**Assemblies in the Common Language Runtime**

**.NET Framework 4.5**

Assemblies are the building blocks of .NET Framework applications; they form the fundamental unit of deployment, version control, reuse, activation scoping, and security permissions. An assembly is a collection of types and resources that are built to work together and form a logical unit of functionality. An assembly provides the common language runtime with the information it needs to be aware of type implementations. To the runtime, a type does not exist outside the context of an assembly.

An assembly performs the following functions:

* It contains code that the common language runtime executes. Microsoft intermediate language (MSIL) code in a portable executable (PE) file will not be executed if it does not have an associated assembly manifest. Note that each assembly can have only one entry point (that is, **DllMain**, **WinMain**, or **Main**).
* It forms a security boundary. An assembly is the unit at which permissions are requested and granted. For more information about security boundaries as they apply to assemblies, see [Assembly Security Considerations](http://msdn.microsoft.com/en-us/library/ab4eace3.aspx).
* It forms a type boundary. Every type's identity includes the name of the assembly in which it resides. A type called MyType that is loaded in the scope of one assembly is not the same as a type called MyType that is loaded in the scope of another assembly.
* It forms a reference scope boundary. The assembly's manifest contains assembly metadata that is used for resolving types and satisfying resource requests. It specifies the types and resources that are exposed outside the assembly. The manifest also enumerates other assemblies on which it depends.
* It forms a version boundary. The assembly is the smallest versionable unit in the common language runtime; all types and resources in the same assembly are versioned as a unit. The assembly's manifest describes the version dependencies you specify for any dependent assemblies. For more information about versioning, see [Assembly Versioning](http://msdn.microsoft.com/en-us/library/51ket42z.aspx).
* It forms a deployment unit. When an application starts, only the assemblies that the application initially calls must be present. Other assemblies, such as localization resources or assemblies containing utility classes, can be retrieved on demand. This allows applications to be kept simple and thin when first downloaded. For more information about deploying assemblies, see [Deploying Applications](http://msdn.microsoft.com/en-us/library/6hbb4k3e.aspx).
* It is the unit at which side-by-side execution is supported. For more information about running multiple versions of an assembly, see [Assemblies and Side-by-Side Execution](http://msdn.microsoft.com/en-us/library/fdhkd3a5.aspx).

Assemblies can be static or dynamic. Static assemblies can include .NET Framework types (interfaces and classes), as well as resources for the assembly (bitmaps, JPEG files, resource files, and so on). Static assemblies are stored on disk in portable executable (PE) files. You can also use the .NET Framework to create dynamic assemblies, which are run directly from memory and are not saved to disk before execution. You can save dynamic assemblies to disk after they have executed.

There are several ways to create assemblies. You can use development tools, such as Visual Studio 2005, that you have used in the past to create .dll or .exe files. You can use tools provided in the Windows Software Development Kit (SDK) to create assemblies with modules created in other development environments. You can also use common language runtime APIs, such as [Reflection.Emit](http://msdn.microsoft.com/en-us/library/system.reflection.emit.aspx), to create dynamic assemblies.

**Assembly Benefits**

**.NET Framework 4.5**

Assemblies are designed to simplify application deployment and to solve versioning problems that can occur with component-based applications.

End users and developers are familiar with versioning and deployment issues that arise from today's component-based systems. Some end users have experienced the frustration of installing a new application on their computer, only to find that an existing application has suddenly stopped working. Many developers have spent countless hours trying to keep all necessary registry entries consistent in order to activate a COM class.

Many deployment problems have been solved by the use of assemblies in the .NET Framework. Because they are self-describing components that have no dependencies on registry entries, assemblies enable zero-impact application installation. They also simplify uninstalling and replicating applications.

[**Versioning Problems**](javascript:void(0))

Currently two versioning problems occur with Win32 applications:

* Versioning rules cannot be expressed between pieces of an application and enforced by the operating system. The current approach relies on backward compatibility, which is often difficult to guarantee. Interface definitions must be static, once published, and a single piece of code must maintain backward compatibility with previous versions. Furthermore, code is typically designed so that only a single version of it can be present and executing on a computer at any given time.
* There is no way to maintain consistency between sets of components that are built together and the set that is present at run time.

These two versioning problems combine to create DLL conflicts, where installing one application can inadvertently break an existing application because a certain software component or DLL was installed that was not fully backward compatible with a previous version. Once this situation occurs, there is no support in the system for diagnosing and fixing the problem.

[**An End to DLL Conflicts**](javascript:void(0))

Microsoft® Windows® 2000 began to fully address these problems. It provides two features that partially fix DLL conflicts:

* Windows 2000 enables you to create client applications where the dependent .dll files are located in the same directory as the application's .exe file. Windows 2000 can be configured to check for a component in the directory where the .exe file is located before checking the fully qualified path or searching the normal path. This enables components to be independent of components installed and used by other applications.
* Windows 2000 locks files that are shipped with the operating system in the System32 directory so they cannot be inadvertently replaced when applications are installed.

The common language runtime uses assemblies to continue this evolution toward a complete solution to DLL conflicts.

[**The Assembly Solution**](javascript:void(0))

To solve versioning problems, as well as the remaining problems that lead to DLL conflicts, the runtime uses assemblies to do the following:

* Enable developers to specify version rules between different software components.
* Provide the infrastructure to enforce versioning rules.
* Provide the infrastructure to allow multiple versions of a component to be run simultaneously (called side-by-side execution).

**Assembly Contents**

**.NET Framework 4.5**

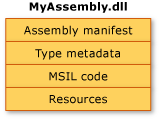
In general, a static assembly can consist of four elements:

* The [assembly manifest](http://msdn.microsoft.com/en-us/library/1w45z383.aspx), which contains assembly metadata.
* Type metadata.
* Microsoft intermediate language (MSIL) code that implements the types.
* A set of resources.

Only the assembly manifest is required, but either types or resources are needed to give the assembly any meaningful functionality.

There are several ways to group these elements in an assembly. You can group all elements in a single physical file, which is shown in the following illustration.

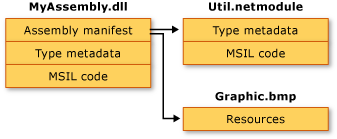
**Single-file assembly**



Alternatively, the elements of an assembly can be contained in several files. These files can be modules of compiled code (.netmodule), resources (such as .bmp or .jpg files), or other files required by the application. Create a multifile assembly when you want to combine modules written in different languages and to optimize downloading an application by putting seldom used types in a module that is downloaded only when needed.

In the following illustration, the developer of a hypothetical application has chosen to separate some utility code into a different module and to keep a large resource file (in this case a .bmp image) in its original file. The .NET Framework downloads a file only when it is referenced; keeping infrequently referenced code in a separate file from the application optimizes code download.

**Multifile assembly**



|  |
| --- |
| **Description: NoteNote** |
| The files that make up a multifile assembly are not physically linked by the file system. Rather, they are linked through the assembly manifest and the common language runtime manages them as a unit. |

In this illustration, all three files belong to an assembly, as described in the assembly manifest contained in MyAssembly.dll. To the file system, they are three separate files. Note that the file Util.netmodule was compiled as a module because it contains no assembly information. When the assembly was created, the assembly manifest was added to MyAssembly.dll, indicating its relationship with Util.netmodule and Graphic.bmp.

As you currently design your source code, you make explicit decisions about how to partition the functionality of your application into one or more files. When designing .NET Framework code, you will make similar decisions about how to partition the functionality into one or more assemblies.

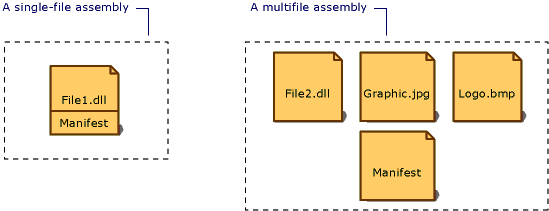
**Assembly Manifest**

**.NET Framework 4.5**

Every assembly, whether static or dynamic, contains a collection of data that describes how the elements in the assembly relate to each other. The assembly manifest contains this assembly metadata. An assembly manifest contains all the metadata needed to specify the assembly's version requirements and security identity, and all metadata needed to define the scope of the assembly and resolve references to resources and classes. The assembly manifest can be stored in either a PE file (an .exe or .dll) with Microsoft intermediate language (MSIL) code or in a standalone PE file that contains only assembly manifest information.

The following illustration shows the different ways the manifest can be stored.

**Types of assemblies**



For an assembly with one associated file, the manifest is incorporated into the PE file to form a single-file assembly. You can create a multifile assembly with a standalone manifest file or with the manifest incorporated into one of the PE files in the assembly.

Each assembly's manifest performs the following functions:

* Enumerates the files that make up the assembly.
* Governs how references to the assembly's types and resources map to the files that contain their declarations and implementations.
* Enumerates other assemblies on which the assembly depends.
* Provides a level of indirection between consumers of the assembly and the assembly's implementation details.
* Renders the assembly self-describing.

[**Assembly Manifest Contents**](javascript:void(0))

The following table shows the information contained in the assembly manifest. The first four items—the assembly name, version number, culture, and strong name information—make up the assembly's identity.

|  |  |
| --- | --- |
| Information | Description |
| Assembly name | A text string specifying the assembly's name. |
| Version number | A major and minor version number, and a revision and build number. The common language runtime uses these numbers to enforce version policy. |
| Culture | Information on the culture or language the assembly supports. This information should be used only to designate an assembly as a satellite assembly containing culture- or language-specific information. (An assembly with culture information is automatically assumed to be a satellite assembly.) |
| Strong name information | The public key from the publisher if the assembly has been given a strong name. |
| List of all files in the assembly | A hash of each file contained in the assembly and a file name. Note that all files that make up the assembly must be in the same directory as the file containing the assembly manifest. |
| Type reference information | Information used by the runtime to map a type reference to the file that contains its declaration and implementation. This is used for types that are exported from the assembly. |
| Information on referenced assemblies | A list of other assemblies that are statically referenced by the assembly. Each reference includes the dependent assembly's name, assembly metadata (version, culture, operating system, and so on), and public key, if the assembly is strong named. |

You can add or change some information in the assembly manifest by using assembly attributes in your code. You can change version information and informational attributes, including Trademark, Copyright, Product, Company, and Informational Version. For a complete list of assembly attributes, see [Setting Assembly Attributes](http://msdn.microsoft.com/en-us/library/4w8c1y2s).

**Global Assembly Cache**

**.NET Framework 4.5**

Each computer where the common language runtime is installed has a machine-wide code cache called the global assembly cache. The global assembly cache stores assemblies specifically designated to be shared by several applications on the computer.

You should share assemblies by installing them into the global assembly cache only when you need to. As a general guideline, keep assembly dependencies private, and locate assemblies in the application directory unless sharing an assembly is explicitly required. In addition, it is not necessary to install assemblies into the global assembly cache to make them accessible to COM interop or unmanaged code.

|  |
| --- |
| **Description: NoteNote** |
| There are scenarios where you explicitly do not want to install an assembly into the global assembly cache. If you place one of the assemblies that make up an application in the global assembly cache, you can no longer replicate or install the application by using the **xcopy** command to copy the application directory. You must move the assembly in the global assembly cache as well. |

There are two ways to deploy an assembly into the global assembly cache:

* Use an installer designed to work with the global assembly cache. This is the preferred option for installing assemblies into the global assembly cache.
* Use a developer tool called the [Global Assembly Cache tool (Gacutil.exe)](http://msdn.microsoft.com/en-us/library/ex0ss12c.aspx), provided by the Windows Software Development Kit (SDK).

|  |
| --- |
| **Description: NoteNote** |
| In deployment scenarios, use Windows Installer 2.0 to install assemblies into the global assembly cache. Use the Global Assembly Cache tool only in development scenarios, because it does not provide assembly reference counting and other features provided when using the Windows Installer. |

Administrators often protect the systemroot directory using an access control list (ACL) to control write and execute access. Because the global assembly cache is installed in a subdirectory of the systemroot directory, it inherits that directory's ACL. It is recommended that only users with Administrator privileges be allowed to delete files from the global assembly cache.

Assemblies deployed in the global assembly cache must have a strong name. When an assembly is added to the global assembly cache, integrity checks are performed on all files that make up the assembly. The cache performs these integrity checks to ensure that an assembly has not been tampered with, for example, when a file has changed but the manifest does not reflect the change.

# Strong-Named Assemblies

**.NET Framework 4.5**

A strong name consists of the assembly's identity—its simple text name, version number, and culture information (if provided)—plus a public key and a digital signature. It is generated from an assembly file (the file that contains the assembly manifest, which in turn contains the names and hashes of all the files that make up the assembly), using the corresponding private key. Microsoft® Visual Studio® .NET and other development tools provided in the Windows Software Development Kit (SDK) can assign strong names to an assembly. Assemblies with the same strong name are expected to be identical.

You can ensure that a name is globally unique by signing an assembly with a strong name. In particular, strong names satisfy the following requirements:

* Strong names guarantee name uniqueness by relying on unique key pairs. No one can generate the same assembly name that you can, because an assembly generated with one private key has a different name than an assembly generated with another private key.
* Strong names protect the version lineage of an assembly. A strong name can ensure that no one can produce a subsequent version of your assembly. Users can be sure that a version of the assembly they are loading comes from the same publisher that created the version the application was built with.
* Strong names provide a strong integrity check. Passing the .NET Framework security checks guarantees that the contents of the assembly have not been changed since it was built. Note, however, that strong names in and of themselves do not imply a level of trust like that provided, for example, by a digital signature and supporting certificate.

When you reference a strong-named assembly, you expect to get certain benefits, such as versioning and naming protection. If the strong-named assembly then references an assembly with a simple name, which does not have these benefits, you lose the benefits you would derive from using a strong-named assembly and revert to DLL conflicts. Therefore, strong-named assemblies can only reference other strong-named assemblies.

# Assembly Security Considerations

**.NET Framework 4.5**

When you build an assembly, you can specify a set of permissions that the assembly requires to run. Whether certain permissions are granted or not granted to an assembly is based on evidence.

There are two distinct ways evidence is used:

* The input evidence is merged with the evidence gathered by the loader to create a final set of evidence used for policy resolution. The methods that use this semantic include **Assembly.Load**, **Assembly.LoadFrom**, and **Activator.CreateInstance**.
* The input evidence is used unaltered as the final set of evidence used for policy resolution. The methods that use this semantic include **Assembly.Load(byte[])** and **AppDomain.DefineDynamicAssembly()**.

Optional permissions can be granted by the [security policy](http://msdn.microsoft.com/en-us/library/33tceax8.aspx) set on the computer where the assembly will run. If you want your code to handle all potential security exceptions, you can do one of the following:

* Insert a permission request for all the permissions your code must have, and handle up front the load-time failure that occurs if the permissions are not granted.
* Do not use a permission request to obtain permissions your code might need, but be prepared to handle security exceptions if permissions are not granted.

|  |
| --- |
| **Description: NoteNote** |
| Security is a complex area, and you have many options to choose from. For more information, see [Key Security Concepts](http://msdn.microsoft.com/en-us/library/z164t8hs.aspx). |

At load time, the assembly's evidence is used as input to security policy. Security policy is established by the enterprise and the computer's administrator as well as by user policy settings, and determines the set of permissions that is granted to all managed code when executed. Security policy can be established for the publisher of the assembly (if it has a signing tool generated signature), for the Web site and zone (in Internet Explorer terms) the assembly was downloaded from, or for the assembly's strong name. For example, a computer's administrator can establish security policy that allows all code downloaded from a Web site and signed by a given software company to access a database on a computer, but does not grant access to write to the computer's disk.

[Strong-Named Assemblies and Signing Tools](javascript:void(0))

You can sign an assembly in two different but complementary ways: with a strong name or using either the [File Signing Tool (Signcode.exe)](http://msdn.microsoft.com/en-us/library/9sh96ycy.aspx) in .NET Framework version 1.0 and 1.1 or [SignTool.exe (Sign Tool)](http://msdn.microsoft.com/en-us/library/8s9b9yaz.aspx) in later versions of the .NET Framework. Signing an assembly with a strong name adds a public key encryption to the file containing the assembly manifest. Strong name signing helps to verify name uniqueness, prevent name spoofing, and provide callers with some identity when a reference is resolved.

However, no level of trust is associated with a strong name, which makes the [File Signing Tool (Signcode.exe)](http://msdn.microsoft.com/en-us/library/9sh96ycy.aspx) and [SignTool.exe (Sign Tool)](http://msdn.microsoft.com/en-us/library/8s9b9yaz.aspx) important. The two signing tools require a publisher to prove its identity to a third-party authority and obtain a certificate. This certificate is then embedded in your file and can be used by an administrator to decide whether to trust the code's authenticity.

You can give both a strong name and a digital signature created using the [File Signing Tool (Signcode.exe)](http://msdn.microsoft.com/en-us/library/9sh96ycy.aspx) or [SignTool.exe (Sign Tool)](http://msdn.microsoft.com/en-us/library/8s9b9yaz.aspx) to an assembly, or you can use either alone. The two signing tools can sign only one file at a time; for a multifile assembly, you sign the file that contains the assembly manifest. A strong name is stored in the file containing the assembly manifest, but a signature created using the [File Signing Tool (Signcode.exe)](http://msdn.microsoft.com/en-us/library/9sh96ycy.aspx) or [SignTool.exe (Sign Tool)](http://msdn.microsoft.com/en-us/library/8s9b9yaz.aspx) is stored in a reserved slot in the portable executable (PE) file containing the assembly manifest. Signing of an assembly using the [File Signing Tool (Signcode.exe)](http://msdn.microsoft.com/en-us/library/9sh96ycy.aspx) or [SignTool.exe (Sign Tool)](http://msdn.microsoft.com/en-us/library/8s9b9yaz.aspx) can be used (with or without a strong name) when you already have a trust hierarchy that relies on [File Signing Tool (Signcode.exe)](http://msdn.microsoft.com/en-us/library/9sh96ycy.aspx) or [SignTool.exe (Sign Tool)](http://msdn.microsoft.com/en-us/library/8s9b9yaz.aspx) generated signatures, or when your policy uses only the key portion and does not check a chain of trust.

|  |
| --- |
| **Description: NoteNote** |
| When using both a strong name and a signing tool signature on an assembly, the strong name must be assigned first. |

The common language runtime also performs a hash verification; the assembly manifest contains a list of all files that make up the assembly, including a hash of each file as it existed when the manifest was built. As each file is loaded, its contents are hashed and compared with the hash value stored in the manifest. If the two hashes do not match, the assembly fails to load.

Because strong naming and signing using the [File Signing Tool (Signcode.exe)](http://msdn.microsoft.com/en-us/library/9sh96ycy.aspx) or [SignTool.exe (Sign Tool)](http://msdn.microsoft.com/en-us/library/8s9b9yaz.aspx) guarantee integrity, you can base code access security policy on these two forms of assembly evidence. Strong naming and signing using the [File Signing Tool (Signcode.exe)](http://msdn.microsoft.com/en-us/library/9sh96ycy.aspx) or [SignTool.exe (Sign Tool)](http://msdn.microsoft.com/en-us/library/8s9b9yaz.aspx) guarantee integrity through digital signatures and certificates. All the technologies mentioned—hash verification, strong naming, and signing using the [File Signing Tool (Signcode.exe)](http://msdn.microsoft.com/en-us/library/9sh96ycy.aspx) or [SignTool.exe (Sign Tool)](http://msdn.microsoft.com/en-us/library/8s9b9yaz.aspx)—work together to ensure that the assembly has not been altered in any way.

# Strong-Named Assemblies

**.NET Framework 4.5**

A strong name consists of the assembly's identity—its simple text name, version number, and culture information (if provided)—plus a public key and a digital signature. It is generated from an assembly file (the file that contains the assembly manifest, which in turn contains the names and hashes of all the files that make up the assembly), using the corresponding private key. Microsoft® Visual Studio® .NET and other development tools provided in the Windows Software Development Kit (SDK) can assign strong names to an assembly. Assemblies with the same strong name are expected to be identical.

You can ensure that a name is globally unique by signing an assembly with a strong name. In particular, strong names satisfy the following requirements:

* Strong names guarantee name uniqueness by relying on unique key pairs. No one can generate the same assembly name that you can, because an assembly generated with one private key has a different name than an assembly generated with another private key.
* Strong names protect the version lineage of an assembly. A strong name can ensure that no one can produce a subsequent version of your assembly. Users can be sure that a version of the assembly they are loading comes from the same publisher that created the version the application was built with.
* Strong names provide a strong integrity check. Passing the .NET Framework security checks guarantees that the contents of the assembly have not been changed since it was built. Note, however, that strong names in and of themselves do not imply a level of trust like that provided, for example, by a digital signature and supporting certificate.

When you reference a strong-named assembly, you expect to get certain benefits, such as versioning and naming protection. If the strong-named assembly then references an assembly with a simple name, which does not have these benefits, you lose the benefits you would derive from using a strong-named assembly and revert to DLL conflicts. Therefore, strong-named assemblies can only reference other strong-named assemblies.

# Assembly Versioning

**.NET Framework 4.5**

All versioning of assemblies that use the common language runtime is done at the assembly level. The specific version of an assembly and the versions of dependent assemblies are recorded in the assembly's manifest. The default version policy for the runtime is that applications run only with the versions they were built and tested with, unless overridden by explicit version policy in configuration files (the application configuration file, the publisher policy file, and the computer's administrator configuration file).

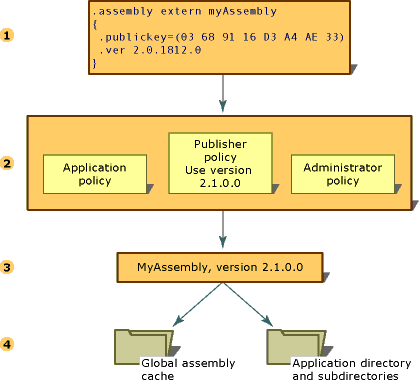
|  |
| --- |
| **Description: NoteNote** |
| Versioning is done only on assemblies with strong names. |

The runtime performs several steps to resolve an assembly binding request:

1. Checks the original assembly reference to determine the version of the assembly to be bound.
2. Checks for all applicable configuration files to apply version policy.
3. Determines the correct assembly from the original assembly reference and any redirection specified in the configuration files, and determines the version that should be bound to the calling assembly.
4. Checks the global assembly cache, codebases specified in configuration files, and then checks the application's directory and subdirectories using the probing rules explained in [How the Runtime Locates Assemblies](http://msdn.microsoft.com/en-us/library/yx7xezcf.aspx).

The following illustration shows these steps.

**Resolving an assembly binding request**



For more information about configuring applications, see [Configuration Files](http://msdn.microsoft.com/en-us/library/1xtk877y.aspx). For more information about binding policy, see [How the Runtime Locates Assemblies](http://msdn.microsoft.com/en-us/library/yx7xezcf.aspx).

[Version Information](javascript:void(0))

Each assembly has two distinct ways of expressing version information:

* The assembly's version number, which, together with the assembly name and culture information, is part of the assembly's identity. This number is used by the runtime to enforce version policy and plays a key part in the type resolution process at run time.
* An informational version, which is a string that represents additional version information included for informational purposes only.

[Assembly Version Number](javascript:void(0))

Each assembly has a version number as part of its identity. As such, two assemblies that differ by version number are considered by the runtime to be completely different assemblies. This version number is physically represented as a four-part string with the following format:

<major version>.<minor version>.<build number>.<revision>

For example, version 1.5.1254.0 indicates 1 as the major version, 5 as the minor version, 1254 as the build number, and 0 as the revision number.

The version number is stored in the assembly manifest along with other identity information, including the assembly name and public key, as well as information on relationships and identities of other assemblies connected with the application.

When an assembly is built, the development tool records dependency information for each assembly that is referenced in the assembly manifest. The runtime uses these version numbers, in conjunction with configuration information set by an administrator, an application, or a publisher, to load the proper version of a referenced assembly.

The runtime distinguishes between regular and strong-named assemblies for the purposes of versioning. Version checking only occurs with strong-named assemblies.

For information about specifying version binding policies, see [Configuration Files](http://msdn.microsoft.com/en-us/library/1xtk877y.aspx). For information about how the runtime uses version information to find a particular assembly, see [How the Runtime Locates Assemblies](http://msdn.microsoft.com/en-us/library/yx7xezcf.aspx).

[Assembly Informational Version](javascript:void(0))

The informational version is a string that attaches additional version information to an assembly for informational purposes only; this information is not used at run time. The text-based informational version corresponds to the product's marketing literature, packaging, or product name and is not used by the runtime. For example, an informational version could be "Common Language Runtime version 1.0" or "NET Control SP 2". On the Version tab of the file properties dialog in Microsoft Windows, this information appears in the item "Product Version".

|  |
| --- |
| **Description: NoteNote** |
| Although you can specify any text, a warning message appears on compilation if the string is not in the format used by the assembly version number, or if it is in that format but contains wildcards. This warning is harmless. |

The informational version is represented using the custom attribute [AssemblyInformationalVersionAttribute](http://msdn.microsoft.com/en-us/library/system.reflection.assemblyinformationalversionattribute.aspx). For more information about the informational version attribute, see [Setting Assembly Attributes](http://msdn.microsoft.com/en-us/library/4w8c1y2s.aspx).

# Assembly Placement

**.NET Framework 4.5**

For most .NET Framework applications, you locate assemblies that make up an application in the application's directory, in a subdirectory of the application's directory, or in the global assembly cache (if the assembly is shared). You can override where the common language runtime looks for an assembly by using the [<codeBase> Element](http://msdn.microsoft.com/en-us/library/efs781xb.aspx) in a configuration file. If the assembly does not have a strong name, the location specified using the [<codeBase> Element](http://msdn.microsoft.com/en-us/library/efs781xb.aspx) is restricted to the application directory or a subdirectory. If the assembly has a strong name, the [<codeBase> Element](http://msdn.microsoft.com/en-us/library/efs781xb.aspx) can specify any location on the computer or on a network.

Similar rules apply to locating assemblies when working with unmanaged code or COM interop applications: if the assembly will be shared by multiple applications, it should be installed into the global assembly cache. Assemblies used with unmanaged code must be exported as a type library and registered. Assemblies used by COM interop must be registered in the catalog, although in some cases this registration occurs automatically.

# Assemblies and Side-by-Side Execution

**.NET Framework 4.5**

Side-by-side execution is the ability to store and execute multiple versions of an application or component on the same computer. This means that you can have multiple versions of the runtime, and multiple versions of applications and components that use a version of the runtime, on the same computer at the same time. Side-by-side execution gives you more control over what versions of a component an application binds to, and more control over what version of the runtime an application uses.

Support for side-by-side storage and execution of different versions of the same assembly is an integral part of strong naming and is built into the infrastructure of the runtime. Because the strong-named assembly's version number is part of its identity, the runtime can store multiple versions of the same assembly in the global assembly cache and load those assemblies at run time.

Although the runtime provides you with the ability to create side-by-side applications, side-by-side execution is not automatic. For more information on creating applications for side-by-side execution, see [Guidelines for Creating Applications and Components for Side-by-Side Execution](http://msdn.microsoft.com/en-us/library/z5e12zb4.aspx).