C# Notes

Introduction to Microsoft .NET

Microsoft .NET is a framework developed by Microsoft which runs primarily on Windows OS. It includes code libraries also known as class libraries. It also contains core engine well known as Common Language Runtime (CLR). Installation of .NET framework comes with few language compiler(s) as well.

Let us start the discussion by asking question Why .NET?

Microsoft .NET framework offers lot many features which makes application programming easy. Here are few points about the offerings from Microsoft .NET which can be Unique Selling Point (USP):

- Over 32+ programming languages are officially supported on .NET platform
- C++, C#, VB.NET, F# languages support is default added in the installation of .NET
- All languages follow one standard called Common Language Specification
- Code written in one language can be referred into other after compilation
- Garbage Collector
- 100+ number of libraries including thousands classes gets installed with .NET framework
- For all code writing; there is one universal editor – Visual Studio
- One can develop desktop application, websites, services, mobile application using .NET
- …… So on

Now, let us try to understand how languages interoperate.

We can use multiple languages for one project! All we have to do is write the reusable code in .NET supported language and compile the same as a library i.e. DLL (Dynamic Linked Library)

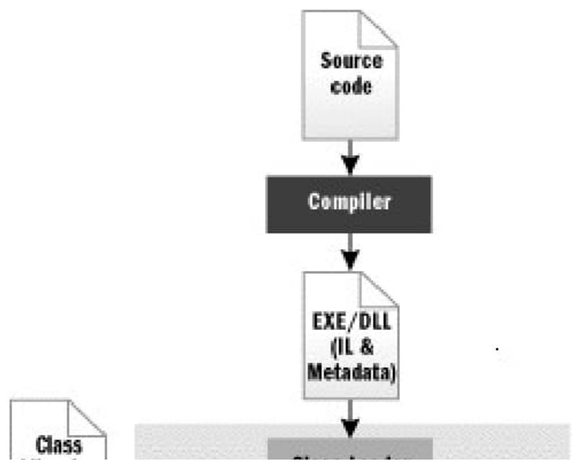
After compilation; if the code written - is - as per the CLS standard then the language specific compiler compiles the language code into common code and not machine code. Common code is referred as **M**icro**s**oft **I**ntermediate **L**anguage (MSIL). It can be seen after disassembling the output. One can use "ILDASM <.NET DLL / EXE path>" command to observe the same.

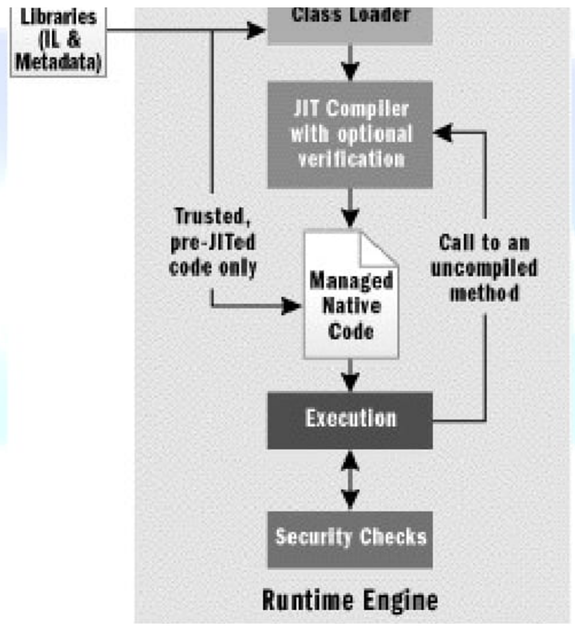
This common code adheres to standard known as Common Type System (CTS)

Types used in common code (i.e. MSIL) are well known as common types. E.g. Integer in .NET supported language is always referred as "System.Int32" in MSIL code as common type. This conversion is done by compiler by referring to Common Type System (CTS).

CPU does not understand MSIL code when it comes to execution. Somebody will have to convert the common code into machine specific code. This job is done by CLR installed along with .NET framework. In turn this responsibility is handed over to - Just in Time (JIT) Compiler. Standard compilation happens while execution demands some non compiled method which means at runtime by CLR. More discussion on the execution of .NET output is discussed in next part.

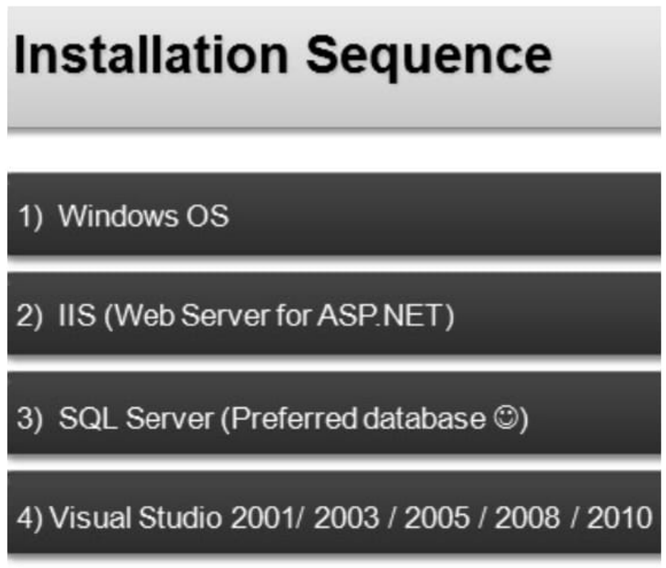
Below figure explains the procedure of execution in short:





Class libraries mentioned in the figure may refer to the reusable code libraries either given by Microsoft during installation or may be owned & created.

While installing Microsoft .NET on developer's machine; following sequence is always recommended. Figure shows editors till Visual Studio (VS) 2010. Actually we have Visual studio 2015 as the latest edition at the time of publishing this book.



Tips & Tricks:

- During installation; IIS and Visual Studio sequence matters. Normally, during the installation of Visual Studio; installer tries to find IIS installation on the machine. This is to make existing IIS know about new extensions on the machine like *.ASPX, *.ASMX etc.
- If this installation sequence goes wrong; then use “ASPNET_REGIIS /i” command on .NET command prompt
- One can open default Visual Studio by typing “DEVENV” command in ‘Windows+R” (Run) prompt

In the next part we shall see .NET cod compilation & execution in detail.

Summary:

We have observed in this part:

- Unique selling points for .NET
- How languages interoperate
- What is .NET framework
- Installation sequence

Assembly Types and Execution

In this part we will discuss about different assembly types in .NET.

In .NET; assembly has three types:

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

Out of these, we are going to deep dive into first two. We are not going to discuss satellite assemblies.

As we know .Net based projects get compiled into EXE/DLL format. In generic way we shall call the output as an assembly. This output – EXE/DLL - is not like any normal windows based binary EXE/DLLs. The packaging structure is different.

Assembly has different sections. Let us consider that we have created a project using C#. We have some image resources referred into the project. After we compile this project, we get an assembly.

So, what this assembly has?

The output assembly has four sections:

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

Details:

- ✓ **Assembly metadata:** This section contains information regarding the version of the assembly, optionally company name etc. This is the section which acts 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 information regarding the types referred in the project.
- ✓ **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**icro**s**oft **I**ntermediate **L**anguage. Since, the code is not yet in binary or machine specific format, it can be ported on any machine which has .NET framework available (.NET framework with the version on which application is built or higher version). **JIT** (**Just i**n **T**ime) compiler available with framework does the job of converting this MSIL code in to the machine specific code.

JIT compile has 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 gets switched off; machine code is removed from the RAM!
  - 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 explicitly
  - Use **NGEN.EXE (N**ative Code **Gen**erator**)** to convert the code in machine specific binary format while installing assembly on client machine.
  - Converted code is stored on the secondary memory
  - Suitable for Windows applications / library kind of applications.

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

Normal binary EXE (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 which executes Executable on behalf of operating system (with reference to Windows OS). So, here is what happens when you click on the normal EXE in windows environment:

- ✓ Once you double click, Windows OS gets interrupted
- ✓ Windows runtime is called
- ✓ EXE has an entry point - which is well known to windows runtime.
- ✓ Windows runtime allocates memory for execution.
- ✓ Reads text section, data section & executes statements.

Before we see what happens when you double click on the .NET EXE / assembly; let us discuss one more term – EXE & DLLs

DLLs can't be executed directly.

DLLs can't have owned memory for execution. DLLs are parasites!

Communication between EXE and DLL is like a Client – Server terminology.

How?

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

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

If you see DLL is meant for reference. It can't execute. EXE can execute. EXE can refer the DLL. So, what happens when the DLL is called from EXE?

Details:

- ✓ When EXE starts execution; it has memory allocated by runtime
- ✓ If EXE refers & calls DLL then DLL consumes memory inside the EXE itself
- ✓ Which means if we consider DLL as Server and EXE as client; DLL gets memory inside client process i.e. EXE
- ✓ In other words; it's like server program gets memory inside client process. So, DLL is also known as InProcess or in-Proc server.

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

- ✓ You double click on the .NET EXE.
- ✓ Windows runtime gets called.
- ✓ It has entry point which is also known as PE (Portable Executable) header. A statement inside PE header calls a DLL named MSCORLIB.DLL
- ✓ If this DLL exists on your machine then no worries.
- ✓ If it fails to find one then execution stops after an exception – "MSCORLIB.DLL was not found"
- ✓ What is this DLL? Let us discuss about this first & then continue the execution discussion.

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 discussion that DLLs don't have own memory space & always execute inside the EXE's memory area. So, what happens in this case? Will there be multiple CLRs 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 (Reference to COM), 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 shall see shared assembly concept in more detail later in this part.

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

- ✓ So, now the MSCORLIB.DLL (**we will refer to this library as CLR from here onwards**) gets loaded into a very special area called as **Domain Neutral Assembly** area or **Shared Assembly** area. This memory actually belongs to another EXE process called as Surrogate EXE process.
- ✓ After this, the CLR takes over the control of execution from Windows runtime & starts executing an assembly
- ✓ 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**.

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

Obviously, CLR does this. When?

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

Remember that 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 shall discuss about the memory management & garbage collection algorithm in details later in the book.

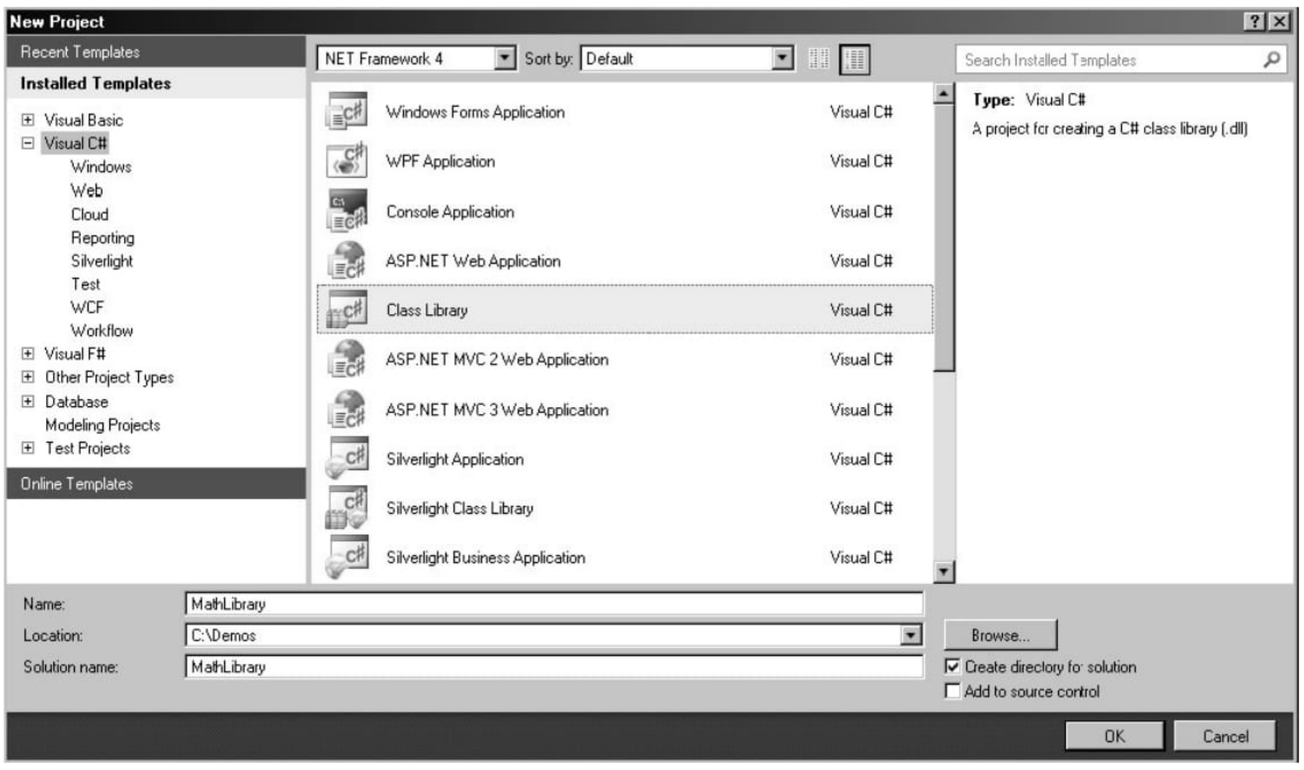
We talked about the EXE, DLL assemblies. But what are private & shared assemblies?

By default every assembly is a private assembly.

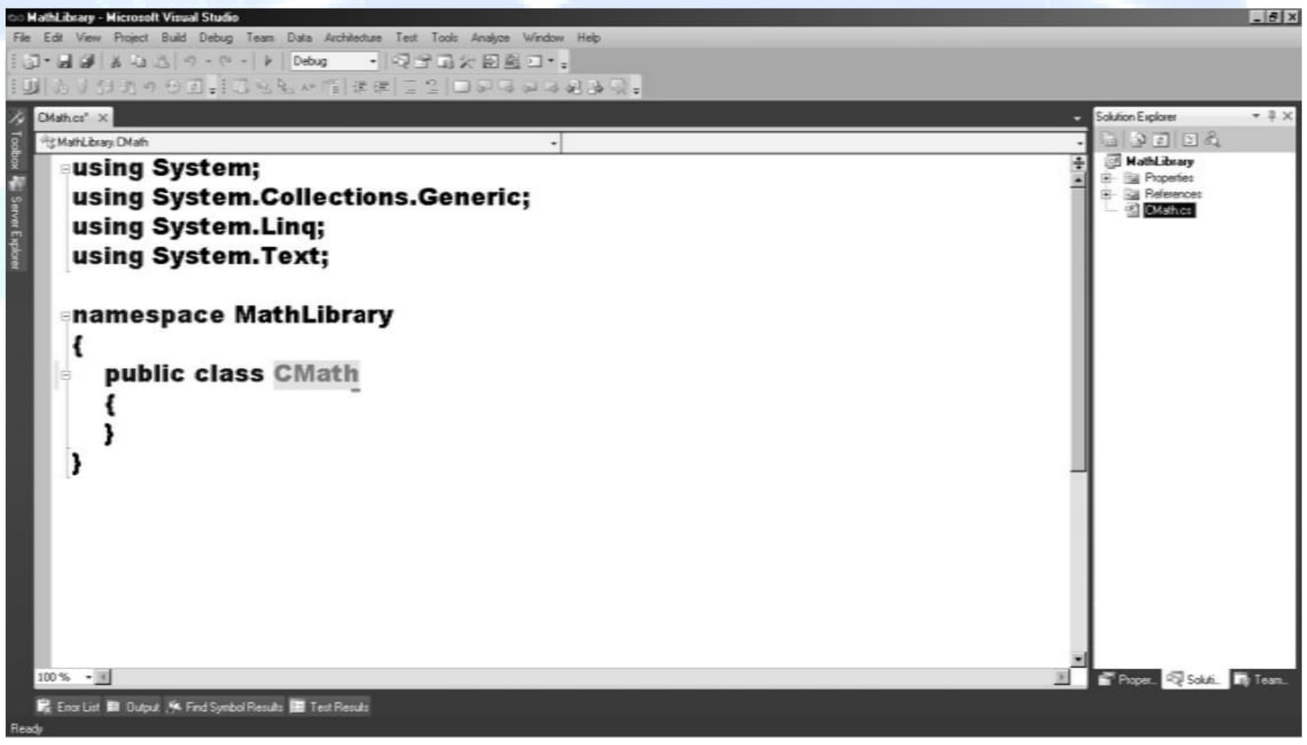
Let us create a program to understand private assembly concept first.

Steps:

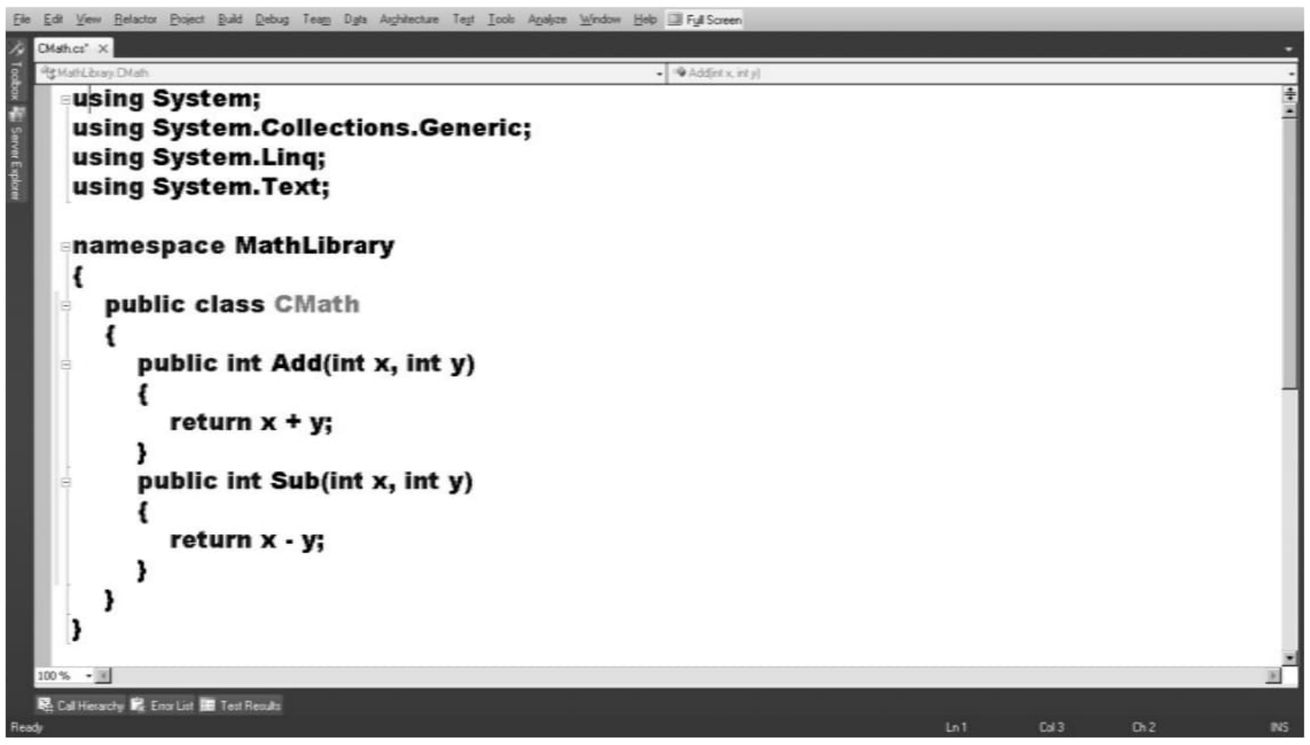
1. Start Visual Studio. Click : **Start -> All Programs -> Microsoft Visual Studio ->Microsoft Visual Studio Icon**
2. Click: **File -> New -> Project**
3. In the dialog box; select C# as a language → select project type as Class Library → Name the project as "MathLibrary" as shown:



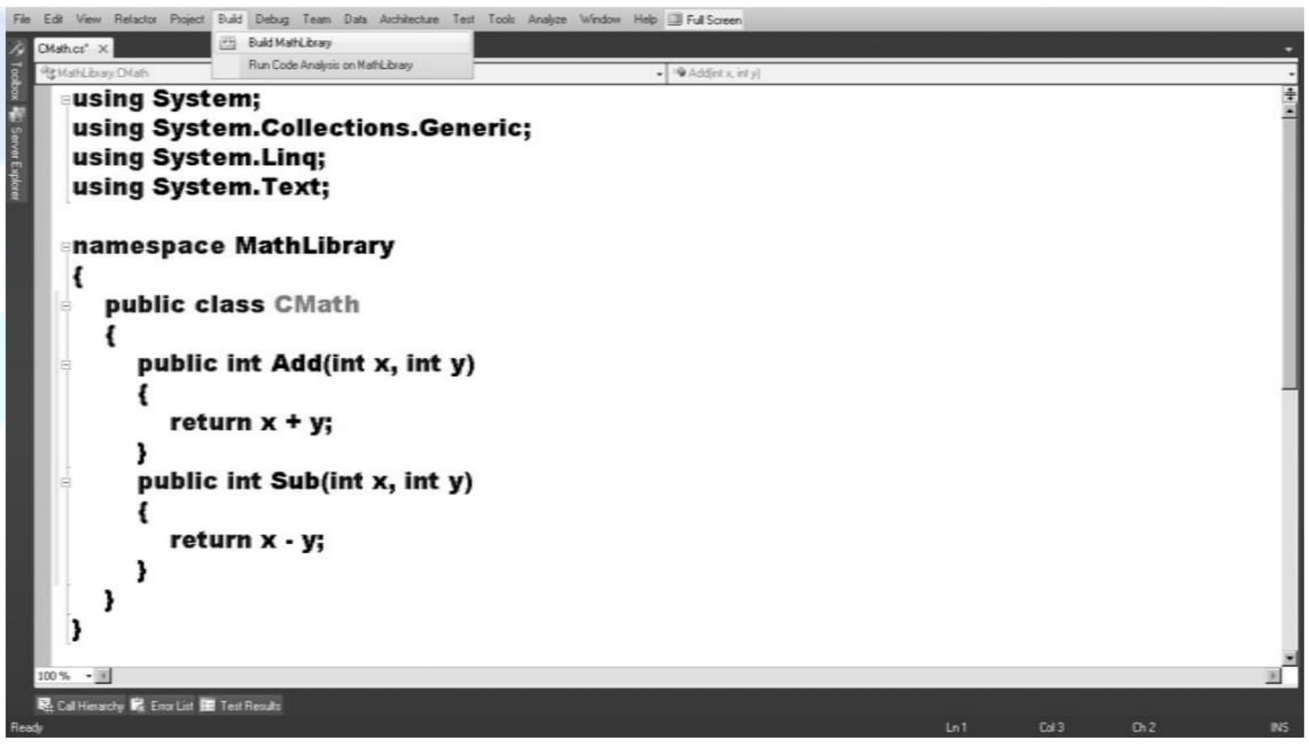
4. Click “Ok”.
5. Name the class file as “CMath.cs” in the solution explorer as well as in the open CS file as shown:



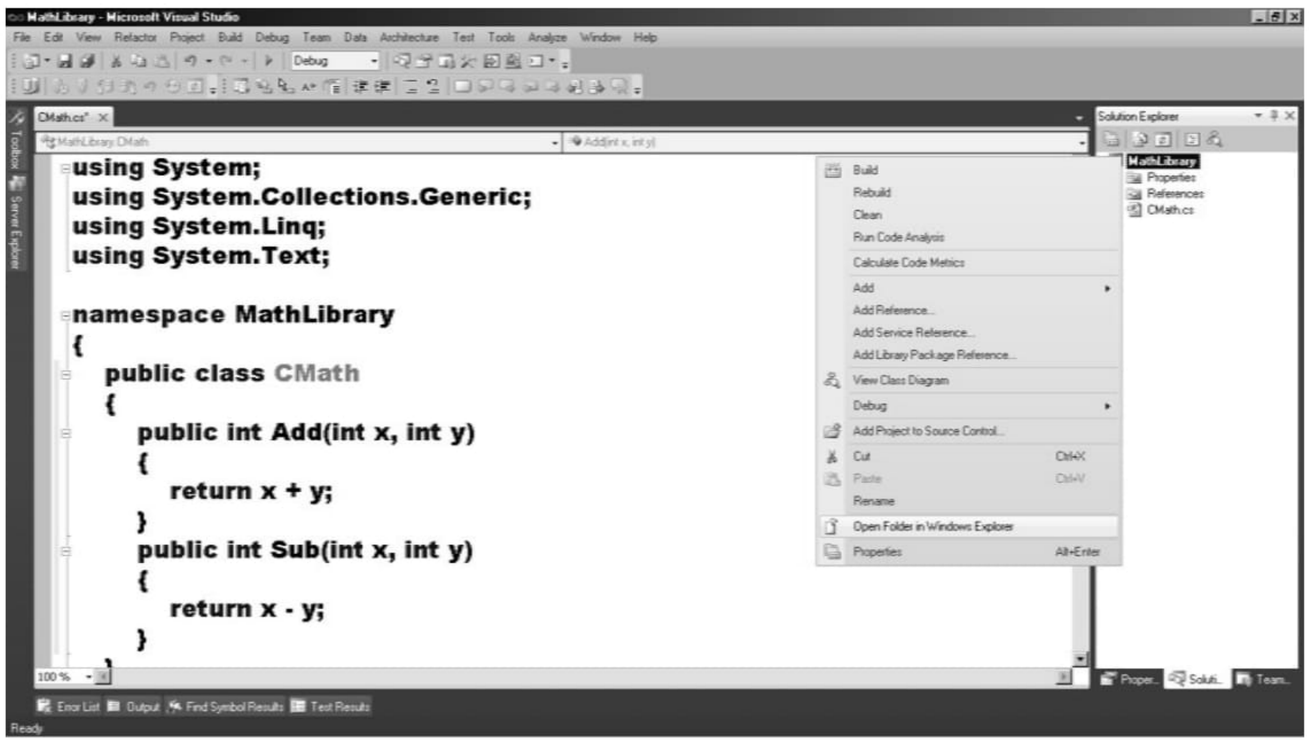
6. Add code for Add & Subtract methods as shown:



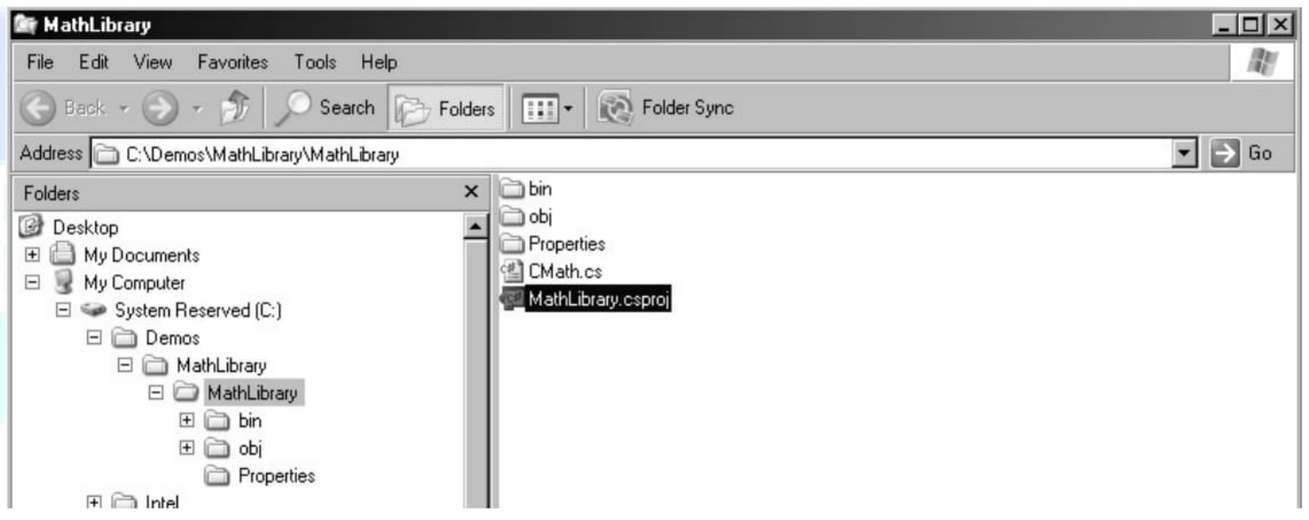
7. Compile the code:



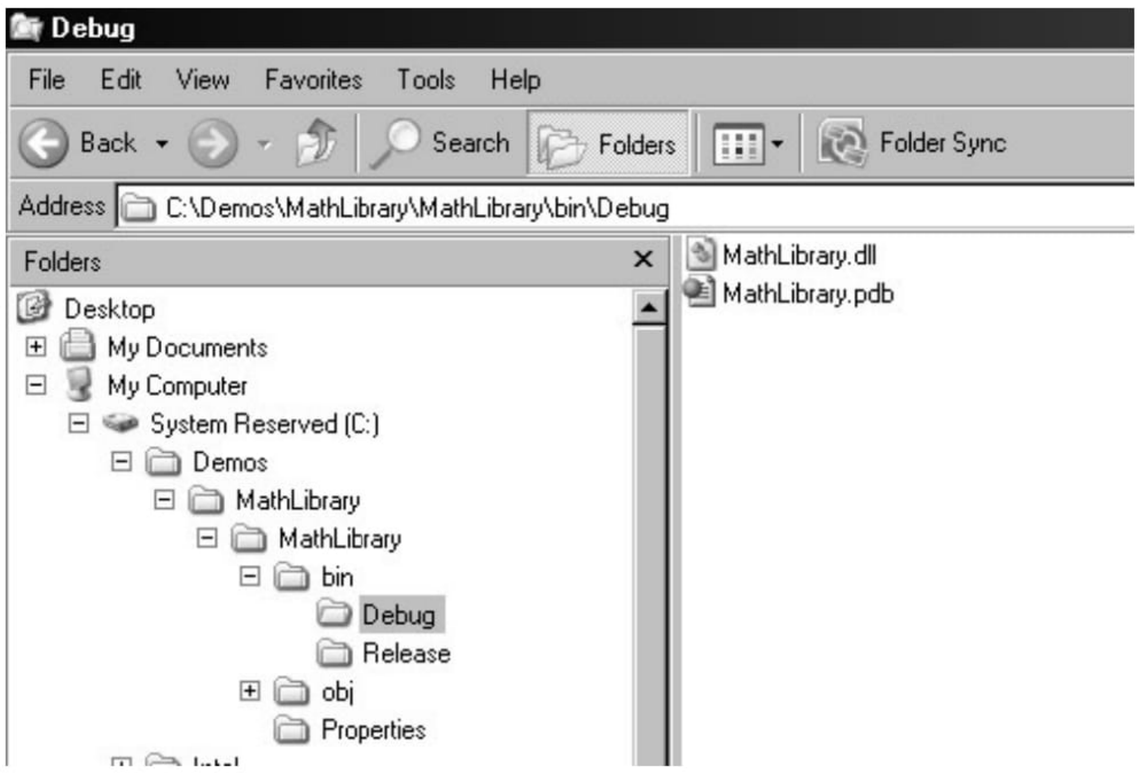
8. Right click on the solution explorer & click Open Folder in Windows Explorer:



9. This action opens the project folder:



10. Go to bin-> debug folder

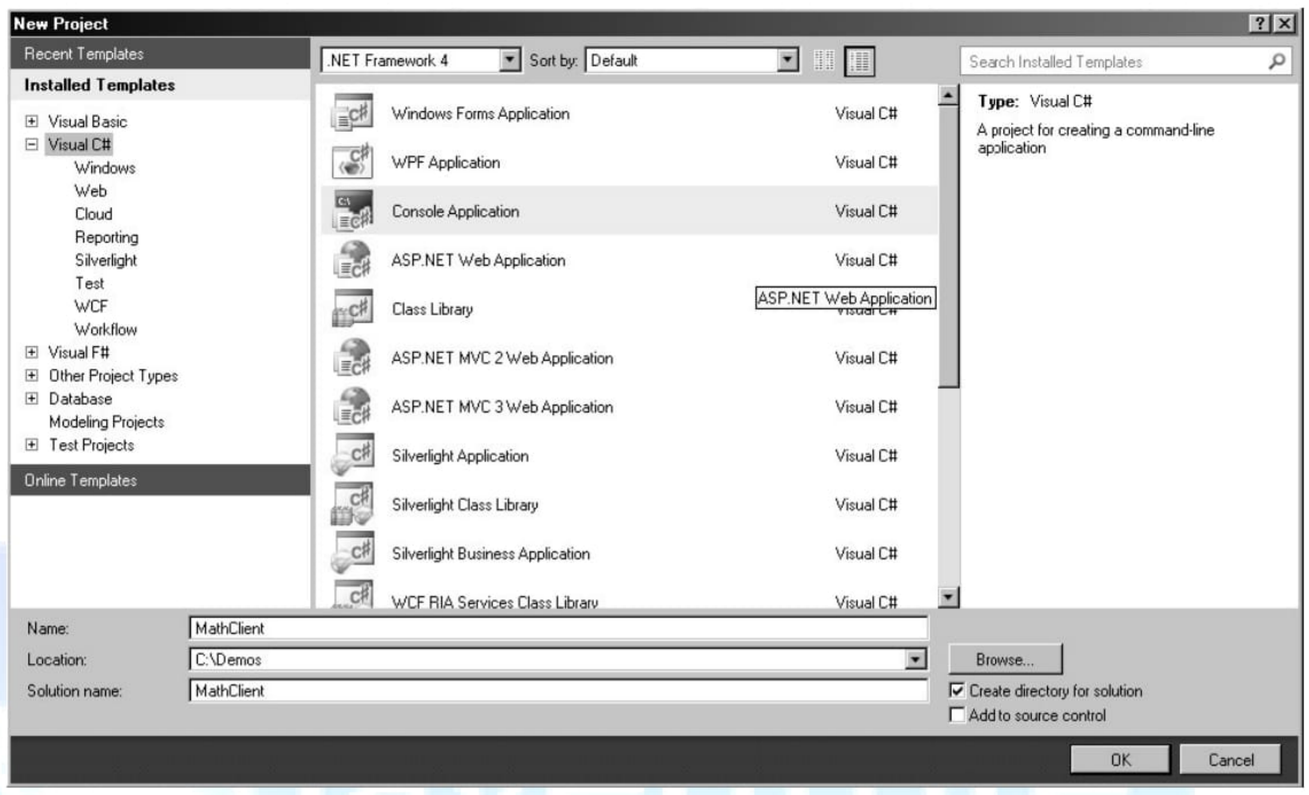


11. See an output assembly (DLL) created called "MathLibrary.DLL". If you double click on the DLL, it will not execute.

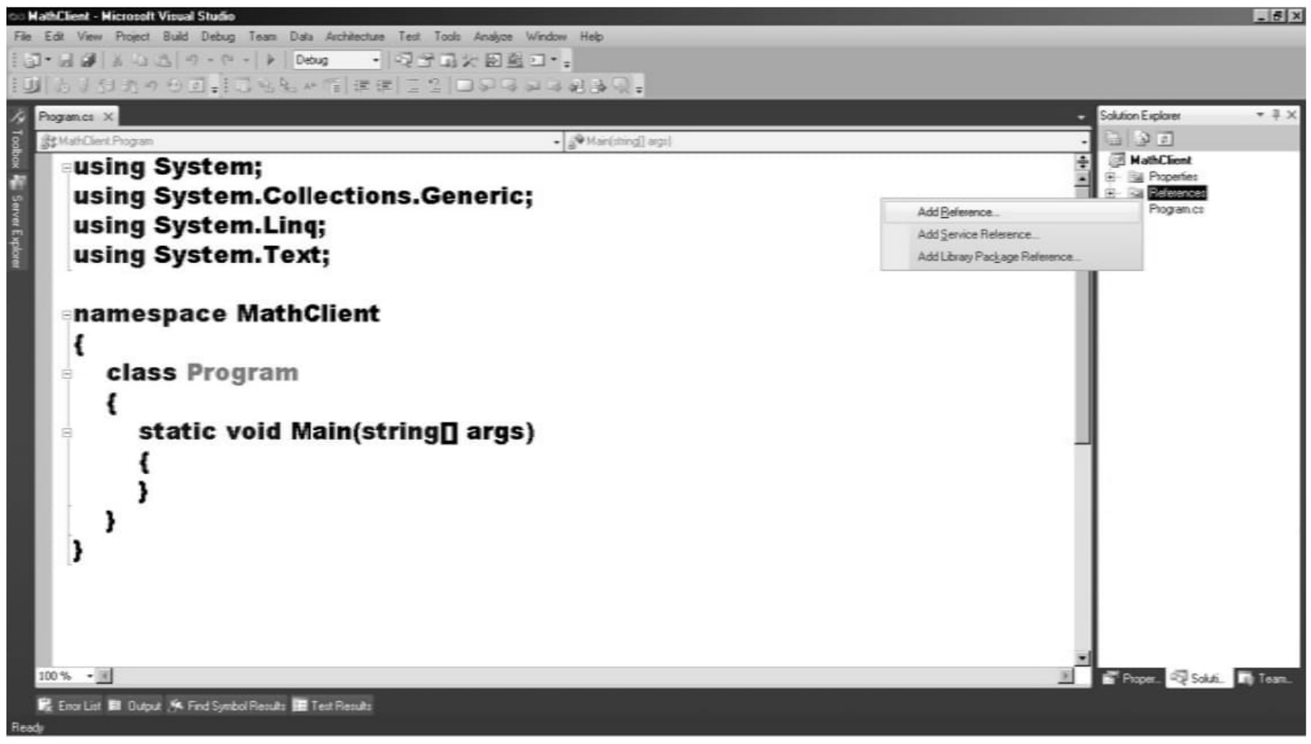
12. This is server (DLL) program. Now let us create a client (EXE) program.

13. See on Next Page:

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

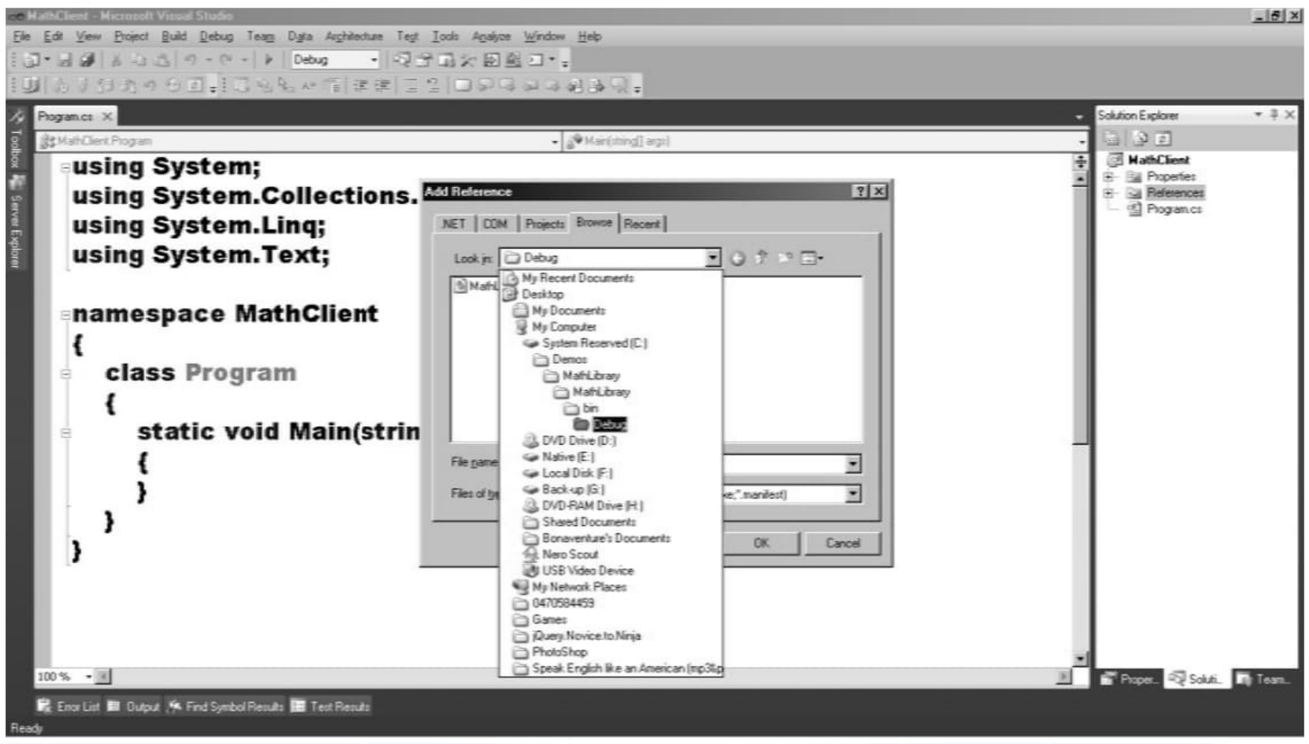


17. Click "Ok"
18. In the solution explorer, right click under references & click "Add Reference":



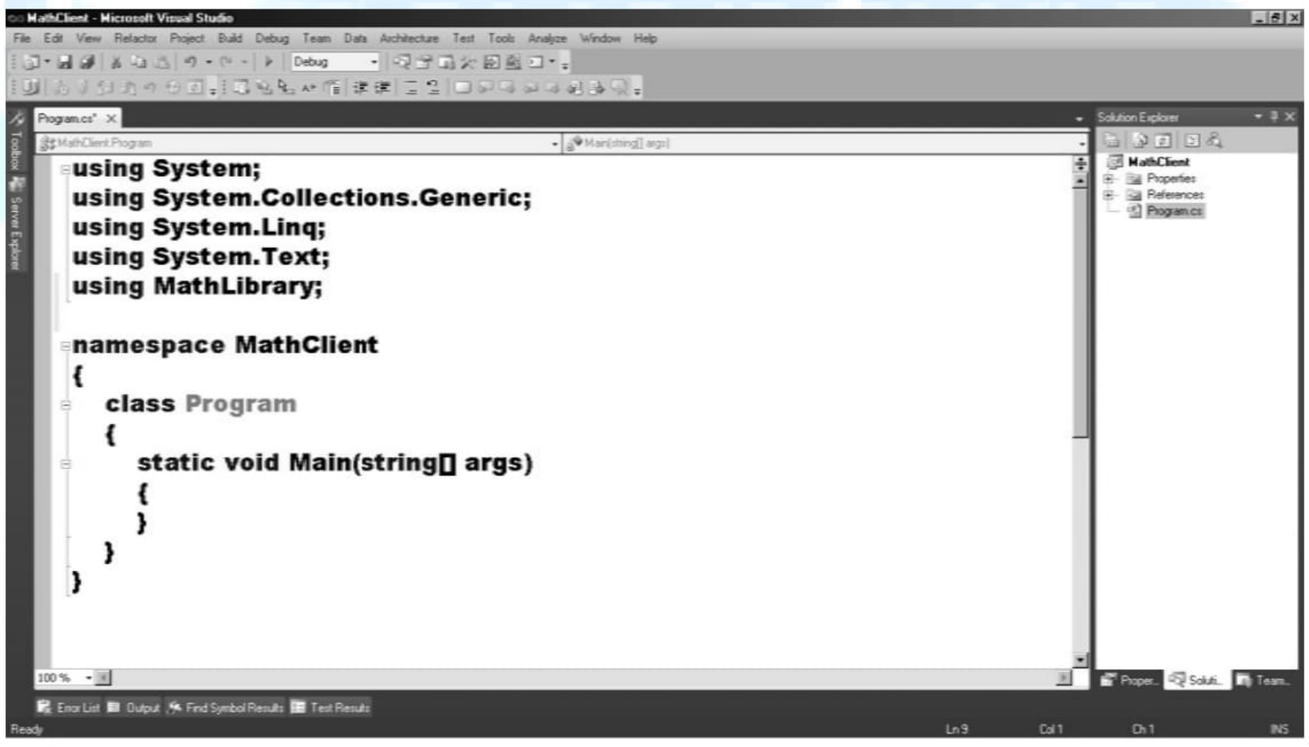
19. In the dialog that opens, click “Browse” Tab:

20. Navigate to previously created DLL.



21. Click “Ok”

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



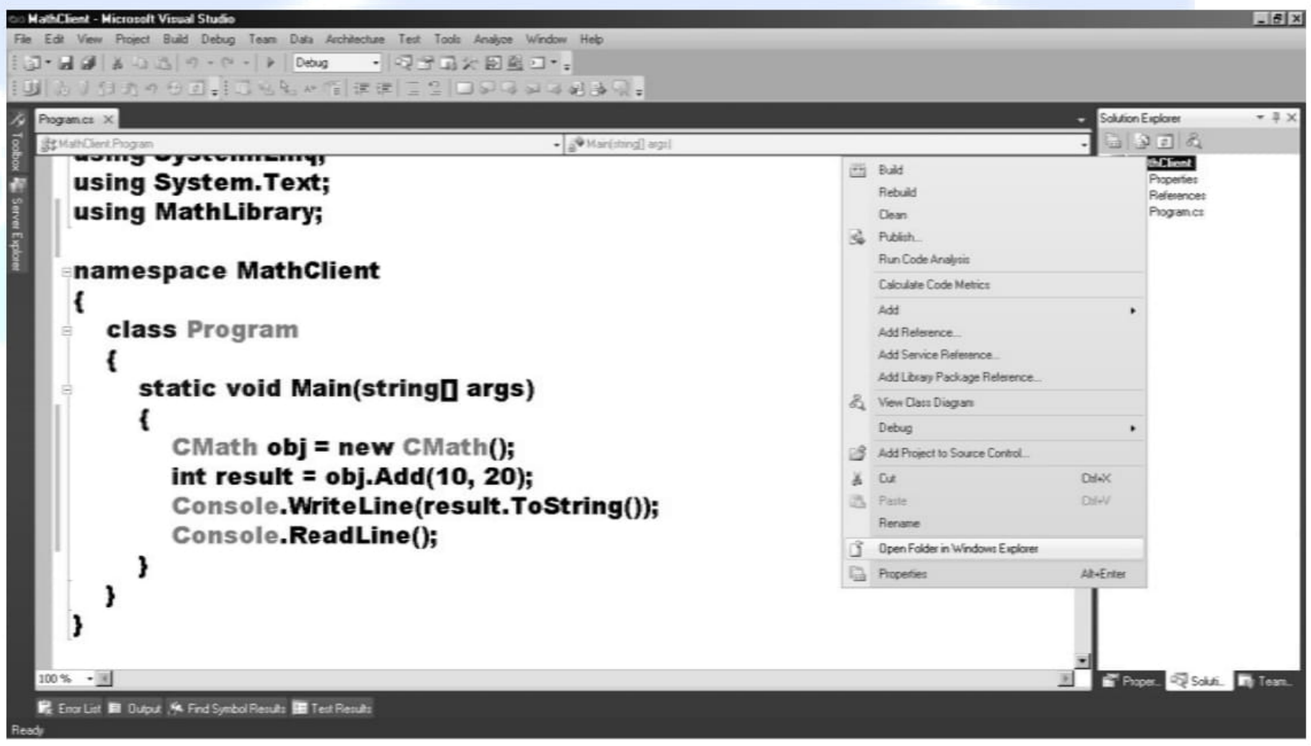
23. Create object of “CMath” Class & call to “Add” function by providing inputs to it.

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

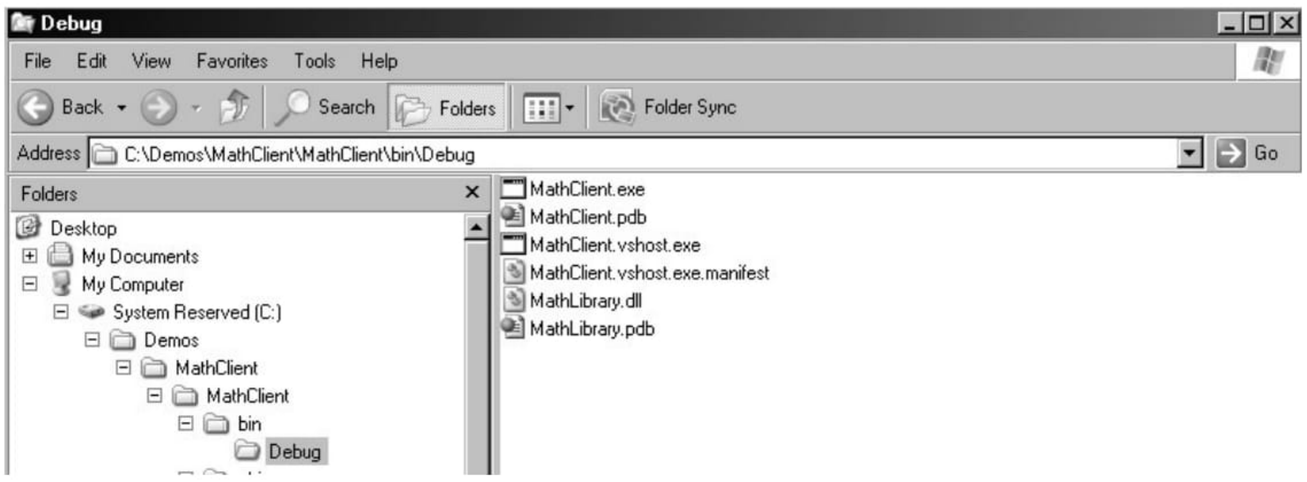
24. Press F5 to run the program. You should see the output as:

25. So, the code is fine. Client calls the server.

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



27. Observe now, what “bin-> debug” folder has:



28. There is a copy of "MathLibrary.DLL" created in the folder.

29. Now what will happen if we make a change in the MathLibrary.DLL? Will the change reflect in client program as well?

30. Answer is **No.**

31. It will not unless & until the Client EXE program is re-compiled with new reference DLL again.

32. This current copy of the DLL is called as "Private copy" or "Private DLL".

33. If we build another Client program referring the same DLL; then that other client will also have similar copy of the DLL inside "bin->debug".

34. If we run both the clients at the same time on same machine then each client will refer to its own copy.

35. If one of the client's private DLL gets corrupt then that will have no impact on the other client running.

36. Which means with private copy helps in running client programs more independently but at the same time, it also created a problem.

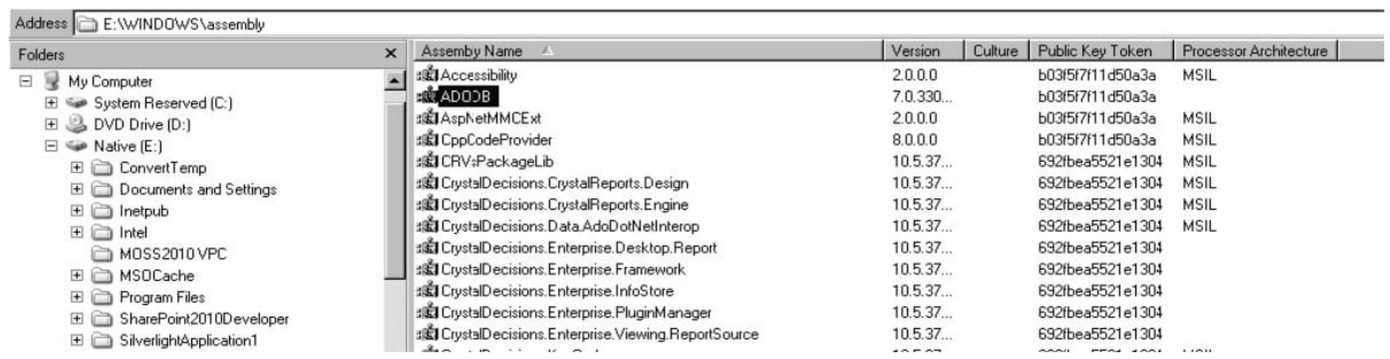
37. These DLLs run inside the memory of client EXEs.

38. The problem is when all clients' needs to be referring new copy or version of the DLL then everyone should be either recompiled or old copy needs to be physically replaced with the new one!

39. Can't there be any location where in if we replace the DLL; every client on the machine will get it from the same location.

40. This is where shared assembly concept comes in picture.

All the shared assemblies are registered with .NET registry (.NET based DLLs are not registered with Windows registry) & can be seen at
[Native Drive]:\\Windows\Assembly folder.



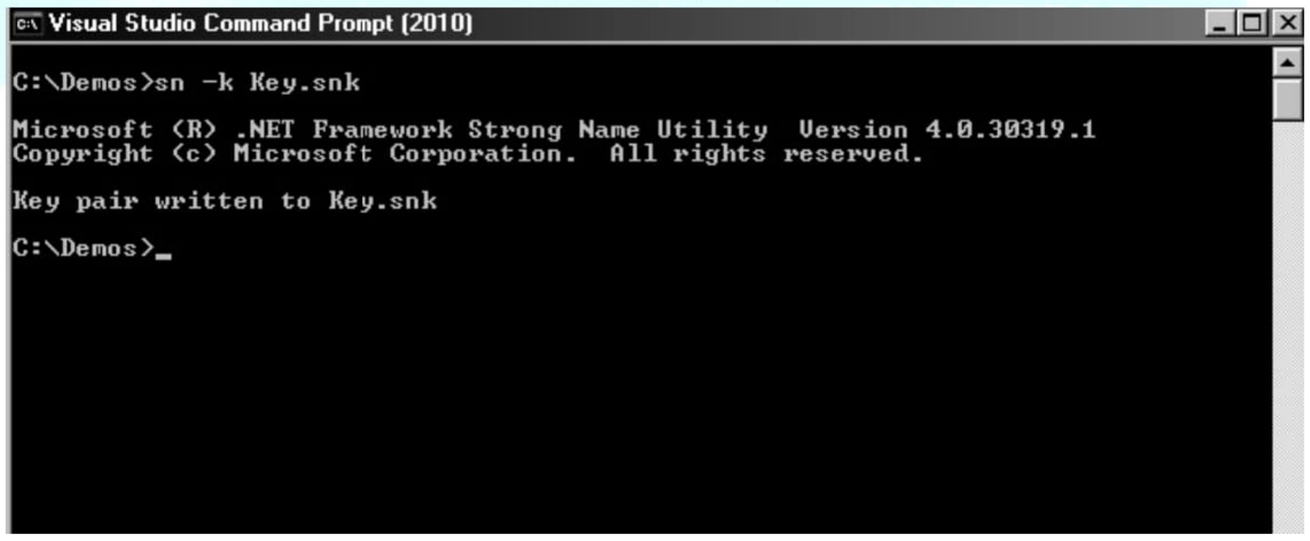
If you see the picture clearly, the existing DLLs have version, culture, and processor architecture & key pair (public, private keys).

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. CLR checks the same using above hash code calculated using strong name key pair. CLR follows integrity check algorithm. To complete the process of making assembly shared; the assembly must be signed with “STRONG NAME KEY PAIR”.

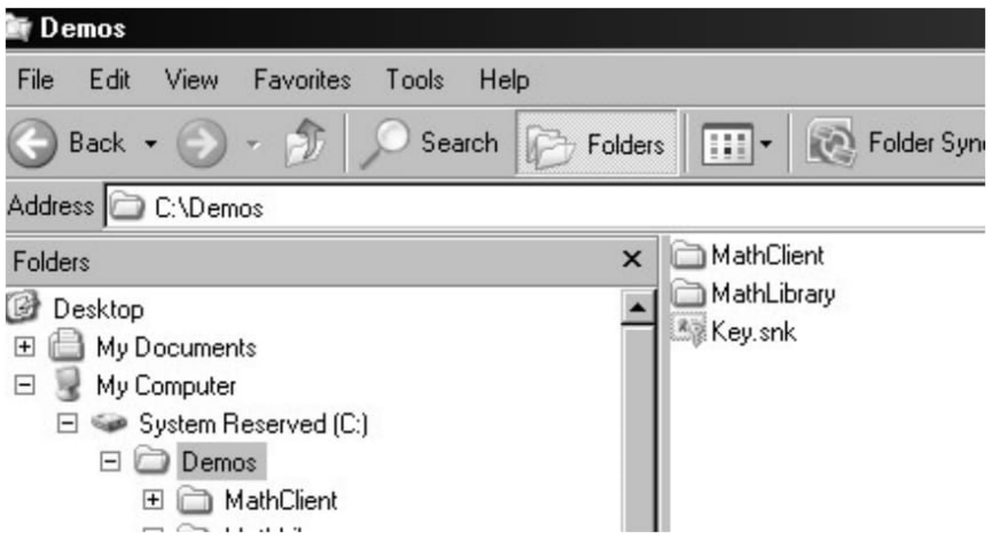
So, to create a shared assembly one should have key pair assigned to the assembly.

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 → Visual Studio Tools → Visual Studio Command Prompt
3. Give a command “sn –k <File name(here it is key.snk)> “



4. On C drive, see the file created.

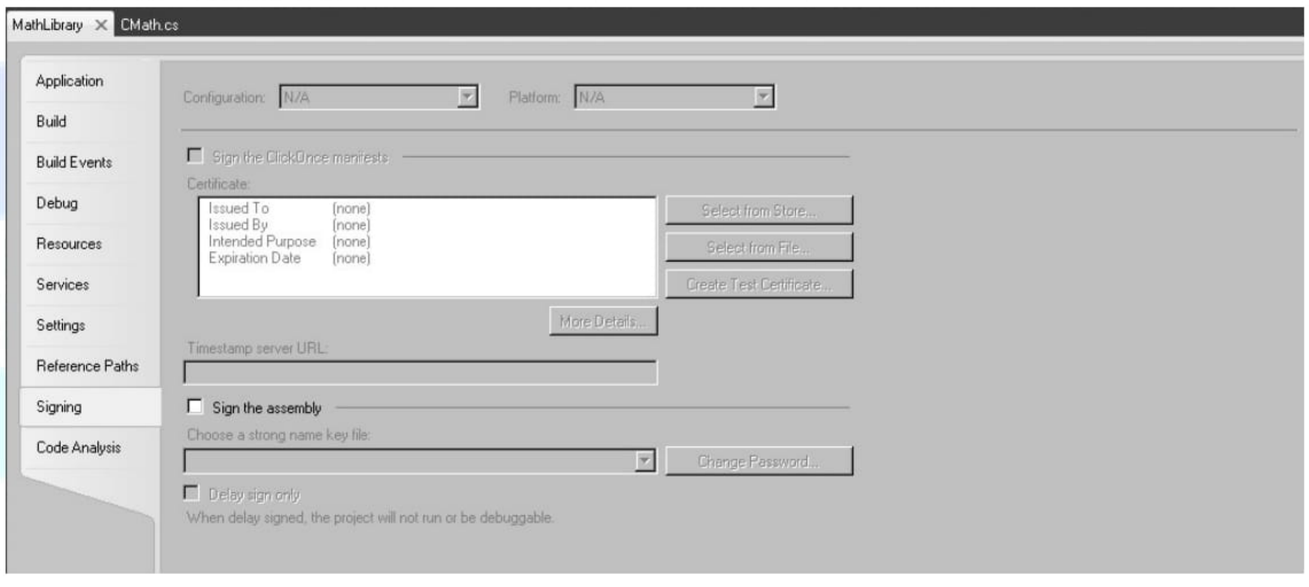


5. Or else there is another way.

6. Right click on the MathLibrary project under solution explorer.

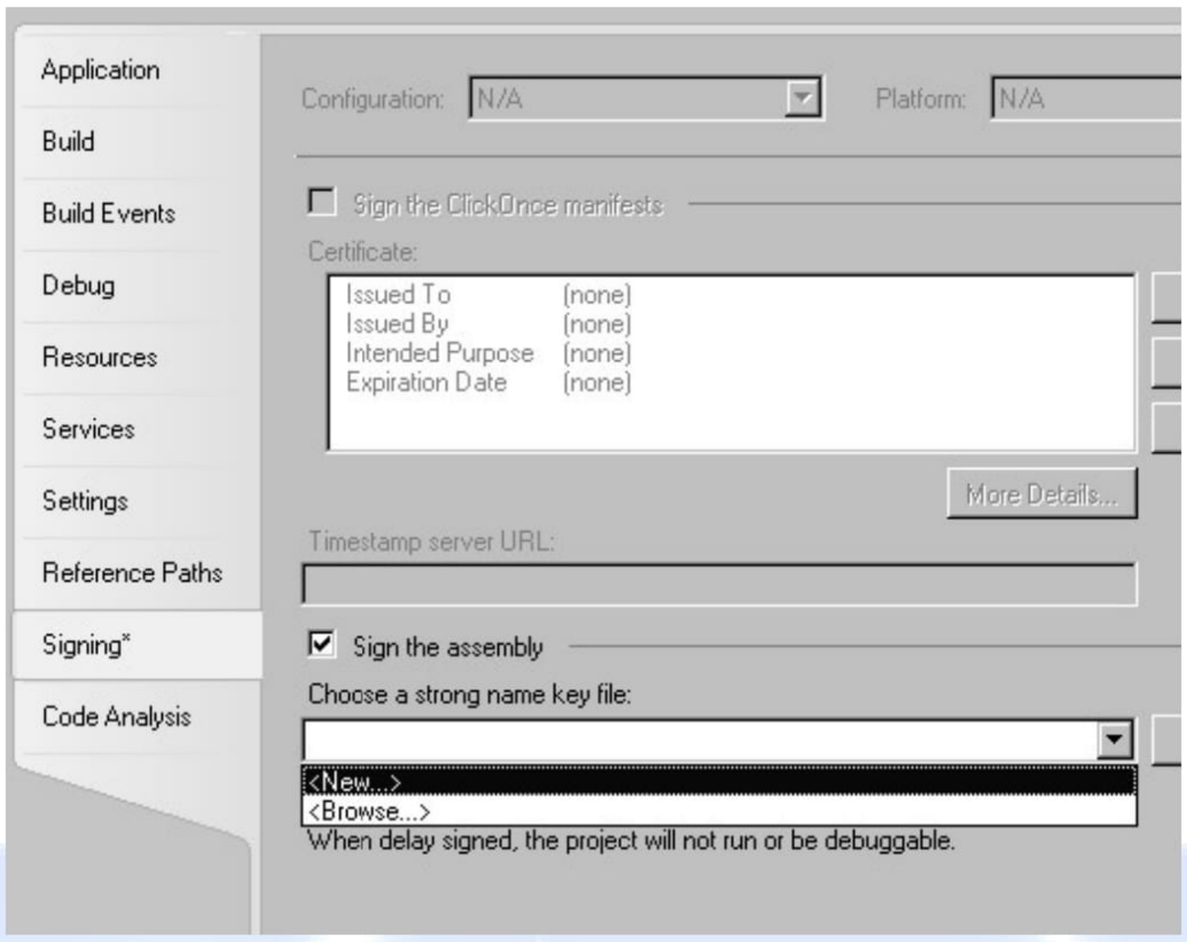
7. Click on Properties menu item.

8. Click on the Signing Tab



9. If you select browse then you can navigate to the file that we created above i.e. 'Key.snk" & compile the project.

10. Otherwise; 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, uncheck the check box & enter name for the file which will have key pair written inside it.

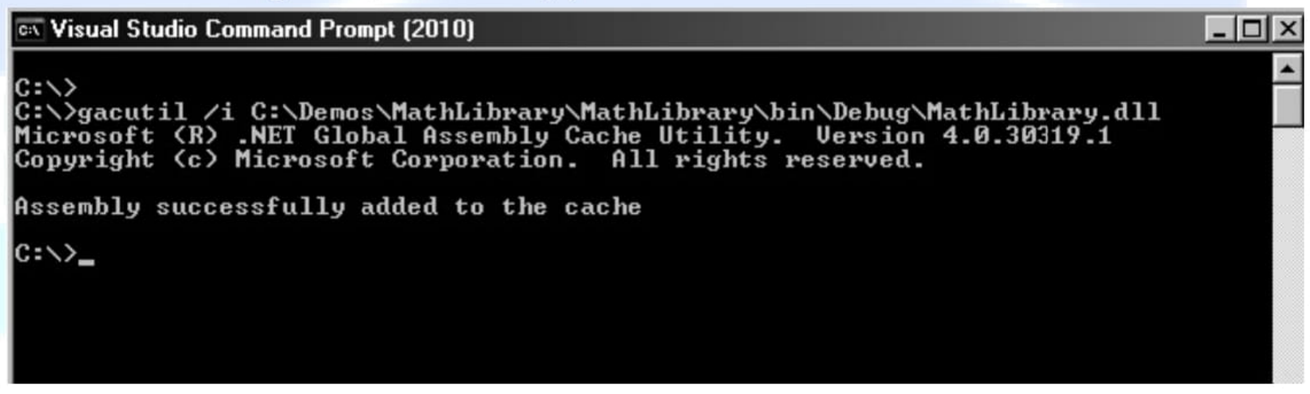
14. Click “Ok” & re-compile the project.

15. You shall 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 → Visual Studio Tools → Visual Studio Command Prompt

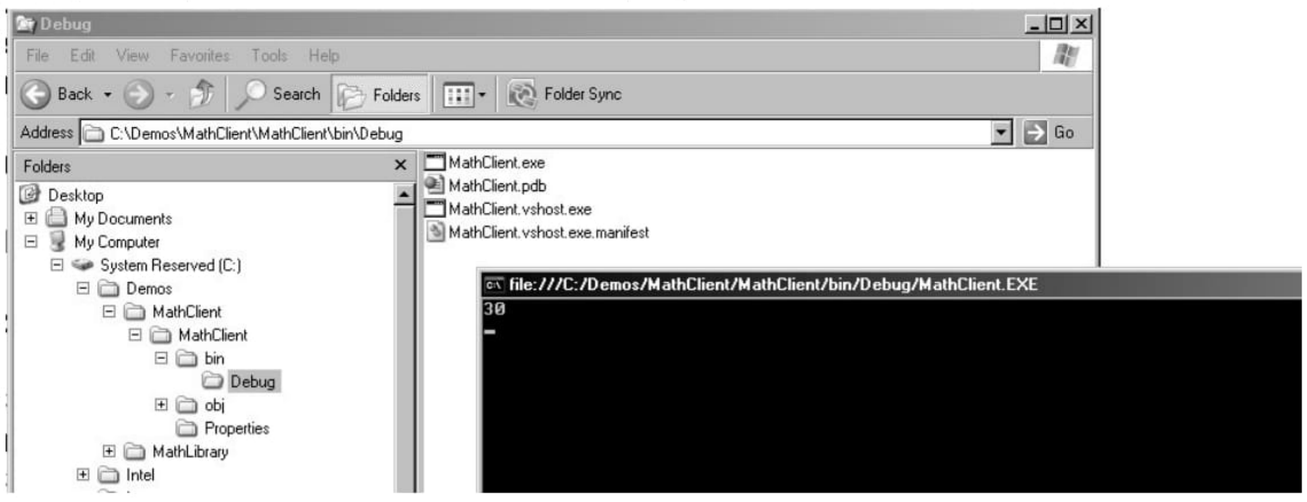
18. Give a command “gacutil /i <assembly path> “. Hit enter.



19. Now open the <Native Drive>:\\Windows\\Assembly Folder

20. You should see the MathLibrary.dll installed. Keep a note year 2010 onwards one may not see the assembly into this folder; however it is there!

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 finds 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, EXE is built with which version of DLL matters. 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.

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 part.

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

Known Issues while practicing:

1. **I can't create strong name key pair using command prompt. It says that I don't have permission. What do I do?**
   Do see that you are working as a "local administrator" while working with Visual Studio Command Prompt or else try to use visual studio wizard by doing a right click on the project under solution explorer & then choosing sign the assembly from vertical tabs.

2. **How do I uninstall the assembly from GAC?**
   Use command ***GACUTIL /u "<Assembly name only - to be uninstalled>"*** on Visual Studio Command Prompt.

3. **Can I copy & paste?**
   No, you won't be able to copy & paste the assembly.

4. **How do we install the assembly on client machine? Since we will not be installing the setup by our own. Is it possible to install the DLL assembly in the GAC while installing the EXE project set up on the client end?**
   Yes, Right click on the EXE project under solution explorer -> click on properties -> under prebuilt command option you need to write the command
   GACUTIL /i "<Assembly to be uninstalled>" which will be executed automatically while client installs the EXE project on the machine. You need to make sure that DLL is also packaged in the setup project which you will hand over to client.

Let us stop the discussion here. More details on CLR can be learned by reading – “CLR via C#” - a book by Jeffrey Richter

Summary:

We have observed in this part:

- What assembly means & its types
- Different types of compilers in .NET
- EX E and DLL differences
- Assembly Execution
- More details on private and shared assembly
- Versioning & probing of assembly

Garbage Collection

CLR is known is the heart of .net framework. We know the following functionalities of CLR:

1. MSIL into native code generation through the JIT compiler.
2. Execution of code
3. Memory management
4. ***Garbage collection***
5. Exception handling
6. Type loader
7. Security check
8. COM marshalling
9. …and many more

In this part, we will discuss following points:

- ✓ Garbage Collection Algorithm (GC Algo).
- ✓ Updates in algorithm in the latest frameworks.
- ✓ Good practices of writing destructors and Finalizers.

Few important concepts:

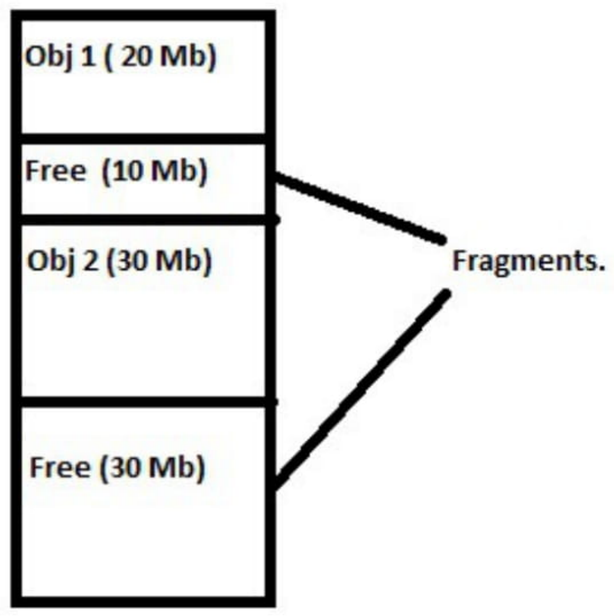
Before we touch base the important concept of GC, let us first discuss few important terminologies:

a. Memory: When it comes to memory, there are two types; in context with the existence. One is the virtual address space, other is the physical memory. Each process will claim its own virtual address space, whereas the physical memory is shared between all the processes.
On a 32 bit computer, a 2 Gb virtual address space is available for each process. As an application developer, in high level programming, we only work with virtual memory and never manipulate the physical memory. GC helps in allocation and freeing of memory.

Virtual memories can be in 3 states:

- Free: There are no references to this block of memory and are free for allocations.
- Reserved: This block of memory is available for use and cannot be used for any other allocation request; but data cannot be stored in it.
- Committed: This block of memory is assigned to physical storage.

b. Fragmentation: Whenever there is a virtual memory request, the Virtual memory manager has to find a single free block to satisfy the request. Even if 1 GB requested memory is available in parts, this request will not be satisfied, since the memory is not continuous. This leads to fragmentation or holes in memory. Hence from time to time these fragments need to be compacted.

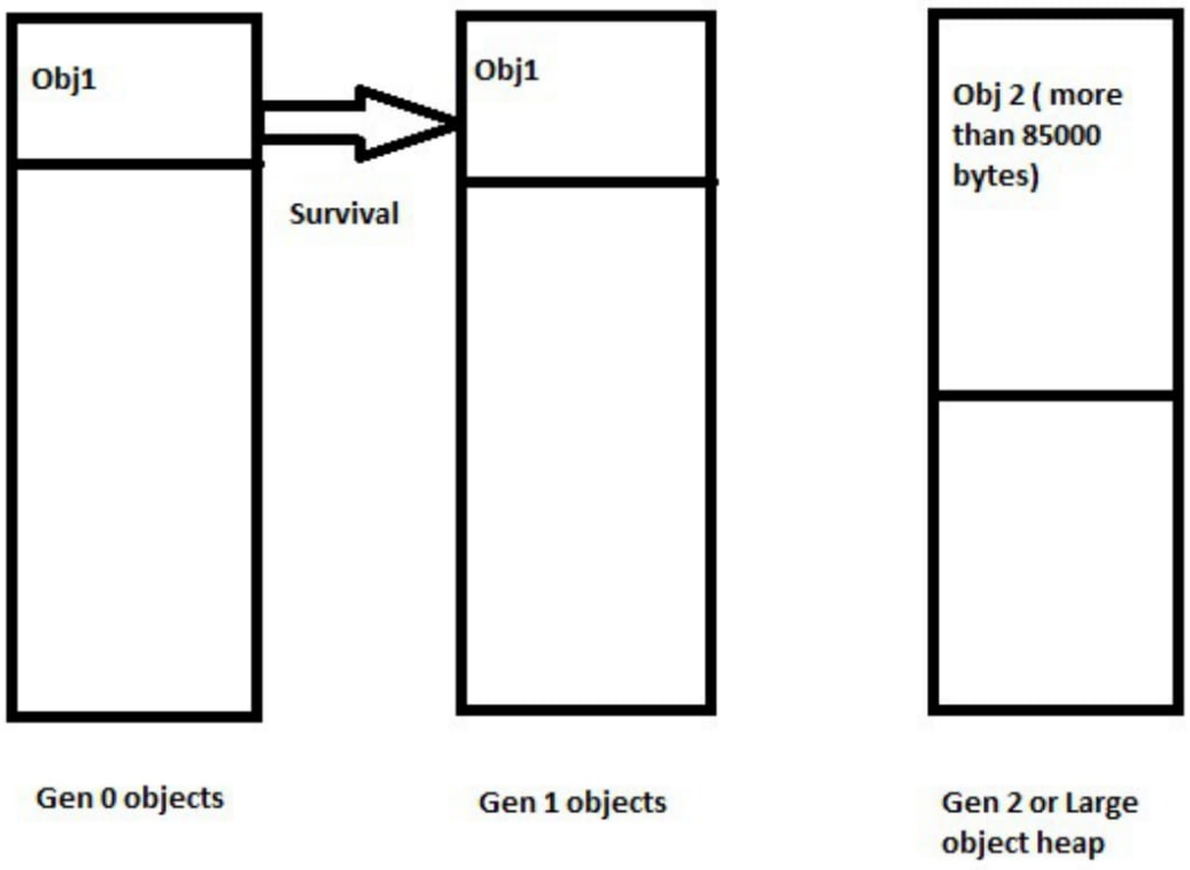


c. Managed heap: The CLR allocates a segment of memory for the program to store and manage objects. This segment is known as “Managed heap”. There is one managed heap for each managed process. All threads in that process will allocate memory for objects on the same heap.
The Win32 *VirtualAlloc* function is used to reserve one segment of memory for managed applications whereas Win32 *VirtualFree* is used to release segments back.
There are two types of heaps: the large object heap and the small object heap. As the name suggests, the large object heap will have very large objects that are 85,000 bytes or more. Whereas the small object heaps will have smaller and temporary variables.

d. Generations: In order to find out the life length of the managed objects, each object falls in a specific generation. Depending upon which generation the object belongs to; it is reclaimed by the GC.
There are 3 generations of objects on the heap:
   - Generation 0: By default, all the newly created objects will fall in Gen 0, unless they are large objects. These are generally short lived temporary variables. The Gen 0 objects face the garbage collection more frequently. Most of the objects are reclaimed in Gen 0; if they survive they are promoted to Gen1.
   - Generation 1: These objects serve as a buffer in between Gen0 and Gen2 objects i.e. the short lived and long lived objects.
   - Generation 2: These objects are long lived objects for e.g. for the entire life of the application.

e. Survivals and promotions: Whenever an object survives a garbage collection, it is known as a survivor and is promoted to next generation. Objects that survive Gen0 garbage collection

are promoted to Gen1 and those which survive Gen1 collection, are promoted to Gen2. Gen2 objects remain in Gen2 itself.



What is Garbage Collector and its algorithm?

Garbage collection is a program which runs automatically to remove unwanted data / objects during the processing. It is initialized by the CLR.
Garbage collector is called when one of the following conditions is true:

1. The system has low physical memory.
2. The memory allocated on the managed heap is getting exhausted and there is more demand of memory.
3. *GC.Collect* method is explicitly called.

The steps carried out when the GC is called are as follows:

1. A marking phase; in which the GC finds and creates a list of all live and dead objects.
2. A relocating phase; in which the addresses of the objects will be updated to go for compaction.
3. A compaction phase; that will reclaim the memory space occupied by the dead objects. The live objects those which have survived the garbage collection are moved towards the older segment of heap.
4. The generations of the survived objects are changed / promoted. Gen2 objects are the long lived objects those who have survived the garbage collection.

5. Normally the large object heaps are not compacted.

GC thus handles the cleaning of unwanted managed resources and frees memory blocks for further memory requirement.
This is how memory is managed for the managed resources. If your program is using unmanaged resources, then there are some good habits which every developer must follow

Using Finalizers and their impact on performance

.Net application developers are at ease and do not have to worry much about cleaning of resources if they are “managed”. But in case the program uses unmanaged resources like “windows handles”, then the developer must clean the resources. GC does not handle the unmanaged resources. The developers can write “Finalizers” to clean up the unmanaged resources. These are similar to C++ destructors (syntactically) but behave rather weirdly at runtime.

```
class Emp
{
    ~Emp()   // destructor
    {
        // cleanup statements...
    }
}
```

Following are the points one has to consider while writing “Finalizers”:

1. The finalizers follow a very non deterministic approach unlike the C++ destructors. The finalizers are highly unpredictable in nature.
2. These methods do not run on your application thread whereas they use the GC’s thread context and may keep the GC engaged for a considerable amount of time.
3. The objects of such classes have a instantiation cost associated with them i.e. they have to be marked separately.
4. When in the code, the developer sets the object of such classes to null, it is expected that when the GC happens, these finalizers will be called and our unmanaged resources will get cleared. But this is not the case. When the GC is called and it traces all the objects, it marks the unwanted managed objects for deletion and moves the objects which have Finalizers implemented in a queue called as “ FReachable Q”.
5. Rest of the objects are reclaimed, promotions and survivals happen and the system moves on.
6. Now the next GC happens after some time and after following the same procedure, the GC now examines the “FReachable Q”. It then starts invoking the finalize methods in no particular order.
7. These methods run on GC thread, which is obviously not a very good idea.

8. Thus if we see, to clean the unmanaged resources, atleast 2 garbage collection cycles were required.
9. Also, we know the fact that if the objects survive the first garbage collection , they are promoted to next generation. This is true for the Fobjects also. Since they do not get cleared in first collection, they get promoted to next generation and so on. This increases the pressure on CLR and causes performance issues.

These were some of the caution points before implementing the finalizers.

We will see one more approach of disposing objects and then we will discuss the best method.

Idisposable Interface

There is an interface available known as “Idisposable” which has a method to implement:”Dispose” method.

In order to clean the unmanaged resources, the Idisposable interface can be explicitly implemented. The Dispose method then has to be explicitly called to clean up the resources. This is a deterministic approach unlike the Finalizers. But there is a dependency on the developer to call the Dispose method. What if the developer forgets to call the Dispose method?

The best approach would be as follows:

1. The developer can write both the Finalizer and also implement the Dispose method.
2. Before calling or setting the object to null, the dispose method can be called to clean up.
3. But if the developer forgets to call the dispose method, then Finalize will be called sometime later; though non deterministic the resources will be cleaned for sure.
4. If the dispose method is called then we do not want the Finalize method to get called again; hence we can call *GC.SuppressFinalize* to ensure that the object is not moved to FReachable Q.

```
public void Dispose()
{
   // Dispose of unmanaged resources.
   Dispose(true);
   // Suppress finalization.
   GC.SuppressFinalize(this);
}
```

This approach ensures that the resources will be cleaned up for sure and can increase the performance.

Latest framework changes:

There are some changes which enhance the performance and can be found from .net framework 4.0 onwards:

1. Gen0 and Gen1 objects are cleaned simultaneously.
2. Notifications can be subscribed for when the Gen2 collections start.
3. Server GC : there is one more background thread created which keeps on cleaning Gen2 objects as required. Due to double GC threads running, the load on application thread minimizes and thus increases the application throughput.

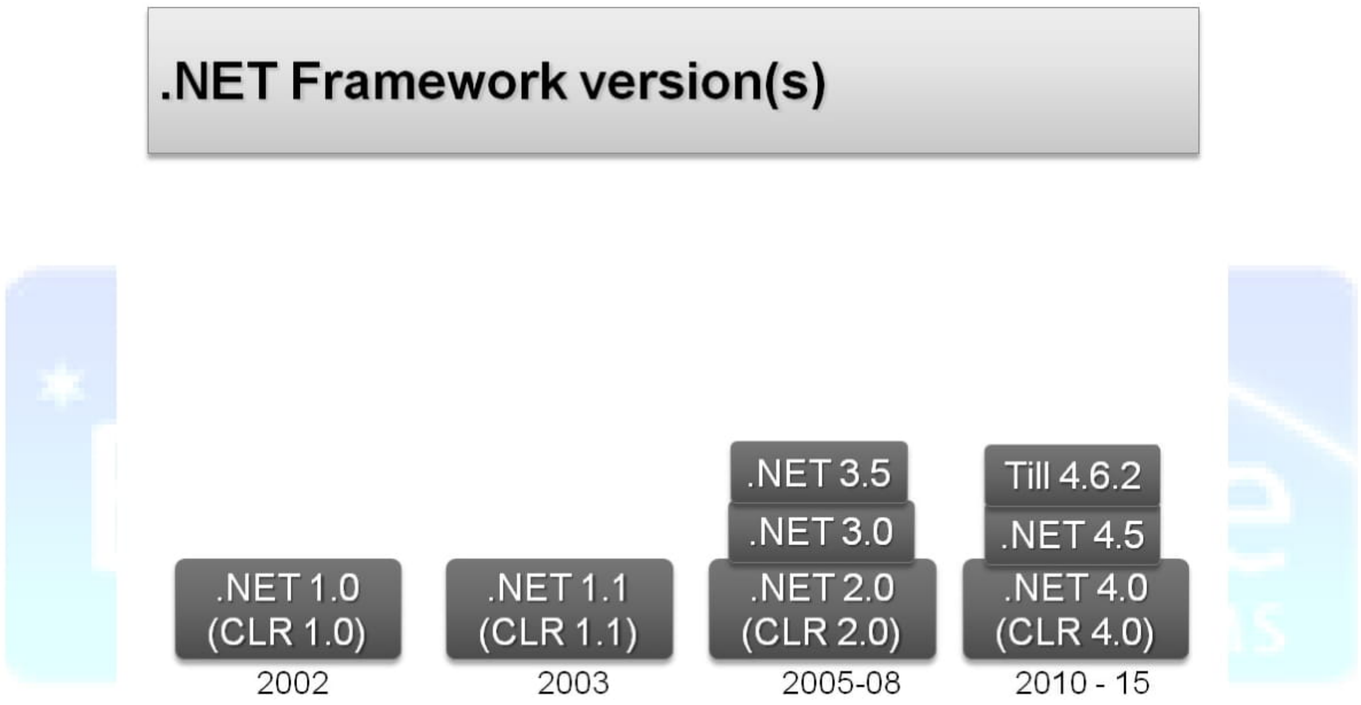
Summary:
In this part on Garbage Collection, we have seen few important terminologies of memory, generations, garbage collection algorithm, best practices of writing finalizers and Idisposable interfaces.
Also, the new features in .net framework 4.0 onwards are highlighted in brief.

Framework Version(s) so far: Horizontal Vs Layer cake architecture

---

We have 12 versions of Microsoft .NET frameworks so far. Out of which 4 can be called as major versions since the base CLR changed in those versions. Rest of the versions can be called as minor versions. In minor versions one can notice launch of new framework libraries but base CLR remains same.

Below snapshot explains those major & minor versions.



Above hierarchy is also known as layer cake architecture. It has horizontally at the base all major versions along with minor ones on top of the base.

As per MSDN documentation few of the important changes with respect to framework versions are:

- .NET 1.0
  - This is the first version of .NET framework
- .NET 1.1
  - ASP.NET and ADO.NET updates
  - Side-by-side execution
- .NET 2.0
  - Generics
  - ASP.NET additions to great extent

- .NET 3.0
  - WPF, WCF, WF, Card Space
- .NET 3.5

    - LINQ
    - Expression Trees
- .NET 4.0
    - Expanded base class libraries
    - Parallel Programming
    - Managed Extensibility Framework (MEF), Code contracts
- .NET 4.5
    - Windows Store apps
    - WPF, WCF, WF, ASP.NET updates
- .NET 4.5.1
    - Support for Windows Phone Store apps
- .NET 4.5.2
    - Profiling improvements
    - ASP.NET updates
- .NET 4.6
    - ASP.NET updates
- .NET 4.6.1
    - Security related APIs
    - Spell checking improvements
- .NET 4.6.2
    - Support for converting Windows Forms and WPF apps to Universal apps.

Every major version of .NET framework was accompanied with launch of Visual Studio. However, only few of the minor version(s) have visual studio launched along with.

We shall discuss, C# features list version wise till 4.0 in the 9th part.

Summary:

We have observed in this part:

- What is layer cake architecture
- Different versions of .NET frameworks
- Noticeable features version wise

Basic Concepts of Programming

While learning any new language, we have to learn the grammar, structure of sentence which are the basics. Similarly while learning C#; we need to learn the basic concepts which are the building blocks of any language so that we can construct the application logic with ease.

In this part, we will be discussing following concepts:

- ✓ Variable declaration
- ✓ Parameters
- ✓ Arrays
- ✓ Loops

Let us commence with variable declarations.

Variable Declarations:

In C#, we follow the same syntax as we do in C++; i.e. the data type is mentioned first then the variable name. These variables can be initialized at the time of declaration or can be accessed later within the scope. The variable names can either begin with a letter or an underscore.

Example 6.1:

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


//Variable declarations

            string firstName = null;
            string lstName = null;
            Console.WriteLine("What is your name? Please enter:");
            firstName = Console.ReadLine();
            lstName = Console.ReadLine();
            Console.WriteLine(" Hi : " + firstName + " " + lstName);
            Console.ReadLine();

            int num=0;
            bool flag=true;
```

In above example 6.1, variable data types like *string, int , bool* , all belong to some or the other base type class like System.String , System. Int32 or System. Boolean respectively.

Following chart shows different data types and their respective ranges and size:

Table 6.1 Integer types:

C# Type	CLR Name	Signed	Size in bits	Inclusive Range
byte	System.Byte	No	8	0-255
sbyte	System.SByte	Yes	8	-128-127
ushort	System.UInt16	No	16	0-65535
short	System.Int16	Yes	16	−32768 - 32767
uint	System.UInt32	No	32	0-4294967295
int	System.Int32	Yes	32	−2147483648 - 2147483647
ulong	System.UInt64	No	64	0 - 18446744073709551615
Long	System.Int64	Yes	64	−9223372036854775808 - 9223372036854775807

Table 6.2 Floating types:

C# Type	CLR Name	Size in bits	Precision	Range
Float	System.Single	32	23 bits (~7 decimal digits)	1.5×10-45 to 3.4×1038
Double	System.Double	64	52 bits (~15 decimal digits)	5.0×10-324 to 1.7×10308

Since we have started the discussion on variable types, let us also include the topic of value types and reference types.

Value types:

A data type is a value type if it holds the data within its own memory location. These include the following:

1. All numeric data types
2. Boolean, Char and Date

3. All structures

Reference types:

A reference type contains a pointer to another memory location that holds the data. They include the following:

1. String
2. All arrays (though their elements are value types)
3. Class types
4. Delegates

Boxing and Unboxing

Boxing is the process in which a **value** type data is converted to type **object** and stored on a managed heap.

For example:

```
int i=123;
object o=(object)i; // boxing
```

Unboxing is the reverse process of boxing. It extracts the value type from the object.

For example:

```
o=123;
i=(int)o; //unboxing
```

Parameters:

Modularity is i.e. writing a function to perform specific tasks is the basis of any OOP language. To communicate with the functions we must pass parameters to them.

In C#, we can pass values / parameters by following ways:

1. Passing By values;
   This is the simplest way of passing parameters, in which only the value is copied in a temporary variable. Operations can be performed on the parameters and then value is returned back if desired. Here no references are maintained.

   For example:

```
class Program
{
    static int add(int a, int b)
    {
```

```
        return a + b;

    }


    static void Main(string[] args)
    {


        int a = 10;
        int b = 10;
        int c = add(a, b);

        Console.WriteLine(" Sum of a & b is : " + c.ToString());
    }
```

}

2. Ref and Out parameters.

When we want to pass the parameters by reference i.e. by actual address we can pass the reference by using *ref* keyword. Here we can actually maintain the reference of the variable we are passing. *Out* keywords are used to return values from the function by maintaining its context /reference. Thus we can return multiple values without specifying the return type of the function. Following example illustrates how we can pass the values by reference and return by using out :

```
static void Addfive(ref int number, out int oparam)
        {
            number = number + 5;
            oparam = number;
        }
        static void Main(string[] args)
        {

            int refparam = 10;
            int outparam;
            Addfive(ref refparam, out outparam);
            Console.WriteLine("Ref parameter: " +
refparam.ToString() + " Out parameter:" + outparam.ToString());
```

```
}
```

In above example , refparam variable is initialized to 10 in the Main function. When we pass this variable by ref, we are actually referring to the address (by using the parameter:"number"). Hence when we add 5 , we are actually adding it to 10 and not 0. Similarly the return type of Addfive function is void, but since oparam is out parameter, we can access this value and address back in Main function.

The statement will thus print the values as 15 and 15 on the console.

Arrays

Arrays are reference type data types, which are very frequently used to store continuous data of any specific type.

Simple, single dimensional arrays are declared as follows:

```
int[] numbers; // simple single dimensional array with no size
specified
numbers = new int[10]; // size allocation of 10 element array
numbers = new int[20];// size allocation of now 20 element array
```

Multidimensional arrays are declared as follows:

```
string[,] names;
names = new string[3, 4];
```

Similarly, we have array of arrays [jagged arrays]:

```
float[][] values = new float[2][];
```

Initializing arrays:

While initializing arrays, we may or may not specify their size.

```
int[] numbers =new int [] {1,2,3,4,5}; // single dimensional array
initialized without size specified
```

Multidimensional arrays can be also initialized in following manner:

```
string[,] names = new string[,] { { "Rahul", "Joshi" }, { "Neha", "Kale" }
};
```

Jagged arrays can be initialized in following manner:

```
int[][] jaggednumbers = new int[2][] { new int[] { 2, 3, 4 }, new int[] {
5, 6, 7, 8, 9 } };
```

Loops:

We have seen some of the basics like data types, variables, arrays. Now it's time to control the flow of the program by embedding logic in it.

There are various looping structures; many of them are already dealt with in languages like C++.

1. If and while statements :
   If loop is generally used to evaluate a particular condition and then execute the dependent statements. While statements will execute all the statements within the loop until the condition specified is true. In following example, the Boolean flag value is initialized to "true". Hence all statements will start executing atleast for the first time. Next , the if loop evaluates whether the input is greater than 10, sets the flag back to true. Else , if the value is between 0 and 10 then flag is set to false, and the while loop is exit.

   Example:

```
int num = 0;
bool flag = true;


while (flag)
{
num = int.Parse(Console.ReadLine());

if (num > 10)
{
    Console.WriteLine("enter between 0 and 10");
    flag = true;

   // Console.ReadLine();
}
else if (num < 10)
{
    Console.WriteLine("Smart enough!!");
    flag = false;
```

```
            Console.ReadLine();
        }

    }
```

2. Do-while loop :
   This is a very common looping structure, in which the statements are executed first then the condition is evaluated. If the condition is false, then the loop is exit.

   Example:

```
//do-while loop
  int i=0;
  do
  {
      Console.WriteLine(i.ToString());
      i = i + 1;

  } while (i < 10);
```

3. For loops:
   These loops have a definite looping pattern sequence. The developer can decide the start and end of the loop as well as the increments and then execute the statements within.

4. For each loop :
   This is a new introduction in C# and different that C++.
   For this concept, we will also learn a new data type / collection known as Arraylist. This is a collection of objects which can dynamically grow and hence has no limit.
   To navigate through this arraylist and other collections, we can make use of for-each loop:

```
ArrayList technologies = new ArrayList();
  technologies.Add("C#");
  technologies.Add("JAVA");
  technologies.Add("Python");
  technologies.Add("JQuery");
  technologies.Add("HRML5");
  foreach (string p in technologies)
  {
      Console.WriteLine(s); // perform operations
      Console.ReadLine();
  }
```

Summary

This part thus deals with very basics, but very important concepts in C#. Variable declaration, parameters, arrays and loops are explained with basic examples.

One can always explore more on these concepts.

Object Oriented Programming (OOP) Concepts

The object oriented languages follow four paradigms:

1. Encapsulation: It means the grouping of related members like properties, methods, events and other members together as a single unit or object.
2. Inheritance: It means the reusability of existing coded functionality. Hence it is the ability to create a new class based on the existing class.
3. Polymorphism: This means many forms. So the multiple classes though having same class members, can exhibit different behavior.
4. Abstraction: Here the term is really abstract. Abstract classes cannot be instantiated. They are generally used at the base of hierarchy for inheritance further. This is a concept or an idea not associated with any particular instance.

C# as well as VB.Net are purely object oriented languages and follow all the principles. We will now one by one explore the basic concepts of OOP:

1. **Encapsulation**: In this principle, we will discuss about class and its members. A class is defined as a unit which encapsulates the related members like data (fields and properties), methods (functions), and events (communication media) for a specific purpose. A class is defined as follows:

```
class Math_operations
 {
     private int variable;
 }
```

An important member of any class is its variables to store data for that particular instance of that class. Unless and until a class is instantiated (not the abstract class), it cannot communicate with our application. We mostly define the variables as private. An instance of class is created as follows:

```
// instantiation of class object, default constructor called.
  Math_operations obj = new Math_operations();
```

Even if the constructor is not defined, the compiler provides default constructor. We can also write our own constructors in following manner:

```
class Math_operations
    {
        private int variable;
        public string msg = null;
        public Math_operations() //default constructor
        {
```

```
            variable = 0;
        }
        public Math_operations(string sentmsg) //overloaded
constructor
        {
            msg = sentmsg;
        }

    }
```

An overloaded constructor is the one with variable number of parameters passed with same name as that of the class.

```
Math_operations objnew = new Math_operations("Hello");
```

We can declare as many overloads as we want.

Properties:

In the above discussion, we have seen variable declaration and constructors. To access the private data variables, we must specify properties. Properties can be of 2 types: getters and setters. They are in short, public methods having same name as the fields and with a specific purpose i.e. either getting or setting value.

Explicit properties can be written specifying the get and set methods and providing access to private variables as follows:

```
        public int Variable
        {
            get
            {
                return variable;
            }
            set
            {
                variable = value;
            }
}
```

The properties can be used in following manner:

```
            Math_operations obj = new Math_operations();

            obj.Variable = 3; // setter is called

            int temp = obj.Variable; // getter is called
```

Auto properties:

This is a feature provided in which we need not declare / define any variable internally for the property. The auto property declared in following manner will have its private variable internally defined by the compiler:

```
public int Auto_Variable { get; set; } // auto properties
```

In above auto property declaration, an integer variable will be automatically created by the compiler.

Methods / Functions:
A method is an action which can be performed by the object. A method can be overloaded or overridden, more of which we will see in further discussions.

Destructors:

We have already seen in previous part of Garbage collection, the use of destructors. Though it is the task of the CLR, we can write our own code to clean up the unmanaged resources.

Events:

Events are also a very important member of any class. They basically help in notifying the class or object that something has happened to take further action. We will discuss more on events in Part 9.

2. **Inheritance:** Let us now move on to the next concept in OOP. Inheritance helps or is meant for reuse of coded functionality. Any class can make use of existing functionality as well as extend / modify the methods provided the base class permits it. Here we must first understand the different access levels supported in C#

C# Modifier	Defination
public	The type or member can be accessed by any other code in the same assembly or another assembly that references it.
private	The type or member can only be accessed by code in the same class.
protected	The type or member can only be accessed by code in the same class or in a derived class.
internal	The type or member can be accessed by any code in the same assembly, but not from another assembly.
protected internal	The type or member can be accessed by any code in the same assembly, or by any derived class in another assembly.

The class whose members are inherited is called as base class and the class which inherits from a base class is called as derived class.

The basic syntax of inheritance is as follows:

```
class base_class
{
}

class derived_class : base_class
{
}
```

In above declaration, all the members; except the private members of the base class will be inherited in the derived class and by creating instance of the derived class , we can access them.

By default all classes are inheritable, unless they are specified as *sealed*. If a class is specified as sealed, then it is not allowed to inherit it. This class can be instantiated though. Generally this class can mark the end of the hierarchy of classes.

```
sealed class sealed_class
 {
 }
```

Instance of a sealed class can be created in same manner as we do for normal classes;

```
sealed_class obj_sealed = new sealed_class();
```

Abstract classes: As against the sealed class, the abstract classes are meant for inheritance only. They cannot be instantiated. They are generally the base of the hierarchy of classes.

As seen in below example, the *person class* is defined as *abstract class*. It has an abstract method defined in it : *wagecalculation* . This abstract class will not have any implementation. It has to be implemented by the derived class. The wagecalculation can change if the type of person is employee or owner.
Lets have 2 more classes : Employee and Owner which both derive from person class and implement their own logic in wagecalculation.

```
abstract class Person
    {
        public abstract double wagecalculation();
```

```
    }


class Emp:Person
    {
        private int hrs;
        private double rate;
        private int _Id;
        public int Id
        {
            get
            {

                return _Id;
            }
            set
            {
                _Id = value;

            }

        }

        public override double wagecalculation()
        {
            return (hrs * rate);

        }
}
```

In above example, you can see a keyword: *override*. This was used to give a meaning or some implementation to the abstract method. Similarly, for normal classes, the developers can give the provision of overriding any normal implemented method in base class to be overridden in derived class, by specifying the keyword *virtual.*
In below example, in Emp class, there are 2 methods: *CheckEmp* and *DelEmp* which both will have some logic. There is one more class ITEmployee which is more specific to IT field and their business logic may or may not be different from base class Emp. So if the developer gives the provision to override the 2 methods, by specifying virtual keyword, then the two methods can be overridden in derived class.

```
class Emp:Person
    {

        private int hrs;
        private double rate;
        private int _Id;

        protected virtual void checkEmp()
```

```
        {
            // logic to check employee status
        }
        public virtual void DelEmp()
        {
            // logic to delete the employee
        }
    }

class ITEmployee : Emp
    {
        protected override void checkEmp()
        {
            base.checkEmp();
        }
        public override void DelEmp()
        {
            base.DelEmp();
        }
    }
```

Thus the base class functionality can be resused, or changed completely. This is termed as dynamic polymorphism. We have one more terminology in .net , namely static polymorphism:

3. Static Polymorphism

   Let us now discuss another interesting concept in polymorphism. We have learnt the term overloading in C++. Similarly we have function overloading in C#, where at compile time itself we specify the different signatures we are allowing for a particular functionality.

   This will be clearer from below example:

```
class base_class
    {
        public int Addint(int a, int b, int c)
        {
            return a + b + c;

        }

        public float Addfloat(float a, float b, float c, float d)
        {
            return a + b + c + d;

        }
    }
```

While calling the functions, we can give calls to specific functions in following way:

```
namespace OOPDemo
{
    class Program
    {
        static void Main(string[] args)
        {


            base_class obj_b = new base_class();
            obj_b.
                  Addfloat
            sealed    Addint      = new sealed_class();
                  Equals
            // ins GetHashCode    s object, default constructor called.
            Math_o GetType       w Math_operations();
                  ToString
            obj.Variable = 5; // setter is called
            int temp = obj.Variable; // getter is called


            Math_operations objnew = new Math_operations("Hello");
```

Interfaces:

In C++, we have seen that we can have both the types of inheritances viz. multiple and multilevel, whereas in C# we cannot implement multiple inheritance i.e. we cannot have 2 parent classes inherited in a single base class. This can be achieved by using and implementing interfaces.
Interfaces, like classes, define a set of properties, methods, and events. But unlike classes, interfaces do not provide implementation. They are implemented by classes, and defined as separate entities from classes. An interface represents a contract, in that a class that implements an interface must implement every aspect of that interface exactly as it is defined.

For e.g.:

```
interface sampleI1
{
    // just method declaration
    int sampleMethod(int i);
    void setData();
}

class implementor:sampleI1
{
    public int sampleMethod(int i)
    {
```

```
            throw new NotImplementedException();
        }

        public void setData()
        {
            throw new NotImplementedException();
        }
}
```

Multiple interfaces can be implemented in a single class.

Summary:

In this part we have seen all the principles of object oriented programming. We will explore more on the principles of OOD in the part of “SOLID Principles”. Each C# developer thus must be very thorough about the OOP concept before designing any class, module, functions etc.

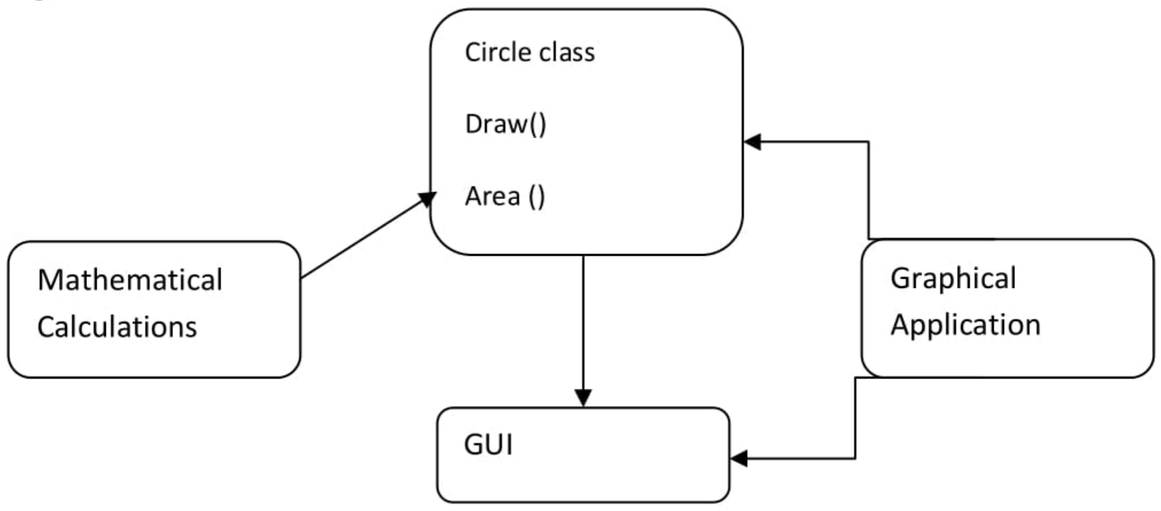
S.O.L.I.D. Principles with .NET

In the object oriented programming part, we have already seen the 4 basic principles of OOPS. Any object oriented programming language, must follow these and the object oriented design pattern. There are 5 principles that expose the dependency management aspects. What is dependency management? The “class” is necessarily the basis of OOP. Hence the classes that we write must be such designed that they are least dependent on any other code / classes. If there is a small change in the class, it should be limited to itself and should not affect the purpose or functionality of other classes which may or may not be related to them. If the dependencies are well managed, the code remains flexible, robust and reusable.

The class design is thus very important. The first five principles are the principles of class design, which every developer must desire and try to follow.
These are known as the SOLID principles:

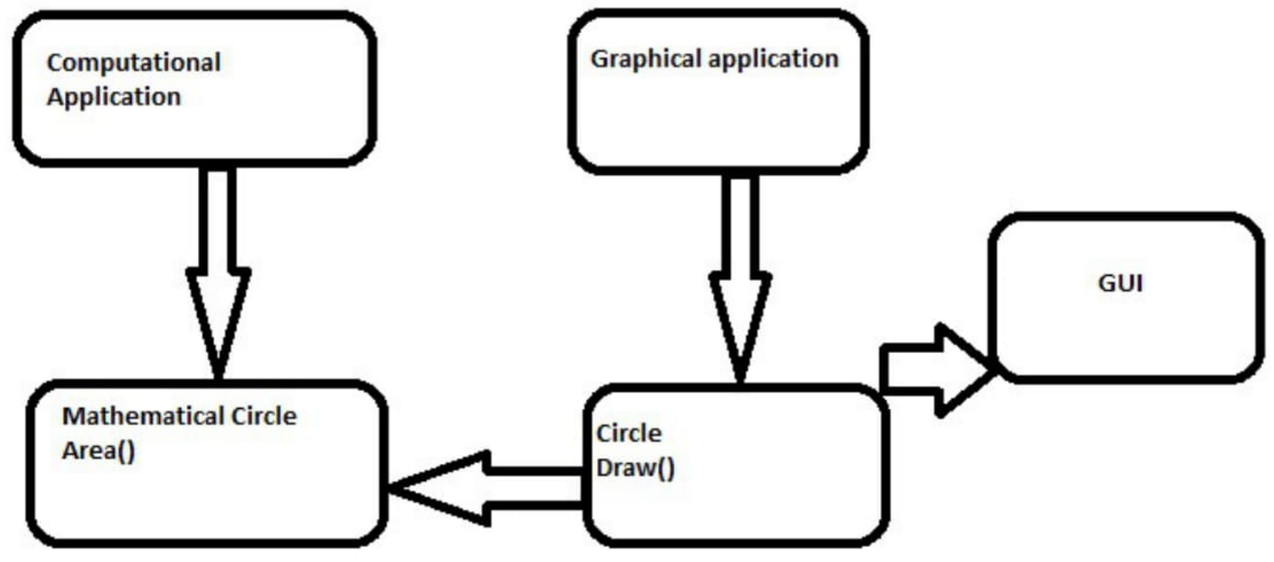
1. **SRP: Single Responsibility Principle :**
   Consider a class “Circle” which has two functionalities: a.) area() b.) draw () . This class is accessed by two different applications. One application is responsible for drawing the circle and hence is related to GUI. On the other hand, second application calculates the area of the circle by using mathematical operations. Thus you can understand that here the “Circle” class has 2 responsibilities: one is to calculate the area and other is to draw the circle. In this example , if the area calculation changes , then the class has to be rebuild and redeployed, even if the “draw” functionality does not change. Responsibility is thus defined as the reason to change. If as a developer if you can think of more than one motive to change, then the class has more than one responsibility. This is thus a bad design.

   Diagram:



A better design would be thus to separate the two responsibilities of the “Circle” class.

Diagram:



We can similarly consider one more example of “Employee “class. Here, let us consider there are two responsibilities: CalculateSalary() and StoreEmp(). Here it is very obvious that the Calculate salary functionality is very dynamic and will change very frequently. Whereas the store employee functionality will remain persistent. Binding the business rules with persistence may create problems in long run. Thus it is advisable to separate these two functionalities or responsibilities.

Thus SRP is one of the simplest principles to understand but hard to implement. As a developer, we can always find out ways to club together the responsibilities, but the art of s/w design lies in the separation of responsibilities.

2. **OCP: Open Closed Principle:**

   This principle is related to the very important principle of abstraction.
   It states that “Software entities like classes, modules, functions etc should be open for extension but closed for modification”. While we are designing any system, one thing must be very clear in mind that it is bound to change in due course of time. The s/w developers and designers must accept this fact. Thus, when a module changes in due course of time, and hence the subsequent or dependent code also changes, then this is a bad design. The program will then become fragile, rigid, unpredictable and also non reusable. A good design is to code modules that would never change. If the requirements change, then add new code to extend the behavior, but never modify the same.
   The modules that follow this OCP have following attributes:

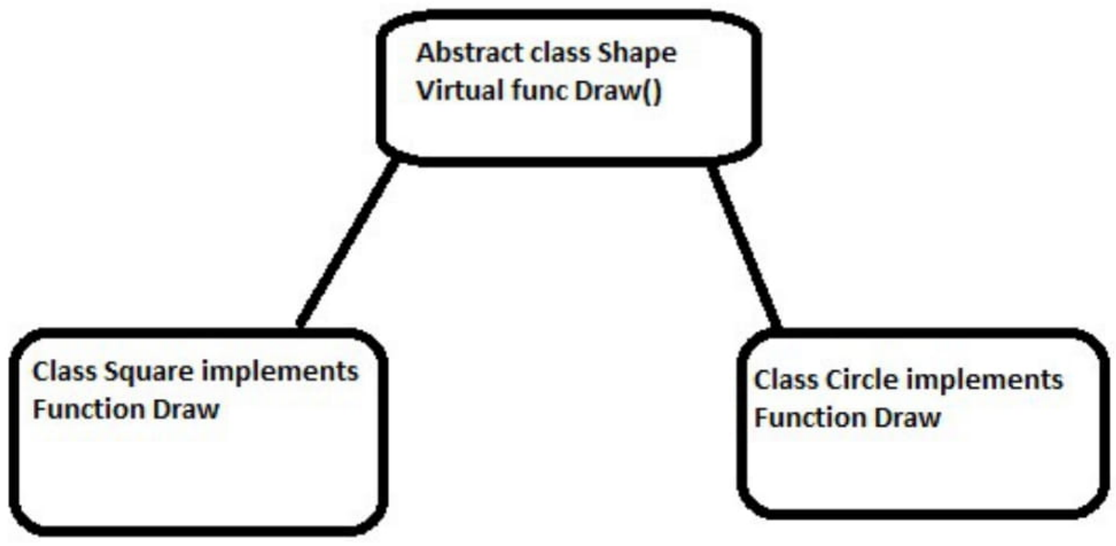
1. Open for extension: In order to cater the new requirements or changes, the modules are open for extending the behavior. So the module can behave in a new and different way.
2. Closed for modification: The source code of such modules cannot be violated that is cannot be changed.

Consider an example here: we have a class to draw shapes : circle and square , in which we have a function :" DrawAllshapes" . The module has a switch statement for identifying which shape to draw. This module does not follow the OCP. Whenever there is a requirement of drawing new shape, then the code of this module has to be changed. It cannot be extended. To follow OCP, we can consider a new design. We design an abstract class : Shape and a virtual function Draw in it. Consider two more classes :"Square" and "Circle" which derive from the base class . Now the "DrawAllshapes" module will remain the same.

Diagram: Following code snippet does not follow the OCP.

```
void DrawAllShapes(ShapePointer list[], int n)
{
int i;
for (i=0; i<n; i++)
{
struct Shape* s = list[i];
switch (s->itsType)
{
case square:
DrawSquare((struct Square*)s);
break;
case circle:
DrawCircle((struct Circle*)s);
break;
}
}
```

Following modification follows OCP:



3. **LSP: The Liskow Substitution Principle**

The principle says that “Derived classes must be substitutable for their base class”. Thus we should be able to use any derived class instead of a parent class and have it behave in the same manner without any modification. This principle is just an extension to the OCP i.e. Open closed principle. We must thus ensure that the derived class must extend the base class without changing the behavior. A real life example would be: A father who is doctor; has a son who wants to be a cricketer and hence the son can’t replace his father even though both belong to same family.

Consider below example of 2 classes : Class “Rectangle” and class “Square”. Let us say Rectangle has two virtual properties:

```
public class Rectangle
  {
      public virtual int Height { get; set; }
      public virtual int Width { get; set; }
  }
```

The class Square inherits from class Rectangle in following manner:

```
public class Square : Rectangle
  {
      private int _height;
      private int _width;
      public override int Height
```

```
{
    get
    {
        return _height;
    }
    set
    {
        _height = value;
        _width = value;
    }
}
public override int Width
{
    get
    {
        return _width;
    }
    set
    {
        _width = value;
        _height = value;
    }
}

}
```

In order to calculate area of both the shapes we write a class: AreaCalculator:

```
public class AreaCalculator
        {
            public static int CalculateArea(Rectangle r)
            {
                return r.Height * r.Width;
            }

            public static int CalculateArea(Square s)
            {
                return s.Height * s.Height;
            }
        }
```

We can test above example by creating 2 different objects of Rectangle and Square and then call the AreaCalculator class. Let us now try to substitute the object of Rectangle (base class) with object of Square (derived class). Here the calculation will go for a toss.

```
public void RectanglefromSquare()
        {
            Rectangle newRectangle = new Square();
            newRectangle.Height = 4;
            newRectangle.Width = 6;
            var result = AreaCalculator.CalculateArea(newRectangle);
```

```
        }
```

The expected result is 24, whereas we will get the answer as 36. Thus the LSP is not followed.

To solve this problem and follow LSP, we get rid of AreaCalculator class.

```
public abstract class Shape
        {
 public abstract int Area();

        }
```

Rectangle class will be written as follows:

```
public class Rectangle :Shape
  {
      public  int Height { get; set; }
      public  int Width { get; set; }
      public override int Area()
      {
          return Height * Width;
      }
  }
```

Square class will be written as follows:

```
public class Square : Shape
  {
      public int Sides;
      public override int Area()
      {
          return Sides * Sides;
      }

  }
```

Thus above example thus follows the OCP as well as LSP.

4. ISP: The Interface Segregation Principle

This principle states that “Clients should not be forced to implement interfaces they don’t use.”
Instead of having a big fat interface, it is always beneficial to have small interfaces.
We can define it in another way. An interface should be more closly related to the code that uses it than code that implements it. So the methods on the interface are defined by which methods the client code needs than which methods the class implements. So clients should not be forced to depend upon interfaces that they don't use.

We have seen the principle of SRP. Same applies to interfaces. The developer must not be forced to implement the interface if the object doesn't share the purpose.
Let us consider one example which violates the ISP.

Consider a IT firm in which we have an interface: Ilead

```
public Interface ILead
{
   void CreateSubTask();
   void AssginTask();
   void WorkOnTask();
}
```

There is one more class Teamlead which implements this interface and inturn its methods:

```
public class TeamLead : ILead
{
  public void AssignTask()
  {
    //Code to assign a task.
  }
  public void CreateSubTask()
  {
    //Code to create a sub task
  }
  public void WorkOnTask()
  {
    //Code to implement perform assigned task.
  }
}
```

There is one more entity now : Manager , who also implements Ilead , but he never works on a task:

```
public class Manager: ILead
{
  public void AssignTask()
  {
    //Code to assign a task.
  }
  public void CreateSubTask()
  {
    //Code to create a sub task.
  }
  public void WorkOnTask()
  {
    throw new Exception("Manager can't work on Task");
  }
```

```
}
```

This violates the ISP. Let us separate the responsibilities by adding one more interface:

```
public interface IProgrammer
{
  void WorkOnTask();
}
```

An interface that provide contracts to manage the tasks:

```
public interface ILead
{
  void AssignTask();
  void CreateSubTask();
}
```

Now we implement the following classes as:

```
public class Programmer: IProgrammer
{
  public void WorkOnTask()
  {
    //code to implement to work on the Task.
  }
}
public class Manager: ILead
{
  public void AssignTask()
  {
    //Code to assign a Task
  }
  public void CreateSubTask()
  {
  //Code to create a sub taks from a task.
  }
}
```

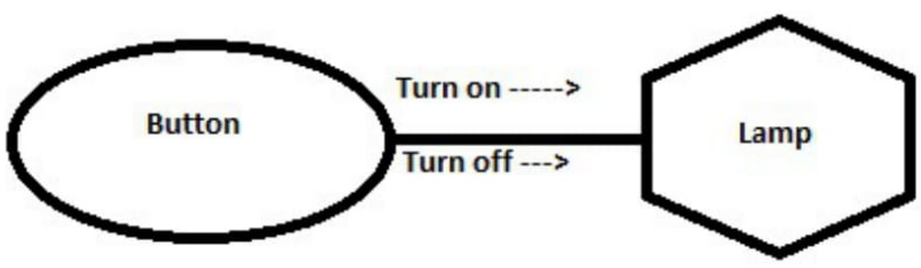
TeamLead can manage tasks and can work on them if needed. Then the TeamLead class should implement both of the IProgrammer and ILead interfaces.

```
public class TeamLead: IProgrammer, ILead
{
  public void AssignTask()
  {
    //Code to assign a Task
  }
  public void CreateSubTask()
  {
```

```
    //Code to create a sub task from a task.
  }
  public void WorkOnTask()
  {
    //code to implement to work on the Task.
  }
}
```

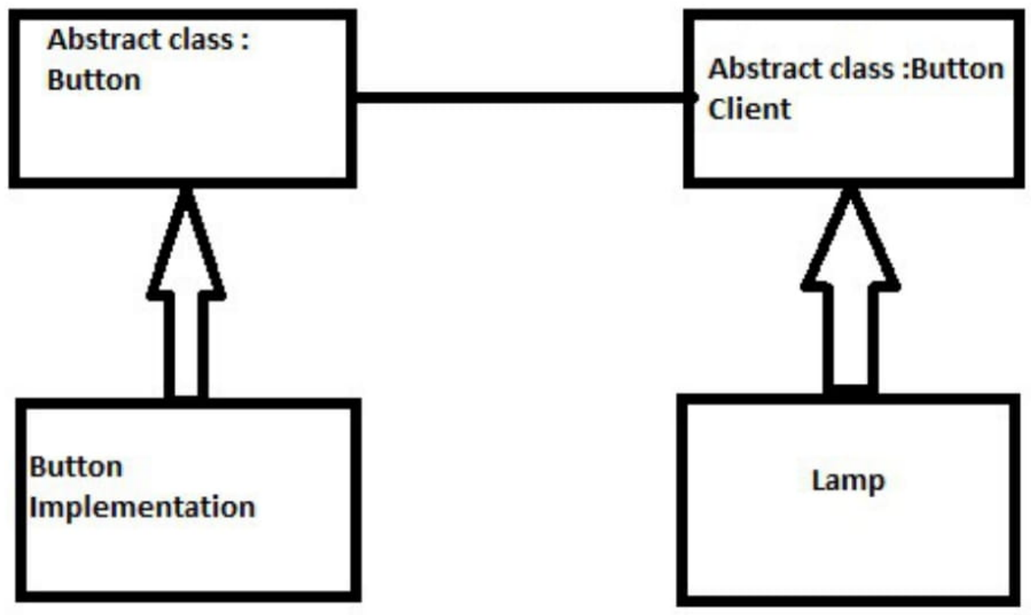
5. DIP: Dependency Inversion Principle:
   This dependency inversion principle states that high level modules/classes should not depend upon low level modules/ classes. Both may depend upon abstractions. Secondly , abstractions should not depend upon details, whereas details should depend upon abstractions.

   We will see a real life example to explain this principle. We can apply the DIP wherever a class sends message to another class. For this , let us consider a button object and a lamp object. A button object is dependent upon the external triggers of on and off. This button could be a UI button class or a physical button which is operated by human fingers. Its basic nature is to switch on and off and pass on this message to the lamp object.
   The lamp object will receive the messages from the button object and will either illuminate the light of some kind or switch it off. The physical action here is not so important.
   If we design such kind of system that Button object controls Lamp object, how will it look like?
   We can simply represent a “contains” relationship. The Button class can contain the Lamp instance:



   Now consider that , we change the lamp class , then the button class must also be recompiled in that case. Moreover if we want to control any other equipment other than lamp, then in that case above design fails. It does not follow the DIP.

Hence , here the high level policy is dependent upon low level modules and abstractions depend upon details.
We can modify above design in following way to suite and follow the DIP:



In above example the abstraction of Button class is isolated from detailed implementation.
The button class knows nothing of the physical mechanism and also knows nothing about the lamp. Thus the high level policy is reusable with any kind of button and with any kind of device that needs to be controlled.
It is not affected with changes in low level mechanisms.

Summary:
In the above part, we have seen the characteristics and principles of a good design (OOAD). Furthermore, one can also explore the principles about packages.

Events & Delegates

In modern programming world, we usually come across many UI centric applications. All these applications wait for user action to happen like clicking on a button, or a link, hovering over an image etc.

All these are known as actions generated by the user or any other programming logic. Events are thus a message sent by an object to notify the occurrence of an action. The object that raises the event is known as the event sender and the one who receives it and responds to it is known as event receiver.

In event communication, the event sender class does not know which object or method will receive (handle) the events it raises. What is needed is an intermediary (or pointer-like mechanism) between the source and the receiver. The .NET Framework defines a special type that provides the functionality of a function pointer.

A delegate is a class that can hold a reference to a method. Unlike other classes, a delegate class has a signature, and it can hold references only to methods that match its signature. A delegate is thus equivalent to a type-safe function pointer or a callback. While delegates have other uses, the discussion here focuses on the event handling functionality of delegates. A delegate declaration is sufficient to define a delegate class. The declaration supplies the signature of the delegate, and the common language runtime provides the implementation. The following example shows an event delegate declaration.

Consider following class: BankAccount

It has data field member like balance and a property to access the same. This property also has the logic embedded to check the minimum amount allowed/ required and raises an event accordingly.

The 2 events that are declared are lowbalance and sufficientbalance.

```
class BankAccount
    {
        public delegate void Mydel(float val);
        public event Mydel lowbalance;
        public event Mydel sufficientbalance;

        private float balance;
        public float Bal
        {
            get
            {
                return balance;
            }
            set
            {
                balance = value;
                if (value < 500)
                {
                    lowbalance(balance);

                }
                else
                {
                    sufficientbalance(balance);
                }

            }

    }


}
```

Now , while raising the event , we have to specify an eventhandler via the delegate who will handle the event at runtime. This event handler must have the same signature as that of the delegate. The delegate is thus a function pointer.

```
BankAccount objAcc = new BankAccount();
            objAcc.lowbalance += new BankAccount.Mydel(objAcc_lowbalance);
            objAcc.sufficientbalance += new
BankAccount.Mydel(objAcc_sufficientbalance);

            objAcc.Bal = 1000;
```

Following are the event handlers that are generated by Visual Studio or we can specify them:

```
    static void objAcc_sufficientbalance(float val)
        {
            Console.WriteLine("Insufficient Balance!");
```

```
}

static void objAcc_lowbalance(float val)
{
    Console.WriteLine("Insufficient Balance!");

}
```

The use of delegates is involved in event handlers, callbacks, asynchronous calls and multithreading, among other uses.

Delegates vs. Interfaces

Delegates and interfaces are similar in that they enable the separation of specification and implementation. Multiple independent authors can produce implementations that are compatible with an interface specification. Similarly, a delegate specifies the signature of a method, and authors can write methods that are compatible with the delegate specification. When should you use interfaces, and when should you use delegates?

Delegates are useful when:

- A single method is being called.
- A class may want to have multiple implementations of the method specification.
- It is desirable to allow using a static method to implement the specification.
- An event-like design pattern is desired .
- The caller has no need to know or obtain the object that the method is defined on.
- The provider of the implementation wants to "hand out" the implementation of the specification to only a few select components.
- Easy composition is desired.

Interfaces are useful when:

- The specification defines a set of related methods that will be called.
- A class typically implements the specification only once.
- The caller of the interface wants to cast to or from the interface type to obtain other interfaces or classes.

Summary:

Events and delegates are an inseparable part of event driven approach and are used as function pointers (like in C++). To explore more, one can also study more on the types of delegates :

1. Single caste delegate

Multicast delegates

C# version wise features at a glance

We have seen basic features of C# including Object Oriented Programming, Arrays, Types, Classes and most important events – delegates. All the features discussed so far exists with all versions of C# since those are the basic ones.

Kindly note, if any feature is launched in C# 1.X then it continues to be there in 2.X, 3.X, *.X as well. Microsoft continued its efforts to improve C#. Microsoft has launched six versions of C# so far. If we consider first version as C# basics learned so far then below is list of versions as related noticeable features:

Year / Framework / C# Version	Features
2002 / 1.0 / C# 1.0	• Basics
2005 / 2.0 / C# 2.0	• **Generics** A concept similar to 'TEMPLATE' in C++. This helps one write the function code irrespective of the parameter types. • **Partial Classes** This feature helps maintain design and code separate in terms of maintenance. • **Anonymous Methods** This approach helps developer write a method without a name. Actual name is offered by compiler. • **Nullable Type** With this feature values type variables can be assigned with NULL value. This helps bridging the gap between database NULL and programming NULL • **Iterator** This helps in looping through array or collections using FOR EACH Loop
2008 / 3.5 / C# 3.0	• Implicit Type (VAR) This is to be used when one need not know what is on the right hand side of the assignment. Whether on the right hand side of the assignment operator exists a value type or reference type; VAR changes itself accordingly. It is more or less compiler feature than C# • Auto Property GET, SET function body along with private member will be

	automatically defined. • Object Initializer Instead of writing more of ceremony code to create and call constructors for a class; this becomes quickest way to set the values for class members using bracket { } based syntax • Anonymous Type A type without a name. Class name will be offered by compiler. More or less compiler feature • Partial Method A method partially written in one partial class and defined in other partial class. Refers to the partial class feature from C# 2.0 • Extension Method Instead of changing existing class's code to extend it one can author separate class and can call the method inside new class as if it's a method of first class. Even if the source code for first class is not available or in case first class is sealed; still one can extend any class • Lambda Expression A way to define delegate without a name calling anonymous method. Used heavily in LINQ queries • Expression Tree With this features; function statements are considered as if those are nodes of a binary tree and executed • LINQ Language Integrated Query is a way with which SQL like queries can be fired on collections, SQL Mapped Objects, and XML Nodes.
2010 / 4.0 / C# 4.0	• Dynamic Type This type of variable gets to know the data being stored at runtime. It uses 'reflection' technique to find out the data's type at runtime and casts itself accordingly. • Optional , Named parameters Easy alternative to writing overloaded methods. Similar to default parameter concept in C++ but only change is sequence of the parameter can be changed unlike C++
2012 / 4.5 / C# 5.0	• Async, Await A new way to methods in Asynchronous manner

2015 / 4.6 / C# 6.0	• Auto Property Initializers A way to initialize 'AUTO' properties while being declared inside a class. This is dependent on the ;Auto Property feature in C# 3.0 • Primary Constructors A way of defining constructor with generic parameters.

In upcoming parts, we shall see important features from various versions of C# in detail.

Summary:

We have observed in this part:

- List of features in different versions of C#
- Short description of those features

Collections

In any language, collection types play a very important role. C# .NET supports multiple collections. In this part we are going to discuss various collections and answers to questions like where and why to use specific collections. In particular, we shall discuss about below types of collections:

- Arrays
- Object Arrays
- ArrayList
- Hashtable
- Generic List
- Generic Dictionary

Lets us start with first type.

Arrays:

As any language may have this common collection, C# also has it. Array means a group of elements. Array has fixed size and can't be re-dimensioned in C#. If one has to define an array in C#; either the elements needs to be known prior or size must be fixed in prior. E.g.

In below syntax; we have used arrays with elements known in prior:

```
int[] arr = new int[] { 10, 20, 30 };
```

Elements in above array can be accessed using index e.g. arr[1] will return 20. Index refers to the position of element inside array. While referring to the position, index starts with zero.

In second syntax example below, we have defined array size first and elements later:

```
 int[] arr = new int[3];

arr[0] = 10;

arr[1] = 20;

arr[2] = 30;
```

Same indexing concept discussed earlier is also applicable in second example.

Although arrays are vital in any programming language; we have two main problems:

1. Array always contain same type of elements
2. Arrays have fixed size

Can we address this problem? If yes how? Let us address these problems one after another.

If one wants to have arrays which can contain any type of elements; then one will have to define array of type 'OBJECT'. Let us talk about the OBJECT arrays now:

Object Arrays

We will have to go to little basics. In .NET, we have universal type called 'OBJECT' which can hold anything – value type data or reference type data.

E.g. both of the following statements are true in .NET:

1. Object obj = 100;
2. Object obj = new Customer();

First statement value actually was supposed to be on stack; which gets casted into reference type called OBJECT. (OBJECT is reference type because OBJECT is a class!) Getting the data over stack or inside value type into OBJECT is actually called as BOXING. With more boxing statements in code; efficiency of program reduces.

In second statement; Customer class object gets wrapped into another reference type wrapper called OBJECT!

Since OBJECT can store anything; it becomes very difficult when one needs to get the data from object type variable. Like in first case; one can't blindly convert object type into INT! Why? Because who knows what it contains?

So, it is always better to compare the type inside object type variable and then do the casting like:

```
if (obj is int)
{
  int j = Convert.ToInt32(obj);
}
```

In this case converting object again into integer type is well known as un-boxing. With more un-boxing statements in code; efficiency of program reduces.

While converting OBJECT's data into Customer type; one must do the casting. However, in this case as well; it is recommended that compare and then do the casting instead of blindly doing it. Code for the same will be:

```
if (obj is Customer)
{
  Customer c = (Customer)obj;
}
```

Considering above problems with OBJECT type; if one decides to have an array - which can contain any type of data - and decides to OBJECT array; we will have casting and boxing problem in bulk!

E.g. consider below statement:

```
object[] arr = new object[3];
arr[0] = 10;
arr[1] = 20;
arr[2] = new Customer();
```

In above code; element zero and one are value types & are stored in OBJECT. This is boxing. Element three is reference type & is stored in OBJECT. This is casting. So, twice boxing and once casting has happened in the code. While getting the values back one will have to go through twice un-boxing and once re-casting! This is going to reduce the performance. So, using OBJECT type array is going to have this problem all the time.

Now, let us focus on second problem.

Can we somehow make this OBJECT array expand instead of fixed size specification?

That means we want to have OBJECT array by no size limit! In such case we can use ARRAYLIST type collection in C# .NET.

Let us discuss on the same:

ARRAYLIST:

Arraylist collection in .NET works like an OBJECT array with resizable behavior! One will have to import SYSTEM.COLLECTIONS namespace in to the project to get ARRAYLIST. Syntax for declaring and using ARRAYLIST is simple. Sample code is given below:

```
ArrayList arr = new ArrayList();

arr.Add(10);

arr.Add(20);

arr.Add(new Customer());
```

It can be seen from the code above; one can continue adding any type of data into the ARRAYLIST. So, it is array of OBJECTs internally. However, specification of size is not required. If seen from different angle code looks like: “SINGLE OBJECT OF A ARRAYLIST CLASS IS TREATED & USED AS IF IT IS AN ARRAY OF OBJECTS” This concept in C# is well known as INDEXER.

In the code above, similar to object array; twice boxing and once casting happens.

While, referring values back (e.g consider element number zero) from the ARRAYLIST; one will have to use below code:

```
if (arr[0] is int)

{

  int j = Convert.ToInt32(arr[0]);

}

if (arr[0] is Customer)

{

  Customer c = (Customer)arr[0];

}
```

As can be seen from the above code; we have twice un-boxing and once re-casting. This is same problem we discussed in OBJECT array.

So what we have we achieved using ARRAYLIST over OBJECT ARRAY?

Just a size is not limited in case of ARRAYLIST!

Similarity is boxing & casting problem remains same as that of OBJECT array!

Now, let us talk about one more problem!

What if the size of the ARRAYLIST contents go to thousand and one needs to find specific element?

Solution to this problem is - iteration through all the elements - using FOR EACH or FOR loop to find the element needed. Sample code for the same is given below:

```
foreach (object obj in arr)
    {
        if (obj is int)
            {
                //Some logic here
                int j = Convert.ToInt32(obj);
            }
        if (obj is Customer)
            {
                //Some logic here
                Customer c = (Customer)obj;
            }
    }
```

A second approach: If one knows the index of the element then use the index to find the element inside ARRAYLIST like:

```
Object obj = arr[514];
//Conversion code after this
```

Is there any way with which in - below mentioned conditions & assumptions - one can find the element quickly without iteration?

Conditions & Assumptions:

1. ARRAYLIST is used to store Customer class objects only.
2. 1000s of Customer class objects exist in ARRAYLIST
3. Each customer has unique identity which is alphanumeric ID
4. Index of the element to be searched in not known to the end user
5. Search of the specific element is suggested based on ID!

If one uses an ARRAYLIST then above problem can't be solved without iteration and comparison of ID! This process will definitely take time as one will have to touch many objects in the collection to find required one.

In such cases, where one needs to find element quickly in ARRAYLIST like collection; we can think of using another collection in .NET – called as HASHTABLE!

Let us talk about it now.

HASHTABLE:

As we know the problem with ARRAYLIST now, how can HASHTABLE solve the same? Actually HASHTABLE is one of the two dimensional collections in .NET. This collection has key and value pair representing single element. It means one element in HASHTABLE is actually a pair!

With HASHTABLE one can store many elements like ARRAYLIST and each element can be attached with unique value which can be a number or characters – also known as key.

To solve the problem given above; let us use below code:

```
Customer customer1 = new Customer();
customer1.ID = "MyID1";
//---------SOME MORE DETAILS AFTER THIS
//-------------------------------------
//-------------------------------------
Hashtable table = new Hashtable();
table.Add(customer1.ID, customer1);
```

As one can see; HASHTABLE, uses method ADD similar to ARRAYLIST to store an element. However, it takes two values as an input. One is KEY and other is called VALUE.

VALUE is actually an element one wants to store in the collection!

In this case, we have used Customer's alphanumeric ID itself as a KEY. Imagine we have 1000s of such Customer class objects in this collection with unique IDs for each one.

How can one find specific element from the HASHTABLE?

Use "KEY" to find the "VALUE" - like:

```
object obj = table["MyID1"];
```

Above line will return Customer1 object into the 'obj'. Why we have OBJECT type on the left hand side of EQUALTO instead of Customer? Well, HASHTABLE stores all keys and respective values as an OBJECT!

It means BOXING, CASTING is in too much bulk in HASHTABLE.

If one decides to iterate through all the objects in HASHTABLE then the loop can be run using the KEYS collection like below:

```
foreach (object key in table.Keys)
{
  object obj = table[key];
  //LOGIC HERE
}
```

Now, so far we have used few collection types in .NET and each has some or other problem!

- Arrays with specific type must have sizes defined & one kind of data only
- Object Arrays can store anything but again have size and boxing, casting problems
- Arraylist can solve the size issue but sticks with boxing and casting problems
- In Arralylist, finding element without knowing an index needs iteration thoroughly
- Hashtable can help in searching element based on UNIQUE keys but continues with boxing and casting problem as KEY and VALUE both are stored as OBJECTs in hashtable

If one wants to have Customer type array of unlimited size then what can be done?

Answer is to use GENERIC Collection in .NET!

Let us understand the concept of GENERICs first and come back to collections in some time. Let us a simple example of authoring a function in simple C# class which swaps the elements using REF keyword (i.e. SWAPPING using variable passed by reference):

```
public class Test
  {
     public void Swap(ref int x, ref int y)
     {
        int z = x;
        x = y;
        y = z;
     }
  public void Swap(ref string x, ref string y)
     {
        string z = x;
        x = y;
        y = z;
     }
  }
```

Notice, logic for swapping of INTEGERS and STRING in overloaded method called SWAP is almost same. Only TYPEs have changes! Can we write down the logic for SWAP function in such a way that the TYPE of the variable itself can be passed as a parameter! Well this concept is not new for C++ developers. In C++, we know this concept by the name TEMPLATEs. How can we achieve the same in C#?

Template like concept in C++ is offered in C# by the name GENERICS. So, how can we use GENERICs in above example? Here goes the code:

```
public class Test<T>
   {
      public void Swap(ref T x, ref T y)
      {
         T z = x;
         x = y;
         y = z;
      }
   }
```

Here <T> denotes a generic type which can get the value later on at runtime (Runtime means while program is going to get compiled from MSIL into Native code). Instead of using <T> as a word one can use any characters or words!

How do we use this SWAP method for swapping of two integers then? Follow the code:

```
int x = 100;
int y = 200;
Test<int> test = new Test<int>();
test.Swap(ref x, ref y);
//Logic to print the x and y values.
//x & y will show swapped values now!
```

As one can see, we have mentioned <T> as integer while declaring object of type TEST.

This feature helps one write the logic without worrying the datatype!

Microsoft did build generic classes and some of them are collection classes too!

Let us discuss now about GENERIC COLLECTION CALLED AS LIST<T>

One will have to import SYSTEM.COLLECTIONS.GENERIC to get LIST<T> method and approach to access inside elements remains same as that of ARRAYLIST. Refer the code below to add elements inside LIST of CUSTOMERS:

```
Customer c1 = new Customer();

Customer c2 = new Customer();

Customer c3 = new Customer();

List<Customer> customers = new List<Customer>();

customers.Add(c1);

customers.Add(c2);

customers.Add(c3);
```

In order to fetch the element by index, one can use code below:

```
Customer c = customers[2];
```

Please note, since “customers” is a collection of type Customer only; there is no need to do the casting in this example.

If one wants to iterate through all the elements; use FOR or FOREACH loop like below:

```
foreach (Customer c in customers)

{

  //use "c" here in the code

}
```

One can use other collection types in .NET like STACK or QUEUE.

In this collection only one old problem continues now. What if one wants to find an element from the collection quickly using some identity? Solution for this problem remains same i.e. either know the index and refer specific element or iterate through entire collection and find respective element.

Can we have behavior like hashtable but without OBJECT's BOXING and CASTING problems?

Answer is YES!

We can use another two dimensional GENERIC type collection in .NET called as DICTIONARY.

How do we use it? - In same way as that of HASHTABLE. Refer code below:

```
Dictionary<int, Customer> customers =  new Dictionary<int, Customer>();
 customers.Add(c1.ID, c1);
 customers.Add(c2.ID, c3);
 customers.Add(c3.ID, c3);
```

While referring values back from the dictionary; one can either use KEY to find VALUE / ELEMENT like code given below:

```
Customer c = customers[2];
```

Or one can iterate through "KEYs" collection and find the respective element. Refer code below:

```
foreach (int key in customers.Keys)
    {
       Customer c = customers[key];
       //Use c here onwards!
    }
```

Thus dictionary becomes a very widely used collection in .NET because:

- It does not have limits
- One store and define KEYs and VALUEs and the type can be defined while declaring the DICTIONARY itself!
- CASTING is not required if the VALUE is of some type other then OBJECT!
- Using DICTIONARY<OBJECT, OBJECT> is like using HASHTABLE only!

Summary:

We have discussed important collections used in C# .NET and tried to understand problems and respective solutions. We discussed about below collections:

- Arrays
- Object Arrays
- ArrayList
- HashTable
- List<T>
- Dictionary<TKey, TValue>

One can very well write own collection by inheriting the base classes!

C# Features - I

We have seen few important concepts like collections and generics introduced in C# 2.0 in previous part. In this part, we will discuss four important concepts:

1. Partial classes
2. Anonymous methods
3. Iterators
4. Nullable types

Partial Classes:

We are already aware of the way a structure or a class is written in previous version of C# as well as in C++. Till now we know that the implementation of a class / structure resides in a single source file. In bigger projects, where the business logic is too vast and complex; the responsibility of coding is distributed between multiple developers. If now the class source file is in use by one developer, then any other developer will just have the read only access. To facilitate this, the class can span between multiple source files, but it will be compiled as single unit.

It is possible to split the definition of a class or a structure, over two or more source files. Each source file contains a section of the type or method definition, and all parts are combined when the application is compiled.

They will be useful in following scenarios:

- When working on large projects, spreading a class over separate files enables multiple programmers to work on it at the same time.
- When working with automatically generated source, code can be added to the class without having to recreate the source file. Visual Studio uses this approach when it creates Windows Forms, Web service wrapper code, and so on. You can create code that uses these classes without having to modify the file created by Visual Studio.

Partial classes will require *partial* keyword and same name ofcourse:

Coded in Src1.cs file

```
namespace OOPDemo
{
    public partial class Src1
    {
    }
```

```
}
```
Coded in Src2.cs file
```
namespace OOPDemo
{
    public partial class Src1
    {
    }
}
```

Some very important considerations are as follows:

The **partial** keyword indicates that other parts of the class, struct, or interface can be defined in the namespace. All the parts must use the **partial**keyword. All the parts must be available at compile time to form the final type. All the parts must have the same accessibility, such as **public**, **private**, and so on.
If any part is declared abstract, then the whole type is considered abstract. If any part is declared sealed, then the whole type is considered sealed. If any part declares a base type, then the whole type inherits that class.
All the parts that specify a base class must agree, but parts that omit a base class still inherit the base type. Parts can specify different base interfaces, and the final type implements all the interfaces listed by all the partial declarations. Any class, struct, or interface members declared in a partial definition are available to all the other parts. The final type is the combination of all the parts at compile time.

Anonymous methods

In Part 9, we have seen the usage of delegates. Delegates could be used till C# 2.0 only with named methods. That is, when we are associating an event with event handler, we always had to specify the signature of the delegate and pass a function pointer i.e. address of the method to be invoked.

In continuation with the same example which we saw in Part 9 :

```
BankAccount objAcc = new BankAccount();
objAcc.lowbalance += new BankAccount.Mydel(objAcc_lowbalance);
```

Here we have used the named method : objAcc_lowbalance

Now, if we use the anonymous method here, we do not have to declare the delegate separately and define the signature. We can simply write the code in following way:

```
objAcc.sufficientbalance += delegate {Console.WriteLine("Sufficient
Balance!"); };
```

Thus, when we are declaring the event sufficientbalance, we have to just specify the keyword delegate and then provide the implementation. Here we are talking about no parameters being passed. But if we want to pass parameters, this can be done in following manner:

Iterators:

Many a times it happens that we have huge lists of names, objects of specific types etc. unless and until we iterate through the data, we cannot use it. We may or may not know the size list or even the type of lists. The various methods to iterate through a list are as follows:

```
string[] cities = {"New York","Paris","London"};
foreach(string city in cities)
{
    Console.WriteLine(city);
}
```

In above example, we can iterate through the list simply using a *foreach* loop.

One more way is to implement the IEnumerable interface.

```
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
public interface IEnumerator
{
    object Current { get;}
    bool MoveNext();
    void Reset();
}
```

In C# 2.0, this iteration, is made much easier. We need not implement the IEnumerable interface. When the compiler detects the compiler detects the iterator, it will automatically generate the Current, MoveNext and dispose method.

Nullable Types:

To learn more about Nullable types, we should first understand some significant differences between value types and reference types. Value types cannot hold a very significant value of *null.* Whereas, when we say the reference type holds nothing, it necessarily means that it holds *null* value. This is a very promising feature for the developers, since they are sure when to expect null values returned from a function or pass null values. Also, this becomes useful while assigning value to any db value.

It is difficult when we want to interpret a null integer value retrieved from database or from XML document. Then we have to simulate it , as if it can hold null value.

- Either a wrapper structure is created, holding 2 values, one is integer value and other a Boolean to hold either 0 or 1 to interpret null integer. This is of course burden on the developer.
- A wrapper class can be written to maintain both value and Boolean value. But this also poses a burden on the garbage collection.
- Boxing – unboxing techniques also are a burden on the garbage collector.

In order to overcome this, in C# 2.0, System.Nullable <T> generic structure is introduced as follows:

```
public struct System.Nullable<T>
{
    public Nullable(T value);
    public static explicit operator T( T? value );
    public static implicit operator T?( T value );
    public bool HasValue { get; }
    public T Value { get; }
    public override bool Equals( object other );
    public override int GetHashCode();
    public T GetValueOrDefault();
    public T GetValueOrDefault( T defaultValue );
    public override string ToString();
}
```

The <T> can be replaced by any value type in the following manner;

```
Nullable<int> varnull=null;
    varnull= 3;
    if (varnull.HasValue)
    {
        // perform operations
    }

    else
    {
```

```
        // it has null ; perform next set of operations
    }
```

C# Features - II

Microsoft continued its efforts to enhance C# in this version as well.

Many features listed in this version may not make enough impact when seen individually. When seen as a group of features to create best in performance application; these feature(s) make more sense. So, what do we have in a plate?

C# 3.0 was launched in 2008 along with Visual Studio 2008. Below are the basic features listed from C# 3.0:

1. Implicit Type
2. Auto Property
3. Object Initializer
4. Anonymous Type
5. Extension Method
6. Partial Method
7. Lambda Expression
8. Expression Trees
9. **L**anguage **IN**tegrated **Q**uery
    - To Collection
    - To SQL
    - To Dataset
    - To XML

Out of all the features listed above; we shall see first 6 features in this part. Rest of the features will be discussed in .

Let us start with first feature in C# 3.0:

Implicit Type:

This feature refers to a new type launched in C# 3.0 known as “VAR”. This is different from the Java script “var” type.

While we use object data type in .NET to store / retrieve value type data; boxing and un boxing keeps on reducing performance.

E.g. while we use below statement in C#; boxing happens:

object obj = 100;

And when we use below statement in C#; un boxing happens:

```
int i = (int) obj;
```

Why do we have to mention the type while retrieving the data from object type?

Answer is, object type in .NET can be used to store anything including any reference type data or value type. Value type data is stored on stack. Reference type data is stored on heap and reference is maintained on stack.

How about there exists a type which when assigned with data; it would evaluate at compile time and will make sure the data either will get stored on stack or heap. Later on when one needs the data; there would be no need to convert the same!

You wished and .NET has it. It's VAR!

How does it work?

At compile time; left hand side variable in below statement - will decide whether it is going to hold reference type or value type - which is mentioned on the right hand side of "=".

e.g.

```
var v = 100;
```

- Will make the 'v' to be strictly value type storing integer storing value 100 on stack.

```
var v = new Customer();
```

- Will make the 'v' to be strictly reference type storing customer class type objects on heap

However, var has certain standards set.

Ex: if on the right hand of side value is 100; var will consider the value as integer and not short although the range of the value is in limits of short.

So, in what cases we will use var?

Answer is when we don't know what is going to be there on right hand side of '='.

But let me tell you; such situation won't appear in programming that you / me have done so far!

We always know what is on the right hand side of '='

Either we know the data, type if not the function on the right hand side will tell us the return type!

Then why we have this type?

Well, Wait! We are yet to meet the situation. We shall revisit the same in some time.

Also, look at the drawbacks of the same:

- 'var' can be always used in local scope
- 'var' can't be declared as global also can't be specified as a parameter or return type of the method.

Auto Property:

This is frequently used feature but usage is more than what developers understand.

What is this?

With this feature; one does not have to define a private variable which is normally manipulated using GET and SET function(s).

- *"Wait! Are GET() and SET() functions?"*
- *"Yes!"*
- *"How?"*
- *"Look for the answer right here ↓":*

When one writes a private variable and GET SET property in following manner:

```
Private string _Name;

Public string Name

{

 Get {return _Name;}

 Set  {_Name = value;}

}
```

After compilation; C# compiler will anyway convert the code above in MSIL as function(s) which look similar to function(s) like:

1) Get_Name() returning string
2) Set_Name(string value)

So, when one needs to manipulate the private member then one writes the code in GET{}, SET {}. Actually, one can very well massage the data as well. E.g. Validation, data checking can be done in GET and SET.

Given assurance of data security; what if one just wants to set or get the value to and from the private member? If one wants to only set or get value; then how about we not writing the code for GET or SET or even for private member declaration!

It's not like no code is written! It is more like - left to compiler!

So, how do we write the code then?

Here it is:

Public string Name {get; set;}

That's it! Compiler will write a private member after compilation! You will not know the name; also setting and getting variable code is automatically written!

This concept is called as auto property!

Object Initializer:

When one class has many private members; what do you do to assign values to those members?

You end up defining properties. Either you call setter property individually after creating object of the class or you may optionally accept the values in constructor and set the respective properties / members.

- *"What if number of elements to be initialized is flexible? Should you end up writing many over loaded constructors?"*
- *"No! At least not with this version of C#!"*

One can define object of the class and initialize properties as per needed using this concept of object initialize.

Here is the syntax for the same:

If Customer is the class like below:	Then, this is how object initializer can be used to define object of Customer class:

public class Customer
{
public int No { get; set; }
public string Name { get; set; }
public string Address {
get;set;}
}
	Customer cObj = new Customer() { No = 1, Name = "ABC"};
	
	Keep a note; here we have used object initializer syntax to initialize only two properties!

Object initializer concept makes the code go little short!

How does it work?

Well, again compiler plays a very vital role here. It breaks your statement:

```
Customer cObj = new Customer() { No = 1, Name = "ABC"};
```

Into similar line which you would have written in normal case like:

```
Customer cObj = new Customer();
 cObj. No =1;
cObj.Name = “ABC”;
```

- *“Will we really see something similar?”*
- *“Yes! Check MSIL buddy! Use ILDASM command”*

So, what can be seen is; it reduces the amount code written while initializing members.

Anonymous Type:

This feature when seen individually seems redundant. It is not!

From the features discussed so far, it is clear that most the features are because of the compiler’s work. E.g. In case of:

- Implicit type: compiler makes the variable - strictly typed – based on the right hand side assignment.
- Auto property: compiler creates the private member (which can be seen in the MSIL) based on the property type you define.

Similar to above discussed features; anonymous type is also compiler’s Job.

We know what below statement means:

```
Var cObj = new Customer() { No = 1, Name = "ABC", Address = "Pune" };
```

Compiler will strictly cast ‘cObj ‘as Customer type object. This statement onwards cObj can only hold Customer type objects.

Anonymous type object in C# declaration is little bit weird! It goes like this:

```
Var obj = new { No = 1, Name = "ABC", Address = "Pune" };
```

- *"If 'Var' is going to get its type based on the right hand side assignment; what type of objet is there on the right hand side?"*
- *"Does it look like an object definition?"*
- *"Yes"*
- *"How?"*
- *"It contains **new** keyword"*
- *"This is object of a type which we did not create… It is created by compiler"*
- *"Can we see the type? 'YES' - but in MSIL only"*

Here it is: (See in below MSIL snippet; It is defined as <>f_AnonymousType0)

Now, why do we have this feature? Where are we going to use it?

This feature is very important and makes sense in LINQ queries – which is another feature we are yet to discuss.

Are you curious about this now?

Well, Most of the feature(s) listed on first page; are going to be used while writing LINQ queries.

Some of the limitations that one should know about 'Anonymous Type' are listed below:

- Anonymous Type can't be inherited. Because we don't write and know the type itself. It is created by Compiler.
- Anonymous Type objects are more of read only. Look at the MSIL code discussed earlier in this part. You will notice that 'ONLY GET PROPERTIES' exists. There are no 'SET' properties. It means while defining the object whatever data is assigned to properties declared inside { } brackets – is going to stay as it is. Such objects can be used for representing the data. One just can't reassign the data. If we try setting the data to such properties we will end up in exception
- E.g. If we try to set new value to 'Address' property:
  Obj.Address = "Mumbai";
  We would end up in exception at compile time:

Property or indexer 'AnonymousType#1.Address' cannot be assigned to -- it is read only

We shall discuss more on this type during LINQ topic.

Extension Method:

In one of my project, I received this requirement where in we required 'STRING' class functionalities and additionally few more. Since the additional functionalities required were more in number; Inheritance seemed to be obvious solution.

In the library project (.DLL) one of the developer, added a new Class called "STRINGExtension" and tried inheriting .NET "STRING" class.

- *(TING!) "Exception!"*
  STRINGExtension': cannot derive from sealed type 'string'

It means String class in .NET is sealed and can't be inherited.

Only one option was left to the developer then – writing a utility class which will do additional functionality on the string type parameter passed.

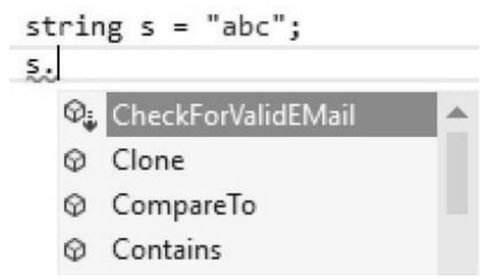
After little research; we got to know that it is easy.

One can very well extend any class even if it is sealed by writing an extension method.

- *"WAIT! WHAT? Is it not a defeat to the 'SEALED', 'FINAL' concept in OOP?"*
- *"No! "*
- *"How come?"*
- *"Well, while using extension method, it looks like we have inherited the class. Actually, while writing it is more of writing utility class only!"*
- *"This is how we added extension method 'CheckForValidEMail()' as an extension to sealed type STRING" :*

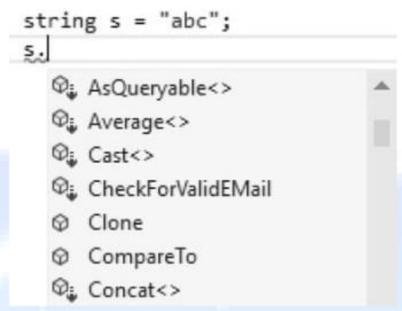
```
public static class STRINGExtension
  {
    public static bool CheckForValidEMail(this string str, object ParameterForNoReason)
     {
        //Logic goes here
        return true;
     }
  }
```

One will have to import the namespace in which this STRINGExtension class is written. While, using string this class in the project; one can easily see intellisense against string class suggesting CheckForValidEMail() method:



Extension methods can be identified in intellisense window with down arrow ↓ symbol.

If you wish to see the usage of Extension methods; just import “System.Linq” in to your project and see what happens in above intellisense window:



As you can notice, after importing “System.Linq” into the project; you can notice many extension method(s) appear in intellisense.

- *“Are all those extension method(s) in intellisense extend the type “STRING?”*
- *“No! Most of them are extending type “<T>” i.e. GENERIC(s) “*
- *“How many parameters do I need to pass in order to call CheckForValidEMail() method?”*
- *“You need to pass only one i.e. ParameterForNoReason”*
- *“Why?”*
- *“Because **OBJECT WITH WHICH WE CALL THE EXTENSION METHOD IS AUTOMATICALLY PASSED AS A FIRST PARAMETER”***

This feature is very important when one does not want to create new version of the code or function(s) or libraries. Rather it is more frequently used to launch minor framework(s) which can be referred into the projects without replacing the basic functionality libraries.

- “Can extension methods be used with Anonymoud Types?”
- “Dude! What does <T> mean?”

Partial Method:

This feature is always misunderstood by many as a useless feature!

FILE 1 Contents (Imagine this is auto generated class with partial method declared only using some tool or means)	FILE 2 Contents (Imagine this is the class that developer writes with partial method containing actual logic)

`namespace A`
`{`
`  public partial class Customer`
`  {`
`  partial void Validate(object SampleParameter);`
`  }`
`}`
	`namespace A`
	`{`
	`  public partial class Customer`
	`  {`
	`  public partial void Validate(object SampleParameter)`
	`  {`
	`Console.WriteLine(SampleParameter.ToString());`
	`  }`
	`  }`
	`}`

This is easy to misunderstand kind of feature. Partial method can be declared inside one partial class and can be defined inside another partial class - with the same name - inside same named namespace in the project. Due to below limitation(s) developers misunderstand it:

1) Partial methods can be declared and defined once in partial classes only
2) Partial methods have 'VOID' as return type
3) Partial methods are by default private and can be accessed inside the class only

Now, why would someone go for partial methods then?

Answer is there in the table above.

Notice, one partial method declared in File 1. Imagine this partial class is not written by a developer. Consider it is auto generated using framework like Entity framework.

Let us explore the example.

Consider you want to create ans 'Employee' class with No, Name, Address GET/SET properties. You want to use this class's objects to hold data from Employee table in Database.

Instead of you writing the class matching to database table's column structure; you decide to use a entity framework.

Entity framework or any ORM tool can help generate the entity classes like Employee by simple drag and drop!

One can see the code generated automatically inside the EDMX extension based file(s).

Now, consider Entity framework has written a property automatically which is going to GET / SET the respective private member. You want to validate the data before actual GET/SET takes place. So you open the source code from EDMX extension file and try to manipulate it.

- *TADA! Look at the top side message in the file.*
  "This file is auto generated….Do not modify…….. & so on… "

This is very common message written at the top of all auto generated file(s).

Now, what to do? If one wants to validate the GET / SET properties?

Well, since 2005 C# every code generator tool is consistently generating all classes as partial classes. ORM tool like Entity framework also generates the partial class along with Partial method declared like 'Validate'

In our example we are considering one input parameter but actually in such validate methods one can observe 2 parameters! One is name of the property and one is value in the form of object type.

Well one can also notice, this declared method being called at respective SET methods.

- *"What if I don't define the partial method?"*
- *"Don't worry! CLR is very intelligent! If it finds method is declared but not defined it simply ignores the call given to such method!"*
- *"Will there be any confusion if I define the method more than once?"*
- *"You can't define the body more than once for declared partial method! You try and you get compile time error!"*

So, for code generator frameworks 'Partial method' concept is absolute must!

Rest of the feature we shall see in next part!

Summary:

We have observed in this part:

- Features available in C# after 3.0 version
- Need for implicit type, auto property, object initialize, anonymous type, partial and extension method

C# Features - III

This part contains references to features discussed in . So, before reading this part, it is recommended that you read .

We are going to continue discussing C# 3.0 features in this part. We shall discuss:

1) Lambda Expression
2) Expression Trees
3) **L**anguage **IN**tegrated **Q**uery (LINQ)
    a. To Collection
    b. To SQL
    c. To Dataset
    d. To XML

Let us start one by one.

Lambda Expression:

"A lambda expression is an anonymous function that you can use to create delegates" - MSDN

If you recall events and delegate(s) from part(s); we used delegates (which are function pointers) in combination with events.

Consider we have a function defined like:

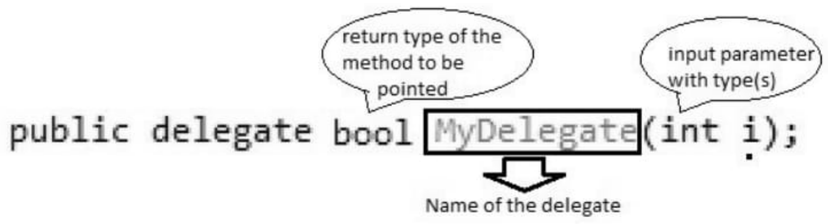
```
public bool check(int i)
 {
     return (i > 10);
 }
```

'Check' function accepts integer type parameter and returns true or false by comparing whether input integer is greater than 10 or not.

One can always call this function by invoking it like:

```
bool result = check(20);
```

Now if you recall there is one more way to call this function i.e. by using delegates. One will have to declare a delegate more like:



In order to use this delegate to call check() method; one will have to use below code:

```
MyDelegate pointer = new MyDelegate(check);
bool result = pointer(20);
```

Now, this way of calling a method is used in many scenarios including when we use EVENTs or THREADs in C# code.

EVENT, THREAD requires the use of delegates / function pointers. But pointers can be used without EVENT, THREAD. Like used in above code.

If the method reuse is not intended then we can also use Anonymous Method concept discussed earlier in C# 2.0 features. So what if anonymous method needs to be pointed with the delegate that we just declared.

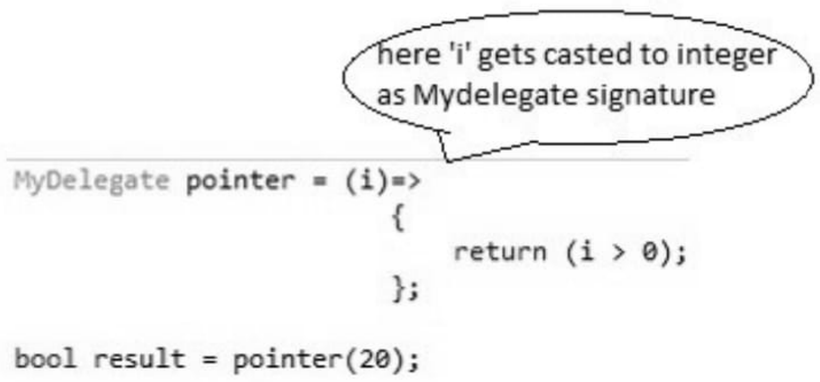
Note: Anonymous methods are the methods without name. Compiler takes the responsibility of offering names to these methods. Since the method names can only be seen in MSIL; we can't call the method by name. In order to call such method using delegate(s) is more compulsory.

Here is the code; if check() method is anonymous and needs to be called using MyDelegate:

```
MyDelegate pointer = delegate(int i)
                                {
                                      return (i>0);
                                };

bool result = pointer(20);
```

Look at the method without a name.

We can even avoid declaring the delegate with specific structure though. We can always use a concept of Lambda Expression. Look at below syntax which cut shorts the above code:



In above example; symbol => stands for 'Goes to'

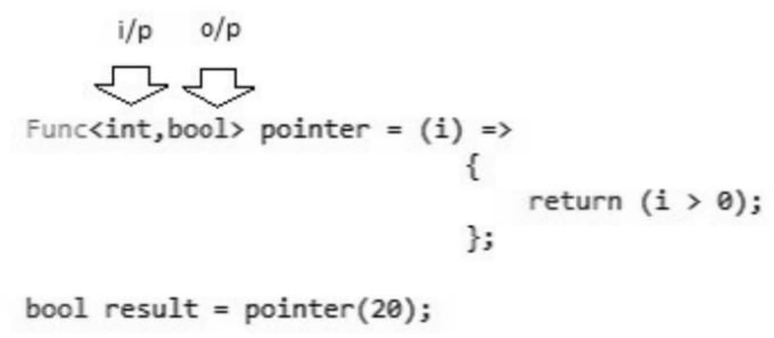
The way one calls this pointer remains same.

If you look from the other angle; we just have called an anonymous method using pointer to which 'i' variable's type is unknown on the right hand side. Rather it will be determined based on the MyDelegate pointer's definition.

With C# 3.0 onwards; we can make above statement shorter! Instead of defining a delegate 'MyDelegate' we can use generic type delegates like FUNC or ACTION.

- *"What! "*
- *"YES… you can use generic delegate FUNC to call a method which may take some data as input and returns something"*
- *"What is ACTION for?"*
- *"You can use generic delegate ACTION to call a method which may take some data as input and returns VOID"*

Here is how we can use FUNC delegate in above example:



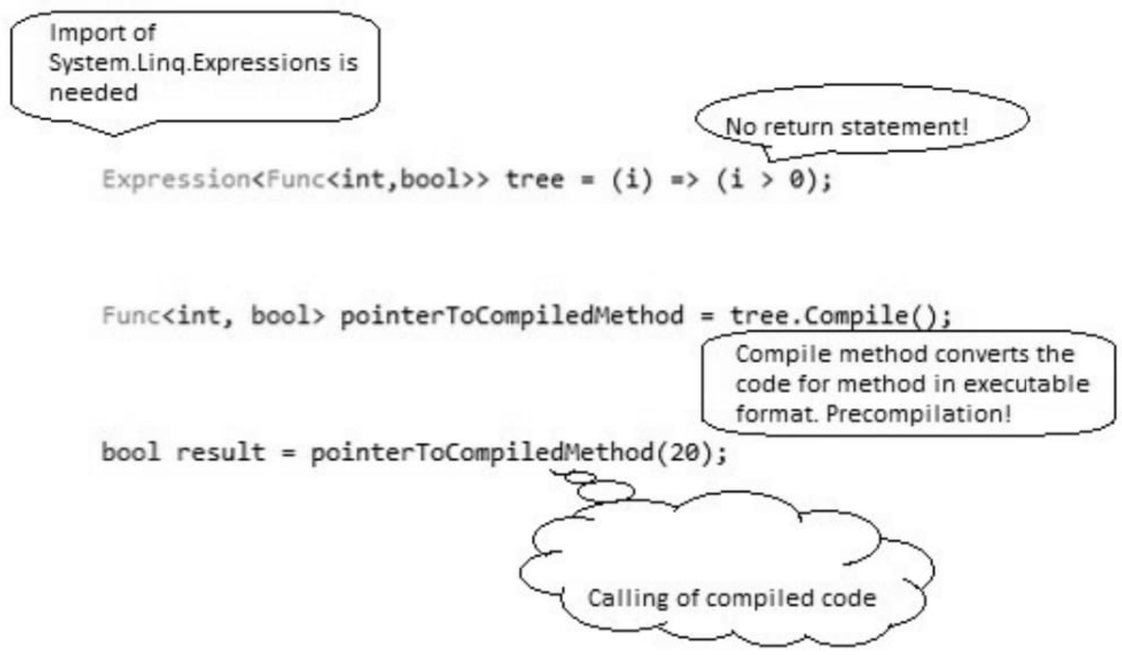
So, what is the real benefit we get out of this?

When we call check() method normally - versus - using Lambda expression along with FUNC delegate; one can see sudden performance improvement. This is because of the while executing the code using FUNC, Lambda code can be optionally compiled before actually it gets called. In order to compile this code optionally, one will have to use Expression Tree type in code.

Expression Tree:

Expression trees represent code in a tree-like data structure, where each node is an expression.

So, in order to create Expression Tree for above code; we will have use code given below:



- *"How come this improves the performance as the compile statement is in the code itself! Is it not true that we are spending some CPU cycles for Compile statement as well?"*
- *"Yes! But this is one time JOB and when one calls 'pointerToCompiledMethod' again; then compilation is not required! Because code would be in memory in compiled format!"*
- *"Well, but it's going to help only in case when multiple calls to method exist! Isn't it?*
- *"If you wish to call this compiled method but don't want to spend CPU cycles in this program then; we have another method instead of Compile()"*
- *"What is it?"*
- *"It's called "CompileToMethod()". It takes TypeBuilder object as a parameter. With rest of the dependent parameters in this method; one can pre-compile the code and save it inside an assembly - as a method - using AssemblyBuilder's Save() method"*
- *"Is this reflection technique?"*
- *"YES! YOU GOT IT BUDDY!"*

It is observed that if a normal call to a method takes 300ms then after Compile() method same will be reduced to single digit e.g. 3ms using expression tree!

Language INtegrated Query (LINQ):

Most of us are good at firing queries on database. Ever imagined how would it be if one can fire SQL like queries on collection, arrays or XML?

What do we do to find a Customer class object from the collection whose address starts with ‘M’?

Consider below collection of customer objects:

```
List<Customer> customers = new List<Customer>() {
            new Customer(){ No =1 , Name = "Amit", Address = "Pune"},
            new Customer(){ No =2 , Name = "Manoj", Address = "Mumbai"},
            new Customer(){ No =3 , Name = "Chetan", Address = "Manglore"},
            new Customer(){ No =4 , Name = "Prem", Address = "Panji"}
            };
```

As we can see the sample data; we have two customer objects with Address starting with ‘M’
It means after looping through entire collection; we shall get two objects. But this is sample data, may get any number of objects or one or none based on real time data. How to hold the data then? Should we go ahead and declare a collection or single object?

That’s the last thing to worry about. First let us be concerned about- “Can we make this easy”?

Yes!

Let us use LINQ to collection by firing SQL like queries on the collection. In order to hold the result let us use ‘VAR’ type discussed in earlier part.

- *“Why ‘VAR’?”*
- *“We don’t know what we are going to get!”*

Here goes the code for the same:

```
var result = from c in customers
             where c.Address.StartsWith("M")
             select c;
```

In the code above the SQL like syntax used is similar to Hibernate Query Language (HQL). Hibernate is one of the popular ORM framework.

Syntax is not straight forward like SQL SELECT statement. Had it been SQL one would have written query like:

“SELECT * FROM CUSTOMERS WHERE ADDRESS = ‘M%’ ”

Why do we have such a weird syntax?
Answer is - In the LINQ query used above; ‘FROM’, ‘WHERE’, and ‘SELECT’ are not keywords! Those are actually methods!! Output of one method becomes an input to another method. It is more of chain of methods.

Good point about this query is the query does not execute at the mentioned statement. It means if one tries to execute only LINQ statement above; result will actually have nothing!

LINQ query executes only when result is going to get used for the first time. It is called differed execution. Sometime it is also referred as 'Lazy execution'

It means when one actually tries to use 'result' variable e.g. may be iteration of the result; then the query executes.

However if one changes the query like given below; one may see the result with data immediately:

```
var result = (from c in customers
              where c.Address.StartsWith("M")
              select c).ToList();
```

In above statement; change is we have asked to convert the entire result of the query in to LIST format. It means we have asked for immediate result! Hence in this case query executes immediately.

Using this example, we have connected 'VAR' with LINQ. Let us connect more dots to see most of the C# 3.0 features at a glance.

In above scenario; we are selecting entire Customer object data. What if one needs only Number and Name?

We can define a new type to hold limited data then. E.g. Consider a new class defined like:

```
public class DataHolder
    {
        public int CNo { get; set; }
        public string CName { get; set; }
    }
```

Well, we will have to change the query like:

```
var result = (from c in customers
              where c.Address.StartsWith("M")
              select new DataHolder { CNo = c.No, CName = c.Name }).ToList();
```

Result will be List of `DataHolder` objects.

Look at the way we have defined CNo and CName. Do you remember anything?
YES! These are auto properties.

Also, can you see the way new `DataHolder` object is created everytime?
YES! That is the same concept we learned - Object Initializer!

What is the purpose of `DataHolder` class? If it is only to hold the data of the query and present on the screen then we can simple put the class out. Let us replace the same using Anonymous Type!

```
var result = ( from c in customers
               where c.Address.StartsWith("M")
               select new { CNo = c.No, CName = c.Name }).ToList();
```

Isn't it simple? Look at 'VAR', Auto Property, Object Initializer and Anonymous type concepts together in single query itself!

LINQ also offers its queries to node collection like XML or rows collection like Dataset Tables. However we are going restrict our self to this simple query itself at this stage.

Summary:

We have observed in this part:

- Advanced features available in C# after 3.0 version
- A new way to define delegate - Lambda Expression
- Performance improvement because of Expression Tree
- Usage of most of the C# 3.0 features in LINQ query

File IO & Serialization

A file is a persistent storage of ordered and named collection of bytes. .net framework offers varied types of classes via System.IO to read and write data on data streams and files. They thus allow the transfer of data either to or from a storage medium.

We will see some of the important classes in this namespace.

Files and Directories:

Some of the very commonly used classes are as follows:

- File - provides static methods for creating, copying, deleting, moving, and opening files, and helps create a FileStream object.
- FileInfo - provides instance methods for creating, copying, deleting, moving, and opening files, and helps create a FileStream object.
- Directory - provides static methods for creating, moving, and enumerating through directories and subdirectories.
- DirectoryInfo - provides instance methods for creating, moving, and enumerating through directories and subdirectories.
- Path - provides methods and properties for processing directory strings in a cross-platform manner.

Streams:

Stream is nothing but a sequence of bytes. Stream in .net is an abstract base class. It could be a file, an input output device, an IPC, or a TCP/IP socket. The programmer is thus isolated from the specific details of operating system. Basic operations that can be performed on a stream are read, write and seek.

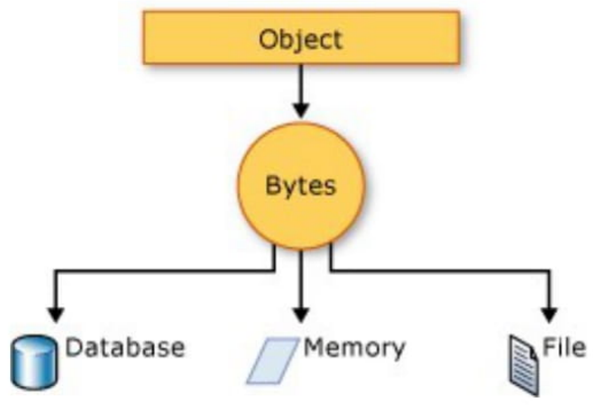
We will see more about *FileStream* class.

```
            string path = @"c:\temp\MyTest.txt";

                // Delete the file if it exists.
                if (File.Exists(path))
                {
                    File.Delete(path);
                }

                FileStream fs = new FileStream(path,
FileMode.OpenOrCreate);
                try
                {
```

```
                byte[] b = new byte[1024];
                string[] names = new string[5];
                names[0] = "This is new line..";
                names[1] = "This is some next..\n";
                names[2] = "This is some more next..\n";
                names[3] = "This may be last ..\n";
                names[4] = "This is last..\n";
                foreach (string str in names)
                {
                    byte[] info = new
UTF8Encoding(true).GetBytes(str);
                    fs.Write(info, 0, info.Length);

                }
            }
            catch
            {

            }
            finally
            {
                fs.Close();

            }
            //Open the stream and read it back.
            using (FileStream fsnew = File.OpenRead(path))
            {
                byte[] b = new byte[1024];
                UTF8Encoding temp = new UTF8Encoding(true);
                while (fsnew.Read(b, 0, b.Length) > 0)
                {
                    Console.WriteLine(temp.GetString(b));
                }
            }
```

Serialization

Serialization is the process of converting an object into a stream of bytes in order to store the object or transmit it to memory, a database, or a file. Its main purpose is to save the state of an object in order to be able to recreate it when needed. The reverse process is called de serialization.



The object is serialized to a stream, which carries not just the data, but information about the object's type, such as its version, culture, and assembly name. From that stream, it can be stored in a database, a file, or memory.

Uses for Serialization

Serialization allows the developer to save the state of an object and recreate it as needed, providing storage of objects as well as data exchange. Through serialization, a developer can perform actions like sending the object to a remote application by means of a Web Service, passing an object from one domain to another, passing an object through a firewall as an XML string, or maintaining security or user-specific information across applications.

How to make an object serializable?

In order to serialize an object, it should be marked as “Serializable”. If we do not want any particular field to be serialized, then we can mention: Nonserialized attribute.

Types of serialization:

There are basically 3 types of serialization:

1. Binary: Binary serialization uses binary encoding to produce compact serialization for uses such as storage or socket-based network streams.

2. XML: XML serialization serializes the public fields and properties of an object, or the parameters and return values of methods, into an XML stream that conforms to a specific XML Schema definition language (XSD) document.

3. SOAP Serialization: XML serialization can also be used to serialize objects into XML streams that conform to the SOAP specification. SOAP is a protocol based on XML, designed specifically to transport procedure calls using XML. As with regular XML

serialization, attributes can be used to control the literal-style SOAP messages generated by an XML Web service.

Now consider the following demonstration, in which an Employee class is marked as *Serializable*

```
[Serializable]
    class Emp
    {
        private int Eid;
        private string Ename;
        public int _Eid
        {
            get
            {
                return this.Eid;
            }
            set
            {
                this.Eid = value;
            }
        }
        public string _Ename
        {
            get
            {
                return this.Ename;
            }
            set
            {
                this.Ename = value;
            }
        }
    }
```

```
string path=@"D:\demo\data.bin";
            FileStream fs=new
FileStream(path,FileMode.OpenOrCreate,FileAccess.Write);
            BinaryFormatter bf = new BinaryFormatter(); //Binary
serialization
            Emp e = new Emp();
            e._Eid = 101;
            e._Ename = "Sharma";
            bf.Serialize(fs, e);     // Serialize the object in a file
            Console.WriteLine("Object serialized..");
            Console.ReadLine();
            fs.Close();
            Emp enew = null;
            fs=new FileStream(path,FileMode.Open,FileAccess.Read);
```

```
            enew = (Emp)bf.Deserialize(fs);   //Deserialize the object
            Console.WriteLine("Emp id: " + enew._Eid.ToString() + " Emp
name : " +    enew._Ename.ToString());
            Console.ReadLine();
```

Code snippet for SOAP and XML:

Summary:

In this part we have seen the basics of File I/O and a very important technique of preserving the states of object by serializing them.

Reflection & Attributes

Reflection in .Net framework:

In previous parts we have seen different types like classes, interfaces, value types, reference types etc. We are aware that all these types derive from namespace: System. Type. The assembly is a collection of all such types and their metadata in turn. The assembly loader uses this metadata while loading the assembly.

Assemblies contain modules, modules contain types, and types contain members. This is the hierarchy of assembly.

In .Net, the developer can also detect and dynamically create the instance of a type, bind the type to an existing object or get the type from an existing object. The classes in the System.Reflection namespace , together with System.Type enable you to obtain information about loaded assemblies and the types defined within them. Reflection can also be used to type instances at run time, and to invoke and access them.

Following are some of the classes in System.Reflection namespace which will be used frequently:

- Use Assembly to define and load assemblies, load modules that are listed in the assembly manifest, and locate a type from this assembly and create an instance of it.
- Use Module to discover information such as the assembly that contains the module and the classes in the module. You can also get all global methods or other specific, nonglobal methods defined on the module.
- Use ConstructorInfo to discover information such as the name, parameters, access modifiers (such as public or private), and implementation details (such as abstract or virtual) of a constructor. Use the GetConstructors or GetConstructor method of a Type to invoke a specific constructor.
- Use MethodInfo to discover information such as the name, return type, parameters, access modifiers (such as public or private), and implementation details (such as abstract or virtual) of a method. Use the GetMethods or GetMethod method of a Type to invoke a specific method.
- Use FieldInfo to discover information such as the name, access modifiers (such as public or private) and implementation details (such as static) of a field, and to get or set field values.
- Use EventInfo to discover information such as the name, event-handler data type, custom attributes, declaring type, and reflected type of an event, and to add or remove event handlers.
- Use PropertyInfo to discover information such as the name, data type, declaring type, reflected type, and read-only or writable status of a property, and to get or set property values.

- Use ParameterInfo to discover information such as a parameter's name, data type, whether a parameter is an input or output parameter, and the position of the parameter in a method signature.
- Use CustomAttributeData to discover information about custom attributes when you are working in the reflection-only context of an application domain. CustomAttributeData allows you to examine attributes without creating instances of them.

Consider following example:

```
using System;
using System.Reflection;

public class MathClass
{
   public virtual int AddNumb(int numb1,int numb2)
   {
     int result = numb1 + numb2;
     return result;
   }

}

class MyMainClass
{
  public static int Main()
  {
    Console.WriteLine ("\nReflection.MethodInfo");
    // Create MathClass object
    MathClass myClassObj = new MathClass();
    // Get the Type information.
    Type myTypeObj = myClassObj.GetType();
    // Get Method Information.
    MethodInfo myMethodInfo = myTypeObj.GetMethod("AddNumb");
    object[] mParam = new object[] {5, 10};
    // Get and display the Invoke method.
    Console.Write("\nFirst method - " + myTypeObj.FullName + "
returns " +
                              myMethodInfo.Invoke(myClassObj,
mParam) + "\n");
    return 0;
  }
}
```

The type myTypeObj will now have all the information about MathClass. We can thus check if the type is a class, if it is a class, then is it abstract or a regular class.

The `myMethodInfo` will have all information about the method: " AddNumb".

Dynamically creating an instance of a class can be done in following manner:

```
Type type1 = typeof(MyClass2);
object obj = Activator.CreateInstance(type1);
```

A method can be invoked dynamically in following manner:

```
myMethodInfo.Invoke(myClassObj, mParam);
```

Following code can be also used to invoke members:

```
using System;

using System.Reflection;

namespace Reflection

{

 class Class1

 {

  public int AddNumb(int numb1, int numb2)

  {

   int ans = numb1 + numb2;

   return ans;

  }


 [STAThread]

 static void Main(string[] args)

 {

   Type type1 = typeof(Class1);

   //Create an instance of the type

   object obj = Activator.CreateInstance(type1);

   object[] mParam = new object[] {5, 10};
```

```
    //invoke AddMethod, passing in two parameters
    int res = (int)type1.InvokeMember("AddNumb", BindingFlags.InvokeMethod,
                                null, obj, mParam);
    Console.Write("Result: {0} \n", res);
  }
 }
}
```

Custom Attributes:

Attributes are characteristics or qualities that tell us more about any entity. This entity could be a class, a property, a structure etc. This is also a part of metadata. We can use the technique of reflection to access the attributes at runtime. We will come to know about this in following discussion:

Let us first see how to create custom attributes:

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

namespace CustomAttributesdemo
{
    class Class1
    {
    }

    [System.AttributeUsage(System.AttributeTargets.Class |
                           System.AttributeTargets.Struct)]
    public class Author : System.Attribute
    {
        private string name;
        public double version;

        public Author(string name)
        {
            this.name = name;
            version = 1.0;
        }
}
[Author ("XYZ",version=1.0)]
public class Employee
{
```

```
    }
}
```

In above example, the class Author inherits from System.Attribute class and hence it can in turn behave as an attribute. The target here is a class or structure. When we have applied the attribute on a class, we are calling the constructor, where version is a named parameter.

The Custom attributes are created and applied in the above manner. But they will be of no use, if the programmer cannot access them. We can do this by using reflection technique.

"GetCustomAttributes" is the key method which can be used to dynamically access the attributes. Consider following code for the same:

```
// Multiuse attribute.
[System.AttributeUsage(System.AttributeTargets.Class |
                       System.AttributeTargets.Struct,
                       AllowMultiple = true)  // Multiuse attribute.
]
public class Author : System.Attribute
{
    string name;
    public double version;

    public Author(string name)
    {
        this.name = name;

        // Default value.
        version = 1.0;
    }

    public string GetName()
    {
        return name;
    }
}

// Class with the Author attribute.
[Author("XYZ")]
public class EmpClass
{
    // ...
}

// Class without the Author attribute.
public class SecondClass
{
    // ...
}

// Class with multiple Author attributes.
```

```
[Author("ABC"), Author("XYZ", version = 2.0)]
public class ThirdClass
{
    // ...
}

class TestAuthorAttribute
{
    static void Test()
    {
        PrintAuthorInfo(typeof(EmpClass));
        PrintAuthorInfo(typeof(SecondClass));
        PrintAuthorInfo(typeof(ThirdClass));
    }

    private static void PrintAuthorInfo(System.Type t)
    {
        System.Console.WriteLine("Author information for {0}", t);

        // Using reflection.
        System.Attribute[] attrs =
System.Attribute.GetCustomAttributes(t);  // Reflection.

        // Displaying output.
        foreach (System.Attribute attr in attrs)
        {
            if (attr is Author)
            {
                Author a = (Author)attr;
                System.Console.WriteLine("   {0}, version {1:f}",
a.GetName(), a.version);
            }
        }
    }
}
```

There are following types of attributes in C#:

- Global Attributes
- Obsolete Attributes
- Conditional Attributes

A very good example of attribute usage can be Entity framework. Entity framework is ORM framework from Microsoft. Just like any other ORM framework; it can help developer generate the entity class or Data Transfer Object (DTO) through tables. However, one of the unique functionality of Entity Framework is its Model First approach!

One can write a class first and then generate table automatically. Wow! How is that?
Consider below employee class code:

In order to get [Table] and [Column] attributes in program; import namespace "System.ComponentModel.DataAnnotations.Schema"

```
[Table("Employee")]
   public class Emp
   {
      [Column]
      public int No { get; set; }
      [Column]
      public string Name { get; set; }
      [Column]
      public string Address { get; set; }
  }
```

Entity Framework reads these attributes using reflection and generates SQL query like below:

"CREATE TABLE Employee (No int, Name varchar(50), Addresse varchar(50))"

This way, even CLR, compiler, certain frameworks like entity, PRISM can help generate code or take custom actions.

Memory Mapped Files

We have seen in the File I/O part, the different classes like Binary Readers and Writers, Stream Readers and Writers, String and Text Readers and Writers are used for permanent storage of data in files. These files are stored on hard drives. These files which are created are actual physical files. Each time a file handle is associated with the file to read, write or seek the file. A process thread is associated with the file operations. Unless and until our process thread leaves the control of that file, nobody else can access it. When the current thread is involved in the very long file I/O operations, then there may arise a deadlock situation. The UI thread will be blocked and it will appear as if the application is doing nothing. In this situation, if any other applications thread needs to access the same file, then the other application cannot, unless the file is released. This will in turn affect the performances.

To share the resources for e.g. very large files on same machine and thus to enable IPC (Inter process communication), a new methodology of: Memory mapped files was introduced in .net f/w 4.0.

A memory mapped file contains the contents of a file in virtual memory. So there can be multiple views created of the same file and can thus be accessed by multiple processes. This feature thus makes use of managed code to access the memory mapped files.

There are two types of memory mapped files:

1. Persisted memory mapped files: These files are necessarily associated with a source file on a disk. Multiple processes can access the file and data is saved to the source file when last process has finished working with it. This is generally used when large files are to be accessed.
2. Non persisted memory mapped files: These files are not associated with any source file on the disk. When last process finishes, the data is lost and the file is reclaimed by the GC (Garbage collector). These are suitable for creating shared memories for IPC.

The memory manager of the OS is used or helps in accessing the files; hence the concept of paging is used in accessing memory, which is very fast.

Following are some of the major advantages of memory mapped files:

1. Provide unique ways for mapping memory in the windows application programming interface. It permits an app to map its virtual address space to a file on disk.
2. The data is stored in RAM and accessed via paging.
3. No disk access is required or involved hence the technique is faster.
4. Smaller read and write operations are chached.

Consider following code snippet:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using System.IO.MemoryMappedFiles;
using System.Runtime.InteropServices;

namespace Memorymappeddemo2
{
    public struct MyColor
    {
        public short Red;
        public short Green;
        public short Blue;
        public short Alpha;

        // Make the view brigher.
        public void Brighten(short value)
        {
            Red = (short)Math.Min(short.MaxValue, (int)Red + value);
            Green = (short)Math.Min(short.MaxValue, (int)Green +
value);
            Blue = (short)Math.Min(short.MaxValue, (int)Blue +
value);
            Alpha = (short)Math.Min(short.MaxValue, (int)Alpha +
value);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            // Assumes another process has created the memory-mapped
file.
            using (var mmf = MemoryMappedFile.OpenExisting("ImgA"))
            {
                using (var accessor =
mmf.CreateViewAccessor(4000000, 2000000))
                {
                    int colorSize = Marshal.SizeOf(typeof(MyColor));
                    MyColor color;

                    // Make changes to the view.
```

```
                        for (long i = 0; i < 1500000; i += colorSize)
                        {
                            accessor.Read(i, out color);
                            color.Brighten(20);
                            accessor.Write(i, ref color);
                        }
                    }
                }
            }
        }
    }
```

Summary:

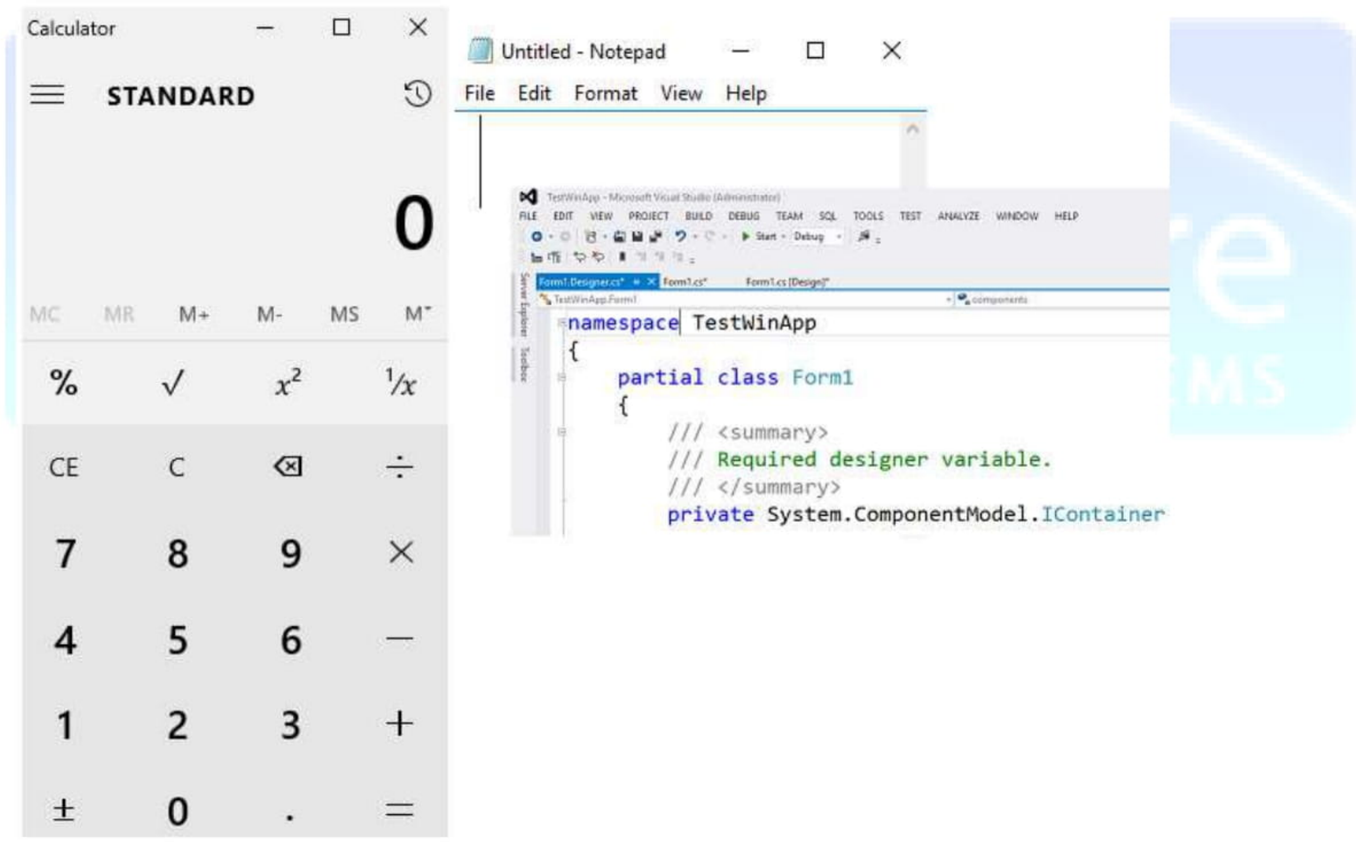
Memory mapped files thus provide the developers a new methodology for IPC. It is much faster since it makes use of the cache memory and paging.

Introduction to WinForm(s)

So far, we have seen C# language features using console based applications. Creating UI based applications is very easy in C# .NET. In this part we shall see an introduction to the same. We will not be diving into each control or UI element available in Visual Studio.

There are 1000s of examples of windows based applications! MS OFFICE, Visual Studio, Calculator, Media Player, Browser are frequently used application on your machine. These all are desktop based windows form application(s). Some of them are developed using .NET. For example Visual Studio 2010 inwards every Visual Studio is created using .NET itself!

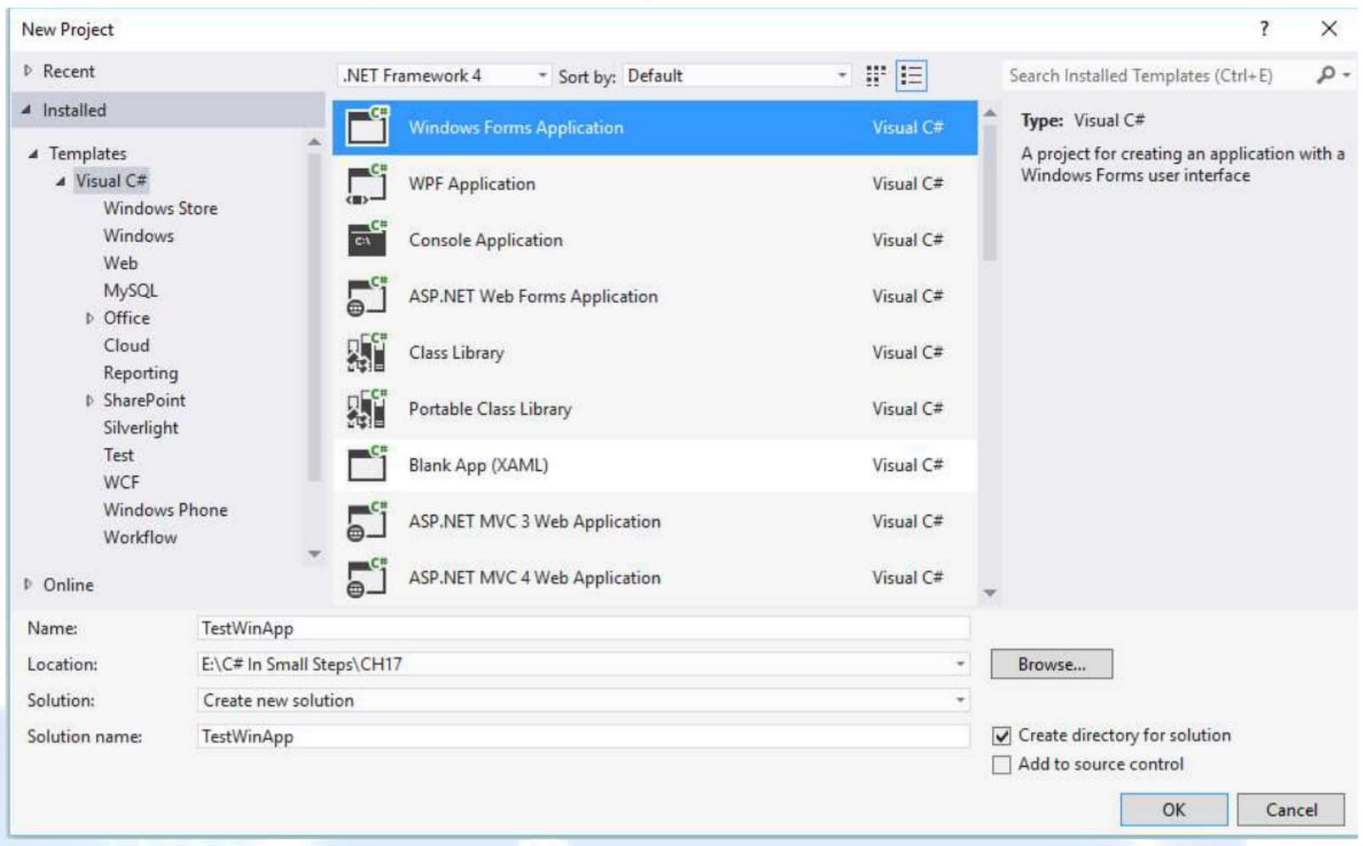
Every well known desktop based application has a commonly seen behavior. E.g. Most of the applications have Minimize, Maximize, Close buttons on the right top corners.



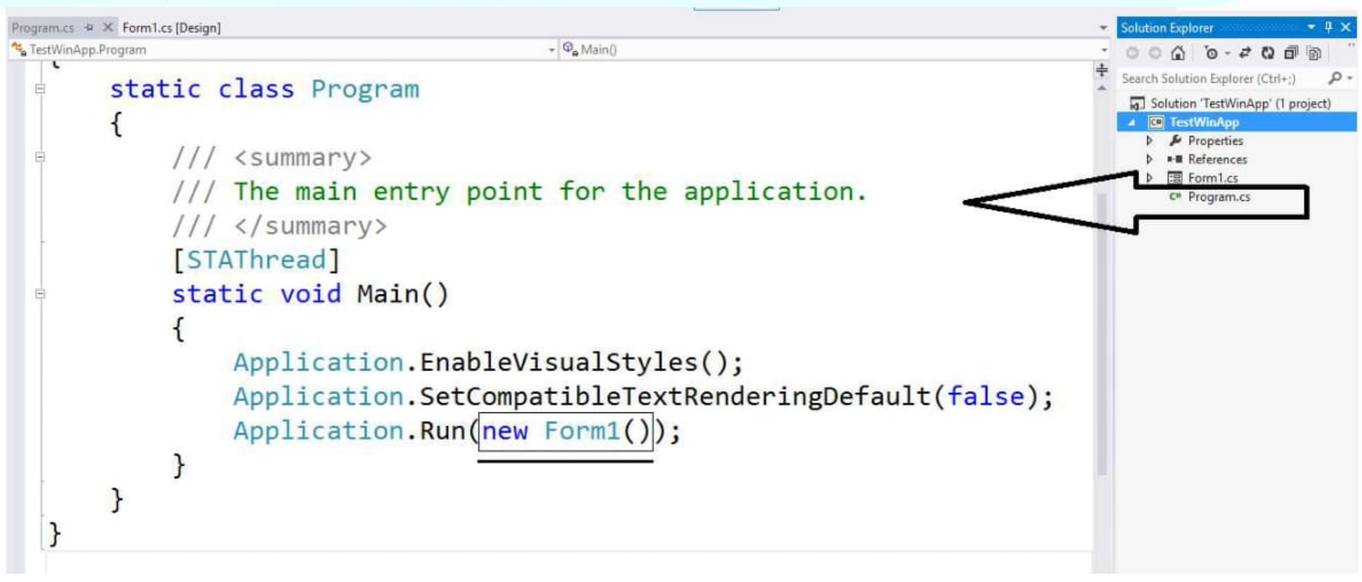
This common look and feel along with behavior is already present in the class defined in .NET called as FORM. This class can be referred from namespace “SYSTEM.WINDOW.FORMS”

Windows forms applications output is an executable file. Let us quickly start visual studio and create a simple “hello world” like application to understand the obvious steps:

1. Start visual studio.
2. Click File->New Project
3. Select “Windows Forms Application” from the options as shown in below snapshot:

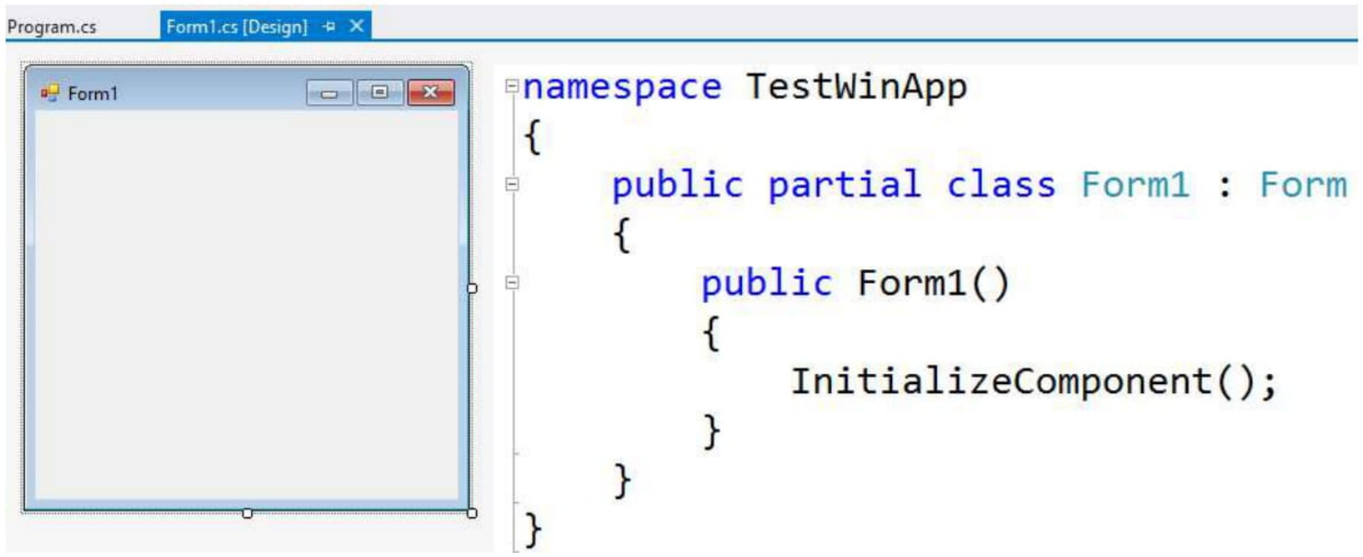


4. Change the name of the project as shown above to “TestWinApp” . Make sure you specify the path for storing the project.
5. Click “Ok”
6. A project default structure shall open in Visual Studio. On the right hand side, you shall find “Solution Explorer” in which find Program.CS file. Double click the file and open it.

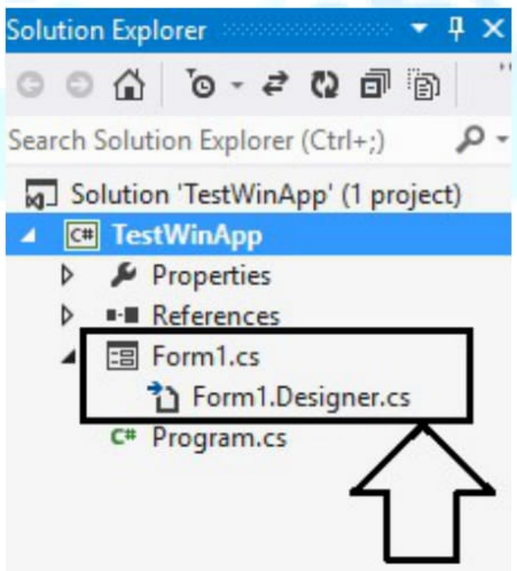


7. Notice, as the program starts with “MAIN” method, there is a call to “FORM1” class object. This Form1 is a class with default code. It inherits from the base class called “FORM” which in turn is defined in SYSTEM.WINDOWS.FORM.

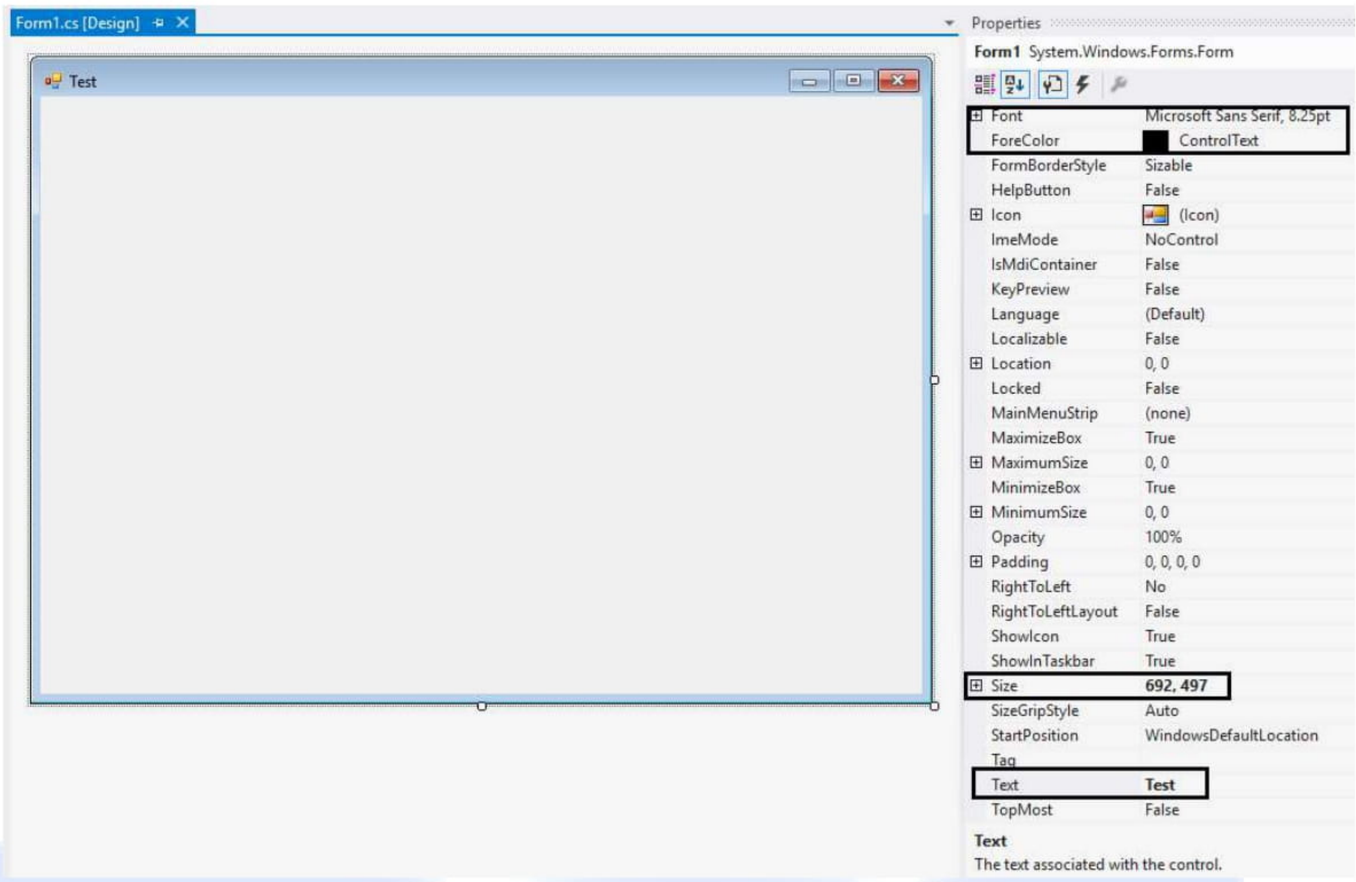
8. This base class consist of visual functionality like default color, Min, Max buttons as discussed earlier.
9. You may notice visual design and source code for the Form1 is available for modification



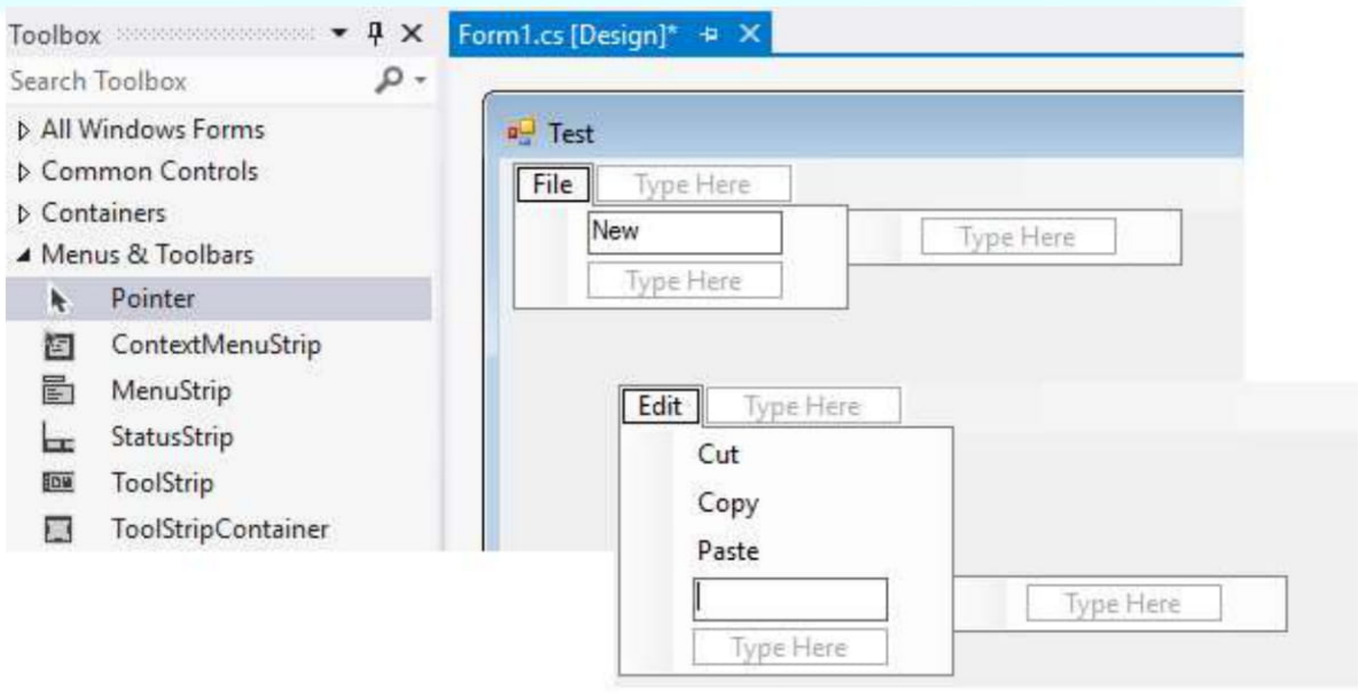
10. As discussed in C# features earlier, to separate UI design related code from the logic; we use Partial class concept. In this application; UI design code and developer written source is separated using partial class:
Source code is normally in: FORM1.cs (FORM1 being default name for new form)
Design code is normally in: FORM1.DESIGNER.CS file



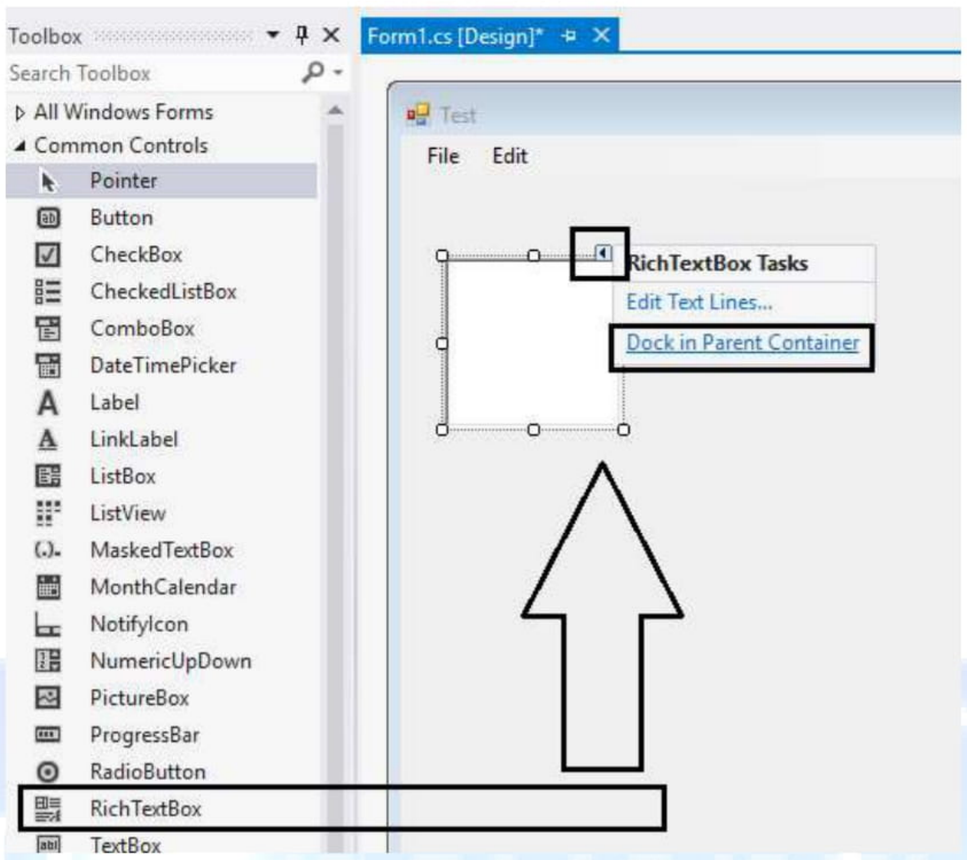
11. Double click & open “Form1.cs” from Solution Explorer. Design surface should open.
12. Right click on the empty surface & select properties. Optionally one may use “F4” key to achieve the same.
13. In the properties window; one can set values for CLASS properties. These properties for FORM1 class are inherited with default values from the base class.
14. Change the “title” of the form. Optionally you may change other values like height, width, size, font etc. See the step in snapshot below:



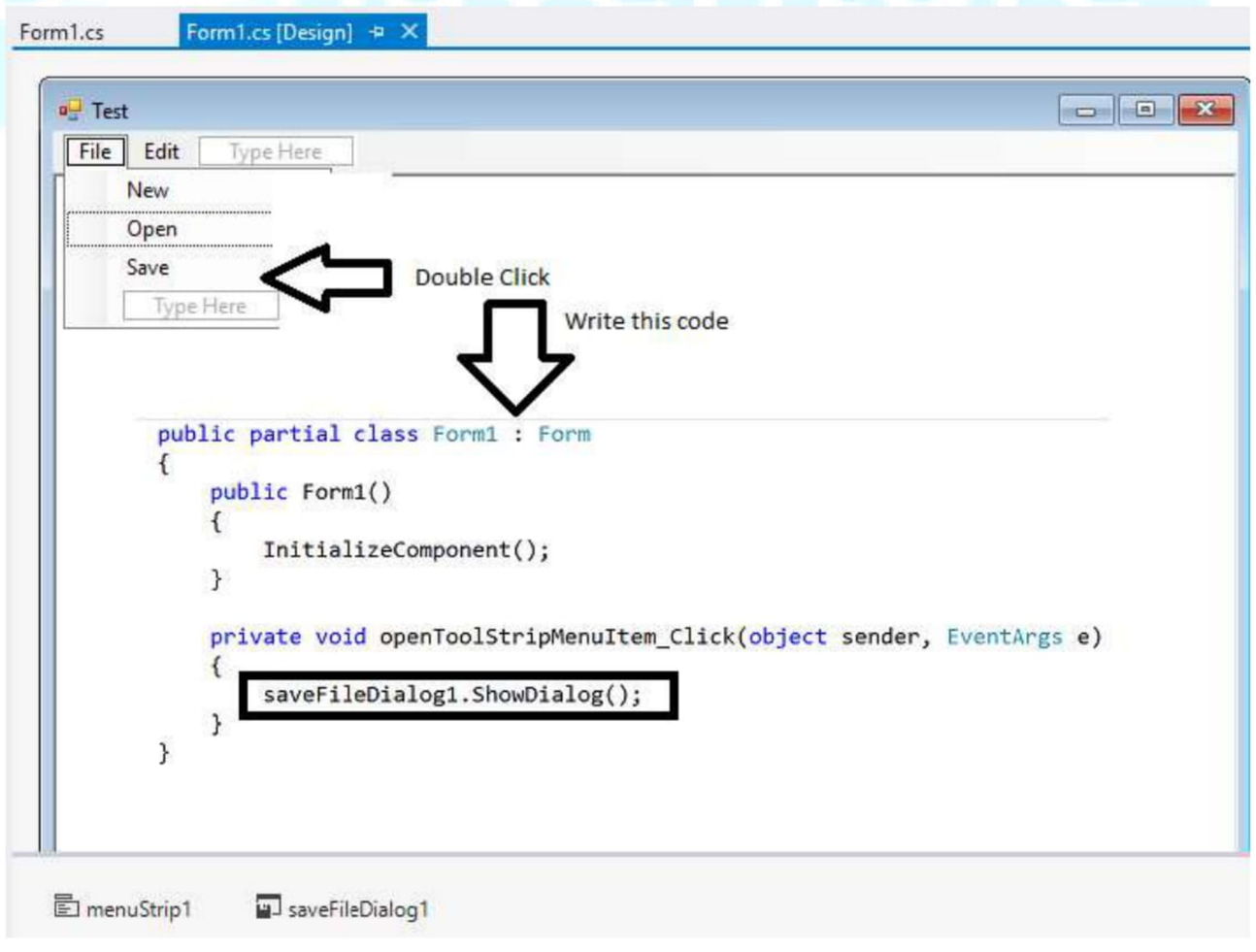
15. Let us use few of the controls. Make sure you FORM1's design surface is open.
16. Click "View->ToolBox"
17. From the toolbox, drag & drop a component "MenuStrip" from the Menus & Toolbars section.
18. Make the changes in the menu display as shown in below snapshot:



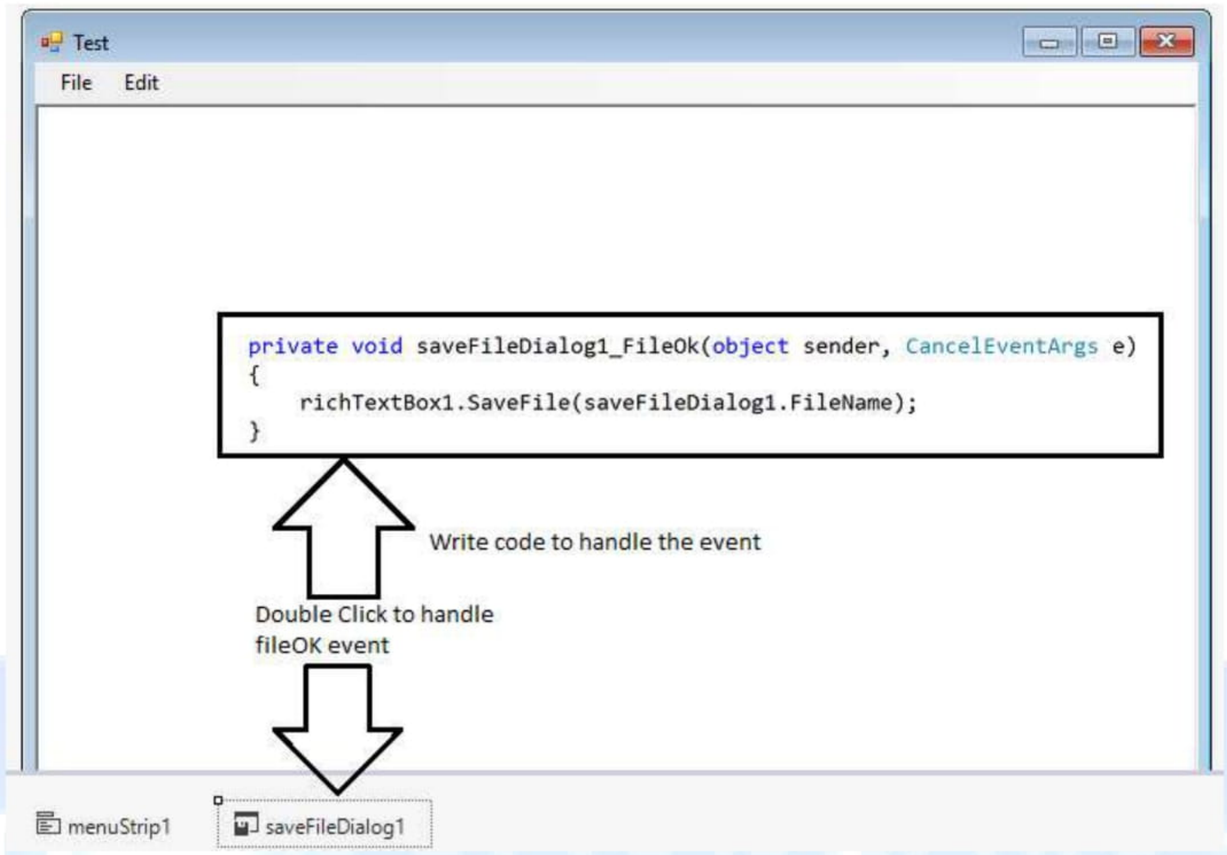
19. Now, drag and drop “RichTextBox” control on the “FORM1” design surface from “Common Controls” section
20. Click on “smart tag” as shown in snapshot below and click -> “Dock in parent container”.



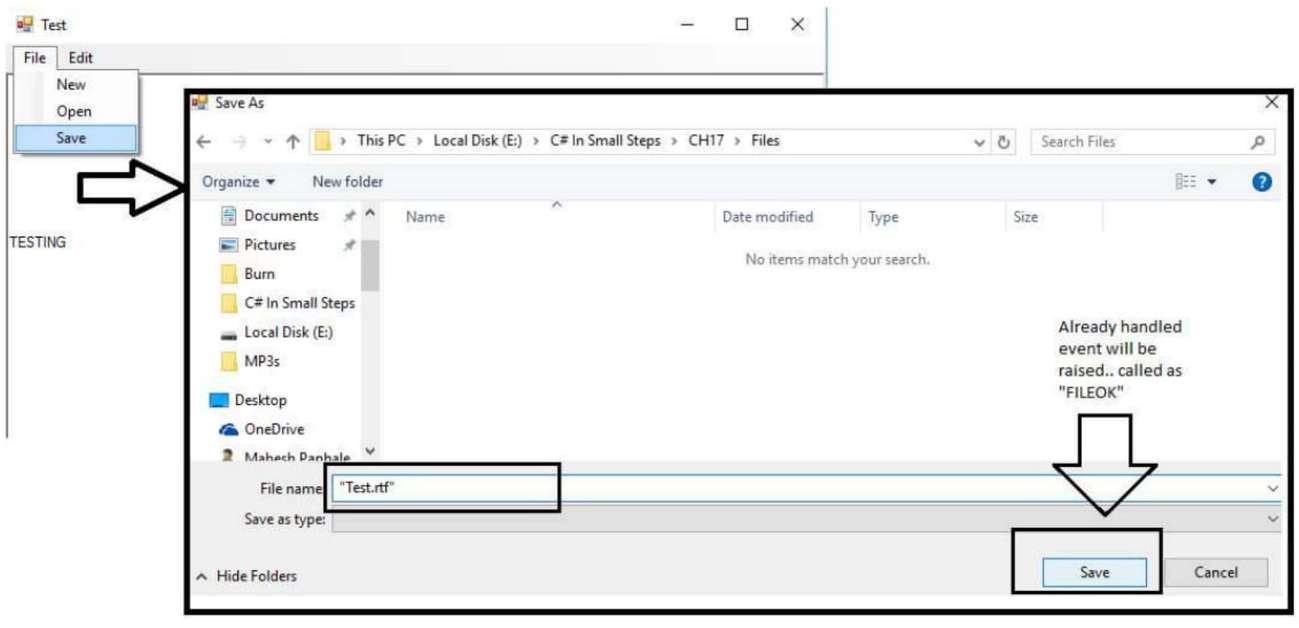
21. This step shall expand the “RichTextBox” control and occupies entire FORM1 area!
22. Now, click -> “Save” menu item and write mentioned code in snapshot.



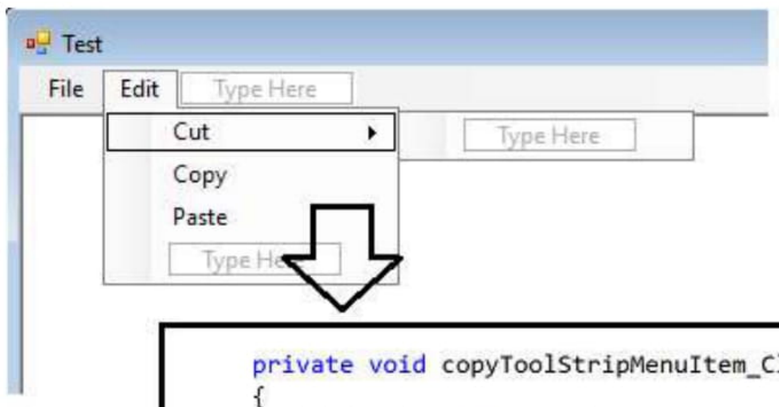
23. Now, from the component bar at the bottom of “FORM1” design area. Double click on “saveFileDialog1” control to handle the event called as “FILEOK”
24. Write the code for this event “FILEOK” as shown in below snapshot:



25. Now, hit “F5” key on keyboard to run the application.
26. After typing “TEST”data into the rich text box; click -> “Save” button.
27. This action should bring SaveDialog Box in front.
28. Write the file name with which you would like to store the data. Since it’s a rich text box; file extension like “RTF” is always recommended. Refer below snapshot:



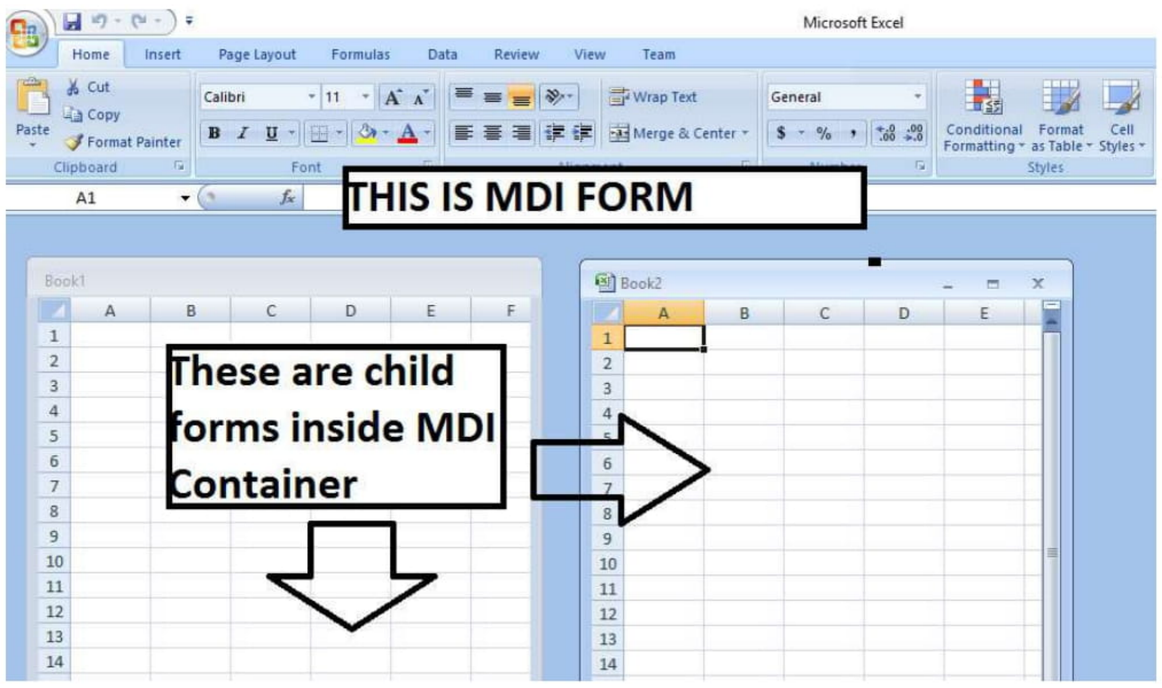
29. To handle “Cut”, “Copy”, or “Paste” like typical commands; one need not study “CLIP BOARD”. Rich text box control, directly offers methods to do such things. Write down the code to handle respective button’s “CLICK ” event:




```
private void copyToolStripMenuItem_Click(object sender, EventArgs e)
{
    richTextBox1.Copy();
}

private void pasteToolStripMenuItem_Click(object sender, EventArgs e)
{
    richTextBox1.Paste();
}

private void cutToolStripMenuItem_Click(object sender, EventArgs e)
{
    richTextBox1.Cut();
}
```

30. Similar way many controls can be tested and used in the application.

One can keep on creating objects of FORM classes and show respective forms. Some applications like MS Excel have forms displayed within another Form.



Outer form acts more like a container windows and inner forms behave like content. If the container window gets closed, then every content window closes. However reverse may not have same impact. Closing inner window form may not close the outer one.

Such forms can be added into the project by referring to new MDI form.

You can explore 100s of controls available with default .NET and Visual studio installation. Some of the popular most controls are:

- TextBox
- Button
- Label
- Tooltip
- Datagrid
- ErrorProvider
- ListView
- Various dialog boxes
- Treeview

Most of the control's details and examples are available on MSDN in well documented manner. You can refer below link to explore more on these controls:

Link:

https://msdn.microsoft.com/en-us/library/system.windows.forms.control(v=vs.110).aspx

Summary:

In this part, we discussed about

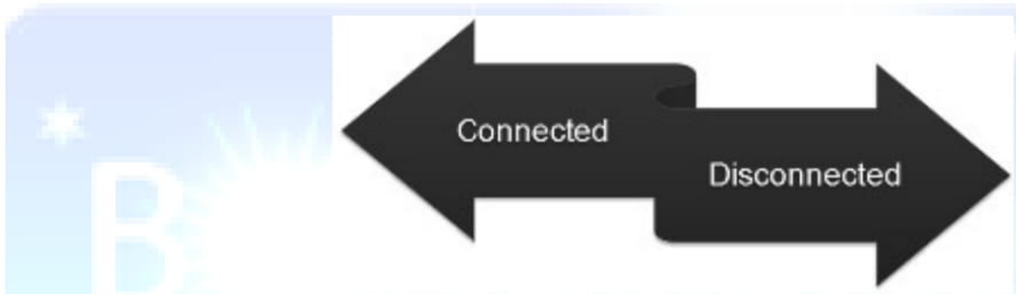
- What windows form applications are
- Meaning of MDI application
- Few controls examples

ADO .NET Connected Architecture

No enterprise application is complete without some or other database. When it comes to working with database, Microsoft .NET has a preference to SQL Server. However that does not stop any code from integrating itself with Oracle or other databases.

Microsoft offers numbers of database related classes clubbed into namespaces and in turn into libraries. Most of the database functionality is given in the framework base class library called *System.Data.dll* . This library is automatically installed on the machine with .NET framework.

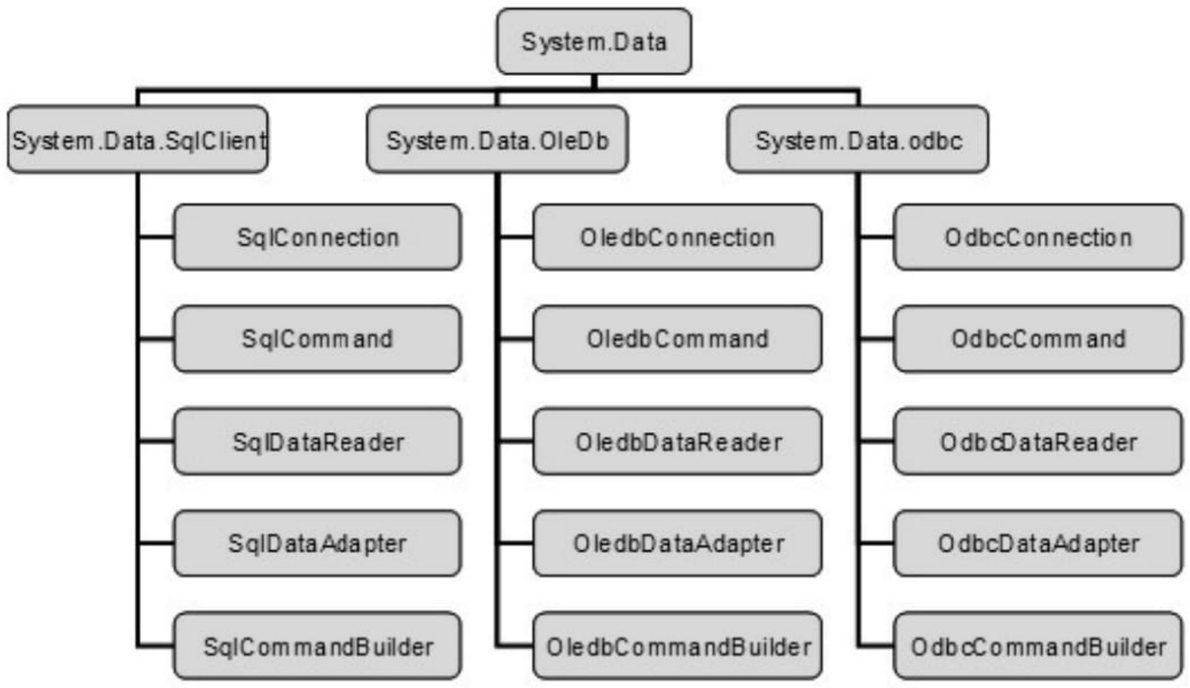
Microsoft .NET has two approaches of working with any database:



We shall focus on connected architecture only in this part.
Let us try to understand basic steps & prerequisite to work with database. We are going to consider SQL Server database only in this part.

1. First of all, we should have database with table(s).
2. One also needs to know the credential(s) to be used with database
3. Code normally starts by telling details about - how to connect with the database - to a .NET based program
4. These details are offered to .NET program using Connection type object. Actual type name(s) vary based on database
5. Once connection is established between .NET program and database; program needs to fire the SQL query using command class object. Actual class name vary based on database
6. Result of the query can be collected using data reader type object
7. In connected architecture connection to the database needs to opened and closed through code explicitly
8. However, in disconnected architecture; this job on opening and closing connection is outsourced to one more type of object called as adapter. Here as well, actual class name vary based on database
9. A connection to the dataset must remain open while program reads the data from the server.

Below figure shows important class hierarchy in ADO.NET:



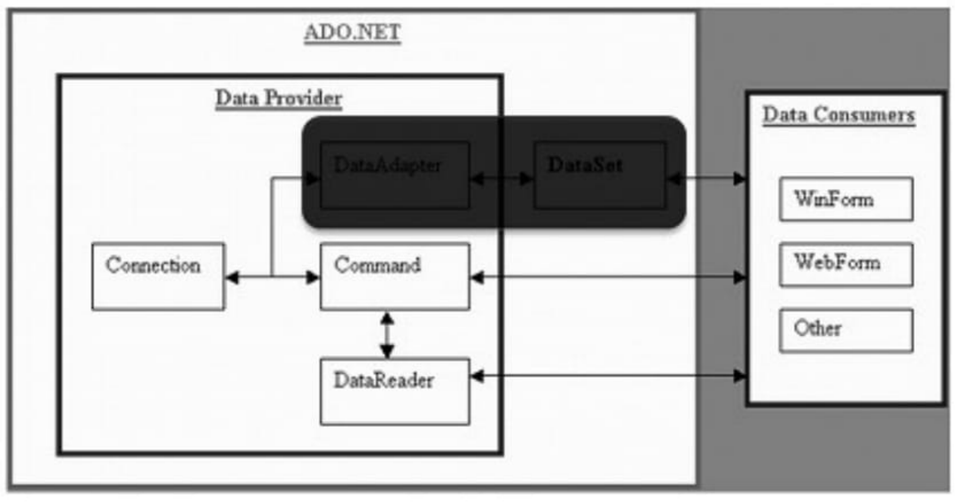
As we can see, SQL Server related classes are grouped under the namespace (group of classes) System.Data.SQLClient.

OleDB and ODBC use mediators a.k.a. drivers based on standard with the same name.

However we can use these objects as well to connect to SQL Server. Using SQL Server specific classes let developer access SQL Server related special features into .NET code. Hence, using SQL Client namespace is important while communicating with SQL Server.

For example, SQL Server offers a functionality to return result of the query directly in the form of XML. One can use a very special method (ReadXML) given in SQLCommand class to read that XML. Other classes like ODBCComand, OleDBCommand do not provide such features.

Below figure conveys the way communication happens between database and client application written using .NET:



Let us discuss each of the elements in detail:

Data Consumer:
These are client application which demands data manipulation. Either the ADO.NET code us written inside consumer project or as a service.

Connection object:
While coding with SQL Server this object is actually SqlConnection type. This object takes credentials and database details as an input. While working with SQL Server, one will have to provide minimum details to connection object as:

Connection String Parameter Name	**Description**
Data Source	Identifies the server. Could be local machine, machine domain name, or IP Address.
Initial Catalog	Data base name.
Integrated Security	Set to SSPI to make connection with user's Windows login
User ID	Name of user configured in SQL Server.
Password	Password for SQL Server User ID.

One can define the SQL Connection object then as:

```
SqlConnection cn = new SqlConnection ("Data Source=DatabaseServer;Initial
Catalog=DatabaseName;User ID=YourUserID;Password=YourPassword")
);
```

Command object:
While coding with SQL Server this object is actually SqlCommand type. This object takes SQL query and SqlConnection object as input. While working with SQL Server, one will have to provide minimum below details to SqlCommand class object like:

SqlCommand myCommand = new SqlCommand("select * from Emp", cn);

In the above code snippet; - "Select * from Emp" – is a SQL query and 'cn' refers to a SQLConnection object

Datareader object:
While coding with SQL Server this object is actually SqlDataReader type. This object gets the reference to data from SQLCommand object. We will have to use "Read()" method of SqlDataReader to get the data via a sequential stream. To read this data, we must pull data from a table row-by-row. Once a row has been read, the previous row is no longer available. SqlDataReader is also referred as - read only – forward only - cursor.

SqlDataReader myDataReader = myCommand.ExecuteReader();

Kindly note, SqlDataReader is dependent on SqlCommand object.
As shown below; In order to read the data from the database server, one will have to use Read() method:

```
while (myDataReader.Read())
      {
         //Read the data one row at a time - considering
         //each time mydataReader is a collection of column values
      }
```

So, if we connect all the DOTs (.) then this is how code entire code from console application looks like:
(Considering Emp table from the database has ENo, EName and EAddress as columns)

```
class Program
{
static void Main(string[] args)
{
   SqlConnection cn = new SqlConnection();
   cn.ConnectionString =
      @"Data Source=.\SQLEXPRESS;Integrated Security=True;
      Database=Test";
    cn.Open();
    string strSQL = "Select * From Emp";
    SqlCommand myCommand = new SqlCommand(strSQL, cn);
    SqlDataReader myDataReader =
      myCommand.ExecuteReader(CommandBehavior.CloseConnection);
```

```
    while (myDataReader.Read())
      {
      Console.WriteLine(“ENo: {0}, EName: {1}, EAddress: {2}.",
        myDataReader[“ENo"].ToString(),
        myDataReader[“EName"].ToString(),
        myDataReader[“EAddress"].ToString());
}
Console.ReadLine();
}
}
```

Above code explains “Select” query on SQL Server database.
When one needs to fire Insert, Update or Delete queries on SQL Server database; code changes.

DML queries like Insert. Update, Delete do return number of rows affected and not the data. So, in such queries DataReader is not expected to collect anything. So a different method from SqlCommand class is used.

We use “ExecuteNonQuery()” method of SqlCommand object to get the number of rows affected then.

Below code snippet shows Insert query syntax using C# console application:

```
Class Program
{
 static void Main(string[] args)
 {
   SqlConnection cn = new SqlConnection();
   cn.ConnectionString =
      @"Data Source=.\SQLEXPRESS;Integrated Security=True;
      Database=Test";
    cn.Open();
    string UnformattedstrSQL = “Insert into Emp values({0},‘{1}’,‘{2}’)";
    string strSQL = string.Format(UnformattedstrSQL, 5,’Sachin’, ‘Pune’);
    SqlCommand myCommand = new SqlCommand(strSQL, cn);
    int noOfRowsAffected =
      myCommand. ExecuteNonQuery();

    Console.WriteLine(“No of rows affected: {0}", noOfRowsAffected };
    Console.ReadLine();
 }
}
```

Similar way, one can fire Update and Delete queries.

For updating the name of the Employee based on the ID or No parameter use below code:

```
Class Program
{
 static void Main(string[] args)
 {
   SqlConnection cn = new SqlConnection();
   cn.ConnectionString =
      @"Data Source=.\SQLEXPRESS;Integrated Security=True;
      Database=Test";
    cn.Open();
   string UnformattedstrSQL=“Update Emp set Ename =‘{0}’ where Eno={1})";
   string strSQL = string.Format(UnformattedstrSQL, ChangedMugdha,5);
   SqlCommand myCommand = new SqlCommand(strSQL, cn);
   int noOfRowsAffected =
     myCommand.ExecuteNonQuery(CommandBehavior.CloseConnection);

   Console.WriteLine(“No of rows affected: {0}", noOfRowsAffected };
   Console.ReadLine();
 }
}
```

For deleting a record from database based on the primary key; in this case ‘ENo’ refer below code:

```
Class Program
{
 static void Main(string[] args)
 {
   SqlConnection cn = new SqlConnection();
   cn.ConnectionString =
      @"Data Source=.\SQLEXPRESS;Integrated Security=True;
      Database=Test";
    cn.Open();
   string UnformattedstrSQL=“Delete from Emp where Eno={0})";
   string strSQL = string.Format(UnformattedstrSQL, 5);
   SqlCommand myCommand = new SqlCommand(strSQL, cn);
   int noOfRowsAffected =
     myCommand.ExecuteNonQuery(CommandBehavior.CloseConnection);

   Console.WriteLine(“No of rows affected: {0}", noOfRowsAffected };
   Console.ReadLine();
 }
}
```

We shall focus on disconnected architecture in the next part.

Summary:

We have seen basics of ADO.NET connected architecture in this part from the console application perspective. We discussed:

- Basics of ADO.NET
- Quick difference in architecture
- Object hierarchy in ADO.NET
- How to fire Select, Insert, Update, Delete queries on SQL Server

ADO .NET Disconnected Architecture

We discussed about connected architecture in previous part. Basic difference between connected and disconnected is in responsibility of opening & closing the connection with database. In connected developer has to open & close the connection. Where in disconnected this responsibility is outsourced to an object called data adapter.

Some objects used in the code remain same between ADO.NET connected & disconnected architecture such as (from SQL Server perspective):

- SQLConnection
- SQLCommand

SQLConnection is required to convey data adapter object - which database to connect to. Data adapter internally builds and uses SQLCommand object. Developer need not create or provide the same.

May be the basic question one may have why do we need disconnected architecture as connected architecture is available all over the technologies. Java, PHP like frameworks have connected architecture similar to .NET. However disconnected is unique to .NET!

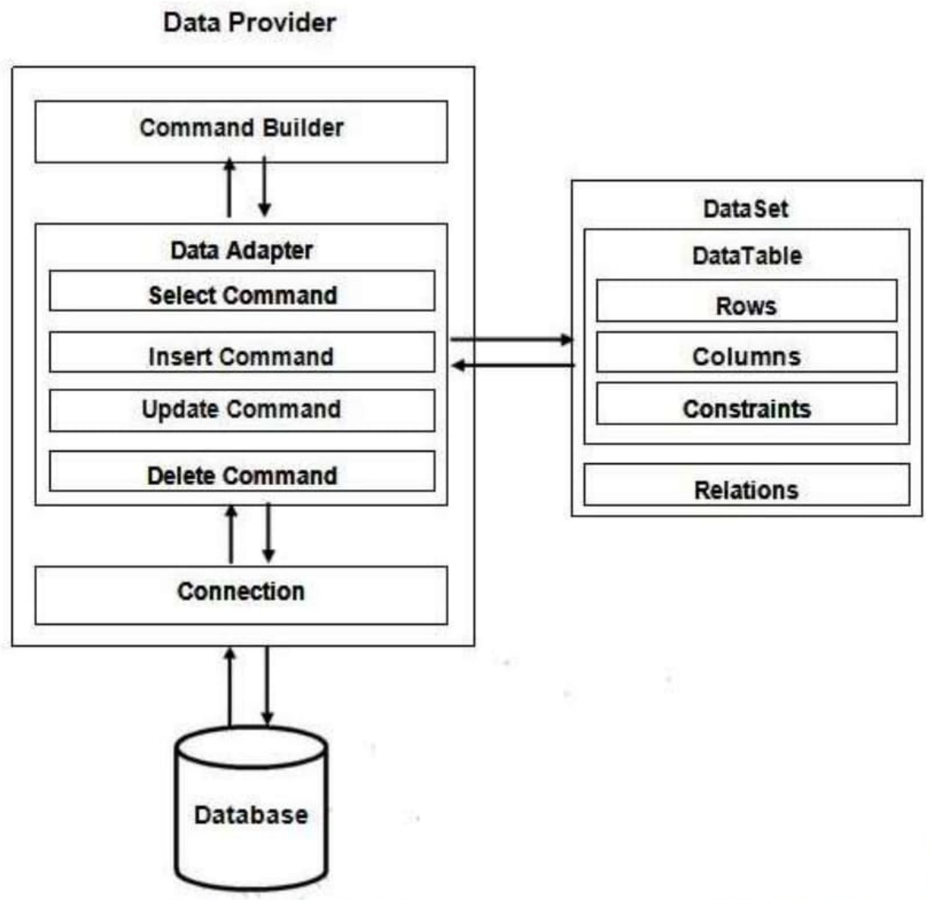
Why one needs disconnected (Sample Scenarios):

- In case of long running application(s); most of the time when data is static; instead of getting same data many number of times; we prefer to fetch & store the data using disconnected architecture. We use dataset to store the data into memory
- When one needs to fetch data chunks from multiple databases and for analysis purpose may want to establish some or other relation within the data chunks; we can use disconnected architecture

One should not use disconnected architecture if data changes very frequently.

What happens behind the scene when one uses disconnected architecture in .NET application?

Let us start with basic figure below:



1. First a SQL Connection object is required. This connection need not be explicitly opened. It is opened automatically by data adapter.
2. Data adapter closes the same immediately when the work is done.
3. Data adapter needs command object which can be used to fire “Select” like query on database
4. Most of the time this select query is passed as a parameter to data adapter object
5. Data adapter can use its FILL() function to open the connection and fire the SELECT query
6. However data adapter FILL() method expects data to be collected in the special object
7. This special object is called as DataSet – which is a collection of datatable class object
8. Datatable object presents actual data in the tabular format in memory
9. Once the data is filled into the dataset / data table; connection gets closed automatically.
10. Most of the time - data inside the dataset – is used for display or analysis purpose.
11. If anybody wants to do Insert / Update / Delete operation(s) then one will have to work with dataset – table as if it’s a collection
12. However, changes w.r.t. will be done inside data set; in order to update the same with database; one will have to use data adapter again.
13. In update case; data adapter does not have any query. All the required DML queries then will be generated by one more helper object called as CommandBuilder. With SQL Server database, it is called as SQLCommandBuilder
14. While working with updates on the client side, one may need to search a record inside the dataset – table. By default data table object inside dataset - do not have primary key like database table. To make the search easy, one can optionally ask data adapter to bring primary key information from database server side to client side (inside dataset – table). This can be conveyed to data adapter before calling the FILL() method.

15. One will have to set data adapter ‘s missingschmaaction property
16. Dataset – table can be optionally stored on HDD in the form of XML
17. Data adapter fires one query at a time on database - while updating changes inside data set to database table.
18. If any query fails; then rest of the queries are put on hold
19. If one wants data adapter to continue executing the rest of the statements; then data adapter can be optionally conveyed about the same by setting “CONTINUEUPDATEONERROR” property to “TRUE”
20. Every row data inside dataset table has status which is used by data adapter to understand what kind of query it needs to get from command builder.

Below are the code snippets to perform SELECT, INSERT, UPDATE DELETE queries using disconnected architecture:

```
//Connection to Database

SqlConnection con = new SqlConnection("Data Source=SQLEXPRESS;Initial
Catalog=SOMEDBNAME;Integrated Security=True ");

//Data Adapter

SqlDataAdapter  da = new SqlDataAdapter("select * from Emp",con);

//A friend of data adapter which can generate required command objects

SqlCommandBuilder cmb = new SqlCommandBuilder(da);

//Empty dataset declaration

DataSet ds = new DataSet();

//Below line tells data adapter to fetch primary key information from database along with data

da.MissingSchemaAction = MissingSchemaAction.AddWithKey;

//At below line, connection opens, query gets fired and connection gets closed!

da.Fill(ds, "emp");

//Below line is taken from windows application; which show data in dataset into data grid control

dataGridView1.DataSource = ds.Tables["emp"];
```

INSERT Code:

```
// Below line empty data row with the structure of dataset - table

DataRow rw = ds.Tables[0].NewRow();

//Below line refers data row's first column by index i.e. '0'

rw[0] = Convert.ToInt32(100);

//One can also refer column by NAME as well

rw["Name"] = "Sachin";

rw["Address"] = "Mumbai";

//Actual addition into the dataset- tables happens in below line

ds.Tables[0].Rows.Add(rw);

//In order to update above row in final database; data adapter is used again

da.update(ds, "Emp")
```

Update Code:

```
//Below code tries to find a row inside data set – table with "No" value equal to 100

Datarow rw = ds.Tables[0].Rows.Find(100);

//After finding the row; one can use "rw" to set or update new values to the row

rw["Name"] = "MSD";

rw["Address"] = "Pune";

//In order to update above row in final database; data adapter is used again

da.update(ds, "Emp");
```

Delete Code:

```
//Below code tries to find a row inside data set – table with "No" value equal to 100

Datarow rw = ds.Tables[0].Rows.Find(100);

//Below line actually deletes the row from dataset - table

rw.Delete();

da.update(ds, "Emp");  // this line is for updating the change on database server side.
```

Summary:

In this part we discussed about:

- Basic difference between connected and disconnected architecture.
- Scenarios where disconnected architecture can be used
- Objects used in disconnected architecture
- Code snippets for the Select, Insert, Update and Delete using disconnected architecture
- Behind the scene details for disconnected architecture

We shall discuss few miscellaneous feature of ADO.NET in the next part.

ADO .NET Features

In previous parts we have seen the ADO.NET architectures: connected and disconnected. It provides us with a superficial layer through which a variety of classes are exposed via System.Data.dll. These classes help the programmers to connect to the variety of data sources like SQL, ORACLE, MS ACCESS, XML etc. The retrieval, manipulation and updation of data thus become very easy.

With the release of .Net framework 2.0 and SQL Server 2005, there are some remarkable changes in ADO.NET 2.0. We will see following features:

- ✓ Multiple Active Result Sets (MARS)
- ✓ Asynchronous Processing
- ✓ Bulk Copy Operations
- ✓ Batch Processing

1. Multiple Active Result Sets (MARS)

We have seen in the previous versions of .net f/w, which we can work with a single SqlDataReader with a single open connection. Each time we want to open a new SqlDataReader with the same connection, we have to close the previous connection reference and then open a new one.
This new feature of MARS, as the name suggests , allows multiple active data readers open on the same connection.
Allows an application to have more than one SqlDataReader open on a connection when each instance of **SqlDataReader** is started from a separate command.
It allows execution of multiple batches on a single connection.

How to enable MARS?
If we enable the MARS feature for use with SQL Server 2005, each command object used adds a new session to the connection.
By default the MARS facility is disabled. We can enable this by adding "MultipleActiveResultSets=True" key pair to the connection string. The modified connection string will thus look like:

```
 string connectionString = "Data Source=MSSQL1;" +
"Initial Catalog=NorthWind;Integrated Security=SSPI" +
"MultipleActiveResultSets=True";
```

Alternatively, if the feature is to be switched off, we can set "MultipleActiveResultSets=False".

There are a few considerations which should be known while using MARS in your applications:

- MARS operations execute synchronously on the server. Parallel execution at the server is NOT a MARS feature.
- A logical session is created, when a connection is opened with MARS enabled. This will cause additional overhead. To enhance the performance, the SQLClient caches the MARS session within a connection. Up to 10 sessions can be cached at a time.
- MARS operations are not thread safe.
- An application can check the MARS support by reading SqlConnection.ServerVersion value, which should be 9 for SQL Server 2005.

2. Asynchronous Processing:

Asynchronous programming is a core feature of .Net f/w 2.0 and ADO.NET 2.0 takes full advantage of the same.

Some command operations may take significant amount of time for execution. The current thread is thus blocked until this execution is finished. It is thus beneficial to put the long running operations at the background and let the foreground thread handle the current operations. This is multithreaded application. .net framework supports man such design patterns. ADO.Net supports these design patterns by making use of SqlCommand class methods like:
BeginExecuteNonQuery, BeginExecuteReader, and BeginExecuteXmlReader methods, paired with the EndExecuteNonQuery,EndExecuteReader, and EndExecuteXmlReader.

In order to use; asynchronous processing in C# .NET program; one will have to make below addition into the connection string:

```
string connectionString = "Data Source=MSSQL1; Initial
Catalog=NorthWind;Integrated Security=SSPI;
Asynchronous Processing=true";
```

3. Bulk Copy Operations:

You must have noticed from previous features that ADO.Net 2.0 mostly focuses on the performances. Microsoft SQL server previously supported a very popular command line utility: **bcp** for copying large files into tables or views in SQL server databases.
The SqlBulkCopy class provides similar functionality. It can be used to write huge data into only SQL Server tables. The source is not limited to SQL server. Any source tables which can be loaded into DataTable can be copied into SQL server.

Following are the steps to be followed while make bulk copy operations:

- Connect to the source server and obtain the data to be copied. Data can also come from other sources, if it can be retrieved from an IDataReader or DataTable object.
- Connect to the destination server (unless you want SqlBulkCopy to establish a connection for you).
- Create a SqlBulkCopy object, setting any necessary properties.
- Set the DestinationTableName property to indicate the target table for the bulk insert operation.
- Call one of the WriteToServer methods.
- Optionally, update properties and call WriteToServer again as necessary.
- Call Close, or wrap the bulk copy operations within a Using statement.

In order to use bulk copy feature in C#.NET program; use SqlBulkCopy class object like below:

SqlBulkCopy sbc = new SqlBulkCopy("Destination SQL Server Database Connection");

sbc.WriteToServer("Either a DataReader, DataTable kind of objects as a parameter here");

4. **Batch Processing:**

Generally, any application which is n-tier will have typically 3 layers: UI, Business logic and then the Database layer. Each of the layers communicates with each other. Whenever there are any commands to be executed on the SQL server, a round trip to the server has to be made. Instead of sending one operation at a time and increase the execution time, ADO.Net allows to group INSERT, UPDATE and DELETE operations from Dataset or DataTable via DataAdapter.

Since there is reduction in round trips, there is significant performance gain.

Batch updates are supported by SqlClient and OracleClient.

In previous versions, while working with disconnected architecture, the DataSet is made with all the changes like insert, update or delete. While making these updates to the SQL Server table, the Update method of SqlDataAdapter is called. This method checked row by row the changes happened in the dataset and then performed the updates row by row to the server.

Hence each time a network round trip will be made to the server.

The DataAdapter exposes a property UpdateBatchSize. This can be set to any positive value.

If we set this to 5, then a batch of statements will be made and submitted as a batch to the server and hence roundtrips are avoided.

If it is set to 0, then the batch size is set to maximum number , the server can handle.

naventure
SYSTEMS

C# Features: IV

With the introduction of VS2010 and .Net framework 4.0, there are few feature additions in C# 4.0. We will cover following features in this part:

- ✓ Dynamic Types
- ✓ Optional Parameters and named arguments
- ✓ Optional ref keyword while using COM

Dynamic Types:

C# is a strongly typed language. With the evolution of C# from 1.0 till 5.0, we can see that the language has evolved from statically typed to dynamically typed. With this new feature, there is a addition of new keyword: *dynamic.* This is a new pseudo type and is treated as System.Object. Thus any member access , or application of an operator on a value of such type is permitted without any type checking and resolution is postponed till runtime. This is also known as “duck typing”.

Consider following code snippet:

```
static int Getlength(dynamic obj)
        {
            return obj.Length;

        }

        static void Main(string[] args)
        {

            int length = 0;

            length = Getlength("Hello World"); // a string has length
property
            length = Getlength(new int[] { 10, 20, 30 }); // array has
length property
            length = Getlength(43); // this will throw exception at
runtime, since integer does not have Getlength

}
```

The dynamic keyword is used to declare the parameter in Getlength function and dynamically , parameters are sent to this function.

Similarly, we can also declare dynamic variables:

```
dynamic foo = "Hello";
Console.WriteLine(foo);
foo = 123;
Console.WriteLine(foo);
foo = true;
Console.WriteLine(foo);
```

This features helps tremendously in giving COM calls, when the type of arguments or return types are not known. The dynamic languages like Ruby-Python can thus communicate with C# efficiently.

Named and optional parameters:

This is a very interesting feature introduced in C# 4.0. While defining a function, the parameters with default values can be specified in the signature in the following manner:

```
public static void Optionalmethod(int firstparam, string secondparam =
"Optional", double thirdparam = 400.09)
      {

      }
```

In the above declaration, just the first parameter is mandatory and must be passed while giving a call to the function. Rest of the parameters even if not passed can take the default values specified in the signature above:

```
Optionalmethod(40); //this is same as
OptionalMethod(40,"Optional",400.09)
Optionalmethod(50, "I m different"); // this is same as
OptionalMethod(40,"I am different",400.09)
```

In above example , if we call the method in following manner , it will definitely give an error:

```
Optionalmethod(60, 500.09);
```

Here the compiler expects the second argument as a string parameter, but it finds a double value.

This conflict can be resolved by using following syntax:

```
Optionalmethod(60, thirdparam:500.09);
```

Here we specify the name of the optional parameter like :
[parameter_name]:[parameter_value].

Optional ref keyword when using COM:

This feature is introduced to have seamless COM communication.

Consider that the following is the COM method with the signature:

void Increment(ref int x);

The invocation to this method can be done in following manner:

Increment(1);

OR

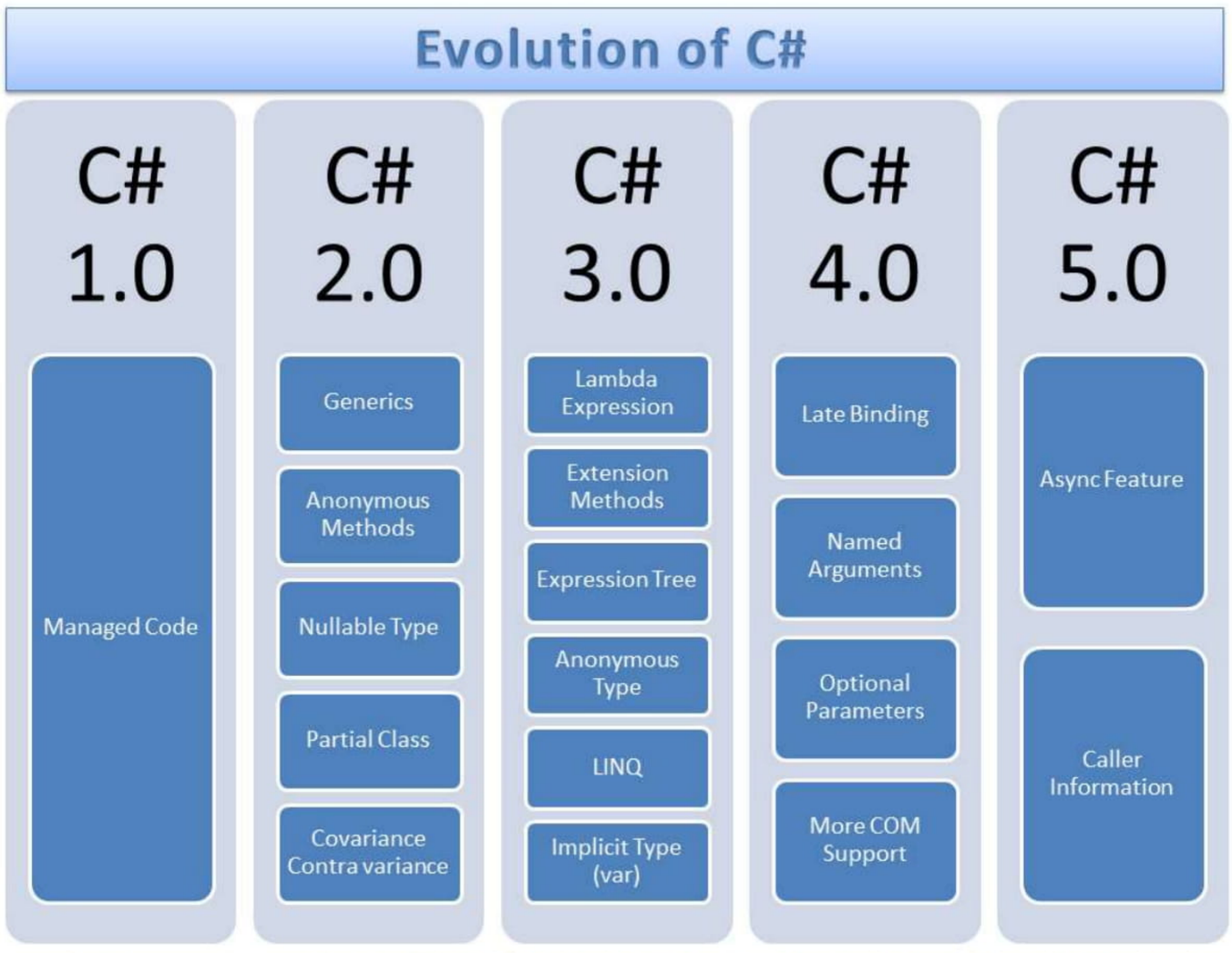
Int y=0;

Increment(ref y);

Summary

We have seen some very interesting features introduced in C# 4.0.

Consider following evolution:


Evolution of C#
C# 1.0
Managed Code
C# 2.0
Generics
Anonymous Methods
Nullable Type
Partial Class
Covariance Contra variance
C# 3.0
Lambda Expression
Extension Methods
Expression Tree
Anonymous Type
LINQ
Implicit Type (var)
C# 4.0
Late Binding
Named Arguments
Optional Parameters
More COM Support
C# 5.0
Async Feature
Caller Information

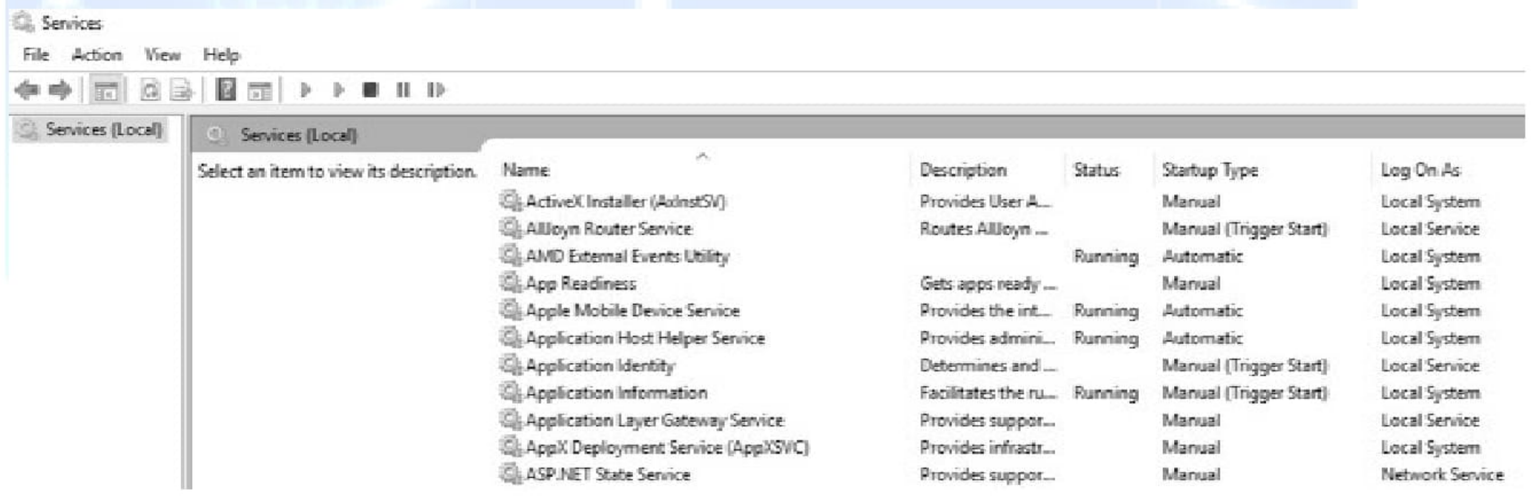
Windows Service(s)

Windows service is a program which operates at the background. Normally this program is EXECUTABLE (i.e. with EXE extension). Most of the time programs like auto update, reminders are windows services running at the background. These application(s) do not have User Interface for interaction.

These programs optionally start at the 'OS startup'. Creating windows service programs using .NET is too easy as we have default project template available for the same.

In order to see existing windows services installed on the machine:

1. Go to Control Panel
2. Click 'Administrative Tools'
3. Click 'View Local Services'
   (Optionally one may press 'Windows + R' & give a command 'services.msc')
4. Observe the services already installed on the machine:



If you notice; each service installed has:

- Name
- Optionally description
- Status
- Start up Type:
  - Manual: The service must be manually started after installation
  - Automatic: The service will start whenever the computer reboots
  - Disabled: The service cannot be started

  In order to specify start-up type for the service; one will have to use ServiceInstaller class.

- Log on as: It defines under which credential's context the service is running. Values can be
    - Local Service: This account is a predefined local account used by service
    - Network Service: Useful when service needs to connect to remote server
    - Local System: Service works under the current user's credentials

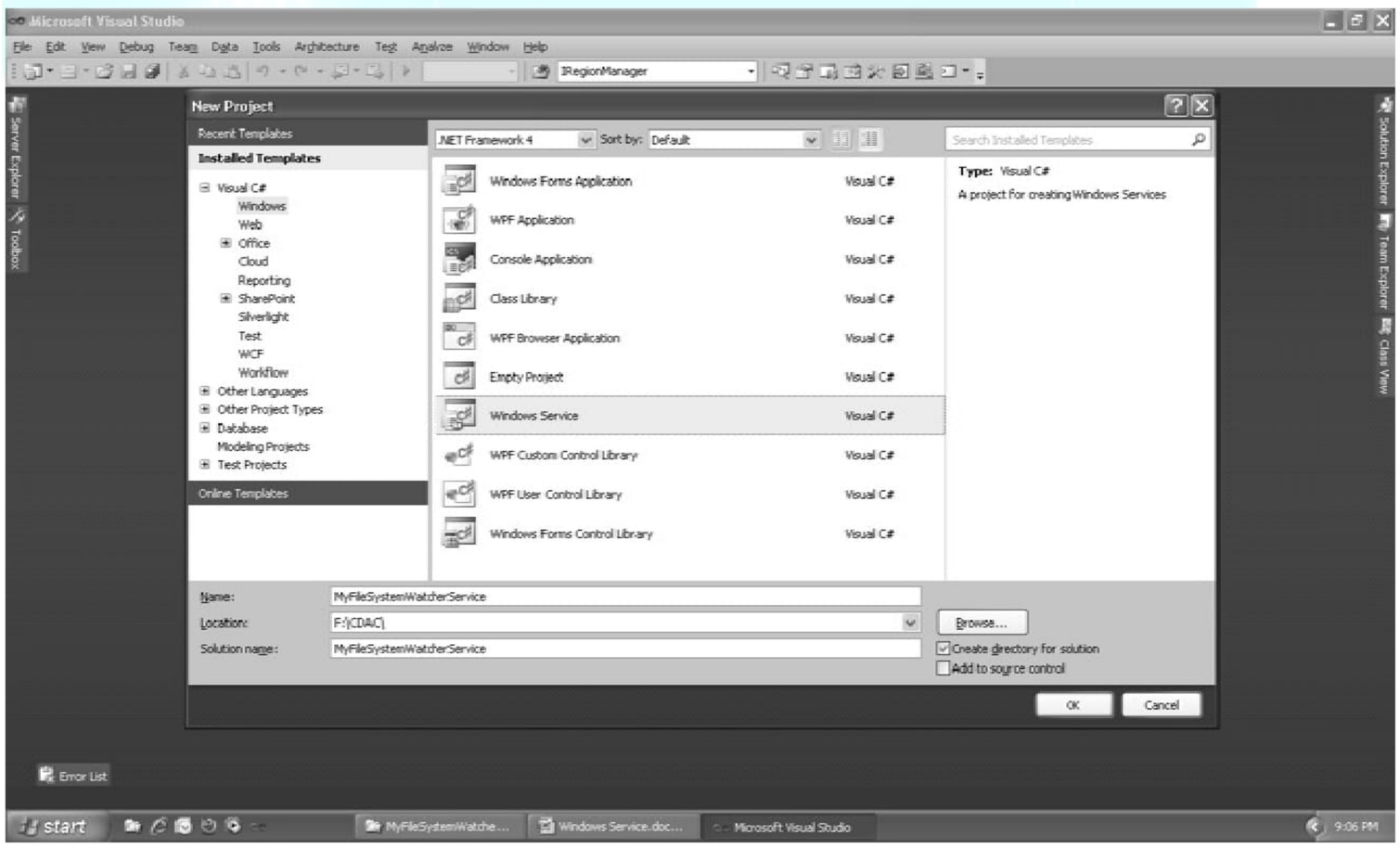
    In order to specify 'Account type' for the service; one will have to use ServiceProcessInstaller class.

After these few important points; let us build one windows service step by step.
So what are going to build to understand windows service concept?

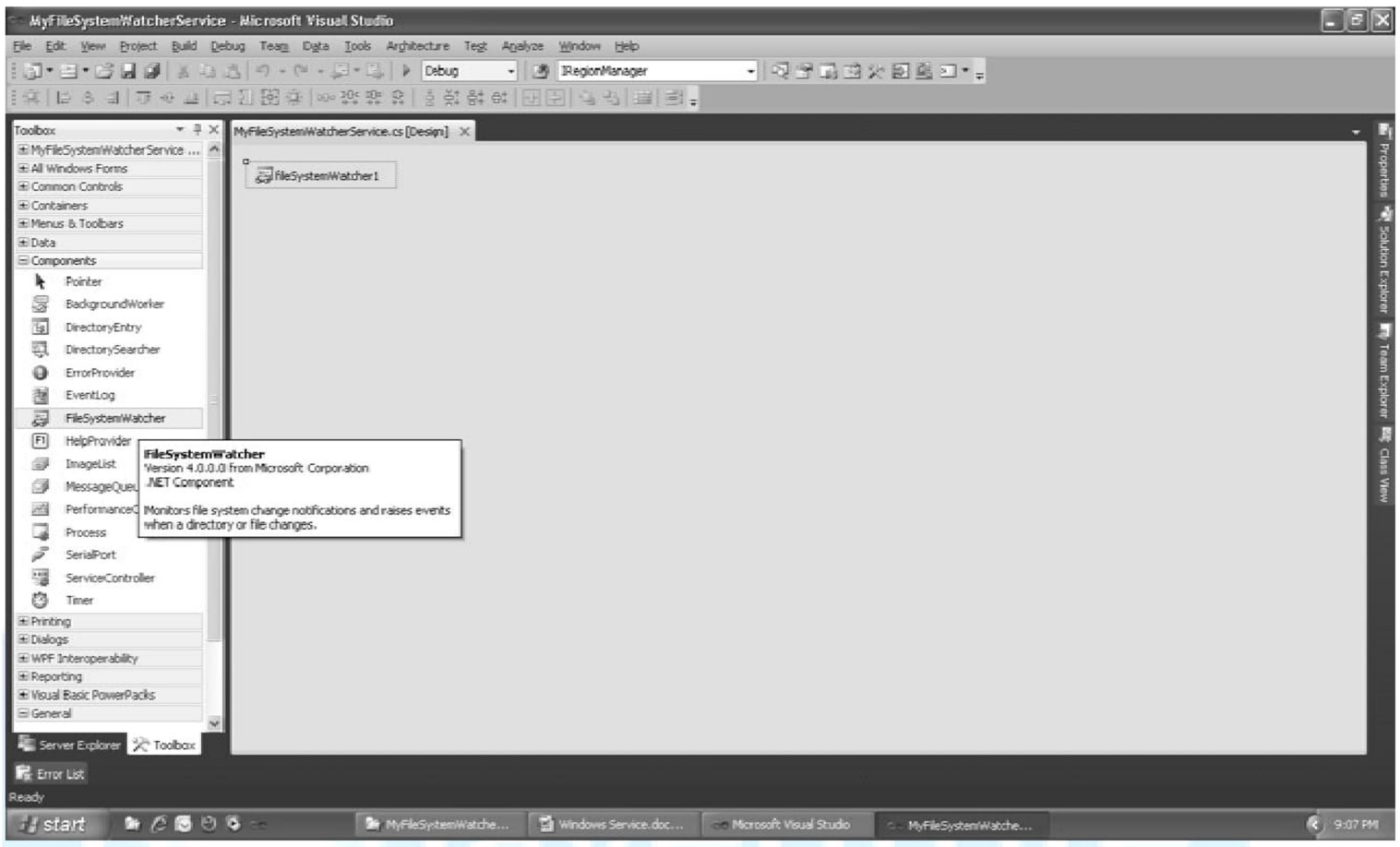
Details: We are going to create a windows service which will keep on watching a specific folder on hard drive. Any changes to this folder will be noted down by the windows service.
It seems we are going to write huge amount of logic; but it is simple. Majority of the functionality including looking for changes are already done when we start using 'FileSystemWatcher' control / class in .NET.

Let us start step by step:

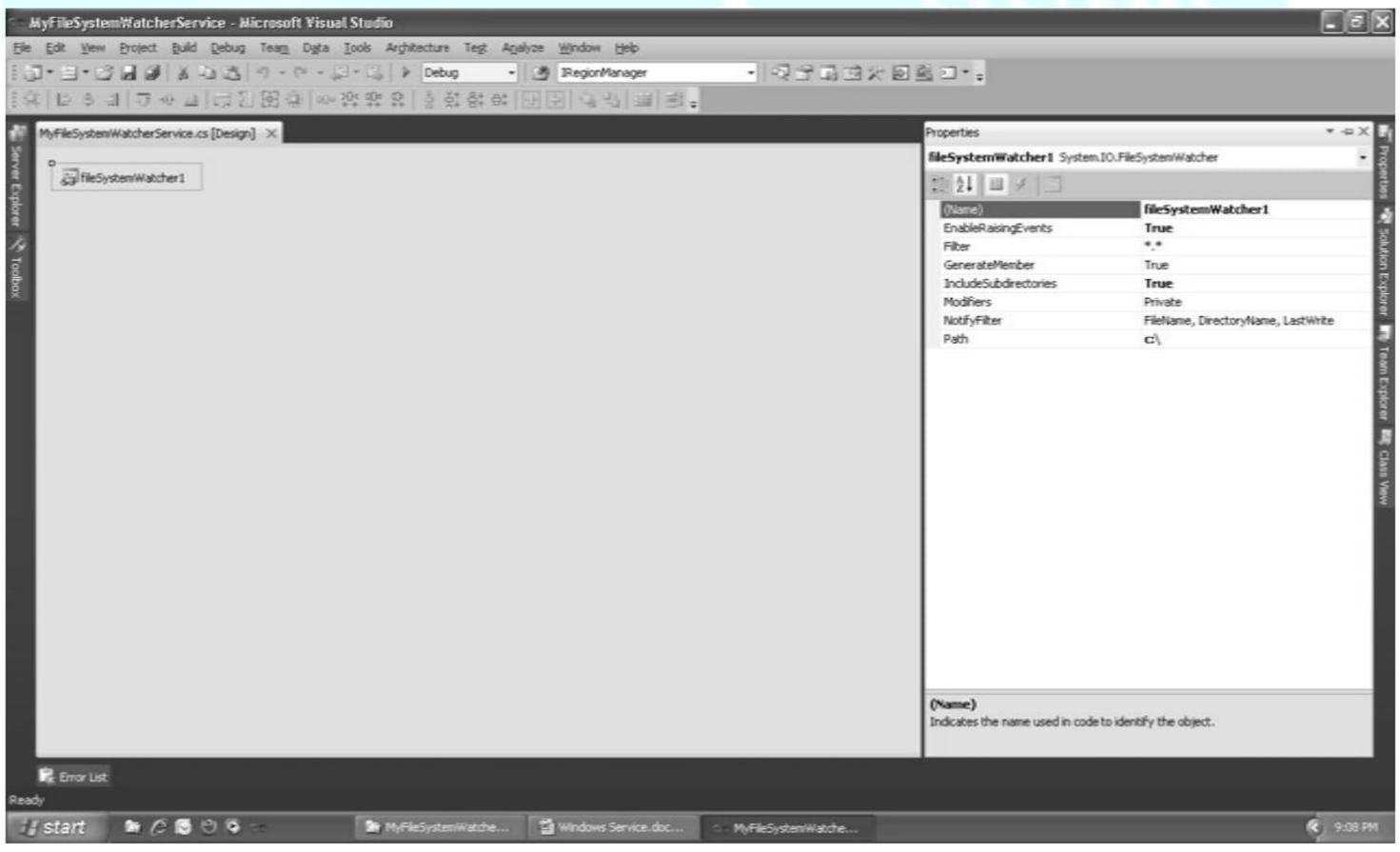
1. Start Visual Studio (Any Version VS 2010 onwards will do)
2. Click File->New->Project
3. Select Visual C# -> Windows (Left Pane) and Windows Service (Right Pane) Ref below snapshot:



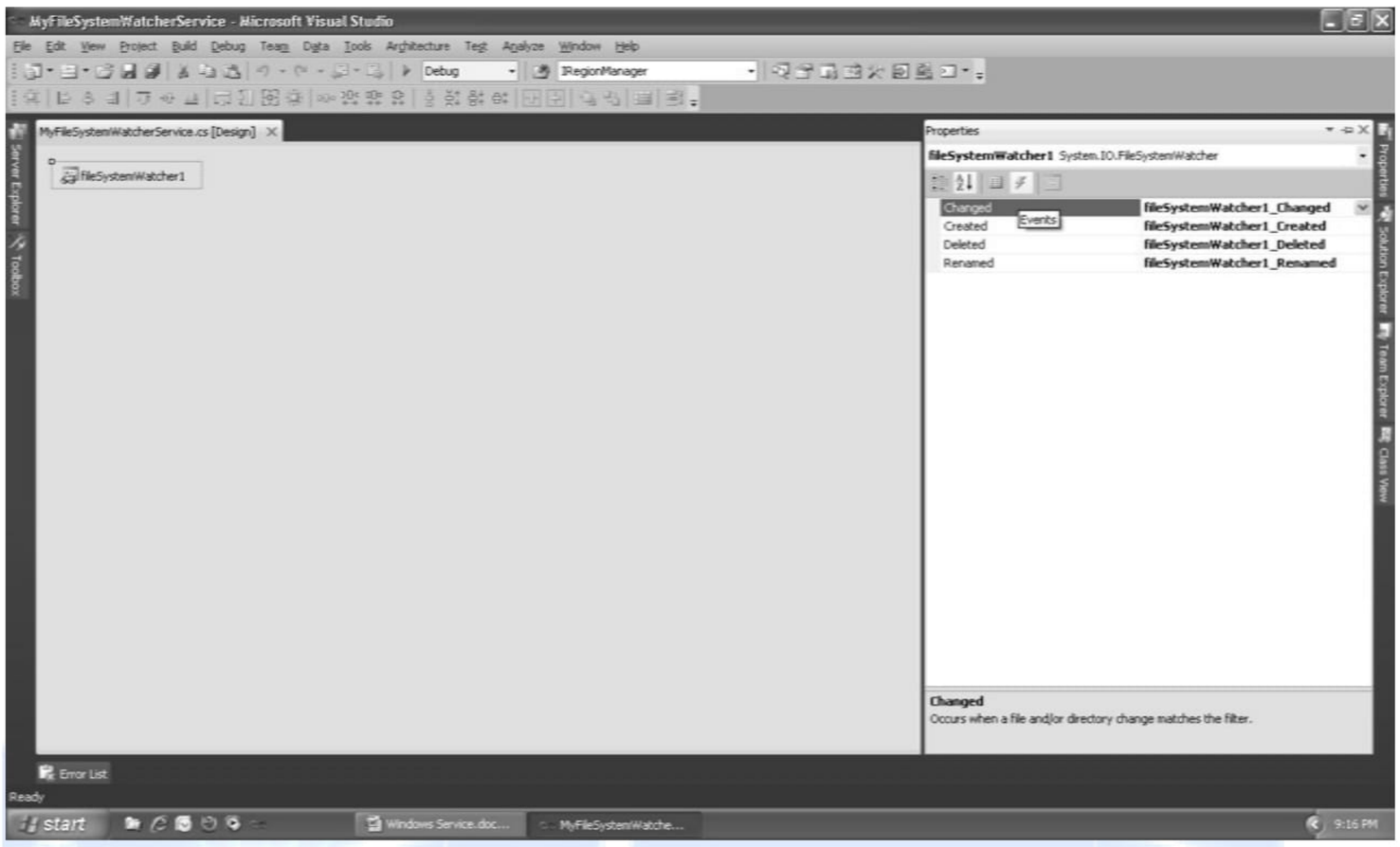
4. Name project as ‘MyFileSystemWatcherService’
5. Drag & Drop ‘FileSystemWatcher’ Control from the components section on to the service designer like shown below.



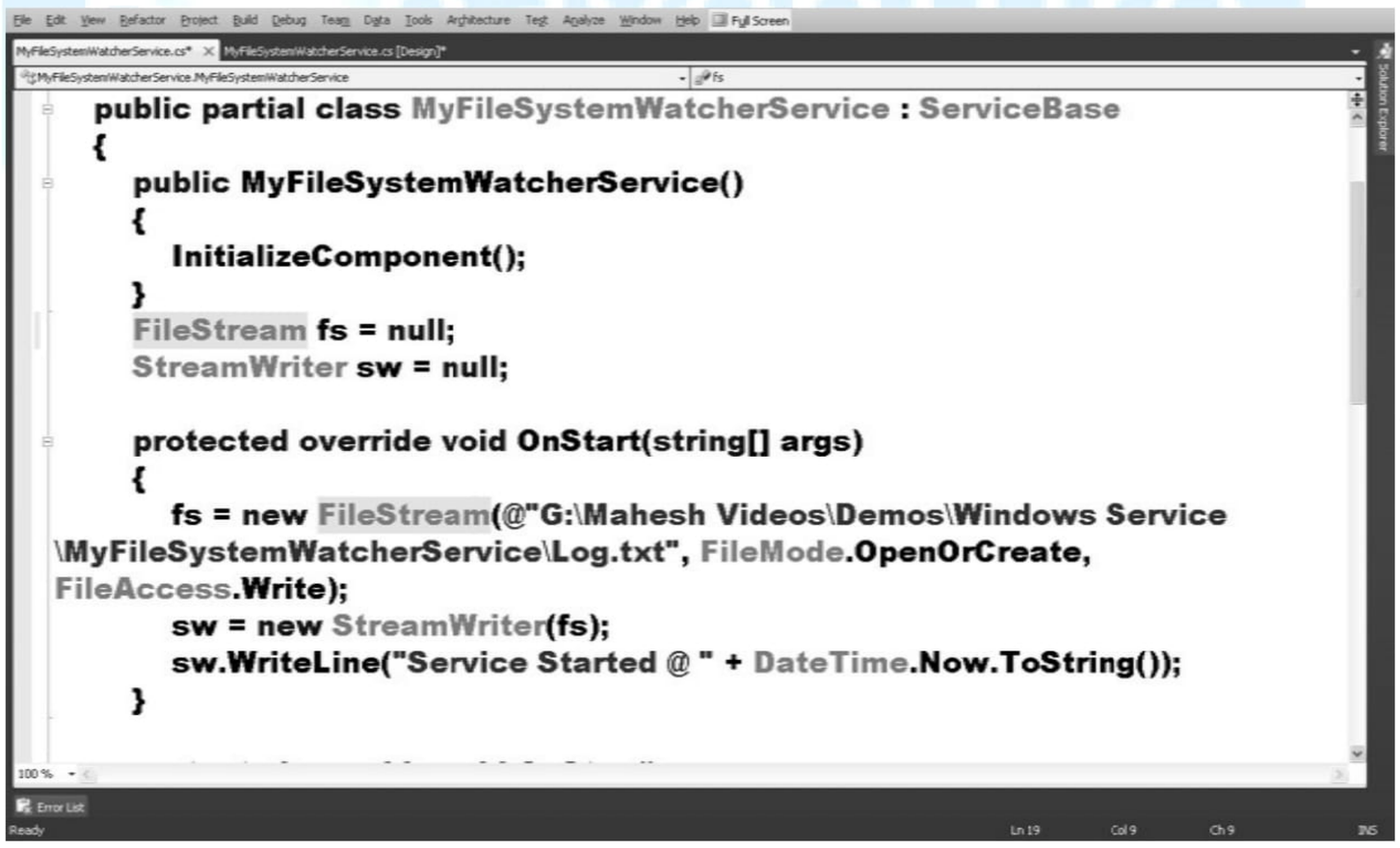
6. Set the drive or folder to - watch for – under FileSyetsmWatcher control properties:



7. Handle FileSystemWatcherEvents using properties windows:



8. Add below code snippet inside the code file.

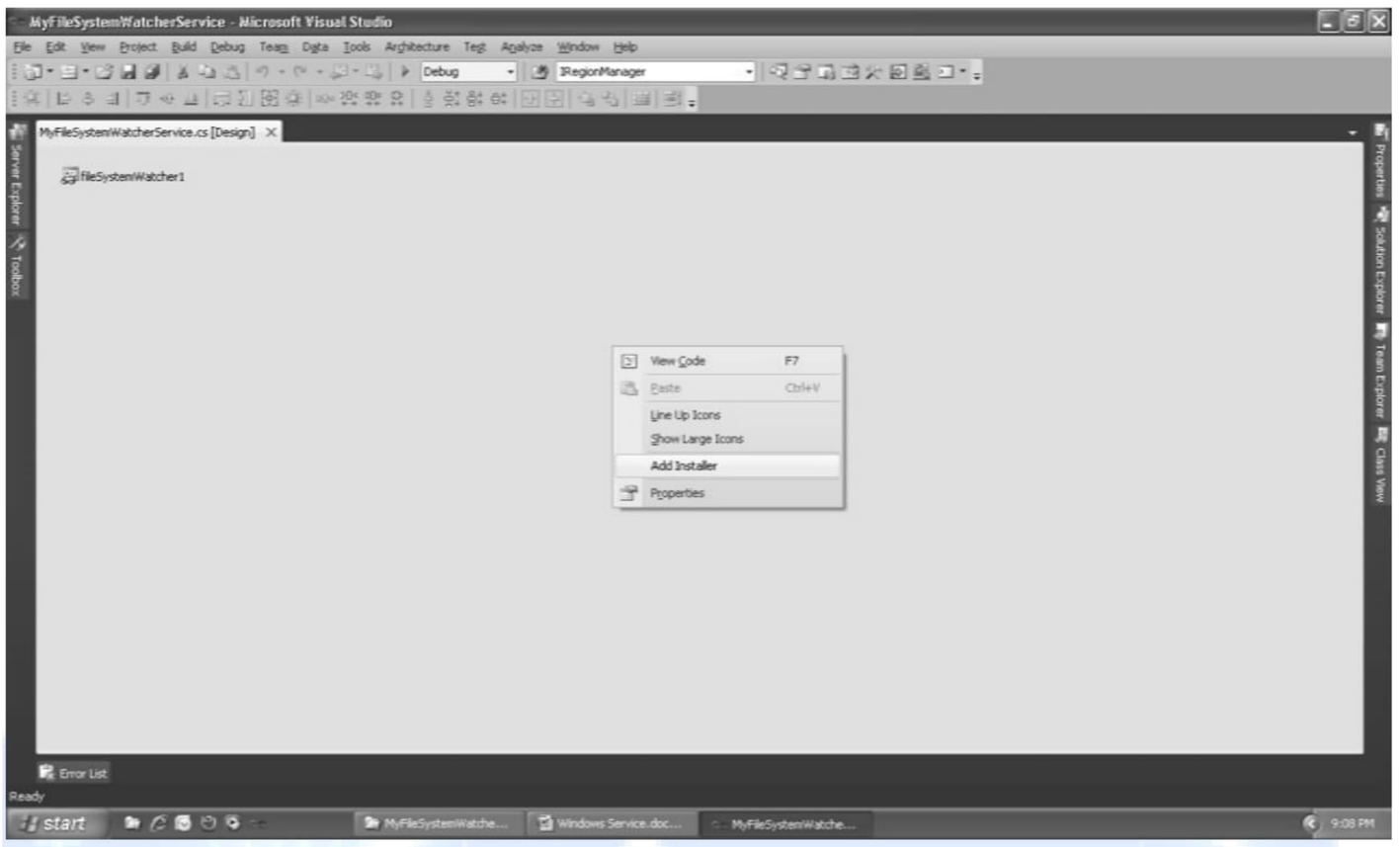


9. Continue adding code as shown below:

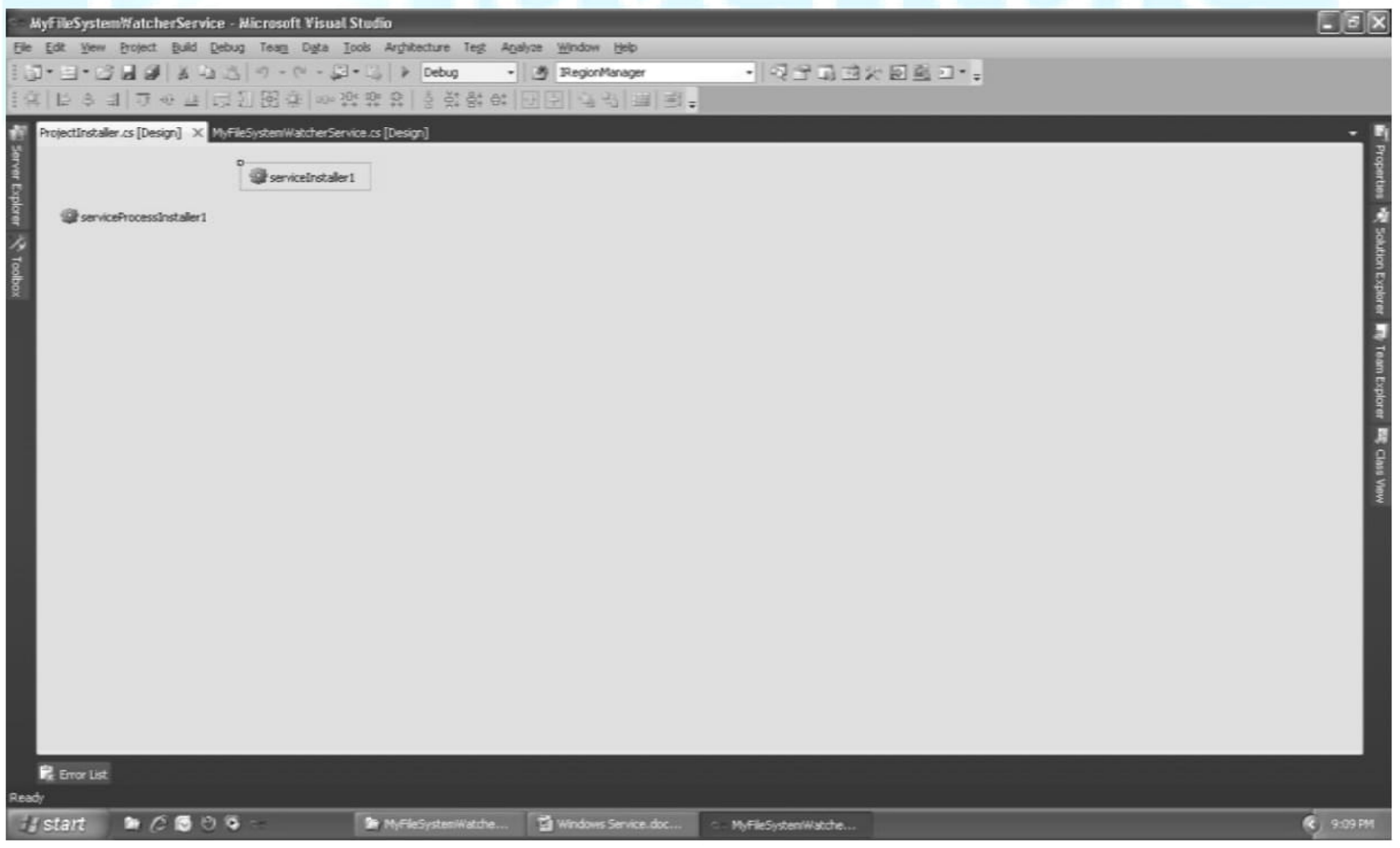
```
        protected override void OnStop()
        {
            sw.WriteLine("Service Stopped @ " + DateTime.Now.ToString());
            sw.Flush();
            sw.Close();
            fs.Close();
        }

        private void fileSystemWatcher1_Changed(object sender,
FileSystemEventArgs e)
        {
            sw.WriteLine("Content Changed for "+ e.FullPath +   " @ " +
DateTime.Now.ToString());
        }
```

```
        private void fileSystemWatcher1_Created(object sender,
FileSystemEventArgs e)
        {
            sw.WriteLine("Created  " + e.FullPath + " @ " + DateTime.Now.ToString());
        }

        private void fileSystemWatcher1_Deleted(object sender,
FileSystemEventArgs e)
        {
            sw.WriteLine("Deleted " + e.FullPath + " @ " + DateTime.Now.ToString());
        }

        private void fileSystemWatcher1_Renamed(object sender,
RenamedEventArgs e)
        {
            sw.WriteLine("Renamed " + e.OldFullPath + " to " + e.FullPath + " @ " +
DateTime.Now.ToString());
        }
```

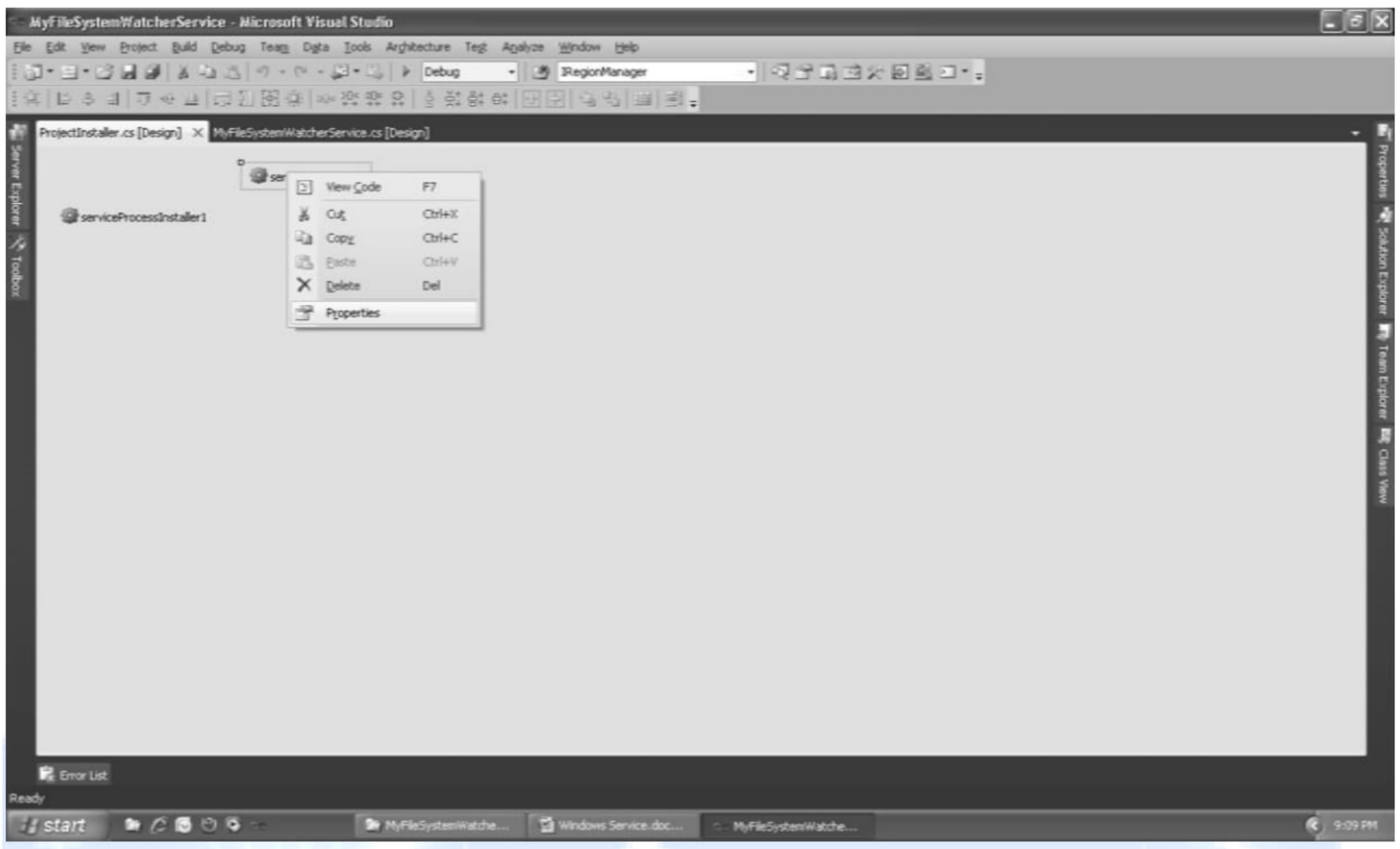
10. After adding source code. Right click on design window. Click ‘Add Installer’ from the context menu. See below snapshot.



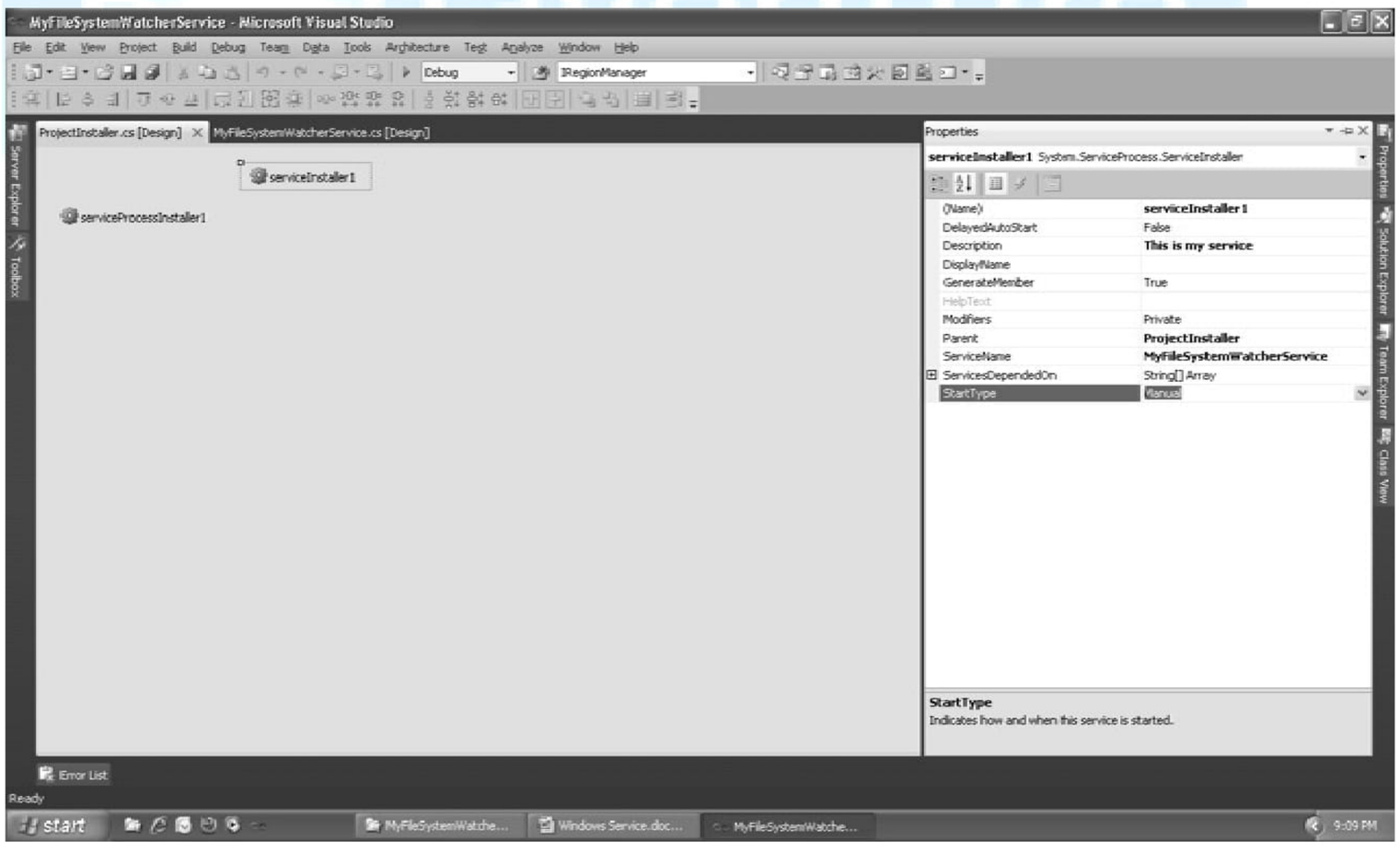
11. Two objects will be added into the project automatically. One is ServiceInstaller and other is ServiceProcessInstaller. We already have discussed importance of these two objects earlier.



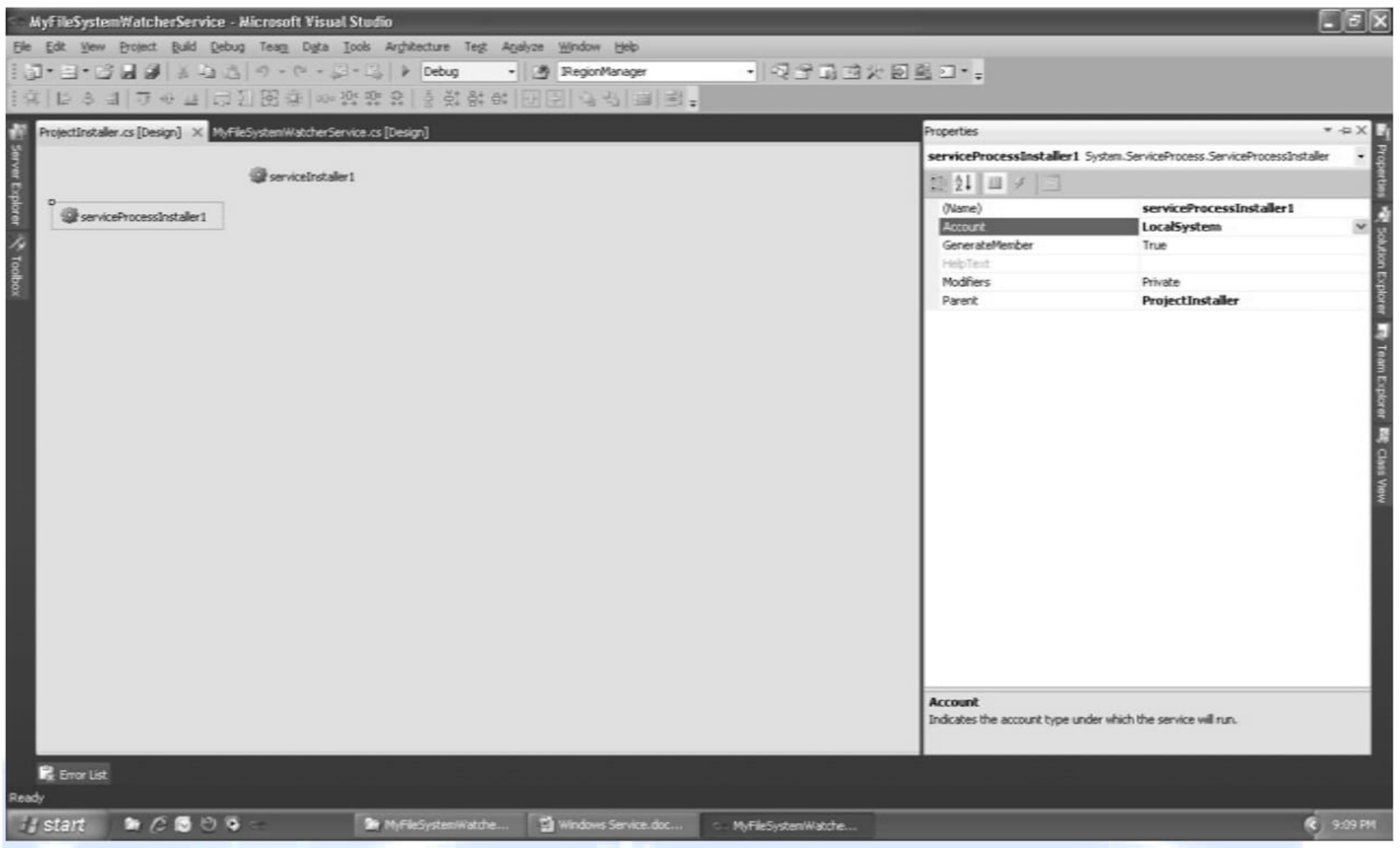
12. To set properties of ServiceInstaller; right click on the ServiceInstaller control. Click properties.



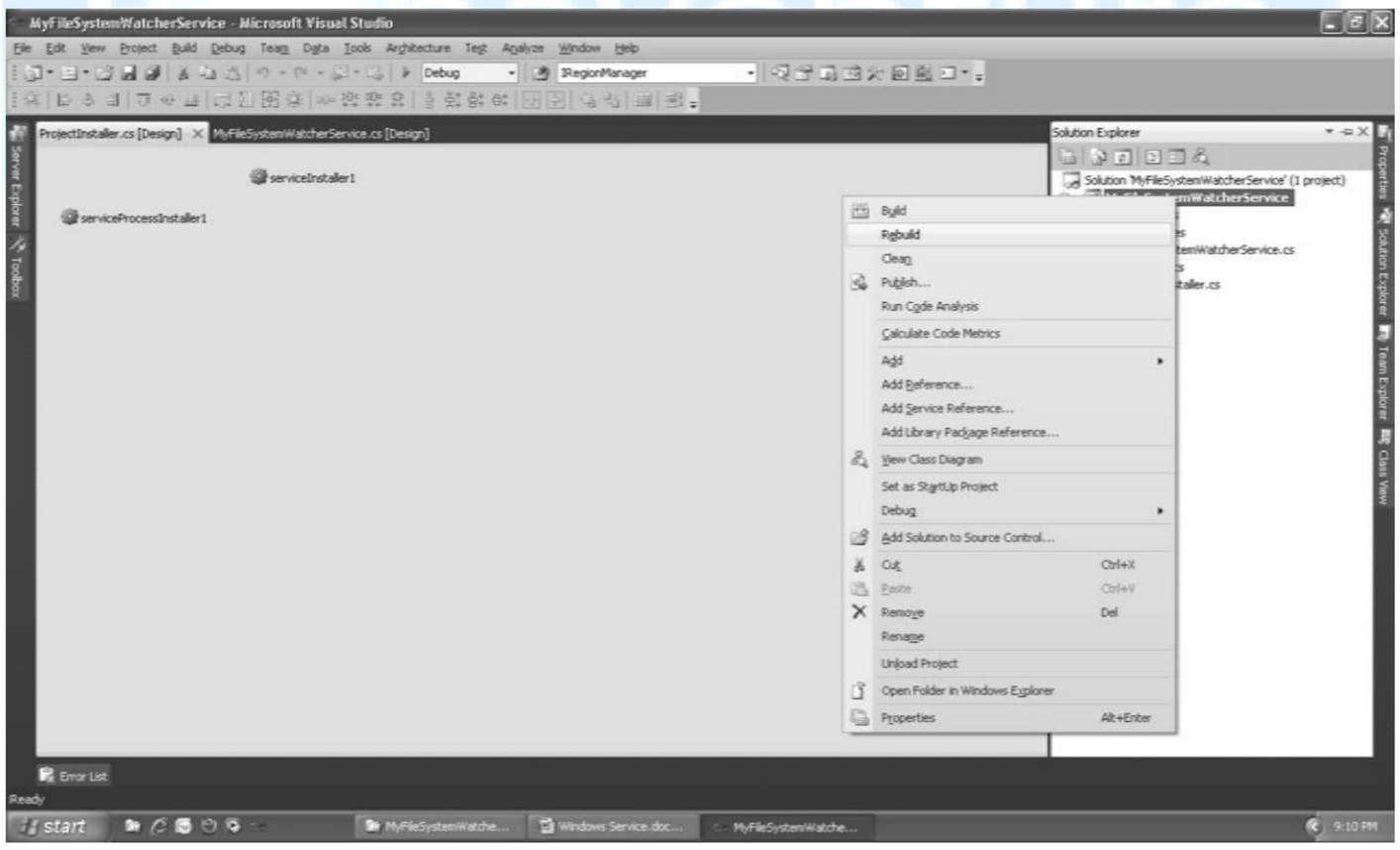
13. Set startup type for the service as manual



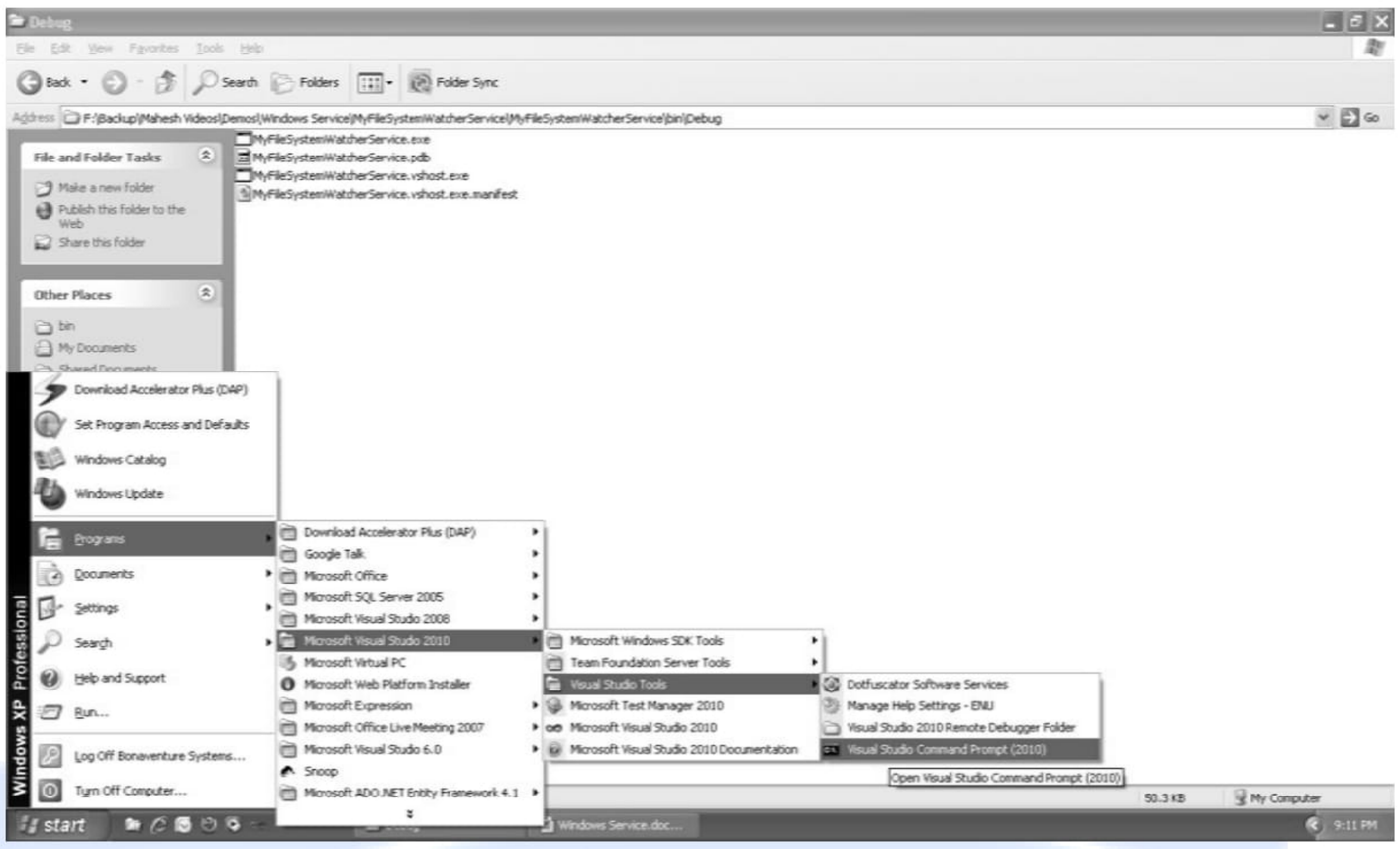
14. Set up account type using ServiceProcessInstaller in same way.



15. Compile the project. Make sure no errors exist in the code.

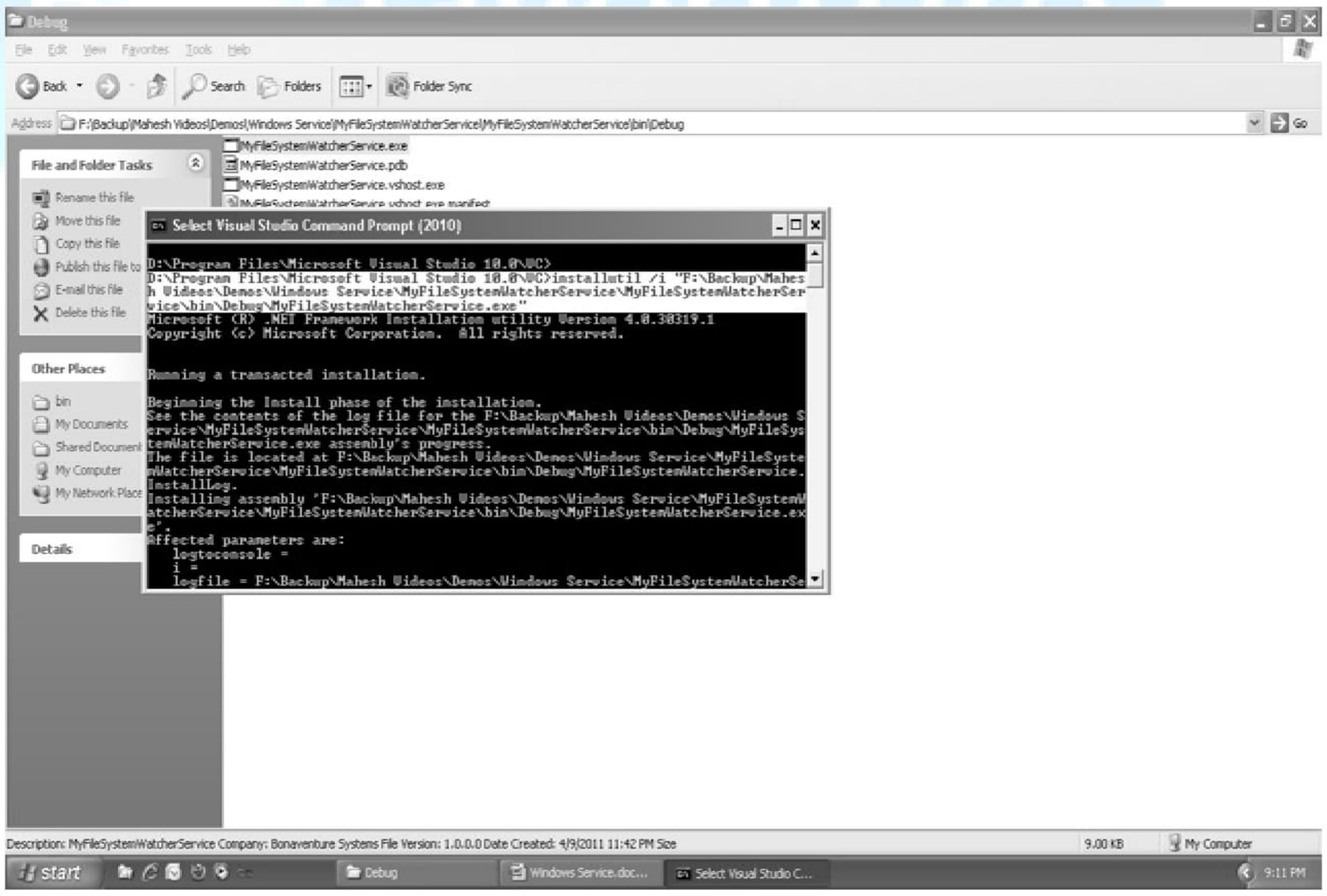


16. Open Visual Studio Command Prompt as an Administrator.

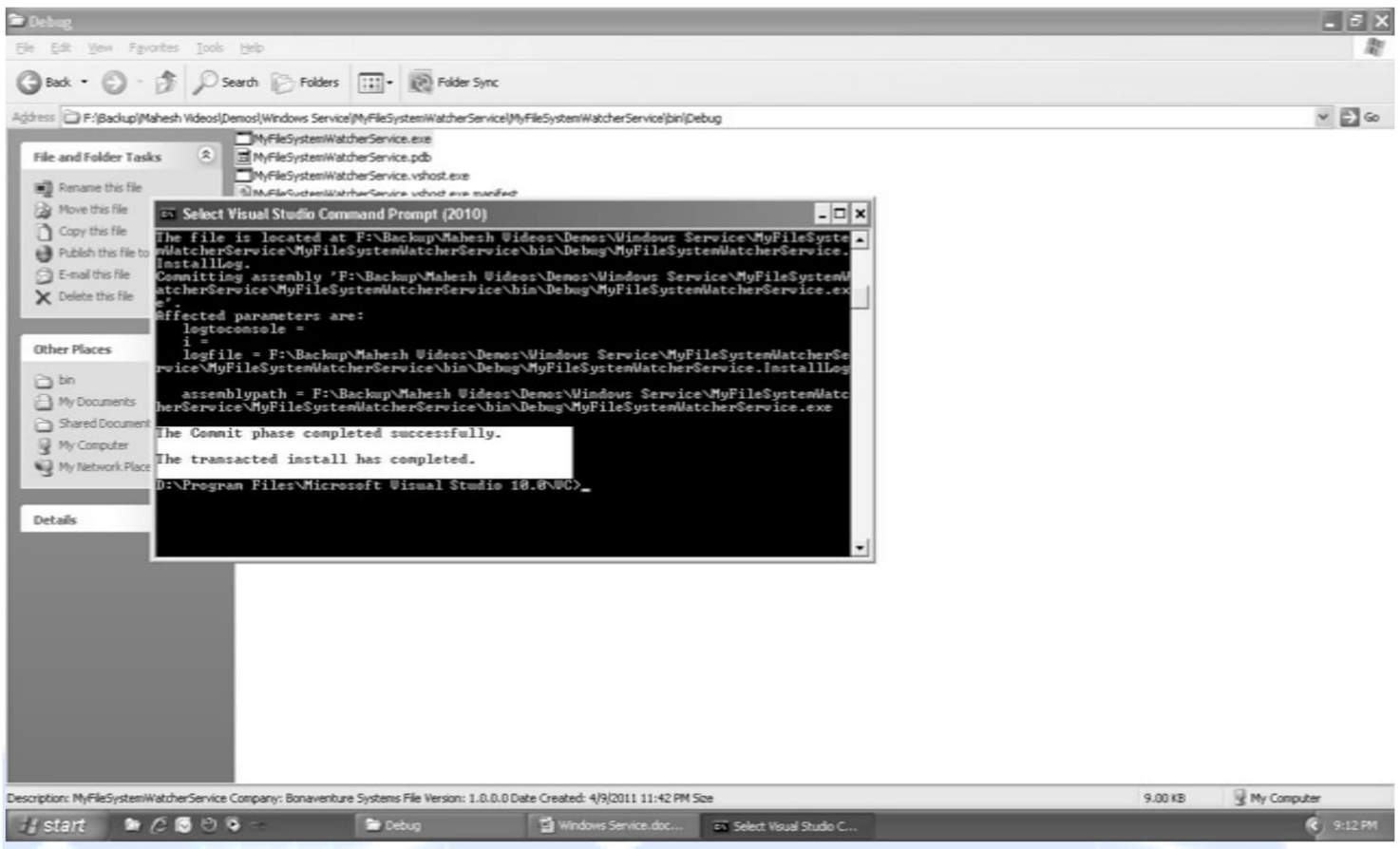


17. Provide command:

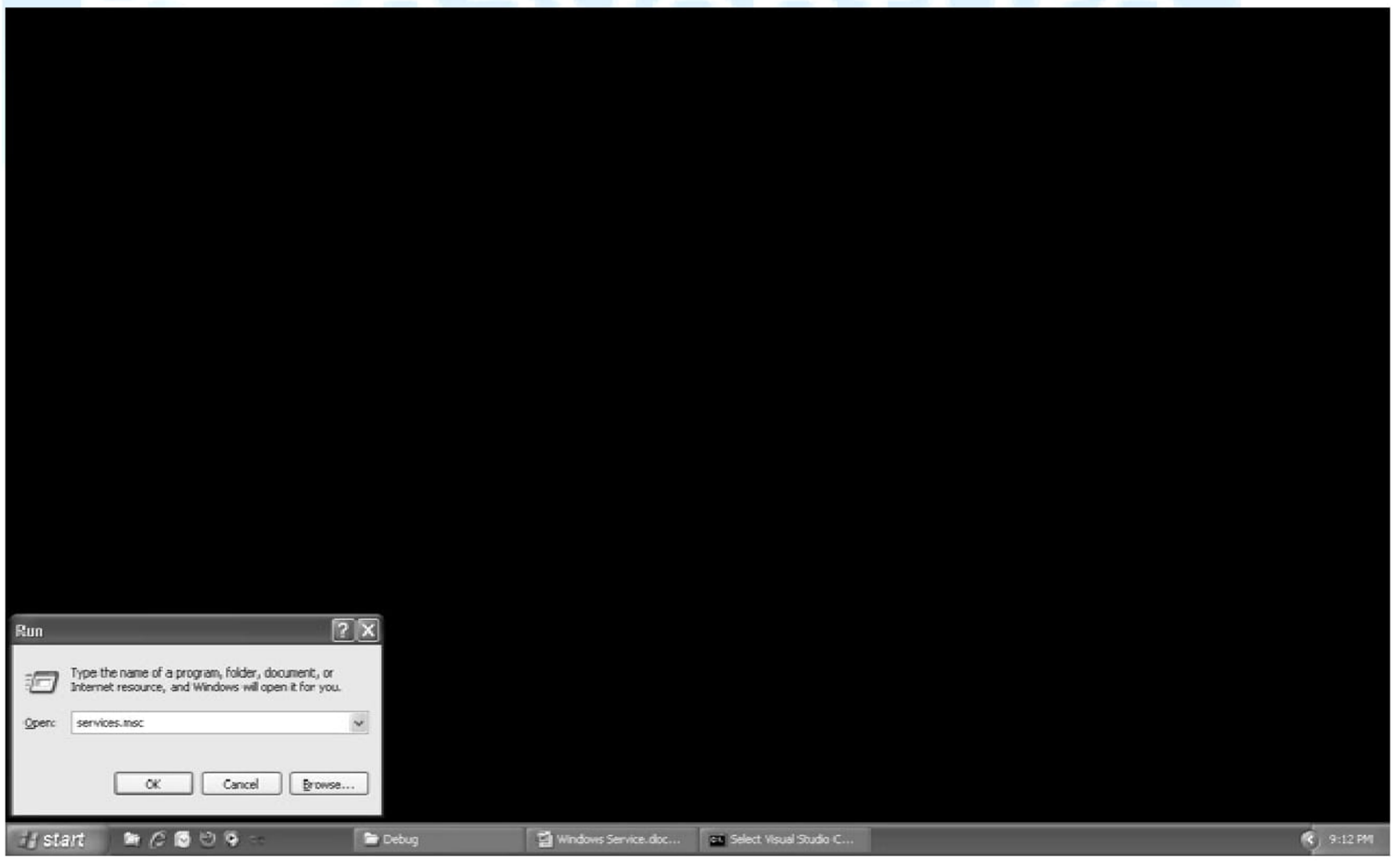
    INSTALLUTIL /i "<PATH OF MyFILESYSTEMWATCHER PROGRAM EXE>"



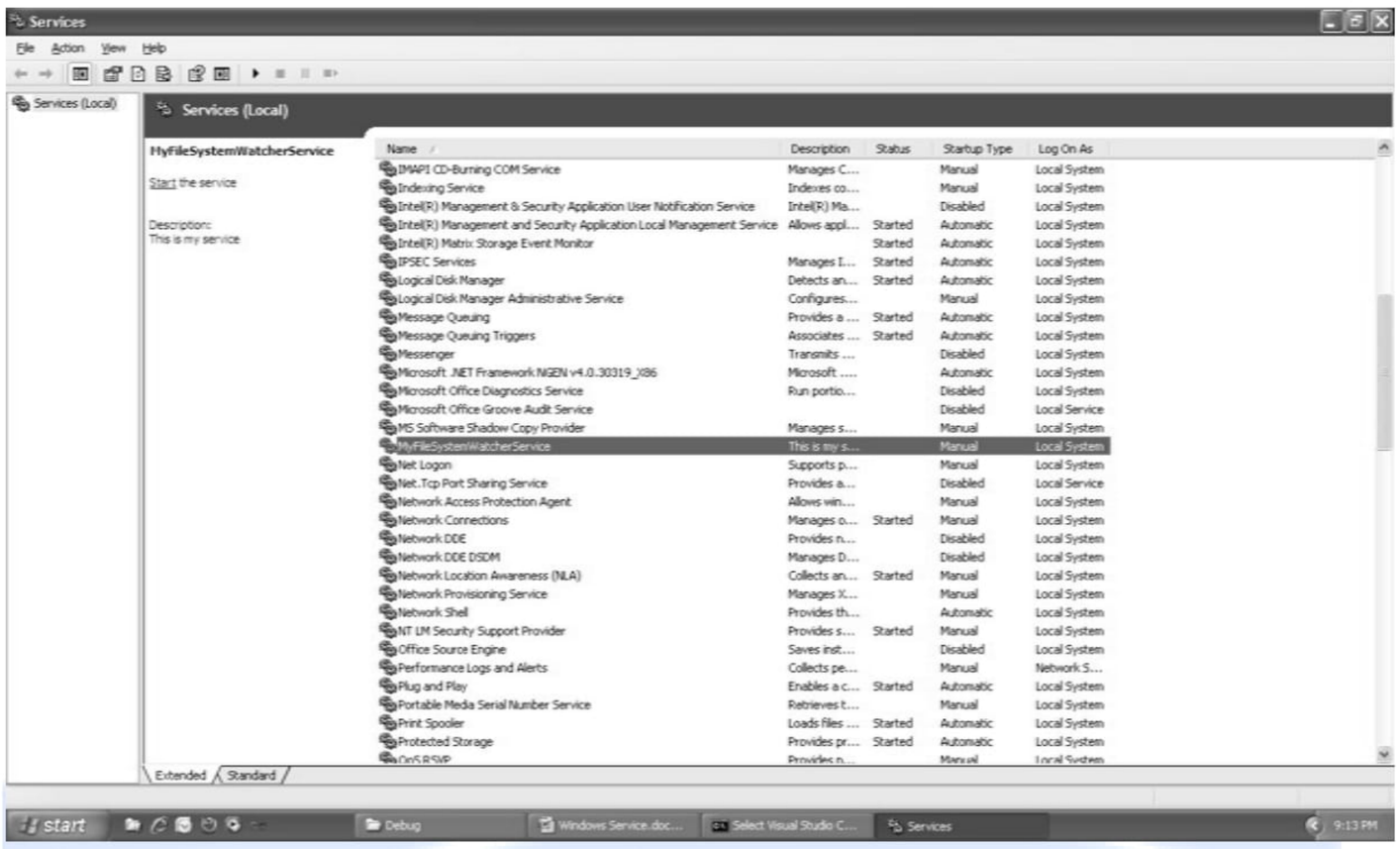
18. Make sure you get highlighted message : “Commit phase completed successfully.”



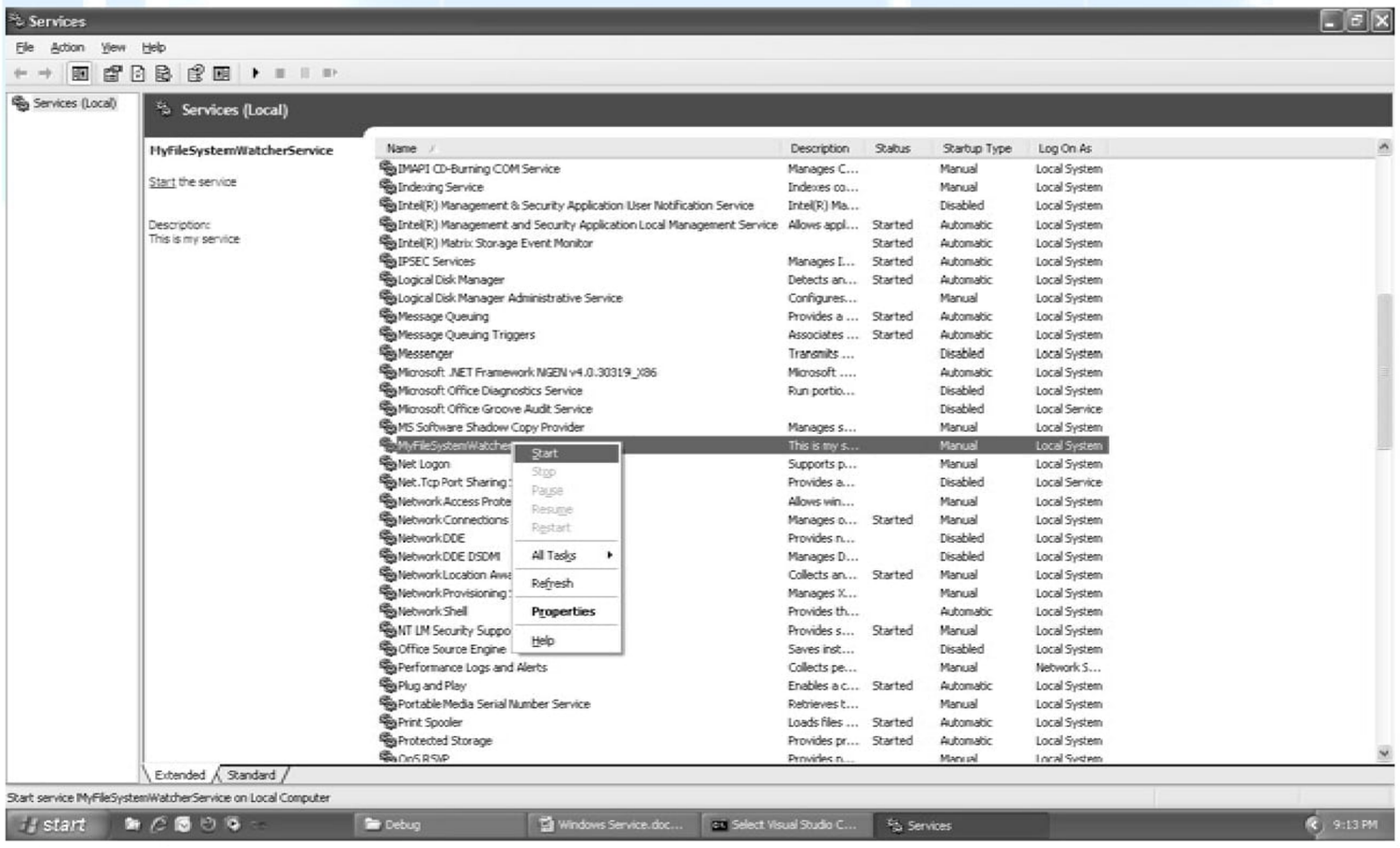
19. Use “Windows Key + R” and type “services.msc” command into it. Click “OK”



20. Observe the service “MyFileSystemWatcherService” listed in the existing service(s) section



21. Right click on the MyFileSystemWatcherService listed. Click “Start”



22. Observe the log created as per the code and witness the service’s behavior.

One can create Windows service's using timer, threads and make many interesting applications.

Summary:

We have observed in this part:

- What are windows service(s)
- Basic terms related with the Windows Services
- Implementation using C#
- Deployment of the windows service using "InstallUtil" command