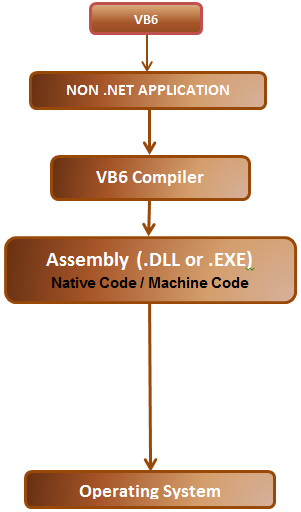
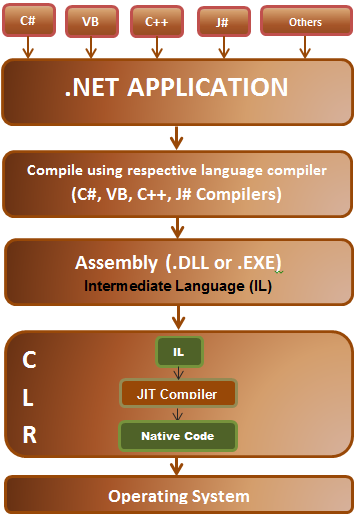
.NET Program Execution - Part 1

Let us first understand how VB6 or C++ programs (Non Dotnet applications) used to execute.  
We know that computers only understand machine level code. Machine level code is also called as **native or binary code**. So, when we execute a VB6 or C++ program, the respective language compiler, compiles the respective language source code into native code, which can then be understood by the underlying operating system and hardware. This process is depicted in the image below.

   
Native code is specific (**native**) to the operating system on which it is generated. If you take this compiled native code and try to run on another operating system it will fail. So the problem with this style of program execution is that, it is not portable from one platform to another platform.

Let us now understand, how a .Net program executes. Using dotnet we can create different types of applications. A few of the common types of .NET applications include Web, Windows, Console and Mobile Applications. Irrespective of the type of the application, when you execute any .NET application the following happens  
1. The .NET application gets compiled into Intermediate language (IL). IL is also referred as Common Intermediate language (CIL) and Microsoft Intermediate language (MSIL). Both .NET and non .NET applications generate an assembly. Assemblies have an extension of .DLL or .EXE. For example if you compile a windows or Console application, you get a .EXE, where as when we compile a web or Class library project we get a .DLL. The difference between a .NET and NON .NET assembly is that, DOTNET Assembly is in intermediate language format where as NON DOTNET assembly is in native code format.

2. NON DOTNET applications can run directly on top of the operating system, where as DOTNET applications run on top of a virtual environment called as Common Language Runtime (CLR). CLR contains a component called Just In-Time Compiler (JIT), which will convert the Intermediate language into native code which the underlying operating system can understand.  
  
So, in .NET the application execution consists of 2 steps  
1. Language compiler, compiles the Source Code into Intermediate Language (IL)  
2. JIT compiler in CLR converts, the IL into native code which can then be run on the underlying operating system.  
  
This process is shown in the image below.  
   
Since, a .NET assembly is in Intermedaite Language format and not native code, .NET assemblies are portable to any platform, as long as the target platform has the Common Language Runtime (CLR). The target platform's CLR converts the Intermedaite Language into native code that the underlying operating system can understand. Intermediate Languge is also called as managed code. This is because CLR manages the code that runs inside it. For example, in a VB6 program, the developer is responsible for de-allocating the memory consumed by an object. If a programmer forgets to de-allocate memory, we may run into hard to detecct out of memory exceptions. On the other hand a .NET programmer need not worry about de-allocating the memory consumed by an object. Automatic memory management, also known as grabage collection is provided by CLR. Apart, from garbage collection, there are several other benefits provided by the CLR, which we will discuss in a later session. Since, CLR is managing and executing the Intermediate Language, it (IL) is also called as managed code.  
  
.NET supports different programming languages like C#, VB, J#, and C++. C#, VB, and J# can only generate managed code (IL), where as C++ can generate both managed code (IL) and un-managed code (Native code).  
  
The native code is not stored permanently anywhere, after we close the program the native code is thrown awaya. When we execute the program again, the native code gets generated again.  
  
.NET program is similar to java program execution. In java we have byte codes and JVM (Java Virtual Machine), where as in .NET we Intermediate Language and CLR (Common Language Runtime)

### ILDASM and ILASM - Part 2

From Part 1 of DotNet basics videos, we understood that, compiling any .NET application would produce an assembly. Assemblies have an extension of .DLL or .EXE. For example if you compile a windows or Console application, you get a .EXE, where as when we compile a web or Class library project we get a .DLL. [Please watch Part 1, if you haven't done so already.](http://csharp-video-tutorials.blogspot.com/2012/07/net-program-execution-part-1.html)

The entire source code of a project is compiled into Intermediate Language and packaged into the assembly. A .NET assembly consists of **Manifest**and**Intermediate language**. Manifest contains metadata about the assembly like the name, version, culture and strong name information. Metadata also contains information about the referenced assemblies. Each reference includes the dependent assembly's name, assembly metadata (version, culture, operating system, and so on), and public key, if the assembly is strong named.  
  
**Some information in the assembly manifest can be modified using attributes. For example to modify the version number follow these steps**  
**1.** Expand the properties folder in solution explorer. Every project in .NET has a properties folder.  
**2.** Open AssemblyInfo.cs file that is present under properties folder.  
**3.** In this file, you should see AssemblyVersion attribute, which is defaulted to 1.0.0.0. Change this to 2.0.0.0 and rebuild the solution.  
**4.** Now open the assembly using ILDASM.exe  
  
**To peek inside an assembly with ILDASM follow these steps.**  
**1.** Navigate to Visual Studio Command Prompt (Start -> All Programs -> Microsoft Visual Studio 2010 -> Visual Studio Tools -> Right Click on Visual Studio Command Prompt 2012 and select "Run as Administrator")  
**2.** Once you have the "Visual Studio Command Prompt 2012" open, type in the following command and press enter  
**Ildasm.exe C:\YourDirectoryPath\YourAssembly.exe**  
  
This command should open the assembly and you will find the manifest and the types (classes, structs etc..) in the assembly. At the bottom you can see the version of the assembly.  
  
**If you want to save the Intermediate Language to a text file.**  
**1.** Select File Menu from the ILDASM tool.  
**2.** Select Dump and you will see "Dump Options Window"  
**3.** Click OK on "Dump Options Window"  
**4.** Now enter the file name of your choice. For this example let's enter sample and save it to the C: drive.  
**5.** Now navigate to C: drive in windows explorer and you should see Sample.il  
**6.** Open Sample.il with notepad and you should see assembly metadata and IL(Intermediate Language).  
  
**If you want to rebuild an Assembly from the Sample.il file we can use a tool ILASM.exe**  
**1.** Type the following command in "Visual Studio Command Prompt" and press enter  
**ILASM.exe C:\Sample.il**  
**2.** Now navigate to C: drive in windows explorer and you should see Sample.exe  
  
We use ILDASM (Intermediate Language Disassembler) to peek at the assembly manifest and IL. You can also use this tool to export manifest and IL to a text file.  
  
We use ILASM.exe (Intermediate Language Assembler) to reconstruct an assembly from a text file that contains manifest and IL.

### Strong naming an assembly - Part 3

**Strong naming an assembly or Signing an assembly with strong name.**  
  
  
**In .NET assemblies can be broadly classified into 2 types**  
**1.** Weak Named Assemblies  
**2.** Strong Named Assemblies  
  
**An assembly name consists of 4 Parts**  
**1.** Simple textual name.  
**2.** Version number.  
**3.** Culture information (If provided, otherwise the assembly is language neutral)  
**4.** Public key token

We use AssemblyVersion attribute to specify the Assembly version. The default is 1.0.0.0. The version number of an assembly consists of the following four parts:  
**1.** Major Version  
**2.** Minor Version  
**3.** Build Number  
**4.** Revision Number  
  
You can specify all the values or you can default the Revision and Build Numbers by using the '\*' as shown below:  
[assembly: AssemblyVersion("2.1.\*")]

AssemblyCulture attribute is used for specifying the culture. By default an assembly is language neutral, as the AssemblyCulture attribute contains empty string. If you specify any string other than an empty string for the culture parameter, the assembly becomes a satellite assembly. In fact, compilers use this attribute to distinguish between main assembly (language neutral) and a satellite assembly. We will talk about satellite assemblies in a later session.  
  
We use AssemblyKeyFile attribute to sign the assembly with a strong name. To the constructor of AssemblyKeyFile attribute, we need to pass the path of the key file, that contains the private and public key. To generate the key file  
**1. Open Visual Studio Command Prompt**  
**2. Type the command and press enter: sn.exe -k c:\KeyFile.snk**  
  
The key file with name KeyFile.snk should be generated in the C: drive. In SN.exe, SN stands for Strong Name. Key files have the extension of .snk  
  
Finally, In AssemblyInfo.cs file of the project, specify AssemblyKeyFile attribute as shown below and build the project. This process will strongly name an assembly.  
[assembly: AssemblyKeyFile("KeyFile.snk")]  
  
**A strongly named assembly should have all of the following**  
**1.** The textual assembly name.  
**2.** The assembly Version number.  
**3.** The assembly should have been signed with private/public key pair.  
  
If the assembly is not signed with private/public key pair, the assembly is weak named and not guaranteed to be unique, and may cause DLL hell. Strong named assemblies are guaranteed to be unique and solves DLL hell. You cannot install an assembly into GAC unless, the assembly is strongly named.   
  
**In the upcoming video sessions we will discuss**  
**1.** What is GAC. How and when to install an assembly into GAC?  
**2.** What is DLL HELL?  
**3.** How is DLL HELL solved with .NET?

### What is GAC. How and when to install an assembly into GAC - Part 4

**GAC** stands for **Global Assembly Cache** and contains strong named assemblies. Assemblies in the GAC can be shared by all applications running on that machine, without having to copy the assembly locally. It is recommended to install an assembly into GAC, only when required and shared by applications, otherwise they should be kept private. You shouldn't add an assembly into the GAC, if you wish to deploy your application to another machine using XCopy deployment. This is because in XCopy deployment, we only copy the application files to the target machine and not the GAC contents. XCopy deployment is simply copying files from one location to another.

With the introduction of .NET 4.0, we have 2 GAC's. One for DotNet 2.0 to 3.5 assemblies and the other for .NET 4.0 assemblies. The following are the paths for the 2 GAC's  
**1.** C:\Windows\Assembly - For .NET 2.0 - 3.5 assemblies  
**2.** C:\WINDOWS\Microsoft.NET\assembly - For .NET 4.0 assemblies

To install an assembly into the GAC, the assembly must be strongly named, otherwise you get an error stating - Failure adding assembly to the cache: Attempt to install an assembly without a strong name. There are 2 ways to install an assembly into GAC.  
**1.** Simply Drag and Drop  
**2.** Use GacUtil.exe (GAC Utility Tool)  
  
To install an assembly using gacutil, use the following command. This command installs SampleAssembly.dll into the GAC. If you have build this project using .NET framwork 4.0 then look in C:\WINDOWS\Microsoft.NET\assembly, else look in C:\Windows\Assembly.  
Gacutil -i C:\SampleProject\SampleAssembly.dll  
  
**Note:** If you are using Visual Studio 2010, then by default the target framework for any new project is .NET 4.0. If you want to change the target framework, right click the project and select properties. In the properties window, you can change the target framework version.  
  
To uninstall an assembly from the GAC, using GAC utility, use the following command.   
Gacutil -u MyClassLibrary  
  
If there are multiple versions of MyClassLibrary assembly, in the GAC, then all these versions will be removed by the above command. If you want to remove only one of the assemblies then specify the full name as shown below.  
gacutil -u ClassLibrary,Version=1.0.0.0,PublicKeyToken=eeaabf36d7783129  
  
**Note:** Please make sure there are no spaces between Comma(,) and the words "Version" and PublicKeyToken, otherwise you get an error stating Unknown option: Version=1.0.0.0.Also, don't specify the assembly extension (.dll or .exe) when uninstalling, otherwise the assembly will not be uninstalled. You will just get a message stating Number of assemblies uninstalled = 0

### How .NET finds the assemblies during program execution - Part 5

**1.** .NET figures out what version is needed : Usually the information about the dependant assemblies is present in the application's assembly manifest. CLR checks the application configuration file, publisher policy file(if exists), and machine config file for information that overrides the version information stored in the calling assembly's manifest.   
**2.** .NET searches GAC (Global Assembly Cache) : .NET searches GAC only if the assembly is strongly named.  
**3.** If the assembly is not found in the GAC, and if there is a .config file, then .NET searches the location in the cofiguration file, else .NET searches directory containing the executable (.EXE)  
**4.** If the assembly is not found, the application terminates with error.  
  
**Note: Version checking is not done for Weakly Named Assemblies (Assemblies without a strong name)**

### What is DLL HELL in .NET

Let us try and understand DLL HELL problem with an example. Please refer to the image below. [If you want to know how dll hell problem is solved in .net, you can read this article.](http://venkataspinterview.blogspot.com/2011/06/how-is-dll-hell-problem-solved-in-net.html)

|  |
| --- |
| http://1.bp.blogspot.com/-GdfFIGaLmQs/TeU4FcNPW6I/AAAAAAAAAGk/lPEo1eZNHm0/s1600/DllHell.png |

**1.** I have 2 applications, A1 and A2 installed on my computer.   
  
**2.** Both of these applications use shared assembly shared.dll  
  
**3.** Now, I have a latest version of Application - A2 available on the internet.  
  
**4.** I download the latest version of A2 and install it on my machine.  
  
**5.** This new installation has over written Shared.dll, which is also used by Application - A1.  
  
**6.** Application - A2 works fine, but A1 fails to work, because the newly installed Shared.dll is not backward compatible.  
  
So, DLL HELL is a problem where one application will install a new version of the shared component that is not backward compatible with the version already on the machine, causing all the other existing applications that rely on the shared component to break. With .NET versioning we donot have DLL HELL problem any more.  
  
[How is the DLL HELL problem solved in .NET, is another very important .net interview question. Click here to read.](http://venkataspinterview.blogspot.com/2011/06/how-is-dll-hell-problem-solved-in-net.html)  
  
**If you can improve this answer further, please feel free to do so by submitting the form below. Thank you very much for your contribution.**

### How is the DLL HELL problem solved in .NET [Please read What is dll hell? interview question, before proceeding with this article.](http://venkataspinterview.blogspot.com/2011/05/what-is-dll-hell-in-net.html)  In short, the dll hell problem is solved in .NET by signing the shared assemblies with strong name. [Please follow this article, to understand the process of strong naming an assembly.](http://venkataspinterview.blogspot.com/2011/06/what-is-process-for-strong-naming.html) In dot net all the shared assemblies are usually in the GAC. GAC stands for Global Assembly Cache. The path for GAC is **C:\[OperatingSystemDirectory]\assembly**. For example on my computer the path is **C:\WINDOWS\assembly**. The image below shows the shared assemblies in the GAC.

|  |
| --- |
| http://3.bp.blogspot.com/-3h9wj4-LqZU/TeaQG75xBjI/AAAAAAAAAGo/j6cSV-ZZHsA/s1600/GAC.png |

Only strong named assemblies can be copied into GAC. Strong named assemblies in .NET has 4 pieces in its name as listed below.  
**1.** Simple Textual Name  
**2.** Version Number  
**3.** Culture  
**4.** Public Key Token  
  
All these four pieces put together, is called as the fully qualified name of the assembly. In the GAC image above **Accessibility assembly**has a **version of 2.0.0.0**.  
   
**Now consider the example below:**  
**1**. I have 2 applications, **Application - A1** and **Application - A2** which relies on the shared assembly **Accessibility.dll (Version 2.0.0.0)** as shown in the image below.

|  |
| --- |
| http://4.bp.blogspot.com/-Wy2MvPARtTA/TeaROJ-kFYI/AAAAAAAAAGs/kQODaRaugfQ/s1600/Dll+Hell+Solved.png |

**2.** Now, I have a latest version of **Application - A2** available on the internet.  
**3.** I download the latest version of **A2** and install it on my machine.   
**4.** This new installation copies a newer version of **Accessibility.dll**into the GAC with version **3.0.0.0.**  
**5.** So, in the GAC we now have **2 versions of Accessibility.dll.**   
**6.** **Application - A1** continues to use **Accessibility.dll (version 2.0.0.0)** and **Application - A2 uses Accessibility.dll (version 3.0.0.0)**   
**7.** So, now the assemblies are able to reside side by side in the GAC. For this reason dot net assemblies are also said to be supporting **side by side execution.  
  
If you can improve this answer further, please feel free to do so by submitting the form below. Thank you very much for your contribution.**