Managed and Unmanaged Code in .NET

Back to: [C#.NET Tutorials For Beginners and Professionals](https://dotnettutorials.net/course/csharp-dot-net-tutorials/)

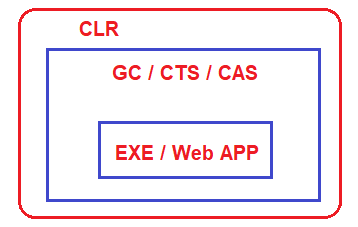
**Managed and Unmanaged Code in .NET**

In this article, I am going to discuss the **Managed and Unmanaged Code in C#.NET**. Please read our previous article where we discussed [**Common Language Specification (CLS)**](https://dotnettutorials.net/lesson/common-language-specification/) in detail. At the end of this article, you will understand what are Managed Code and Unmanaged code in C# and how they are executed in .NET Application in detail.

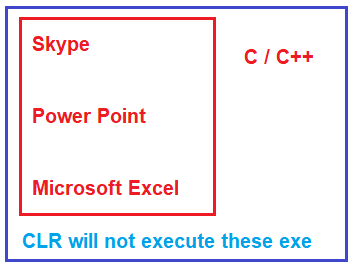
**Understanding the Managed and Unmanaged Code in C#.NET Application:**

Whenever you create any exe (i.e. console application, windows application, class library project, etc.) or web application (i.e. ASP.NET MVC, Web API, ASP.NET, etc.) in .NET Framework using visual studio and using any .NET supported programming language such as C#, VB, etc., then these applications are run completely under the control of CLR (Common Language Runtime).

That means, if your applications having unused objects, then CLR will clean those objects using Garbage Collector. If your application wants to communicate with other applications, then it will make sure that CTS (Common Type System) and CLS are available. CLR uses CAS (Code Access Security) if your application has the proper rights to execute4. The CLR will load your application and unload your application, etc. So, for better understanding, please have a look at the following image.



Now, let say, you have also used other third-party exe in your .NET application like Skype, PowerPoint, Microsoft Excel, etc. These “exe” are not made in dot net, they are made using other programming languages such as C, C++.



When you use these “exe” in your application, then these are not run by CLR. Even though you are running these “exe” in dot net applications, they are going to run under their own environment. For example, if one exe is developed using C or C++, then that exe will run under the C or C++ runtime environment. In the same line, if the exe is created using VB6, then it is going to run under the VB6 runtime environment.

**What exactly is the managed and unmanaged code in .NET?**

The codes which run under the complete control of CLR are called Managed Code in .NET. These kinds of code (Managed code in C#) are run by dot net runtime environment. If the dot net framework is not installed or if dot net runtime is not available, then these kinds of codes are not going to be executed. CLR will provide all the facilities and features of .NET to the managed code execution like Language Interoperability, Automatic memory management, Exception handling mechanism, code access security, etc.

On the other hand, Skype, PowerPoint, Microsoft Excel does not require dot net runtime, they run under their own environment. So, in short, the code (exe, web app) which not run under the control of CLR is called unmanaged code in .NET. CLR will not provide any facilities and features of .NET to the unmanaged code in C# execution like Language Interoperability, Automatic memory management, Exception handling mechanism, code access security, etc.

What is a .Net Assembly?

The .NET assembly is the standard for components developed with the Microsoft.NET. Dot NET assemblies may or may not be executable, i.e., they might exist as the executable (.exe) file or dynamic link library (DLL) file. All the .NET assemblies contain the definition of types, versioning information for the type, meta-data, and manifest. The designers of .NET have worked a lot on the component (assembly) resolution.

An assembly can be a single file or it may consist of the multiple files. In the case of multi-file, there is one master module containing the manifest while other assemblies exist as non-manifest modules. A module in .NET is a subpart of a multi-file .NET assembly. Assembly is one of the most interesting and extremely useful areas of .NET architecture along with reflections and attributes.

.NET supports three kinds of assemblies:

1. private
2. shared
3. satellite

**Private Assembly**

Private assembly requires us to copy separately in all application folders where we want to use that assembly’s functionalities; without copying, we cannot access the private assembly features and power. Private assembly means every time we have one, we exclusively copy into the BIN folder of each application folder.

**Public Assembly**

Public assembly is not required to copy separately into all application folders. Public assembly is also called Shared Assembly. Only one copy is required in system level, there is no need to copy the assembly into the application folder.

Public assembly should install in GAC.

Shared assemblies (also called strong named assemblies) are copied to a single location (usually the Global assembly cache). For all calling assemblies within the same application, the same copy of the shared assembly is used from its original location. Hence, shared assemblies are not copied in the private folders of each calling assembly. Each shared assembly has a four-part name including its face name, version, public key token, and culture information. The public key token and version information makes it almost impossible for two different assemblies with the same name or for two similar assemblies with a different version to mix with each other.

Learn [How to Create and Use a Shared Assembly](https://www.c-sharpcorner.com/uploadfile/5eae74/creating-and-using-shared-assembly/)

**GAC (Global Assembly Cache)**

When the assembly is required for more than one project or application, we need to make the assembly with a strong name and keep it in GAC or in the Assembly folder by installing the assembly with the GACUtil command.

**Satellite Assembly**

Satellite assemblies are used for deploying language and culture-specific resources for an application.

Common Language Runtime in .NET

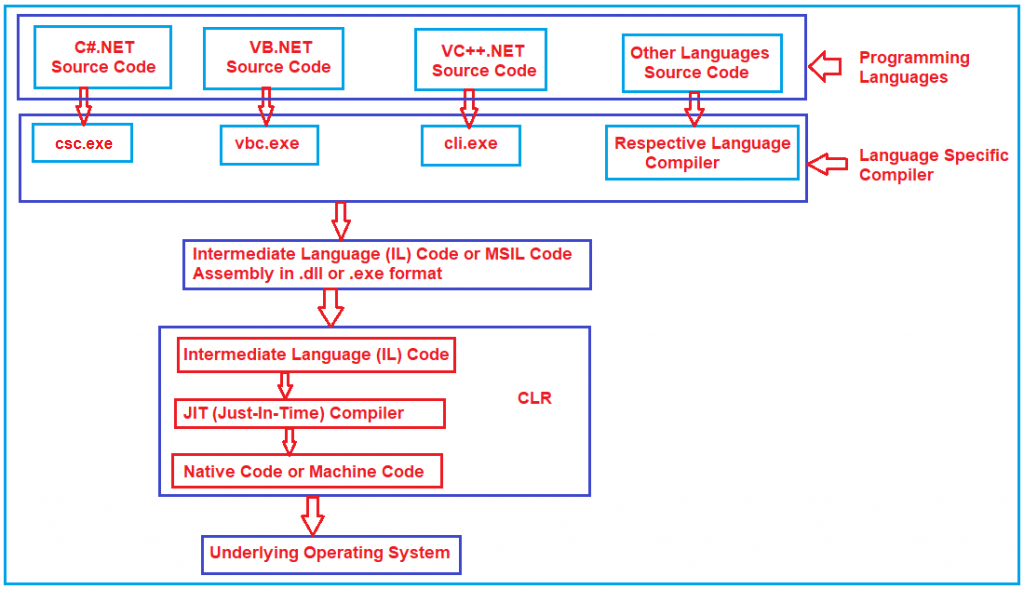
Back to: [C#.NET Tutorials For Beginners and Professionals](https://dotnettutorials.net/course/csharp-dot-net-tutorials/)

**Common Language Runtime (CLR) in C#.NET:**

In this article, I am going to discuss the **Common Language Runtime (CLR) in .NET Framework**. Please read our previous article before proceeding to this article where we gave a brief introduction to the [**DOT NET Framework**](https://dotnettutorials.net/lesson/dotnet-framework/). At the end of this article, you will understand all about CLR in C# with examples. But before understanding CLR in .NET, let us first understand how a .NET application is compiled and run.

**How is a .NET application Compiled and Run?**

In order to understand how exactly a .NET Application is compiled and run, please have a look at the following image.



First, the developer has to write the code using any dot net supported programming languages such as C#, VB, J#, etc. Then the respective language compiler will compile the program code and generate something called **Microsoft Intermediate language (MSIL) or Intermediate language (IL)** code. For example, if the programming language is **C#**, then the compiler is **CSC** and if the programming language is **VB**, then the compiler will be **VBC**. This **Intermediate Language (IL)** code is half compiled code i.e. partially compiled code and cannot be executed directly by the Operating System. To execute this **Microsoft Intermediate language (MSIL) or Intermediate language (IL)** code on your machine, the .NET Framework provides something called **Common Language Runtime (CLR)** which takes the responsibility to execute your **Microsoft Intermediate language (MSIL) or Intermediate language (IL)**Code.

The CLR takes the IL (Intermediate Language) code and gives it to something called JIT (Just-in-Time) Compiler. The JIT compiler reads each and every line of the IL code and converts it to machine-specific instructions (i.e. into binary format) which can be executed by the underlying Operating System.

**What is Intermediate Language (IL) Code in .NET Framework?**

The Intermediate Language or [**IL code**](https://dotnettutorials.net/lesson/intermediate-language/) in .NET Framework is a half compiled or partially compiled or CPU-independent partially compiled code and this code can not be executed by Operating System.

**Why Partial Compiled Code or why not fully compiled Code?**

As a developer, you may be thinking about why the respective language compiler generates partially compiled code or why not fully compiled code i.e. machine code or binary code in .NET Framework. The reason is very simple. We don’t know in what kind of environment .NET Code is going to be run (for example, Windows XP, Windows 7, Windows 10, Windows Server, etc.). In other words, we don’t know what operating system is going to run our application; we also don’t know the CPU configuration, Machine Configuration, Security Configuration, etc. So, the Microsoft Intermediate language (MSIL) or Intermediate language (IL) code is partially compiled, and at runtime, this Microsoft Intermediate language (MSIL) or Intermediate language (IL) code is compiled to machine-specific instructions or you can say binary code using environmental properties such as Operating System, CPU, Machine Configuration, etc. by the CLR in .NET Framework.

**Common Language Runtime (CLR) in .NET Framework:**

CLR is the heart of the .NET Framework and it contains the following components.

1. Security Manager
2. JIT Compiler
3. Memory Manager
4. [**Garbage Collector**](https://dotnettutorials.net/lesson/garbage-collector/)
5. [**Exception Manager**](https://dotnettutorials.net/lesson/exception-handling-csharp/)
6. [**Common Language Specification (CLS)**](https://dotnettutorials.net/lesson/common-language-specification/)
7. [**Common Type System (CTS)**](https://dotnettutorials.net/lesson/common-type-system/)

Let us discuss what each of these components does in detail.

**Security Manager:**

There are basically two components to manage security. They are as follows:

1. **CAS (Code Access Security)**
2. **CV (Code Verification)**

These two components are basically used to check the privileges of the current user that the user is allowed to access the assembly or not.  The Security Manager also checks what kind of rights or authorities this code has and whether it is safe to be executed by the Operating System. So, basically, these types of checks are maintained by the Security Manager in .NET Application.

**JIT Compiler:**

The JIT (Just-In-Time) Compiler is responsible for Converting the MSIL code into native code (Machine Code or Binary code) that is executed by the Operating System. The native code (Machine Code or Binary code) is directly understandable by the system hardware. JIT compiles the code just before the execution and then saves this translation in memory.

**Memory Manager:**

The memory manager component of CLR in the .NET Framework allocates the necessary memory for the variables and objects that are to be used by the application.

**Garbage Collector:**

When a dot net application runs, lots of objects are created. At a given point in time, it is possible that some of those objects are not used by the application. So, [**Garbage Collector in .NET Framework**](https://dotnettutorials.net/lesson/garbage-collector/) is nothing but is a **Small Routine** or you can say it’s a **Background Process Thread** that runs periodically and try to identify what objects are not being used currently by the application and de-allocates the memory of those objects.

**Exception Manager:**

The [**Exception Manager**](https://dotnettutorials.net/lesson/exception-handling-csharp/) component of CLR in the .NET Framework redirects the control to execute the catch or finally blocks whenever an exception has occurred at runtime.

**Common Type System (CTS) in .NET Framework:**

The .NET Framework supports many programming languages such as C#, VB.NET, J#, etc. Every programming language has its own data type. One programming language data type cannot be understood by other programming languages. But, there can be situations where we need to communicate between two different programming languages. For example, we need to write code in the VB.NET language and that code may be called from C# language. In order to ensure smooth communication between these languages, the most important thing is that they should have a [**Common Type System (CTS**](https://dotnettutorials.net/lesson/common-type-system/)) which ensures that data types defined in two different languages get compiled to a common data type.

CLR in .NET Framework will execute all programming language’s data types. This is possible because CLR having its own data types which are common to all programming languages. At the time of compilation, all language-specific data types are converted into CLR’s data type. This data type system of CLR is common to all .NET Supported Programming languages and this is known as the [**Common Type System**](https://dotnettutorials.net/lesson/common-type-system/)(CTS).

**CLS (Common Language Specification) in .NET Framework:**

[**CLS (Common Language Specification)**](https://dotnettutorials.net/lesson/common-language-specification/)is a part of CLR in the .NET Framework. The .NET Framework supports many programming languages such as C#, VB.NET, J#, etc. Every programming language has its own syntactical rules for writing the code which is known as **language specification**. One programming language syntactical rules (language specification) cannot be understood by other programming languages. But, there can be situations where we need to communicate between two different programming languages. In order to ensure smooth communication between different .NET Supported Programming Languages, the most important thing is that they should have **Common Language Specifications**which ensures that language specifications defined in two different languages get compiled to a Common Language Specification.

CLR in .NET Framework will execute all programming language’s code. This is possible because CLR having its own language specification (syntactical rules) which are common to all .NET Supported Programming Languages. At the time of compilation, every language compiler should follow this language specification of CLR and generate the MSIL code. This language specification of CLR is common for all programming languages and this is known as [**Common Language Specifications (CLS)**](https://dotnettutorials.net/lesson/common-language-specification/)**.**

Intermediate Language (ILDASM & ILASM) Code in C#

Back to: [C#.NET Tutorials For Beginners and Professionals](https://dotnettutorials.net/course/csharp-dot-net-tutorials/)

**Intermediate Language (ILDASM & ILASM) in C#.NET**

**ILDASM** stands for Intermediate Language disassembler and **ILASM** stands for Intermediate language assembler. As part of this article, we are going to discuss the following pointers and at the end of this article, you will understand all about Intermediate Language (IL Code) in C#.

1. **What happens when we compile a .NET Application?**
2. **Understanding the Intermediate Language (IL Code) in C#?**
3. **What are ILDASM and ILASM?**
4. **How to view the Intermediate Language code in C#?**
5. **What is Manifest?**
6. **How to export the Intermediate Language code to a text file?**
7. **How to rebuild an assembly from a text file which contains manifest and IL?**

**What happens when we compile a .NET Application?**

When we compile any .NET application. it will generate an assembly with the extension of either a .DLL or an .EXE. Irrespective of whether it is a .DLL or .EXE, an assembly consists of two things i.e. **Manifest and Intermediate language**. Let us understand how the Intermediate Language and Manifest look like in .NET Framework with an example.

**Understanding Intermediate Language (ILDASM and ILASM) Code in C#:**

In order to understand Intermediate Language Code (ILDASM and ILASM) in C#, let us create a simple console application. Once you create the console application, please modify the Program class as shown below.

**using** *System;*

**namespace** *ILDASMDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Console.WriteLine**(**"Understanding ILDASM and ILASM"**)**;

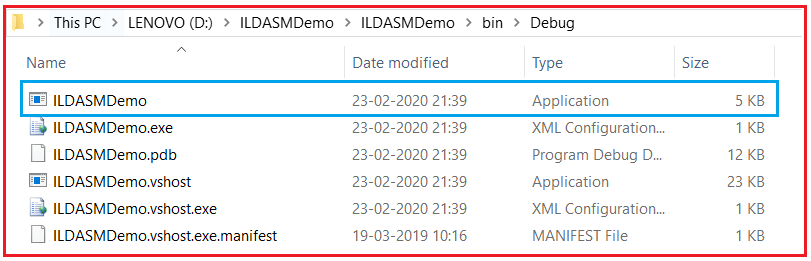
Console.Read**()**;

**}**

**}**

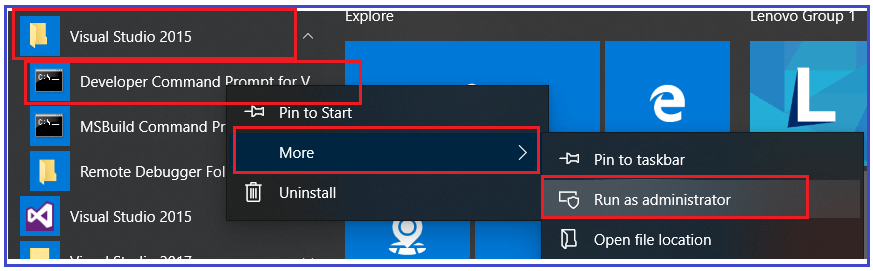
**}**

Now, build the application. Once you build the application, the above source code is compiled and intermediate language code generated and packaged into an assembly. In order to see the assembly, just right-click on the Project and select **Open Folder in File Explorer** option and then go to the **bin => Debug** folder and you should see an assembly with .exe extension as shown in the below image as it is a console application.

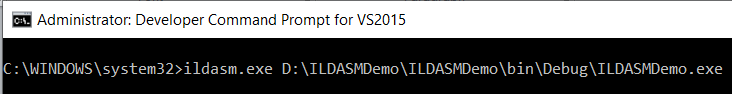


**How to view the Intermediate Language Code in .NET Framework?**

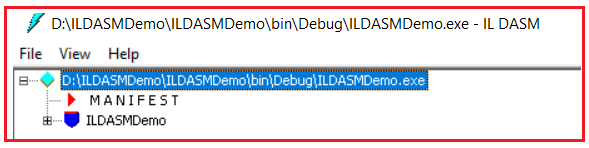
The .NET framework provides a nice tool called **ILDASM (Intermediate Language DisAssember)** to view the code of the intermediate language in C#.NET. In order to use the ILDASM tool, you need to follow the below steps. Open Visual Studio Command Prompt in Administrator mode as shown in the below image.



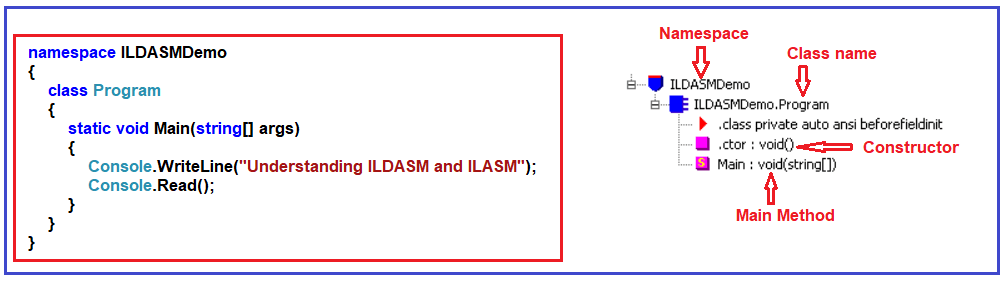
Once you open the visual studio command prompt in administrative mode, then type the “**Ildasm.exe C:\YourDirectoryPath\YourAssembly.exe**” command and press enter. Here, you need to provide the exe path where your exe is generated. My exe is generated in the path “**D:\ILDASMDemo\ILDASMDemo\bin\Debug\ILDASMDemo.exe**”, so I execute the following code in the command prompt:



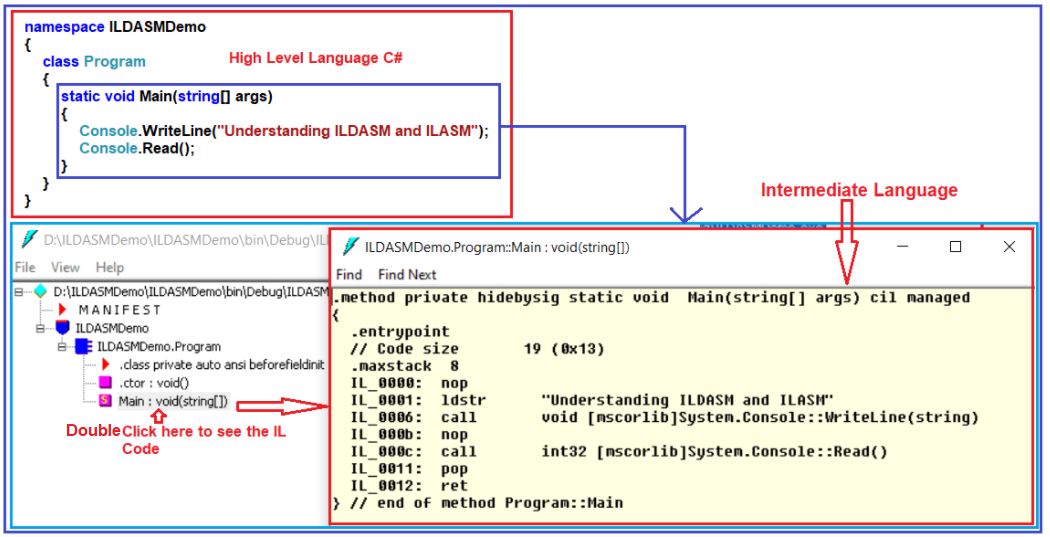
Once you type the above command and press enter, it should open the following ILDASM window.



As you can see, the assembly is consists of two things (**Manifest and Intermediate language**). Let us first discuss the intermediate language code and then we will discuss what Manifest is. Now, let us expand the ILDASMDemo and compare it with our code. For better understanding, please have a look at the below image. There is a constructor in ILDASM and this is because by default the .NET Framework provides a default constructor when there is no constructor in your class. You can also the Main method in the intermediate language code/

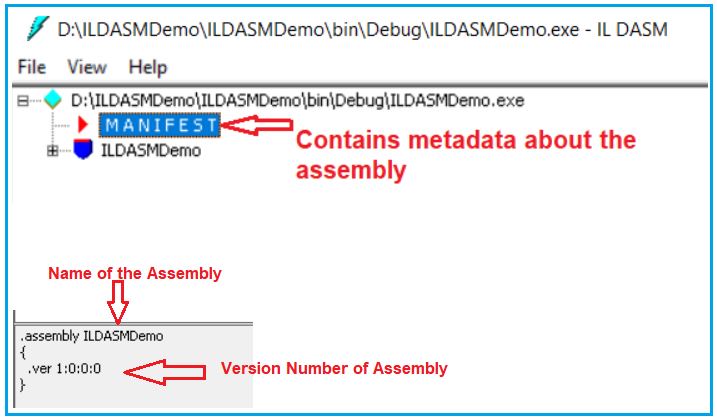


Now, double click on the Main method on the ILDASM window to see the intermediate language generated for the Main method as shown below.



**What is Manifest?**

Manifest contains metadata about the assembly like the name of the assembly, the version number of the assembly, culture, and strong name information as shown in the below image.

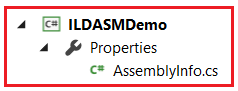


Metadata also contains information about the referenced assemblies. Each reference includes the dependent assembly’s name, assembly metadata (version, culture, operating system, and so on), and public key, if the assembly is strongly named.

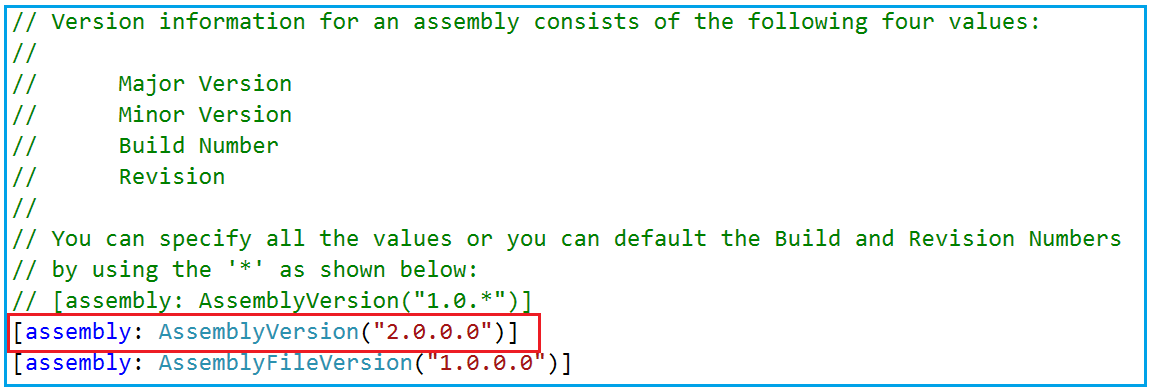
**How to change Assembly info?**

It is also possible to change or modify some of the information in the assembly manifest using attributes. For example, if you want to modify the version number, then you need to follow the below steps.

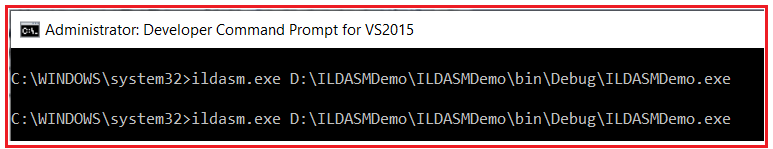
Open **AssemblyInfo.cs** class file which is present under the **Properties** folder as shown below. Every project in .NET has a properties folder.



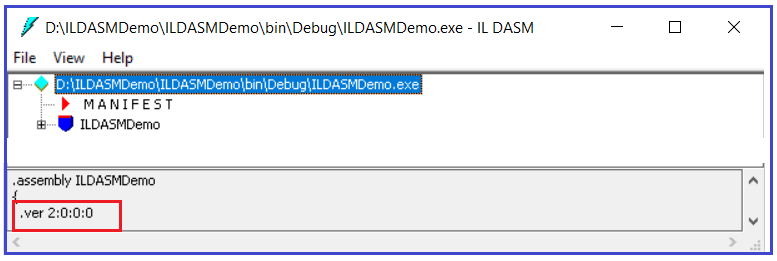
In this file, you will find one attribute called the **AssemblyVersion** attribute, which is by default set to 1.0.0.0. Now, change this value to 2.0.0.0 as shown below.



Now, rebuild the solution. But before that close the ILDASM window otherwise you will get an error. Once you rebuild the solution then open the assembly using the same **ILDASM.exe** in the command prompt as shown below.



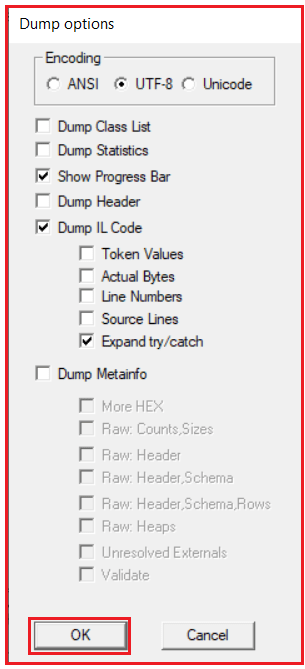
Once you execute the above command, it should open the assembly. At the bottom, you can find the updated version number of the assembly as expected as shown below.



**How to export the Intermediate Language code to a text file?**

If you want to export or save the Intermediate Language code into a text file, then you need to follow the below steps.

Select **File Menu** Option from the **ILDASM tool** and then select **Dump** and you will see “**Dump Options Window**” and click on the **OK** button on the “Dump Options Window” as shown below.



Now you need to enter the file name as per your choice. I am entering the file name as **MyFile** and save it to the **D:** drive. Now navigate to D: drive in windows explorer and you should see MyFile.il Now, open MyFile.il with notepad and you should see assembly metadata and IL code.

**How to rebuild an assembly from a text file which contains manifest and IL?**

If you want to rebuild an assembly from IL code then you need to use a tool called ILASM.exe. So, let’s go and create an assembly from the file (MyFile.il) that we just save. In order to rebuild an assembly, please follow the below steps.

Type the following command in “Visual Studio Command Prompt” and press enter  
         **ILASM.exe D:\MyFile.il**  
Now navigate to D: drive in windows explorer and you should see MyFile.exe. So, in short, we use **ILASM.exe** (Intermediate Language Assembler) to reconstruct an assembly from a text file that contains manifest and IL.

Garbage Collector in .NET Framework

Back to: [C#.NET Tutorials For Beginners and Professionals](https://dotnettutorials.net/course/csharp-dot-net-tutorials/)

**Garbage Collector in .NET Framework**

In this article, I am going to discuss the **Garbage Collector in .NET** Framework with Examples. Please read our previous article where we discussed [**Managed and Unmanaged Code in .NET**](https://dotnettutorials.net/lesson/managed-and-unmanaged-code/) Application. At the end of this article, you will understand what is Garbage Collector in .NET Framework and how does it work? As part of this article, we are going to discuss the following pointers in detail.

1. **What is Garbage Collector in .NET?**
2. **What are the different Generations of Garbage collectors?**
3. **How using a destructor in a class we end up in a double garbage collector loop?**
4. **How we can solve the double loop problems using finalize dispose patterns?**

**What is Garbage Collector in .NET Application?**

When a dot net application runs, lots of objects are created. At a given point in time, it is possible that some of those objects are not used by the application. Garbage Collector in .NET Framework is nothing but is a Small Routine or you can say it’s a Background Process Thread that runs periodically and try to identify what objects are not being used currently by the application and de-allocates the memory of those objects.

So, Garbage Collector is nothing but, it is a feature provided by CLR which helps us to clean or destroy unused managed objects. By cleaning or destroying those unused managed objects, basically reclaims the memory.

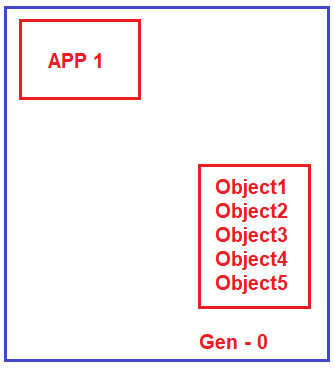
**Note:** The Garbage Collector will destroy only the unused managed objects. It does not clean unmanaged objects. If you want to learn what exactly is managed and unmanaged objects, please read our previous article.

**Garbage Collector Generations in .NET:**

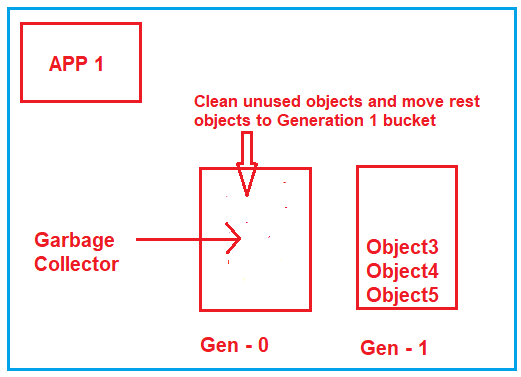
Let us understand what Garbage Collector Generations are and how does it affect Garbage Collector performance? There are three generations. They are Generation 0, Generation 1, and Generation 2.

**Understanding Generation 0, 1, and 2:**

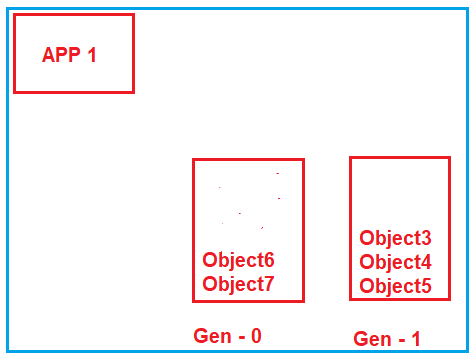
Let say you have a simple application called App1. As soon as the application started it creates 5 managed objects. Whenever any new objects (fresh objects) are created, they are moved into a bucket called Generation 0. For better understanding please have a look at the following image.



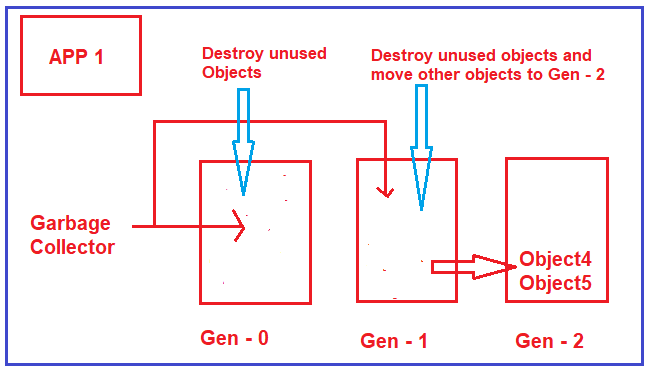
We know our hero Mr. Garbage Collector runs continuously as a background process thread to check whether there are any unused managed objects so that it reclaims the memory by cleaning those objects. Now, let say two objects (Object1 and Object2) are not needed by the application. So, Garbage Collector will destroy these two objects (Object1 and Object2) and reclaims the memory from Generation 0 bucket. But the remaining three objects (Object3, Object4, and Object5) are still needed by the application. So, the Garbage collector will not clean those three objects. What Garbage Collector will do is, he will move those three managed objects (Object3, Object4, and Object5) to Generation 1 bucket as shown in the below image.



Now, let say your application creates two more fresh objects (Object6 and Object7). As fresh objects, they should be created in Generation 0 bucket as shown in the below image.



Now, again Garbage Collector runs and it comes to Generation 0 bucket and checks which objects are used. Let say both objects (Object6 and Object7) are unused by the application, so it will remove both the objects and reclaims the memory. Now, it goes to the Generation 1 bucket, and checks which object are unused. Let say Object4 and Object5 are still needed by the application while object3 is not needed. So, what Garbage Collector will do is, it will destroy Object3 and reclaims the memory as well as it will move Objec4 and Object5 to Generation 2 bucket which is shown in the below image.



**What are Generations?**

Generations are nothing but, will define how long the objects are staying in the memory. Now the question that should come to your mind is why do we need Generations?

**Why do we need Generations?**

Normally, when we are working with big applications, they can create thousands of objects. So, for each of these objects, if the garbage collector goes and checks if they are needed or not, it’s really pain or it’s a bulky process. By creating such generations what it means if an object in Generation 2 buckets it means the Garbage Collector will do fewer visits to this bucket. The reason is, if an object move to Generation 2, it means it will stay more time in the memory. It’s no point going and checking them again and again.

So, in simple words, we can say that Generations 0, 1, and 2 will helps to increase the performance of the Garbage Collector. The more the objects in Gen 0, the better the performance and the more the memory will be utilized in an optimal manner.

**How using a destructor in a class we end up in a double garbage collector loop?**

As we already discussed garbage collectors will only clean up the managed code. In other words, for any kind of unmanaged code, for those codes to clean up has to be provided by unmanaged code, the garbage collector does not have any control over them to clean up the memory.

For example, let say you have a class called MyClass in VB6, then you have to expose some function let say CLeanUp() and in that function, you have to write the logic to clean up the unmanaged code. From your dot net code, you simply need to call that method (CLeanUp()) to initiate the clean-up.

The point, or the section from where you would like to call the Clean-Up is the destructor of a class. This looks to be the best place to write the clean-up code. But, there is a big problem associated with it when you write clean-up in a destructor. Let us understand what the problem is?

When you define a destructor in your class, the Garbage Collector before destroying the object, will go and ask the question to the class, do you have a destructor, if you have a destructor, then move the object to the next generation bucket. In other words, it will not clean up the object having a destructor at that moment itself even though it is not used. So, it will wait for the destructor to run, and then it will go and clean up the object. Because of this, you will find more objects in generation 1 and Generation 2 as compared to Generation 0.

**Example: Using Destructor**

Please create a console application and then copy and paste the following code in it in the Program class.

**using** *System;*

**namespace** *GCDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**for(int** i = 0; i **<**= 1000000; i++**)**

**{**

MyClass obj = new MyClass**()**;

**}**

Console.Read**()**;

**}**

**}**

**public** **class** MyClass

**{**

~MyClass**()**

**{**

//Unmanaged code clean up

**}**

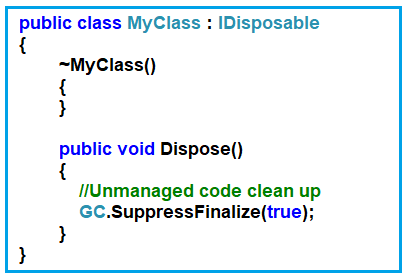
**}**

**}**

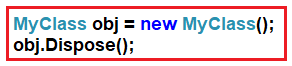
So, if you writing the clean-up code in your destructor, then you will end up creating more objects in Generation 1 and Generation 2 which means you are not utilizing the memory properly.

**How to Overcome the above Problem?**

This problem can be overcome by using something called finalize dispose pattern. In order to implement this, your class should implement the IDisposable interface and provide the implementation for the Dispose method. Within the Dispose method, you need to write the clean-up code for unmanaged objects and in the end, you need to call GC.SuppressFinalize(true) method by passing true as the input value. This method tells suppress any kind of destructor and just go and clean up the objects. For better understanding, please have a look at the following image.



Once you have used to object, then you need to call the Dispose method so that the double garbage collector loop will not happen as shown below.



**The complete code is given below.**

**using** *System;*

**namespace** *GCDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**for(int** i = 0; i **<**= 1000000; i++**)**

**{**

MyClass obj = new MyClass**()**;

obj.Dispose**()**;

**}**

Console.Read**()**;

**}**

**}**

**public** **class** MyClass : IDisposable

**{**

~MyClass**()**

**{**

**}**

**public** **void** Dispose**()**

**{**

//Unmanaged code clean up

GC.SuppressFinalize**(true)**;

**}**

**}**

**}**

Now, the question that should come to your mind is why the destructor is there? The reason is as a developer you may forget to call the Dispose method once you use the object. In that case, the destructor will invoke and it will go and clean up the object.

# Common Type System in .NET Framework

Back to: [C#.NET Tutorials For Beginners and Professionals](https://dotnettutorials.net/course/csharp-dot-net-tutorials/)

## ****Common Type System (CTS) in .NET Framework****

In this article, I am going to discuss the **Common Type System in .NET Framework**. Please read our previous article, where we discussed the [**Intermediate Language in .NET Framework**](https://dotnettutorials.net/lesson/intermediate-language/) with Examples. At the end of this article, you will understand what is Common Type System (CTS) in C# and how CTS in .NET works?

##### ****What is**** ****the Common Type System in .NET Framework?****

The .NET Framework supports many programming languages such as C#, VB.NET, J#, etc. Every programming language has its own data type. One programming language data type cannot be understood by other programming languages. But, there can be situations where we need to communicate between two different programming languages. For example, we need to write code in the VB.NET language and that code may be called from C# language. In order to ensure smooth communication between these languages, the most important thing is that they should have a Common Type System (CTS) which ensures that data types defined in two different languages get compiled to a common data type.

CLR in .NET Framework will execute all programming language’s data types. This is possible because CLR having its own data types which are common to all programming languages. At the time of compilation, all language-specific data types are converted into CLR’s data type. This data type system of CLR is common to all .NET Supported Programming languages and this is known as the Common Type System (CTS).

##### ****Example: Common Type System in .NET Framework****

Let us understand Common Type System (CTS) in .NET Framework with an example. What we are going to do is, we will create two applications. One Application using C# Language and the other one is using VB.NET Language. And then we will try to see the IL code of both of these applications and then we will try to see how the CTS looks like.

##### ****Understanding CTS in .NET:****

Here we are going to create two class library projects. One class library project using C# language and the other class library project using the VB language.

##### ****Creating C# Class Library Project:****

Create a class library project with the name **CsharpClassLibrary** and using the **C#** programming language. Once you create the C# class library project then add a class called **Calculator** and then copy and paste the following code in it.

**namespace** *CsharpClassLibrary*

**{**

**public** **class** Calculator

**{**

**public** **int** Calculate**()**

**{**

**int** a = 10, b = 20;

**int** c = a + b;

**return** c;

**}**

**}**

**}**

##### ****Creating VB Class Library Project:****

To the same CsharpClassLibrary solution, let us add another class library project with the name as **VBClassLibrary** but using **VB** as the programming language. Once you created the VB Class library project then add a class called **Calculator** to this project and copy and paste the following code in it.

Public Class Calculator

Public Function Calculate**()** As Integer

Dim a As Integer = 10

Dim b As Integer = 10

Dim c As Integer

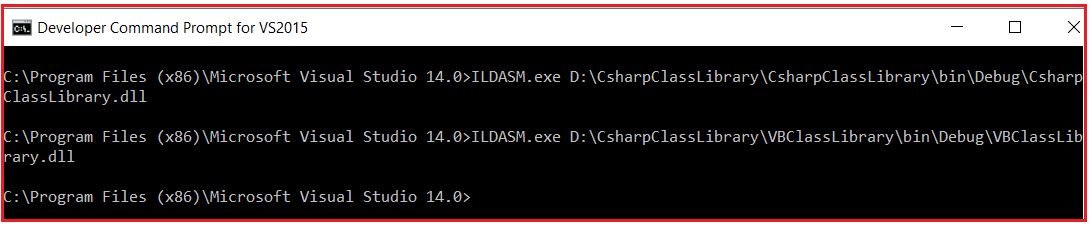
c = a + b

Return c

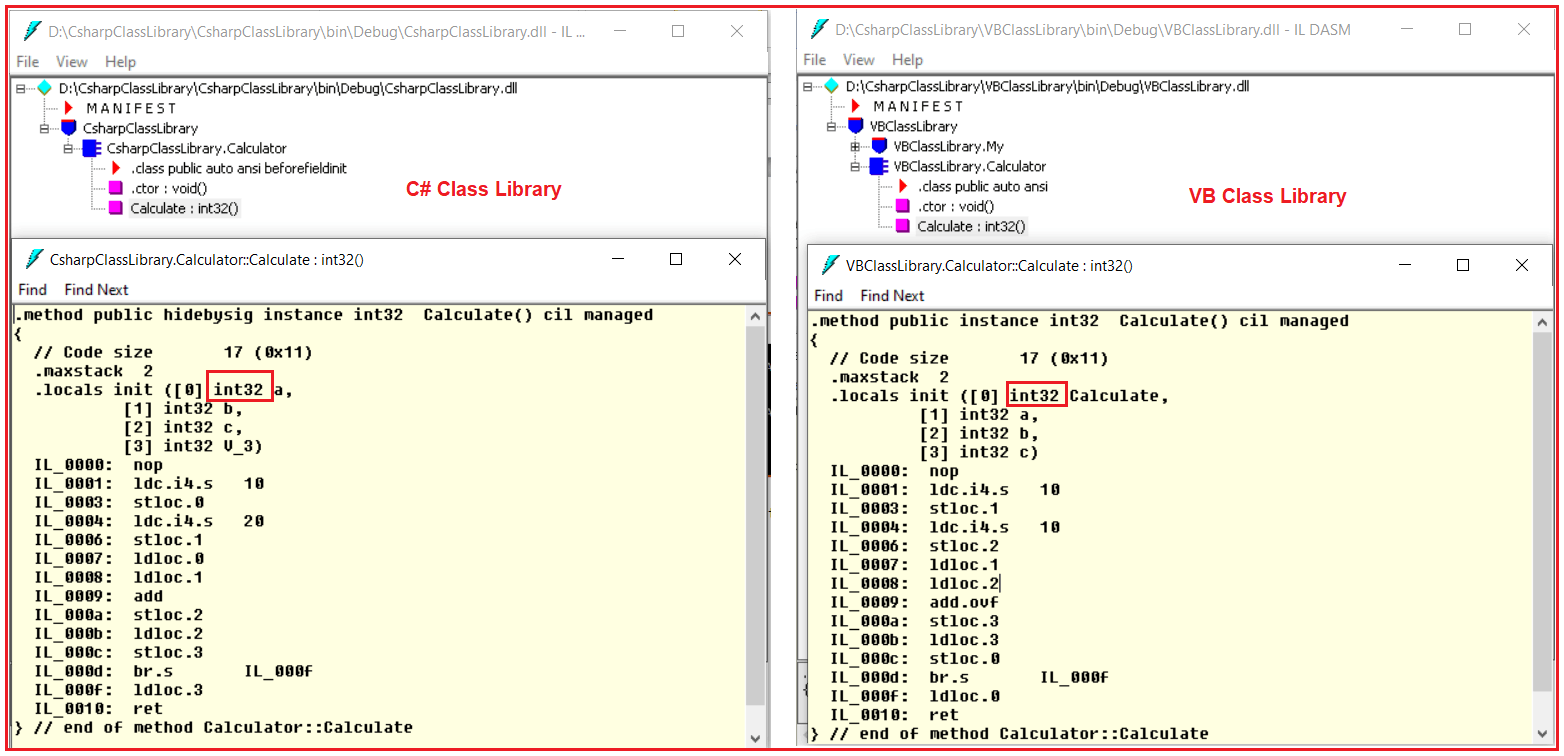
End Function

End Class

Now, build the solution which should generate the respected .dlls. In our previous article, we discussed [**how to use the ILDASM tool to see the IL code**](https://dotnettutorials.net/lesson/intermediate-language/) in detail. So, let us open Visual Studio Command Prompt in Administrative mode and run two instances of the **ILDASM.exe** command i.e. one for VB dot net DLL file and the other one for the C# DLL file as shown in the below image.



Now, let us open the IL code of the Calculate method of both the class library project as shown in the below image. Please have a look at the integer variable in the IL code which is int32 in this case. In the C# class library project, we use int as the data type to declare variables a, b and c whereas in the VB class library project we use Integer as the data type to declare the variables a, b, and c. At the end of the day, both the data types are compiled to a common data type i.e. int32.



Whether you write the code in VB.NET or in C#.NET, if it following the dot net rules or specifications, at the end of the day it is compiled into a Common Type System (CTS) in .NET Framework which is common for all .NET Supported Programming Languages.

Common Language Specification in .NET Framework

Back to: [C#.NET Tutorials For Beginners and Professionals](https://dotnettutorials.net/course/csharp-dot-net-tutorials/)

**Common Language Specification (CLS) in .NET Framework**

In this article, I am going to discuss **Common Language Specification (CLS) in .NET Framework**. The Common Language Specification is also called as CLS in .NET Framework**.** Please read our previous article, where we discussed the [**Common Type System (CTS) in .NET Framework**](https://dotnettutorials.net/lesson/common-type-system/). At the end of this article, you will understand what is Common Language Specification (CLS) in C# and how CLS works in .NET Framework?

**What is Common Language Specification (CLS) in .NET Framework?**

CLS (Common Language Specification) is a part of CLR in the .NET Framework. The .NET Framework supports many programming languages such as C#, VB.NET, J#, etc. Every programming language has its own syntactical rules for writing the code which is known as a language specification. One programming language syntactical rules (language specification) cannot be understood by other programming languages. But, there can be situations where we need to communicate between two different programming languages. In order to ensure smooth communication between different .NET Supported Programming Languages, the most important thing is that they should have Common Language Specifications which ensures that language specifications defined in two different languages get compiled to a Common Language Specification.

CLR in .NET Framework will execute all programming language’s code. This is possible because CLR having its own language specification (syntactical rules) which are common to all .NET Supported Programming Languages. At the time of compilation, every language compiler should follow this language specification of CLR and generate the MSIL code. This language specification of CLR is common for all programming languages and this is known as Common Language Specifications (CLS).

In order to understand this concept, what we will do here is, we will violate the common language specifications and then we will see what happens.

**Example to understand CLS in .NET Framework:**

As we know C# is case sensitive whereas VB is not case sensitive. That means in C#, you can declare the same member name multiple times with case differences but it is not possible in VB. So, let us create two class library projects. One using C# and the other one is using VB Programming Language. Then we will try to consume the C# class library project in the VB class library project.

**Creating a C# Class Library Project:**

Create a class library project with the name **CsharpClassLibrary** using the C# programming language. Once you create the C# class library project then add a class called **Calculator** and then copy and paste the following code in it. As you can see we have created two methods with the same name but with case differences. One method starts with capital C while the other one starts with a small c.

**namespace** *CsharpClassLibrary*

**{**

**public** **class** Calculator

**{**

**public** **int** Calculate**()**

**{**

**int** a = 10, b = 20;

**int** c = a + b;

**return** c;

**}**

**public** **int** calculate**()**

**{**

**int** a = 10, b = 20;

**int** c = a + b;

**return** c;

**}**

**}**

**}**

**Creating VB Class Library Project:**

To the same CsharpClassLibrary solution, let us add another class library project with the name as VBClassLibrary but using VB as the programming language. Here, in this project we want to use the C# class library project, so first add a reference to the **CsharpClassLibrary** project. Then create a class with the name TestClass and copy-paste the following code in it.

Imports CsharpClassLibrary

Public Class TestClass

Public Sub TestMethod**()**

Dim obj As New Calculator**()**

obj.Calculate**()**

End Sub

End Class

Now, when you try to build the VB Class Library project, you will get the below error. This is because VB is not case sensitive and it found two methods with the same name. That means we are violating the Common Language Specifications in the C# code.

**‘Calculate’ is ambiguous because multiple kinds of members with this name exist in class ‘Calculator’.**

Now, let us change the second method name to Calculate2 as shown below.

**namespace** *CsharpClassLibrary*

**{**

**public** **class** Calculator

**{**

**public** **int** Calculate**()**

**{**

**int** a = 10, b = 20;

**int** c = a + b;

**return** c;

**}**

**public** **int** Calculate2**()**

**{**

**int** a = 10, b = 20;

**int** c = a + b;

**return** c;

**}**

**}**

**}**

With the above changes in place, now, build the VB class library project and the build should succeed as expected. Now, you may have one question, how to check whether the code is CLSCompliant or not.

**How to check the code is CLS Compliant or not?**

In order to check whether your code is following the Common Language Specifications or not, first, you have to enable CLS Compliant in **AssemblyInfo.cs** file. So, go to the C# Class Library Project and open the **AssemblyInfo.cs** file which is present inside the **Properties** folder. Once you open the **AssemblyInfo.cs** class file, then follow the below 2 steps.

**Step1: Import the System namespace as**  
**using System;**

**Step2: Add the following CLSCompliant attribute at the bottom of this file and set its value to true**  
**[assembly: CLSCompliant(true)]**

With the above changes in place in the **AssemblyInfo.cs** file, now modify the Calculator class as shown below.

**namespace** *CsharpClassLibrary*

**{**

**public** **class** Calculator

**{**

**public** **int** Calculate**()**

**{**

**int** a = 10, b = 20;

**int** c = a + b;

**return** c;

**}**

**public** **int** calculate**()**

**{**

**int** a = 10, b = 20;

**int** c = a + b;

**return** c;

**}**

**}**

**}**

Now, when you build the C# Class Library Project you will get the following warning.

Common Language Specification (CLS) in C#.NET Framework

Common Language Runtime in .NET

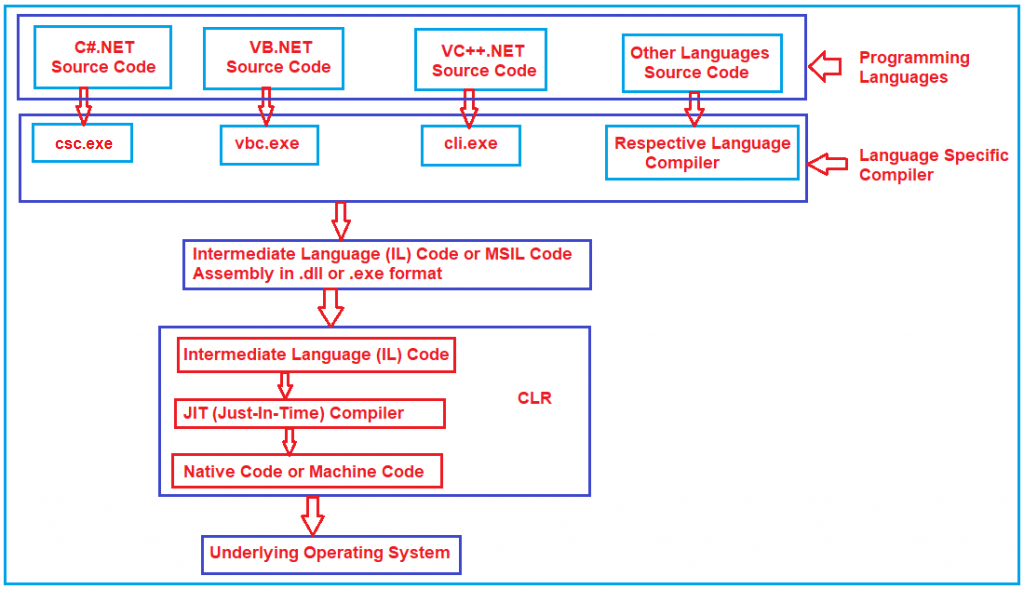
Back to: [C#.NET Tutorials For Beginners and Professionals](https://dotnettutorials.net/course/csharp-dot-net-tutorials/)

**Common Language Runtime (CLR) in C#.NET:**

In this article, I am going to discuss the **Common Language Runtime (CLR) in .NET Framework**. Please read our previous article before proceeding to this article where we gave a brief introduction to the [**DOT NET Framework**](https://dotnettutorials.net/lesson/dotnet-framework/). At the end of this article, you will understand all about CLR in C# with examples. But before understanding CLR in .NET, let us first understand how a .NET application is compiled and run.

**How is a .NET application Compiled and Run?**

In order to understand how exactly a .NET Application is compiled and run, please have a look at the following image.



First, the developer has to write the code using any dot net supported programming languages such as C#, VB, J#, etc. Then the respective language compiler will compile the program code and generate something called **Microsoft Intermediate language (MSIL) or Intermediate language (IL)** code. For example, if the programming language is **C#**, then the compiler is **CSC** and if the programming language is **VB**, then the compiler will be **VBC**. This **Intermediate Language (IL)** code is half compiled code i.e. partially compiled code and cannot be executed directly by the Operating System. To execute this **Microsoft Intermediate language (MSIL) or Intermediate language (IL)** code on your machine, the .NET Framework provides something called **Common Language Runtime (CLR)** which takes the responsibility to execute your **Microsoft Intermediate language (MSIL) or Intermediate language (IL)**Code.

The CLR takes the IL (Intermediate Language) code and gives it to something called JIT (Just-in-Time) Compiler. The JIT compiler reads each and every line of the IL code and converts it to machine-specific instructions (i.e. into binary format) which can be executed by the underlying Operating System.

**What is Intermediate Language (IL) Code in .NET Framework?**

The Intermediate Language or [**IL code**](https://dotnettutorials.net/lesson/intermediate-language/) in .NET Framework is a half compiled or partially compiled or CPU-independent partially compiled code and this code can not be executed by Operating System.

**Why Partial Compiled Code or why not fully compiled Code?**

As a developer, you may be thinking about why the respective language compiler generates partially compiled code or why not fully compiled code i.e. machine code or binary code in .NET Framework. The reason is very simple. We don’t know in what kind of environment .NET Code is going to be run (for example, Windows XP, Windows 7, Windows 10, Windows Server, etc.). In other words, we don’t know what operating system is going to run our application; we also don’t know the CPU configuration, Machine Configuration, Security Configuration, etc. So, the Microsoft Intermediate language (MSIL) or Intermediate language (IL) code is partially compiled, and at runtime, this Microsoft Intermediate language (MSIL) or Intermediate language (IL) code is compiled to machine-specific instructions or you can say binary code using environmental properties such as Operating System, CPU, Machine Configuration, etc. by the CLR in .NET Framework.

**Common Language Runtime (CLR) in .NET Framework:**

CLR is the heart of the .NET Framework and it contains the following components.

1. Security Manager
2. JIT Compiler
3. Memory Manager
4. [**Garbage Collector**](https://dotnettutorials.net/lesson/garbage-collector/)
5. [**Exception Manager**](https://dotnettutorials.net/lesson/exception-handling-csharp/)
6. [**Common Language Specification (CLS)**](https://dotnettutorials.net/lesson/common-language-specification/)
7. [**Common Type System (CTS)**](https://dotnettutorials.net/lesson/common-type-system/)

Let us discuss what each of these components does in detail.

**Security Manager:**

There are basically two components to manage security. They are as follows:

1. **CAS (Code Access Security)**
2. **CV (Code Verification)**

These two components are basically used to check the privileges of the current user that the user is allowed to access the assembly or not.  The Security Manager also checks what kind of rights or authorities this code has and whether it is safe to be executed by the Operating System. So, basically, these types of checks are maintained by the Security Manager in .NET Application.

**JIT Compiler:**

The JIT (Just-In-Time) Compiler is responsible for Converting the MSIL code into native code (Machine Code or Binary code) that is executed by the Operating System. The native code (Machine Code or Binary code) is directly understandable by the system hardware. JIT compiles the code just before the execution and then saves this translation in memory.

**Memory Manager:**

The memory manager component of CLR in the .NET Framework allocates the necessary memory for the variables and objects that are to be used by the application.

**Garbage Collector:**

When a dot net application runs, lots of objects are created. At a given point in time, it is possible that some of those objects are not used by the application. So, [**Garbage Collector in .NET Framework**](https://dotnettutorials.net/lesson/garbage-collector/) is nothing but is a **Small Routine** or you can say it’s a **Background Process Thread** that runs periodically and try to identify what objects are not being used currently by the application and de-allocates the memory of those objects.

**Exception Manager:**

The [**Exception Manager**](https://dotnettutorials.net/lesson/exception-handling-csharp/) component of CLR in the .NET Framework redirects the control to execute the catch or finally blocks whenever an exception has occurred at runtime.

**Common Type System (CTS) in .NET Framework:**

The .NET Framework supports many programming languages such as C#, VB.NET, J#, etc. Every programming language has its own data type. One programming language data type cannot be understood by other programming languages. But, there can be situations where we need to communicate between two different programming languages. For example, we need to write code in the VB.NET language and that code may be called from C# language. In order to ensure smooth communication between these languages, the most important thing is that they should have a [**Common Type System (CTS**](https://dotnettutorials.net/lesson/common-type-system/)) which ensures that data types defined in two different languages get compiled to a common data type.

CLR in .NET Framework will execute all programming language’s data types. This is possible because CLR having its own data types which are common to all programming languages. At the time of compilation, all language-specific data types are converted into CLR’s data type. This data type system of CLR is common to all .NET Supported Programming languages and this is known as the [**Common Type System**](https://dotnettutorials.net/lesson/common-type-system/)(CTS).

**CLS (Common Language Specification) in .NET Framework:**

[**CLS (Common Language Specification)**](https://dotnettutorials.net/lesson/common-language-specification/)is a part of CLR in the .NET Framework. The .NET Framework supports many programming languages such as C#, VB.NET, J#, etc. Every programming language has its own syntactical rules for writing the code which is known as **language specification**. One programming language syntactical rules (language specification) cannot be understood by other programming languages. But, there can be situations where we need to communicate between two different programming languages. In order to ensure smooth communication between different .NET Supported Programming Languages, the most important thing is that they should have **Common Language Specifications**which ensures that language specifications defined in two different languages get compiled to a Common Language Specification.

CLR in .NET Framework will execute all programming language’s code. This is possible because CLR having its own language specification (syntactical rules) which are common to all .NET Supported Programming Languages. At the time of compilation, every language compiler should follow this language specification of CLR and generate the MSIL code. This language specification of CLR is common for all programming languages and this is known as [**Common Language Specifications (CLS)**](https://dotnettutorials.net/lesson/common-language-specification/)**.**

.NET Program Execution Process

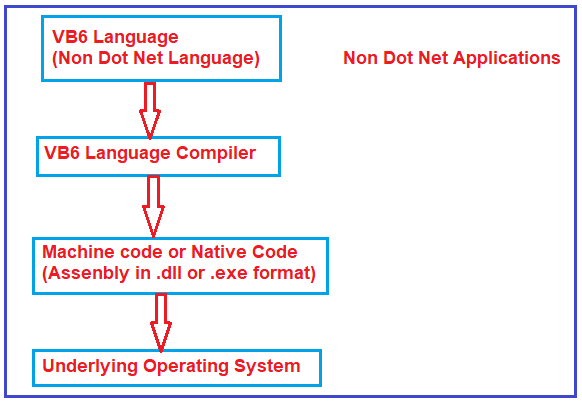
Back to: [C#.NET Tutorials For Beginners and Professionals](https://dotnettutorials.net/course/csharp-dot-net-tutorials/)

**.NET Program Execution Process Flow:**

As .NET Developers, we should know when we create an application, how the application is compiled, and how the application is executed by the .NET Framework. But before understanding the **.NET Program Execution** process, let us first understand how non-dot net applications such as C, VB6, and C++ programs are executed.

**Non-DOT NET Program Execution** **Process**:

We know that computers only understand machine-level code. The Machine-level code is also known as native code or binary code. So when we compile a C, VB6, or C++ program the respective language compiler compiles the respective language source code and generates the native machine code (also called binary code) which can be understood by the underlying operating system and the system hardware. The above process is shown in the below image.



The Native code or machine code that is generated by the respective language compiler is specific to the operating system on which it is generated. If we take this compiled native code and try to run it on another operating system, then it will fail. So the problem with this style of program execution is that it is not portable from one platform to another platform. That means it is platform-dependent.

**.NET Program Execution Process:**

.NET we can create different types of applications such as Console, Windows, Web, and Mobile Applications. Irrespective of the type of application when we execute any .NET application the following things are happening in order

The .NET application Source Code gets compiled into Microsoft Intermediate language (MSIL) which is also called Intermediate language (IL) or Common Intermediate language (CIL) code. Both .NET and Non-DOTNET applications generate an assembly when we compile the application. Generally, the assemblies have an extension of. DLL or .EXE based on the type of application we compiled. For example, if we compile a Window or Console application in .NET, we get an assembly of type .EXE whereas when we compile a Web or Class Library Project in .NET, we get an assembly of type .DLL.

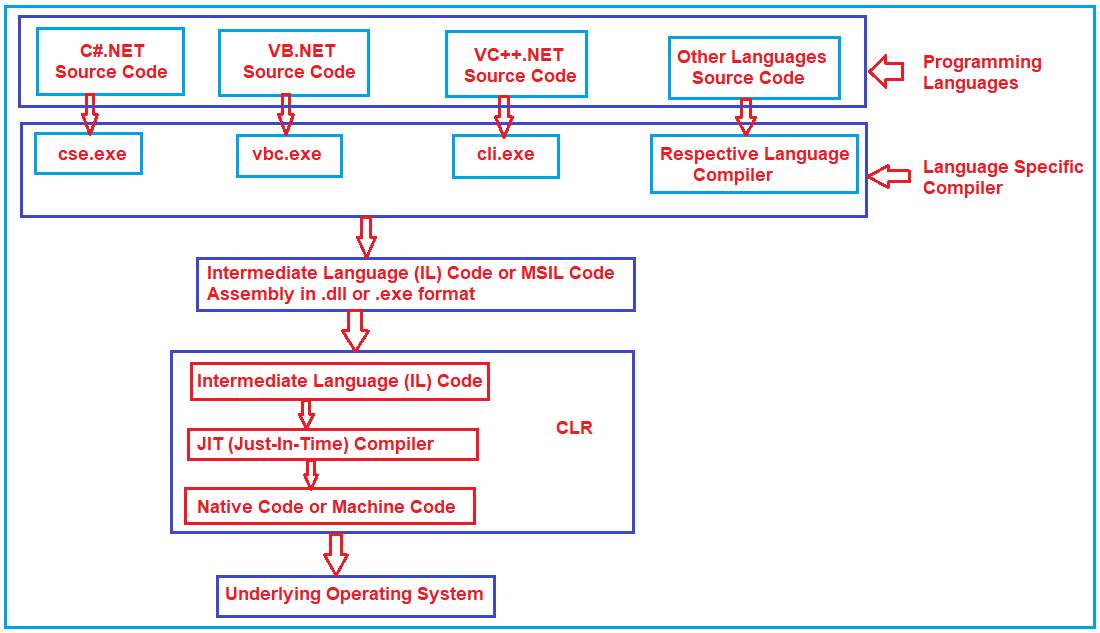
The difference between a .NET and NON-DOTNET assembly is that .NET Assembly is an Intermediate Language format whereas NON-.NET assembly is in native code format.

The NON .NET applications can run directly on top of the operating system as the NON-DOTNET assembly contains the native code whereas .NET applications run on top of a virtual environment called **Common Language Runtime (CLR)**. CLR contains a component called **Just-In-Time Compiler (JIT)** which will convert the Intermediate language into native code which can be understood by the underlying operating system.

**.NET Program Execution Steps:**

In .NET, the application execution consists of 2 steps. They are as follows:

In step1 the respective language compiler compiles the Source Code into Intermediate Language (IL) and in the 2nd step, the JIT compiler of CLR will convert the Intermediate Language (IL) code into native code (Machine Code or Binary Code) which can then be executed by the underlying operating system. The above process is shown in the image below.



As the .NET assembly is in Intermediate Language (IL) format and not in native code or machine code, the .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 Intermediate Language code into native code or machine code that the underlying operating system can understand.

Intermediate Language code is also called 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, then it may get out of memory exceptions. On the other hand, a .NET programmer needs not worry about de-allocating the memory consumed by an object. Automatic memory management is also known as garbage 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 the managed code.

.NET supports different programming languages like C#, VB, J#, and C++. C#, VB, and J# can only generate managed code (IL) whereas C++ can generate both managed code (IL) and unmanaged code (Native code).

The native code is not stored permanently anywhere after we close the program the native code is thrown away. When we execute the program again the native code gets generated again.

The .NET program is similar to java program execution. In Java, we have bytecodes and JVM (Java Virtual Machine) whereas in .NET we have Intermediate Language and CLR (Common Language Runtime).

**SECURITY**

This article addresses the security aspects of assemblies, which are the building blocks of .NET Framework applications. They form the fundamental unit of deployment, version control, reuse, activation scope, and security authorization. This last aspect, security authorization, is the focal point of this article.

Microsoft took a big stride toward improved configuration and maintenance of software systems by allowing administrators rather than developers to determine the permissions a segment of code should have at runtime. Now such decisions fall under the domain of professionals who really know the runtime environment.

During installation of a new application in the Windows Operating System (OS), the setup package may overwrite existing shared dynamic-link libraries (DLLs) to update the module. (Of course, well-designed installation programs prompt you about duplicate file names.) The overwriting occurs because you cannot have multiple versions of DLLs in a shared environment. This often causes existing applications to function incorrectly because they may depend on a specific version of the shared DLL, which may have been overwritten! The most recently installed version of the shared DLL can overwrite a previously installed yet more current version. The version most recently installed prevails for all of the installed applications referencing that DLL module, potentially causing enormous problems for administrators.

The C# compiler processes your source code and produces Microsoft intermediate language (MSIL) assembly files. Additionally, it embeds metadata in your assemblies. The .NET metadata is merely a cluster of information, specifically, the declared types, methods, fields, properties, and events implemented in the files of the assembly, made persistent in binary data. The metadata is always embedded in the same executable file and DLL as the MSIL code. For example, the metadata contains reference data to find the actual code for every implemented method that the common language runtime (CLR) uses.

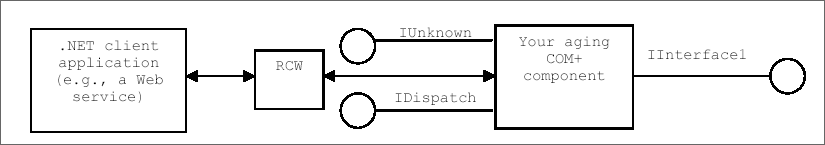
Not surprisingly, in the .NET Framework, assemblies are also versioned like DLLs that are based on the Component Object Model (COM). However, multiple versions of an assembly can coexist on the same computer without causing trouble. This allows applications to run with the assemblies they were built with. Assemblies contain all the metadata related to them, so no separate files are necessary to describe the contents of an assembly.

As with COM+, the .NET Framework code that accesses computing resources must be assigned permission to use those resources. The system administrator can configure security policies to grant access to resources based on the caller's identity and the origin of the code. For example, suppose you develop code that accesses any registry key. At execution time, the framework requests Security in .NET 711 IUnknown IInterface1 permission to access the registry. This way the system can make sure that the assembly in which the executed program exists has permission to access the registry key.

Formerly, all applications installed on a computer had to be registered before they could be used. Most installation programs not only copy files to the computer but also add various information to the registry and create various hard settings needed to uninstall that application later.

It is safe to say that we have reached the end of the COM+ era and the beginning of the .NET era. With the advent of .NET Framework, we no longer need to develop COM+ components unless they are truly necessary in order to interact with legacy applications. Of course, we will still use legacy COM+ components with our current Web services and other .NET applications during the migration to .NET applications.

Figure 22.1 depicts the typical way of exposing a COM+ component interface via the .NET framework. You can use any .NET client application, such as a Web service in which the .NET client application itself is a server application, to expose the component interface. In other words, the .NET client application (Web service) is an intermediary application to the actual COM+ component or server.



**Figure 22.1: Client Access Through Runtime Callable Wrapper**

The runtime callable wrapper (RCW) of .NET wraps the COM+ components' exposed interfaces. The RCW mediates between the COM+ component and the CLR. RCW exposes the COM+ components to .NET clients as if they were native .NET components. RCW exposes the .NET clients to COM+ components as if they were standard COM clients.

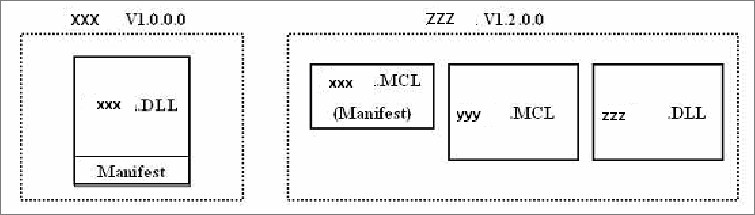
As a .NET client developer, you can generate the RCW using either Visual Studio .NET (VS.NET) or the command-line-based type library import utility of the .NET software development kit (SDK). With the VS.NET approach, you add a COM+ reference to your active project; with the .NET SDK approach, you use a command-line tool called tlbimp.exe.

To install an application within the .NET Framework, you just copy the application's files into a directory on the client computer. There are two assembly types in .NET: private and public. You usually copy the private assemblies to the application installation directory. Any subfolder within the installation directory is also an appropriate place for private assemblies. You can uninstall private assemblies by deleting them from the local directory. You install shared public assemblies either to the global assembly cache (GAC) or to another global location on the computer disk. When any client application wants to reference and use those classes in a public (shared) or private assembly, it consults its configuration files to locate and then load the shared or private assembly. If the desired assembly cannot be found in the client application's configuration files, the application has the option of searching the system's configuration files.

The manifest information cluster is the most significant part of an assembly. The manifest of an assembly contains the following metadata information:

* Assembly name: a textual string name of the assembly. IDispatch .NET client application (e.g., a Web service) RCW Your aging COM+ component
* Version information: a major and minor version number, a revision number, and a build number. Version information is used by the CLR to enforce version policy. It defines which version of the assembly is loaded and run when an assembly is referenced from an application.
* Strong-name information: a cryptographic identifier, which the author of the assembly provides, and the public key of the publisher, which is used to create the strong-name signature for the assembly.
* Culture information: the culture information that the assembly supports.
* List of all files in the assembly: a hash of each file that was present when the manifest was built and the relative path to the files. The list also contains the simple name, public key, and versioning information of each dependent assembly.
* Type reference list: a list that identifies supported types.

Figure 22.2 depicts the differences between single-file and multifile assemblies. In some instances, the manifest actually resides inside an assembly.



**Figure 22.2: Single-file (XXX) and Multifile (ZZZ) Assemblies and Their Manifests**

**Conclusion**

Hope this article would have helped you in understanding Security in .NET. See other articles on the website on .NET and C#.

# Serialization (C#)

* Article
* 09/15/2021
* 4 minutes to read
* 10 contributors

Serialization is the process of converting an object into a stream of bytes 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 deserialization.

## How serialization works

This illustration shows the overall process of serialization:



The object is serialized to a stream that carries the data. The stream may also have information about the object's type, such as its version, culture, and assembly name. From that stream, the object 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 re-create it as needed, providing storage of objects as well as data exchange. Through serialization, a developer can perform actions such as:

* Sending the object to a remote application by using a web service
* Passing an object from one domain to another
* Passing an object through a firewall as a JSON or XML string
* Maintaining security or user-specific information across applications

## JSON serialization

The [System.Text.Json](https://docs.microsoft.com/en-us/dotnet/api/system.text.json) namespace contains classes for JavaScript Object Notation (JSON) serialization and deserialization. JSON is an open standard that is commonly used for sharing data across the web.

JSON serialization serializes the public properties of an object into a string, byte array, or stream that conforms to [the RFC 8259 JSON specification](https://tools.ietf.org/html/rfc8259). To control the way [JsonSerializer](https://docs.microsoft.com/en-us/dotnet/api/system.text.json.jsonserializer) serializes or deserializes an instance of the class:

* Use a [JsonSerializerOptions](https://docs.microsoft.com/en-us/dotnet/api/system.text.json.jsonserializeroptions) object
* Apply attributes from the [System.Text.Json.Serialization](https://docs.microsoft.com/en-us/dotnet/api/system.text.json.serialization) namespace to classes or properties
* [Implement custom converters](https://docs.microsoft.com/en-us/dotnet/standard/serialization/system-text-json-converters-how-to)

## Binary and XML serialization

The [System.Runtime.Serialization](https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization) namespace contains classes for binary and XML serialization and deserialization.

Binary serialization uses binary encoding to produce compact serialization for uses such as storage or socket-based network streams. In binary serialization, all members, even members that are read-only, are serialized, and performance is enhanced.

**Warning**

Binary serialization can be dangerous. For more information, see **[BinaryFormatter security guide](https://docs.microsoft.com/en-us/dotnet/standard/serialization/binaryformatter-security-guide)**.

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. XML serialization results in strongly typed classes with public properties and fields that are converted to XML. [System.Xml.Serialization](https://docs.microsoft.com/en-us/dotnet/api/system.xml.serialization) contains classes for serializing and deserializing XML. You apply attributes to classes and class members to control the way the [XmlSerializer](https://docs.microsoft.com/en-us/dotnet/api/system.xml.serialization.xmlserializer) serializes or deserializes an instance of the class.

### Making an object serializable

For binary or XML serialization, you need:

* The object to be serialized
* A stream to contain the serialized object
* A [System.Runtime.Serialization.Formatter](https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.formatter) instance

Apply the [SerializableAttribute](https://docs.microsoft.com/en-us/dotnet/api/system.serializableattribute) attribute to a type to indicate that instances of the type can be serialized. An exception is thrown if you attempt to serialize but the type doesn't have the [SerializableAttribute](https://docs.microsoft.com/en-us/dotnet/api/system.serializableattribute) attribute.

To prevent a field from being serialized, apply the [NonSerializedAttribute](https://docs.microsoft.com/en-us/dotnet/api/system.nonserializedattribute) attribute. If a field of a serializable type contains a pointer, a handle, or some other data structure that is specific to a particular environment, and the field cannot be meaningfully reconstituted in a different environment, then you may want to make it nonserializable.

If a serialized class contains references to objects of other classes that are marked [SerializableAttribute](https://docs.microsoft.com/en-us/dotnet/api/system.serializableattribute), those objects will also be serialized.

### Basic and custom serialization

Binary and XML serialization can be performed in two ways, basic and custom.

Basic serialization uses .NET to automatically serialize the object. The only requirement is that the class has the [SerializableAttribute](https://docs.microsoft.com/en-us/dotnet/api/system.serializableattribute) attribute applied. The [NonSerializedAttribute](https://docs.microsoft.com/en-us/dotnet/api/system.nonserializedattribute) can be used to keep specific fields from being serialized.

When you use basic serialization, the versioning of objects may create problems. You would use custom serialization when versioning issues are important. Basic serialization is the easiest way to perform serialization, but it does not provide much control over the process.

In custom serialization, you can specify exactly which objects will be serialized and how it will be done. The class must be marked [SerializableAttribute](https://docs.microsoft.com/en-us/dotnet/api/system.serializableattribute) and implement the [ISerializable](https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.iserializable) interface. If you want your object to be deserialized in a custom manner as well, use a custom constructor.

REFLECTION:

<https://docs.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/reflection#:~:text=Reflection%20provides%20objects%20that%20encapsulate,access%20its%20fields%20and%20properties.>

<https://www.tutorialspoint.com/csharp/csharp_reflection.htm>

<https://www.geeksforgeeks.org/what-is-reflection-in-c-sharp/>

Security

<https://docs.microsoft.com/en-us/dotnet/standard/security/>

<https://www.c-sharpcorner.com/UploadFile/puranindia/security-in-net/>