods defined in that assembly are called, it necessarily involves a performance hit at run time. In most cases, that performance hit is acceptable. More importantly, the code generated by the JIT compiler is bound to the process that triggered the compilation. It cannot be shared across multiple processes. To allow the generated code to be shared across multiple invocations of an application or across multiple processes that share a set of assemblies, the common language runtime supports an ahead-of-time compilation mode. This ahead-of-time compilation mode uses the [Ngen.exe (Native Image Generator)](http://msdn.microsoft.com/en-us/library/6t9t5wcf.aspx) to convert MSIL assemblies to native code much like the JIT compiler does. However, the operation of Ngen.exe differs from that of the JIT compiler in three ways:

* It performs the conversion from MSIL to native code before rather than while running the application.
* It compiles an entire assembly at a time, rather than a method at a time.
* It persists the generated code in the Native Image Cache as a file on disk.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifCode Verification

As part of compiling MSIL to native code, the MSIL code must pass a verification process unless an administrator has established a security policy that allows the code to bypass verification. Verification examines MSIL and metadata to find out whether the code is type safe, which means that it only accesses the memory locations it is authorized to access. Type safety helps isolate objects from each other and therefore helps protect them from inadvertent or malicious corruption. It also provides assurance that security restrictions on code can be reliably enforced.

The runtime relies on the fact that the following statements are true for code that is verifiably type safe:

* A reference to a type is strictly compatible with the type being referenced.
* Only appropriately defined operations are invoked on an object.
* Identities are what they claim to be.

During the verification process, MSIL code is examined in an attempt to confirm that the code can access memory locations and call methods only through properly defined types. For example, code cannot allow an object's fields to be accessed in a manner that allows memory locations to be overrun. Additionally, verification inspects code to determine whether the MSIL has been correctly generated, because incorrect MSIL can lead to a violation of the type safety rules. The verification process passes a well-defined set of type-safe code, and it passes only code that is type safe. However, some type-safe code might not pass verification because of some limitations of the verification process, and some languages, by design, do not produce verifiably type-safe code. If type-safe code is required by the security policy but the code does not pass verification, an exception is thrown when the code is run.