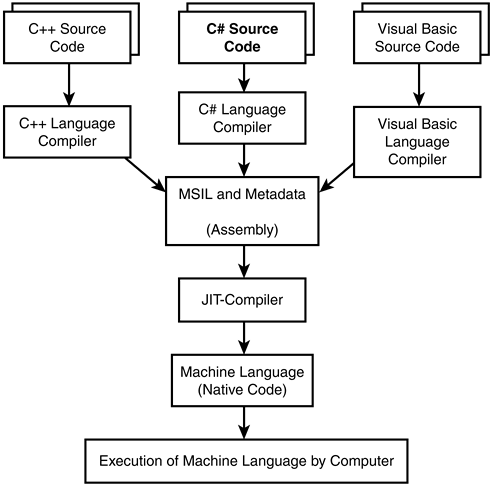
* When we create a project in Visual Studio .Net and compile it, assemblies are created. These **assemblies** are the fundamental units of applications in the .NET Framework. An assembly can contain classes, structures, interfaces, and resources that an application requires.
* A chunk of (precompiled) code that can be executed by the .NET runtime environment.
* An assembly is a collection of types and resources that forms a logical unit of functionality. All types in the .NET Framework must exist in assemblies; the common language runtime does not support types outside of assemblies. Each time you create a Microsoft Windows® Application, Windows Service, Class Library, or other application with Visual Basic .NET, you're building a single assembly.
* Assembly physically exist as DLLs or EXEs
* Assemblies include both executable application files that you can run directly from Windows without the need for any other programs (these have a .exe file extension), and libraries (which have a .dll extension) for use by other applications.
* Assemblies are the basic building blocks required for any application to function in the .NET realm. They are partially compiled code libraries that form the fundamental unit of deployment, versioning, activation scoping, reuse, and security. Typically, assemblies provide a collection of types and resources that work together to form a logical unit of functionality.
* Assembly is a compiled output of program which are used for easy deployment of an application. They are executables in the form of *exe* or *dll*. It also is a collection of resources that were used while building the application and is responsible for all the logical functioning.
* Assembly is a logical collection of smallest unit in .NET Framework.
* Microsoft **.Net Assembly** is a logical unit of code that contains code which the Common Language Runtime (CLR) executes.
* An assembly is a file that is automatically generated by the compiler upon successful compilation of every .NET application.
* Assembly can be either a Dynamic Link Library(DLL) or an executable file(EXE).
* It is generated only once for an application and upon each subsequent compilation the assembly gets updated.
* The entire process will run in the background of your application.



There is a tool called **ildasm** which can used to view the assembly information. There is also a .Net reflector which can be used to dig into assembly details.

## Why use Assemblies?

Assembly overcomes the famous DLL Hell problem. Previously, we were forced to have one version of DLL installed or rather registered on the system. The drawback was that if the same DLL is used in two components, fixing problems in one component would lead to causing problems on the other as the same DLL was used in both components. Now, with DLL we can have multiple versions of the same DLL . Also, we do not use registry anymore. If any assembly needs to be shared it is through Global Assembly Cache (GAC) and it can hold multiple versions of the same DLL.

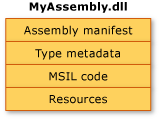
* In any programming environment it is of particular importance to be able to encapsulate dedicated functionality in libraries. We like to build modules and must be in the position to use the same code in different locations. Additionally, we want to save resources and have binaries loaded only if required by the functionality a program is using at a given time (**dynamic linking**). Libraries can be part of a single application or shared for the use by other programs, implemented by our own or even by other companies.
* The goal of the assembly model is the elimination of DLL Hell. Under the current COM/COM+ model, a catalog of DLLs is centralized in the Windows Registry. When a new version of a DLL is published, the registry re-references the catalog to point to the new DLL. This centralized registration paradigm makes it challenging for multiple applications to depend on the same DLL. Most often, the application binds to a DLL in a centralized location, rather than run multiple versions of a component by using side-by-side execution.
* The .NET Framework makes it easy to run multiple versions of a component because it stores assemblies in local application directories by default. This isolates the assembly from use by any other application and protects the assembly from system changes.
* In the past Dynamic Link Libraries (DLL) were located by the system environment's PATH setting and the Windows Registry. **Globally and shared DLLs were stored in the Windows system folder.**The identification was specified by the DLL's file name. These mechanisms do have certain drawbacks. One is that we can't move applications in our file system because the Windows Registry does have paths stored. This is especially on bigger systems a significant problem, if more disk storage has to be added, data has to be moved, disk layouts reorganized and so on. Another thing is that DLLs can be overwritten by mistake or on purpose, for instance by other applications' installation programs. These newer DLLs may not be compatible anymore and other programs using them may crash.
* **With Microsoft.NET the Windows Registry is not used anymore** and the libraries are stored either together with an application as **Private Assemblies** or in a Global Assembly Cache (GAC) as **Global Assemblies** to be shared among applications. Assemblies are signed and verified to recognise content modifications. They are identified by name and version to resolve incompatibility issues.

**Functions of an assembly:**

1. It contains code that the common language runtime executes. It is a managed environment.
2. It forms a security boundary. An assembly is the unit at which permissions are requested and granted
3. It is the unit at which side-by-side execution is supported.
4. It forms a deployment unit. When an application starts, only the assemblies that the application initially calls must be present. Other assemblies, such as localization resources or assemblies containing utility classes, can be retrieved on demand.
5. It forms a version boundary. The assembly is the smallest versionable unit in the common language runtime; all types and resources in the same assembly are versioned as a unit.

**Components of an assembly:**

* An assembly is a logical unit that is made up of the following four different types of components:
  1. Assembly manifest
  2. MSIL source code
  3. Type metadata
  4. Resources



**What is metadata?**

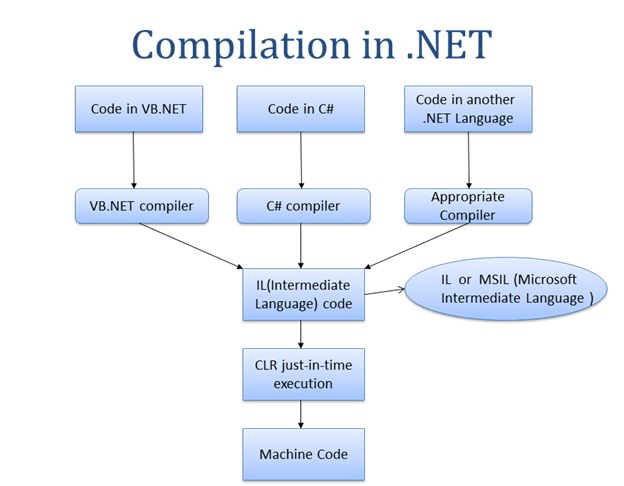
An assembly metadata describes every data type and member defined in the code. It stores the description of an assembly, such as name, version, culture, public key of an assembly along with the types exported, other assemblies dependent on this assembly, and security permissions needed to run the application. In addition, it stores the description of types, such as the name, visibility, base class, interfaces implemented, and members, such as methods, fields, properties, events, and nested types.  
  
It also stores attributes. Metadata is stored in binary format. Therefore, metadata of an assembly is sharable among applications that execute on various platforms. It can also be exported to other applications to give information about the services and various features of an application.

**What is Assembly Manifest?**

* An [Assembly](http://vb.net-informations.com/framework/assembly.htm) Manifest is a file that containing [Metadata](http://vb.net-informations.com/framework/metadata.htm) about .NET Assemblies. Assembly Manifest contains a collection of data that describes how the elements in the assembly relate to each other. It describes the relationship and dependencies of the components in the Assembly, versioning information, scope information and the security permissions required by the Assembly.
* The Assembly Manifest can be stored in Portable Executable (PE) file with [Microsoft Intermediate Language](http://vb.net-informations.com/framework/microsoft_intermediate_language.htm) (MSIL) code. You can add or change some information in the Assembly Manifest by using assembly attributes in your code. The Assembly Manifest can be stored in either a PE file (an .exe or .dll) with [Microsoft Intermediate Language](http://vb.net-informations.com/framework/microsoft_intermediate_language.htm) (MSIL) code or in a standalone PE file that contains only assembly manifest information. Using ILDasm, you can view the manifest information for any managed DLL.
* The followings are the contents of an Assembly Manifest:
* **Assembly name** - Represents a text string that specifies the assembly's name.
* **Version number** - Represents a major and minor version number, as well as a revision and build number. The CLR makes use of these numbers to enforce version policy.
* **Culture** - Represents information of the culture or language, which the assembly supports. An assembly is a container of only resources containing culture- or language-specific information.
* **Strong name information** - Represents the public key from the publisher, if a strong name is assigned to an assembly.
* **List of all files in the assembly** - Represents a hash of each file contained in the assembly and a file name.
* **Type reference information** - Represents the information used at the runtime to map a type reference to the file that contains its declaration and implementation.
* **Information on referenced assemblies** - Represents a list of other assemblies that are statically referenced by the assembly. Each reference includes the names of dependent assemblies, assembly metadata (version, culture, operating system, and so on), and public key, if the assembly is strong named.

**MSIL Code:**

* MSIL stands for Microsoft Intermediate Language. We can call it as Intermediate Language (IL) or Common Intermediate Language (CIL). During the compile time , the compiler convert the source code into Microsoft Intermediate Language (MSIL) .Microsoft Intermediate Language (MSIL) is a CPU-independent set of instructions that can be efficiently converted to the native code. During the runtime the [Common Language Runtime](http://vb.net-informations.com/framework/common_language_runtime.htm) (CLR)'s [Just In Time](http://vb.net-informations.com/framework/just_in_time_compiler.htm) (JIT) compiler converts the Microsoft Intermediate Language (MSIL) code into native code to the Operating System.
* When a compiler produces Microsoft Intermediate Language (MSIL), it also produces [Metadata](http://vb.net-informations.com/framework/metadata.htm). The Microsoft Intermediate Language (MSIL) and Metadata are contained in a portable executable (PE) file. Microsoft Intermediate Language (MSIL) includes instructions for loading, storing, initializing, and calling methods on objects, as well as instructions for arithmetic and logical operations, control flow, direct memory access, exception handling, and other operations.



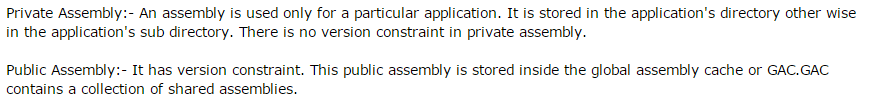
**How Assembly Manifest is useful?**

When the Common Language Runtime (CLR) loads an assembly, it first reads the manifest to get the information regarding assembly.

**Types of Assemblies:**

Assemblies are classified into:

1. Static and Dynamic Assemblies
2. Single-File and Multifile Assemblies
3. Private and Shared Assemblies
4. Satellite and Resource-only Assemblies



* **Single-File Assembly:**

When the components of an assembly are grouped in a single physical file, it is known as **Single-file assembly.**

A single-file assembly stores the manifest, type metadata,   
IL, and resources in a single file. You can use this   
approach for simple applications when you need to develop   
an assembly for a small-scale deployment.

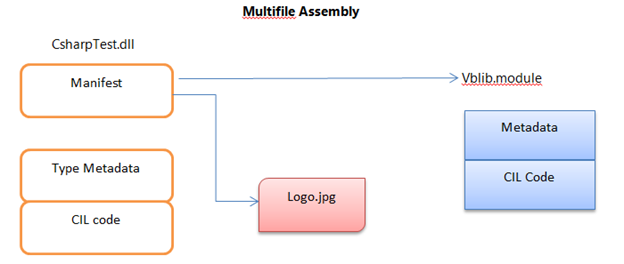
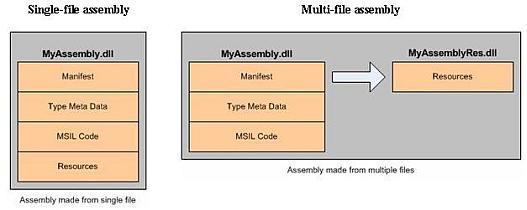
A single-file assembly consists of a single .exe or .dll file.

* **Multi-File Assembly:**

When the components of an assembly are contained in several files, it is known as **Multifile assembly**.

When developing large applications, you can split the   
application into smaller modules and deploy different   
development teams to work on various modules in a parallel   
mode. In this situation, you can use multi-file assemblies,   
with the different teams developing and compiling the   
modules. Using the assembly linker, you can integrate the   
modules into an application, where the assembly is broken   
into multiple files. Although the application environment   
contains several modules that can be developed by separate   
teams, the modules are closely related in functionality.

For example, you can have different modules related to the   
system administration functionality, developed over a   
period of time. Using the multi-file assembly feature   
of .NET, you can group all administration functionality   
modules into an assembly to facilitate component versioning   
and maintenance.

****

**Private Assembly:**

Assemblies which are used by single application are called "Private Assemblies".

Assemblies are private in scope if only one application can   
access them. Private assemblies are only available to   
clients in the same directory structure as the assembly. As   
a result, the .NET assembly resides in the same directory   
structure as the client executable. You can use a private   
assembly when it is specific to a client application and no   
other client application refers to it.

When the .net code gets compiled it generates an assembly which is stored in **bin** folder.

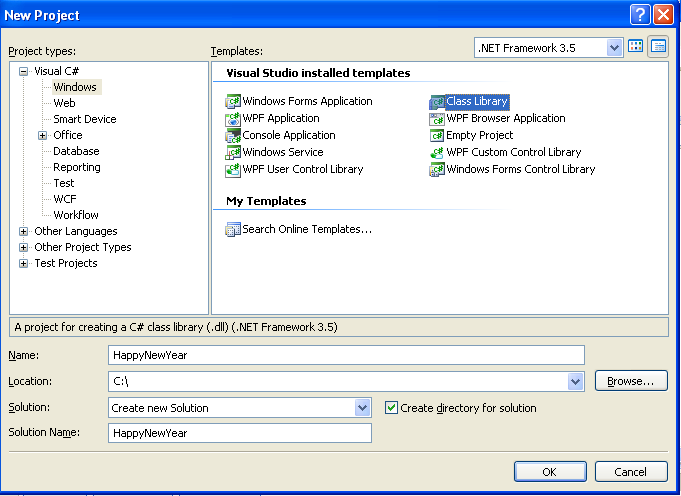
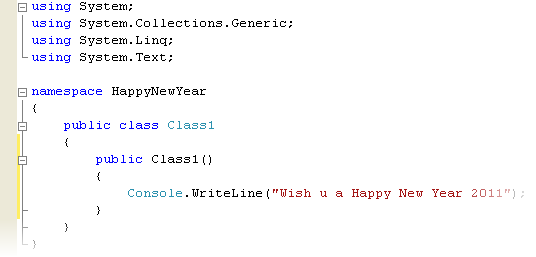
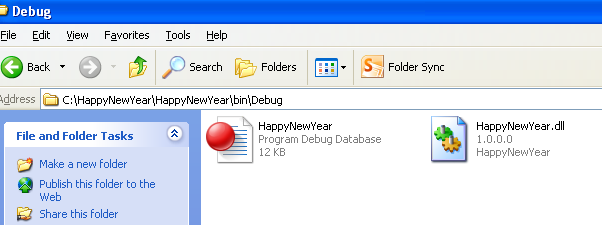
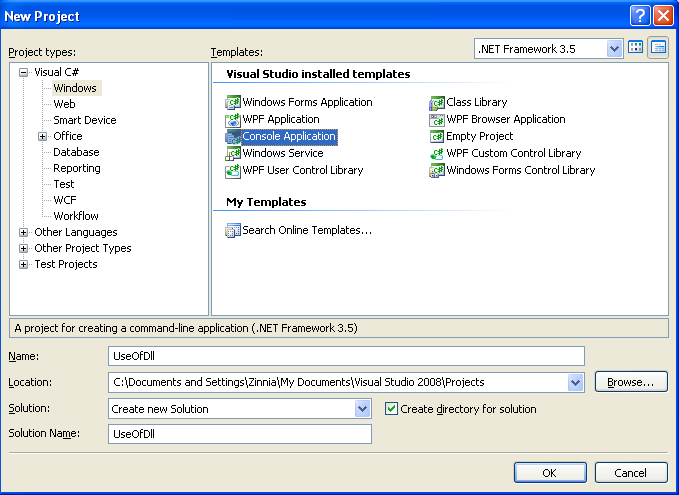
* When you deploy an assembly which can be use by single application, than this assembly is called a private assembly.
* Private assemblies can be used by only one application they are deployed with.
* Private assemblies are deployed in the directory where the main application is installed.

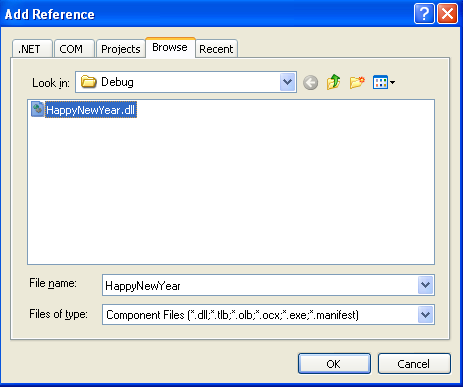
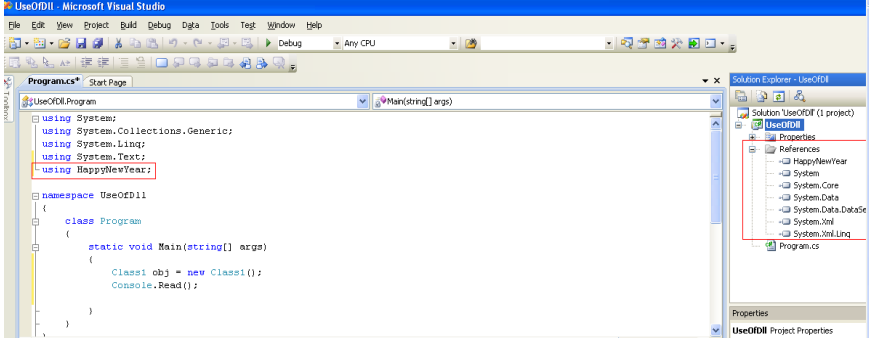
**Shared Assembly:**

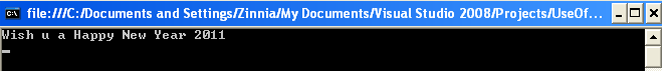
* When you deploy an assembly which can be used by several application, than this assembly is called shared assembly.
* Shared assemblies are stored in a special folder called Global Assembly Cache (GAC), which is accessible by all applications.
* Shared assemblies must have a strong name. A strong name consists of an assembly name, a version number, a culture, a public key and an optional digital signature.
* GAC is capable of maintaining multiple copies of an assembly with the same name but different versions.

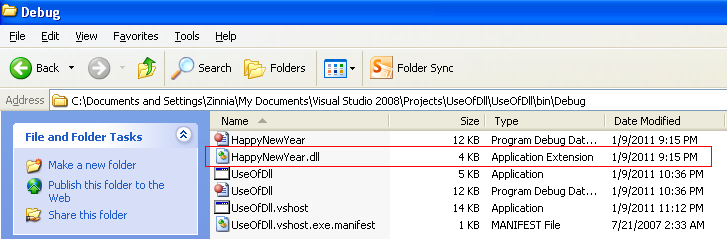
* A private assembly is an assembly that is available to particular application where they are kept.
* Assemblies which are used by single application are called "Private Assemblies".
* It is used only by the single application and is stored in the application's install directory.
* If an assembly is copied in to the respective application in which we would like to use is known as local assembly. if any changes made to the copy that will not reflect the copies in other applications.
* Private assembly can be used by only one application.
* Private assembly will be stored in the specific application's directory or sub-directory.
* There is no other name for private assembly.
* Strong name is not required for private assembly.
* Private assembly doesn't have any version constraint.
* By default, all assemblies you create are examples of private assembly. Only when you associate a strong name to it and store it in GAC, it becomes a public assembly.

**Building a Private Assemblies:**

**Start->All Programs->MS VS 2008->MS VS 2008->File->New->Project->ClassLibrary**  
I have named it as **HappyNewYear**. The code follows here:  
  
Now it is the time to compile. So click at **Build->Build Solution**.  
With this, **HappyNewYear.dll** is created at **C:\HappyNewYear\HappyNewYear\bin\Debug**  
  
  
We have to use our .dll file in new ConsoleApplication. So create a new ConsoleApplication. It is named as **UseOfDll**.  
  
  
Right Click on **Solution Explorer->Add Reference->**

With this **Add Reference** Dialog box open. From here, we have to browse our respective .dll file and press **OK**button.  
  
Now, **Solution Explorer** shows, that .dll file is being added under **References**. We have used the **HappyNewYear** namespace.  
  
  
Let me run the program and get the output.

  
Here using **Private Assembly**, .**dll** gets copied in the client application folder.



Create assembly in .Net by opening [Visual studio](http://www.forpromocode.com/#msvs2012) Framework=>Create Project=>Select Class Library=>give name as “PrivateAssembly” and add below code.

 namespace PrivateAssembly

{

    public class Class1

    {

        public string display()

        {

            return "This is from Private Assembly";

        }

    }

}

Build this application. If you build this application in Debug mode you will find PrivateAssembly.dll in Debug folder of bin folder or if you build this application in release mode you will find the PrivateAssembly.dll in release folder of bin folder.

Now we have to create Client Application to use above created private assembly. For that by open Visual studio Framework=>Create Project=>Select Console Application and give name as ReferPrivateAssembly.

Add PrivateAssembly.dll by right click on solution explorer and select Add reference, add PrivateAssembly.dll file. Then you will find this PrivateAssembly.dll file in your client application folder bin directory. Add below code to call assembly methods.

namespace ReferPrivateAssembly

{

    class Program

    {

        static void Main(string[] args)

        {

            PrivateAssembly.Class1 obj = new PrivateAssembly.Class1();

            Console.WriteLine(obj.display());

            Console.ReadLine();

        }

    }

}   
The output display as shown below by calling PrivateAssembly display() method.



## Benefits of Private Assemblies

* Private assemblies are installed in a directory named *bin* located under the application directory. These files are private to the application.
* No versioning is required, as long as it is the same version as the one with which the application was built.
* There is no configuration or signing
* It is great for small utility Assemblies/ application specific code

**Probing for Private Assemblies:**

**Probing** is the process of locating assembly (*dll*) required to the Application.

In .Net framework when resolve an assembly reference, **CLR** first checks **GAC**, then search the application directory in specific locations.

The .NET runtime resolve the location of a private assembly using a technique termed probing, which is much less invasive than it sounds. Probing is the process of mapping an external assembly reference (i.e., [.assembly extern]) to the correct corresponding binary file.

Probing can be defined as Runtime Searching for the Assembly required to the Application.

When it comes to private assembly Runtime will search for Application Folder and subfolder with in a library name for the particular assembly

**The Identity of a Private Assembly:**

The identity of a private assembly consists of a friendly string name and numerical version, both of which are recorded in the assembly manifest. The friendly name is created based on the name of the binary module that contains the assembly's manifest.

**Private Assemblies and XML Configuration Files:**

When we want to extend probing with other directories then provide the information to the runtime in a special file called Configuration file.

When the .NET runtime is instructed to bind to an assembly, the first step is to determine the presence of an application configuration file. These optional files contain XML tags that control the binding behavior of the launching application. By law, configuration files must have the same name as the launching application and take a \*.config file extension.

Configuration files can be used to specify any optional subdirectories to be searched during the process of binding to private assemblies.

For example, assume our main directory is called MyRadApplication, which contains a number of subdirectories (\Images, \Bin, \SavedGames, \OtherCoolStuff). Using application configuration files, you can instruct the runtime where it should probe while attempting to locate the set of private assemblies used by the launching application.

let's create a simple configuration file for the previous CSharpCarClient application. Our goal is to move the referenced assembly (CarLibrary) from the Debug folder into a new subdirectory named Foo \ Bar.

create a new configuration file named CSharpCarClient.exe.config (notepad will do just fine) and save it into the same folder containing the CSharpCarClient.exe application. The beginning of an application configuration file is marked with the <Configuration. tag. Before the closing </Configuration. tag, specify an assemblyBinding row, which is used to specify alternative locations to search for a given assembly, using the privatePath attribute (FYI, multiple subdirectories can be specified using a semicolon delimited list):

<configuration>

<runtime>

<assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">

<probing privatePath="foo\bar"/>

</assemblyBinding>

</runtime>

</configuration>

Once you are done, save the file and launch the client. You will find that the CSharpCarClient application runs without a hitch. As a final test, change the name of your configuration file and attempt to run the program once again.

# Private Assembly

When we compile the C# Program Console Application an exe will be created.

## To Create dll from the Command Prompt.

Open Note pad and Type the Following  
Example:

namespace MyApp  
{  
public class MathC  
{  
private int a,b;  
private int res;  
public void Accept( int A,int B)  
{  
a =A;  
b= B;  
}  
public int Sum()  
{  
res = a+b;  
return res;  
}  
}  
}  
  
Save it as Mydll.cs  
  
Compilation : csc /t:library Mydll.cs  
  
Mydll.dll is Produced 

## Calling this assembly in Client Application

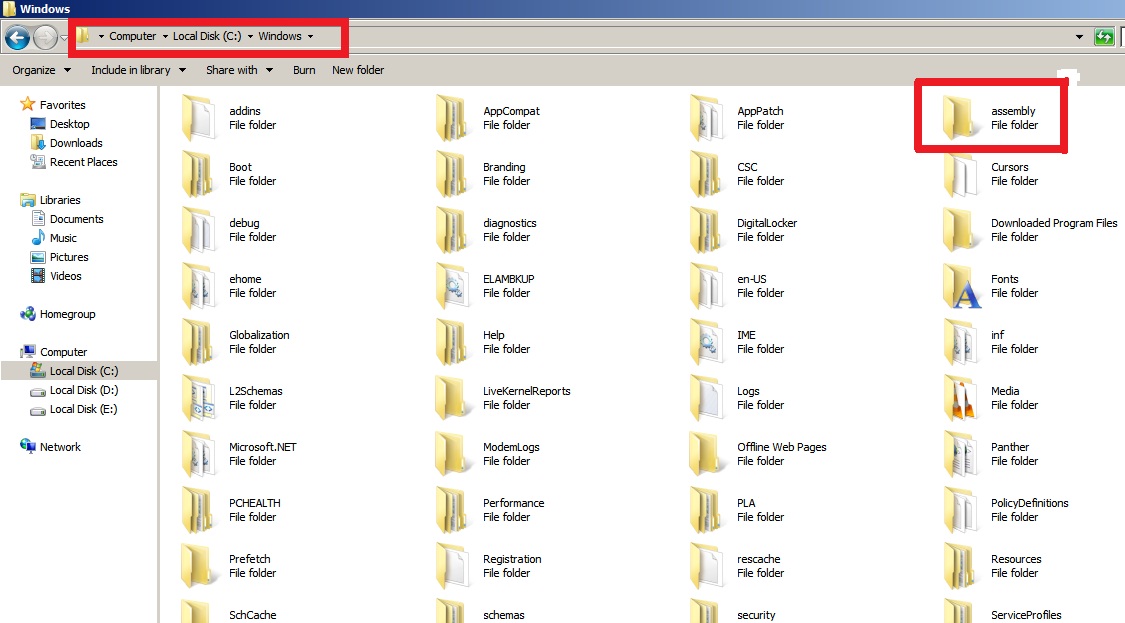
using System;  
using MyApp;  
  
class C{  
static void Main()  
{  
MathC obj = new MathC();  
obj.Accept(10,20);  
Console.WriteLine("Obj value is"+obj.Sum());  
}  
}  
  
Save it as ClientApp.cs  
  
If we compile csc ClientApp.cs it gives error  
  
The correct compilation is csc /r: mydll.dll ClientApp.cs   
  
/r refers to reference switch.  
It’s a request to the complier to verify name spaces (all) with in System.dll by default. When we want to provide a custom library to verify the namespace then we require reference switch option  
  
Hence this will produce ClientApp.exe   
And the result is executed  
  
C:\>ClientApp  
Obj value is30  
  
C:\>md Dir1  
  
C:\>copy PAssembly.dll Dir1  
1 file(s) copied.  
  
C:\>del PAssembly.dll  
  
C:\>ClientApp  
  
LOG: Attempting download of new URL file:///C:/PAssembly.DLL.  
LOG: Attempting download of new URL file:///C:/PAssembly/PAssembly.DLL.  
LOG: Attempting download of new URL file:///C:/PAssembly.EXE.  
LOG: Attempting download of new URL file:///C:/PAssembly/PAssembly.EXE.  
  
Run Time will search for Mydll.dll in C Drive  
And gives the Error Message

## Probing

Probing can be defined as Runtime Searching for the Assembly required to the Application  
When it comes to private assembly Runtime will search for Application Folder and subfolder with in a library name for the particular assembly   
When we want to extend probing with other directories then provide the information to the runtime in a special file called Configuration file  
  
Save it as ClientApp.exe.Config  
  
<configuration>  
<runtime>  
<assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">  
  
<probing privatePath="Dir1"/>  
</assemblyBinding>  
  
</runtime>  
</configuration>  
  
assemblyBinding Contains information about assembly version redirection and the locations of assemblies.  
configuration The root element in every configuration file used by the common language runtime and .NET Framework applications.  
runtime Contains information about assembly binding and garbage collection.  
  
Now go to VS Command Prompt and just run ClientApp  
Then It will search for the Assembly in the Located path as Dir1 in the Specific Root Directory and all the Subfolders   
We get the output as : Sum is 30  
**Built in Library vs Custom Library**

Built in Library defined by the MS comes from System NameSpace  
Console Class  
Custom Library defined by the Developer that is Mydll  
MathC is the class   
Import the name space with ‘using’ keyword

* A shared Assembly is also called as strong named assemblies or public assemblies.
* A shared Assembly is one that can be referenced by more than one application.
* When you deploy an assembly which can be used by several application, than this assembly is called shared assembly.
* If multiple applications need to access an Assembly, we should add or store the Assembly to a special location (separate location) called the [Global Assembly Cache](http://vb.net-informations.com/framework/global-assembly-cache.htm) (GAC) with a strong name assigned to it. (Shared Assemblies are kept in **Global Assembly Cache**).
* Multiple applications using a shared assembly, an application must have a key (token) for authorization (permission or approval) and authority (power).
* When .dll and main file are not in same folder & main file can't direct access the .dll file. It must take permission for Runtime manager for using that .dll, and then the collection to run the file can be known as shared assembly.
* Client application using Shared Assemblies need not have its own copy of dll. Shared Assemblies are present in C:\WINDOWS\assembly folder OR C:\Winnt\Assembly.

* Suppose we are creating a general purpose DLL which provides functionality to be used by a variety of applications. Now, instead of each client application having its own copy of DLL we can place the DLL in 'global assembly cache'. Such assemblies are called as **shared assemblies**.
* For example, imagine that you have created a DLL containing information about your business logic. This DLL can be used by your client application. In order to run the client application, the DLL must be included in the same folder in which the client application has been installed. This makes the assembly private to your application. Now suppose that the DLL needs to be reused in different applications. Therefore, instead of copying the DLL in every client application folder, it can be placed in the global assembly cache using the GAC tool. These assemblies are called shared assemblies.
* To make an assembly a shared assembly, it has   
  to be strongly named. In order to share an assembly with many applications, it must have a   
  strong name.
* GAC is capable of maintaining multiple copies of an assembly with the same name but different versions.[ It has version constraint]
* Build it as a shared assembly by giving it a cryptographically strong name.
* Shared Assemblies are implemented for re-usability.

**What is Global Assembly Cache?**

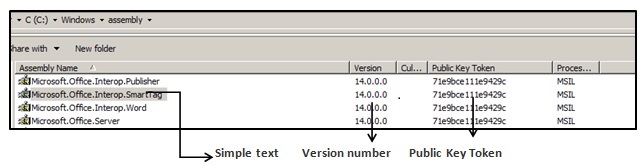
GAC is a central repository (cache) in a system in which assemblies are registered to share among various applications that execute on local or remote machines. .NET Framework provides the GAC tool (gacutil.exe utility), which is used to view and change the content of GAC of a system. Adding new assemblies to GAC and removing assemblies from GAC are some of the tasks that can be performed by using the gacutil.exe utility. GAC can contain multiple versions of the same .NET assembly. CLR checks GAC for a requested assembly before using information of configuration files.

* When you wish to create an assembly that can be used by numerous applications on a given machine, the first step is to create a unique shared name (strong name) for the assembly.
* A shared assembly must be assigned a "shared name or Strong Name".
* Strong Names are a way of uniquely identifying assemblies written for the .Net platform. They are not, as commonly believed, a tool for enabling security.
* Whenever, an assembly is deployed in GAC to make it shared, a strong name needs to be assigned to it for its unique identification. A strong name contains an assembly's complete identity - the assembly name, version number, and culture information of an assembly. A public key and a digital signature, generated over the assembly, are also contained in a strong name. A strong name makes an assembly identical in GAC.[ A shared name contains the following information:

1. A friendly string name and optional culture information (just like a private assembly).
2. A version identifier.
3. A public/private key pair.
4. A digital signature.]

* A strong name is a .NET assembly name combined with its version number and other information to uniquely identify the assembly.
* A strong name consists of five parts:

1. **Simple Name** – Usually the name of the file (without the extension) that contains the assembly
2. **Public Key** – RSA cryptographic public key that helps verify the assembly’s authenticity
3. **Version** – Four-part version number, in the form of Major.Minor.Build.Revision
4. **Culture** – Target audience for the assembly, such as “neutral” (default audience), “en-us” (English – United States) or “fr” (France) etc.
5. **Processor Architecture** – Defines the assembly’s format, such as MSIL (intermediate language) or x86 (binary for Intel x86 processors)



* Example of strong name is “Mini-Launcher, Version=0.3.612.24542, Culture=neutral, PublicKeyToken=ffa52ed9739048b4, Processor Architecture=”MSIL”.

## Why use strong names?

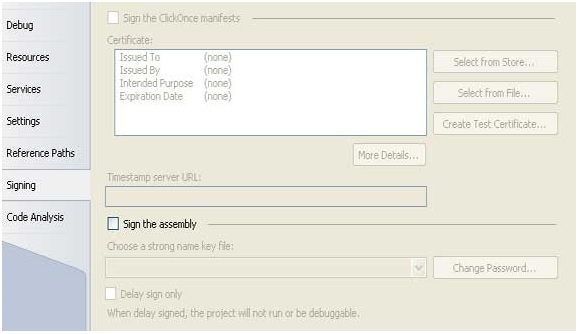
* [**Strong names**](http://www.codeproject.com/dotnet/StrongNameExplained.asp) are required to store shared assemblies in the global assembly cache (GAC). This is because the GAC allows multiple versions of the same assembly to reside on your system simultaneously, so that each application can find and use its own version of your assembly. This helps avoid [**DLL Hell**](http://en.wikipedia.org/wiki/DLL_hell), where applications that may be compiled to different versions of your assembly could potentially break because they are all forced to use the same version of your assembly.
* Another reason to use strong names is to make it difficult for hackers to [**spoof**](http://en.wikipedia.org/wiki/Spoofing_attack) your assembly, in other words, replace or inject your assembly with a virus or malicious code.

## What is a strong name key file?

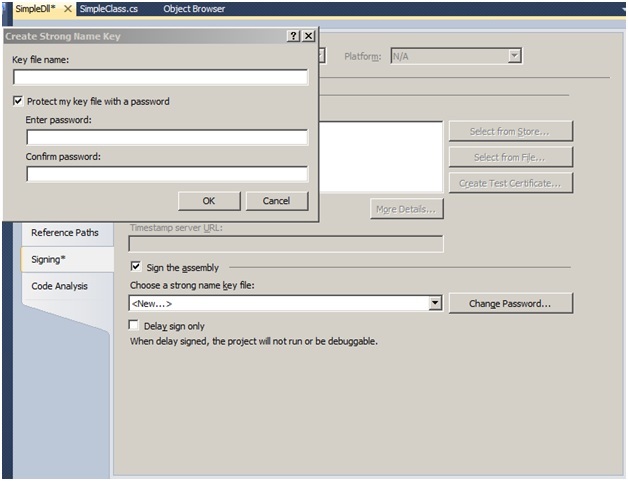
* A strong name key file has a .snk extension and contains a unique public-private key pair. You use the strong name key file to digitally sign your assembly. Note that this type of file is not secure, as the private key in a .snk file can be easily compromised.
* For added protection, Visual Studio can encrypt a strong name key file, which produces a file with the .pfx (Personal Information exchange) extension. The .pfx file is more secure because whenever someone attempts to use the encrypted key, she will be prompted for the password.

## How do I create a strong name key file for a .NET assembly?

* Visual Studio makes it easy to create a strong name key file:

1. Select your assembly project in the Visual Studio Solution Explorer.
2. Click the **Properties** button. The project properties will appear in the main window.
3. Select the **Signing**tab.
4. Check the **Sign the assembly**checkbox [click to select the **Sign the assembly** check box].

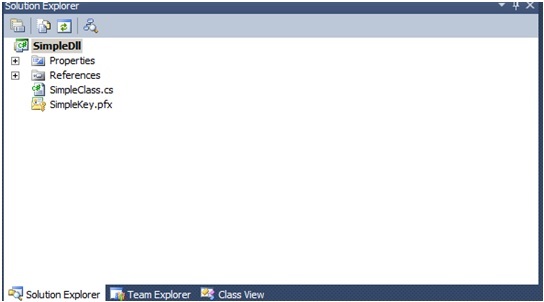
1. Under Choose a strong name key file, select <New> by drop down list.
2. In the Create Strong Name Key dialog box, click to select the Protect my key file with a password check box.



1. In the **Key file name** box, type **SimpleKey**. [In the **Key file name** text box, type the desired key name. Typically this is the name of your assembly but can be anything. Visual Studio will automatically append the proper file extension.]
2. If desired, you can protect the strong name key file with a password. To do so, check the **Protect my key file with a password** checkbox, then enter and confirm the password.
3. In the **Enter password** box, type the password that you want to use.
4. In the **Confirm password** box, type the same password, and then click **OK**.



1. **SimpleKey.pfx** will be created.
2. If you give the Password in Key file name **.pfx** file will be created, else **.snk** file will be created in project



1. Build the Solution in Release Mode.

* When you compile your project, Visual Studio will automatically sign your assembly with the new strong name key you have just created.
* if you prefer to **use the command-line**, you can create a key pair file with the [**strong name utility sn.exe**](http://msdn2.microsoft.com/en-us/library/k5b5tt23(VS.80).aspx) in the .NET SDK, for example:

sn -k MyKey.snk

Where mykey is strong key name (name of the key you created) and .snk is string name key extension.

Then you reference that key file to when compiling your code with the [**C# compiler csc.exe**](http://msdn2.microsoft.com/en-us/library/2fdbz5xd(VS.80).aspx):

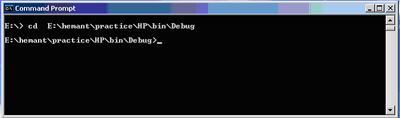
csc /keyfile:MyKey.snk MyCodeFile.cs

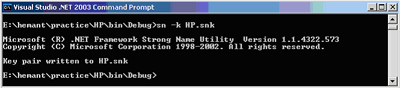
**Steps for Giving strong name to DLL:**

1. Open .net command prompt.
2. Go to the folder contanig DLL.
3. Type sn -k test.snk, you can use any file name instead of test. This will create  test .snk file in that folder.
4. Open the assemblyinfo.cs file of project.
5. Type file path  in this tag [assembly:AssemblyKeyFile@"E:\hemant\practice\HP\bin\Debug\HP.snk")]
6. Build application, finally your strong name created for your DLL.

**Example**

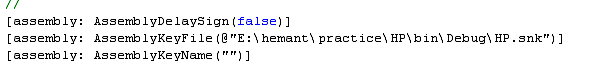
* Created class library path E:\hemant\practice\HP
* Open command prompt go to Dll folder path.

  
Type sn -k HP.snk

  
This create HP.snk file in E:\hemant\practice\HP\bin\Debug folder.

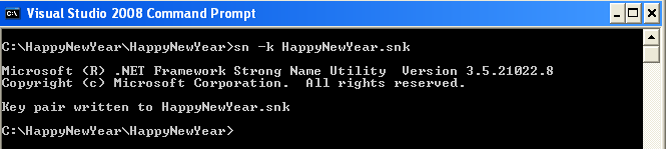
* Open the AssemblyInfo.cs file of project.

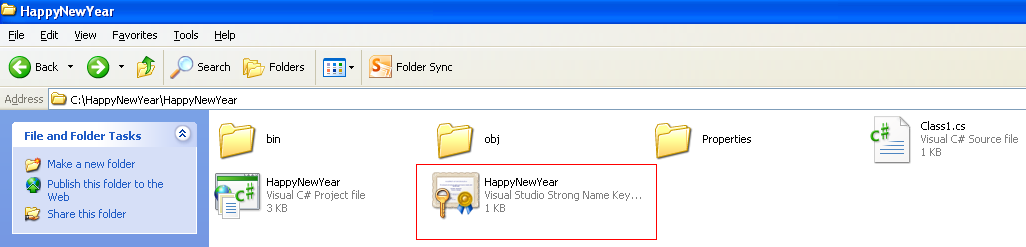
Type  path  of snk file in following attribue.



* Build class library.

Another example

 Key pair is created now. The following figure shows the strong name key pair is created.



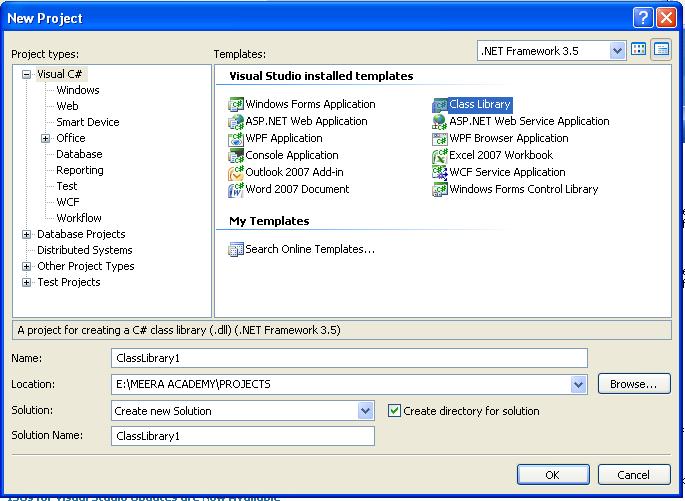
**What does it mean to sign an assembly?**

* .NET uses digital signatures to verify the integrity of an assembly.
* The signatures are generated and verified using [**public key cryptography**](http://en.wikipedia.org/wiki/Public-key_cryptography), specifically the [**RSA**](http://en.wikipedia.org/wiki/RSA) public key algorithm and [**SHA-1**](http://en.wikipedia.org/wiki/SHA-1#SHA-1_hashes) hash algorithm.
* The developer uses a pair of cryptographic keys: a public key, which everyone can see, and a private key, which the developer must keep secret.
* To create a strong-named assembly, the developer signs the assembly with his private key when building the assembly. When the system later loads the assembly, it verifies the assembly with the corresponding public key
* Following steps are involved in creating shared assemblies :

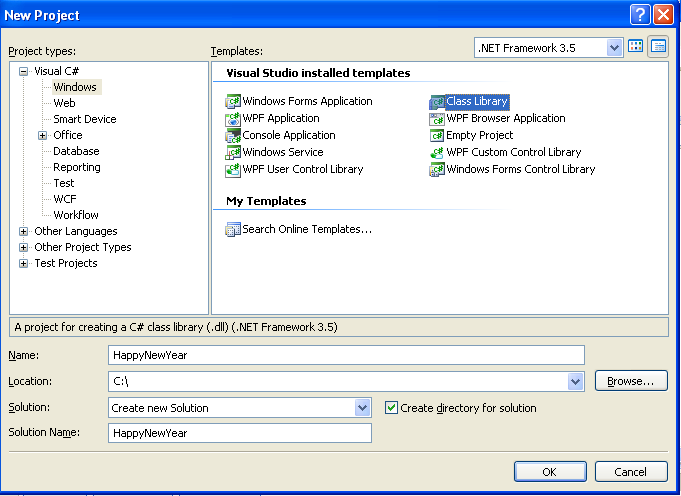
1. Create your DLL/EXE **source code.**
2. Generate unique assembly name using **SN** **utility**
3. Sign your DLL/EXE with the private key by modifying **AssemblyInfo** file.
4. **Compile** your DLL/EXE.
5. Place the resultant DLL/EXE in global assembly cache using **AL** **utility**

**Create your DLL/EXE source code:**

STEP 1: The Visual Studio and Create New **Class Library** Project.



I have named it as **HappyNewYear**.



**The code follows here:**

Using System;

Using System.Collections.Generic;

Using System.Linq;

Using System.Text;

namespace HappyNewYear

{

public class class1

{

Public class1()

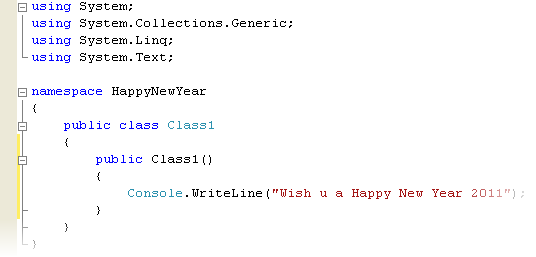
{

Console.WriteLine(“Wish you Happy New Year”);

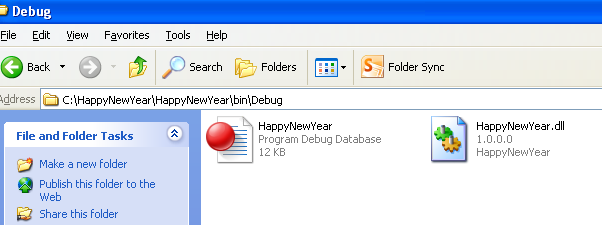
}

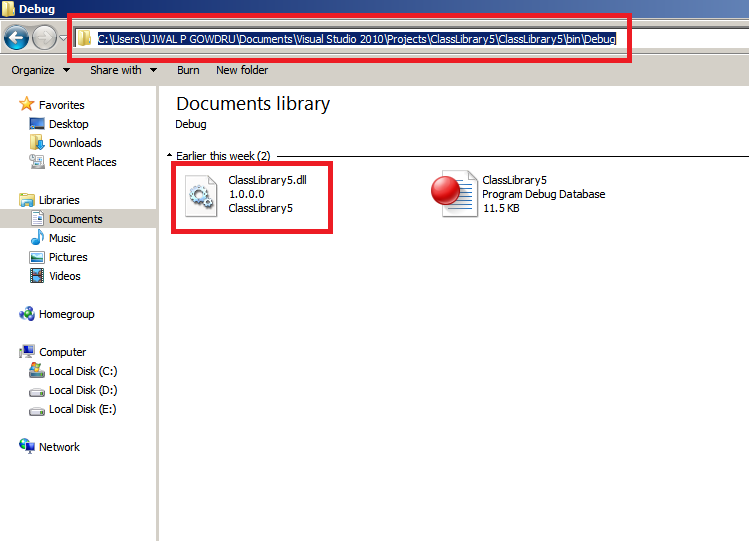
}

}



* Now it is the time to compile. So click at **Build->Build Solution**.
* With this, **HappyNewYear.dll** is created at **C:\HappyNewYear\HappyNewYear\bin\Debug.**





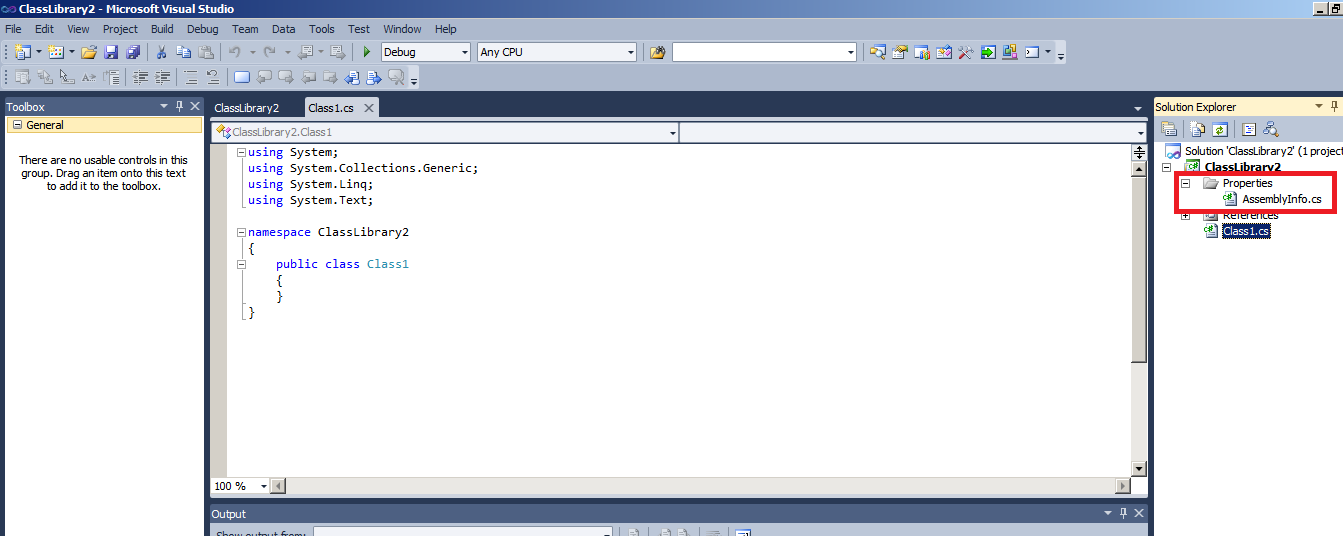
**Step 2: Generate unique assembly name using SN utility:**

**[**Create unique assembly name using SN utility (Shared Name utility).**]**

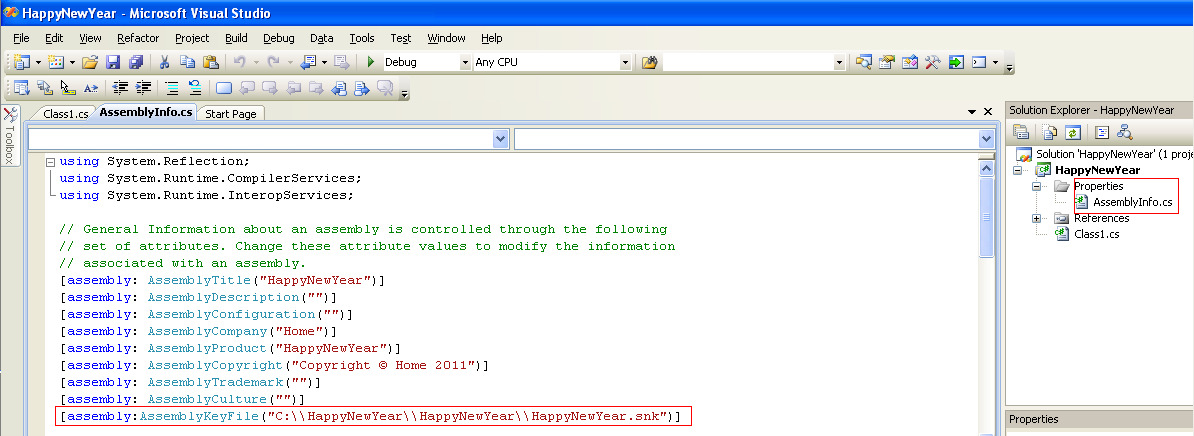
Follow the above steps like creating strong name in using IDE and command prompt.

**Step 3: Sign your DLL/EXE with the private key by modifying AssemblyInfo file:**

**AssemblyInfo.cs** file is present under Properties in the Solution Explorer. Here i have to give the path of the HappyNewYear.snk file.

[assembly: AssemblyKeyFile("C:\\HappyNewYear\\HappyNewYear\\HappyNewYear.snk")]  


* Double click assemblyinfo.cs file



**Step 4: Compile your DLL/EXE:[** Compile the dll again to get the assembly signed.**]**

**Step 5: Place the resultant DLL/EXE in global assembly cache using AL utility.**

Shared assemblies can be deployed into GAC by using below methods.

**Method 1:**

**Drag and Drop Method:**

Just drag and drop/ Copy Paste an assembly to an assembly folder (C:\Windows\assembly).

**Method 2:**

**Using Visual Studio Command prompt:**

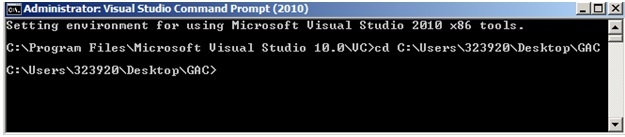
**Step 1:**

 Go to Microsoft Visual Studio 2010 -> Visual Studio Tools ->Visual Studio Command Prompt (2010).

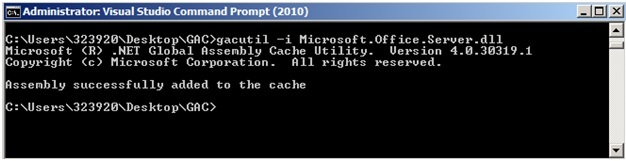


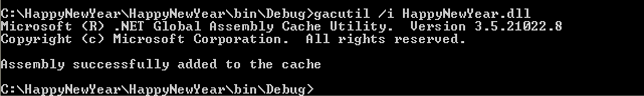
**Step 2:**

Navigate to the folder where the DLL is located by using cd Command.

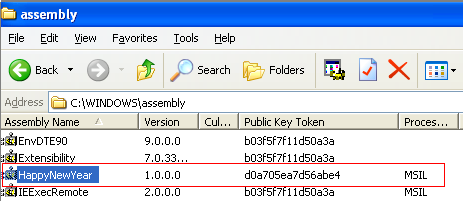
  
**Step 3:**

Use the **gacutil –I** <*assembly name* > command to install assembly into GAC.

**

****Step 4:**

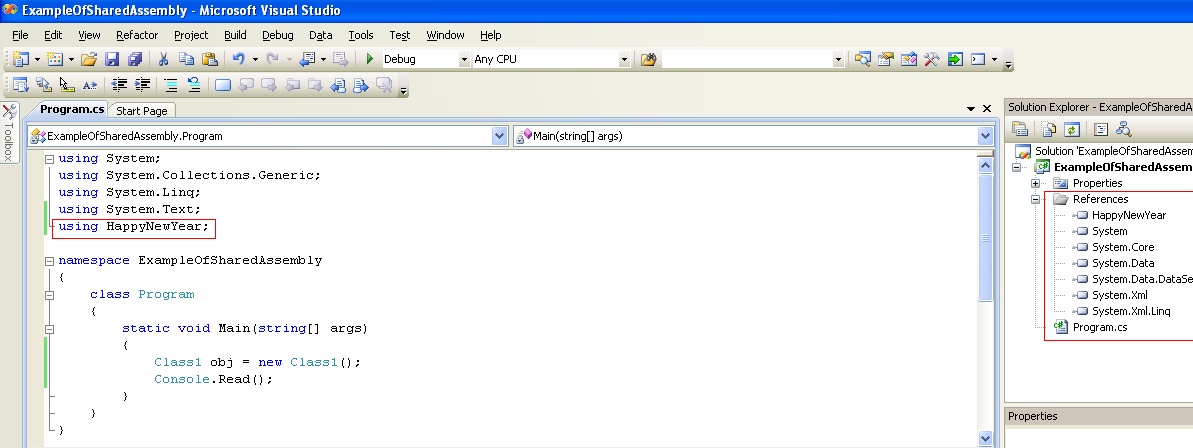
Once installation successfully completes, go to C:\Windows\assembly (GAC) location and you will find the dll.

****

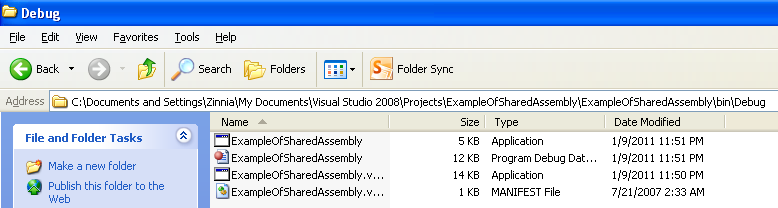
**To remove a shared assembly, from the command line enter:**

gacutil.exe /u myassembly.dll

**Step 5: use this Shared Assembly in one application.**



Here no copy of dll is created within the Debug folder of the application.



# Implementation Steps of Shared Assemblies

**1. Create a Class Library Project.**

* Create a new “Class Library” project.
* Write the required code in the project.

**2. Create a strong name key.**

* Right-click on the project in the “Solution Explorer” and choose “Properties”.
* In the project properties, select the check box “Sign the assembly”.
* In the “Choose a strong name key file” drop down, select “<New>” option.
* In the “Create strong name key” dialog box, enter the name of the strong name key file.
* If password security is not required, uncheck the “Protect my key file with a password” checkbox.
* Click on OK.

**3.Customize the “Assembly Information” (AssemblyInfo.cs).**

* This is the optional step.
* To change the additional details of the assembly like displayed name, version, company, copyright, description etc., open **“AssemblyInfo.cs”** file from the “Properties” folder in the Solution Explorer.

**4.Generate the DLL File.**

* Build the class library project.
* Then the “.dll” file will be generated in the “bin\debug” folder of your class library project.

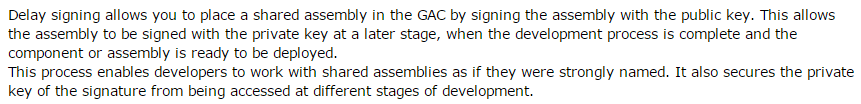
**5.Write the assembly into GAC (Global Assembly Cache).**

* Open the following folder.  
  **C:\Windows\Assembly**
* Drag and drop the “.DLL file” from “bin\Debug” folder into the “c:\windows\assembly” folder.
* After dragging, the name of your shared assembly will appear in the existing list.
* Now, the shared assembly is ready. The rest of your work involved with the usage of the shared assembly.

**6.Invoke the Shared Assembly.**

* Create the executable project (Console application/windows application).
* Click on “Project” – “Add Reference”.
* Click on “Browse” tab.
* Open the class library project’s “bin\Debug” folder.
* Select the “DLL file” and click on OK.
* Then the reference to the selected shared assembly will be added to the current project.
* Then you can construct objects for the required class in the class library and perform required activities on that.

**Understanding Delay signing:**



Delay signing allows you to place a shared assembly in the GAC by signing the assembly with just the public key.This allows the assembly to be signed with the private key at a later stage, when the development process is complete and the component or assembly is ready to be deployed. This process enables developers to work with shared assemblies as if they were strongly named, and it secures the private key of the signature from being accessed at different stages of development.

**Difference between Private Assembly and Public Assembly**

|  |  |
| --- | --- |
| **Public Assembly** | **Private Assembly** |
| Public assembly can be used by multiple applications. | Private assembly can be used by only one application. |
| Public assembly is stored in GAC (Global Assembly Cache). | Private assembly will be stored in the specific application's directory or sub-directory. |
| Public assembly is also termed as shared assembly. | There is no other name for private assembly. |
| Strong name has to be created for public assembly. | Strong name is not required for private assembly. |
| Public assembly should strictly enforce version constraint. | Private assembly doesn't have any version constraint. |
| An example to public assembly is the actuate report classes which can be imported in the library and used by any application that prefers to implement actuate reports. | By default, all assemblies you create are examples of private assembly. Only when you associate a strong name to it and store it in GAC, it becomes a public assembly. |

Example Code to use

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

public class strong

{

public string Greeting(string name)

{

return ("Your assembly says Hi : " + name);

}

static void Main(string[] args)

{

}

}

using System;

namespace BAJComponents

{

public class Sample

{

public string GetData()

{

return "hello world";

}

}

}

using System;

using BAJComponents;

public class SampleTest

{

static void main()

{

sample x= new sample();

string s= x.getdata();

console.writeline(s);

}

}

using System;  
namespace SharedAssembly  
{  
    public class Bike  
    {  
        public void start()  
        {  
            Console.WriteLine("kick start ");  
        }  
   }  
}

using System;  
using SharedAssembly;  
public class MainP  
{  
    public static void Main(string []args)  
    {  
     Bike bk=new Bike();  
     bk.start();  
     Console.Read();  
   }  
}

using System;

using System.Reflection;

[assembly: AssemblyKeyFile("shrd.snk")]

namespace SharedAssembly

{

    public class Fruit

    {

        public void start()

        {

            Console.WriteLine("Mango Apple");

        }

   }

}

using System;

using SharedAssembly;

public class Program

{

    public static void Main(string [ ]args)

    {

     Fruit f=new Fruit();

     f.start();

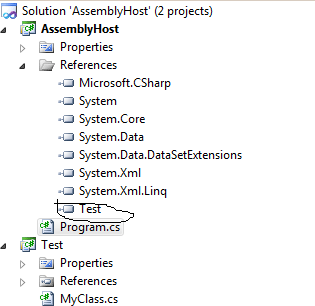
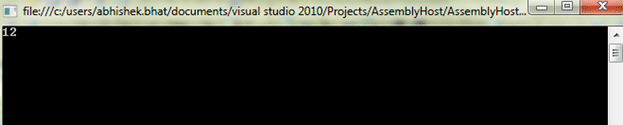
     Console.Read ( );

   }

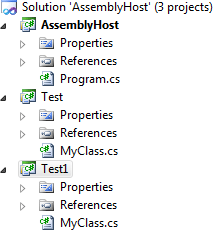
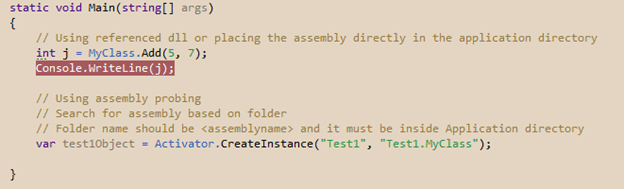
}

namespace MyApp  
{  
public class MathC  
{  
private int a,b;  
private int res;  
public void Accept( int A,int B)  
{  
a =A;  
b= B;  
}  
public int Sum()  
{  
res = a+b;  
return res;  
}  
}  
}

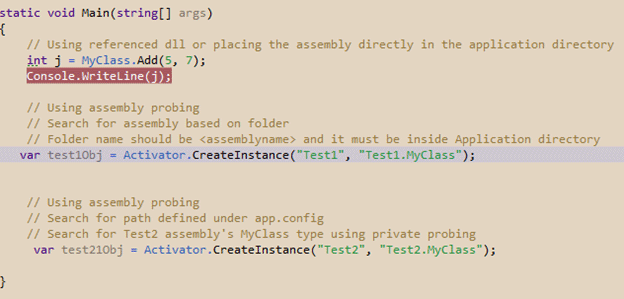
using System;  
using MyApp;  
  
class C{  
static void Main()  
{  
MathC obj = new MathC();  
obj.Accept(10,20);  
Console.WriteLine("Obj value is"+obj.Sum());  
}  
}  
Save it as ClientApp.cs  
If we compile csc ClientApp.cs it gives error  
The correct compilation is csc /r: mydll.dll ClientApp.cs   
/r refers to reference switch.

**1. Creating a simple assembly and referencing other assembly to it**  
  
I have a created an assembly called "AssemblyHost" and added reference to "Test" assembly. I am calling method "Add" defined within Test assembly's class.  
  
**Test Assembly**  
  
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
   
namespace Test  
{  
    public class MyClass  
    {  
        public static int Add(int i,int j)  
        {  
            return (i + j);  
        }  
    }  
}  
  
**AssemblyHost assembly**  
  
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using Test;  
  
namespace AssemblyHost  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            int j = MyClass.Add(5, 7);  
            Console.WriteLine(j.ToString());  
            Console.ReadLine();  
        }  
    }  
}   
  
  
**Output**  
  
  
**2. Assembly Probing - .Net runtime searches for the assembly in the following path :**

* Look for the dll or exe under the Application directory (generally bin folder) or AppBase - Example shown above works in this fashion
* Look for the assembly folder under the Application directory (generally bin/<assemblyname>) and try to find dll or exe .
* Private probing using configuration file where you can define probe path. The same can also be done using Dotnet configuration utility (mscorcfg.msc) which is available in older versions of Dotnet. Dotnet 4.0 does not have this.

     
Suppose I create one more project "Test1" as seen below :   
  
  
When I try to load the new assembly without referencing, I get an error. However, if copy and paste a folder "Test1" under my application directory (bin/Debug) and copy the dll... works  
  
**Application Directory - AssemblyHost/bin/Debug  
  
Folder Path - AssemblyHost/bin/Debug/Test  
  
Dll Path - Folder Path/Test.dll**   


* Look for probing path defined under configuration file

I have to define my application config file as seen below to search for dll or exes to be loaded into my running assembly. The below configuration searches within MyLib folder inside my Application Directory and searches for all dlls.  
  
<?xml version="1.0" encoding="utf-8" ?>  
<configuration>  
   
    
    <runtime>  
      <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">  
        <probing privatePath="MyLib"/>  
      </assemblyBinding>  
    </runtime>  
    
   
</configuration>   


using System;

using System.Collections.Generic;

using System.Text;

namespace SampleLibrary

{

    public class Algebra

    {

        public double Addition(double x, double y)

        {

            return x + y;

        }

        public double Subtraction(double x, double y)

        {

            return x - y;

        }

        public double Multiplication(double x, double y)

        {

            return x \* y;

        }

        public double Division(double x, double y)

        {

            return x / y;

        }

    }

}

using System;

using SampleLibrary;

public class Exercise

{

    static void Main()

    {

        Algebra alg = new Algebra();

        double number1 = 100;

        double number2 = 50;

        double result = alg.Addition(number1, number2);

        Console.Write(number1);

        Console.Write(" + ");

        Console.Write(number2);

        Console.Write(" = ");

        Console.WriteLine(result);

    }

}