Chapter 17. Assemblies

An *assembly* is the basic unit of deployment in .NET and is also the container for all types. An assembly contains compiled types with their Intermediate Language (IL) code, runtime resources, and information to assist with versioning and referencing other assemblies. An assembly also defines a boundary for type resolution. In .NET, an assembly comprises a single file with a *.dll* extension.

> **NOTE**
>
> When you build an executable application in .NET, you end up with two files: an assembly (*.dll*) and an executable launcher (*.exe*) appropriate to the platform you're targeting.
>
> This differs from what happens in .NET Framework, which generates a *portable executable* (PE) assembly. A PE has an *.exe* extension and acts both as an assembly and an application launcher. A PE can simultaneously target 32- and 64-bit versions of Windows.

Most of the types in this chapter come from the following namespaces:

```
System.Reflection
System.Resources
System.Globalization
```

What's in an Assembly

An assembly contains four kinds of things:

An assembly manifest

Provides information to the CLR, such as the assembly's name, version, and other assemblies that it references

An application manifest

> Provides information to the operating system, such as how the assembly should be deployed and whether administrative elevation is required

Compiled types

> The compiled IL code and metadata of the types defined within the assembly

Resources

> Other data embedded within the assembly, such as images and localizable text

Of these, only the *assembly manifest* is mandatory, although an assembly nearly always contains compiled types (unless it's a resource assembly. See "Resources and Satellite Assemblies").

The Assembly Manifest

The assembly manifest serves two purposes:

- It describes the assembly to the managed hosting environment.
- It acts as a directory to the modules, types, and resources in the assembly.

Assemblies are thus *self-describing*. A consumer can discover all of an assembly's data, types, and functions—without needing additional files.

NOTE

An assembly manifest is not something you add explicitly to an assembly—it's automatically embedded into an assembly as part of compilation.

Here's a summary of the functionally significant data stored in the manifest:

- The simple name of the assembly
- A version number (`AssemblyVersion`)
- A public key and signed hash of the assembly, if strongly named
- A list of referenced assemblies, including their version and public key
- A list of types defined in the assembly
- The culture it targets, if a satellite assembly (`AssemblyCulture`)

The manifest can also store the following informational data:

- A full title and description (`AssemblyTitle` and `AssemblyDescription`)
- Company and copyright information (`AssemblyCompany` and `AssemblyCopyright`)
- A display version (`AssemblyInformationalVersion`)
- Additional attributes for custom data

Some of this data is derived from arguments given to the compiler, such as the list of referenced assemblies or the public key with which to sign the assembly. The rest comes from assembly attributes, indicated in parentheses.

NOTE

You can view the contents of an assembly's manifest with the .NET tool *ildasm.exe*. In Chapter 18, we describe how to use reflection to do the same programmatically.

Specifying assembly attributes

Commonly used assembly attributes can be specified in Visual Studio on the project's Properties page, on the Package tab. The settings on that tab are added to the project file (*.csproj*).

To specify attributes not supported by the Package tab, or if not working with a *.csproj* file, you can specify assembly attributes in source code (this is often done in a file called *AssemblyInfo.cs*).

A dedicated attributes file contains only `using` statements and assembly attribute declarations. For example, to expose internally scoped types to a unit test project, you would do this:

```
using System.Runtime.CompilerServices;

[assembly:InternalsVisibleTo("MyUnitTestProject")]
```

The Application Manifest (Windows)

An application manifest is an XML file that communicates information about the assembly to the OS. An application manifest is embedded into the startup executable as a Win32 resource during the build process. If present, the manifest is read and processed before the CLR loads the assembly—and can influence how Windows launches the application's process.

A .NET application manifest has a root element called `assembly` in the XML namespace `urn:schemas-microsoft-com:asm.v1`:

```
<?xml version="1.0" encoding="utf-8"?>
<assembly manifestVersion="1.0" xmlns="urn:schemas-microsoft-com:asm.v1">
```

```
  <!-- contents of manifest -->
</assembly>
```

The following manifest instructs the OS to request administrative elevation:

```
<?xml version="1.0" encoding="utf-8"?>
<assembly manifestVersion="1.0" xmlns="urn:schemas-microsoft-com:asm.v1">
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
    <security>
      <requestedPrivileges>
        <requestedExecutionLevel level="requireAdministrator" />
      </requestedPrivileges>
    </security>
  </trustInfo>
</assembly>
```

(UWP applications have a far more elaborate manifest, described in the *Package.appxmanifest* file. This includes a declaration of the program's capabilities, which determine permissions granted by the OS. The easiest way to edit this file is with Visual Studio, which displays a dialog when you double-click the manifest file.)

Deploying an application manifest

You can add an application manifest to a .NET project in Visual Studio by right-clicking your project in Solution Explorer, selecting Add, then New Item, and then choosing Application Manifest File. Upon building, the manifest will be embedded into the output assembly.

NOTE

The .NET tool *ildasm.exe* is blind to the presence of an embedded application manifest. Visual Studio, however, indicates whether an embedded application manifest is present if you double-click the assembly in Solution Explorer.

Modules

The contents of an assembly are actually packaged within an intermediate container, called a *module*. A module corresponds to a file containing the contents of an assembly. The reason for this extra layer of containership is to allow an assembly to span multiple files, a feature present in .NET Framework but absent in .NET 5+ and .NET Core. Figure 17-1 illustrates the relationship.

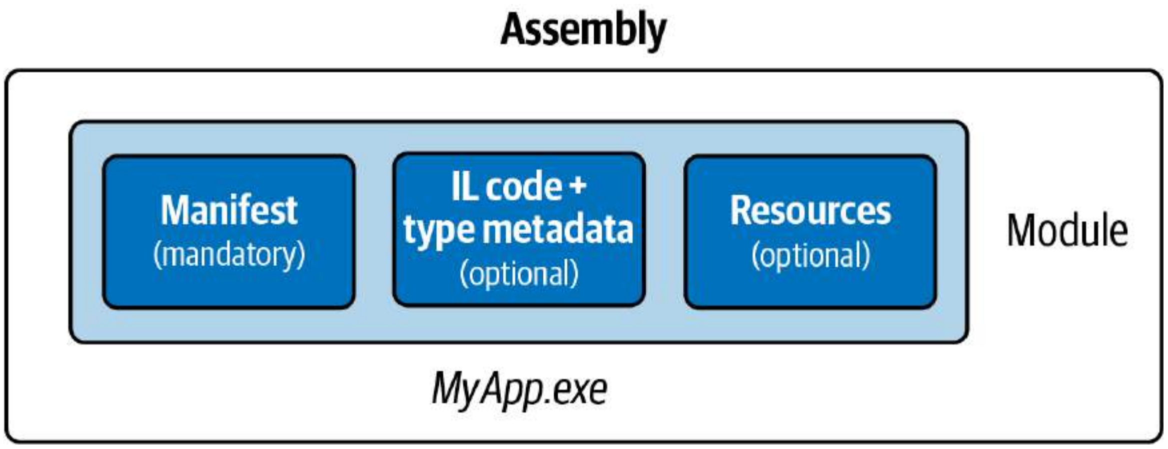

Figure 17-1. Single-file assembly

Although .NET does not support multifile assemblies, at times you need to be aware of the extra level of containership that modules impose. The main scenario is with reflection (see "Reflecting Assemblies" and "Emitting Assemblies and Types").

The Assembly Class

The `Assembly` class in `System.Reflection` is a gateway to accessing assembly metadata at runtime. There are a number of ways to obtain an assembly object: the simplest is via a `Type`'s `Assembly` property:

```
Assembly a = typeof (Program).Assembly;
```

You can also obtain an `Assembly` object by calling one of `Assembly`'s static methods:

`GetExecutingAssembly`

Returns the assembly of the type that defines the currently executing function

`GetCallingAssembly`

Does the same as `GetExecutingAssembly` but for the function that called the currently executing function

`GetEntryAssembly`

Returns the assembly defining the application's original entry method

After you have an `Assembly` object, you can use its properties and methods to query the assembly's metadata and reflect upon its types. Table 17-1 shows a summary of these functions.

Table 17-1. Assembly members

Functions	Purpose	See the section...
`FullName`, `GetName`	Returns the fully qualified name or an `AssemblyName` object	"Assembly Names"
`CodeBase`, `Location`	Location of the assembly file	"Loading, Resolving, and Isolating Assemblies"
`Load`, `LoadFrom`, `LoadFile`	Manually loads an assembly into memory	"Loading, Resolving, and Isolating Assemblies"
`GetSatelliteAssembly`	Locates the satellite assembly of a given culture	"Resources and Satellite Assemblies"
`GetType`, `GetTypes`	Returns a type, or all types, defined in the assembly	"Reflecting and Activating Types"
`EntryPoint`	Returns the application's entry method, as a `MethodInfo`	"Reflecting and Invoking Members"
`GetModule`, `GetModules`, `ManifestModule`	Returns all modules, or the main module, of an assembly	"Reflecting Assemblies"
`GetCustomAttribute`, `GetCustomAttributes`	Returns the assembly's attributes	"Working with Attributes"

Strong Names and Assembly Signing

NOTE

Strongly naming an assembly was important in .NET Framework for two reasons:

- It allowed the assembly to be loaded into the “Global assembly cache.”
- It allowed the assembly to by referenced by other strongly named assemblies.

Strong naming is much less important in .NET 5+ and .NET Core, because these runtimes do not have a global assembly cache, nor do they impose the second restriction.

A *strongly named* assembly has a unique identity. It works by adding two bits of metadata to the manifest:

- A *unique number* that belongs to the authors of the assembly
- A *signed hash* of the assembly, proving that the unique number holder produced the assembly

This requires a public/private key pair. The *public key* provides the unique identifying number, and the *private key* facilitates signing.

NOTE

Strong-name-signing is not the same as *Authenticode*-signing. We cover Authenticode later in this chapter.

The public key is valuable in guaranteeing the uniqueness of assembly references: a strongly named assembly incorporates the public key into its identity.

In .NET Framework, the private key protects your assembly from tampering, in that without your private key, no one can release a modified version of the assembly without the signature breaking. In practice, this is of use when loading an assembly into .NET Framework’s global assembly cache. In .NET 5+ and .NET Core, the signature is of little use because it’s never checked.

Adding a strong name to a previously “weak” named assembly changes its identity. For this reason, it pays to strong-name an assembly from the outset if you think the assembly might need a strong name in the future.

How to Strongly Name an Assembly

To give an assembly a strong name, first generate a public/private key pair with the *sn.exe* utility:

```
sn.exe -k MyKeyPair.snk
```

> **NOTE**
>
> Visual Studio installs a shortcut called *Developer Command Prompt for VS*, which starts a command prompt whose `PATH` contains development tools such as *sn.exe*.

This manufactures a new key pair and stores it to a file called *MyKeyPair.snk*. If you subsequently lose this file, you will permanently lose the ability to recompile your assembly with the same identity.

You can sign an assembly with this file by updating your project file. From Visual Studio, go to the Project Properties window, and then, on the *Signing* tab, select the “Sign the assembly” checkbox and select your *.snk* file.

The same key pair can sign multiple assemblies—they’ll still have distinct identities if their simple names differ.

Assembly Names

An assembly’s “identity” comprises four pieces of metadata from its manifest:

- Its simple name
- Its version (“0.0.0.0” if not present)

- Its culture (“neutral” if not a satellite)
- Its public key token (“null” if not strongly named)

The simple name comes not from any attribute, but from the name of the file to which it was originally compiled (less any extension). So, the simple name of the *System.Xml.dll* assembly is “System.Xml.” Renaming a file doesn’t change the assembly’s simple name.

The version number comes from the `AssemblyVersion` attribute. It’s a string divided into four parts as follows:

```
major.minor.build.revision
```

You can specify a version number as follows:

```
[assembly: AssemblyVersion ("2.5.6.7")]
```

The culture comes from the `AssemblyCulture` attribute and applies to satellite assemblies, described later in the section “Resources and Satellite Assemblies”.

The public key token comes from the strong name supplied at compile time, as we discussed in the preceding section.

Fully Qualified Names

A fully qualified assembly name is a string that includes all four identifying components, in this format:

```
simple-name, Version=version, Culture=culture, PublicKeyToken=public-key
```

For example, the fully qualified name of *System.Private.CoreLib.dll* is *System.Private.CoreLib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=7cec85d7bea7798e*.

If the assembly has no `AssemblyVersion` attribute, the version appears as `0.0.0.0`. If it is unsigned, its public key token appears as `null`.

An `Assembly` object's `FullName` property returns its fully qualified name. The compiler always uses fully qualified names when recording assembly references in the manifest.

NOTE

A fully qualified assembly name does not include a directory path to assist in locating it on disk. Locating an assembly residing in another directory is an entirely separate matter that we pick up in "Loading, Resolving, and Isolating Assemblies".

The AssemblyName Class

`AssemblyName` is a class with a typed property for each of the four components of a fully qualified assembly name. `AssemblyName` has two purposes:

- It parses or builds a fully qualified assembly name.
- It stores some extra data to assist in resolving (finding) the assembly.

You can obtain an `AssemblyName` object in any of the following ways:

- Instantiate an `AssemblyName`, providing a fully qualified name.
- Call `GetName` on an existing `Assembly`.
- Call `AssemblyName.GetAssemblyName`, providing the path to an assembly file on disk.

You can also instantiate an `AssemblyName` object without any arguments and then set each of its properties to build a fully qualified name. An `AssemblyName` is mutable when constructed in this manner.

Here are its essential properties and methods:

```
string      FullName    { get; }           // Fully qualified name
string      Name        { get; set; }      // Simple name
Version     Version     { get; set; }      // Assembly version
CultureInfo CultureInfo { get; set; }      // For satellite assemblies
string      CodeBase    { get; set; }      // Location

byte[]      GetPublicKey();                // 160 bytes
void        SetPublicKey (byte[] key);
byte[]      GetPublicKeyToken();           // 8-byte version
void        SetPublicKeyToken (byte[] publicKeyToken);
```

`Version` is itself a strongly typed representation, with properties for `Major`, `Minor`, `Build`, and `Revision` numbers. `GetPublicKey` returns the full cryptographic public key; `GetPublicKeyToken` returns the last eight bytes used in establishing identity.

To use `AssemblyName` to obtain the simple name of an assembly:

```
Console.WriteLine (typeof (string).Assembly.GetName().Name);
// System.Private.CoreLib
```

To get an assembly version:

```
string v = myAssembly.GetName().Version.ToString();
```

Assembly Informational and File Versions

Two further assembly attributes are available for expressing version-related information. Unlike `AssemblyVersion`, the following two attributes do not affect an assembly's identity and so have no effect on what happens at compile-time or at runtime:

`AssemblyInformationalVersion`

> The version as displayed to the end user. This is visible in the Windows File Properties dialog box as Product Version.

Any string can go here, such as "5.1 Beta 2." Typically, all of the assemblies in an application would be assigned the same informational version number.

`AssemblyFileVersion`

This is intended to refer to the build number for that assembly. This is visible in the Windows File Properties dialog box as File Version. As with `AssemblyVersion`, it must contain a string consisting of up to four numbers separated by periods.

Authenticode Signing

Authenticode is a code-signing system whose purpose is to prove the identity of the publisher. Authenticode and *strong-name* signing are independent: you can sign an assembly with either or both systems.

Although strong-name signing can prove that assemblies A, B, and C came from the same party (assuming the private key hasn't been leaked), it can't tell you who that party was. To know that the party was Joe Albahari—or Microsoft Corporation—you need Authenticode.

Authenticode is useful when downloading programs from the internet, because it provides assurance that a program came from whoever was named by the Certificate Authority and was not modified in transit. It also prevents the "Unknown Publisher" warning when running a downloaded application for the first time. Authenticode signing is also a requirement when submitting apps to the Windows Store.

Authenticode works with not only .NET assemblies, but also unmanaged executables and binaries such as *.msi* deployment files. Of course, Authenticode doesn't guarantee that a program is free from malware—although it does make it less likely. A person or entity has been willing to put its name (backed by a passport or company document) behind the executable or library.

NOTE

The CLR does not treat an Authenticode signature as part of an assembly's identity. However, it can read and validate Authenticode signatures on demand, as you'll see soon.

Signing with Authenticode requires that you contact a *Certificate Authority* (CA) with evidence of your personal identity or company's identity (articles of incorporation, etc.). After the CA has checked your documents, it will issue an X.509 code-signing certificate that is typically valid for one to five years. This enables you to sign assemblies with the *signtool* utility. You can also make a certificate yourself with the *makecert* utility; however, it will be recognized only on computers on which the certificate is explicitly installed.

The fact that (non-self-signed) certificates can work on any computer relies on public key infrastructure. Essentially, your certificate is signed with another certificate belonging to a CA. The CA is trusted because all CAs are loaded into the OS. (To see them, go to the Windows Control Panel and then, in the search box, type **`certificate`**. In the Administrative Tools section, click "Manage computer certificates." This launches the Certificate Manager. Open the node Trusted Root Certification Authorities and click Certificates.) A CA can revoke a publisher's certificate if leaked, so verifying an Authenticode signature requires periodically asking the CA for an up-to-date list of certification revocations.

Because Authenticode uses cryptographic signing, an Authenticode signature is invalid if someone subsequently tampers with the file. We discuss cryptography, hashing, and signing in Chapter 20.

How to Sign with Authenticode

Obtaining and installing a certificate

The first step is to obtain a code-signing certificate from a CA (see the sidebar that follows). You can then either work with the certificate as a password-protected file or load the certificate into the computer's certificate store. The benefit of doing the latter is that you can sign without needing to specify a password. This is advantageous because it prevents having a password visible in automated build scripts or batch files.

WHERE TO GET A CODE-SIGNING CERTIFICATE

Just a handful of code-signing CAs are preloaded into Windows as root certification authorities. These include Comodo, GoDaddy, GlobalSign, DigiCert, Thawte, and Symantec.

There are also resellers such as K Software that offer discounted code-signing certificates from the aforementioned authorities.

The Authenticode certificates issued by K Software, Comodo, GoDaddy, and GlobalSign are advertised as less restrictive in that they will also sign non-Microsoft programs. Aside from this, the products from all vendors are functionally equivalent.

Note that a certificate for SSL cannot generally be used for Authenticode signing (despite using the same X.509 infrastructure). This is, in part, because a certificate for SSL is about proving ownership of a domain; Authenticode is about proving who you are.

To load a certificate into the computer's certificate store, open the Certificate Manager as described earlier. Open the Personal folder, right-click its Certificates folder, and then pick All Tasks/Import. An import wizard guides you through the process. After the import is complete, click the View button on the certificate, go to the Details tab, and copy the certificate's *thumbprint*. This is the SHA-256 hash that you'll subsequently need to identify the certificate when signing.

NOTE

If you also want to strong-name-sign your assembly, you must do so *before* Authenticode signing. This is because the CLR knows about Authenticode signing, but not vice versa. So, if you strong-name-sign an assembly *after* Authenticode-signing it, the latter will see the addition of the CLR's strong name as an unauthorized modification and consider the assembly tampered.

Signing with signtool.exe

You can Authenticode-sign your programs with the *signtool* utility that comes with Visual Studio (look in the *Microsoft SDKs\ClickOnce\SignTool* folder under *Program Files*). The following signs a file called *LINQPad.exe* with the certificate located in the computer's *My Store* called "Joseph Albahari," using the secure SHA256 hashing algorithm:

```
signtool sign /n "Joseph Albahari" /fd sha256 LINQPad.exe
```

You can also specify a description and product URL with `/d` and `/du`:

```
... /d LINQPad /du http://www.linqpad.net
```

In most cases, you will also want to specify a *time-stamping server*.

Time stamping

After your certificate expires, you'll no longer be able to sign programs. However, programs that you signed *before* its expiry will still be valid—if you specified a *time-stamping server* with the `/tr` switch when signing. The CA will provide you with a URI for this purpose: the following is for Comodo (or K Software):

```
... /tr http://timestamp.comodoca.com/authenticode /td SHA256
```

Verifying that a program has been signed

The easiest way to view an Authenticode signature on a file is to view the file's properties in Windows Explorer (look in the Digital Signatures tab). The *signtool* utility also provides an option for this.

Resources and Satellite Assemblies

An application typically contains not only executable code, but also content such as text, images, or XML files. Such content can be represented in an assembly through a *resource*. There are two overlapping use cases for resources:

- Incorporating data that cannot go into source code, such as images
- Storing data that might need translation in a multilingual application

An assembly resource is ultimately a byte stream with a name. You can think of an assembly as containing a dictionary of byte arrays keyed by string. You can see this in *ildasm* if you disassemble an assembly that contains a resource called *banner.jpg* and a resource called *data.xml*:

```
.mresource public banner.jpg
{
  // Offset: 0x00000F58 Length: 0x000004F6
}
.mresource public data.xml
{
  // Offset: 0x00001458 Length: 0x0000027E
}
```

In this case, *banner.jpg* and *data.xml* were included directly in the assembly—each as its own embedded resource. This is the simplest way to work.

.NET also lets you add content through intermediate *.resources* containers. These are designed for holding content that might require translation into different languages. Localized *.resources* can be packaged as individual satellite assemblies that are automatically picked up at runtime, based on the user's OS language.

Figure 17-2 illustrates an assembly that contains two directly embedded resources, plus a *.resources* container called *welcome.resources*, for which we've created two localized satellites.

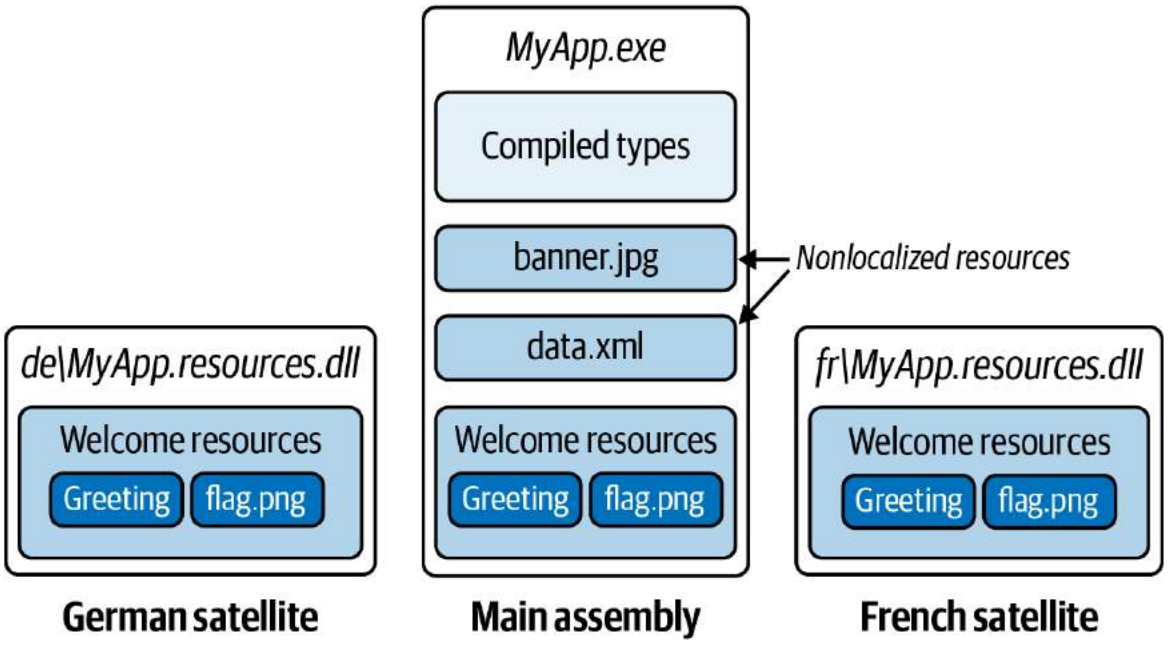

Figure 17-2. Resources

Directly Embedding Resources

NOTE

Embedding resources into assemblies is not supported in Windows Store apps. Instead, add any extra files to your deployment package, and access them by reading from your application `StorageFolder` (`Package.Current.InstalledLocation`).

To directly embed a resource using Visual Studio:

- Add the file to your project.
- Set its build action to Embedded Resource.

Visual Studio always prefixes resource names with the project's default namespace, plus the names of any subfolders in which the file is contained.

So, if your project's default namespace was `Westwind.Reports` and your file was called *banner.jpg* in the folder *pictures*, the resource name would be *Westwind.Reports.pictures.banner.jpg*.

NOTE

Resource names are case sensitive. This makes project subfolder names in Visual Studio that contain resources effectively case sensitive.

To retrieve a resource, you call `GetManifestResourceStream` on the assembly containing the resource. This returns a stream, which you can then read as any other:

```
Assembly a = Assembly.GetEntryAssembly();

using (Stream s = a.GetManifestResourceStream ("TestProject.data.xml"))
using (XmlReader r = XmlReader.Create (s))
  ...

System.Drawing.Image image;
using (Stream s = a.GetManifestResourceStream ("TestProject.banner.jpg"))
  image = System.Drawing.Image.FromStream (s);
```

The stream returned is seekable, so you can also do this:

```
byte[] data;
using (Stream s = a.GetManifestResourceStream ("TestProject.banner.jpg"))
  data = new BinaryReader (s).ReadBytes ((int) s.Length);
```

If you've used Visual Studio to embed the resource, you must remember to include the namespace-based prefix. To help avoid error, you can specify the prefix in a separate argument, using a *type*. The type's namespace is used as the prefix:

```
using (Stream s = a.GetManifestResourceStream (typeof (X), "data.xml"))
```

`X` can be any type with the desired namespace of your resource (typically, a type in the same project folder).

NOTE

Setting a project item's build action in Visual Studio to Resource within a Windows Presentation Foundation (WPF) application is *not* the same as setting its build action to Embedded Resource. The former actually adds the item to a *.resources* file called *<AssemblyName>.g.resources*, whose content you access through WPF's `Application` class, using a URI as a key.

To add to the confusion, WPF further overloads the term "resource." *Static resources* and *dynamic resources* are both unrelated to assembly resources!

`GetManifestResourceNames` returns the names of all resources in the assembly.

.resources Files

.resources files are containers for potentially localizable content. A *.resources* file ends up as an embedded resource within an assembly—just like any other kind of file. The difference is that you must do the following:

- Package your content into the *.resources* file to begin with
- Access its content through a `ResourceManager` or *pack URI* rather than a `GetManifestResourceStream`

.resources files are structured in binary and so are not human-editable; therefore, you must rely on tools provided by .NET and Visual Studio to work with them. The standard approach with strings or simple data types is to use the *.resx* format, which can be converted to a *.resources* file either by Visual Studio or the `resgen` tool. The *.resx* format is also suitable for images intended for a Windows Forms or ASP.NET application.

In a WPF application, you must use Visual Studio's "Resource" build action for images or similar content needing to be referenced by URI. This applies whether localization is needed or not.

We describe how to do each of these in the following sections.

.resx Files

A *.resx* file is a design-time format for producing *.resources* files. A *.resx* file uses XML and is structured with name/value pairs as follows:

```
<root>
  <data name="Greeting">
    <value>hello</value>
  </data>
  <data name="DefaultFontSize" type="System.Int32, mscorlib">
    <value>10</value>
  </data>
</root>
```

To create a *.resx* file in Visual Studio, add a project item of type Resources File. The rest of the work is done automatically:

- The correct header is created.
- A designer is provided for adding strings, images, files, and other kinds of data.
- The *.resx* file is automatically converted to the *.resources* format and embedded into the assembly upon compilation.
- A class is written to help you access the data later on.

NOTE

The resource designer adds images as typed `Image` objects (*System.Drawing.dll*) rather than as byte arrays, making them unsuitable for WPF applications.

Reading .resources files

NOTE

If you create a *.resx* file in Visual Studio, a class of the same name is generated automatically with properties to retrieve each of its items.

The `ResourceManager` class reads *.resources* files embedded within an assembly:

```
ResourceManager r = new ResourceManager ("welcome",
                                          Assembly.GetExecutingAssembly());
```

(The first argument must be namespace-prefixed if the resource was compiled in Visual Studio.)

You can then access what's inside by calling `GetString` or `GetObject` with a cast:

```
string greeting = r.GetString ("Greeting");
int fontSize = (int) r.GetObject ("DefaultFontSize");
Image image = (Image) r.GetObject ("flag.png");
```

To enumerate the contents of a *.resources* file:

```
ResourceManager r = new ResourceManager (...);
ResourceSet set = r.GetResourceSet (CultureInfo.CurrentUICulture,
                                    true, true);
foreach (System.Collections.DictionaryEntry entry in set)
  Console.WriteLine (entry.Key);
```

Creating a pack URI resource in Visual Studio

In a WPF application, XAML files need to be able to access resources by URI. For instance:

```
<Button>
```

```
  <Image Height="50" Source="flag.png"/>
</Button>
```

Or, if the resource is in another assembly:

```
<Button>
  <Image Height="50" Source="UtilsAssembly;Component/flag.png"/>
</Button>
```

(`Component` is a literal keyword.)

To create resources that can be loaded in this manner, you cannot use *.resx* files. Instead, you must add the files to your project and set their build action to Resource (not Embedded Resource). Visual Studio then compiles them into a *.resources* file called *<AssemblyName>.g.resources*—also the home of compiled XAML (*.baml*) files.

To load a URI-keyed resource programmatically, call `Application.GetResourceStream`:

```
Uri u = new Uri ("flag.png", UriKind.Relative);
using (Stream s = Application.GetResourceStream (u).Stream)
```

Notice we used a relative URI. You can also use an absolute URI in exactly the following format (the three commas are not a typo):

```
Uri u = new Uri ("pack://application:,,,/flag.png");
```

If you'd rather specify an `Assembly` object, you can retrieve content instead with a `ResourceManager`:

```
Assembly a = Assembly.GetExecutingAssembly();
ResourceManager r = new ResourceManager (a.GetName().Name + ".g", a);
using (Stream s = r.GetStream ("flag.png"))
  ...
```

A `ResourceManager` also lets you enumerate the content of a *.g.resources* container within a given assembly.

Satellite Assemblies

Data embedded in *.resources* is localizable.

Resource localization is relevant when your application runs on a version of Windows built to display everything in a different language. For consistency, your application should use that same language, too.

A typical setup is as follows:

- The main assembly contains *.resources* for the default, or *fallback*, language.
- Separate *satellite assemblies* contain localized *.resources* translated to different languages.

When your application runs, .NET examines the language of the current OS (from `CultureInfo.CurrentUICulture`). Whenever you request a resource using `ResourceManager`, the runtime looks for a localized satellite assembly. If one's available—and it contains the resource key you requested—it's used in place of the main assembly's version.

This means that you can enhance language support simply by adding new satellites—without changing the main assembly.

NOTE

A satellite assembly cannot contain executable code, only resources.

Satellite assemblies are deployed in subdirectories of the assembly's folder, as follows:

```
programBaseFolder\MyProgram.exe
```

```
                    \MyLibrary.exe
                    \XX\MyProgram.resources.dll
                    \XX\MyLibrary.resources.dll
```

XX refers to the two-letter language code (such as “de” for German) or a language and region code (such as “en-GB” for English in Great Britain). This naming system allows the CLR to find and load the correct satellite assembly automatically.

Building satellite assemblies

Recall our previous *.resx* example, which included the following:

```
<root>
  ...
  <data name="Greeting"
    <value>hello</value>
  </data>
</root>
```

We then retrieved the greeting at runtime as follows:

```
ResourceManager r = new ResourceManager ("welcome",
                                         Assembly.GetExecutingAssembly());
Console.Write (r.GetString ("Greeting"));
```

Suppose that we want this to instead write “hallo” if running on the German version of Windows. The first step is to add another *.resx* file named *welcome.de.resx* that substitutes *hello* for *hallo*:

```
<root>
  <data name="Greeting">
    <value>hallo<value>
  </data>
</root>
```

In Visual Studio, this is all you need to do—when you rebuild, a satellite assembly called *MyApp.resources.dll* is automatically created in a subdirectory called *de*.

Testing satellite assemblies

To simulate running on an OS with a different language, you must change the `CurrentUICulture` using the `Thread` class:

```
System.Threading.Thread.CurrentThread.CurrentUICulture
  = new System.Globalization.CultureInfo ("de");
```

`CultureInfo.CurrentUICulture` is a read-only version of the same property.

NOTE

A useful testing strategy is to ℓѻ¢αℓïʐɘ into words that can still be read as English but do not use the standard Roman Unicode characters.

Visual Studio designer support

The designers in Visual Studio provide extended support for localizing components and visual elements. The WPF designer has its own workflow for localization; other `Component`-based designers use a design-time-only property to make it appear that a component or Windows Forms control has a `Language` property. To customize for another language, simply change the `Language` property and then start modifying the component. All properties of controls that are attributed as `Localizable` will be saved to a *.resx* file for that language. You can switch between languages at any time just by changing the `Language` property.

Cultures and Subcultures

Cultures are split into cultures and subcultures. A culture represents a particular language; a subculture represents a regional variation of that language. The .NET runtime follows the `RFC1766` standard, which represents cultures and subcultures with two-letter codes. Here are the codes for English and German cultures:

```
En
de
```

Here are the codes for the Australian English and Austrian German subcultures:

```
en-AU
de-AT
```

A culture is represented in .NET with the `System.Globalization.CultureInfo` class. You can examine the current culture of your application, as follows:

```
Console.WriteLine (System.Threading.Thread.CurrentThread.CurrentCulture);
Console.WriteLine (System.Threading.Thread.CurrentThread.CurrentUICulture);
```

Running this on a computer localized for Australia illustrates the difference between the two:

```
en-AU
en-US
```

`CurrentCulture` reflects the regional settings of the Windows Control Panel, whereas `CurrentUICulture` reflects the language of the OS.

Regional settings include such things as time zone and the formatting of currency and dates. `CurrentCulture` determines the default behavior of such functions as `DateTime.Parse`. Regional settings can be customized to the point where they no longer resemble any particular culture.

`CurrentUICulture` determines the language in which the computer communicates with the user. Australia doesn't need a separate version of English for this purpose, so it just uses the US one. If I spent a couple of months working in Austria, I would go to the Control Panel and change my `CurrentCulture` to Austrian-German. However, given that I can't speak German, my `CurrentUICulture` would remain US English.

`ResourceManager`, by default, uses the current thread's `CurrentUICulture` property to determine the correct satellite assembly to load. `ResourceManager` uses a fallback mechanism when loading resources. If a subculture assembly is defined, that one is used; otherwise, it falls back to the generic culture. If the generic culture is not present, it falls back to the default culture in the main assembly.

Loading, Resolving, and Isolating Assemblies

Loading an assembly from a known location is a relatively simple process. We refer to this as *assembly loading*.

More commonly, however, you (or the CLR) will need to load an assembly knowing only its full (or simple) name. This is called *assembly resolution*. Assembly resolution differs from loading in that the assembly must first be located.

Assembly resolution is triggered in two scenarios:

- By the CLR, when it needs to resolve a dependency
- Explicitly, when you call a method such as `Assembly.Load(AssemblyName)`

To illustrate the first scenario, consider an application comprising a main assembly plus a set of statically referenced library assemblies (dependencies), as shown in this example:

```
AdventureGame.dll    // Main assembly
Terrain.dll          // Referenced assembly
UIEngine.dll         // Referenced assembly
```

By "statically referenced," we mean that *AdventureGame.dll* was compiled with references to *Terrain.dll* and *UIEngine.dll*. The compiler itself does not need to perform assembly resolution, because it's told (either explicitly or by MSBuild) where to find *Terrain.dll* and *UIEngine.dll*. During compilation, it writes the *full names* of the Terrain and UIEngine assemblies into the metadata of *AdventureGame.dll* but no information on where to find them. So, at runtime, the Terrain and UIEngine assemblies must be *resolved*.

Assembly loading and resolution is handled by an *assembly load context* (ALC); specifically, an instance of the `AssemblyLoadContext` class in `System.Runtime.Loader`. Because *AdventureGame.dll* is the main assembly for the application, the CLR uses the *default ALC* (`AssemblyLoadContext.Default`) to resolve its dependencies. The default ALC resolves dependencies first by looking for and examining a file called *AdventureGame.deps.json* (which describes where to find dependencies), or if not present, it looks in the application base folder, where it will find *Terrain.dll* and *UIEngine.dll*. (The default ALC also resolves the .NET runtime assemblies.)

As a developer, you can dynamically load additional assemblies during the execution of your program. For example, you might want to package optional features in assemblies that you deploy only when those features have been purchased. In such a case, you could load the extra assemblies, when present, by calling `Assembly.Load(AssemblyName)`.

A more complex example would be implementing a plug-in system whereby the user can provide third-party assemblies that your application detects and loads at runtime to extend your application’s functionality. The complexity arises because each plug-in assembly might have its own dependencies that must also be resolved.

By subclassing `AssemblyLoadContext` and overriding its assembly resolution method (`Load`), you can control how a plug-in finds its dependencies. For example, you might decide that each plug-in should reside in its own folder, and its dependencies should also reside in that folder.

ALCs have another purpose: by instantiating a separate `AssemblyLoadContext` for each (plug-in + dependencies), you can keep each isolated, ensuring that their dependencies load in parallel and do not interfere with one another (nor the host application). Each, for instance, can have its own version of JSON.NET. Hence, in addition to *loading* and *resolution*, ALCs also provide a mechanism for *isolation*. Under certain conditions, ALCs can even be *unloaded*, freeing their memory.

In this section, we elaborate on each of these principles and describe the following:

- How ALCs handle loading and resolution
- The role of the default ALC
- `Assembly.Load` and contextual ALCs
- How to use `AssemblyDependencyResolver`
- How to load and resolve unmanaged libraries
- Unloading ALCs
- The legacy assembly loading methods

Then, we put the theory to work and demonstrate how to write a plug-in system with ALC isolation.

NOTE

The `AssemblyLoadContext` class is new to .NET 5+ and .NET Core. In .NET Framework, ALCs were present but restricted and hidden: the only way to create and interact with them was indirectly via the `LoadFile(string)`, `LoadFrom(string)`, and `Load(byte[])` static methods on the `Assembly` class. Compared to the ALC API, these methods are inflexible, and their use can lead to surprises (particularly when handling dependencies). For this reason, it's best to favor explicit use of the `AssemblyLoadContext` API in .NET 5+ and .NET Core.

Assembly Load Contexts

As we just discussed, the `AssemblyLoadContext` class is responsible for loading and resolving assemblies as well as providing a mechanism for isolation.

Every .NET `Assembly` object belongs to exactly one `AssemblyLoadContext`. You can obtain the ALC for an assembly, as follows:

```
Assembly assem = Assembly.GetExecutingAssembly();
AssemblyLoadContext context = AssemblyLoadContext.GetLoadContext (assem);
Console.WriteLine (context.Name);
```

Conversely, you can think of an ALC as "containing" or "owning" assemblies, which you can obtain via its `Assemblies` property. Following on from the previous example:

```
foreach (Assembly a in context.Assemblies)
  Console.WriteLine (a.FullName);
```

The `AssemblyLoadContext` class also has a static `All` property that enumerates all ALCs.

You can create a new ALC just by instantiating `AssemblyLoadContext` and providing a name (the name is helpful when debugging), although more commonly, you’d first subclass `AssemblyLoadContext` so that you can implement logic to *resolve* dependencies; in other words, load an assembly from its *name*.

Loading assemblies

`AssemblyLoadContext` provides the following methods to explicitly load an assembly into its context:

```
public Assembly LoadFromAssemblyPath (string assemblyPath);
public Assembly LoadFromStream (Stream assembly, Stream assemblySymbols);
```

The first method loads an assembly from a file path, whereas the second method loads it from a `Stream` (which can come directly from memory). The second parameter is optional and corresponds to the contents of the project debug (*.pdb*) file, which allows stack traces to include source code information when code executes (useful in exception reporting).

With both of these methods, no *resolution* takes place.

The following loads the assembly *c:\temp\foo.dll* into its own ALC:

```
var alc = new AssemblyLoadContext ("Test");
Assembly assem = alc.LoadFromAssemblyPath (@"c:\temp\foo.dll");
```

If the assembly is valid, loading will always succeed, subject to one important rule: an assembly’s *simple name* must be unique within its ALC. This means that you cannot load multiple versions of the same-named assembly into a single ALC; to do this, you must create additional ALCs. We could load another copy of *foo.dll*, as follows:

```
var alc2 = new AssemblyLoadContext ("Test 2");
Assembly assem2 = alc2.LoadFromAssemblyPath (@"c:\temp\foo.dll");
```

Note that types that originate from different `Assembly` objects are incompatible even if the assemblies are otherwise identical. In our example, the types in `assem` are incompatible with the types in `assem2`.

After an assembly is loaded, it cannot be unloaded except by unloading its ALC (see "Unloading ALCs"). The CLR maintains a lock of the file for the duration that it's loaded.

NOTE

You can avoid locking the file by loading the assembly via a byte array:

```
bytes[] bytes = File.ReadAllBytes (@"c:\temp\foo.dll");
var ms = new MemoryStream (bytes);
var assem = alc.LoadFromStream (ms);
```

This has two drawbacks:

- The assembly's `Location` property will end up blank. Sometimes, it's useful to know where an assembly was loaded from (and some APIs rely on it being populated).
- Private memory consumption must increase immediately to accommodate the full size of the assembly. If you instead load from a filename, the CLR uses a memory-mapped file, which enables lazy loading and process sharing. Also, should memory run low, the OS can release its memory and reload as required without writing to a page file.

LoadFromAssemblyName

`AssemblyLoadContext` also provides the following method, which loads an assembly by *name*:

```
public Assembly LoadFromAssemblyName (AssemblyName assemblyName);
```

Unlike the two methods just discussed, you don't pass in any information to indicate where the assembly is located; instead you're instructing the ALC to *resolve* the assembly.

Resolving assemblies

The preceding method triggers *assembly resolution*. The CLR also triggers assembly resolution when loading dependencies. For example, suppose that assembly A statically references assembly B. To resolve reference B, the CLR triggers assembly resolution on whichever *ALC assembly A was loaded into*.

> **NOTE**
>
> The CLR resolves dependencies by triggering assembly resolution—whether the triggering assembly is in the default or a custom ALC. The difference is that with the default ALC, the resolution rules are hardcoded, whereas with a custom ALC, you write the rules yourself.

Here's what then happens:

1. The CLR first checks whether an identical resolution has already taken place in that ALC (with a matching full assembly name); if so, it returns the `Assembly` it returned before.

2. Otherwise, it calls the ALC's (virtual protected) `Load` method, which does the work of locating and loading the assembly. The default ALC's `Load` method applies the rules we describe in "The Default ALC". With a custom ALC, it's entirely up to you how you locate the assembly. For instance, you might look in some folder and then call `LoadFromAssemblyPath` when you find the assembly. It's also perfectly legal to return an already-loaded assembly from the same or another ALC (we demonstrate this in "Writing a Plug-In System").

3. If Step 2 returns null, the CLR then calls the `Load` method on the default ALC (this serves as a useful "fallback" for resolving .NET runtime and common application assemblies).

4. If Step 3 returns null, the CLR then fires the `Resolving` events on both ALCs—first, on the default ALC and then on the original ALC.

5. (For compatibility with .NET Framework): If the assembly still hasn't been resolved, the `AppDomain.CurrentDomain.AssemblyResolve` event fires.

NOTE

After this process completes, the CLR does a "sanity check" to ensure that whatever assembly was loaded has a name that's compatible with what was requested. The simple name must match; the public key token must match *if specified*. The version need not match —it can be higher or lower than what was requested.

From this, we can see that there are two ways to implement assembly resolution in a custom ALC:

- Override the ALC's `Load` method. This gives your ALC "first say" over what happens, which is usually desirable (and essential when you need isolation).
- Handle the ALC's `Resolving` event. This fires only *after* the default ALC has failed to resolve assembly.

NOTE

If you attach multiple event handlers to the `Resolving` event, the first to return a non-null value wins.

To illustrate, let's assume that we want to load an assembly that our main application knew nothing about at compile time, called *foo.dll*, located in *c:\temp* (which is different from our application folder). We'll also assume that *foo.dll* has a private dependency on *bar.dll*. We want to ensure that when we load *c:\temp\foo.dll* and execute its code, *c:\temp\bar.dll* can correctly resolve. We also want to ensure that `foo` and its private dependency, `bar`, do not interfere with the main application.

Let’s begin by writing a custom ALC that overrides `Load`:

```
using System.IO;
using System.Runtime.Loader;

class FolderBasedALC : AssemblyLoadContext
{
  readonly string _folder;
  public FolderBasedALC (string folder) => _folder = folder;

  protected override Assembly Load (AssemblyName assemblyName)
  {
    // Attempt to find the assembly:
    string targetPath = Path.Combine (_folder, assemblyName.Name + ".dll");

    if (File.Exists (targetPath))
      return LoadFromAssemblyPath (targetPath);   // Load the assembly

    return null;    // We can’t find it: it could be a .NET runtime assembly
  }
}
```

Notice that in the `Load` method, we return `null` if the assembly file is not present. This check is important because *foo.dll* will also have dependencies on the .NET BCL assemblies; hence, the `Load` method will be called on assemblies such as `System.Runtime`. By returning null, we allow the CLR to fall back to the default ALC, which will correctly resolve these assemblies.

NOTE

Notice that we didn’t attempt to load the .NET runtime BCL assemblies into our own ALC. These system assemblies are not designed to run outside the default ALC, and attempts to load them into your own ALC can result in incorrect behavior, performance degradation, and unexpected type incompatibility.

Here’s how we could use our custom ALC to load the *foo.dll* assembly in *c:\temp*:

```
var alc = new FolderBasedALC (@"c:\temp");
Assembly foo = alc.LoadFromAssemblyPath (@"c:\temp\foo.dll");
...
```

When we subsequently begin calling code in the `foo` assembly, the CLR will at some point need to resolve the dependency on *bar.dll*. This is when the custom ALC's `Load` method will fire and successfully locate the *bar.dll* assembly in *c:\temp*.

In this case, our `Load` method is also capable of resolving *foo.dll*, so we could simplify our code to this:

```
var alc = new FolderBasedALC (@"c:\temp");
Assembly foo = alc.LoadFromAssemblyName (new AssemblyName ("foo"));
...
```

Now, let's consider an alternative solution: instead of subclassing `AssemblyLoadContext` and overriding `Load`, we could instantiate a plain `AssemblyLoadContext` and handle its `Resolving` event:

```
var alc = new AssemblyLoadContext ("test");
alc.Resolving += (loadContext, assemblyName) =>
{
  string targetPath = Path.Combine (@"c:\temp", assemblyName.Name + ".dll");
  return alc.LoadFromAssemblyPath (targetPath);   // Load the assembly
};
Assembly foo = alc.LoadFromAssemblyName (new AssemblyName ("foo"));
```

Notice now that we don't need to check whether the assembly exists. Because the `Resolving` event fires *after* the default ALC has had a chance to resolve the assembly (and only when it fails), our handler won't fire for the .NET BCL assemblies. This makes this solution simpler, although there's a disadvantage. Remember that in our scenario, the main application knew nothing about *foo.dll* or *bar.dll* at compile time. This means that it's possible for the main application to itself depend on assemblies called *foo.dll* or *bar.dll*. If this were to occur, the `Resolving` event would never fire, and the application's `foo` and `bar` assemblies would load instead. In other words, we would fail to achieve *isolation*.

NOTE

Our `FolderBasedALC` class is good for illustrating the concept of assembly resolution, but it's of less use in real life because it cannot handle platform-specific and (for library projects) development-time NuGet dependencies. In "AssemblyDependencyResolver", we describe the solution to this problem, and in "Writing a Plug-In System", we give a detailed example.

The Default ALC

When an application starts, the CLR assigns a special ALC to the static `AssemblyLoadContext.Default` property. The default ALC is where your startup assembly loads, along with its statically referenced dependencies and the .NET runtime BCL assemblies.

The default ALC looks first in the *default probing* paths to automatically resolve assemblies (see "Default probing"); this normally equates to the locations indicated in the application's *.deps.json* and *.runtimeconfig.json* files.

If the ALC cannot find an assembly in its default probing paths, its `Resolving` event fires. Handling this event lets you load the assembly from other locations, which means that you can deploy an application's dependencies to additional locations, such as subfolders, shared folders, or even as a binary resource inside the host assembly:

```
AssemblyLoadContext.Default.Resolving += (loadContext, assemblyName) =>
{
  // Try to locate assemblyName, returning an Assembly object or null.
  // Typically you'd call LoadFromAssemblyPath after finding the file.
  // ...
};
```

The `Resolving` event in the default ALC also fires when a custom ALC fails to resolve (in other words, when its `Load` method returns `null`) and the default ALC is unable to resolve the assembly.

You can also load assemblies into the default ALC from outside the `Resolving` event. Before proceeding, however, you should first determine whether you can solve the problem better by using a separate ALC or with the approaches we describe in the following section (which use the *executing* and *contextual* ALCs). Hardcoding to the default ALC makes your code brittle because it cannot as a whole be isolated (for instance, by unit testing frameworks or by LINQPad).

If you still want to proceed, it's preferable to call a *resolution method* (i.e., `LoadFromAssemblyName`) rather than a *loading method* (such as `LoadFromAssemblyPath`)—especially if your assembly is statically referenced. This is because it's possible that the assembly might already be loaded, in which case `LoadFromAssemblyName` will return the already-loaded assembly, whereas `LoadFromAssemblyPath` will throw an exception.

(With `LoadFromAssemblyPath`, you can also run the risk of loading the assembly from a place that's inconsistent with where the ALC's default resolution mechanism would find it.)

If the assembly is in a place where the ALC won't automatically find it, you can still follow this procedure and additionally handle the ALC's `Resolving` event.

Note that when calling `LoadFromAssemblyName`, you don't need to provide the full name; the simple name will do (and is valid even if the assembly is strongly named):

```
AssemblyLoadContext.Default.LoadFromAssemblyName ("System.Xml");
```

However, if you include the public key token in the name, it must match with what's loaded.

Default probing

The default probing paths normally comprise the following:

- Paths specified in *AppName.deps.json* (where *AppName* is the name of your application's main assembly). If this file is not present, the application base folder is used instead.
- Folders containing the .NET runtime system assemblies (if your application is Framework-dependent).

MSBuild automatically generates a file called *AppName.deps.json*, which describes where to find all of its dependencies. These include platform-agnostic assemblies, which are placed in the application base folder, and platform-specific assemblies, which are placed in the *runtimes* subdirectory under a subfolder such as *win* or *unix*.

The paths specified in the generated *.deps.json* file are relative to the application base folder—or any additional folders that you specify in the `additionalProbingPaths` section of the *AppName.runtimeconfig.json* and/or *AppName.runtimeconfig.dev.json* configuration files (the latter is intended only for the development environment).

The "Current" ALC

In the preceding section, we cautioned against explicitly loading assemblies into the default ALC. What you usually want, instead, is to load/resolve into the "current" ALC.

In most cases, the "current" ALC is the one containing the currently executing assembly:

```
var executingAssem = Assembly.GetExecutingAssembly();
var alc = AssemblyLoadContext.GetLoadContext (executingAssem);

Assembly assem = alc.LoadFromAssemblyName (...);  // to resolve by name
        // OR: = alc.LoadFromAssemblyPath (...);  // to load by path
```

Here’s a more flexible and explicit way to obtain the ALC:

```
var myAssem = typeof (SomeTypeInMyAssembly).Assembly;
var alc = AssemblyLoadContext.GetLoadContext (myAssem);
...
```

Sometimes, it’s impossible to infer the “current” ALC. For example, suppose that you were responsible for writing the .NET binary serializer (we describe serialization in the online supplement at *http://www.albahari.com/nutshell*). A serializer such as this writes the full names of the types that it serializes (including their assembly names), which must be *resolved* during deserialization. The question is, which ALC should you use? The problem with relying on the executing assembly is that it will return whatever assembly contains the deserializer, not the assembly that’s *calling* the deserializer.

The best solution is not to guess but to ask:

```
public object Deserialize (Stream stream, AssemblyLoadContext alc)
{
  ...
}
```

Being explicit maximizes flexibility and minimizes the chance of making mistakes. The caller can now decide what should count as the “current” ALC:

```
var assem = typeof (SomeTypeThatIWillBeDeserializing).Assembly;
var alc = AssemblyLoadContext.GetLoadContext (assem);
var object = Deserialize (someStream, alc);
```

Assembly.Load and Contextual ALCs

To help with the common case of loading an assembly into the currently executing ALC; that is:

```
var executingAssem = Assembly.GetExecutingAssembly();
var alc = AssemblyLoadContext.GetLoadContext (executingAssem);
Assembly assem = alc.LoadFromAssemblyName (...);
```

Microsoft has defined the following method in the `Assembly` class:

```
public static Assembly Load (string assemblyString);
```

as well as a functionally identical version that accepts an `AssemblyName` object:

```
public static Assembly Load (AssemblyName assemblyRef);
```

(Don't confuse these methods with the legacy `Load(byte[])` method, which behaves in a totally different manner—see "The Legacy Loading Methods".)

As with `LoadFromAssemblyName`, you have a choice of specifying the assembly's simple, partial, or full name:

```
Assembly a = Assembly.Load ("System.Private.Xml");
```

This loads the `System.Private.Xml` assembly into whatever ALC the *executing code's assembly* is loaded in.

In this case, we specified a simple name. The following strings would also be valid, and all would have the same result in .NET:

```
"System.Private.Xml, PublicKeyToken=cc7b13ffcd2ddd51"
"System.Private.Xml, Version=4.0.1.0"
"System.Private.Xml, Version=4.0.1.0, PublicKeyToken=cc7b13ffcd2ddd51"
```

If you choose to specify a public key token, it must match with what's loaded.

NOTE

The Microsoft Developer Network (MSDN) cautions against loading an assembly from a partial name, recommending that you specify the exact version and public key token. Their rationale is based on factors relevant to .NET Framework, such as the effects of the Global Assembly Cache and Code Access Security. In .NET 5+ and .NET Core, these factors aren't present, and it's generally safe to load from a simple or partial name.

Both of these methods are strictly for *resolution*, so you cannot specify a file path. (If you populate the `CodeBase` property in the `AssemblyName` object, it will be ignored.)

WARNING

Don't fall into the trap of using `Assembly.Load` to load a statically referenced assembly. All you need do in this case is refer to a type in the assembly and obtain the assembly from that:

```
Assembly a = typeof (System.Xml.Formatting).Assembly;
```

Or, you could even do this:

```
Assembly a = System.Xml.Formatting.Indented.GetType().Assembly;
```

This prevents hardcoding the assembly name (which you might change in the future) while triggering assembly resolution on the *executing code's* ALC (as would happen with `Assembly.Load`).

If you were to write the `Assembly.Load` method yourself, it would (almost) look like this:

```
[MethodImpl(MethodImplOptions.NoInlining)]
Assembly Load (string name)
{
  Assembly callingAssembly = Assembly.GetCallingAssembly();
```

```
    var callingAlc = AssemblyLoadContext.GetLoadContext (callingAssembly);
    return callingAlc.LoadFromAssemblyName (new AssemblyName (name));
  }
```

EnterContextualReflection

`Assembly.Load`'s strategy of using the calling assembly's ALC context fails when `Assembly.Load` is called via an intermediary, such as a deserializer or unit test runner. If the intermediary is defined in a different assembly, the intermediary's load context is used instead of the caller's load context.

NOTE

We described this scenario earlier, when we talked about how you might write a deserializer. In such cases, the ideal solution is to force the caller to specify an ALC rather than inferring it with `Assembly.Load(string)`.

But because .NET 5+ and .NET Core evolved from .NET Framework—where isolation was accomplished with application domains rather than ALCs—the ideal solution is not prevalent, and `Assembly.Load(string)` is sometimes used inappropriately in scenarios in which the ALC cannot be reliably inferred. An example is the .NET binary serializer.

To allow `Assembly.Load` to still work in such scenarios, Microsoft has added a method to `AssemblyLoadContext` called `EnterContextualReflection`. This assigns an ALC to `AssemblyLoadContext.CurrentContextualReflectionContext`. Although this is a static property, its value is stored in an `AsyncLocal` variable, so it can hold separate values on different threads (but still be preserved throughout asynchronous operations).

If this property is non-null, `Assembly.Load` automatically uses it in preference to the calling ALC:

```
Method1();

var myALC = new AssemblyLoadContext ("test");
using (myALC.EnterContextualReflection())
```

```
{
   Console.WriteLine (
     AssemblyLoadContext.CurrentContextualReflectionContext.Name);  // test

   Method2();
}

// Once disposed, EnterContextualReflection() no longer has an effect.
Method3();

void Method1() => Assembly.Load ("...");    // Will use calling ALC
void Method2() => Assembly.Load ("...");    // Will use myALC
void Method3() => Assembly.Load ("...");    // Will use calling ALC
```

We previously demonstrated how you could write a method that's functionally similar to `Assembly.Load`. Here's a more accurate version that takes the contextual reflection context into account:

```
[MethodImpl(MethodImplOptions.NoInlining)]
Assembly Load (string name)
{
  var alc = AssemblyLoadContext.CurrentContextualReflectionContext
     ?? AssemblyLoadContext.GetLoadContext (Assembly.GetCallingAssembly());

  return alc.LoadFromAssemblyName (new AssemblyName (name));
}
```

Even though the contextual reflection context can be useful in allowing legacy code to run, a more robust solution (as we described earlier) is to modify the code that calls `Assembly.Load` so that it instead calls `LoadFromAssemblyName` on an ALC that's passed in by the caller.

NOTE

.NET Framework has no equivalent of `EnterContextualReflection`—and does not need it—despite having the same `Assembly.Load` methods. This is because with .NET Framework, isolation is accomplished primarily with *application domains* rather than ALCs. Application domains provide a stronger isolation model whereby each application domain has its own default load context, so isolation can still work even when only the default load context is used.

Loading and Resolving Unmanaged Libraries

ALCs can also load and resolve native libraries. Native resolution is triggered when you call an external method that's marked with the `[DllImport]` attribute:

```
[DllImport ("SomeNativeLibrary.dll")]
static extern int SomeNativeMethod (string text);
```

Because we didn't specify a full path in the `[DllImport]` attribute, calling `SomeNativeMethod` triggers a resolution in whatever ALC contains the assembly in which `SomeNativeMethod` is defined.

The virtual *resolving* method in the ALC is called `LoadUnmanagedDll`, and the *loading* method is called `LoadUnmanagedDllFromPath`:

```
protected override IntPtr LoadUnmanagedDll (string unmanagedDllName)
{
  // Locate the full path of unmanagedDllName...
  string fullPath = ...
  return LoadUnmanagedDllFromPath (fullPath);    // Load the DLL
}
```

If you're unable to locate the file, you can return `IntPtr.Zero`. The CLR will then fire the ALC's `ResolvingUnmanagedDll` event.

Interestingly, the `LoadUnmanagedDllFromPath` method is protected, so you won't usually be able to call it from a `ResolvingUnmanagedDll` event handler. However, you can achieve the same result by calling the static `NativeLibrary.Load`:

```
someALC.ResolvingUnmanagedDll += (requestingAssembly, unmanagedDllName) =>
{
  return NativeLibrary.Load ("(full path to unmanaged DLL)");
};
```

Although native libraries are typically resolved and loaded by ALCs, they don't "belong" to an ALC. After it's loaded, a native library stands on its own and takes responsibility for resolving any transitive dependencies that it might have. Furthermore, native libraries are global to the process, so it's not possible to load two different versions of a native library if they have the same filename.

AssemblyDependencyResolver

In "Default probing", we said that the default ALC reads the *.deps.json* and *.runtimeconfig.json* files, if present, in determining where to look to resolve platform-specific and development-time NuGet dependencies.

If you want to load an assembly into a custom ALC that has platform-specific or NuGet dependencies, you'll need to somehow reproduce this logic. You could accomplish this by parsing the configuration files and carefully following the rules on platform-specific monikers, but doing so is not only difficult, but the code that you write will break if the rules change in a later version of .NET.

The `AssemblyDependencyResolver` class solves this problem. To use it, you instantiate it with the path of the assembly whose dependencies you want to probe:

```
var resolver = new AssemblyDependencyResolver (@"c:\temp\foo.dll");
```

Then, to find the path of a dependency, you call the `ResolveAssemblyToPath` method:

```
string path = resolver.ResolveAssemblyToPath (new AssemblyName ("bar"));
```

In the absence of a *.deps.json* file (or if the *.deps.json* doesn't contain anything relevant to *bar.dll*), this will evaluate to *c:\temp\bar.dll*.

You can similarly resolve unmanaged dependencies by calling `ResolveUnmanagedDllToPath`.

A great way to illustrate a more complex scenario is to create a new Console project called `ClientApp` and then add a NuGet reference to *Microsoft.Data.SqlClient*. Add the following class:

```
using Microsoft.Data.SqlClient;

namespace ClientApp
{
  public class Program
  {
    public static SqlConnection GetConnection() => new SqlConnection();
    static void Main() => GetConnection();   // Test that it resolves
  }
}
```

Now build the application and look in the output folder: you'll see a file called *Microsoft.Data.SqlClient.dll*. However, this file *never loads* when run, and attempting to explicitly load it throws an exception. The assembly that actually loads is located in the *runtimes\win* (or *runtimes/unix*) subfolder; the default ALC knows to load it because it parses the *ClientApp.deps.json* file.

If you were to try to load the *ClientApp.dll* assembly from another application, you'd need to write an ALC that can resolve its dependency, *Microsoft.Data.SqlClient.dll*. In doing so, it would be insufficient to merely look in the folder where *ClientApp.dll* is located (as we did in "Resolving assemblies"). Instead, you'd need to use `AssemblyDependencyResolver` to determine where that file is located for the platform in use:

```
string path = @"C:\source\ClientApp\bin\Debug\netcoreapp3.0\ClientApp.dll";
var resolver = new AssemblyDependencyResolver (path);
var sqlClient = new AssemblyName ("Microsoft.Data.SqlClient");
Console.WriteLine (resolver.ResolveAssemblyToPath (sqlClient));
```

On a Windows machine, this outputs the following:

```
C:\source\ClientApp\bin\Debug\netcoreapp3.0\runtimes\win\lib\netcoreapp2.1
\Microsoft.Data.SqlClient.dll
```

We give a complete example in "Writing a Plug-In System".

Unloading ALCs

In simple cases, it's possible to unload a nondefault `AssemblyLoadContext`, freeing memory and releasing file locks on the assemblies it loaded. For this to work, the ALC must have been instantiated with the `isCollectible` parameter `true`:

```
var alc = new AssemblyLoadContext ("test", isCollectible:true);
```

You can then call the `Unload` method on the ALC to initiate the unload process.

The unload model is cooperative rather than preemptive. If any methods in any of the ALC's assemblies are executing, the unload will be deferred until those methods finish.

The actual unload takes place during garbage collection; it will not take place if anything from outside the ALC has any (nonweak) reference to anything inside the ALC (including objects, types, and assemblies). It's not uncommon for APIs (including those in the .NET BCL) to cache objects in static fields or dictionaries—or subscribe to events—and this makes it easy to create references that will prevent an unload, especially if code in the ALC uses APIs outside its ALC in a nontrivial way. Determining the cause of a failed unload is difficult and requires the use of tools such as WinDbg.

The Legacy Loading Methods

If you're still using .NET Framework (or writing a library that targets .NET Standard and want to support .NET Framework), you won't be able to use the `AssemblyLoadContext` class. Loading is accomplished instead by using the following methods:

```
public static Assembly LoadFrom (string assemblyFile);
public static Assembly LoadFile (string path);
public static Assembly Load (byte[] rawAssembly);
```

`LoadFile` and `Load(byte[])` provide isolation, whereas `LoadFrom` does not.

Resolution is accomplished by handling the application domain's `AssemblyResolve` event, which works like the default ALC's `Resolving` event.

The `Assembly.Load(string)` method is also available to trigger resolution and works in a similar way.

LoadFrom

`LoadFrom` loads an assembly from a given path into the default ALC. It's a bit like calling `AssemblyLoadContext.Default.LoadFromAssemblyPath` except for the following:

- If an assembly with the same simple name is already present in the default ALC, `LoadFrom` returns that assembly rather than throwing an exception.
- If an assembly with the same simple name is *not* already present in the default ALC and a load takes place, the assembly is given a special "LoadFrom" status. This status affects the default ALC's resolution logic, in that should that assembly have any dependencies in the *same folder*, those dependencies will resolve automatically.

NOTE

.NET Framework has a *Global Assembly Cache* (GAC). If the assembly is present in the GAC, the CLR will always load from there instead. This applies to all three loading methods.

`LoadFrom`'s ability to automatically resolve transitive same-folder dependencies can be convenient—until it loads an assembly that it shouldn't. Because such scenarios can be difficult to debug, it can be better to use `Load(string)` or `LoadFile` and resolve transitive dependencies by handling the application domain's `AssemblyResolve` event. This gives you the power to decide how to resolve each assembly and allows for debugging (by creating a breakpoint inside the event handler).

LoadFile and Load(byte[])

`LoadFile` and `Load(byte[])` load an assembly from a given file path or byte array into a new ALC. Unlike `LoadFrom`, these methods provide isolation and let you load multiple versions of the same assembly. However, there are two caveats:

- Calling `LoadFile` again with the identical path will return the previously loaded assembly.
- In .NET Framework, both methods first check the GAC and load from there instead if the assembly is present.

With `LoadFile` and `Load(byte[])`, you end up with a separate ALC per assembly (caveats aside). This enables isolation, although it can make it more awkward to manage.

To resolve dependencies, you handle the `AppDomain`'s `Resolving` event, which fires on all ALCs:

```
AppDomain.CurrentDomain.AssemblyResolve += (sender, args) =>
{
  string fullAssemblyName = args.Name;
  // return an Assembly object or null
  ...
};
```

The `args` variable also includes a property called `RequestingAssembly`, which tells you which assembly triggered the resolution.

After locating the assembly, you can then call `Assembly.LoadFile` to load it.

NOTE

You can enumerate all of the assemblies that have been loaded into the current application domain with `AppDomain.CurrentDomain.GetAssemblies()`. This works in .NET 5+, too, where it's equivalent to the following:

```
AssemblyLoadContext.All.SelectMany (a => a.Assemblies)
```

Writing a Plug-In System

To fully demonstrate the concepts that we've covered in this section, let's write a plug-in system that uses unloadable ALCs to isolate each plug-in.

Our demo system will initially comprise three .NET projects:

Plugin.Common (library)

Defines an interface that plug-ins will implement

Capitalizer (library)

A plug-in that capitalizes text

Plugin.Host (console application)

Locates and invokes plug-ins

Let's assume that the projects reside in the following directories:

```
c:\source\PluginDemo\Plugin.Common
c:\source\PluginDemo\Capitalizer
c:\source\PluginDemo\Plugin.Host
```

All projects will reference the Plugin.Common library, and there will be no other interproject references.

NOTE

If Plugin.Host were to reference Capitalizer, we wouldn't be writing a plug-in system; the central idea is that the plug-ins are written by third parties after Plugin.Host and Plugin.Common have been published.

If you're using Visual Studio, it can be convenient to put all three projects into a single solution for the sake of this demo. If you do so, right-click the Plugin.Host project, choose Build Dependencies > Project Dependencies, and then tick the Capitalizer project. This forces Capitalizer to build when you run the Plugin.Host project, without adding a reference.

Plugin.Common

Let's begin with Plugin.Common. Our plug-ins will perform a very simple task, which is to transform a string. Here's how we'll define the interface:

```
namespace Plugin.Common
{
  public interface ITextPlugin
  {
    string TransformText (string input);
  }
}
```

That's all there is to Plugin.Common.

Capitalizer (plug-in)

Our Capitalizer plug-in will reference Plugin.Common and contain a single class. For now, we'll keep the logic simple so that the plug-in has no extra dependencies:

```
public class CapitalizerPlugin : Plugin.Common.ITextPlugin
{
  public string TransformText (string input) => input.ToUpper();
}
```

If you build both projects and look in Capitalizer's output folder, you'll see the following two assemblies:

```
Capitalizer.dll      // Our plug-in assembly
Plugin.Common.dll    // Referenced assembly
```

Plugin.Host

Plugin.Host is a console application with two classes. The first class is a custom ALC to load the plug-ins:

```
class PluginLoadContext : AssemblyLoadContext
{
  AssemblyDependencyResolver _resolver;

  public PluginLoadContext (string pluginPath, bool collectible)
    // Give it a friendly name to help with debugging:
    : base (name: Path.GetFileName (pluginPath), collectible)
  {
    // Create a resolver to help us find dependencies.
    _resolver = new AssemblyDependencyResolver (pluginPath);
  }

  protected override Assembly Load (AssemblyName assemblyName)
  {
    // See below
    if (assemblyName.Name == typeof (ITextPlugin).Assembly.GetName().Name)
      return null;

    string target = _resolver.ResolveAssemblyToPath (assemblyName);

    if (target != null)
      return LoadFromAssemblyPath (target);

    // Could be a BCL assembly. Allow the default context to resolve.
    return null;
  }

  protected override IntPtr LoadUnmanagedDll (string unmanagedDllName)
  {
    string path = _resolver.ResolveUnmanagedDllToPath (unmanagedDllName);

    return path == null
      ? IntPtr.Zero
      : LoadUnmanagedDllFromPath (path);
  }
}
```

In the constructor, we pass in the path to the main plug-in assembly as well as a flag to indicate whether we'd like the ALC to be collectible (so that it can be unloaded).

The `Load` method is where we handle dependency resolution. All plug-ins must reference Plugin.Common so that they can implement `ITextPlugin`. This means that the `Load` method will fire at some point to resolve Plugin.Common. We need to be careful because the plug-in's output folder is likely to contain not only *Capitalizer.dll* but also its own copy of *Plugin.Common.dll*. If we were to load this copy of *Plugin.Common.dll* into the `PluginLoadContext`, we'd end up with two copies of the assembly: one in the host's default context and one in the plug-in's `PluginLoadContext`. The assemblies would be incompatible, and the host would complain that the plug-in does not implement `ITextPlugin`!

To solve this, we check explicitly for this condition:

```
if (assemblyName.Name == typeof (ITextPlugin).Assembly.GetName().Name)
  return null;
```

Returning null allows the host's default ALC to instead resolve the assembly.

NOTE

Instead of returning null, we could return `typeof(ITextPlugin).Assembly`, and it would also work correctly. How can we be certain that `ITextPlugin` will resolve on the host's ALC and not on our `PluginLoadContext`? Remember that our `PluginLoadContext` class is defined in the `Plugin.Host` assembly. Therefore, any types that you statically reference from this class will trigger an assembly resolution on the ALC into which *its assembly*, `Plugin.Host`, was loaded.

After checking for the common assembly, we use `AssemblyDependencyResolver` to locate any private dependencies that the plug-in might have. (Right now, there will be none.)

Notice that we also override the `LoadUnamangedDll` method. This ensures that if the plug-in has any unmanaged dependencies, these will load correctly, too.

The second class to write in Plugin.Host is the main program itself. For simplicity, let's hardcode the path to our Capitalizer plug-in (in real life, you might discover the paths of plug-ins by looking for DLLs in known locations or reading from a configuration file):

```
class Program
{
  const bool UseCollectibleContexts = true;

  static void Main()
  {
    const string capitalizer = @"C:\source\PluginDemo\"
      + @"Capitalizer\bin\Debug\netcoreapp3.0\Capitalizer.dll";

    Console.WriteLine (TransformText ("big apple", capitalizer));
  }

  static string TransformText (string text, string pluginPath)
  {
    var alc = new PluginLoadContext (pluginPath, UseCollectibleContexts);
    try
    {
      Assembly assem = alc.LoadFromAssemblyPath (pluginPath);
```

```
        // Locate the type in the assembly that implements ITextPlugin:
        Type pluginType = assem.ExportedTypes.Single (t =>
                          typeof (ITextPlugin).IsAssignableFrom (t));

        // Instantiate the ITextPlugin implementation:
        var plugin = (ITextPlugin)Activator.CreateInstance (pluginType);

        // Call the TransformText method
        return plugin.TransformText (text);
      }
      finally
      {
        if (UseCollectibleContexts) alc.Unload();    // unload the ALC
      }
    }
  }
```

Let's look at the `TransformText` method. We first instantiate a new ALC for our plug-in and then ask it to load the main plug-in assembly. Next, we use Reflection to locate the type that implements `ITextPlugin` (we cover this in detail in Chapter 18). Then, we instantiate the plug-in, call the `TransformText` method, and unload the ALC.

NOTE

If you needed to call the `TransformText` method repeatedly, a better approach would be to cache the ALC rather than unloading it after each call.

Here's the output:

```
BIG APPLE
```

Adding dependencies

Our code is fully capable of resolving and isolating dependencies. To illustrate, let's first add a NuGet reference to *Humanizer.Core*, version 2.6.2. You can do this via the Visual Studio UI or by adding the following element to the *Capitalizer.csproj* file:

```
  <ItemGroup>
    <PackageReference Include="Humanizer.Core" Version="2.6.2" />
  </ItemGroup>
```

Now, modify `CapitalizerPlugin`, as follows:

```
using Humanizer;
namespace Capitalizer
{
  public class CapitalizerPlugin : Plugin.Common.ITextPlugin
  {
    public string TransformText (string input) => input.Pascalize();
  }
}
```

If you rerun the program, the output will now be this:

```
BigApple
```

Next, we create another plug-in called Pluralizer. Create a new .NET library project and add a NuGet reference to *Humanizer.Core*, version 2.7.9:

```
  <ItemGroup>
    <PackageReference Include="Humanizer.Core" Version="2.7.9" />
  </ItemGroup>
```

Now, add a class called `PluralizerPlugin`. This will be similar to `CapitalizerPlugIn`, but we call the `Pluralize` method instead:

```
using Humanizer;
namespace Pluralizer
{
  public class PluralizerPlugin : Plugin.Common.ITextPlugin
  {
    public string TransformText (string input) => input.Pluralize();
  }
}
```

Finally, we need to add code to the Plugin.Host's `Main` method to load and run the Pluralizer plug-in:

```
static void Main()
{
  const string capitalizer = @"C:\source\PluginDemo\"
    + @"Capitalizer\bin\Debug\netcoreapp3.0\Capitalizer.dll";

  Console.WriteLine (TransformText ("big apple", capitalizer));

  const string pluralizer = @"C:\source\PluginDemo\"
    + @"Pluralizer\bin\Debug\netcoreapp3.0\Pluralizer.dll";

  Console.WriteLine (TransformText ("big apple", pluralizer));
}
```

The output will now be like this:

```
BigApple
big apples
```

To fully see what's going on, change the `UseCollectibleContexts` constant to false and add the following code to the `Main` method to enumerate the ALCs and their assemblies:

```
foreach (var context in AssemblyLoadContext.All)
{
  Console.WriteLine ($"Context: {context.GetType().Name} {context.Name}");

  foreach (var assembly in context.Assemblies)
      Console.WriteLine ($"  Assembly: {assembly.FullName}");
}
```

In the output, you can see two different versions of Humanizer, each loaded into its own ALC:

```
Context: PluginLoadContext Capitalizer.dll
```

```
    Assembly: Capitalizer, Version=1.0.0.0, Culture=neutral, PublicKeyToken=...
    Assembly: Humanizer, Version=2.6.0.0, Culture=neutral, PublicKeyToken=...
  Context: PluginLoadContext Pluralizer.dll
    Assembly: Pluralizer, Version=1.0.0.0, Culture=neutral, PublicKeyToken=...
    Assembly: Humanizer, Version=2.7.0.0, Culture=neutral, PublicKeyToken=...
  Context: DefaultAssemblyLoadContext Default
    Assembly: System.Private.CoreLib, Version=4.0.0.0, Culture=neutral,...
    Assembly: Host, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
    ...
```

NOTE

Even if both plug-ins were to use the same version of Humanizer, the isolation of separate assemblies can still be beneficial because each will have its own static variables.