

Assembly Types & Execution

In this chapter we will discuss about different assembly types in .NET. Basically, we have three different types of assemblies namely:

- ✓ **Private Assembly**
- ✓ **Shared Assembly**
- ✓ **Satellite Assembly**

Out of these, we are going to deep dive into first two & see other aspects related to them apart from normal basics. We are not going to see satellite assemblies in this chapter

We know .Net based projects are typically going to get compiled into EXE/DLL format. In generic way we will call this output as an assembly. This output – EXE/DLL - is not like any normal windows based binary EXE/DLLs. Rather the packaging structure is different. It is suitable to support .NET features like language interoperability, memory management, GC by CLR.

Assembly (this is what we will refer to EXE/DLL) has different sections into it. Let us say, that we have created a project using language like C#/VB.Net etc. We have used some images like resources into the project. We have used some types (integer, character, custom classes etc.) in the project. After we compile this project, we get an output i.e. assembly.

What this assembly has?

Well, the output assembly has majorly four sections:

- ✓ **Assembly Metadata**
- ✓ **Resources**
- ✓ **Type Metadata**
- ✓ **MSIL**

Let us see these sections:

- ✓ **Assembly metadata:** This section contains information regarding the version of the assembly, optionally company name etc. This is the section which will be acting just like an index for rest of the sections.
- ✓ **Resources:** This section will contain resources used in the project like images. These can be references or embedded resources.
- ✓ **Type Metadata:** This section has manifest or data regarding the types referred in the project like types used.
- ✓ **MSIL:** This section is most important. After compilation of the C#, VB.NET code, language specific compiler converts the code into intermediate language. MSIL stands for **M**icrosoft **I**ntermediate **L**anguage. Since, the code is not yet in binary or machine specific format, it can be ported on any machine with .NET framework available (.NET framework with the version on which application is built or higher version). **JIT (Just in Time)**

compiler available with framework does the job of converting this MSIL code in to the machine specific code.

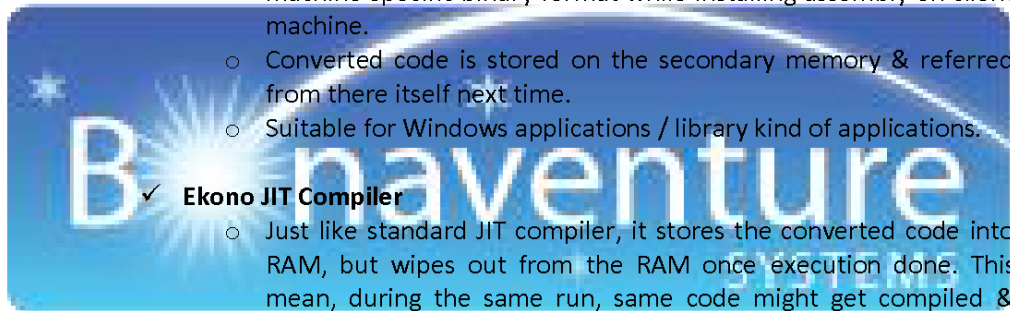
JIT in turn has three different **flavors**:

✓ **Standard JIT Compiler:**

- Shipped by default
- Used / Called by default by CLR
- Compiles the code in to machine specific binary format
- Keeps the code in RAM
- If application goes down/ machine's power off - machine code is gone!
- Next time – it starts again
- Suitable for web applications where the machine & web application keeps on running 24 X 7

✓ **Pre – JIT Compiler:**

- Need to call specifically
- Use **NGEN.EXE (Native Code Generator)** to convert the code in machine specific binary format while installing assembly on client machine.
- Converted code is stored on the secondary memory & referred from there itself next time.
- Suitable for Windows applications / library kind of applications.



✓ **Ekono JIT Compiler**

- Just like standard JIT compiler, it stores the converted code into RAM, but wipes out from the RAM once execution done. This mean, during the same run, same code might get compiled & cleaned from the RAM number of times.
- It works as -> Compile MSIL on reference -> Store binary in RAM -> Wipe after execution -> Compile MSIL on reference -> so on ...
- Suitable for mobile based application, where RAM is very less & need to be used effectively.

Let us discuss now, **how an assembly executes**:

Normal binary EXE (just like one you create using C++) has structure fixed. It has text section, data section, binary statements etc. This structure is very well known to windows runtime who executes the EXE on behalf of operating system (We are talking about Windows OS). So, here is what happens around when you click on the normal EXE in windows environment:

- ✓ Once you double click, Windows OS gets interrupted (all time hit event☺).
- ✓ Windows runtime comes in picture.
- ✓ Then runtime tries to see what happened.
- ✓ Tries to get control over the EXE for execution.
- ✓ EXE has an entry point which is well known to windows runtime.

- ✓ Windows runtime realizes that it can execute the EXE & it knows the structure very well.
- ✓ It then, allocates memory for execution.
- ✓ Reads text section, data section & executes statements.

Now, see what will happen around when you double click on the .NET EXE assembly:

Wait! Before that let us discuss one more term – **EXE & DLLs:**

One of the important point to note here is by default **DLLs can't execute them self.**

They can't have their own memory for execution. **They are parasites!**

It is just like a Client – Server terminology. **How come?**

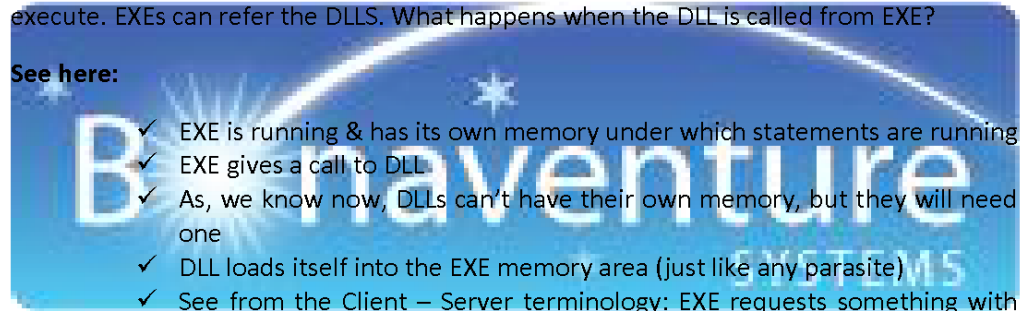
See,

What Client does? – Always requests for something to Server.

What Server does? – Always responds to the client's requests.

Same here, if you see DLLs are meant for reference. They just can't execute. EXEs can execute. EXEs can refer the DLLs. What happens when the DLL is called from EXE?

See here:



- ✓ EXE is running & has its own memory under which statements are running
- ✓ EXE gives a call to DLL
- ✓ As, we know now, DLLs can't have their own memory, but they will need one
- ✓ DLL loads itself into the EXE memory area (just like any parasite)
- ✓ See from the Client – Server terminology: EXE requests something with input & DLL replies with output.
- ✓ So, DLL is active as a Server & EXE is acting as a Client
- ✓ But something more interesting is: Server (DLL) runs inside the memory of Client (EXE) process.
- ✓ So, DLLs are also called as In-Process Server (that runs in the process of Client). In short we refer these as In-Proc Server.

Now, let us see what will happen around when you double click on the .NET EXE assembly:

- ✓ You double click on the .NET EXE which is an assembly.
- ✓ Windows runtime is called.
- ✓ Runtime tries to analyze the structure.
- ✓ It sees a statement in - so called PE(Portable Executable) header (acts just like an entry point) - which calls a DLL called MSCORLIB.DLL
- ✓ If this DLL exists on your machine then no worries.
- ✓ If it fails to find one then execution stops after an exception that the DLL was not found.

- ✓ What this DLL is? Let us discuss about this first & then continue the execution part.

What is MSCORLIB? (Alias MSCOREE – Microsoft Common Object Runtime Execution Environment)

It is CLR (Common Language Runtime) – heart & soul of .NET framework!

About Common Language Runtime:

CLR is a heart of .NET framework. It does lot of important tasks like:

- ✓ MSIL into native code conversion through JIT compiler
- ✓ Execution
- ✓ Memory allocation
- ✓ Garbage Collection
- ✓ Exception Handling
- ✓ Type Loader
- ✓ Security Check
- ✓ COM Marshalling
- ✓ Many more

Since, these are very core tasks, so is the CLR in terms of .NET.

If you have noticed above, it's a DLL. So what? - Obviously, it will need memory for execution. Where will CLR get the memory? We know from previous section that DLLs don't have their own memory space & always execute inside the EXE area. So, what happens in this case? Will there be multiple CLR's when you run multiple applications (EXEs) on your machine?

CLR is a special kind of DLL. It can have its own memory for execution.

If you remember, in VB 6.0, apart from conventional DLLs, we have another library like concept which can have its own memory for execution. It was **ActiveX EXE** – kind of library which needs to be referred & can have its own memory!

But, we don't have such ActiveX concept here.

Rather **in terms of out of process memory, similar concept here is Shared Assembly.**

We will see shared assembly concept in more detail later in this chapter.

If so far the things are clear then let us continue to our discussion about execution:

- ✓ So, now the MSCORLIB.DLL (**we will refer to this as CLR from here onwards**) exists if you have .NET framework exists on your machine. It is then loaded into a very special area called as **Domain Neutral Assembly** area or **Shared Assembly** area

- ✓ After this, the CLR takes over the control of execution from Windows runtime & starts executing an assembly.
- ✓ So, what it does then?
- ✓ CLR starts reading the assembly top to bottom.
- ✓ First it reads the **assembly metadata** – with which it can understand the rest of the structure of an assembly
- ✓ It then, **loads the resources, types** required to run the assembly or program
- ✓ Then. It gives a call to **JIT compiler to convert the MSIL code into the native code.**
- ✓ Depending on what type of application (Web/Windows/Mobile) specific **JIT compiler (Standard/Pre/Ekono)** can be called or loaded depending upon the case.

Who allocates the memory for application EXE or in this case our EXE assembly then?

Obviously, CLR does this. When?

When it comes in picture for the first time, it gets some 'X' amount of memory from OS for management. Then onwards CLR will act as if a virtual operating system. Every program (assembly running) under the CLR will ask to CLR for any memory issues.

Any program will not directly talk with OS at any time. So, CLR acts as a virtual OS within the Root OS.

So, EXE gets the memory from CLR. We will discuss about the memory management & garbage collection algorithm in details later in the book.

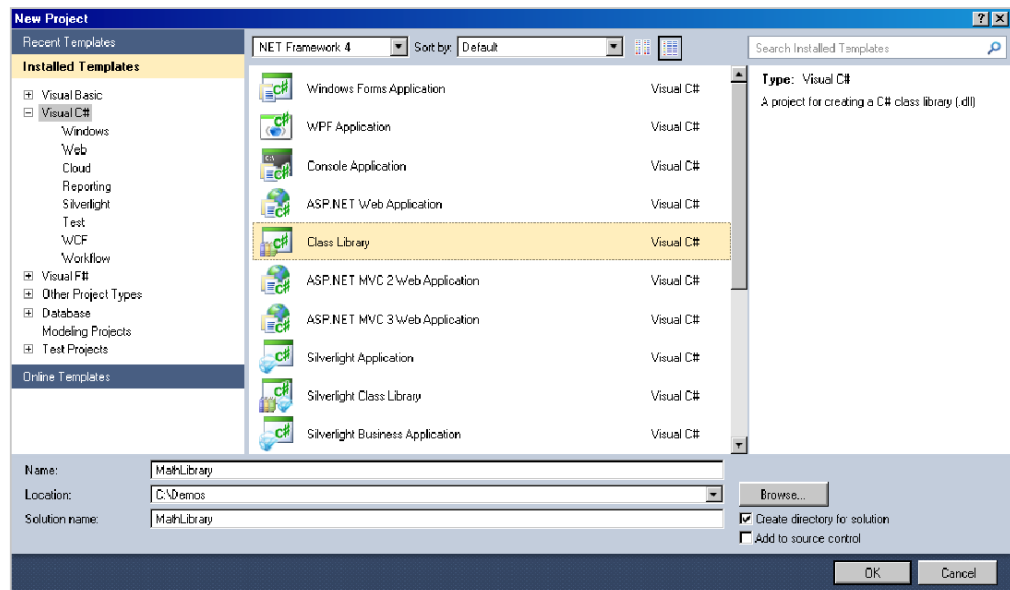
We talked about the EXE, DLL assemblies. But what are private & shared assemblies?

By default every assembly is a private assembly.

Let us create a program to understand private assembly concept first.

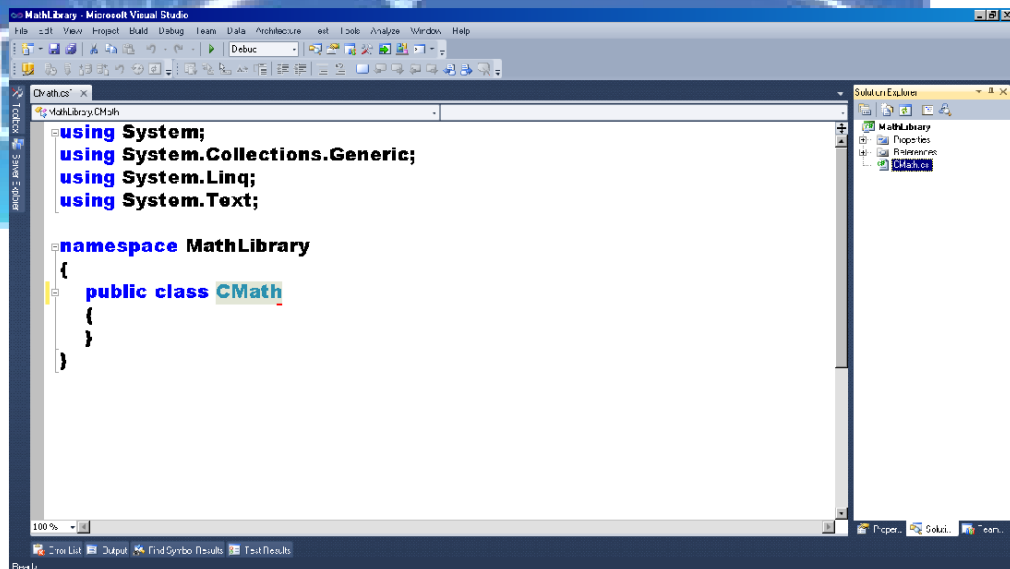
Steps:

1. Start Visual Studio 2010 by clicking on: **Start -> All Programs -> Microsoft Visual Studio 2010 -> Microsoft Visual Studio 2010 Icon**
2. Click on: **File -> New -> Project**
3. In the dialog: Choose C# as a language → Choose project type as Class Library → Name the project as MathLibrary as shown:

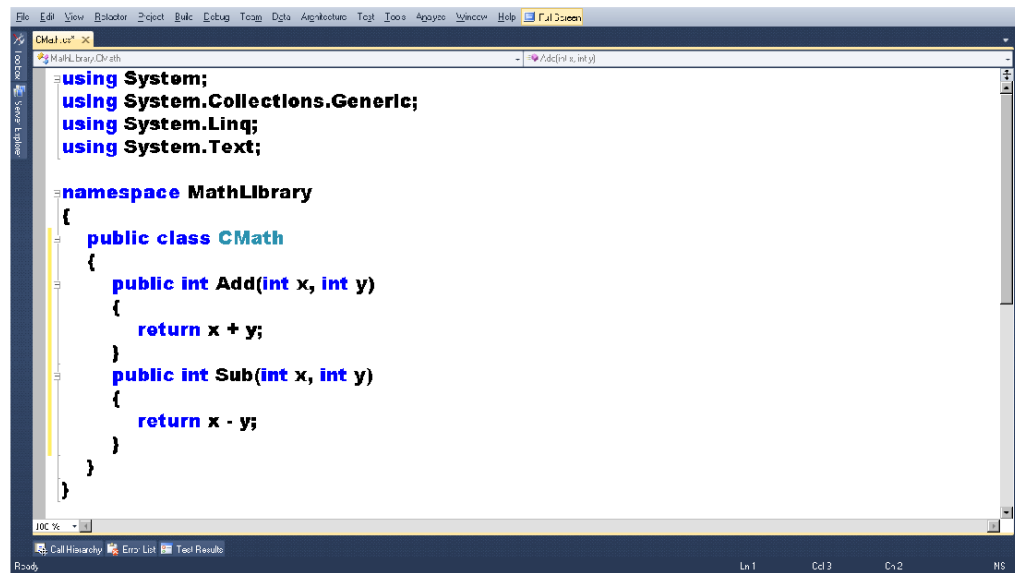


4. Click on Ok Button.

5. Name the class file as CMath.cs in the solution explorer as well as in the open CS file as shown:



6. Add code for Add & Subtract methods as shown:

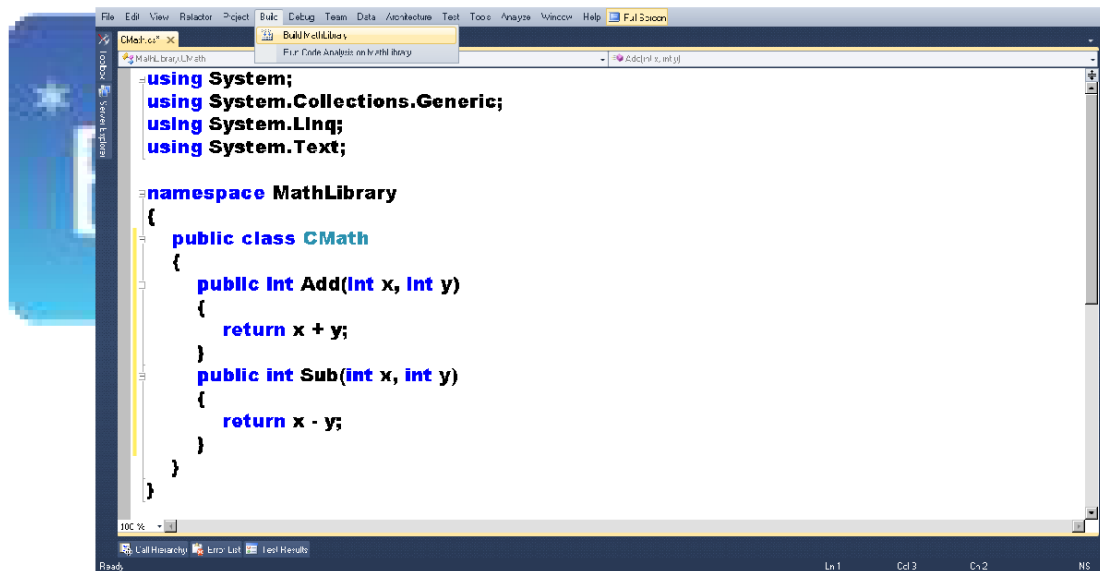


A screenshot of the Visual Studio IDE. The main editor window displays the code for `CMat.cs`. The code includes using statements for `System`, `System.Collections.Generic`, `System.Linq`, and `System.Text`. It defines a `MathLibrary` namespace containing a `CMath` class with two methods: `Add(int x, int y)` and `Sub(int x, int y)`. The `Add` method returns `x + y`, and the `Sub` method returns `x - y`. The Solution Explorer on the left shows the project structure with `MathLibrary\CMat.cs` selected. The status bar at the bottom indicates 'Ready' and shows line and column coordinates.

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

namespace MathLibrary
{
    public class CMath
    {
        public int Add(int x, int y)
        {
            return x + y;
        }
        public int Sub(int x, int y)
        {
            return x - y;
        }
    }
}
```

7. Compile the code:

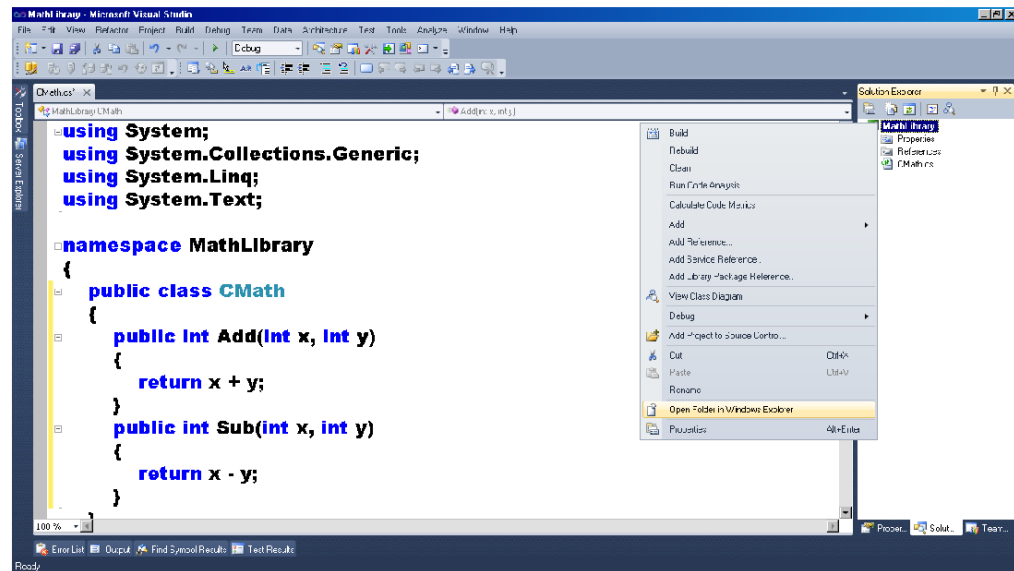


A screenshot of the Visual Studio IDE, similar to the previous one, but with a compilation error. The code is the same as in the first screenshot. However, the `CMat.cs` file in the Solution Explorer is highlighted in red, indicating an error. The error list at the bottom shows a 'Build' error: 'Error 1: The type 'int' is not defined. Please check the namespace or assembly reference.' This error occurs because the `int` type is not explicitly imported by the using statements. The status bar at the bottom now shows 'Error' instead of 'Ready'.

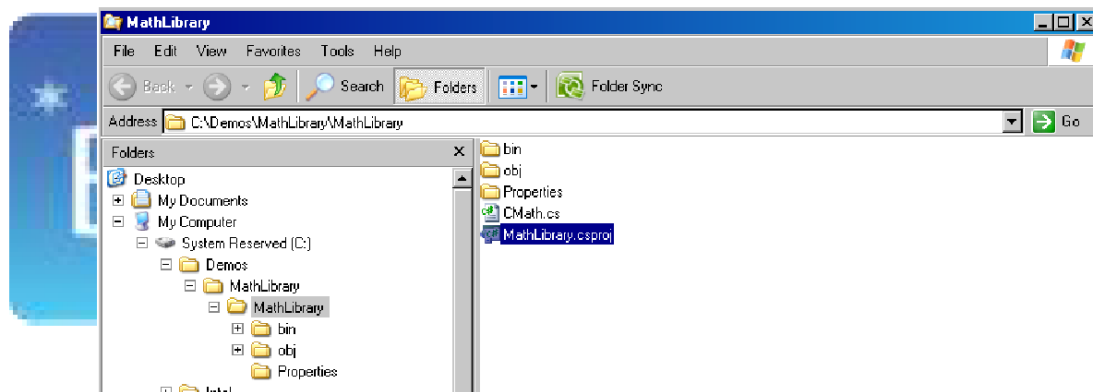
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace MathLibrary
{
    public class CMath
    {
        public int Add(int x, int y)
        {
            return x + y;
        }
        public int Sub(int x, int y)
        {
            return x - y;
        }
    }
}
```

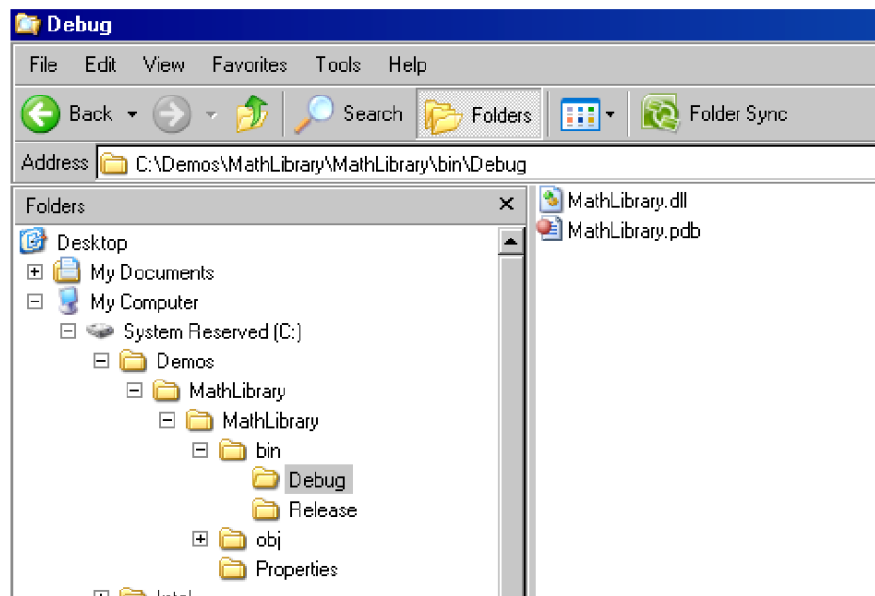
8. Right click on the solution explorer & click on Open Folder in Windows Explorer:



9. It opens:



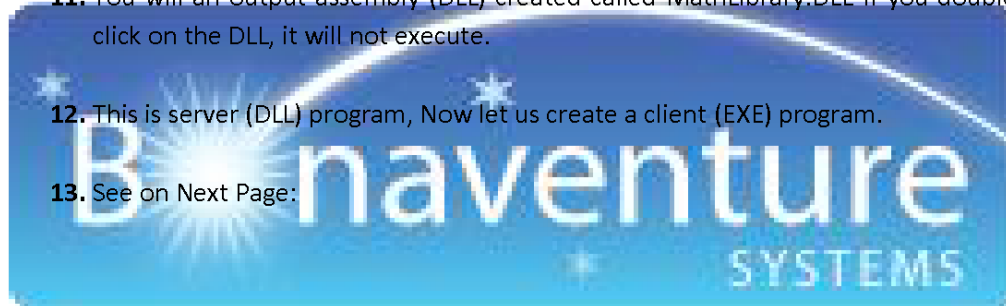
10. Go to bin-> debug folder



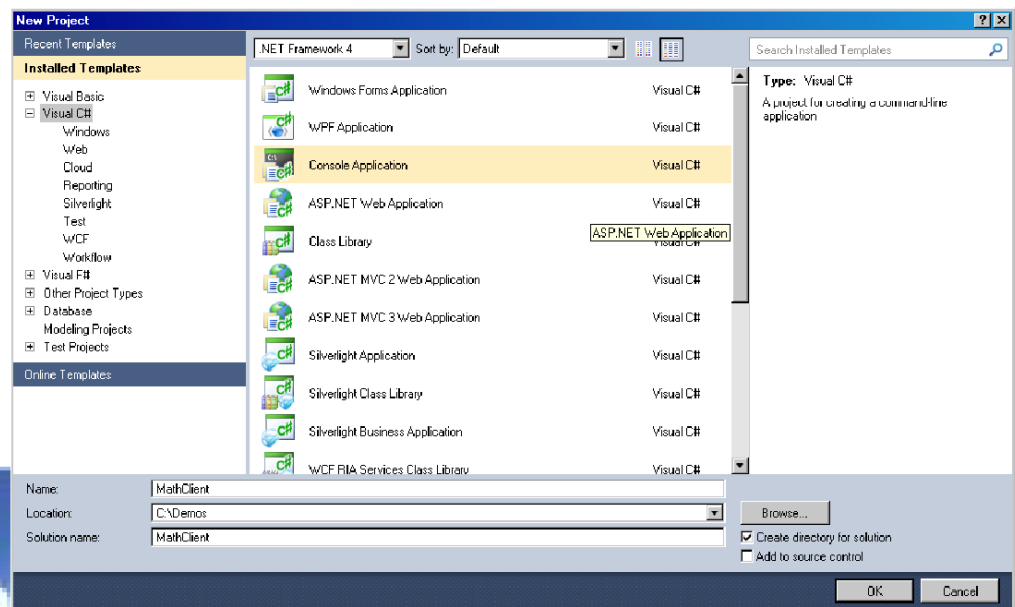
11. You will an output assembly (DLL) created called MathLibrary.DLL If you double click on the DLL, it will not execute.

12. This is server (DLL) program, Now let us create a client (EXE) program.

13. See on Next Page:

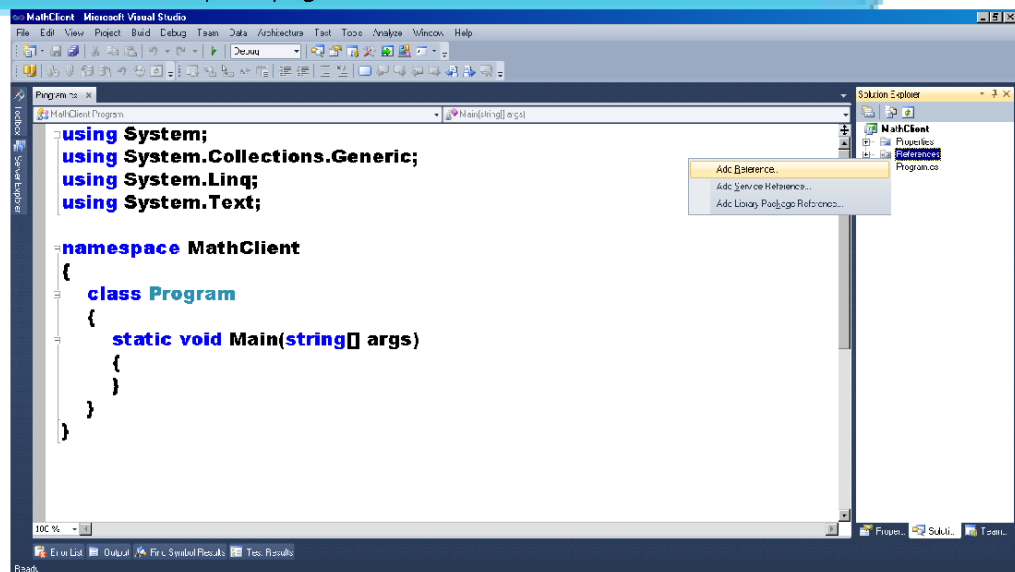


14. Start one more time Visual Studio 2010 by clicking on: **Start -> All Programs -> Microsoft Visual Studio 2010 -> Microsoft Visual Studio 2010 Icon**
15. Click on: **File -> New -> Project**
16. In the dialog: Choose C# as a language → Choose project type as Console Application → Name the project as MathClient as shown:



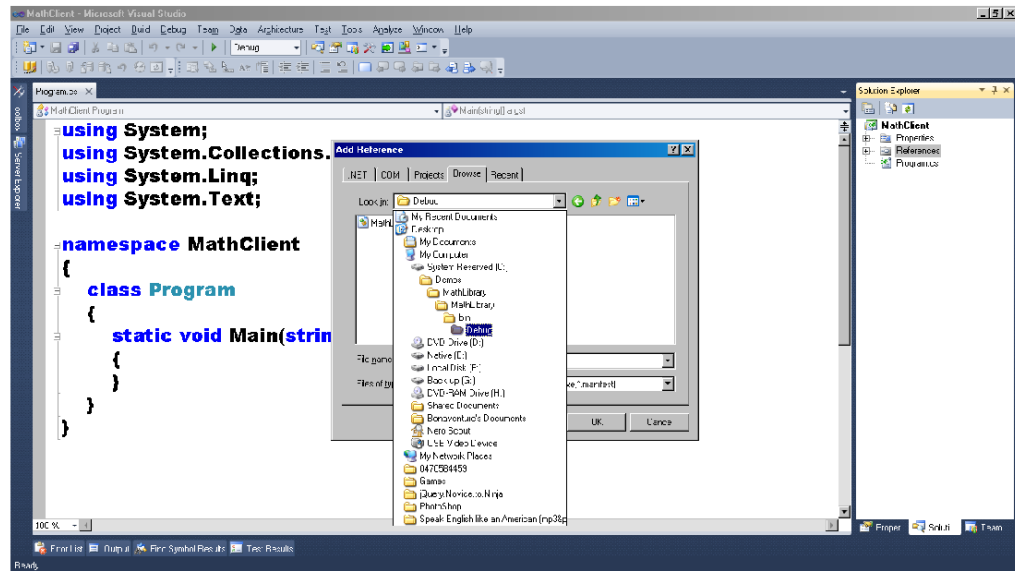
17. Click on Ok

18. In the solution explorer, right click on references & click on Add Reference:



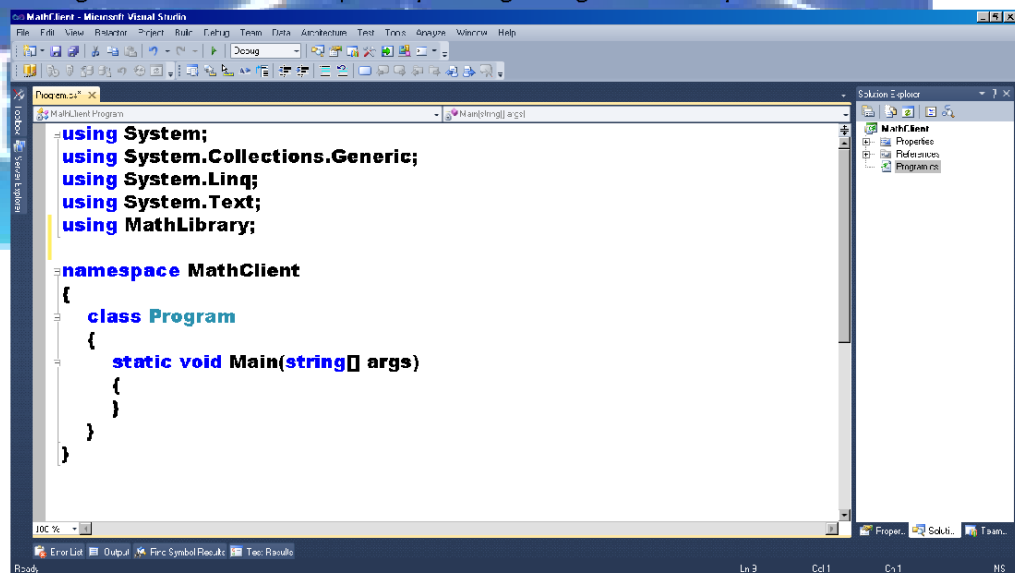
19. In the dialog that opens, click on Browse Tab:

20. Navigate to previously created DLL.



21. Click on Ok.

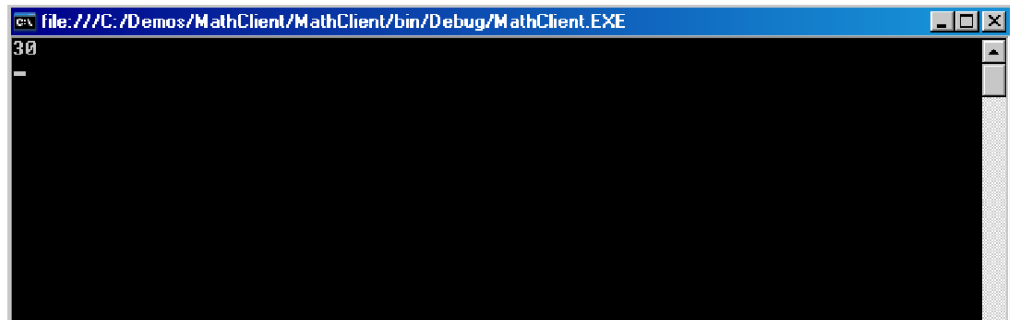
22. In Program.CS file add namespace by writing: Using MathLibrary;



23. Create object of CMath Class & call to Add function by providing inputs to it.

```
class Program
{
    static void Main(string[] args)
    {
        CMath obj = new CMath();
        int result = obj.Add(10, 20);
        Console.WriteLine(result.ToString());
        Console.ReadLine();
    }
}
```

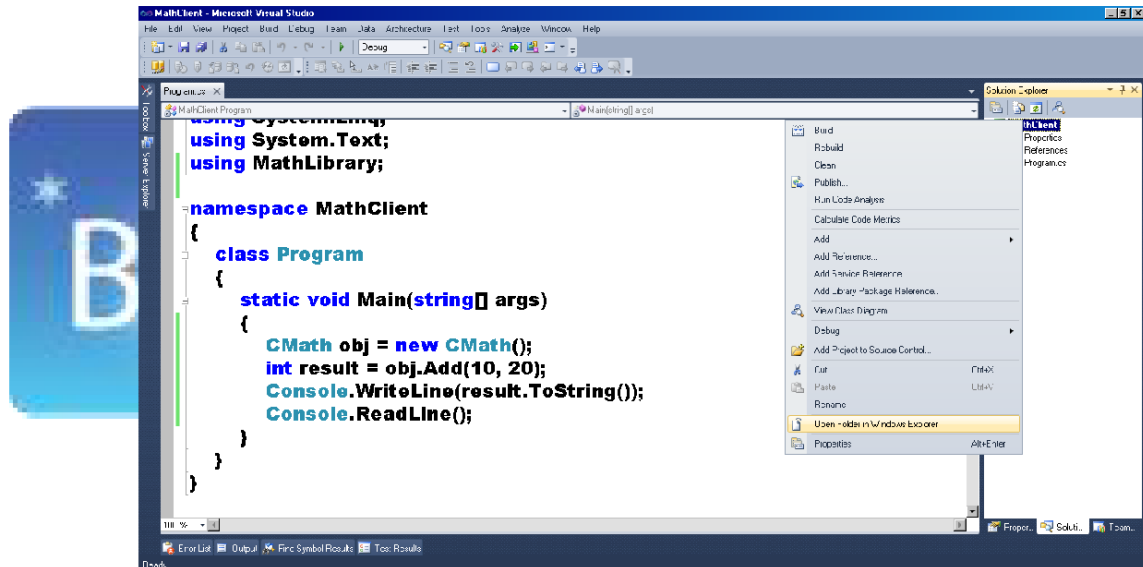
24. Press F5 to run the program. You should see the output as:



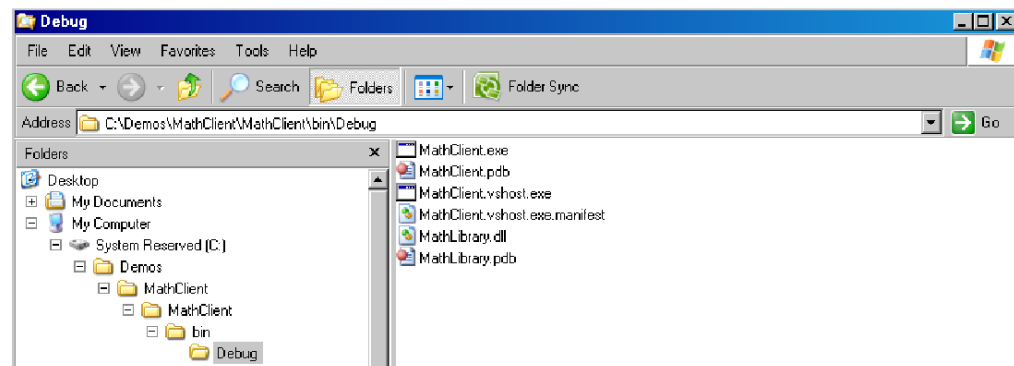
25. So, the code is fine. Client is calling Server & we are getting the output.

26. All the assembly execution that we discussed above is completely hidden

27. Now, right click on solution explorer. Click on Open Folder in Windows Explorer.



28. Observe now, what the folder bin-> debug has:



- ✓ This DLL is private assembly. This is just a simple coded DLL with nothing extra done apart from compilation.
- ✓ When CLR gives a call while executing this DLL then it simply loads the DLL inside the EXEs memory space without even checking whether the DLL is safe to execute. Somebody might have written /tampered the DLL with malicious code.
- ✓ To enable this security checking, the above mentioned keys are used namely – private key & public key.
- ✓ These are special prime numbers (one prime other co - prime) generated with the help of an algorithm called RSA (stands for three people who invented this algorithm – **R**ivest, **S**hammir, **A**dleman).
- ✓ One number is called private key & other is public key.
- ✓ Later on after generating these keys, hashing algorithm is applied on the assembly. The contents of assembly are hashed (encrypted) using hashing algorithm with private key as a second operand. This hash code is then stored in assembly metadata.
- ✓ The encrypted output can't be decrypted. Then what's the use of it?
- ✓ This hash code acts just like a checksum bits. Later on, when this DLL is to be loaded by CLR, it simply does the same task of hashing again and then tries to check old hash (from metadata) with new hash. If the comparison is successful then there is no problem. But if the comparison fails then the security exception comes.
- ✓ Public key is used for unique identification of the assembly.

Since the shared assembly has to be loaded in shared memory area or in a common area outside the client EXE, CLR must be careful while loading any assembly. It should not be tampered one. It checks the same using above discussed process. To do the above discussed process, the assembly must have private – public key pair attached.

So, to create a shared assembly one should have key pair assigned to the assembly.

Let us create one pair & attach with MathLibrary.DLL

1. Create a strong name key pair (private key + public key) using command prompt as shown below.
2. Click on start → All Programs → Microsoft Visual Studio 2010 → Visual Studio Tools → Visual Studio Command Prompt
3. Give a command “sn -k <File name(here it is key.snk)> “

```
Visual Studio Command Prompt (2010)

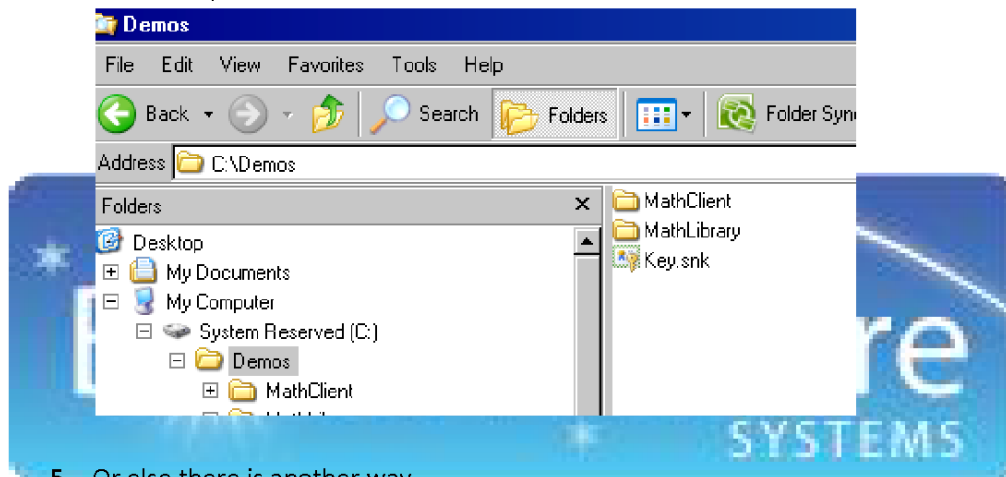
C:\Demos>sn -k Key.snk

Microsoft (R) .NET Framework Strong Name Utility Version 4.0.30319.1
Copyright (c) Microsoft Corporation. All rights reserved.

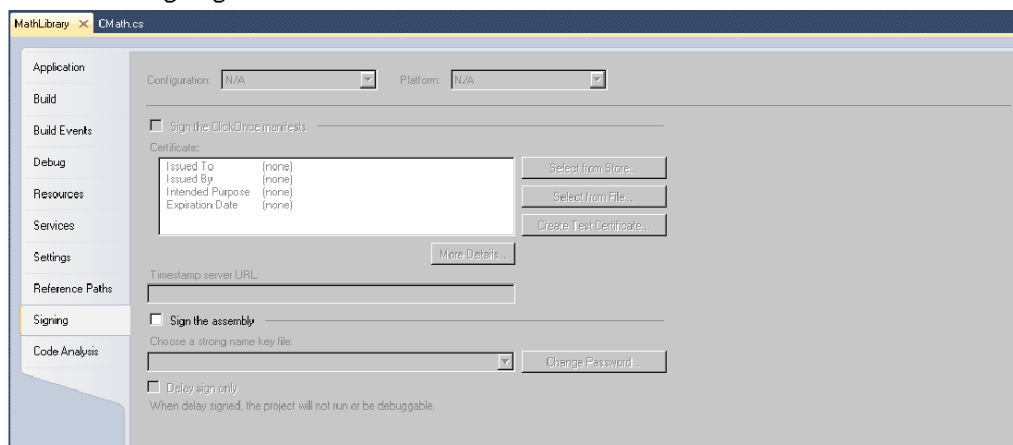
Key pair written to Key.snk

C:\Demos>
```

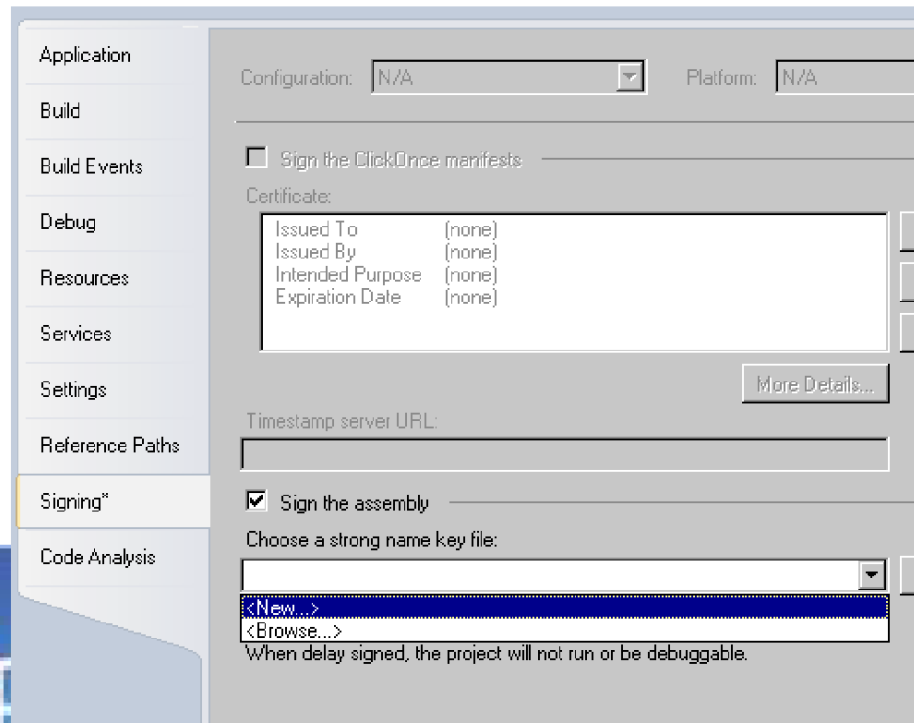
4. On C drive, see the file created.



5. Or else there is another way.
6. Right click on the MathLibrary project under solution explorer.
7. Click on Properties menu item.
8. Click on the Signing Tab



9. If you choose browse then you can navigate to the file that we created above i.e. Key.snk & compile the project.
10. Else click on sign the assembly then choose <New> from the dropdown.



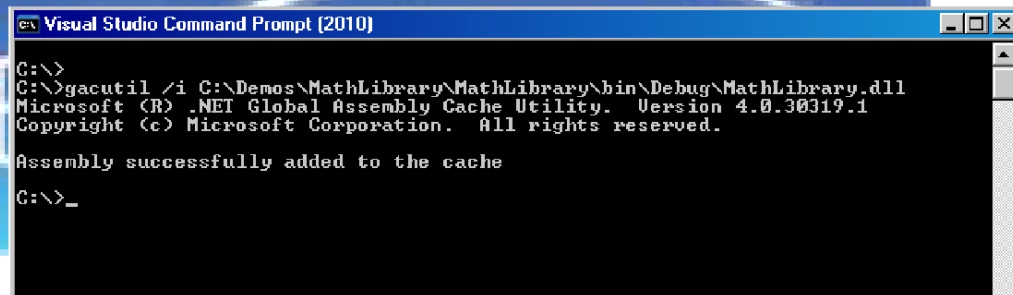
11. A dialog box will be shown



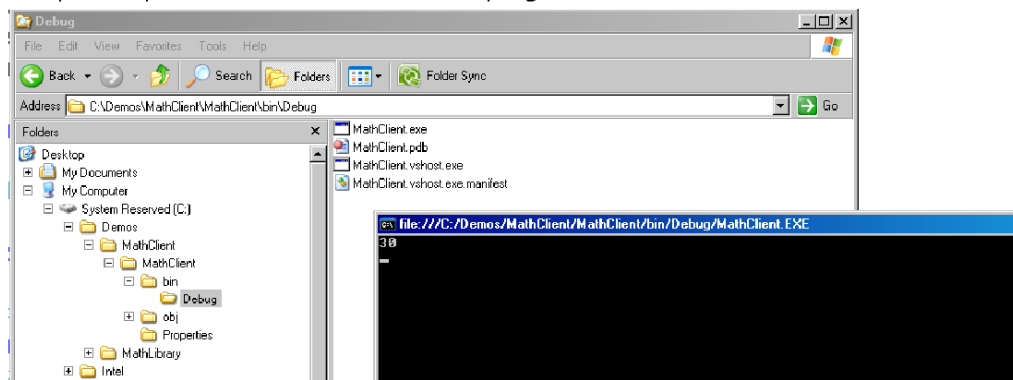
12. If you want to protect the file with password then keep the box checked & provide password in below text boxes.
13. We are not going to protect the file with password. So, let us uncheck the check box & enter name for the file which will have key pair written inside it.



14. After this click on Ok & compile the project.
15. You will see that the file exists in the solution explorer.
16. Now, we have key pair added to the DLL or assembly.
17. To install this assembly, into the global assembly cache (GAC) or shared assembly area, Click on start → All Programs → Microsoft Visual Studio 2010 → Visual Studio Tools → Visual Studio Command Prompt
18. Give a command "gacutil /i <assembly path> ". Hit enter.



19. Now open the <Native Drive>:\Windows\Assembly Folder
20. You should see the MathLibrary.dll installed.
21. Now, recompile the MathClient EXE client program.



22. See, if it runs. It works.

23. In the output folder you will not see private copy of the MathLibrary.dll since it is shared now just like CLR DLL!

24. We have successfully created shared DLL.

Generally, while executing EXE program whenever CLR sees any DLL reference then that **DLL is first searched in GAC area** by CLR. If not found in the GAC area then it searches local copy. If not found there as well then it throws an exception.

What will happen if there are two copies one at the shared area (or in GAC) & one as a private copy with EXE & their versions are different?

Generally, a DLL is referred with which reference, the EXE is built. So, if EXE is built with DLL with 1.0 version then CLR will always look for 1.0 versioned DLL. If it finds it in GAC then it is referred as a shared assembly else private copy is searched.

But to make sure that specific version is always referred one must attach key pair with the DLL.

What will happen if there are two versions of the same DLL inside the GAC? Then what will happen & which DLL will the CLR refer?

Same answer. It will refer the DLL with which reference, the EXE is built.

What if one wants to redirect reference to other versioned DLL without recompiling the EXE program?

If you want to target some other version of the DLL for the EXE built with previous / newer version of the same DLL then you will have to tell CLR to refer other DLL & tell that it's not tampering rather just a versioning policy.

So, you will have to add version policy block into the APP.config file as shown below:

This concept is called as Probing.

Probing Element in Application (EXE) Configuration file:

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="<DLL Assembly Name>"
          publicKeyToken="<public key of the DLL Assembly>"
          culture="en-us" />
        <bindingRedirect oldVersion="1.0.0.0" newVersion="2.0.0.0" />
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

This is where we will end our chapter.

Do see following list of issues that you may face while doing practical.

