Contents

Assemblies and the Global Assembly Cache (C#)

1/23/2019 • 3 minutes to read • Edit Online

Assemblies form the fundamental unit of deployment, version control, reuse, activation scoping, and security permissions for a .NET-based application. Assemblies take the form of an executable (.exe) file or dynamic link library (.dll) file, and are the building blocks of the .NET Framework. They provide the common language runtime with the information it needs to be aware of type implementations. You can think of an assembly as a collection of types and resources that form a logical unit of functionality and are built to work together.

Assemblies can contain one or more modules. For example, larger projects may be planned in such a way that several individual developers work on separate modules, all coming together to create a single assembly. For more information about modules, see the topic How to: Build a Multifile Assembly.

Assemblies have the following properties:

- Assemblies are implemented as .exe or .dll files.
- You can share an assembly between applications by putting it in the global assembly cache. Assemblies must be strong-named before they can be included in the global assembly cache. For more information, see Strong-Named Assemblies.
- Assemblies are only loaded into memory if they are required. If they are not used, they are not loaded. This means that assemblies can be an efficient way to manage resources in larger projects.
- You can programmatically obtain information about an assembly by using reflection. For more information, see Reflection (C#).
- If you want to load an assembly only to inspect it, use a method such as ReflectionOnlyLoadFrom.

Assembly Manifest

Within every assembly, there is an *assembly manifest*. Similar to a table of contents, the assembly manifest contains the following:

- The assembly's identity (its name and version).
- A file table describing all the other files that make up the assembly, for example, any other assemblies you created that your .exe or .dll file relies on, or even bitmap or Readme files.
- An *assembly reference list*, which is a list of all external dependencies—.dlls or other files your application needs that may have been created by someone else. Assembly references contain references to both global and private objects. Global objects reside in the global assembly cache, an area available to other applications. Private objects must be in a directory at either the same level as or below the directory in which your application is installed.

Because assemblies contain information about content, versioning, and dependencies, the applications you create with C# do not rely on Windows registry values to function properly. Assemblies reduce .dll conflicts and make your applications more reliable and easier to deploy. In many cases, you can install a .NET-based application simply by copying its files to the target computer.

For more information see Assembly Manifest.

Adding a Reference to an Assembly

To use an assembly, you must add a reference to it. Next, you use the using directive to choose the namespace of the items you want to use. Once an assembly is referenced and imported, all the accessible classes, properties, methods, and other members of its namespaces are available to your application as if their code were part of your source file.

In C#, you can also use two versions of the same assembly in a single application. For more information, see extern alias.

Creating an Assembly

Compile your application by clicking **Build** on the **Build** menu or by building it from the command line using the command-line compiler. For details about building assemblies from the command line, see Command-line Building With csc.exe.

> **NOTE**
>
> To build an assembly in Visual Studio, on the **Build** menu choose **Build**.

See also

- C# Programming Guide
- Assemblies in the Common Language Runtime
- Friend Assemblies (C#)
- How to: Share an Assembly with Other Applications (C#)
- How to: Load and Unload Assemblies (C#)
- How to: Determine If a File Is an Assembly (C#)
- How to: Create and Use Assemblies Using the Command Line (C#)
- Walkthrough: Embedding Types from Managed Assemblies in Visual Studio (C#)
- Walkthrough: Embedding Type Information from Microsoft Office Assemblies in Visual Studio (C#)

Friend Assemblies (C#)

1/23/2019 • 3 minutes to read • Edit Online

A *friend assembly* is an assembly that can access another assembly's internal types and members. If you identify an assembly as a friend assembly, you no longer have to mark types and members as public in order for them to be accessed by other assemblies. This is especially convenient in the following scenarios:

- During unit testing, when test code runs in a separate assembly but requires access to members in the assembly being tested that are marked as `internal` .
- When you are developing a class library and additions to the library are contained in separate assemblies but require access to members in existing assemblies that are marked as `internal` .

Remarks

You can use the InternalsVisibleToAttribute attribute to identify one or more friend assemblies for a given assembly. The following example uses the InternalsVisibleToAttribute attribute in assembly A and specifies assembly `AssemblyB` as a friend assembly. This gives assembly `AssemblyB` access to all types and members in assembly A that are marked as `internal` .

> **NOTE**
>
> When you compile an assembly (assembly `AssemblyB`) that will access internal types or internal members of another assembly (assembly *A*), you must explicitly specify the name of the output file (.exe or .dll) by using the **/out** compiler option. This is required because the compiler has not yet generated the name for the assembly it is building at the time it is binding to external references. For more information, see /out (C#) .

```
using System.Runtime.CompilerServices;
using System;

[assembly: InternalsVisibleTo("AssemblyB")]

// The class is internal by default.
class FriendClass
{
    public void Test()
    {
        Console.WriteLine("Sample Class");
    }
}

// Public class that has an internal method.
public class ClassWithFriendMethod
{
    internal void Test()
    {
        Console.WriteLine("Sample Method");
    }

}
```

Only assemblies that you explicitly specify as friends can access `internal` types and members. For example, if assembly B is a friend of assembly A and assembly C references assembly B, C does not have access to `internal` types in A.

The compiler performs some basic validation of the friend assembly name passed to the InternalsVisibleToAttribute attribute. If assembly *A* declares *B* as a friend assembly, the validation rules are as follows:

- If assembly *A* is strong named, assembly *B* must also be strong named. The friend assembly name that is passed to the attribute must consist of the assembly name and the public key of the strong-name key that is used to sign assembly *B*.

  The friend assembly name that is passed to the InternalsVisibleToAttribute attribute cannot be the strong name of assembly *B*: do not include the assembly version, culture, architecture, or public key token.

- If assembly *A* is not strong named, the friend assembly name should consist of only the assembly name. For more information, see How to: Create Unsigned Friend Assemblies (C#).

- If assembly *B* is strong named, you must specify the strong-name key for assembly *B* by using the project setting or the command-line `/keyfile` compiler option. For more information, see How to: Create Signed Friend Assemblies (C#).

The StrongNameIdentityPermission class also provides the ability to share types, with the following differences:

- StrongNameIdentityPermission applies to an individual type, while a friend assembly applies to the whole assembly.

- If there are hundreds of types in assembly *A* that you want to share with assembly *B*, you have to add StrongNameIdentityPermission to all of them. If you use a friend assembly, you only need to declare the friend relationship once.

- If you use StrongNameIdentityPermission, the types you want to share have to be declared as public. If you use a friend assembly, the shared types are declared as `internal`.

For information about how to access an assembly's `internal` types and methods from a module file (a file with the .netmodule extension), see /moduleassemblyname (C#).

See also

- InternalsVisibleToAttribute
- StrongNameIdentityPermission
- How to: Create Unsigned Friend Assemblies (C#)
- How to: Create Signed Friend Assemblies (C#)
- Assemblies and the Global Assembly Cache (C#)
- C# Programming Guide

How to: Create Unsigned Friend Assemblies (C#)

1/23/2019 • 2 minutes to read • Edit Online

This example shows how to use friend assemblies with assemblies that are unsigned.

To create an assembly and a friend assembly

1. Open a command prompt.
2. Create a C# file named `friend_unsigned_A.` that contains the following code. The code uses the InternalsVisibleToAttribute attribute to declare friend_unsigned_B as a friend assembly.

```
// friend_unsigned_A.cs
// Compile with:
// csc /target:library friend_unsigned_A.cs
using System.Runtime.CompilerServices;
using System;

[assembly: InternalsVisibleTo("friend_unsigned_B")]

// Type is internal by default.
class Class1
{
    public void Test()
    {
        Console.WriteLine("Class1.Test");
    }
}

// Public type with internal member.
public class Class2
{
    internal void Test()
    {
        Console.WriteLine("Class2.Test");
    }
}
```

3. Compile and sign friend_unsigned_A by using the following command.

```
csc /target:library friend_unsigned_A.cs
```

4. Create a C# file named `friend_unsigned_B` that contains the following code. Because friend_unsigned_A specifies friend_unsigned_B as a friend assembly, the code in friend_unsigned_B can access `internal` types and members from friend_unsigned_A.

```
// friend_unsigned_B.cs
// Compile with:
// csc /r:friend_unsigned_A.dll /out:friend_unsigned_B.exe friend_unsigned_B.cs
public class Program
{
    static void Main()
    {
        // Access an internal type.
        Class1 inst1 = new Class1();
        inst1.Test();

        Class2 inst2 = new Class2();
        // Access an internal member of a public type.
        inst2.Test();

        System.Console.ReadLine();
    }
}
```

5. Compile friend_unsigned_B by using the following command.

   ```
   csc /r:friend_unsigned_A.dll /out:friend_unsigned_B.exe friend_unsigned_B.cs
   ```

   The name of the assembly that is generated by the compiler must match the friend assembly name that is passed to the InternalsVisibleToAttribute attribute. You must explicitly specify the name of the output assembly (.exe or .dll) by using the `/out` compiler option. For more information, see /out (C# Compiler Options).

6. Run the friend_unsigned_B.exe file.

   The program prints two strings: "Class1.Test" and "Class2.Test".

.NET Framework Security

There are similarities between the InternalsVisibleToAttribute attribute and the StrongNameIdentityPermission class. The main difference is that StrongNameIdentityPermission can demand security permissions to run a particular section of code, whereas the InternalsVisibleToAttribute attribute controls the visibility of `internal` types and members.

See also

- InternalsVisibleToAttribute
- Assemblies and the Global Assembly Cache (C#)
- Friend Assemblies (C#)
- How to: Create Signed Friend Assemblies (C#)
- C# Programming Guide

How to: Create Signed Friend Assemblies (C#)

1/23/2019 • 2 minutes to read • Edit Online

This example shows how to use friend assemblies with assemblies that have strong names. Both assemblies must be strong named. Although both assemblies in this example use the same keys, you could use different keys for two assemblies.

To create a signed assembly and a friend assembly

1. Open a command prompt.
2. Use the following sequence of commands with the Strong Name tool to generate a keyfile and to display its public key. For more information, see Sn.exe (Strong Name Tool).

   a. Generate a strong-name key for this example and store it in the file FriendAssemblies.snk:

   ```
   sn -k FriendAssemblies.snk
   ```

   b. Extract the public key from FriendAssemblies.snk and put it into FriendAssemblies.publickey:

   ```
   sn -p FriendAssemblies.snk FriendAssemblies.publickey
   ```

   c. Display the public key stored in the file FriendAssemblies.publickey:

   ```
   sn -tp FriendAssemblies.publickey
   ```

3. Create a C# file named `friend_signed_A` that contains the following code. The code uses the InternalsVisibleToAttribute attribute to declare friend_signed_B as a friend assembly.

   The Strong Name tool generates a new public key every time it runs. Therefore, you must replace the public key in the following code with the public key you just generated, as shown in the following example.

   ```
   // friend_signed_A.cs
   // Compile with:
   // csc /target:library /keyfile:FriendAssemblies.snk friend_signed_A.cs
   using System.Runtime.CompilerServices;

   [assembly: InternalsVisibleTo("friend_signed_B,
   PublicKey=0024000004800000940000000602000000240000525341310004000001000100e3aedce99b7e10823920206f8e46c
   d5558b4ec7345bd1a5b201ffe71660625dcb8f9a08687d881c8f65a0dcf042f81475d2e88f3e3e273c8311ee40f952db306c02f
   bfc5d8bc6ee1e924e6ec8fe8c01932e0648a0d3e5695134af3bb7fab370d3012d083fa6b83179dd3d031053f72fc1f7da845914
   0b0af5afc4d2804deccb6")]
   class Class1
   {
       public void Test()
       {
           System.Console.WriteLine("Class1.Test");
           System.Console.ReadLine();
       }
   }
   ```

4. Compile and sign friend_signed_A by using the following command.

   ```
   csc /target:library /keyfile:FriendAssemblies.snk friend_signed_A.cs
   ```

5. Create a C# file that is named `friend_signed_B` and contains the following code. Because friend_signed_A specifies friend_signed_B as a friend assembly, the code in friend_signed_B can access `internal` types and

members from friend_signed_A. The file contains the following code.

```
// friend_signed_B.cs
// Compile with:
// csc /keyfile:FriendAssemblies.snk /r:friend_signed_A.dll /out:friend_signed_B.exe friend_signed_B.cs
public class Program
{
    static void Main()
    {
        Class1 inst = new Class1();
        inst.Test();
    }
}
```

6. Compile and sign friend_signed_B by using the following command.

```
csc /keyfile:FriendAssemblies.snk /r:friend_signed_A.dll /out:friend_signed_B.exe friend_signed_B.cs
```

   The name of the assembly generated by the compiler must match the friend assembly name passed to the InternalsVisibleToAttribute attribute. You must explicitly specify the name of the output assembly (.exe or .dll) by using the `/out` compiler option. For more information, see /out (C# Compiler Options).

7. Run the friend_signed_B.exe file.

   The program prints the string "Class1.Test".

.NET Framework Security

There are similarities between the InternalsVisibleToAttribute attribute and the StrongNameIdentityPermission class. The main difference is that StrongNameIdentityPermission can demand security permissions to run a particular section of code, whereas the InternalsVisibleToAttribute attribute controls the visibility of `internal` types and members.

See also

- InternalsVisibleToAttribute
- Assemblies and the Global Assembly Cache (C#)
- Friend Assemblies (C#)
- How to: Create Unsigned Friend Assemblies (C#)
- /keyfile
- Sn.exe (Strong Name Tool)
- Creating and Using Strong-Named Assemblies
- C# Programming Guide

How to: Create and Use Assemblies Using the Command Line (C#)

1/23/2019 • 2 minutes to read • Edit Online

An assembly, or a dynamic linking library (DLL), is linked to your program at run time. To demonstrate building and using a DLL, consider the following scenario:

- `MathLibrary.DLL`: The library file that contains the methods to be called at run time. In this example, the DLL contains two methods, `Add` and `Multiply`.
- `Add`: The source file that contains the method `Add`. It returns the sum of its parameters. The class `AddClass` that contains the method `Add` is a member of the namespace `UtilityMethods`.
- `Mult`: The source code that contains the method `Multiply`. It returns the product of its parameters. The class `MultiplyClass` that contains the method `Multiply` is also a member of the namespace `UtilityMethods`.
- `TestCode`: The file that contains the `Main` method. It uses the methods in the DLL file to calculate the sum and the product of the run-time arguments.

Example

```
// File: Add.cs
namespace UtilityMethods
{
    public class AddClass
    {
        public static long Add(long i, long j)
        {
            return (i + j);
        }
    }
}
```

```
// File: Mult.cs
namespace UtilityMethods
{
    public class MultiplyClass
    {
        public static long Multiply(long x, long y)
        {
            return (x * y);
        }
    }
}
```

```
// File: TestCode.cs

using UtilityMethods;

class TestCode
{
    static void Main(string[] args)
    {
        System.Console.WriteLine("Calling methods from MathLibrary.DLL:");

        if (args.Length != 2)
        {
            System.Console.WriteLine("Usage: TestCode <num1> <num2>");
            return;
        }

        long num1 = long.Parse(args[0]);
        long num2 = long.Parse(args[1]);

        long sum = AddClass.Add(num1, num2);
        long product = MultiplyClass.Multiply(num1, num2);

        System.Console.WriteLine("{0} + {1} = {2}", num1, num2, sum);
        System.Console.WriteLine("{0} * {1} = {2}", num1, num2, product);
    }
}
/* Output (assuming 1234 and 5678 are entered as command-line arguments):
    Calling methods from MathLibrary.DLL:
    1234 + 5678 = 6912
    1234 * 5678 = 7006652
*/
```

This file contains the algorithm that uses the DLL methods, `Add` and `Multiply`. It starts with parsing the arguments entered from the command line, `num1` and `num2`. Then it calculates the sum by using the `Add` method on the `AddClass` class, and the product by using the `Multiply` method on the `MultiplyClass` class.

Notice that the `using` directive at the beginning of the file enables you to use the unqualified class names to reference the DLL methods at compile time, as follows:

```
MultiplyClass.Multiply(num1, num2);
```

Otherwise, you have to use the fully qualified names, as follows:

```
UtilityMethods.MultiplyClass.Multiply(num1, num2);
```

Execution

To run the program, enter the name of the EXE file, followed by two numbers, as follows:

```
TestCode 1234 5678
```

Compiling the Code

To build the file `MathLibrary.DLL`, compile the two files `Add` and `Mult` by using the following command line.

```
csc /target:library /out:MathLibrary.DLL Add.cs Mult.cs
```

The /target:library compiler option tells the compiler to output a DLL instead of an EXE file. The /out compiler

option followed by a file name is used to specify the DLL file name. Otherwise, the compiler uses the first file (`Add.cs`) as the name of the DLL.

To build the executable file, `TestCode.exe`, use the following command line:

```
csc /out:TestCode.exe /reference:MathLibrary.DLL TestCode.cs
```

The **/out** compiler option tells the compiler to output an EXE file and specifies the name of the output file (`TestCode.exe`). This compiler option is optional. The /reference compiler option specifies the DLL file or files that this program uses. For more information, see /reference.

For more information about building from the command line, see Command-line Building With csc.exe.

See also

- C# Programming Guide
- Assemblies and the Global Assembly Cache (C#)
- Creating a Class to Hold DLL Functions

How to: Determine If a File Is an Assembly (C#)

1/23/2019 • 2 minutes to read • Edit Online

A file is an assembly if and only if it is managed, and contains an assembly entry in its metadata. For more information on assemblies and metadata, see the topic Assembly Manifest.

How to manually determine if a file is an assembly

1. Start the Ildasm.exe (IL Disassembler).
2. Load the file you wish to test.
3. If **ILDASM** reports that the file is not a portable executable (PE) file, then it is not an assembly. For more information, see the topic How to: View Assembly Contents.

How to programmatically determine if a file is an assembly

1. Call the GetAssemblyName method, passing the full file path and name of the file you are testing.
2. If a BadImageFormatException exception is thrown, the file is not an assembly.

Example

This example tests a DLL to see if it is an assembly.

```
class TestAssembly
{
    static void Main()
    {

        try
        {
            System.Reflection.AssemblyName testAssembly =

System.Reflection.AssemblyName.GetAssemblyName(@"C:\Windows\Microsoft.NET\Framework\v3.5\System.Net.dll");

            System.Console.WriteLine("Yes, the file is an assembly.");
        }

        catch (System.IO.FileNotFoundException)
        {
            System.Console.WriteLine("The file cannot be found.");
        }

        catch (System.BadImageFormatException)
        {
            System.Console.WriteLine("The file is not an assembly.");
        }

        catch (System.IO.FileLoadException)
        {
            System.Console.WriteLine("The assembly has already been loaded.");
        }
    }
}
/* Output (with .NET Framework 3.5 installed):
    Yes, the file is an assembly.
*/
```

The GetAssemblyName method loads the test file, and then releases it once the information is read.

See also

- AssemblyName
- C# Programming Guide
- Assemblies and the Global Assembly Cache (C#)

How to: Load and Unload Assemblies (C#)

1/23/2019 • 2 minutes to read • Edit Online

The assemblies referenced by your program will automatically be loaded at build time, but it is also possible to load specific assemblies into the current application domain at runtime. For more information, see How to: Load Assemblies into an Application Domain.

There is no way to unload an individual assembly without unloading all of the application domains that contain it. Even if the assembly goes out of scope, the actual assembly file will remain loaded until all application domains that contain it are unloaded.

If you want to unload some assemblies but not others, consider creating a new application domain, executing the code inside that domain, and then unloading that application domain. For more information, see How to: Unload an Application Domain.

To load an assembly into an application domain

1. Use one of the several load methods contained in the classes AppDomain and System.Reflection. For more information, see How to: Load Assemblies into an Application Domain.

To unload an application domain

1. There is no way to unload an individual assembly without unloading all of the application domains that contain it. Use the `Unload` method from AppDomain to unload the application domains. For more information, see How to: Unload an Application Domain.

See also

- C# Programming Guide
- Assemblies and the Global Assembly Cache (C#)
- How to: Load Assemblies into an Application Domain

How to: Share an Assembly with Other Applications (C#)

1/23/2019 • 2 minutes to read • Edit Online

Assemblies can be private or shared: by default, most simple programs consist of a private assembly because they are not intended to be used by other applications.

In order to share an assembly with other applications, it must be placed in the Global Assembly Cache (GAC).

Sharing an assembly

1. Create your assembly. For more information, see Creating Assemblies.
2. Assign a strong name to your assembly. For more information, see How to: Sign an Assembly with a Strong Name.
3. Assign version information to your assembly. For more information, see Assembly Versioning.
4. Add your assembly to the Global Assembly Cache. For more information, see How to: Install an Assembly into the Global Assembly Cache.
5. Access the types contained in the assembly from the other applications. For more information, see How to: Reference a Strong-Named Assembly.

See also

- C# Programming Guide
- Programming with Assemblies

Walkthrough: Embedding Types from Managed Assemblies in Visual Studio (C#)

1/23/2019 • 8 minutes to read • Edit Online

If you embed type information from a strong-named managed assembly, you can loosely couple types in an application to achieve version independence. That is, your program can be written to use types from multiple versions of a managed library without having to be recompiled for each version.

Type embedding is frequently used with COM interop, such as an application that uses automation objects from Microsoft Office. Embedding type information enables the same build of a program to work with different versions of Microsoft Office on different computers. However, you can also use type embedding with a fully managed solution.

Type information can be embedded from an assembly that has the following characteristics:

- The assembly exposes at least one public interface.
- The embedded interfaces are annotated with a `ComImport` attribute and a `Guid` attribute (and a unique GUID).
- The assembly is annotated with the `ImportedFromTypeLib` attribute or the `PrimaryInteropAssembly` attribute, and an assembly-level `Guid` attribute. (By default, Visual C# project templates include an assembly-level `Guid` attribute.)

After you have specified the public interfaces that can be embedded, you can create runtime classes that implement those interfaces. A client program can then embed the type information for those interfaces at design time by referencing the assembly that contains the public interfaces and setting the `Embed Interop Types` property of the reference to `True`. This is equivalent to using the command line compiler and referencing the assembly by using the `/link` compiler option. The client program can then load instances of your runtime objects typed as those interfaces. If you create a new version of your strong-named runtime assembly, the client program does not have to be recompiled with the updated runtime assembly. Instead, the client program continues to use whichever version of the runtime assembly is available to it, using the embedded type information for the public interfaces.

Because the primary function of type embedding is to support embedding of type information from COM interop assemblies, the following limitations apply when you embed type information in a fully managed solution:

- Only attributes specific to COM interop are embedded; other attributes are ignored.
- If a type uses generic parameters and the type of the generic parameter is an embedded type, that type cannot be used across an assembly boundary. Examples of crossing an assembly boundary include calling a method from another assembly or a deriving a type from a type defined in another assembly.
- Constants are not embedded.
- The System.Collections.Generic.Dictionary<TKey,TValue> class does not support an embedded type as a key. You can implement your own dictionary type to support an embedded type as a key.

In this walkthrough, you will do the following:

- Create a strong-named assembly that has a public interface that contains type information that can be embedded.
- Create a strong-named runtime assembly that implements that public interface.

- Create a client program that embeds the type information from the public interface and creates an instance of the class from the runtime assembly.
- Modify and rebuild the runtime assembly.
- Run the client program to see that the new version of the runtime assembly is being used without having to recompile the client program.

> **NOTE**
>
> Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see Personalizing the IDE.

Creating an Interface

To create the type equivalence interface project

1. In Visual Studio, on the **File** menu, choose **New** and then click **Project**.
2. In the **New Project** dialog box, in the **Project Types** pane, make sure that **Windows** is selected. Select **Class Library** in the **Templates** pane. In the **Name** box, type `TypeEquivalenceInterface`, and then click **OK**. The new project is created.
3. In **Solution Explorer**, right-click the Class1.cs file and click **Rename**. Rename the file to `ISampleInterface.cs` and press ENTER. Renaming the file will also rename the class to `ISampleInterface`. This class will represent the public interface for the class.
4. Right-click the TypeEquivalenceInterface project and click **Properties**. Click the **Build** tab. Set the output path to a valid location on your development computer, such as `C:\TypeEquivalenceSample`. This location will also be used in a later step in this walkthrough.
5. While still editing the project properties, click the **Signing** tab. Select the **Sign the assembly** option. In the **Choose a strong name key file** list, click **<New...>**. In the **Key file name** box, type `key.snk`. Clear the **Protect my key file with a password** check box. Click **OK**.
6. Open the ISampleInterface.cs file. Add the following code to the ISampleInterface class file to create the ISampleInterface interface.

```
using System;
using System.Runtime.InteropServices;

namespace TypeEquivalenceInterface
{
    [ComImport]
    [Guid("8DA56996-A151-4136-B474-32784559F6DF")]
    public interface ISampleInterface
    {
        void GetUserInput();
        string UserInput { get; }
 }
}
```

7. On the **Tools** menu, click **Create Guid**. In the **Create GUID** dialog box, click **Registry Format** and then click **Copy**. Click **Exit**.
8. In the `Guid` attribute, delete the sample GUID and paste in the GUID that you copied from the **Create GUID** dialog box. Remove the braces ({}) from the copied GUID.

9. In **Solution Explorer**, expand the **Properties** folder. Double-click the AssemblyInfo.cs file. Add the following attribute to the file.

```
[assembly: ImportedFromTypeLib("")]
```

   Save the file.

10. Save the project.

11. Right-click the TypeEquivalenceInterface project and click **Build**. The class library .dll file is compiled and saved to the specified build output path (for example, C:\TypeEquivalenceSample).

Creating a Runtime Class

To create the type equivalence runtime project

1. In Visual Studio, on the **File** menu, point to **New** and then click **Project**.

2. In the **New Project** dialog box, in the **Project Types** pane, make sure that **Windows** is selected. Select **Class Library** in the **Templates** pane. In the **Name** box, type `TypeEquivalenceRuntime`, and then click **OK**. The new project is created.

3. In **Solution Explorer**, right-click the Class1.cs file and click **Rename**. Rename the file to `SampleClass.cs` and press ENTER. Renaming the file also renames the class to `SampleClass`. This class will implement the `ISampleInterface` interface.

4. Right-click the TypeEquivalenceRuntime project and click **Properties**. Click the **Build** tab. Set the output path to the same location you used in the TypeEquivalenceInterface project, for example, `C:\TypeEquivalenceSample`.

5. While still editing the project properties, click the **Signing** tab. Select the **Sign the assembly** option. In the **Choose a strong name key file** list, click **<New...>**. In the **Key file name** box, type `key.snk`. Clear the **Protect my key file with a password** check box. Click **OK**.

6. Right-click the TypeEquivalenceRuntime project and click **Add Reference**. Click the **Browse** tab and browse to the output path folder. Select the TypeEquivalenceInterface.dll file and click **OK**.

7. In **Solution Explorer**, expand the **References** folder. Select the TypeEquivalenceInterface reference. In the Properties window for the TypeEquivalenceInterface reference, set the **Specific Version** property to **False**.

8. Add the following code to the SampleClass class file to create the SampleClass class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using TypeEquivalenceInterface;

namespace TypeEquivalenceRuntime
{
    public class SampleClass : ISampleInterface
    {
        private string p_UserInput;
        public string UserInput { get { return p_UserInput; } }

        public void GetUserInput()
        {
            Console.WriteLine("Please enter a value:");
            p_UserInput = Console.ReadLine();
        }
    }
}
```

9. Save the project.
10. Right-click the TypeEquivalenceRuntime project and click **Build**. The class library .dll file is compiled and saved to the specified build output path (for example, C:\TypeEquivalenceSample).

Creating a Client Project

To create the type equivalence client project

1. In Visual Studio, on the **File** menu, point to **New** and then click **Project**.
2. In the **New Project** dialog box, in the **Project Types** pane, make sure that **Windows** is selected. Select **Console Application** in the **Templates** pane. In the **Name** box, type `TypeEquivalenceClient`, and then click **OK**. The new project is created.
3. Right-click the TypeEquivalenceClient project and click **Properties**. Click the **Build** tab. Set the output path to the same location you used in the TypeEquivalenceInterface project, for example, `C:\TypeEquivalenceSample`.
4. Right-click the TypeEquivalenceClient project and click **Add Reference**. Click the **Browse** tab and browse to the output path folder. Select the TypeEquivalenceInterface.dll file (not the TypeEquivalenceRuntime.dll) and click **OK**.
5. In **Solution Explorer**, expand the **References** folder. Select the TypeEquivalenceInterface reference. In the Properties window for the TypeEquivalenceInterface reference, set the **Embed Interop Types** property to **True**.
6. Add the following code to the Program.cs file to create the client program.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using TypeEquivalenceInterface;
using System.Reflection;

namespace TypeEquivalenceClient
{
    class Program
    {
        static void Main(string[] args)
        {
            Assembly sampleAssembly = Assembly.Load("TypeEquivalenceRuntime");
            ISampleInterface sampleClass =
                (ISampleInterface)sampleAssembly.CreateInstance("TypeEquivalenceRuntime.SampleClass");
            sampleClass.GetUserInput();
            Console.WriteLine(sampleClass.UserInput);
            Console.WriteLine(sampleAssembly.GetName().Version.ToString());
            Console.ReadLine();
        }
    }
}
```

7. Press CTRL+F5 to build and run the program.

Modifying the Interface

To modify the interface

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, right-click the TypeEquivalenceInterface project, and then click **Properties**. Click the **Application** tab. Click the **Assembly Information** button. Change the **Assembly Version** and **File Version** values to `2.0.0.0`.
3. Open the SampleInterface.cs file. Add the following line of code to the ISampleInterface interface.

   ```
   DateTime GetDate();
   ```

   Save the file.
4. Save the project.
5. Right-click the TypeEquivalenceInterface project and click **Build**. A new version of the class library .dll file is compiled and saved in the specified build output path (for example, C:\TypeEquivalenceSample).

Modifying the Runtime Class

To modify the runtime class

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, right-click the TypeEquivalenceRuntime project and click **Properties**. Click the **Application** tab. Click the **Assembly Information** button. Change the **Assembly Version** and **File Version** values to `2.0.0.0`.
3. Open the SampleClass.cs file. Add the following lines of code to the SampleClass class.

```
public DateTime GetDate()
{
    return DateTime.Now;
}
```

Save the file.

4. Save the project.
5. Right-click the TypeEquivalenceRuntime project and click **Build**. An updated version of the class library .dll file is compiled and saved in the previously specified build output path (for example, C:\TypeEquivalenceSample).
6. In File Explorer, open the output path folder (for example, C:\TypeEquivalenceSample). Double-click the TypeEquivalenceClient.exe to run the program. The program will reflect the new version of the TypeEquivalenceRuntime assembly without having been recompiled.

See also

- /link (C# Compiler Options)
- C# Programming Guide
- Programming with Assemblies
- Assemblies and the Global Assembly Cache (C#)

Walkthrough: Embedding Type Information from Microsoft Office Assemblies in Visual Studio (C#)

1/23/2019 • 3 minutes to read • Edit Online

If you embed type information in an application that references COM objects, you can eliminate the need for a primary interop assembly (PIA). Additionally, the embedded type information enables you to achieve version independence for your application. That is, your program can be written to use types from multiple versions of a COM library without requiring a specific PIA for each version. This is a common scenario for applications that use objects from Microsoft Office libraries. Embedding type information enables the same build of a program to work with different versions of Microsoft Office on different computers without the need to redeploy either the program or the PIA for each version of Microsoft Office.

> **NOTE**
>
> Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see Personalizing the IDE.

Prerequisites

This walkthrough requires the following:

- A computer on which Visual Studio and Microsoft Excel are installed.
- A second computer on which the .NET Framework 4 or higher and a different version of Excel are installed.

To create an application that works with multiple versions of Microsoft Office

1. Start Visual Studio on a computer on which Excel is installed.
2. On the **File** menu, choose **New**, **Project**.
3. In the **New Project** dialog box, in the **Project Types** pane, make sure that **Windows** is selected. Select **Console Application** in the **Templates** pane. In the **Name** box, enter `CreateExcelWorkbook`, and then choose the **OK** button. The new project is created.
4. In **Solution Explorer**, open the shortcut menu for the **References** folder and then choose **Add Reference**.
5. On the **.NET** tab, choose the most recent version of `Microsoft.Office.Interop.Excel`. For example, **Microsoft.Office.Interop.Excel 14.0.0.0**. Choose the **OK** button.
6. In the list of references for the **CreateExcelWorkbook** project, select the reference for `Microsoft.Office.Interop.Excel` that you added in the previous step. In the **Properties** window, make sure that the `Embed Interop Types` property is set to `True`.

> **NOTE**
>
> The application created in this walkthrough runs with different versions of Microsoft Office because of the embedded interop type information. If the `Embed Interop Types` property is set to `False`, you must include a PIA for each version of Microsoft Office that the application will run with.

7. Open the **Program.cs** file. Replace the code in the file with the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using Excel = Microsoft.Office.Interop.Excel;

namespace CreateExcelWorkbook
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] values = {4, 6, 18, 2, 1, 76, 0, 3, 11};

            CreateWorkbook(values, @"C:\SampleFolder\SampleWorkbook.xls");
        }

        static void CreateWorkbook(int[] values, string filePath)
        {
            Excel.Application excelApp = null;
            Excel.Workbook wkbk;
            Excel.Worksheet sheet;

            try
            {
                    // Start Excel and create a workbook and worksheet.
                    excelApp = new Excel.Application();
                    wkbk = excelApp.Workbooks.Add();
                    sheet = wkbk.Sheets.Add() as Excel.Worksheet;
                    sheet.Name = "Sample Worksheet";

                    // Write a column of values.
                    // In the For loop, both the row index and array index start at 1.
                    // Therefore the value of 4 at array index 0 is not included.
                    for (int i = 1; i < values.Length; i++)
                    {
                        sheet.Cells[i, 1] = values[i];
                    }

                    // Suppress any alerts and save the file. Create the directory
                    // if it does not exist. Overwrite the file if it exists.
                    excelApp.DisplayAlerts = false;
                    string folderPath = Path.GetDirectoryName(filePath);
                    if (!Directory.Exists(folderPath))
                    {
                        Directory.CreateDirectory(folderPath);
                    }
                    wkbk.SaveAs(filePath);
            }
            catch
            {
            }
            finally
            {
                sheet = null;
                wkbk = null;

                // Close Excel.
                excelApp.Quit();
                excelApp = null;
            }
        }
    }
}
```

8. Save the project.

9. Press CTRL+F5 to build and run the project. Verify that an Excel workbook has been created at the location specified in the example code: C:\SampleFolder\SampleWorkbook.xls.

To publish the application to a computer on which a different version of Microsoft Office is installed

1. Open the project created by this walkthrough in Visual Studio.

2. On the **Build** menu, choose **Publish CreateExcelWorkbook**. Follow the steps of the Publish Wizard to create an installable version of the application. For more information, see Publish Wizard (Office Development in Visual Studio).

3. Install the application on a computer on which the .NET Framework 4 or higher and a different version of Excel are installed.

4. When the installation is finished, run the installed program.

5. Verify that an Excel workbook has been created at the location specified in the sample code: C:\SampleFolder\SampleWorkbook.xls.

See also

- Walkthrough: Embedding Types from Managed Assemblies in Visual Studio (C#)
- /link (C# Compiler Options)