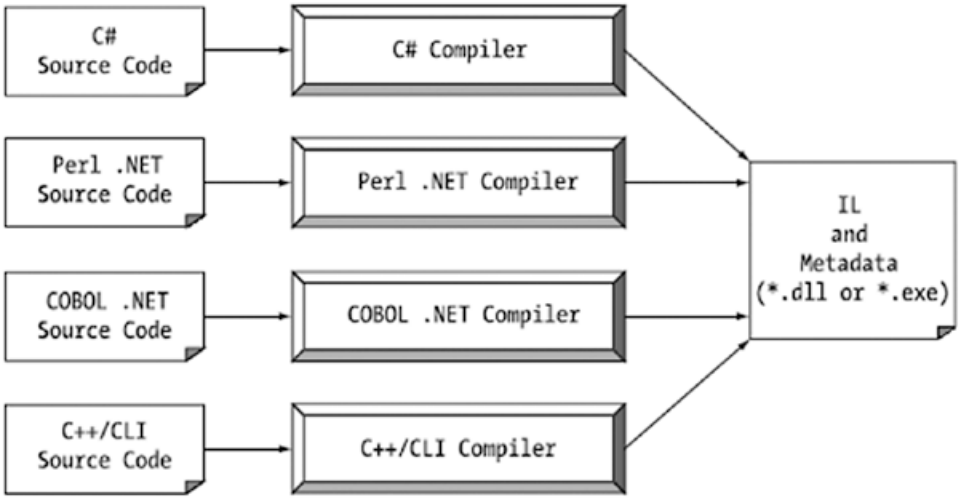
An Overview of .NET Assemblies

Regardless of which .NET language you choose to program with, understand that despite .NET binaries taking the same file extension as unmanaged Windows binaries (*.dll or *.exe), they have absolutely no internal similarities. Specifically, .NET binaries do not contain platform-specific instructions but rather platform-agnostic *Intermediate Language* (*IL*) and type metadata.



IL is also known as Microsoft Intermediate Language (MSIL) or alternatively as the Common Intermediate Language (CIL). Thus, as you read the .NET literature, understand that IL, MSIL, and CIL are all describing essentially the same concept.

When a *.dll or *.exe has been created using a .NET-aware compiler, the binary blob is termed an *assembly*. An **assembly** is a runtime unit consisting of types and other resources. In C#, an assembly is the smallest deployment unit used, and is a component in .NET. As mentioned, an assembly contains CIL code, which is conceptually similar to Java bytecode in that it is not compiled to platform-specific instructions until absolutely necessary. Typically, "absolutely necessary" is the point at which a block of CIL instructions (such as a method implementation) is referenced for use by the .NET runtime.

In addition to CIL instructions, assemblies also contain *metadata* that describes in vivid detail the characteristics of every "type" within the binary. For example, if you have a class named SportsCar, the type metadata describes details such as SportsCar's base class, specifies which interfaces are implemented by SportsCar (if any), and gives full descriptions of each member supported by the SportsCar type. .NET metadata is always present within an assembly and is automatically generated by a .NET-aware language compiler.

Finally, in addition to CIL and type metadata, assemblies themselves are also described using metadata, which is officially termed a *manifest*. The manifest contains information about the current version of the assembly, culture information (used for localizing string and image resources), and a list of all externally referenced assemblies that are required for proper execution.

Let's examine *CIL code*, type *metadata*, and the assembly *manifest* in a bit more detail.

Common Intermediate Language (CIL)

CIL is a language that sits above any particular platform-specific instruction set. After you compile your code file using the C# compiler (csc.exe), you end up with a single-file *.exe assembly that contains a manifest, CIL instructions, and metadata describing each aspect of the code you wrote. Now you might be wondering exactly what is gained by compiling source code into CIL rather than directly to a specific

instruction set. One benefit is language integration: each .NET-aware compiler produces nearly identical CIL instructions. Therefore, all languages are able to interact within a well-defined binary arena. Because assemblies contain CIL instructions rather than platform-specific instructions, CIL code must be compiled on the fly before use. The entity that compiles CIL code into meaningful CPU instructions is a **JIT (just-in-time)** compiler, which sometimes goes by the friendly name of *jitter*.

The .NET runtime environment leverages a JIT compiler for each CPU targeting the runtime, each optimized for the underlying platform. For example, if you are building a .NET application to be deployed to a handheld device (such as a Windows Phone device), the corresponding jitter is well equipped to run within a low-memory environment. On the other hand, if you are deploying your assembly to a back-end company server (where memory is seldom an issue), the jitter will be optimized to function in a high-memory environment. In this way, developers can write a single body of code that can be efficiently JIT compiled and executed on machines with different architectures.

Furthermore, as a given ***jitter compiles CIL instructions into corresponding machine code, it will cache the results in memory in a manner suited to the target operating system***. In this way, if a call is made to a method named PrintDocument(), the CIL instructions are compiled into platform-specific instructions on the first invocation and retained in memory for later use. Therefore, the next time PrintDocument() is called, there is no need to recompile the CIL.

Metada

In addition to CIL instructions, a .NET assembly contains full, complete, and accurate metadata, which describes every type (e.g., class, structure, enumeration) defined in the binary, as well as the members of each type (e.g., properties, methods, events). Thankfully, it is always the job of the compiler (not the programmer) to emit the latest and greatest type metadata. Because .NET metadata is so wickedly meticulous, assemblies are completely self-describing entities.

Metadata is used by numerous aspects of the .NET runtime environment, as well as by various development tools. For example, the IntelliSense feature provided by tools such as Visual Studio is made possible by reading an assembly's metadata at design time. Metadata is also used by various object-browsing utilities, debugging tools, and the C# compiler itself. To be sure, metadata is the backbone of numerous .NET technologies including WCF, reflection, late binding, and object serialization.

Manifest

Last but not least, remember that a .NET assembly also contains metadata that describes the assembly itself (technically termed a *manifest*). Among other details, the manifest documents all external assemblies required by the current assembly to function correctly, the assembly's version number, copyright information, and so forth. Like type metadata, it is always the job of the compiler to generate the assembly's manifest.

The Assembly/Namespace

To keep all the types within the base class libraries well organized, the .NET platform makes extensive use of the *namespace* concept.

A **namespace** is a grouping of semantically related types contained in an assembly or possibly spread across multiple related assemblies. For example, the System.IO namespace contains file I/O–related types, the System.Data namespace defines basic database types, and so on. It is important to point out that *a single assembly (such as mscorlib.dll) can contain any number of namespaces, each of which can contain any number of types.*

Namespace is nothing more than a convenient way for us mere humans to logically understand and organize related types.

The most fundamental namespace to get your head around initially is named System. In fact, you cannot build any sort of functional C# application without at least making a reference to the System namespace, as the core data types (e.g., System.Int32, System.String) are defined here.

However, the .NET base class library defines a number of topmost root namespaces beyond System, the most useful of which is named **Microsoft**. *Any namespace nested within Microsoft (e.g., Microsoft.CSharp, Microsoft.ManagementConsole, Microsoft.Win32) contains types that are used to interact with services unique to the Windows operating system*. Given this point, you should not assume that these types could be used successfully on other .NET-enabled operating systems such as macOS.

You can use namespaces by *using* keyword (ex. using System) or by the *fully qualified name* (ex. System.Windows.Controls.Button).

Many core .NET namespaces are defined within mscorlib.dll, but there are many other types that are defined in the other assembly. A vast majority of the .NET Framework assemblies are located under a specific directory termed the global assembly cache (GAC). On a Windows machine, this can be located by default under C:\Windows\Assembly\GAC. Depending on the development tool you are using to build your .NET applications, you will have various ways to inform the compiler which assemblies you want to include during the compilation cycle.

Just remember that what makes *a namespace unique* is that it contains types that are somehow semantically related.

The Intermediate Language Disassembler utility (ildasm.exe), which ships with the .NET Framework, allows you to load up any .NET assembly and investigate its contents, including the associated manifest, CIL code, and type metadata. This tool allows you to dive deeply into how their C# code maps to CIL and ultimately helps you understand the inner workings of the .NET platform.