Day01_Agenda: - 1. Introduction to the .Net Framework 2. Intermediate Language (IL) 3. Assemblies and their structure, EXEs/DLLs
4. CLR and its functions
- **JIT Compilation**
- **Memory Management**
- **Garbage Collection**
- **AppDomain Management**
- **Memory Management**
- **CLS, CTS**
- **Security**

1. Introduction to .NET Framework

.NET is a software framework which is designed and developed by Microsoft. The first version of .Net framework was 1.0 which came in the year 2002. In easy words, it is a virtual machine for compiling and executing programs written in different languages like C#, VB.Net etc.

It is used to develop Form-based applications, Web-based applications, and Web services. There is a variety of programming languages available on the .Net platform, VB.Net and C# being the most common ones are . It is used to build applications for Windows, phone, web etc. It provides a lot of functionalities and also supports industry standards.

.NET Framework supports more than 60 programming languages in which 11 programming languages are designed and developed by Microsoft. The remaining Non-Microsoft Languages which are supported by .NET Framework but not designed and developed by Microsoft.

11 Programming Languages which are designed and developed by Microsoft are:

- C#.NET
- VB.NET
- C++.NET
- J#.NET
- F#.NET
- JSCRIPT.NET
- WINDOWS POWERSHELL
- IRON RUBY
- IRON PYTHON
- C OMEGA
- ASML (Abstract State Machine Language)

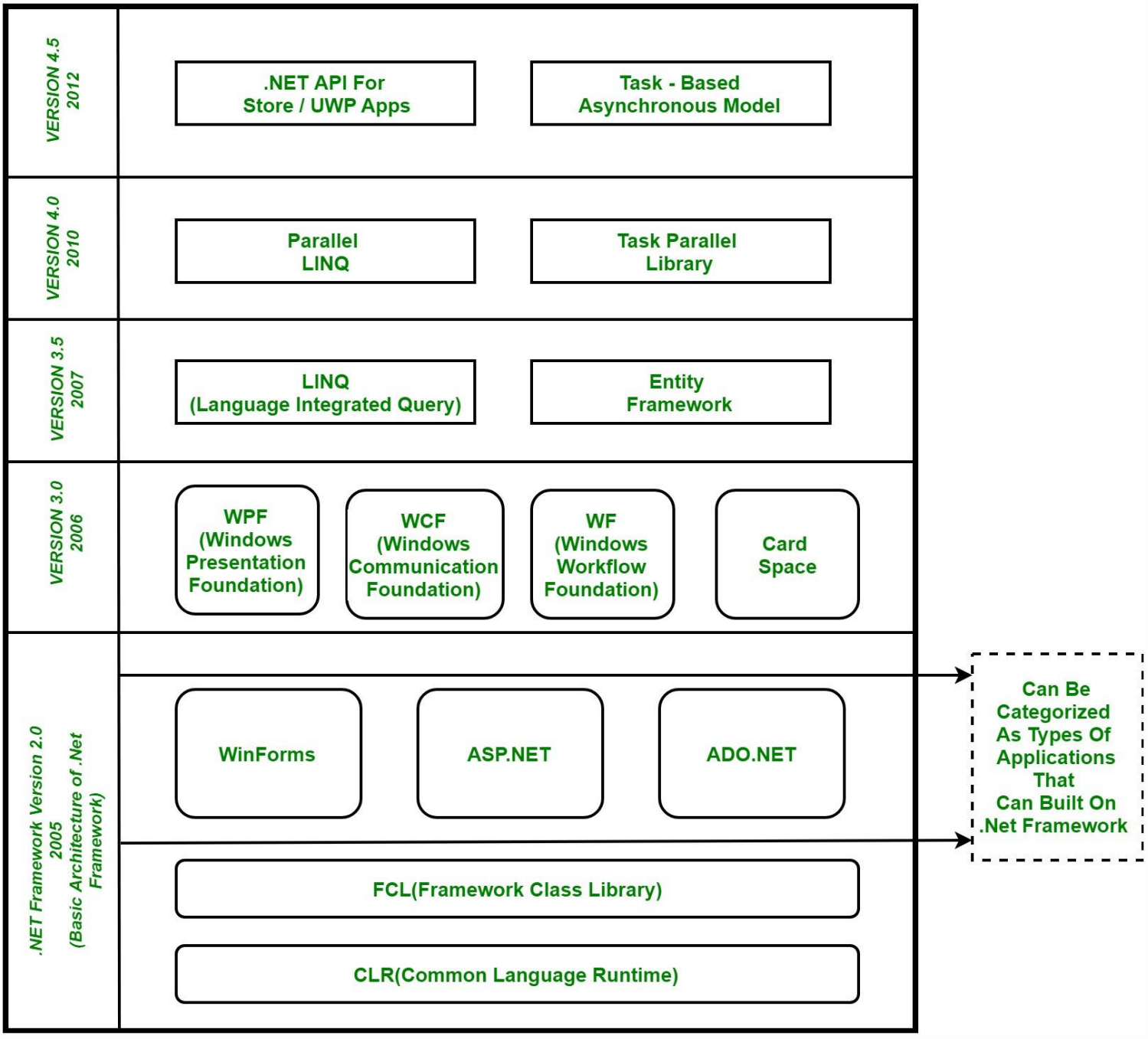
C# | .NET Framework (Basic Architecture and Component Stack)

.NET is a software framework which is designed and developed by Microsoft. The first version of .Net framework was 1.0 which came in the year 2002. In easy words, it is a virtual machine for compiling and executing programs written in different languages like C#, VB.Net etc.
It is used to develop Form-based applications, Web-based applications and Web services. There is a variety of programming languages available on the .Net platform, VB.Net and C# being the most common ones

are . It is used to build applications for Windows, phone, web etc. It provides a lot of functionalities and also supports industry standards.

Basic Architecture and Component Stack of .NET Framework

The first three components from bottom are considered as the basic architecture of .Net framework which came in the year 2005 and after this more components were added by Microsoft in the .Net Framework as following :



1. CLR (Common Language Runtime) : It is a run-time environment which executes the code written in any .NET programming language. .Net framework provides the support for many languages like C#, F#, C++, Cobra, Jscript.Net, VB.Net, Oxygene etc

2. FCL (Framework Class Library): A large number of class libraries are present in this framework which is known as FCL.

3. Types of Applications: Mainly the applications which are built in .Net framework is divided into the following three categories :

- **WinForms :** Form – Based applications are considered under this category. In simple terms, we can say client based applications which read and writes the file system comes under this category.
- **ASP .NET :** Web-Based applications come under this category. ASP.Net is a framework for web and it provides the awesome integration of HTML, CSS and JavaScript which makes it useful to develop the web applications, websites and web services. **Web services were added in .Net Framework 2.0 and considered as a part of ASP.NET web applications.**
- **ADO .NET :** It includes the application which are developed to communicate with the database like MS SQL Server, Oracle etc. comes. It mainly consists of classes that can be used to connect, retrieve, insert and delete data.

4. WPF (Windows Presentation Foundation) : Windows Presentation Foundation (WPF) is a graphical subsystem given by Microsoft which uses DirectX and is used in Windows-based applications for rendering UI (User Interface). WPF was initially released as part of .NET Framework 3.0 in 2006 and previously known as **"Avalon"**.

5. WCF (Windows Communication Foundation) : It is a framework for building connected and service-oriented applications used to transmit the data as asynchronous from one service endpoint to another service point. It was previously known as the **Indigo**.

6. WF (Windows Workflow Foundation) : It is a technology given by Microsoft which provides a platform for building workflows within .Net applications.

7. Card Space : It is a Microsoft .NET Framework software client which is designed to let users provide their digital identity to online services in a secure, simple and trusted way.

8. LINQ (Language Integrated Query) : It is introduced in .Net framework version 3.5. Basically, it is a query language used to make the query for data sources with VB or C# programming languages.

9. Entity Framework : It is open–source ORM (Object Relational Mapping) based framework which comes into .Net Framework version 3.5. It enables the .Net developer to work with database using .Net objects. Before entity framework, .Net developers have performed a lot of things related database. Like to open a connection to the database, developers have to create a Data Set to fetch or submit the data to the database, convert data from the Data Set to .NET objects or vice-versa. It creates the difficulties for developers and also it was the error-prone process, then **"Entity Framework"** comes to automate all these database related activities for the application. So, Entity Framework allows the developers to work at a higher level of abstraction.

Note: REST (Representational State Transfer) and **AJAX** were added in .Net Framework 3.5 as an extension and services of ASP.NET for enhancing web services of .NET Framework.

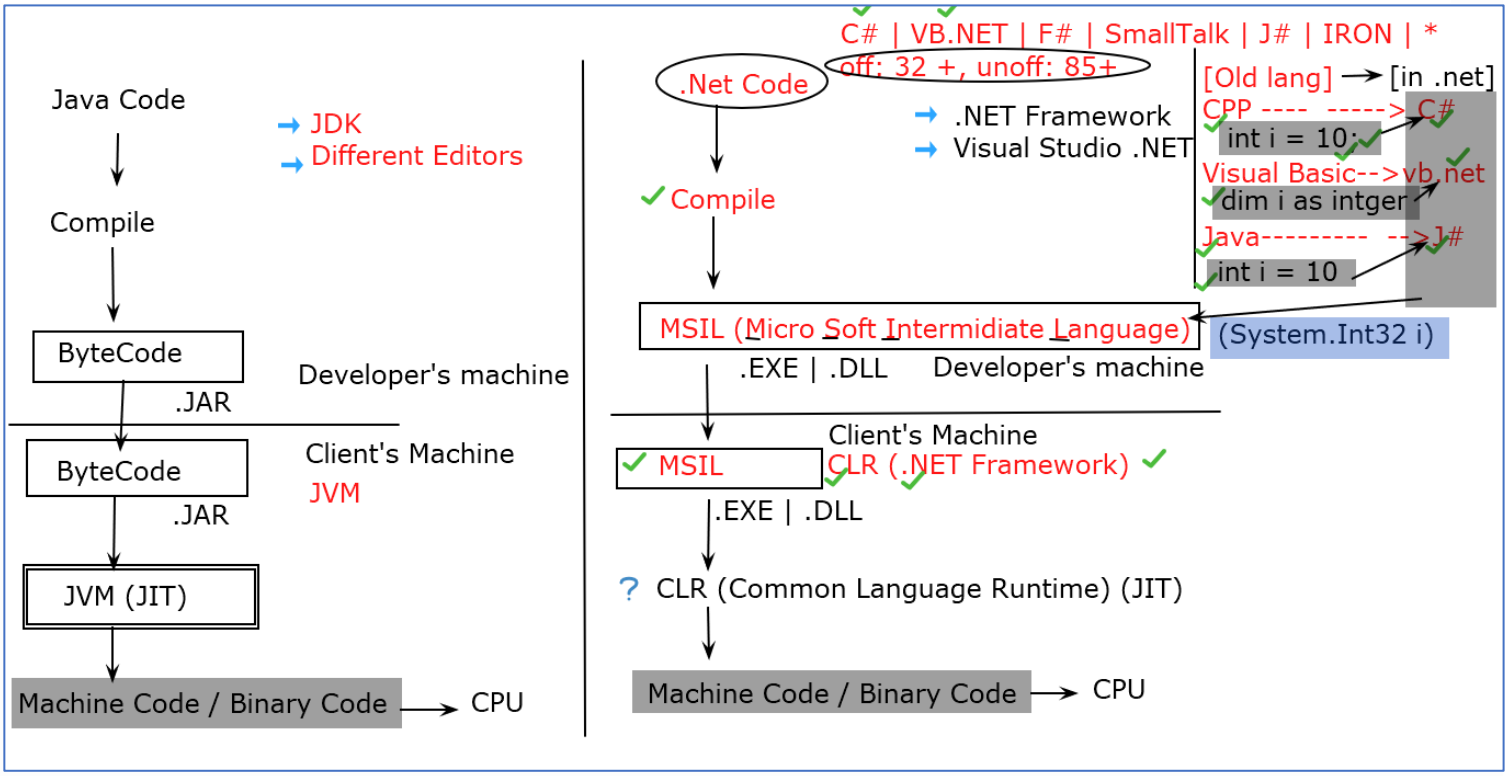
10. Parallel LINQ (Language Integrated Query) : It comes in .Net Framework version 4.0 and also termed as PLINQ. It provides a concurrent query execution engine for **LINQ**. It executes the **LINQ** in parallel such that it tries to use as much processing power system on which it is executing.

11. TPL (Task Parallel Library) : It is a set of public types and APIs. It allows the developers to be more productive by simplifying the process of adding concurrency and parallelism to .Net applications.

12. .NET API For Store/UWP Apps : In 2012, Microsoft added some APIs for creating **UWP(Universal Windows Platform)** apps for Windows using C# or VB.

13. Task-Based Asynchronous Model : It is model used to describe the asynchronous operations and tasks in .Net Framework.

.NET
- supports multiple languages while programming
- works on Windows Platoform
- A version called .NET Core is open source and can be used for other platforms as well
- We can build - desktop , web , Mobile, IOT, ML, Dashboards with .net easily
- We can use code generation - customization along with complete complete code stategy
- C# is popular language amonst all
- Healthcare, Finance, Land domain, ERP, Government



Introduction to C#

C# is a general-purpose, modern and object-oriented programming language pronounced as **"C sharp"**. It was developed by Microsoft led by Anders Hejlsberg and his team within the .Net initiative and was approved by European Computer Manufacturers Association (ECMA) and International Standards Organization (ISO). C# is among the languages for Common Language Infrastructure and the current version of C# is version 7.2. C# is a lot similar to Java syntactically and is easy for the users who have knowledge of C, C++ or Java.

A bit about .Net Framework

.Net applications are multi-platform applications and framework can be used from languages like C++, C#, Visual Basic, COBOL etc. It is designed in a manner so that other languages can use it.
know more about .Net Framework

Why C#?

C# has many other reasons for being popular and in demand. Few of the reasons are mentioned below:

1. **Easy to start:** C# is a high-level language so it is closer to other popular programming languages like C, C++, and Java and thus becomes easy to learn for anyone.
2. **Widely used for developing Desktop and Web Application:**
   C# is widely used for developing web applications and Desktop applications. It is one of the most popular languages that is used in professional desktop. If anyone want to create Microsoft apps, C# is the go-to language.

3. **Community:** The larger the community the better it is as new tools and software's will be developing to make it better. C# has a large community so the developments are done to make it exist in system and not become extinct.

4. **Game Development:**
C# is widely used in game development and will continue to dominate. C# integrate with Microsoft and thus has a large target audience. The C# features such as Automatic Garbage Collection, interfaces, object oriented etc. makes C# a popular game developing language.

Beginning with C# programming:
Finding a Compiler:

Windows: Since the C# is developed within .Net framework initiative by Microsoft, it provide various IDEs to run C# programs: Microsoft Visual Studio, Visual Studio Express, Visual Web Developer
Linux: Mono can be used to run C# programs on Linux.
Programming in C#:
Since the C# is a lot similar to other widely used languages syntactically, it is easier to code and learn in C#.
Programs can be written in C# in any of the widely used text editors like Notepad++, gedit etc. or on any of the compilers. After writing the program save the file with the extension .cs.
Example: A simple program to print **Hello World**

```
// C# program to print Hello World
using System;

namespace HelloWorldApp
{
   class HelloWorld
   {
      // Main function
      static void Main(string[] args)
      {

         // Printing Hello World
         Console.WriteLine("Hello World");

         Console.ReadKey();
      }
   }
}
```

Output:

```
Hello World
```

Explanation:
1. Comments: Comments are used for explaining code and are used in similar manner as in Java or C or C++. Compilers ignore the comment entries and does not execute them. Comments can be of single line

or multiple lines.
Single line Comments:
Syntax:

```
// Single line comment
```

Multi line comments:
Syntax:

```
/* Multi line comments*/
```

2. using System: using keyword is used to include the System namespace in the program.
namespace declaration: A namespace is a collection of classes. The HelloWorldApp namespace contains the class HelloWorld.
3. class: The class contains the data and methods to be used in the program. Methods define the behaviour of the class. Class **HelloWorld** has only one method Main similar to JAVA.
4. static void Main(): static keyword tells us that this method is accessible without instantiating the class. **5. void** keywords tells that this method will not return anything. **Main()** method is the entry-point of our application. In our program, Main() method specifies its behaviour with the statement Console.WriteLine("Hello World"); .
6. Console.WriteLine(): WriteLine() is a method of the Console class defined in the System namespace.
7. Console.ReadKey(): This is for the VS.NET Users. This makes the program wait for a key press and prevents the screen from running and closing quickly.
Note: C# is case sensitive and all statements and expressions must end with semicolon (;).

Advantages of C#:

- C# is very efficient in managing the system. All the garbage is automatically collected in C#.
- There is no problem of memory leak in C# because of its high memory backup.
- Cost of maintenance is less and is safer to run as compared to other languages.
- C# code is compiled to a intermediate language (Common (.Net) Intermediate Language) which is a standard language, independently irrespective of the target operating system and architecture.

Disadvantages of C#:

- C# is less flexible as it depends alot on .Net framework.
- C# runs slowly and program needs to be compiled each time when any changes are made.

Applications:

- C# is widely used for developing desktop applications, web applications and web services.
- It is used in creating applications of Microsoft at a large scale.
- C# is also used in game development in Unity.

Setting up the C#

Prerequisite: Introduction to C#
C# is a general-purpose, modern and object-oriented programming language pronounced as **"C sharp"**. It was developed by Microsoft led by Anders Hejlsberg and his team within the .Net initiative and was approved by European Computer Manufacturers Association (ECMA) and International Standards Organization (ISO). C# is among the languages for Common Language Infrastructure and the current

version of C# is version 7.2. C# is a lot similar to Java syntactically and is easy for the users who have knowledge of C, C++ or Java.

Basic Components involved in process of Setting up the environment in C#

1. **.Net Framework**
   The .NET Framework is a platform for building, deploying, and running Web Services and applications. To run C# applications or any program, it requires installing a .NET Framework component on the system. .NET also supports a lot of programming languages like Visual Basic, Visual C++ etc. And C# is one of the common languages which is included in the .NET Framework. It is consist of two basic components:
   - **Common Language Runtime (CLR):** The .NET Framework contains a run-time environment known as CLR which runs the codes. It provides services to make the development process easy.
   - **Framework Class Library(FCL):** It is a library of classes, value types, interfaces that provide access to system functionality.

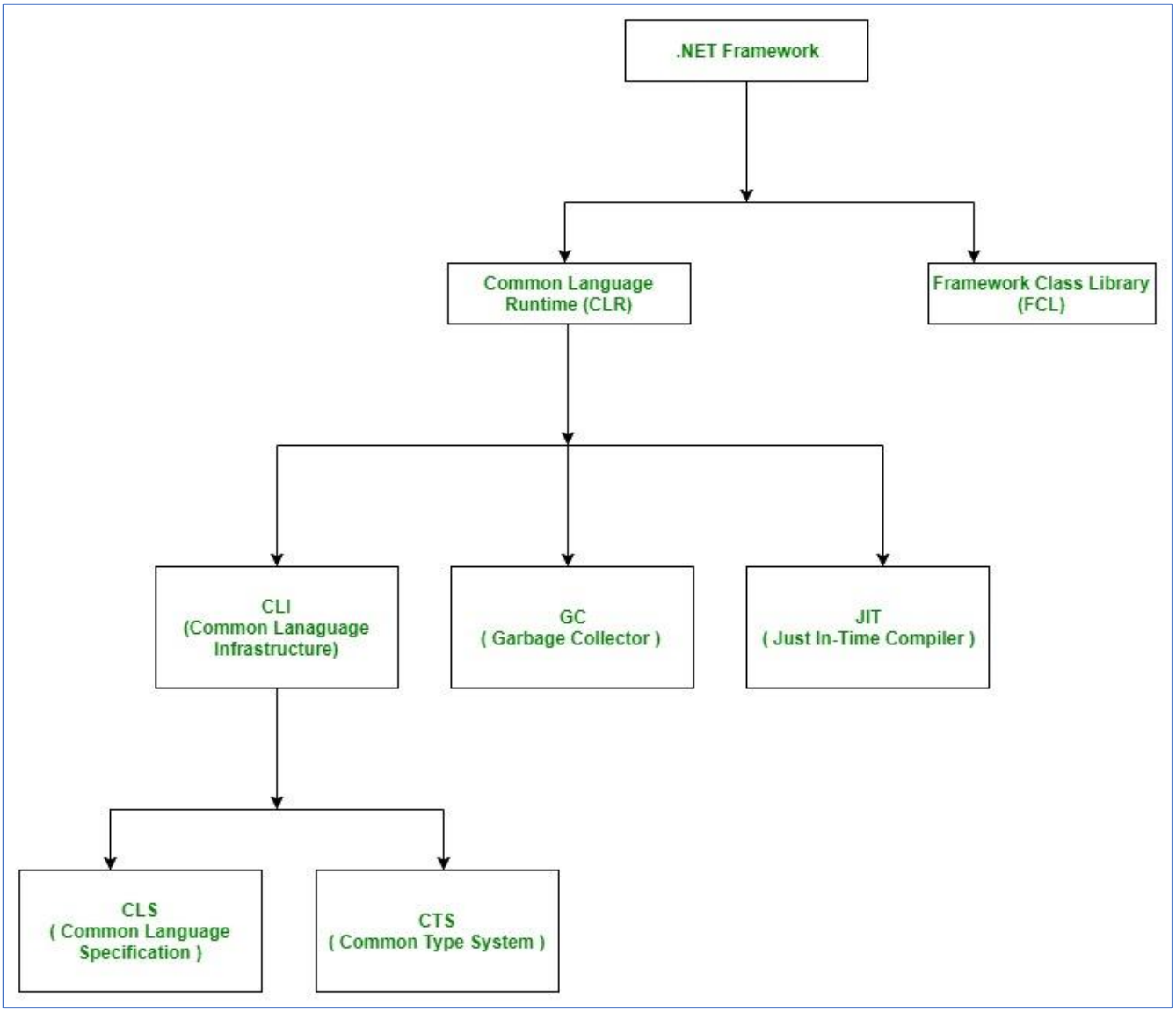
   In Windows Operating System, .NET Framework is installed by default. To know more about .NET Framework versions, click on .NET Framework Versions. of Microsoft Document.

2. **Visual Studio IDE**
   Microsoft has provided an IDE(Integrated Development Environment) tool named Visual Studio to develop applications using different programming languages such as C#, VB(Visual Basic) etc. To install and use Visual Studio for the commercial purpose it must buy a license from the Microsoft. For learning (non-commercial) purpose, Microsoft provided a free Visual Studio Community Version.

Main Components of .NET Framework

Common Language Runtime (CLR): CLR is the basic and Virtual Machine component of the .NET Framework. It is the run-time environment in the .NET Framework that runs the codes and helps in making the development process easier by providing the various services such as remoting, thread management, type-safety, memory management, robustness etc.. Basically, it is responsible for managing the execution of .NET programs regardless of any .NET programming language. It also helps in the management of code, as code that targets the runtime is known as the Managed Code and code doesn't target to runtime is known as Unmanaged code.

Framework Class Library (FCL): It is the collection of reusables, object-oriented class libraries and methods etc that can be integrated with CLR. Also called the Assemblies. It is just like the header files in C/C++ and packages in the java. Installing .NET framework basically is the installation of CLR and FCL into the system. Below is the overview of .NET Framework



Is .NET application platform dependent or platform independent?

The combination of Operating System Architecture and CPU Architecture is known as the platform. Platform dependent means the programming language code will run only on particular Operating System. A .NET application is platform dependent because of the .NET framework which is only able to run on the Windows-based operating system. The .Net application is platform independent also because of Mono framework. Using Mono framework, the .Net application can run on any Operating System including windows. Mono framework is a third-party software developed by Novell Company which is now a part of Micro Focus Company. It is a paid framework.

Important Points:

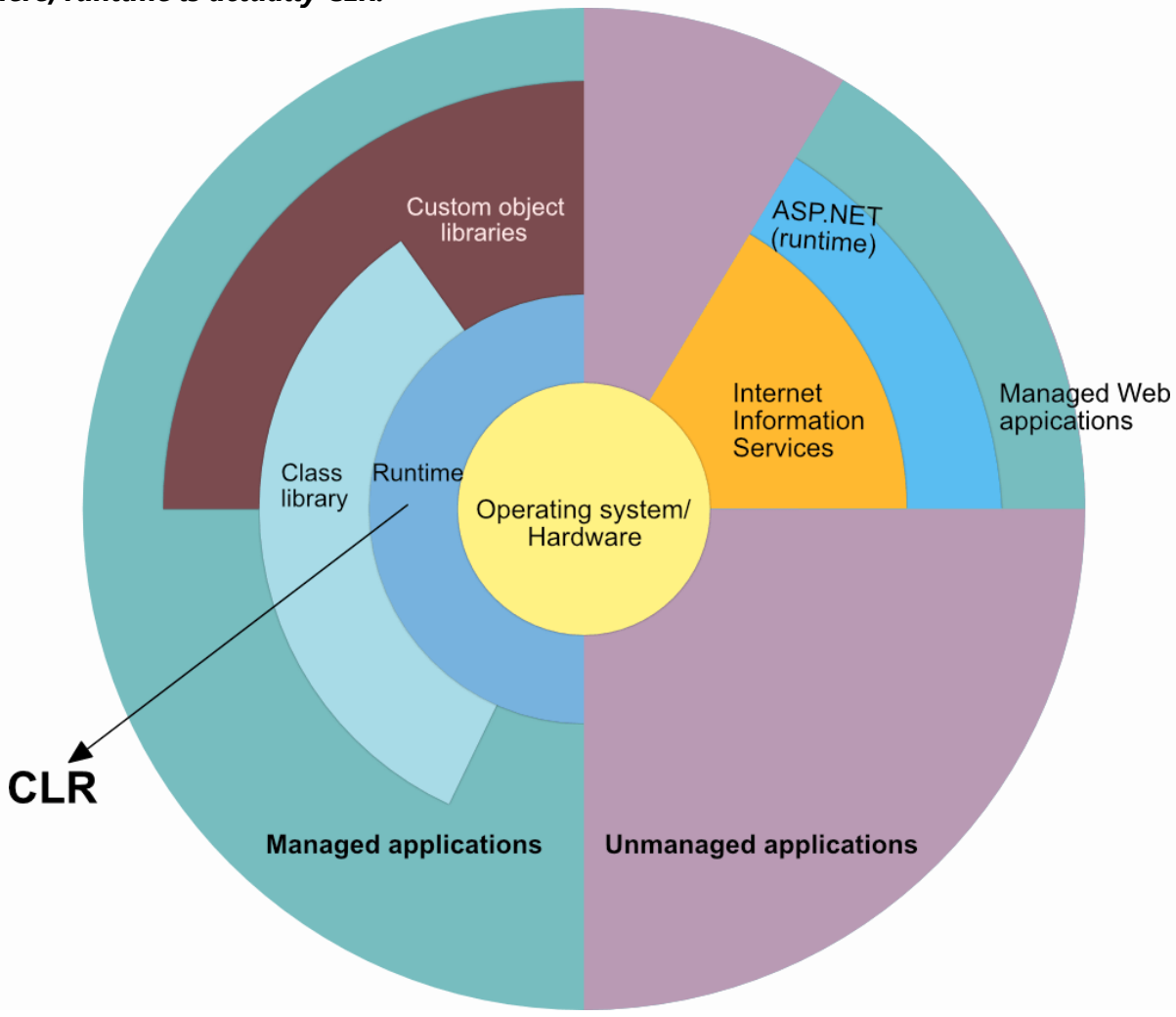
- Visual Studio is the development tool which is used to design and develop the .NET applications. For using Visual Studio, the user has to first install the .NET framework on the system.
- In the older version of Windows OS like XP SP1, SP2 or SP3, .NET framework was integrated with the installation media.
- Windows 8, 8.1 or 10 do not provide a pre-installed version 3.5 or later of .NET Framework. Therefore, a version higher than 3.5 must be installed either from a Windows installation media or from the Internet on demand. Windows update will give recommendations to install the .NET framework.

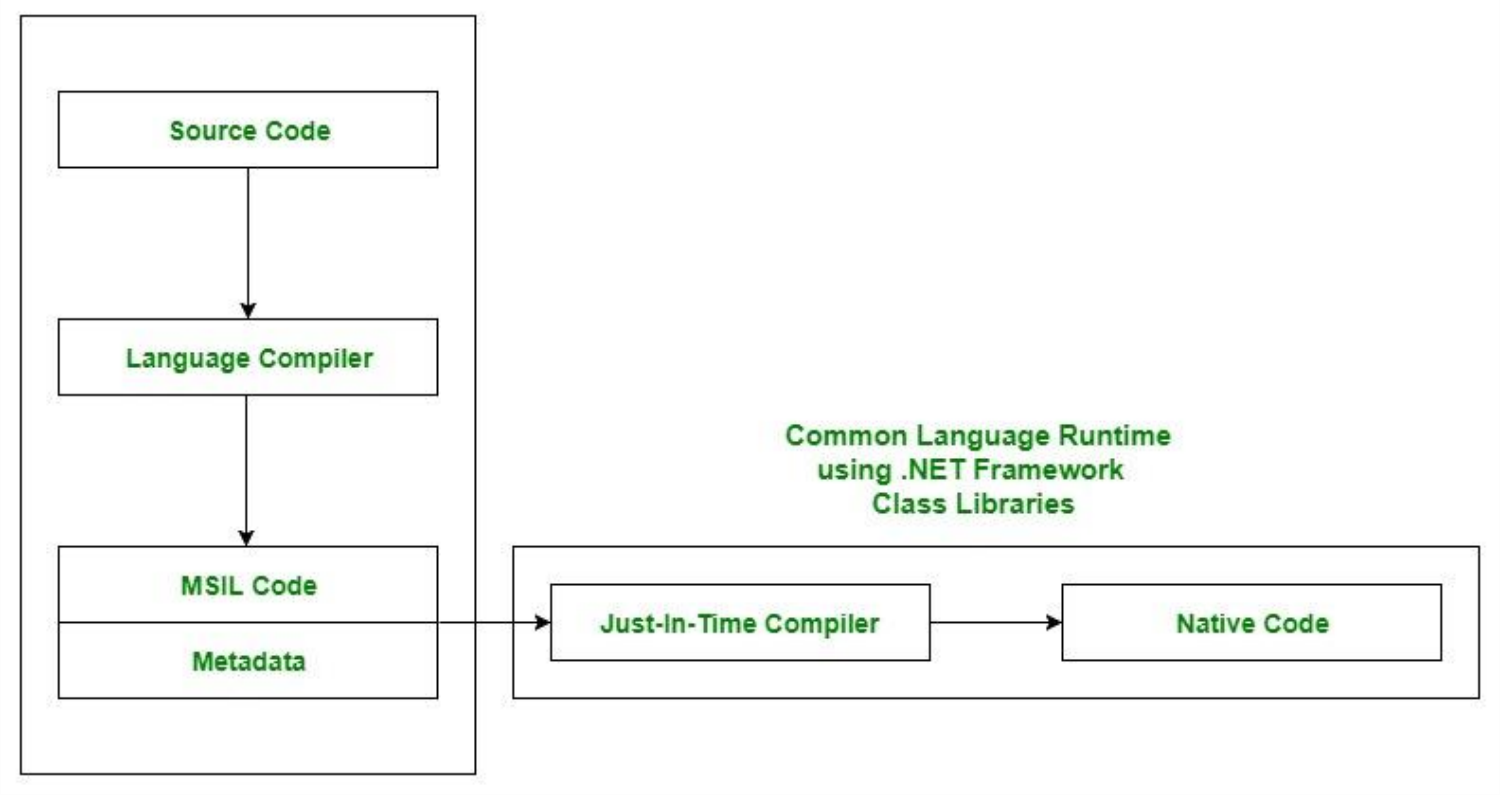
Common Language Runtime (CLR) in C#

CLR is the basic and Virtual Machine component of the **.NET Framework**. It is the **run-time enviornment in the .NET Framework** that runs the codes and helps in making the development process easier by providing the various services. Basically, it is responsible for managing the execution of *.NET programs* regardless of any *.NET* programming language. Internally, CLR implements the *VES(Virtual Execution System)* which is defined in the Microsoft's implementation of the *CLI(Common Language Infrastructure)*.

The code that runs under the Common Language Runtime is termed as the Managed Code. In other words, you can say that CLR provides a managed execution enviornment for the *.NET* programs by improving the security, including the cross language integration and a rich set of class libraries etc. CLR is present in every .NET framework verison.

Below diagram illustrate that how CLR is associated with the operating system/hardware along with the class libraries. Here, runtime is actually CLR.





Main Components of CLR

As the word specify Common which means CLR provides a common runtime or execution environment as there are more than 60 .NET programming languages.
Main components of CLR:

- Common Language Specification (CLS)
- Common Type System (CTS)
- Garbage Collection (GC)
- Just In – Time Compiler (JIT)

Common Language Specification (CLS):
It is responsible for converting the different .NET programming language syntactical rules and regulations into CLR understandable format. Basically, it provides the Language Interoperability. Language Interoperability means to provide the execution support to other programming languages also in .NET framework.

Language Interoperability can be achieved in two ways:

i. **Managed Code:** The MSIL code which is managed by the CLR is known as the Managed Code. For managed code CLR provides **three** .NET facilities:
   - CAS (Code Access Security)
   - Exception Handling
   - Automatic Memory Management.

ii. **Unmanaged Code:** Before .NET development the programming language like .COM Components & Win32 API do not generate the MSIL code. So these are not managed by CLR rather managed by Operating System which is called unmanaged code.

Common Type System (CTS):
Every programming language has its own data type system, so CTS is responsible for the understanding

all the data type system of .NET programming languages and converting them into CLR understandable format which will be a common format.

There are 2 Types of CTS that every .NET programming language have :

a. **Value Types:** Value Types will directly store the value directly into the memory location. These types work with stack mechanism only. CLR allots memory for these at Compile Time.
b. **Reference Types:** Reference Types will contain a memory address of value because the reference types won't store the variable value directly in memory. These types work with Heap mechanism. CLR allots memory for these at Runtime.

Garbage Collector:

It is used to provide the *Automatic Memory Management* feature. Suppose if there is no garbage collector then programmers have to write the memory management codes which will be a kind of overhead on programmers.

JIT(Just In Time Compiler):

It is responsible for converting the CIL(Common Intermediate Language) into machine code or native code using the Common Language Runtime environment.

Java vs C#

C# is a general-purpose, modern and object-oriented programming language pronounced as "C sharp". It was developed by Microsoft led by Anders Hejlsberg and his team.

Java is a general-purpose computer programming language that is concurrent, class-based, object-oriented etc. Java applications are typically compiled to bytecode that can run on any Java virtual machine (JVM) regardless of computer architecture.

Below are some major differences between C# and Java:

Feature	C#	Java
Operator Overloading	C# supports operator overloading for multiple operators.	Java does not support operator overloading.
Runtime Environment	C# supports CLR(Common Language Runtime).	Java supports JVM(Java Virtual Machine).
API Control	C# API are completely controlled by Microsoft.	Java API are controlled by open community process.
Public Classes	In C#, there can be many public classes inside a source code.	In Java there can be only one public class inside a source code otherwise there will be compilation error.
Checked Exceptions	C# does not supports for checked exception. In some cases checked exceptions are very useful for smooth execution of program.	Java supports both checked and unchecked exceptions.

Feature	C#	Java
Platform Dependency	C# code is windows specific. Although Microsoft is working to make it global but till now the major system does not provide support for C#.	Java is a robust and platform independent language. Platform independency of Java is through JVM.
Pointers	In C# pointers can be used only in unsafe mode.	Java does not supports anyway use of pointers.
Conditional Compilation	C# supports for conditional compilation.	Java does not supports for conditional compilation.
goto statement	C# supports for goto statement.	Java does not supports for goto statement. Use of goto statement will cause error in Java code.
Structure and Union	C# supports structures and unions.	Java doesn't support structures and unions.
Floating Point	C# does not supports strictfp keyword that means it result of floating point numbers may not be guaranteed to be same across all platforms.	Java supports strictfp keyword that means its result for floating point numbers will be same for various platform.

C++ vs C#

C# is a general-purpose, modern and object-oriented programming language pronounced as "C sharp". It was developed by Microsoft led by Anders Hejlsberg and his team.
C++ is a statically typed, multiparadigm, and object-oriented programming language. In beginning, C++ was termed as C with classes. It was developed by Bjarne Stroustrup at AT & T Bell Laboratories.
Below are some major differences between C++ and C#:

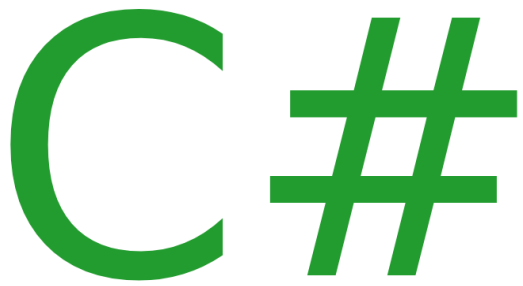
Feature	C++	C#
Memory Management	In C++ memory management is performed manually by the programmer. If a programmer creates an object then he is responsible to destroy that	In C# memory management is performed automatically by the garbage collector. If the programmer creates an object and after the completion of that

Feature	C++	C#
	object after the completion of that object's task.	object's task the garbage collector will automatically delete that object.
Platform Dependency	C++ code can be run on any platform. C++ is used where the application needed to directly communicate with hardware.	C# code is windows specific. Although Microsoft is working to make it global but till now the major system does not provide support for C#.
Multiple Inheritance	C++ support multiple inheritance through classes. Means that a class can extend more than one class at a time.	C# does not support any multiple inheritances through classes.
Bound Checking	In C++ bound checking is not performed by compiler. By mistake, if the programmer tries to access invalid array index then it will give the wrong result but will not show any compilation error.	In C# bound checking in array is performed by compiler. By mistake, if the programmer tries to access an invalid array index then it will give compilation error.
Pointers	In C++ pointers can be used anywhere in the program.	In C# pointers can be used only in unsafe mode.
Language Type	C++ is a low level language.	C# is high level object oriented language.
Level of Difficulty	C++ includes very complex features.	C# is quite easy because it has the well-defined hierarchy of classes.
Application Types	C++ is typically used for console applications.	C# is used to develop mobile, windows, and console applications.
Compilation	C++ code gets converted into machine code directly after compilation.	C# code gets converted into intermediate language code after compilation.

Feature	C++	C#
Object Oriented	C++ is not a pure object-oriented programming language due to the primitive data types.	C# is a pure object-oriented programming language.

Interesting Facts about C#

C# is a general-purpose, modern and object-oriented programming language pronounced as "C Sharp". It was developed by Microsoft led by Anders Hejlsberg and his team within the *.NET* initiative and was approved by the European Computer Manufacturers Association (ECMA) and International Standards Organization (ISO). C# is among the languages for Common Language Infrastructure and the current version of C# is version 7.2. C# is a lot similar to Java syntactically and is easy for users who have knowledge



of C, C++ or Java.

Here are some awesome facts about C# that may interest you:

1. The name of the C Sharp language is stimulated by the musical notation. Here sharp, represent that the that the written note should be made a semitone higher in pitch.
2. Microsoft first time use the name C# in 1988.
3. The syntax of C# language is similar to the C-style family like Java, C, C++.
4. C# language is suitable for writing applications for embedded systems.
5. C# language is good for developing games. It is also used by Unity(the majority leader in commercial game engines) to develop games.
6. C# language contains highest class that supports Generics and Templates.
7. C# supports internationalization.
8. C# language is used to developing web pages, android application, etc.
9. C# and XAML are the main language used to develop Windows Store Apps.
10. C# language has native garbage-collection.

2. Intermediate Language (IL)

What Does Intermediate Language (IL) Mean?

Intermediate language (IL) is an object-oriented programming language designed to be used by compilers for the .NET Framework before static or dynamic compilation to machine code. The IL is used by the .NET Framework to generate machine-independent code as the output of compilation of the source code written in any .NET programming language.

IL is a stack-based assembly language that gets converted to bytecode during execution of a virtual machine. It is defined by the common language infrastructure (CLI) specification. As IL is used for automatic generation of compiled code, there is no need to learn its syntax.

This term is also known as Microsoft intermediate language (MSIL) or common intermediate language (CIL).

3. Assemblies and their structure, EXEs/DLLs

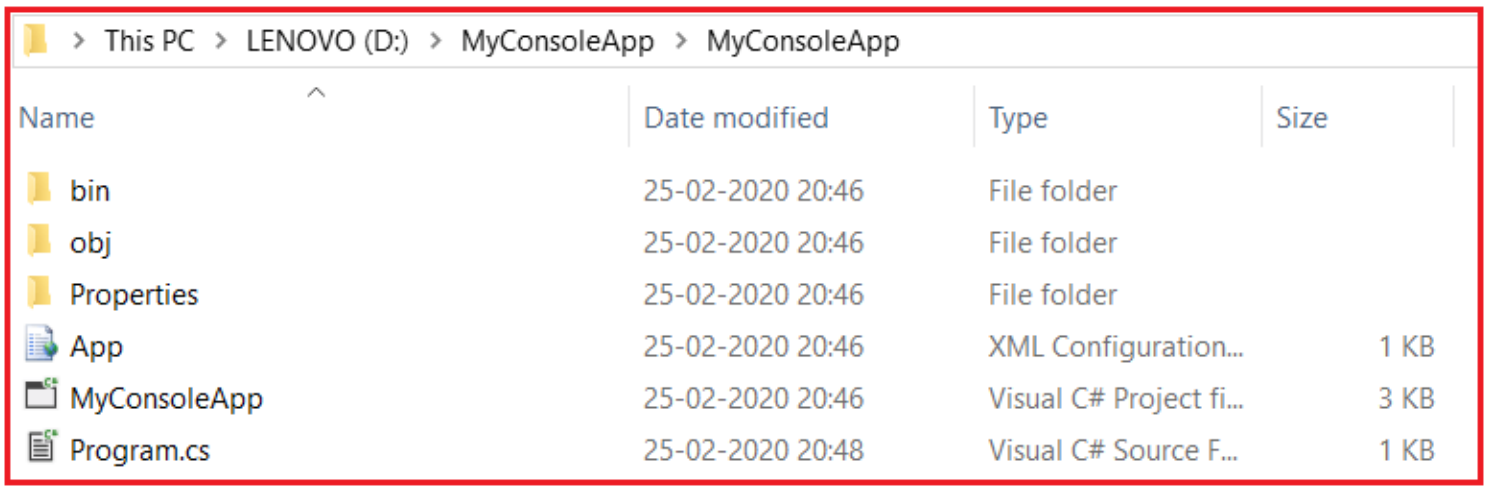
What is an Assembly in .NET?

According to MSDN, Assemblies are the building block of .NET Framework applications; they form the fundamental unit of deployment. In simple words, we can say that Assembly is nothing but a precompiled .NET Code that can be run by CLR (Common Language Runtime).

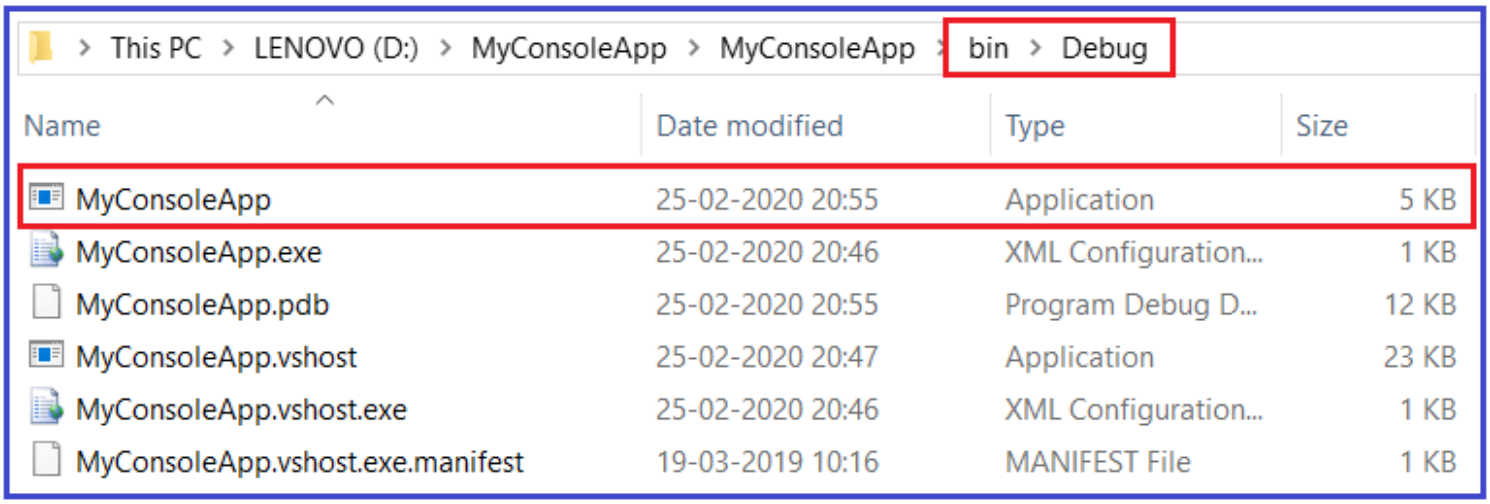
Let us understand the above definition with an example. In order to understand this, let us create a simple console application with the name MyConsoleApp. Once you created the console application then please modify the Program class as shown below.

```
using System;
namespace MyConsoleApp
{
class Program
{
static void Main(string[] args)
{
Console.WriteLine("This is From Console App");
Console.ReadKey();
}
}
}
```

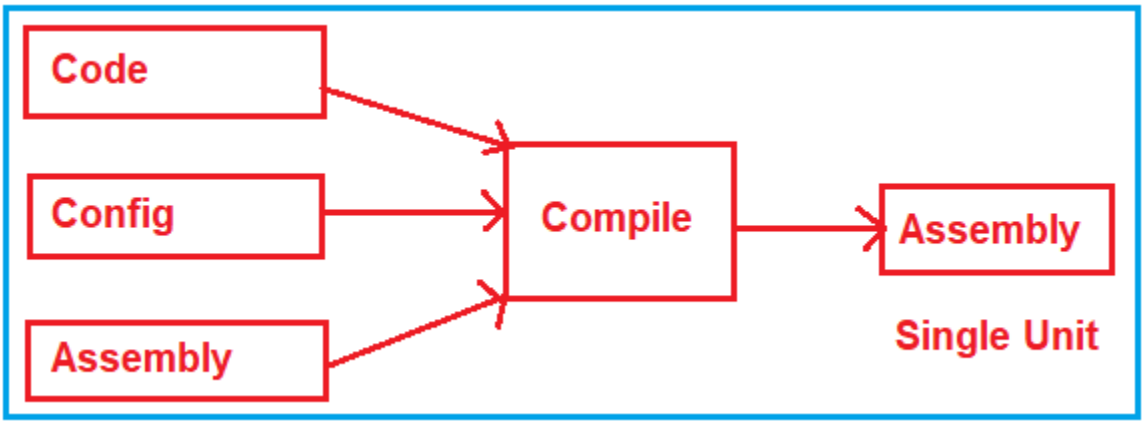
Now, if you right-click on your project and click on Open Folder in File Explorer, then you will find lots of things (Source code i.e. Program.cs class file, Configuration file i.e. App, Properties folder which contains AssemblyInfo.cs class file, etc.) as shown in the below image.



But when you build the application, then it will put the whole thing into a single EXE as shown in the below image. You can find this file under the **bin => Debug** folder. You can copy this single unit i.e. MyConsoleApp.exe and put it anywhere on your computer and from there you can run it.



So, an assembly is nothing but a single unit of deployment or it is a precompiled chunk of code that can be executed by CLR. For better understanding please have a look at the following diagram.



Types of Assemblies of in .NET Framework:

In the .NET Framework, there are two types of assemblies. They are as follows:

1. **EXE (Executable)**
2. **DLL (Dynamic Link Library)**

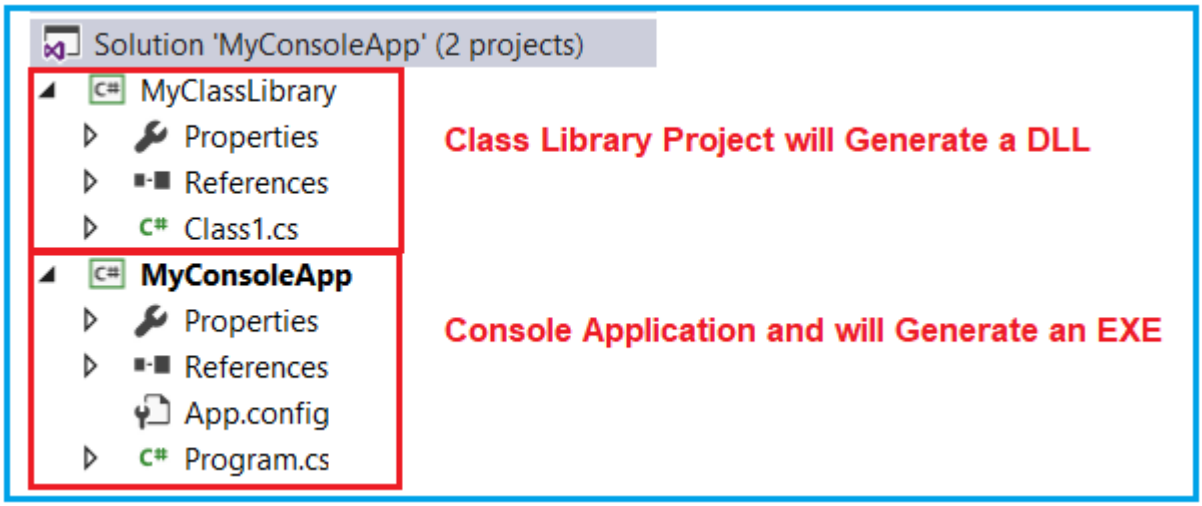
In .NET Framework when we compile a Console Application or a Windows Application, it generates EXE, whereas when we compile a Class Library Project or ASP.NET web application, then it generates DLL. In.NET framework, both EXE and DLL are called assemblies.

Understanding DLL and EXE in .NET Framework:

We already created one console application and we already see that it creates an EXE. Let us see an example of DLL. In order to create a DLL, let us add a class library project to the same solution with the name as MyClassLibrary. Once you created the class library project, it will by default create a class file with the name Class1. Let us modify Class1 as shown below.

```
namespace MyClassLibrary
{
    public class Class1
    {
        public string GetData()
        {
            return "This is from Class Library";
        }
    }
}
```

With this, now our solution contains two projects. One is a console application and the other one is a class library project as shown below.



Now, build the solution and you should get the respected assemblies as expected in their own bin => Debug folder. Now, the question that should come to your mind is what is the difference between the DLL and the EXE?

What is the difference between the DLL and the EXE in .NET Framework?

The EXE is run in its own address space or in its own memory space. If you double click on the MyConsoleApp EXE then you will get the following output. Now, this program is running out of its own memory space.

```
D:\MyConsoleApp\MyConsoleApp\bin\Debug\MyConsoleApp.exe
This is From Console App
```

Without closing this window, again if you double click on the MyConsoleApp EXE, again it will run and will display the same output. This is because now, both the EXE are running in their own memory space. The point that you need to remember is that EXE is an executable file and can run by itself as an application.

Coming to DLL, it cannot be run by itself like EXE. That means **the MyClassLibrary.dll** cannot be invoked or run by himself. It needs a consumer who is going to invoke it. So, a DLL is run inside another memory space. The other memory space can be a console, windows applications, or web applications that should have their own memory space.
For example, you can invoke the DLL from a console application. We have a console called MyConsoleApp and let's see how to invoke the MyClassLibrary.dll from this console application. In order to use the MyClassLibrary.dll inside the MyConsoleApp, first, you need to make a reference to that DLL. Once you add a reference to MyClassLibrary DLL, and then please modify the Program class of Console Application as shown below.

```
using System;
using MyClassLibrary;
namespace MyConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            //Using MyClassLibrary DLL
            Class1 obj = new Class1();
            Console.WriteLine(obj.GetData());

            Console.WriteLine("This is From Console App");
            Console.ReadKey();
        }
    }
}
```

Now, run the application and you should see the following output. Here, the MyClassLibrary DLL is run inside the MyConsoleApp address space.

```
This is from Class Library
This is From Console App
```

So, in short, the difference between them is an EXE is an executable file and can run by itself as an application whereas DLL is usually consumed by an EXE or by another DLL and we cannot run or execute DLL directly.
Now, the question that should come to your mind why do we need DLLs as it is not invoked by themselves. The reason behind the DLL is reusability. Suppose you want some class, or logic, or something else in many applications, then simply put those classes, and logic inside a DLL, and refer to that DLL wherever it is required.

Understanding Assembly

The .NET Framework overcomes the DLL Hell or version issues with existing COM technology by introducing assemblies. Assemblies are a self-describing installation unit, consisting of single or multiple files. Virtually, every file that is developed and executed under the .NET Common Language Runtime (CLR) is called an assembly. One assembly file contains metadata and could be an .EXE, DLL, or Resource file. Now, let's discuss some of the comprehensive benefits provided by the assembly.

1. Assemblies can be deployed as private or shared. Private assemblies reside in the same solution directory. Shared assemblies, on the other hand, are libraries intended to be consumed by numerous applications on a single machine because they are deployed to a central repository called the GAC.

2. The .NET assemblies are assigned a special 4-digit number to concurrently run the multiple versions of an assembly. The 4-digit special number can be specified as "<major>.<minor>.<build>.<revision>".

3. Assembly archives every external assembly reference they must have access to in order to function properly. However, assemblies are self-describing by documenting all the external references in the manifest. The comprehensive details of assemblies such as member function, variable name, base class, interface, and constructors are placed in the metadata so that the CLR does not need to consult the Windows system registry to resolve its location.

4. The .NET Framework offers you to reuse types in a language-independent manner so it does not matter how a code library is packaged.

5. Application isolation is ensured using application domains. A number of applications can run independently inside a single process with an application domain.

6. Installation of an assembly can be as simple as copying all of its files. Unlike COM, there is no need to register them in the Windows system registry.

Modules

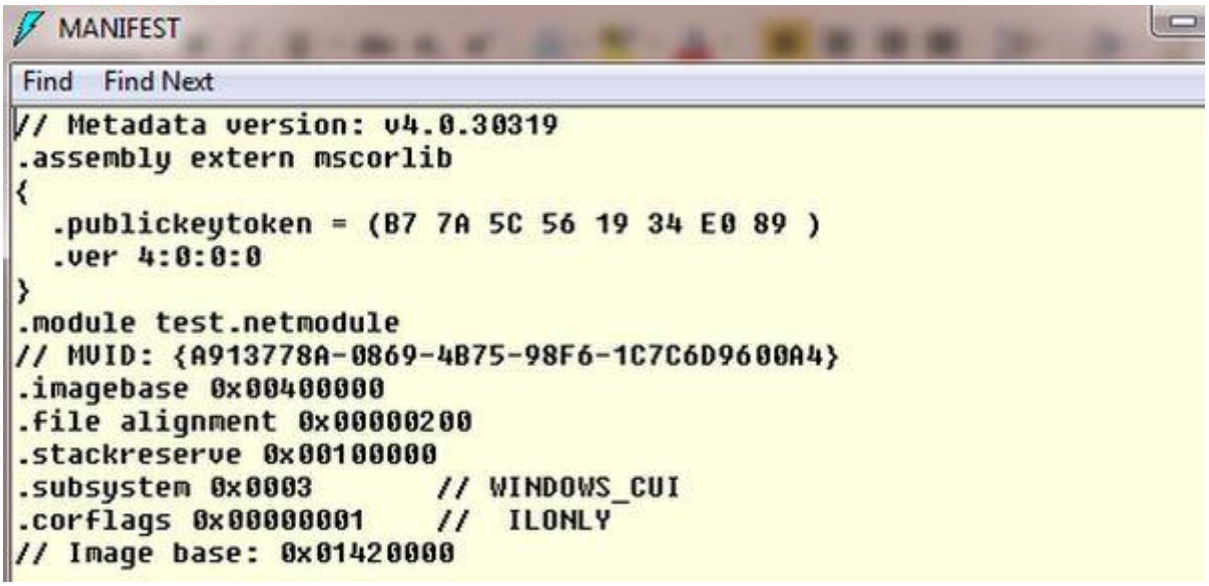
Before delving into assembly types in detail, let's discuss the modules. An assembly is typically composed of multiple modules. A module is a DLL without assembly attributes. To get a better understanding, we are creating a C# class library project as in the following.

```
1.  public class test
2.  {
3.      public test() { }
4.     public test(string fname, string lname)
5.     {
6.         this.FName = fname;
7.          this.LName = lname;
8.     }
9.     public string FName
10.    {
```

```
11.         get;
12.         set;
13.     }
14.     public string LName
15.     {
16.         get;
17.         set;
18.     }
19.     public override string ToString()
20.     {
21.         return FName + " " + LName;
22.     }
23. }
```

A module can be created by csc.exe with the "/module" switch. The following command creates a modules test.netmodule as in the following:
csc /target:module test.cs

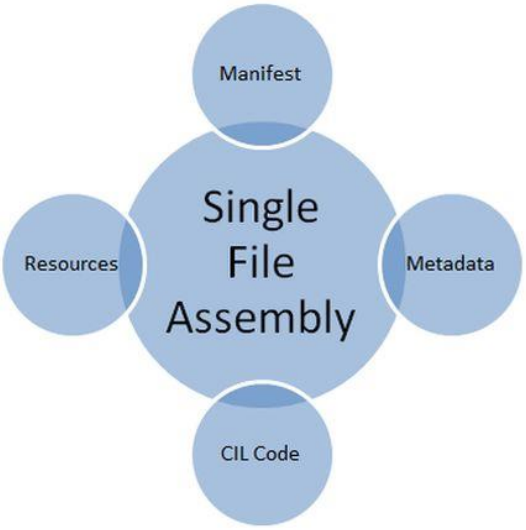
A module also has a manifest, but there is no ".assembly" entry inside the manifest because a module has no assembly attribute. We can view a module manifest using the ildasm utility as in the following:



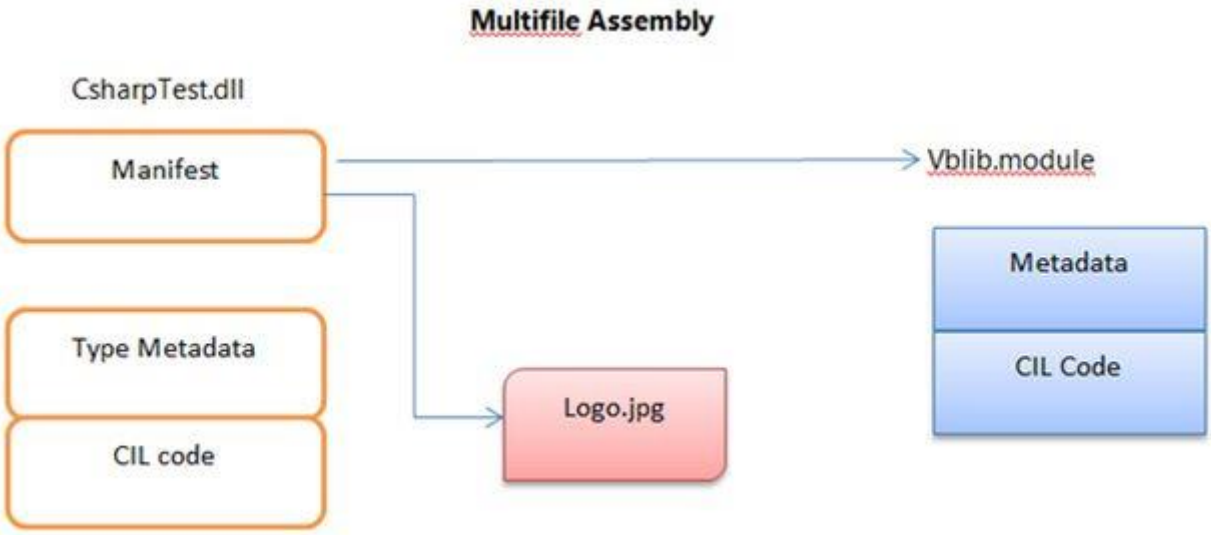
The main objective behind modules is that they can be used for faster startup of assemblies because not all types are inside a single file. The modules are loaded when needed. The second reason is, if you want to create an assembly with more than one programming language then one module could be in VB.NET and another in F#.NET. Finally, these two modules could be included in a single file.

Single file and Multifile Assembly

Technically speaking, an assembly can be formed from a single file and multiple files. A single-file assembly contains all the necessary elements such as CIL code, header files, manifest in a single *.exe or *.dll package.

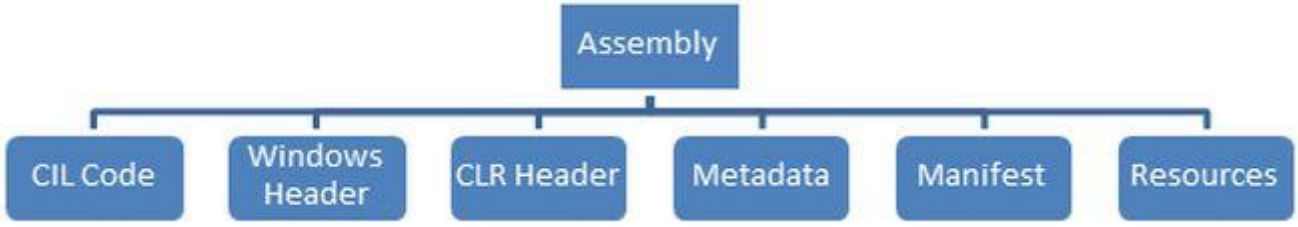


A multifile assembly, on the other hand, is a set of .NET modules that are deployed and versioned as a single unit. Formally speaking, these modules are called primary and secondary modules. The primary module contains an assembly-level manifest and the secondary modules have a *.netmodule extension containing a module-level manifest. The major benefit of a multifile assembly is that they provide a very efficient way to download content.



Assembly Structure

An assembly is comprised of assembly metadata describing the complete assembly, type metadata unfolding the exported type and methods, MSIL code and resources. All these fragments can be inside one file or spread across several files. Structurally speaking, an assembly is composed of the following elements:



CIL code

The CIL code is a CPU and platform-agnostic intermediate language. It can be considered to be the core backbone of an assembly. Given this design, the .NET assemblies can indeed execute on a variety of devices, architectures, and operating systems. At the runtime, the internal CIL is compiled using the Just-In-Time (JIT) compiler for the platform and CPU specific instructions.

```
.method public hidebysig instance string
        HelloMsg(string str) cil managed
{
  // Code size       17 (0x11)
  .maxstack  2
  .locals init ([0] string CS$1$0000)
  IL_0000:  nop
  IL_0001:  ldstr      "Hello: "
  IL_0006:  ldarg.1
  IL_0007:  call       string [mscorlib]System.String::Concat(string,
                                                              string)
  IL_000c:  stloc.0
  IL_000d:  br.s       IL_000f
  IL_000f:  ldloc.0
  IL_0010:  ret
} // end of method test::HelloMsg
```

Understanding the grammar of CIL code can be helpful when you are building a complex application but unfortunately, most .NET developers are not deeply concerned with the details of CIL code.

Dumpbin /headers *.dll/*.exe

CLR File Header

The CLR header is a block of data that all .NET assemblies must support in order to be hosted by the CLR. It typically defines numerous flags that enable the runtime to understand the layout of the managed code. We can view such diverse flags, again by using the dumpbin.exe /clrheader flag as in the following:

```
E:\Temp\NETcomponent\NETcomponent\bin\Debug>dumpbin /clrheader NETcomponent.dll
Microsoft (R) COFF/PE Dumper Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.


Dump of file NETcomponent.dll

File Type: DLL

  clr Header:

              48 cb
            2.05 runtime version
            2078 [     648] RVA [size] of MetaData Directory
               1 flags
                   IL Only
               0 entry point token
               0 [       0] RVA [size] of Resources Directory
               0 [       0] RVA [size] of StrongNameSignature Directory
               0 [       0] RVA [size] of CodeManagerTable Directory
               0 [       0] RVA [size] of VTableFixups Directory
               0 [       0] RVA [size] of ExportAddressTableJumps Directory
               0 [       0] RVA [size] of ManagedNativeHeader Directory


  Summary

        2000 .reloc
        2000 .rsrc
        2000 .text
```

Metadata

The .NET runtimes practice metadata to resolve the location of types within the binary. Assembly metadata comprehensively describes the format of the contained types, as well as the format of external type references. If you press the Ctrl +M keystroke combination, idasm.exe displays the metadata for each type within the DLL file assembly as in the following:

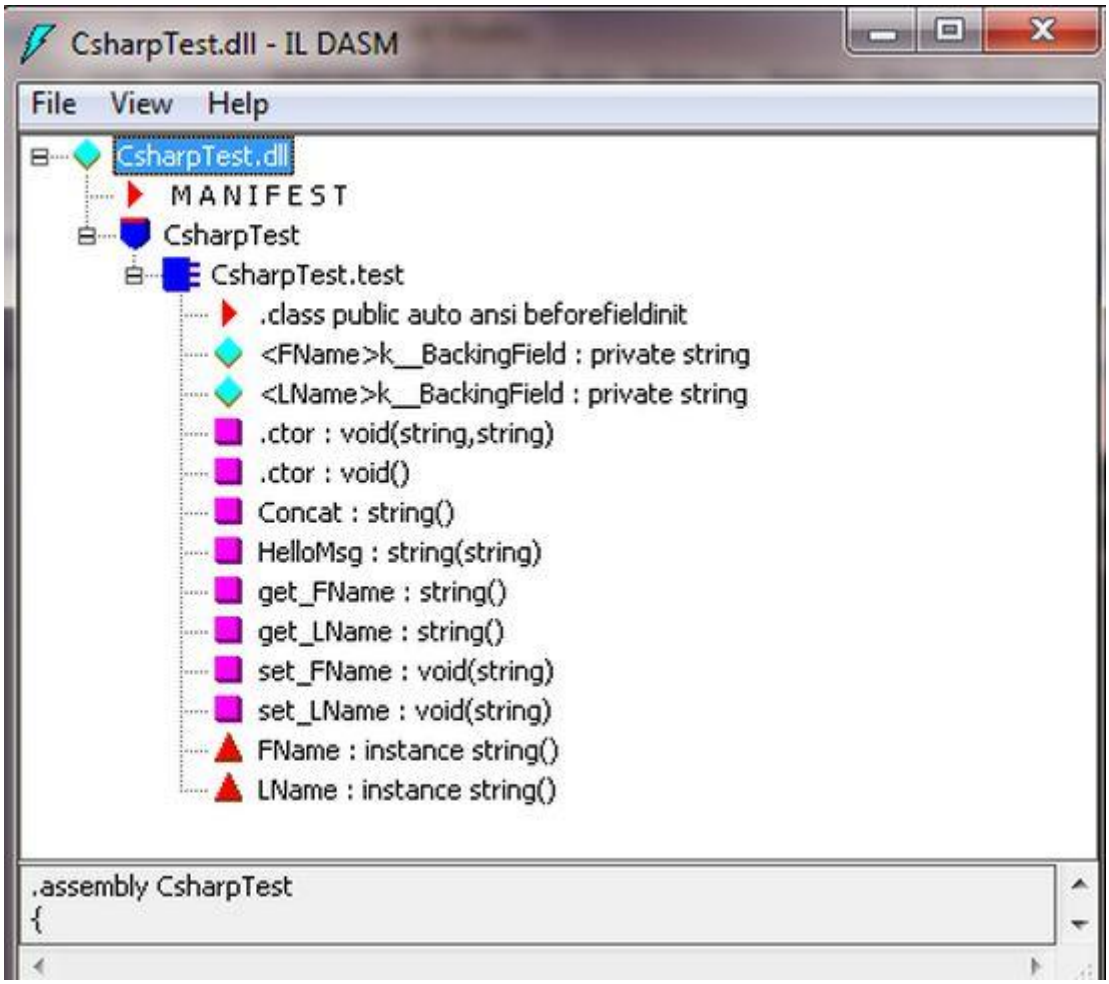
```
============================================================
ScopeName : CsharpTest.dll
MVID      : {DFEEFDEE-08C6-462E-AF18-728B11B23F6D}
============================================================
Global functions
------------------------------------------------------------

Global fields
------------------------------------------------------------

Global MemberRefs
------------------------------------------------------------

TypeDef #1 (02000002)
------------------------------------------------------------
      TypDefName: CsharpTest.test  (02000002)
      Flags     : [Public] [AutoLayout] [Class] [AnsiClass] [BeforeFieldInit]
      Extends   : 01000001 [TypeRef] System.Object
      Field #1 (04000001)
      ----------------------------------------------------------
            Field Name: <FName>k__BackingField (04000001)
            Flags     : [Private]  (00000001)
            CallCnvntn: [FIELD]
            Field type:  String
            CustomAttribute #1 (0c000001)
            ----------------------------------------------------
```

Manifest

The assembly manifest documents each module within the assembly established the version and acknowledges the external reference assemblies with their dependencies. The Assembly manifest is a significant part of an assembly and can be composed of the following parts:

- **Identity**
  It includes version, name, culture, and public key details.
- **Set of Permissions**
  his portion displays the necessary permissions to run an assembly.
- **List of Files**
  It lists all the files belonging to a single-file or multiple-file assembly.
- **External Reference Assemblies**
  The manifest also documents the external reference files that are needed to run an assembly.

We can explore the assembly manifest using the ildasm.exe utility as in the following:



Now, open the CSharpTest.dll manifest by double-clicking the MANIFEST icon. The first code block specifies all the external assemblies, such as mscorlib.dll, required by the current assembly to function correctly. Here, each .assembly extern block is qualified by the .publickeytoken and .ver directive as in the following.

```
// Metadata version: v4.0.30319
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .ver 4:0:0:0
}
```

4. CLR and its functions

- **JIT Compilation**
- **Memory Management**
- **Garbage Collection**
- **AppDomain Management**
- **Memory Management**
- **CLS, CTS**
- **Security**

.NET Common Language Runtime (CLR)

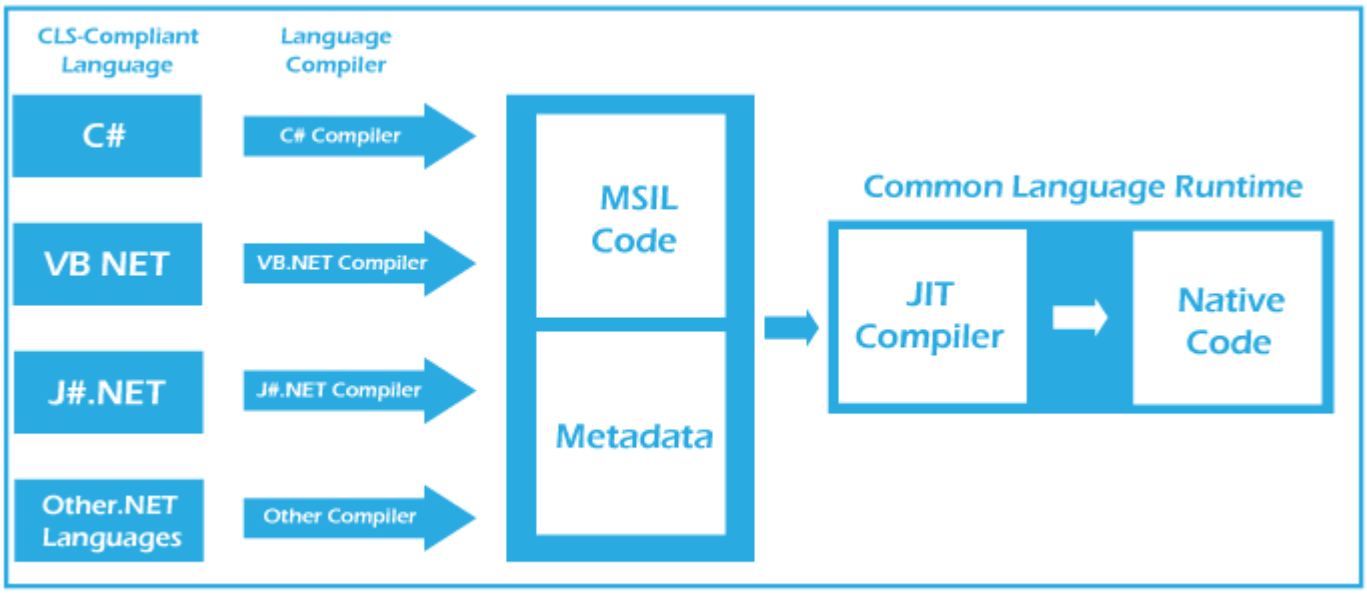
.NET CLR is a runtime environment that manages and executes the code written in any .NET programming language. CLR is the virtual machine component of the .NET framework. That language's compiler compiles the source code of applications developed using .NET compliant languages into CLR's intermediate language called MSIL, i.e., Microsoft intermediate language code. This code is platform-independent. It is comparable to byte code in java. Metadata is also generated during compilation and MSIL code and stored in a file known as the Manifest file. This metadata is generally about members and types required by CLR to execute MSIL code. A just-in-time compiler component of CLR converts MSIL code into native code of the machine. This code is platform-dependent. CLR manages memory, threads, exceptions, code execution, code safety, verification, and compilation.

The following figure shows the conversion of source code into native code.

The above figure converts code into native code, which the CPU can execute.

The main components of CLR are:

- Common type system
- Common language speciation
- Garbage Collector
- Just in Time Compiler
- Metadata and Assemblies



Exceution of a .NET Application

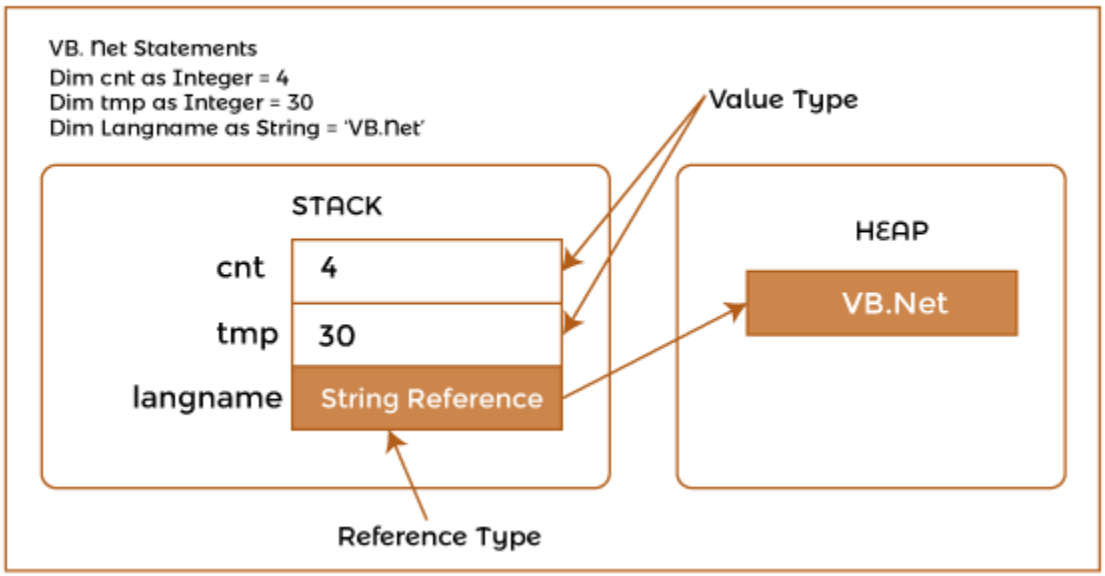
1. Common type system:

CTS provides guidelines for declaring, using, and managing data types at runtime. It offers cross-language communication. For example, VB.NET has an integer data type, and C# has an int data type for managing integers. After compilation, Int32 is used by both data types. So, CTS provides the data types using managed code. A common type system helps in writing language-independent code.

It provides two categories of Types.

1. **Value Type:** A value type stores the data in memory allocated on the stack or inline in a structure. This category of Type holds the data directory. If one variable's value is copied to another, both the variables store data independently. It can be of inbuilt-in types, user-defined, or enumerations types. Built-in types are primitive data types like numeric, Boolean, char, and date. Users in the source code create user-defined types. An enumeration refers to a set of enumerated values represented by labels but stored as a numeric type.

Value Type and Reference Type



2. **Reference Type:** A Reference type stores a reference to the value of a memory address and is allocated on the heap. Heap memory is used for dynamic memory allocation. Reference Type does not hold actual data directly but holds the address of data. Whenever a reference type object is made, it copies the address and not actual data. Therefore two variables will refer to the same data. If data of one Reference Type object is changed, the same is reflected for the other object. Reference types can be self-describing types, pointer types, or interference types. The self-describing types may be string, array, and class types that store metadata about themselves.

2. Common Language Specification (CLS):

Common Language Specification (CLS) contains a set of rules to be followed by all NET-supported languages. The common rules make it easy to implement language integration and help in cross-language inheritance and debugging. Each language supported by NET Framework has its own syntax rules. But CLS ensures interoperability among applications developed using NET languages.

3. Garbage Collection:

Garbage Collector is a component of CLR that works as an automatic memory manager. It helps manage memory by automatically allocating memory according to the requirement. It allocates heap memory to objects. When objects are not in use, it reclaims the memory allocated to them for future use. It also ensures the safety of objects by not allowing one object to use the content of another object.

4. Just in Time (JIT) Compiler:

JIT Compiler is an important component of CLR. It converts the MSIL code into native code (i.e., machine-specific code). The .NET program is compiled either explicitly or implicitly. The developer or programmer calls a particular compiler to compile the program in the explicit compilation. In implicit compilation, the program is compiled twice. The source code is compiled into Microsoft Intermediate Language (MSIL) during the first compilation process. The MSIL code is converted into native code in the second compilation process. This process is called JIT compilation. There are three types of JIT compilers -Pre, Econo, and Normal. Pre JIT Compiler compiles entire MSIL code into native code before execution. Econo JIT Compiler compiles only those parts of MSIL code required during execution and removes those parts that are not required anymore. Normal JIT Compiler also compiles only those parts of MSIL code required during execution but places them in cache for future use. It does not require recompilations of already used parts as they have been placed in cache memory.

5. Metadata:

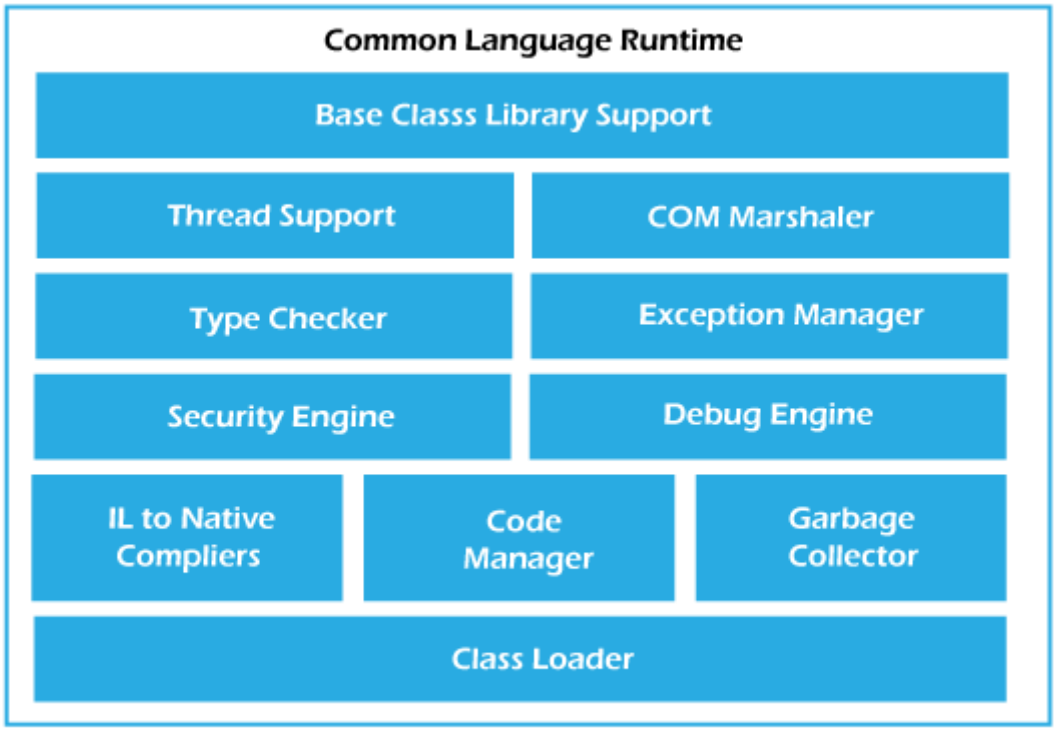
A Metadata is a binary information about the program, either stored in a CLR Portable Executable file (PE) along with MSIL code or in the memory. During the execution of MSIL, metadata is also loaded into memory for proper interpretation of classes and related. Information used in code. So, metadata helps implement code in a language-neutral manner or achieve language interoperability.

6. Assemblies:

An assembly is a fundamental unit of physical code grouping. It consists of the assembly manifest, metadata, MSIL code, and a set of resources like image files. It is also considered a basic deployment unit, version control, reuse, security permissions, etc.

.NET CLR Structure

Following is the component structure of Common Language Runtime.



Components of the Common Language runtime/Architecture of CLR

Base Class Library Support

It is a class library that supports classes for the .NET application.

Thread Support

It manages the parallel execution of the multi-threaded application.

COM Marshaler

It provides communication between the COM objects and the application.

Security Engine

It enforces security restrictions.

Debug Engine

It allows you to debug different kinds of applications.

Type Checker

It checks the types used in the application and verifies that they match the standards provided by the CLR.

Code Manager

It manages code at execution runtime.

Garbage Collector

It releases the unused memory and allocates it to a new application.

Exception Handler

It handles the exception at runtime to avoid application failure.

ClassLoader

It is used to load all classes at runtime.

- **Benefits of CLR:**
- It improves the performance by providing a richly interact between programs at the run time.
- Enhance portability by removing the need of recompiling a program on any operating system that support it.

- Security also increases as it analyze the MSIl instructions whether they are safe or unsafe. Also, the use of delegates in place of function pointers enhance the type safety and security.
- Support automatic memory managment with the help of Garbage Collector.
- Provides cross language integration because CTS inside CLR provides a common standard that activate the different languages to extend and share each other's libraries.
- Provides support to use the components that developed in other .NET programming languages.
- Provide language, platform, and architecture independency.
- It allows the creation of the scalable and multithreaded applications in a easier way as developer has no need to think about the memory management and security issues.

CLR does:

- have JIT Compiler program with itself which converts MSIL into native / binary

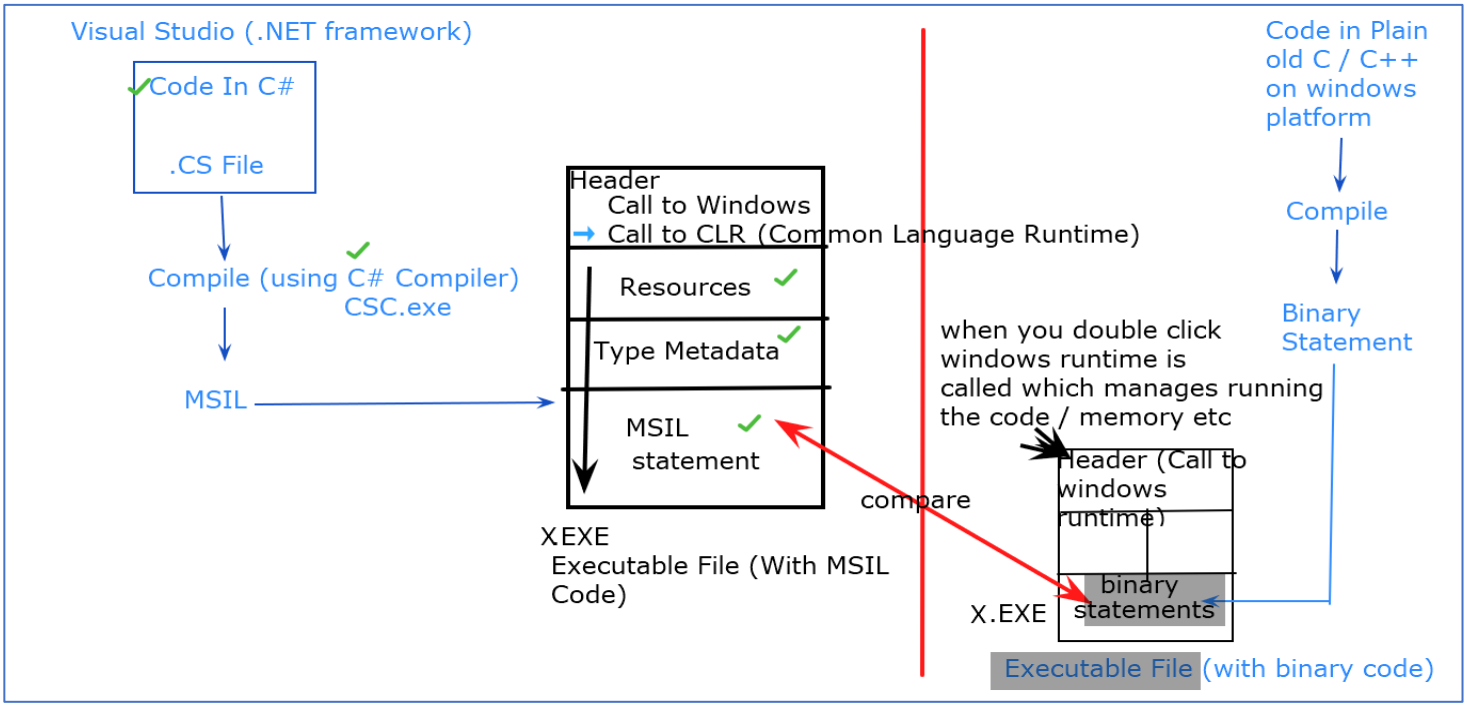
- allocate memory to your program as per demand

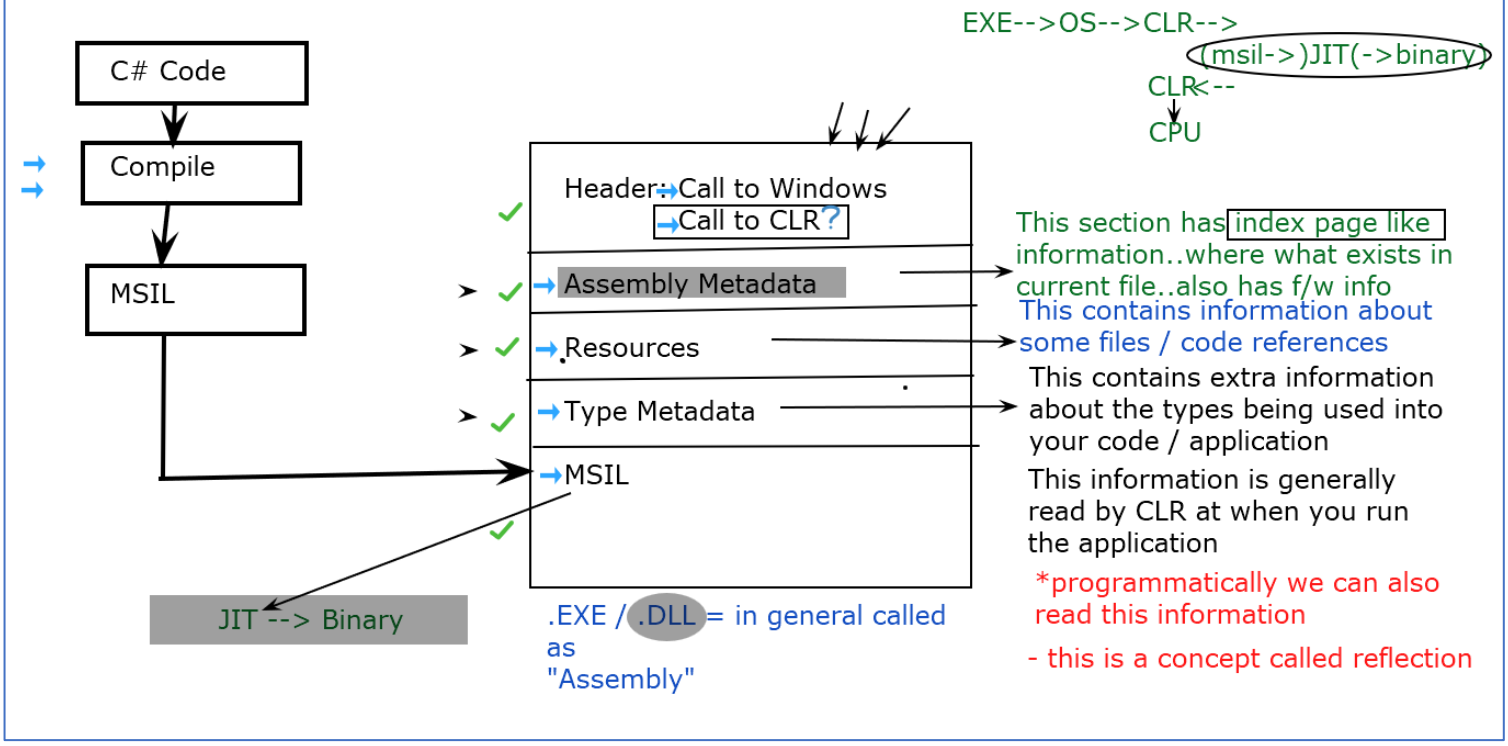
- de-allocate memory based on algorithm via - a program called Garbage Collection

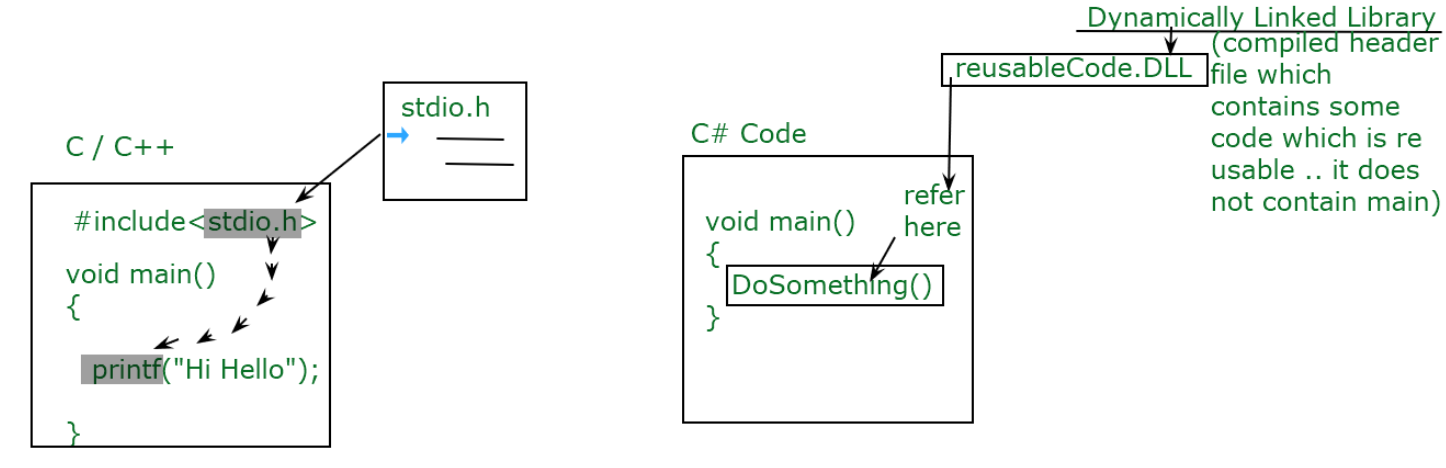
- Load Libraries (DLLs) based on demand

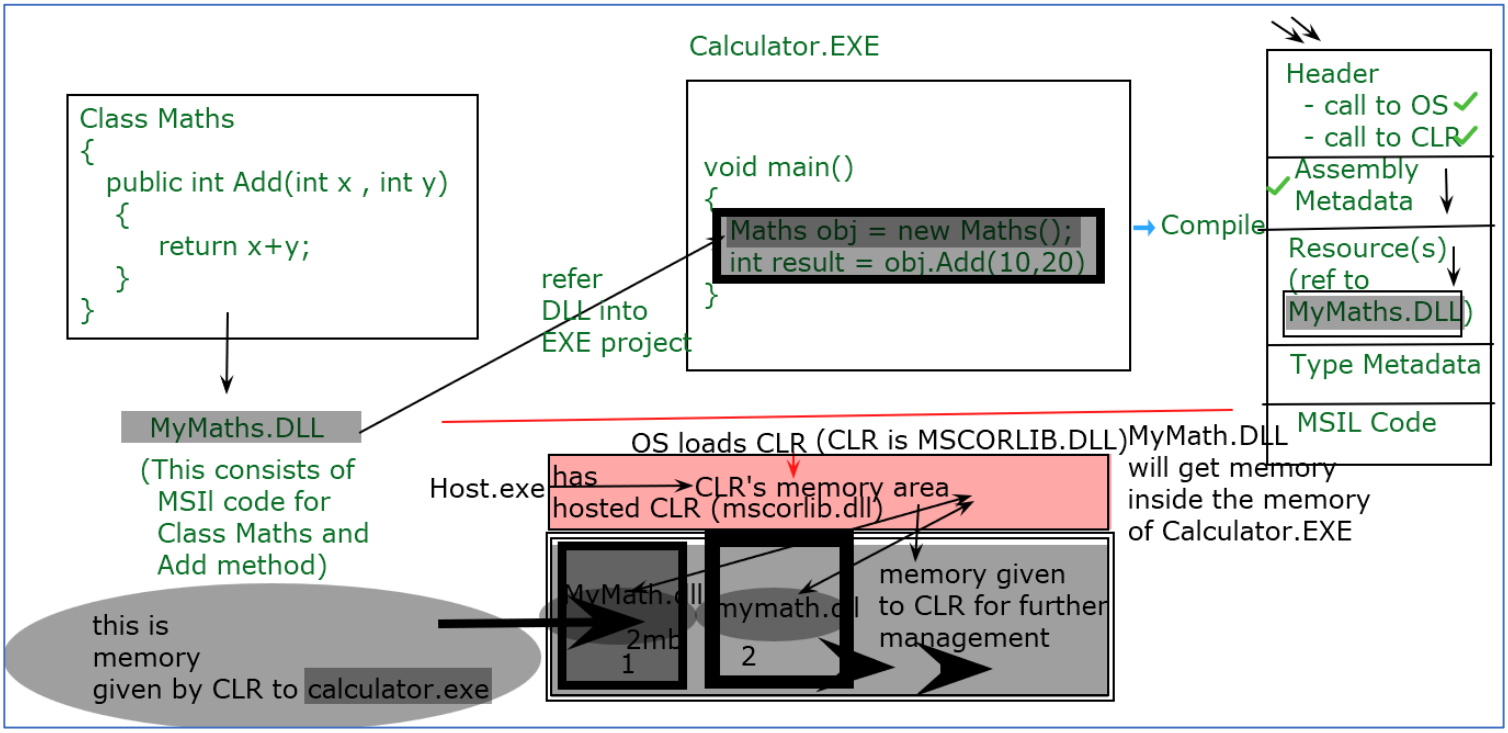
- Exception Handling

- COM [Comoponent Obejct Model - is a old standard for languages] Marshalling or Type conversion from old languages in 90's to current ones (eg. integer in VB 6.0 [from 90's] becomes short in C#)




EXE-->OS-->CLR-->
(msil->)JIT(->binary)
CLR<--
CPU
C# Code
Compile
MSIL
Header:→Call to Windows
→Call to CLR?
→Assembly Metadata
→Resources
→Type Metadata
→MSIL
JIT --> Binary
.EXE / .DLL = in general called as "Assembly"
This section has index page like information..where what exists in current file..also has f/w info
This contains information about some files / code references
This contains extra information about the types being used into your code / application
This information is generally read by CLR at when you run the application
*programmatically we can also read this information
- this is a concept called reflection

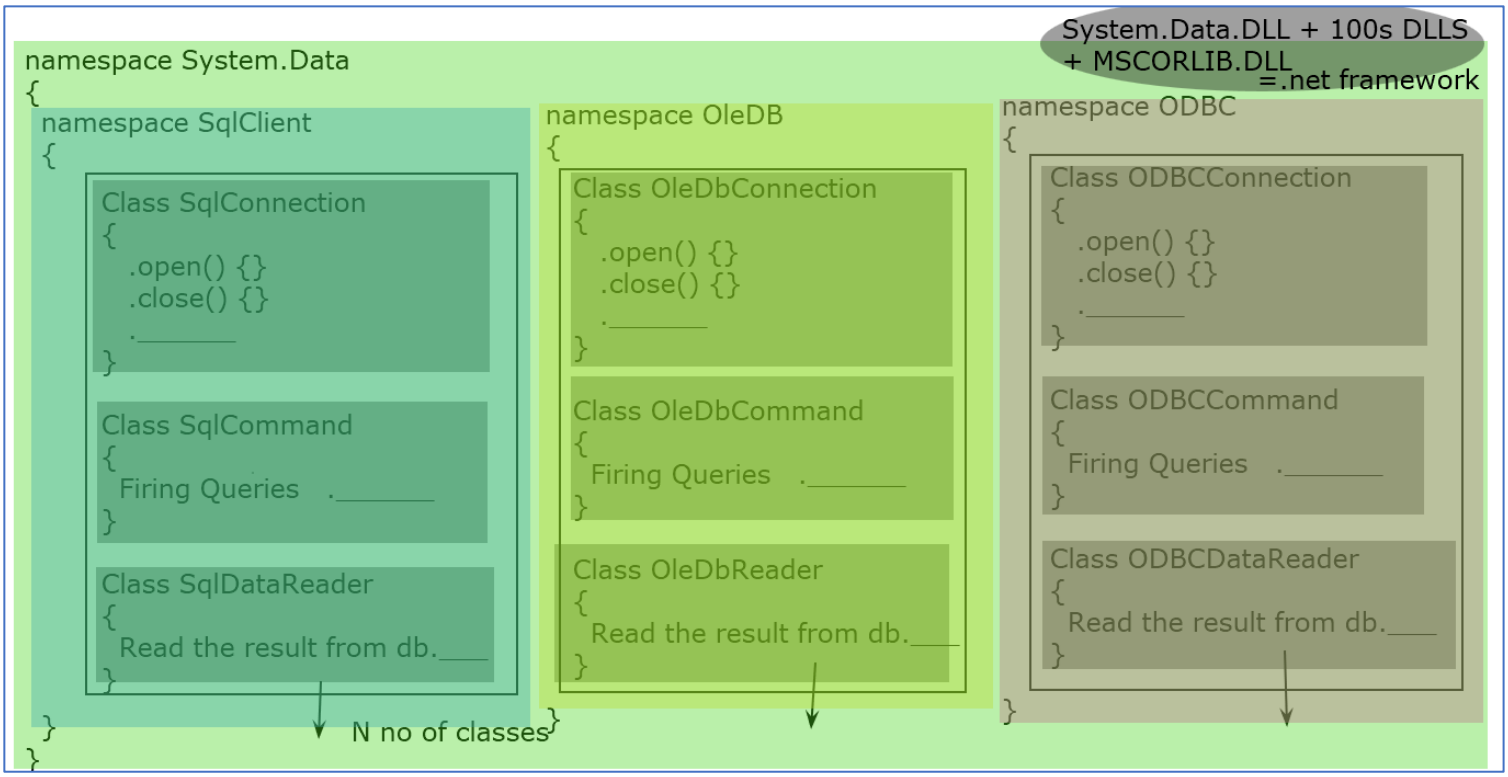

Dynamically Linked Library
(compiled header file which contains some code which is re usable .. it does not contain main)
reusableCode.DLL
stdio.h
C / C++
#include<stdio.h>
void main()
{
printf("Hi Hello");
}
C# Code
void main()
{
DoSomething()
}
refer here



What is .NET
Assembly , CLR, EXE, DLL
Assembly structure
Multiple language support

What does it mean .NET Framework
- For windows
- For other OS (.NET Core)

Capabilities of .NET



Day02_Agenda: 1. .Net Framework, .Net Core, Mono, Xamarin differences

.NET Framework

The .NET Framework was released in 2002 and primarily runs on Windows. It ships as part of the Windows operating system and updates through Windows Update when new versions become available or through optional standalone installers. It includes the class library (.NET Framework Class Library) which provides language interoperability among the supported languages and the Common Language Runtime (CLR), which is the environment where the code is executed.

.NET Core

.NET Framework is a monolithic framework that runs only on Windows and has to be installed for every machine that needs to run .NET apps on Windows. Its monolithic nature also caused a slow release cycle. In today's world where serverless and microservices architectures are heavily adopted, we need a lightweight framework with little memory footprint to run our applications, and the .NET framework doesn't play well in this space. However, Microsoft wants to innovate and satisfy this user base. Thus, .NET Core was built from scratch to lay the foundation for the future of the .NET platform and to include a flexible and modular architecture that would allow Microsoft to innovate on their offerings.

.NET Core is a lightweight, modular, and cross-platform implementation of .NET for creating modern web apps, microservices, libraries, and console applications that run on Windows, Linux, and macOS. .NET Core is composed of Nuget packages so that you can install only the modules or packages needed to run your app, and you can bundle them together with your application and your users do not need to install the framework on their machine.

Its modular architecture means a faster release cycle from the .NET team. It being lightweight, modular, fast and cross-platform means you can use it for microservices applications and deploy them in Linux containers. You can also run it on serverless platforms.

Xamarin

The .NET Framework primarily runs on Windows and if you use Linux, you can't develop .NET applications on Linux or publish your .NET programs to non-Windows operating systems. However, Microsoft later published the Common Language Infrastructure, which is an open specification developed by Microsoft and standardized by ISO and Ecma. With this standard, a version of the .NET Framework was developed to run on Linux and it is called Mono. Mono has evolved to support a wide range of operating systems and applications. Mono is a cross-platform implementation of the .NET Framework and it's used to build desktop and mobile apps for any operating system.

Xamarin is a company that started in 2011 with a focus on building a suite of tools that allowed cross-platform development with C#. They maintained the Mono project, including MonoTouch which allowed developers to build iOS apps with C#, and Mono for Android for building Android apps in C#. The Mono project has evolved since Xamarin took over and they released Xamarin.iOS, Xamarin.Android, Xamarin.Mac, and other tools that enable the development of cross-platform native apps in C#.

Xamarin.iOS and Xamarin.Android are implementations of Mono for iPhone and Android-based smartphones. In February 2016 Microsoft acquired Xamarin Inc. and has gradually integrated the Xamarin platform and tools with the Visual Studio and .NET offering. So today, Xamarin is a platform that extends the .NET developer platform with tools and libraries for building apps for Android, iOS, tvOS, watchOS,

macOS, and Windows. Xamarin apps use Mono as the .NET implementation for running the apps and you can create these projects from Visual Studio or Visual Studio for Mac. The unique features that Xamarin brings to the .NET ecosystem include:

- Base framework for accessing native features
- *Extensible Markup Language*, known as XAML, for building dynamic mobile apps using C#
- *Libraries for common patterns* such as Model View ViewModel (MVVM)
- Platform-specific libraries
- *Editor extensions* to provide syntax highlighting, code completion, designers, and other functionality specifically for developing mobile pages

2. Versions of the Framework

Release History of .NET Framework and its compatibility with the different Windows version

.NET Version	CLR Version	Development tool	Windows Support
1.0	1.0	Visual Studio .NET	XP SP1
1.1	1.1	Visual Studio .NET 2003	XP SP2, SP3
2.0	2.0	Visual Studio 2005	N/A
3.0	2.0	Expression Blend	Vista
3.5	2.0	Visual Studio 2008	7, 8, 8.1, 10
4.0	4	Visual Studio 2010	N/A
4.5	4	Visual Studio 2012	8
4.5.1	4	Visual Studio 2013	8.1
4.5.2	4	N/A	N/A
4.6	4	Visual Studio 2015	10 v1507
4.6.1	4	Visual Studio 2015 Update 1	10 v1511
4.6.2	4	N/A	10 v1607
4.7	4	Visual Studio 2017	10 v1703

.NET Version	CLR Version	Development tool	Windows Support
4.7.1	4	Visual Studio 2017	10 v1709
4.7.2	4	Visual Studio 2017	10v 1803

3. Managed and Unmanaged Code

.NET CLR Functions

Following are the functions of the CLR.

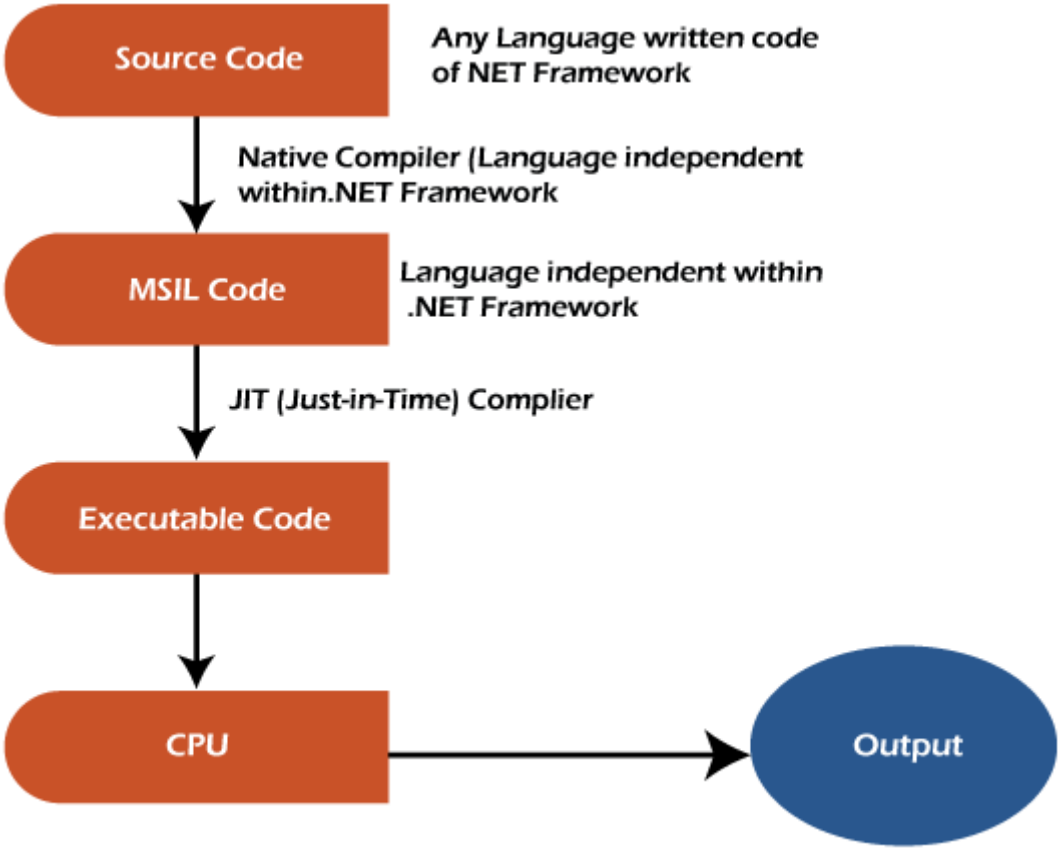
- It converts the program into native code.
- Handles Exceptions
- Provides type-safety
- Memory management
- Provides security
- Improved performance
- Language independent
- Platform independent
- Garbage collection
- Provides language features such as inheritance, interfaces, and overloading for object-oriented programs.

The code that runs with CLR is called managed code, whereas the code outside the CLR is called unmanaged code. The CLR also provides an Interoperability layer, which allows both the managed and unmanaged codes to interoperate.

1. Managed code:

Any language that is written in the .NET framework is managed code. Managed code use CLR, which looks after your applications by managing memory, handling security, allowing cross-language debugging, etc.

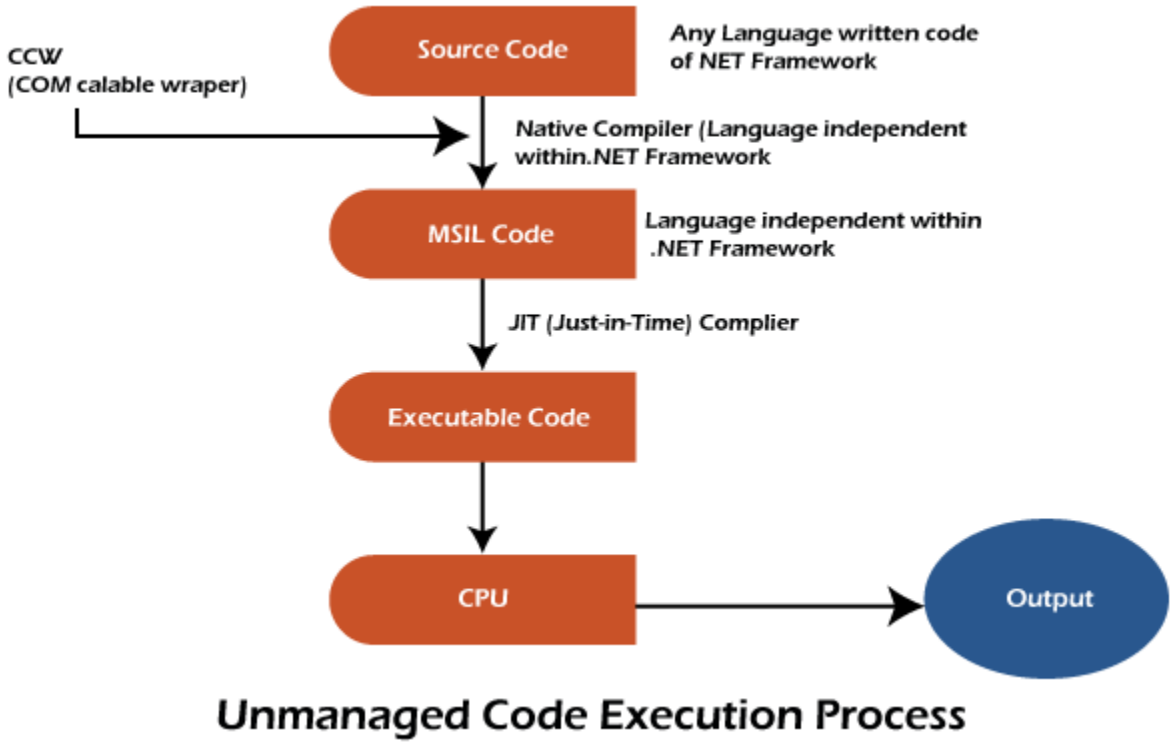
The process of managed code is shown in the figure:



Managed Code Execution Process

2. Unmanaged code:

The code developed outside the .NET framework is known as unmanaged code. Applications that do not run under the control of the CLR are said to be unmanaged. Certain languages such as C++ can be used to write such applications, such as low-level access functions of the operating system. Background compatibility with VB, ASP, and COM are examples of unmanaged code. This code is executed with the help of wrapper classes. The unmanaged code process is shown below:



Unmanaged Code Execution Process

.NET CLR Versions

The CLR updates itself time to time to provide better performance.

.NET version	CLR version
1.0	1.0
1.1	1.1
2.0	2.0
3.0	2.0
3.5	2.0
4	4
4.5	4
4.6	4

4.6	4

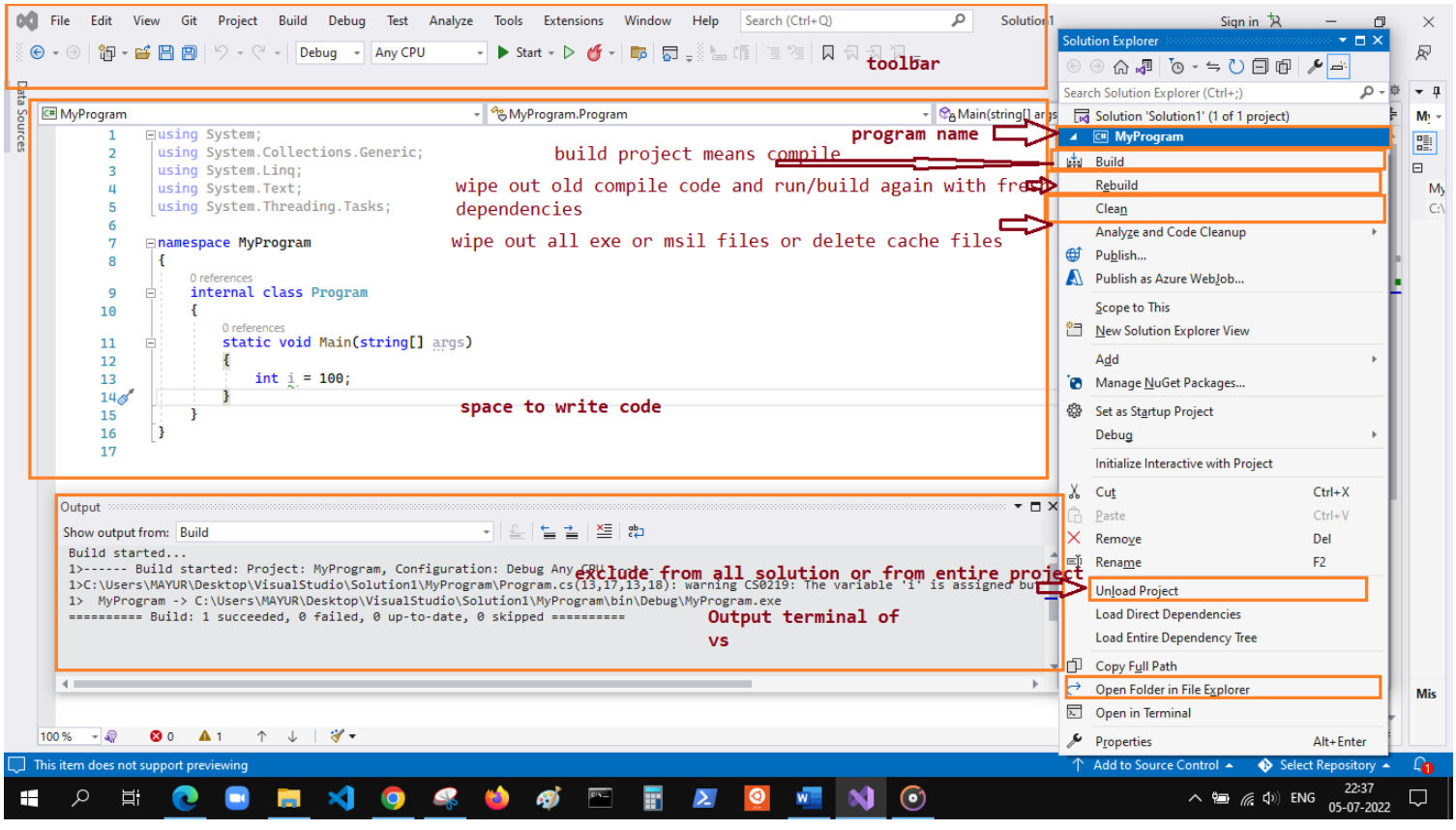
4. Using ILDASM

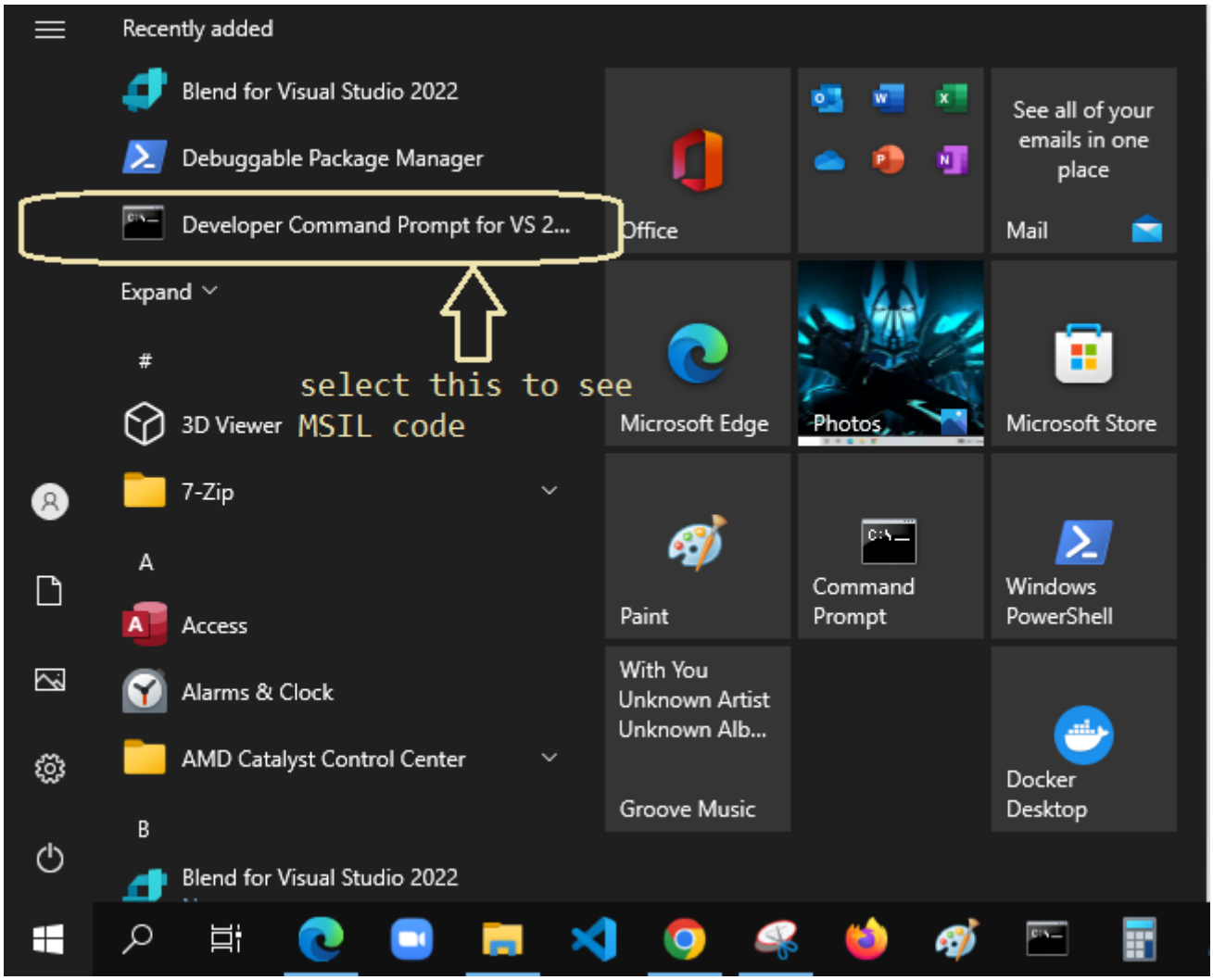
How to see MSIL code via Visual Studio developer command prompt

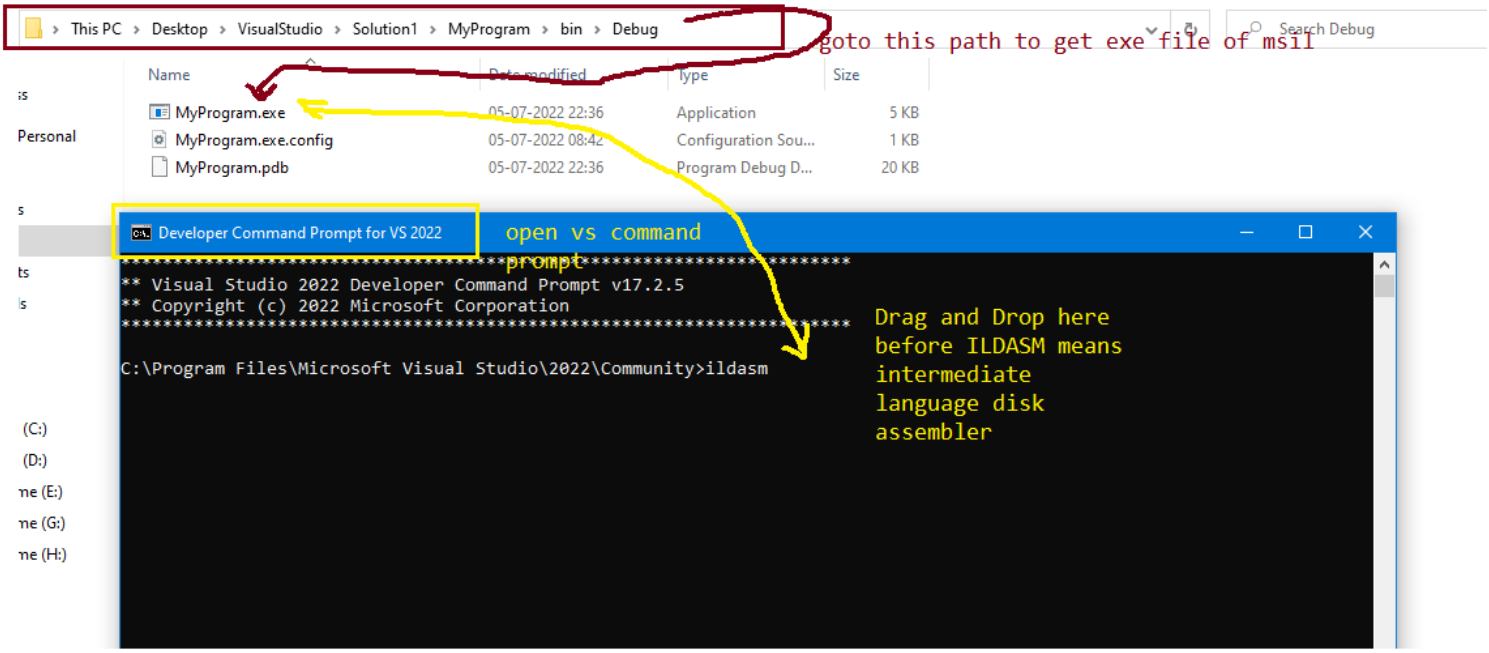
Using ILDASM

CLR does:

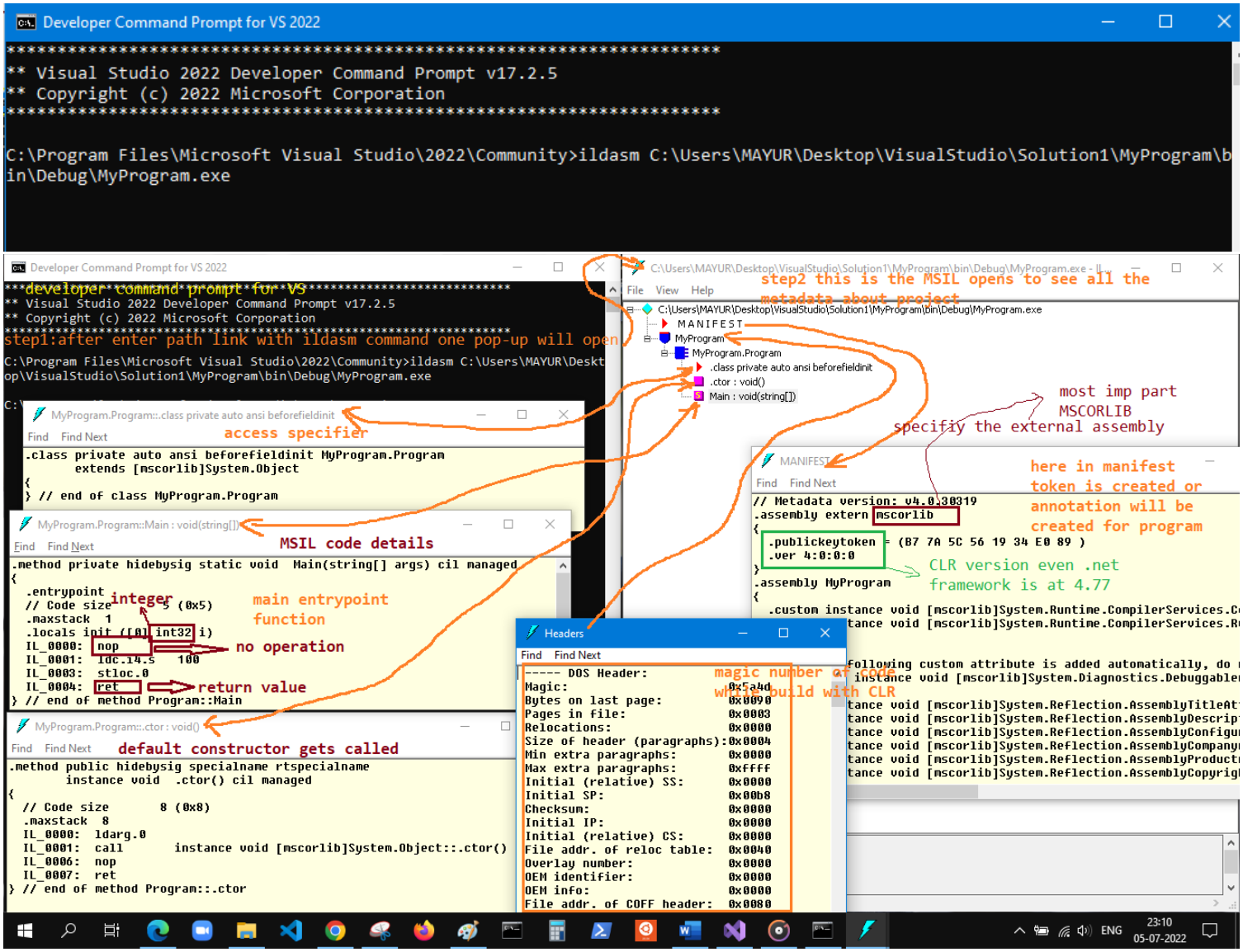
- have JIT Compiler program with itself which converts MSIL into native / binary
- allocate memory to your program as per demand
- de-allocate memory based on algorithm via - a program called Garbage Collection
- Load Libraries (DLLs) based on demand
- Exception Handling
- COM [Comoponent Obejct Model - is a old standard for languages] Marshalling or Type conversion from old languages in 90's to current ones (eg. integer in VB 6.0 [from 90's] becomes short in C#)



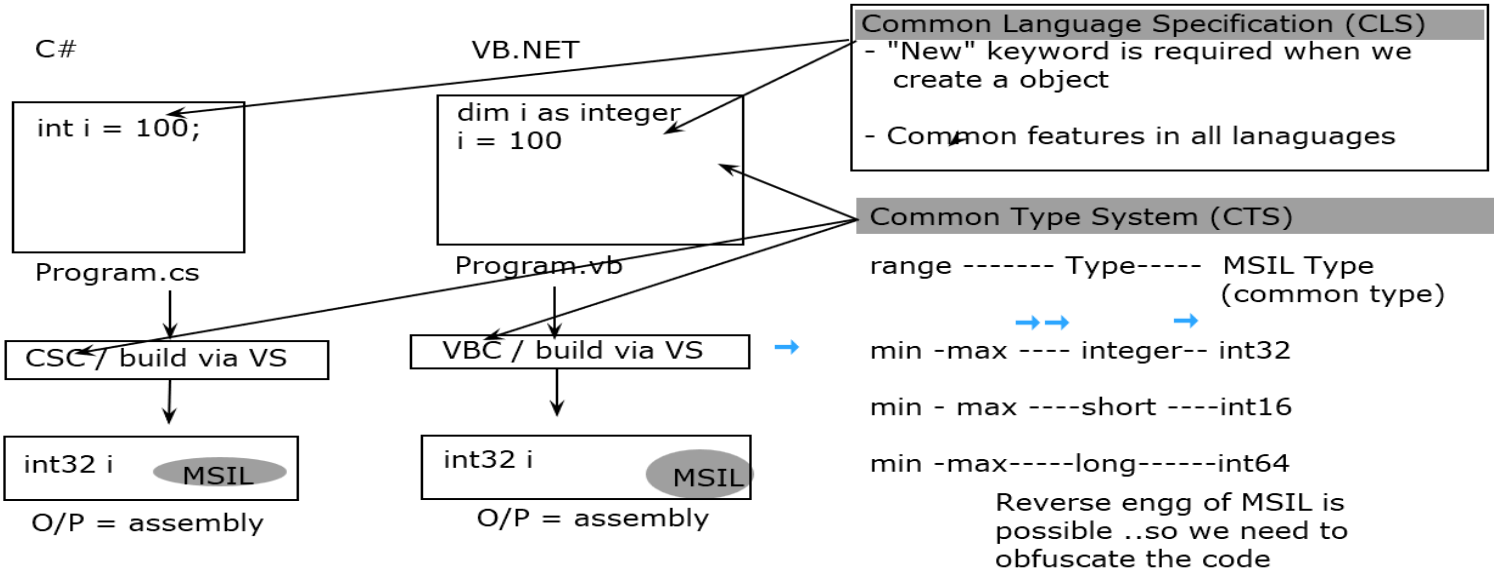


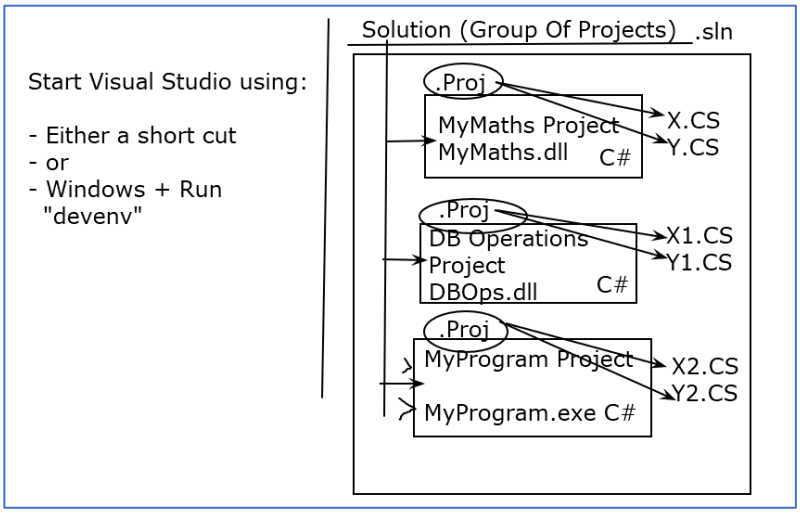


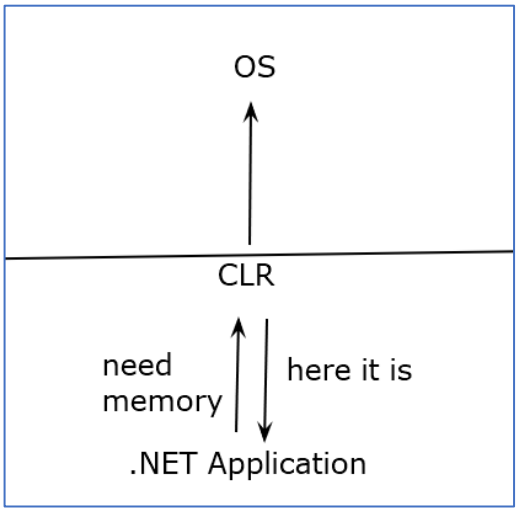
after pasting the path link



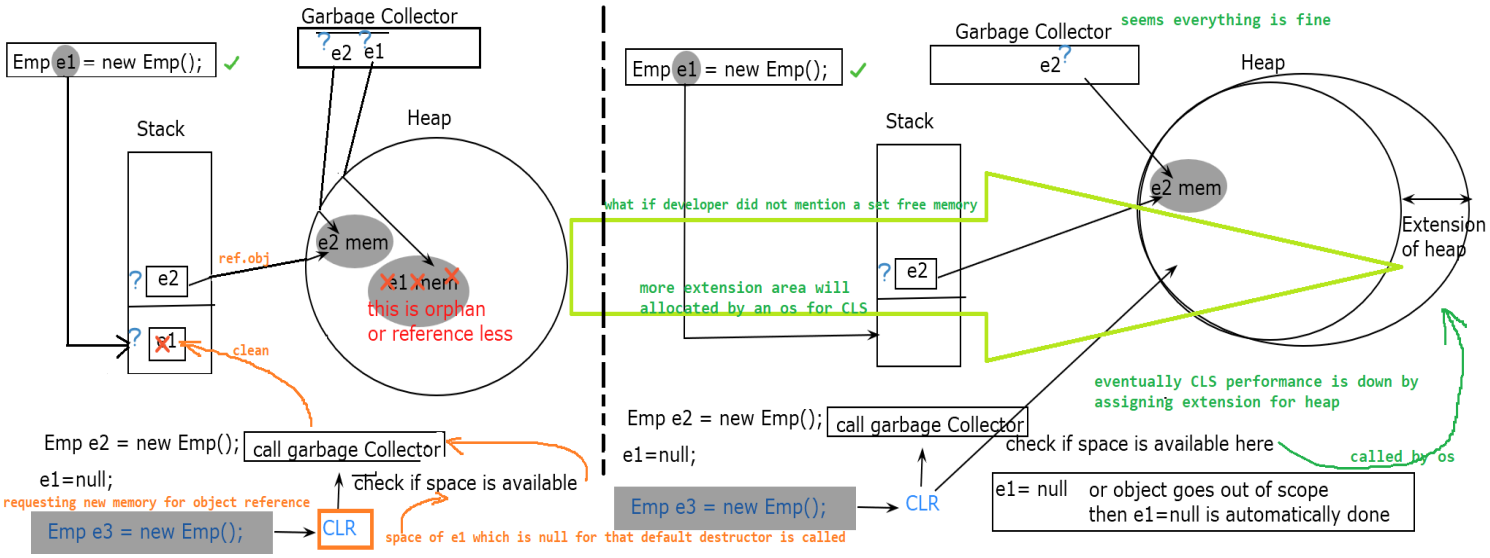
After this MSIL code build, it will hand over to host by os while running of project and then to CLR gives the MSIL code to JIT it will convert it into native code/byte code for machine understandable language after that it will load on RAM.







→Working of Garbage collection in Visual studio code by CLR and memory management of gc



- when you create a object, memory is asked to CLR
- CLR will allocate space when it sees its available
- if not - it would call GC

- GC will compare the entries that it has vs entries over stack
- if it find some reference it has but stack does not have
  --which means either develper must have set up obj = null or object must have gone out of scope

- if GC comes back to CLR telling nothing can be cleaned up ...
  -- CLR will escalate a call to OS
    - we will get extension of memory
      -- which is bad ... bcoz now CLR will have to manage more memory ..
        and its performance can go down..

So, good approach is

1. set obj = null at the earliest in the sense ... when we get to know we dont need it.
2. declare and define scoped objects .. where it will be automatically set = null when scope is over!
3. Avoid global & static objects as far as possible

C# Garbage Collection

Garbage collection is a key component of many modern programming languages, including C#. It's even hard to imagine what programming would look like in C#, and other modern languages like Java, Ruby, and many others, without this tool.

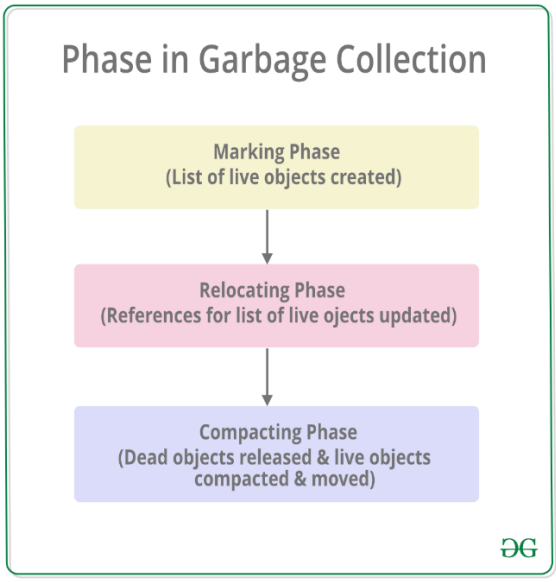
Despite being a valuable asset that makes a better programming experience, garbage collection can still give you a hard time, specifically with performance.

With that in mind, what can a C# developer do to ensure that C# garbage collection acts as a friend instead of a foe? How can you write code in such a way that you reap all the benefits of this tool without suffering from any of the issues it can cause?

What is garbage collection?

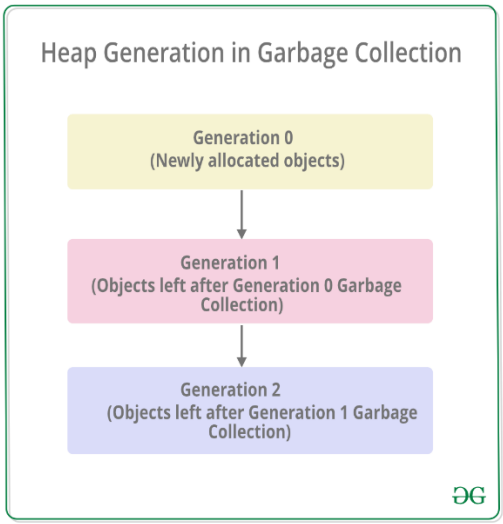
What is garbage? It's something that was once useful, but it's not anymore (like a broken device). Or it might be residues from some activity (vegetable peels, for instance?) In short, garbage is things you want to get rid of, because it wastes space or potentially can cause harm.
Guess what? Our programs also generate garbage. Think about objects that were created, performed their jobs and are now useless, but are still there occupying valuable space in memory. Shouldn't we get rid of them? We should indeed, and this process is sometimes called "memory management."
Older languages required developers to manage memory manually. They'd have to mark objects that were no longer in use as dead or inactive, freeing the memory used by these objects as available for the program.
The problem with this is that manually freeing objects could be an extremely hard and error-prone process. Failure in terminating obsolete objects often resulted in memory leaks. Terminating non-obsolete objects, on the other hand, could result in runtime errors and inconsistent behavior. In short: a real pain in the neck. Manual memory management prevented developers from fully focusing on the business logic of whatever applications they were writing. Instead, it put them in a constant state of worry. Talk about cognitive load!
In response to those problems, garbage collection was created. So here's our definition for garbage collection:

Garbage collection is an automated process that is able to figure out which objects are no longer needed and get rid of them, freeing space in memory.



1. **Marking Phase:** A list of all the live objects is created during the marking phase. This is done by following the references from all the root objects. All of the objects that are not on the list of live objects are potentially deleted from the heap memory.
2. **Relocating Phase:** The references of all the objects that were on the list of all the live objects are updated in the relocating phase so that they point to the new location where the objects will be relocated to in the compacting phase.
3. **Compacting Phase:** The heap gets compacted in the compacting phase as the space occupied by the dead objects is released and the live objects remaining are moved. All the live objects that remain after the garbage collection are moved towards the older end of the heap memory in their original order.

How GC works in C#

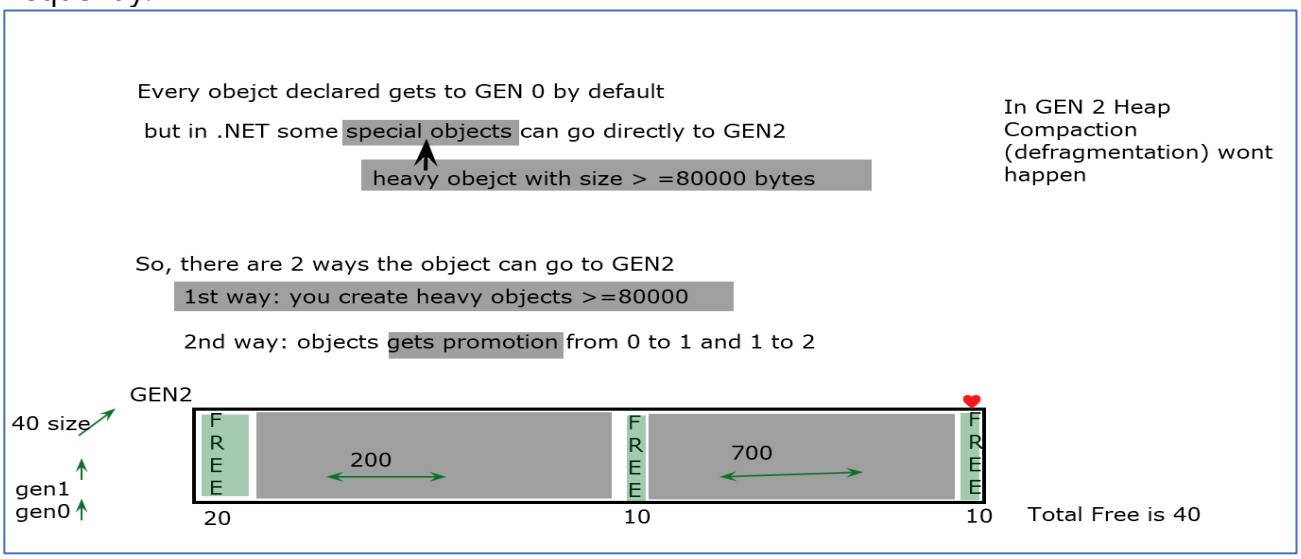


C# garbage collection belongs to the tracing variety. It's often called a generational approach since it employs the concept of generations to figure out which objects are eligible for collection.
Memory is divided into spaces called generations. The collector starts claiming objects in the youngest generation.

Then it promotes the survivors to the next generation. The C# garbage collection uses three generations in total:

- Generation 0—This generation holds short-lived objects. Here's where the collection process happens most often. When you instantiate a new object, it goes in this generation by default. The exceptions are objects whose sizes are equal to or greater than 85,000 bytes. Those objects are large, so they go straight into generation 2.
- Generation 1—This is an intermediate space between the short-lived and long-lived layers.

- Generation 2—Finally, this is the generation that holds objects that live the longest in the application—sometimes as long as the whole duration of the app. GC takes place here less frequently.



According to the Microsoft docs, the following information is what GC uses to determine if an object is live:

- **Stack roots**. Stack variables provided by the just-in-time (JIT) compiler and stack walker. Note that JIT optimizations can lengthen or shorten regions of code within which stack variables are reported to the garbage collector.
- **Garbage collection handles**. Handles that point to managed objects and that can be allocated by user code or by the common language runtime.
- **Static data**. Static objects in application domains that could be referencing other objects. Each application domain keeps track of its static objects.

Before the start of a collection process, the collector stops all threads, except for the one responsible for triggering the collection. Then, the collection happens, following these steps:

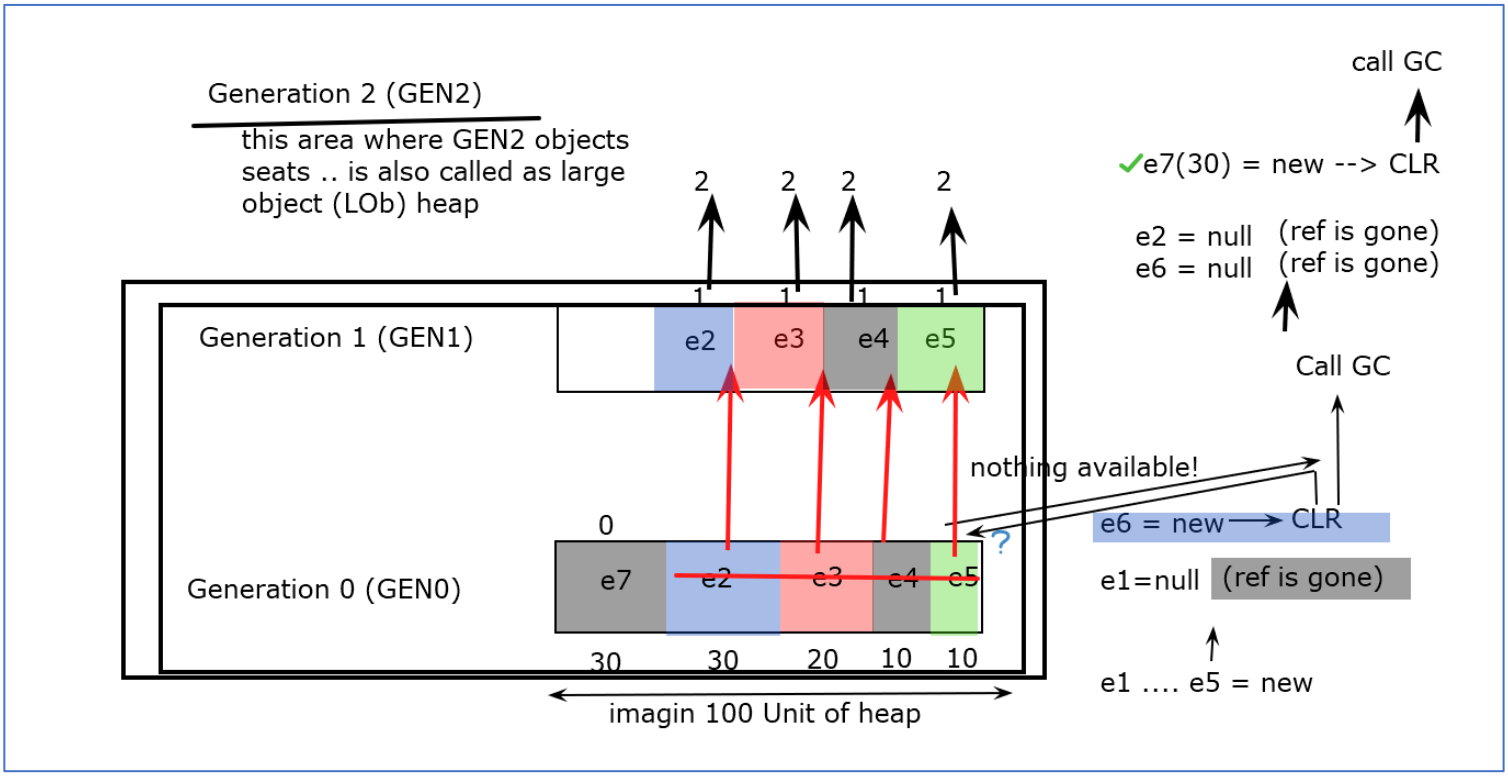
1. The collector fetches all live objects, starting with the root objects cited above.
2. The references to the objects that will be compacted are updated.
3. Finally, the collector eliminates dead objects, reclaims their space, and promotes them to the next generation.

1. Set object = null at the earliest in code .. when u see it is not longer going to get used.
2. Use Scoped objects in code
3. Avoid declaring - larger scope / global / static objects
4. Avoid using larger objects >= 80000 bytes

Use C# GC to your advantage

Developers new to GC will sometimes ask, "When is it appropriate to force the collection to occur?" And the answer is: (almost) never. Think about it. The whole point of this GC thing is to free you from having to manually manage memory. Wouldn't it be self-defeating to make you manually trigger the collection process?

"Fine," you might say. "But is there anything I can *do* about all that? Is there some way to write code so I don't put unnecessary stress on GC?"
Sure, there are some things you can do. You've just learned that generation 0 is the place where collection happens most often. And which kinds of objects live there? You guessed it: short-lived ones. So, one way to avoid putting additional pressure on the GC is to avoid excessive memory allocations, especially objects you know will have short lives.
You can also use structs. These are value objects and, as such, live on the stack. By using structs when it makes sense to, you avoid extra allocations that put more pressure on the GC.



Day03_Agenda: - 1. Console Applications and Class Libraries (Framework and .Net Core)

2. C# Basics

3. Project References, using Classes Data Types in .net and CTS equivalents

4. Methods

- **Method Overloading**
- **Optional Parameters**
- **Named Parameters and Positional Parameters**
- **Using params**
- **Local functions Properties**
- **get, set**
- **Readonly properties**
- **Using property accessors to create Readonly property**

Constructors

Object Initializer
Destructors
Discussion on IDispose. To be implemented after interfaces

Today's point from live lecture –session-01→ 06/07/2022

System.console.writeline(" "); → write on new line
To avoid system name to be written
Using system above namespace
console.writeline(" ");

Press f12 gives you declaration of methods
Press f1 online documentation

Console.ReadLine(); → to pause runtime result/readline will return as String
If you press enter it will terminate

Ri8 click set as startup project another project to be start for execution

Ctrl+k+c for // comment

#Region→ region is ignored in program while compilation
Type re then enter Tab button twice
Is used for readability to enclosed the comment body
#endregion

Ri8 click on commented code goto snippet surround with #region

If else tab tab

For loop tab tab
After that replace i with k press tab it will reflect all changes

Cw tab tab → console.writeline();

While tab tab

Do while tab tab

Switch tab tab

Try catch block tab tab

Typecast→ convert.ToInt32(console.ReadLine();)

Debugging of code→insert breakpoint or click on vertical bar on red dot which is at left side

F10 will jump to next line while breakpoints are applied

TESTING THE CODE→
Immediate window for debugging by ri8 click on code

What if? Execution gone further to revert it click on red dot yellow arrow hold it and drag to upside for re-execution
Only if debugger is on.....

Method overloading→

Interface

1. Console Applications and Class Libraries (Framework and .Net Core)

SESSION-01 programs of C# →

```
using System;
namespace AnotherProgram
{
   class Program
   {
      static void Main(string[] args)
      {
         Console.WriteLine("hi");
         Console.WriteLine("bye");
         Console.ReadLine();
      }
   }
}
```

2. C# Basic Syntax

C# is an object-oriented programming language. In Object-Oriented Programming methodology, a program consists of various objects that interact with each other by means of actions. The actions that an object may take are called methods. Objects of the same kind are said to have the same type or, are said to be in the same class.
For example, let us consider a Rectangle object. It has attributes such as length and width. Depending upon the design, it may need ways for accepting the values of these attributes, calculating the area, and displaying details.
Let us look at implementation of a Rectangle class and discuss C# basic syntax –

```
using System;

namespace RectangleApplication {
  class Rectangle {

    // member variables
```

```
    double length;
    double width;

    public void Acceptdetails() {
      length = 4.5;
      width = 3.5;
    }
    public double GetArea() {
      return length * width;
    }
    public void Display() {
      Console.WriteLine("Length: {0}", length);
      Console.WriteLine("Width: {0}", width);
      Console.WriteLine("Area: {0}", GetArea());
    }
  }
  class ExecuteRectangle {
    static void Main(string[] args) {
      Rectangle r = new Rectangle();
      r.Acceptdetails();
      r.Display();
      Console.ReadLine();
    }
  }
}
```

When the above code is compiled and executed, it produces the following result –

```
Length: 4.5
Width: 3.5
Area: 15.75
```

The *using* Keyword

The first statement in any C# program is

```
using System;
```

The **using** keyword is used for including the namespaces in the program. A program can include multiple using statements.

The *class* Keyword

The **class** keyword is used for declaring a class.

Comments in C#

Comments are used for explaining code. Compilers ignore the comment entries. The multiline comments in C# programs start with /* and terminates with the characters */ as shown below –

```
/* This program demonstrates
```

```
The basic syntax of C# programming
Language */
```

Single-line comments are indicated by the '//' symbol. For example,

```
}//end class Rectangle
```

Member Variables

Variables are attributes or data members of a class, used for storing data. In the preceding program, the *Rectangle* class has two member variables named *length* and *width*.

Member Functions

Functions are set of statements that perform a specific task. The member functions of a class are declared within the class. Our sample class Rectangle contains three member functions: *AcceptDetails*, *GetArea* and *Display*.

Instantiating a Class

In the preceding program, the class *ExecuteRectangle* contains the *Main()* method and instantiates the *Rectangle* class.

Identifiers

An identifier is a name used to identify a class, variable, function, or any other user-defined item. The basic rules for naming classes in C# are as follows −

- A name must begin with a letter that could be followed by a sequence of letters, digits (0 - 9) or underscore. The first character in an identifier cannot be a digit.
- It must not contain any embedded space or symbol such as? - + ! @ # % ^ & * ( ) [ ] { } . ; : " ' / and \. However, an underscore (_) can be used.
- It should not be a C# keyword.

C# Keywords

Keywords are reserved words predefined to the C# compiler. These keywords cannot be used as identifiers. However, if you want to use these keywords as identifiers, you may prefix the keyword with the @ character.

In C#, some identifiers have special meaning in context of code, such as get and set are called contextual keywords.

The following table lists the reserved keywords and contextual keywords in C# −

Reserved Keywords						
abstract	as	base	bool	break	byte	case
catch	char	checked	class	const	continue	decimal
default	delegate	do	double	else	enum	event
explicit	extern	false	finally	fixed	float	for

foreach	goto	if	implicit	in	in (generic modifier)	int
interface	internal	is	lock	long	namespace	new
null	object	operator	out	out (generic modifier)	override	params
private	protected	public	readonly	ref	return	sbyte
sealed	short	sizeof	stackalloc	static	string	struct
switch	this	throw	true	try	typeof	uint
ulong	unchecked	unsafe	ushort	using	virtual	void
volatile	while					
Contextual Keywords						
add	alias	ascending	descending	dynamic	from	get
global	group	into	join	let	orderby	partial (type)
partial (method)	remove	select	set			

4. Method Overloading

- ***Method Overloading*** is the common way of implementing polymorphism. It is the ability to redefine a function in more than one form. A user can implement function overloading by defining two or more functions in a class sharing the same name. C# can distinguish the methods with **different method signatures**. i.e. the methods can have the same name but with different parameters list (i.e. the number of the parameters, order of the parameters, and data types of the parameters) within the same class.
  - Overloaded methods are differentiated based on the number and type of the parameters passed as arguments to the methods.
  - You can not define more than one method with the same name, Order and the type of the arguments. It would be compiler error.
  - The compiler does not consider the return type while differentiating the overloaded method. But you cannot declare two methods with the same signature and different return type. It will throw a compile-time error. If both methods have the same parameter types, but different return type, then it is not possible.

Why do we need Method Overloading?
If we need to do the same kind of the operation in different ways i.e. for different inputs. In the example described below, we are doing the addition operation for different inputs. It is hard to find many different meaningful names for single action.

Different ways of doing overloading methods-
Method overloading can be done by changing:

1. The number of parameters in two methods.
2. The data types of the parameters of methods.
3. The Order of the parameters of methods.

By changing the Number of Parameters

By changing the Data types of the parameters

By changing the Order of the parameters

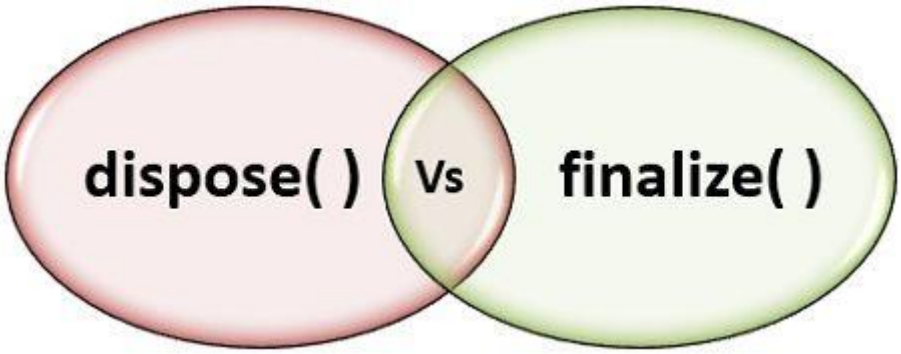
- **Optional Parameters**
- **Named Parameters and Positional Parameters**
- **Using params**
- **Local functions Properties**
- **get, set**
- **Readonly properties**
- **Using property accessors to create Readonly property**

Constructors
Object Initializer
Destructors

No need almost same as Java language

Discussion on IDispose.

Difference Between dispose() and finalize() in C#



Methods dispose() and finalize() are the methods of C# which are invoked to free the unmanaged resources held by an object. The dispose() method is defined inside the interface IDisposable whereas, the method finalize() is defined inside the class object. The main difference between dispose() and finalize() is that the method **dispose**() has to be explicitly invoked by the user whereas, the method **finalize()** is invoked by the garbage collector, just before the object is destroyed.

BASIS FOR COMPARISON	DISPOSE( )	FINALIZE( )
Defined	The method dispose( ) is defined in the interface IDisposable interface.	The method finalize( ) id defined in java.lang.object class.

Syntax	public void Dispose( ){
	// Dispose code here
	}
		protected void finalize( ){
		// finalization code here
		}

Invoked	The method dispose( ) is invoked by the user.	The method finalize( ) is invoked by the garbage collector.
Purpose	Method dispose( ) is used to free unmanaged resources whenever it is invoked.	Method finalize( ) is used to free unmanaged resources before the object is destroyed.
Implementation	The method dispose( ) is to be implemented whenever there is a close( ) method.	The method finalize( ) is to be implemented for unmanaged resources.
Access specifier	The method dispose( ) is declared as public.	The method finalize( ) is declared as private.
Action	The method dispose( ) is faster and instantly disposes an object.	The method finalize is slower as compared to dispose
Performance	The method disposes( ) performs the instantaneous action hence, does not effect the performance of websites.	The method finalize( ) being slower affects the performance of the websites.

Definition of dispose()

The dispose() method releases the unmanaged resources that are held by an object of the class. The unmanaged resources are files, data connections, etc. The method dispose() is declared in the interface **IDisposeable** and it is implemented by the class by implementing the interface IDisposable. This method is not called automatically. The programmer has to implement it manually when you are creating a custom class that will be used by others. The method has the following syntax:

```
public void dispose( ){
// Dispose code here
}
```

In the above syntax, you can observe that the method is declared as public. It is because this method is defined in the interface IDisposable and it has to be implemented by the class that implements this interface. So, to provide accessibility to the implementing class, the method is declared as public.

This method is invoked manually by the code of a program as it is implemented to invoke. The methods performance is fast, and it instantly frees the resources held by the object of a class.

Definition of finalize()

The finalize() method is defined in the **object** class. It is used for cleanup activities. This method is called by the garbage collector when the reference of an object is not used for a long time. Garbage collector frees the managed resources automatically but if you want to free the unmanaged resources like file handle, data connection, etc., the finalize method has to be implemented manually. The garbage collector invokes the method finalize() just before it destroys the object completely.

The syntax of the method finalize():

```
protected void finalize( ){
// finalization code here
}
```

In the syntax above, the method finalize() is declared as protected. The reason behind this is, the method finalize() must not be accessible from outside the class, and it must only be accessible to the garbage collector.

The finalize() method affects the cost of the performance as it does not free the memory instantly. In C# the finalize method is called automatically with destructors.

Key Differences Between dispose() and finalize()

1. The method dispose() is defined in an interface **IDisposable**. On the other hand, the method finalize() is defined in the class **object**.

2. The method dispose() has to be manually invoked inside the code by a programmer, while the method finalize is automatically invoked by the garbage collector before it destroys the object.
3. The method dispose could be invoked anytime, whereas the method finalize is invoked by the garbage collector when it finds that that object has not been referenced for a long time.
4. The method dispose() is implemented in a class after implementing the interface IDisposable. The method finalize() has to be implemented only for **unmanaged resources** because the managed resources are automatically freed by the garbage collector.
5. The access specifier of the method dispose() is public as it is defined in the interface IDisposable and it would be implemented by the class that implements this interface hence, it should be public. On the other hand, the method finalize() has protected access specifier so that it should not be accessible to any member outside the class.
6. The method dispose() is fast and frees the object instantly hence, it does not affects the performance cost. The method finalize() is slower and does not free the resources held by the object instantly.

Conclusion

It is suggested to use method dispose() over the method finalize() as it is faster than finalize. Also, it could be called any time, when needed.

```
using System; //to avoid writing of System.Console.WriteLine();--> Console.WriteLine();

namespace MyProgram
{
  public class Program
   {
      static void Main(string[] args)
      {

         #region Some basic code to start off
         ////ctrl+k+c = comment
         ////ctrl + k + u = uncomment
```

```
        //Print(100);
        //Console.WriteLine("Press Enter to terminate the program");
        //Console.ReadLine();
        #endregion

        #region Basic Declarations
        //int i = 100;
        //Console.WriteLine(i);
        #endregion

        #region If Loop Demo
        //int i = 100;

        //if (i > 20)
        //{
        //    Console.WriteLine("i is greater than 20");
        //}
        //else
        //{
        //    Console.WriteLine("i is less than 20");
        //}
        #endregion

        #region For Demo
        //for (int k = 0; k < 10; k++) //type for and press tab twice, replace i with k then pres tab to reflect all changes
        //{
        //    Console.WriteLine(k);
        //}
        #endregion

        #region Code Snippets
        //while (true) //type while press tab twice
        //{
        //}

        //do        //type do press tab twice.
        //{
        //} while (true);

        //switch (switch_on) //type switch press tab twice
        //{
        //    default:
        //}
```

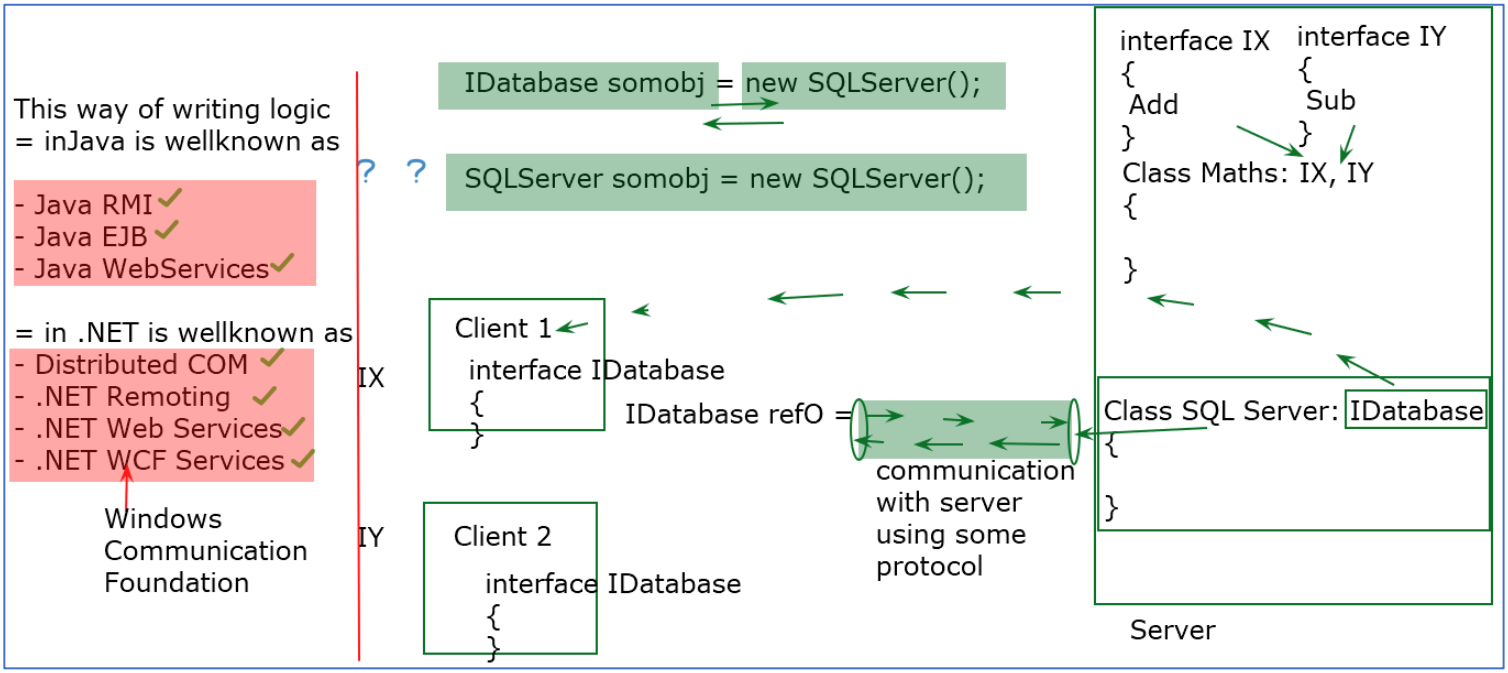
```
        #endregion

        #region Create Simple Object and Call Maths Class Methods
        Maths obj = new Maths();

        Console.WriteLine("Enter value for X");
        string xInStringFormat = Console.ReadLine();
        int x = Convert.ToInt32(xInStringFormat);

        Console.WriteLine("Enter value for Y");
        int y = Convert.ToInt32(Console.ReadLine());

        int additionResult = obj.Add(x, y);

        Console.WriteLine("Result = " + additionResult);

        Console.ReadLine();
        #endregion

        #region MyRegion
        AdvancedMaths advancedMaths = new AdvancedMaths();

        int result = advancedMaths.Square(10);
        Console.WriteLine(result);
        int addResult = advancedMaths.Add(10, 20);
        Console.WriteLine(addResult);
        Console.ReadLine();
#endregion
    }
  }
  public class Maths
  {
    public virtual int Add(int x, int y)
    {
      return x + y;
    }
    public int Sub(int x, int y)
    {
      return x - y;
    }
    public int Mult(int x, int y)
    {
      return x * y;
    }
  }
```

```
  public class AdvancedMaths : Maths
   {
      public int Square(int x)
      {
         return x * x;
      }
      //over riding
      public override int Add(int x, int y) //signature is same
      {
         return x + y + 100; //logic is different than base
      }
      //below code can be considered as overloading across classes
      public int Add(int x , int y , int z)
      {
         return x + y + z;
      }
   }
}
```

```
#region Simple interface with no usage in Main method code
using System;

namespace DemoOOP
{
   class Program
   {
      static void Main(string[] args) //step-4
      {
         Console.WriteLine("Tell us what you want: 1: SQL Server, 2: Oracle");
         int choice = Convert.ToInt32(Console.ReadLine());

         switch (choice) //main switch case
         {
            case 1: //case 1 for Oracle database
               SQLServer obj = new SQLServer();

               Console.WriteLine("Tell us what you want: 1: Insert, 2: Update, 3: Delete");
               int opchoice = Convert.ToInt32(Console.ReadLine());

               switch (opchoice) //switch case for SQLServer
               {
                  case 1:
                     obj.Insert();
                     break;
                  case 2:
```

```
                    obj.Update();
                    break;
                case 3:
                    obj.Delete();
                    break;
                default:
                    Console.WriteLine("Invalid choice of operation");
                    break;
            }
            break;
        case 2: //case 2 for Oracle database
            Oracle obj2 = new Oracle();
Console.WriteLine("Tell us what you want: 1: Insert, 2: Update, 3: Delete");
            int opchoice1 = Convert.ToInt32(Console.ReadLine());
            switch (opchoice1)
            {
                case 1:
                    obj2.Insert();
                    break;
                case 2:
                    obj2.Update();
                    break;
                case 3:
                    obj2.Delete();
                    break;
                default:
                    Console.WriteLine("Invalid choice of operation");
                    break;
            }
            break;
        default:
            Console.WriteLine("Invalid choice");
            break;
      } //switch case scope main
      Console.ReadLine();
    } //main scope
  } //class program scope

  public interface OperationDatabase
  {
    void Insert(); //undeclared functions
    void Update(); //undeclared functions
    void Delete(); //undeclared functions
  }//step-1  Base class
```

```
    public class SQLServer : OperationDatabase
    {
        public void Insert() //functions of OperationDatabase
        {
            Console.WriteLine("SQL Insert called");
        }

        public void Update()//functions of OperationDatabase
        {
            Console.WriteLine("SQL Update called");
        }

        public void Delete()//functions of OperationDatabase
        {
            Console.WriteLine("SQL Delete called");
        }
    } //step-2 SQLServer implements OperationDatabase Derived class

    public class Oracle : OperationDatabase
    {
        public void Insert() //functions of OperationDatabase
        {
            Console.WriteLine("Oracle Insert called");
        }

        public void Update() //functions of OperationDatabase
        {
            Console.WriteLine("Oracle Update called");
        }

        public void Delete() //functions of OperationDatabase
        {
            Console.WriteLine("Oracle Delete called");
        }

    } //step-3 Oracle implements OperationDatabase Derived class
}
 #endregion
```




```
using System;

namespace DemoOOP
{
   class Program
   {
      static void Main(string[] args)
      {
   Console.WriteLine("Tell us what you want: 1: SQL Server, 2: Oracle, 3: MySQL");
         int choice = Convert.ToInt32(Console.ReadLine());

         DatabaseFactory databaseFactory = new DatabaseFactory();
         IDatabase somObject = databaseFactory.GetSomeDatabase(choice);

  Console.WriteLine("Tell us what you want: 1: Insert, 2: Update, 3: Delete");
         int opchoice = Convert.ToInt32(Console.ReadLine());

         switch (opchoice)
         {
            case 1:
               somObject.Insert();
               break;
            case 2:
               somObject.Update();
               break;
            case 3:
               somObject.Delete();
               break;
            default:
               Console.WriteLine("Invalid choice of operation");
```

```
                break;
            }
            Console.ReadLine();
        }
    }

    public class DatabaseFactory
    {
        public IDatabase GetSomeDatabase(int choice)
        {
            IDatabase somobj = null;

            if (choice == 1)
            {
                somobj = new SQLServer();
            }
            else if (choice == 2)
            {
                somobj = new Oracle();
            }
            else
            {
                somobj = new MYSQL();
            }
            return somobj;
        }
    }

    public interface IDatabase
    {
        void Insert();
        void Update();
        void Delete();
    }
    public class MYSQL : IDatabase
    {
        public void Delete()
        {
            Console.WriteLine("MYSQL DELETE");
        }

        public void Insert()
        {
            Console.WriteLine("MYSQL INSERT");
        }
```

```
        public void Update()
        {
            Console.WriteLine("MYSQL UPDATE");
        }
    }

    public class SQLServer : IDatabase
    {
        public void OpenConnection() { }
        public void Insert()
        {
            Console.WriteLine("SQL Insert called");
        }

        public void Update()
        {
        Console.WriteLine("SQL Update called");
        }

        public void Delete()
{
    Console.WriteLine("SQL Delete called");
}
    }

    public class Oracle : IDatabase
{
    public void Insert()
    {
        Console.WriteLine("Oracle Insert called");
    }

    public void Update()
    {
        Console.WriteLine("Oracle Update called");
    }

    public void Delete()
    {
        Console.WriteLine("Oracle Delete called");
    }

}
}
```

Object Oriented Programming (OOP) Concepts

The object oriented languages follow four paradigms:

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

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

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

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

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

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

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

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

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

    }
```

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

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

We can declare as many overloads as we want.

Properties:

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

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

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

            }
}
```

The properties can be used in following manner:

```
            Math_operations obj = new Math_operations();

            obj.Variable = 3; // setter is called

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

Auto properties:

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

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

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

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

Destructors:

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

Events:

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

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

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

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

The basic syntax of inheritance is as follows:

```
class base_class
{
}

class derived_class : base_class
{
}
```

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

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

```
sealed class sealed_class
 {
 }
```

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

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

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

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

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

```
    }

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

                return _Id;
            }
            set
            {
                _Id = value;

            }

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

        }
}
```

In above example, you can see a keyword: *override*. This was used to give a meaning or some implementation to the abstract method. Similarly, for normal classes, the developers can give the provision of overriding any normal implemented method in base class to be overridden in derived class, by specifying the keyword *virtual*.

In below example, in Emp class, there are 2 methods: *CheckEmp* and *DelEmp* which both will have some logic. There is one more class ITEmployee which is more specific to IT field and their business logic may or may not be different from base class Emp. So if the developer gives the provision to override the 2 methods, by specifying virtual keyword, then the two methods can be overridden in derived class.

```
class Emp:Person
    {

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

        protected virtual void checkEmp()
```

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

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

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

3. Static Polymorphism

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

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

        }

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

        }
    }
```

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

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

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

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


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

Interfaces:

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

For e.g.:

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

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

```
            throw new NotImplementedException();
        }

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

Multiple interfaces can be implemented in a single class.

```
using System;
namespace DemoOOP2
{
    class Program
    {
        //Code Writer : Nilesh
        static void Main(string[] args)
        {
            Console.WriteLine("1: PDF, 2: DOCX, 3: TXT, 4: XML");
            int choice = Convert.ToInt32(Console.ReadLine());

            ReportFactory reportFactory = new ReportFactory();
            Report report = reportFactory.GetReport(choice);
            report.GenerateReport();

            Console.ReadLine();
        }
    }

    //Architect: Sachin
    public abstract class Report
    {
        protected abstract void Parse();
        protected abstract void Validate();
        protected abstract void Save();

        public virtual void GenerateReport()
        {
            Parse();
            Validate();
            Save();
        }
    }
    public class ReportFactory
```

```
{
    public Report GetReport(int choice)
    {
        if (choice == 1)
        {
            return new PDF();
        }
        else if (choice == 2)
        {
            return new DOCX();
        }
        else if(choice ==3){
            return new TXT();
        }
        else
        {
            return new XML();
        }
    }
}
//Class Writer : Mahesh
public class PDF: Report
{
    protected override void Parse()
    {
        Console.WriteLine("PDF Data Parsed Successfully"); //imagine this logic is 1000 lines logic
    }

    protected override void Validate()
    {
Console.WriteLine("PDF Data Validated Successfully"); //imagine this logic is 1000 lines logic
    }

    protected override void Save()
    {
Console.WriteLine("PDF Saved Successfully"); //imagine this logic is 10 lines logic
    }
}
//Class Writer : Amit
public class DOCX : Report
{
    protected override void Parse()
    {
Console.WriteLine("DOCX Data Parsed Successfully"); //imagine this logic is 1000 lines logic
    }

    protected override void Validate()
    {
Console.WriteLine("DOCX Data Validated Successfully"); //imagine this logic is 1000 lines logic
    }
```

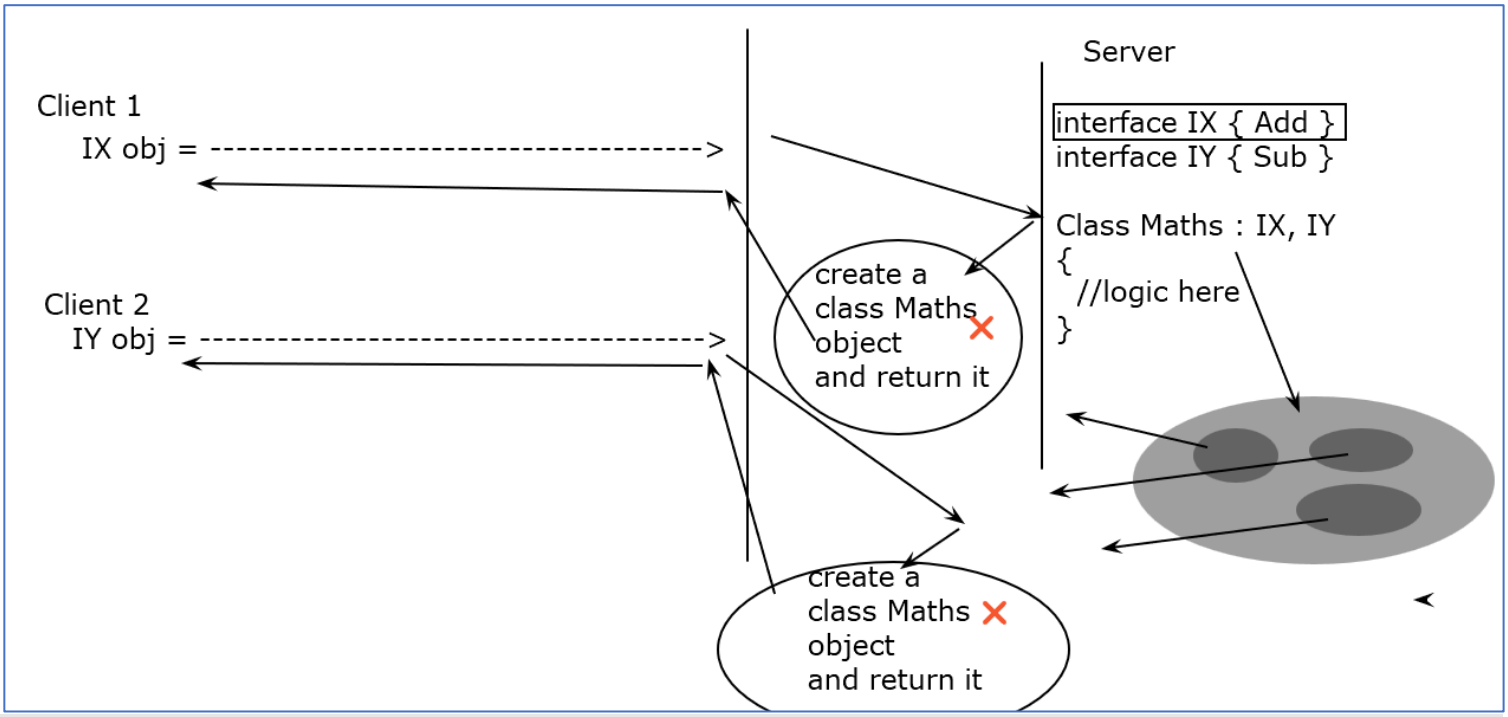
```
        protected override void Save()
        {
Console.WriteLine("DOCX Saved Successfully"); //imagine this logic is 10 lines logic
        }
    }
    public abstract class SpecialReport : Report
    {
        protected abstract void ReValidate();
        public override void GenerateReport()
        {
            Parse();
            Validate();
            ReValidate();
            Save();
        }
    }
    public class TXT : SpecialReport
   {
        protected override void ReValidate()
        {
            Console.WriteLine("TXT Revalidated");
        }
        protected override void Parse()
        {
            Console.WriteLine("TXT Parsed");
        }
        protected override void Save()
        {
            Console.WriteLine("TXT Saved");
        }
        protected override void Validate()
        {
            Console.WriteLine("TXT Validated");
        }
    }
    public class XML : SpecialReport
    {
        protected override void ReValidate()
        {
            Console.WriteLine("XML Revalidated");
        }
        protected override void Parse()
        {
            Console.WriteLine("XML Parsed");
        }
        protected override void Save()
        {
            Console.WriteLine("XML Saved");
        }
```

```
        protected override void Validate()
        {
            Console.WriteLine("XML Validated");
        }
    }

    //Class Writer: Sarang
    //public class TXT : Report
    //{
    //    protected void ReValidate()
    //    {
    //        Console.WriteLine("TXT Revalidated");
    //    }

    //    protected override void Parse()
    //    {
    //        Console.WriteLine("TXT Parsed");
    //    }

    //    protected override void Save()
    //    {
    //        Console.WriteLine("TXT Saved");
    //    }

    //    protected override void Validate()
    //    {
    //        Console.WriteLine("TXT Validated");
    //    }

    //    public override void GenerateReport()
    //    {
    //        Parse();
    //        Validate();
    //        ReValidate();
    //        Save();
    //    }
    //}
}
```

Day04_Agenda: - Session 4: Static Members of a Class • Fields • Methods • Properties • Constructors Static Classes Static local functions Inheritance • Access Specifiers • Constructors in a hierarchy • Overloading in derived class • Hiding, using new • override • sealed methods • Abstract Classes • Abstract Methods • Sealed Classes

1. OOP concept 2. Logger-static 3. Sealed(final)

4. Contentment 5. Getter/setter 6. Constructor → ctor tab tab




```
#region Understanding Singletone Object Code
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DemoOOP3
{
    class Program
    {
        static void Main(string[] args)
        {
            while (true)
            {
                Console.WriteLine("Tell us what you want: 1: SQL Server, 2: Oracle, 3: MySQL");
                int choice = Convert.ToInt32(Console.ReadLine()); //to get required database from
SQLserver mysql and Oracle

                DatabaseFactory dF = new DatabaseFactory();
                Database somObject = dF.GetSomeDatabase(choice); //database somObject to access the
database method like insert update delete

                Console.WriteLine("Tell us what you want: 1: Insert, 2: Update, 3: Delete");
                int opchoice = Convert.ToInt32(Console.ReadLine());

                switch (opchoice)
                {
                    case 1:
                        somObject.Insert();
                        break;
```

```
                case 2:
                    somObject.Update();
                    break;
                case 3:
                    somObject.Delete();
                    break;
                default:
                    Console.WriteLine("Invalid choice of operation");
                    break;
            }
            Console.WriteLine("Do you want to continue? Y / n");
            string ynchoice = Console.ReadLine();
            if (ynchoice == "n")
            {
                break;
            }
        }
        Console.ReadLine();
    }
}
public class DatabaseFactory
{
    public Database GetSomeDatabase(int choice)
    {
        Database somobj = null;

        if (choice == 1)
        {
            somobj = new SQLServer();
        }
        else if (choice == 2)
        {
            somobj = new Oracle();
        }
        else
        {
            somobj = new MYSQL();
        }
        return somobj;
    }
}
//Sachin
public abstract class Database
{
    protected Logger logger = null;
    public Database() //constructor of database
    {
        logger = Logger.GetLogger();
    }
    public abstract void Insert();
    public abstract void Update();
    public abstract void Delete();
}
//Amit
public class MYSQL : Database
{
    public override void Delete()
    {
        Console.WriteLine("MYSQL DELETE");//Imagine this is 100s of lines of code here..
        logger.Log("Delete happened in MYSQL Successfully!");
    }
    public override void Insert()
    {
        Console.WriteLine("MYSQL INSERT");
        logger.Log("Insert happened in MYSQL Successfully!");
    }
```

```
    public override void Update()
    {
        Console.WriteLine("MYSQL UPDATE");
        logger.Log("Update happened in MYSQL Successfully!");
    }
}
//Nilesh
public class SQLServer : Database
{
    public override void Insert()
    {
        Console.WriteLine("SQL Insert called");
        logger.Log("Insert happened in SQL Server Successfully!");
    }
    public override void Update()
    {
        Console.WriteLine("SQL Update called");
        logger.Log("Update happened in SQL Server Successfully!");
    }
    public override void Delete()
    {
        Console.WriteLine("SQL Delete called");
        logger.Log("Delete happened in SQL Server Successfully!");
    }
}

//Mahesh
public class Oracle : Database
{
    public override void Insert()
    {
        Console.WriteLine("Oracle Insert called");
        logger.Log("Insert happened in Oracle Server Successfully!");
    }

    public override void Update()
    {
        Console.WriteLine("Oracle Update called");
        logger.Log("Delete happened in Oracle Server Successfully!");
    }

    public override void Delete()
    {
        Console.WriteLine("Oracle Delete called");
        logger.Log("Delete happened in Oracle Server Successfully!");
    }

}

public class Logger
{
    private static Logger logger = new Logger();
    //private static Logger logger2 = new Logger();
    //since constructor is private ..we can call it only here!
    //You can decide … how many objects are supposed to be created ….
    //like you want to offer single object to people
    //or pool of objects to people..
    private Logger()
    {
        Console.WriteLine("Logger Object Created…");
        //Here we can have Logger Related Initializations…
    }

    public static Logger GetLogger()
    {
        //Here can be some logic which based on some condition return
```

```
            //specific object from pool of Logger objects
            return logger;
        }

        public void Log(string message)
        {
            //You can log the message details into - DB or File or Email or anywhere client has
asked for..
            //as of now for simulation we will put it on console..
            Console.WriteLine("---- Logged: " + message + " at " + DateTime.Now.ToString() + " ---
");
        }
    }
}
#endregion
```

----------%%%%----------

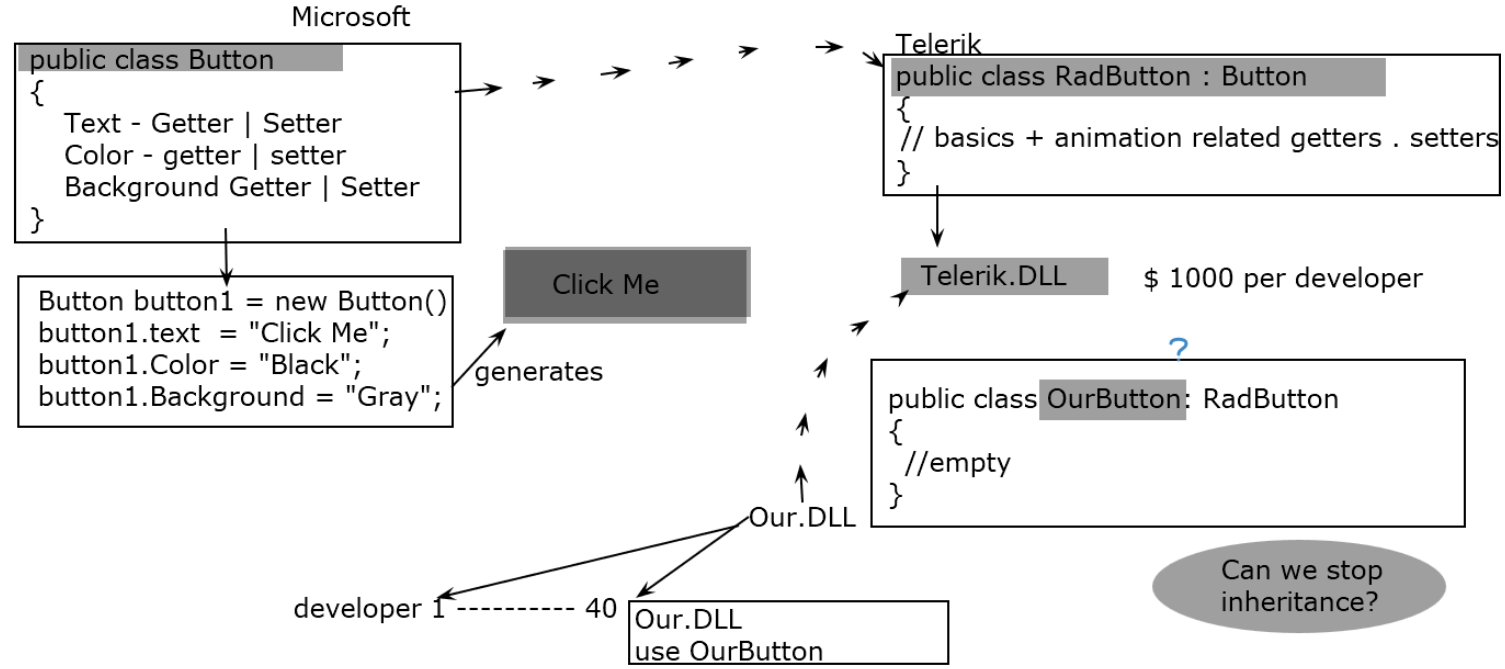
```
#region More Furnished Code
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DemoOOP3
{
    class Program
    {
        static void Main(string[] args)
        {
            while (true)
            {
                Console.WriteLine("Tell us what you want: 1: SQL Server, 2: Oracle, 3: MySQL");
                int choice = Convert.ToInt32(Console.ReadLine());

                DatabaseFactory databaseFactory = new DatabaseFactory();
                Database somObject = databaseFactory.GetSomeDatabase(choice);

                Console.WriteLine("Tell us what you want: 1: Insert, 2: Update, 3: Delete");
                int opchoice = Convert.ToInt32(Console.ReadLine());

                switch (opchoice)
                {
                    case 1:
                        somObject.Insert();
                        break;
                    case 2:
                        somObject.Update();
                        break;
                    case 3:
                        somObject.Delete();
                        break;
                    default:
                        Console.WriteLine("Invalid choice of operation");
                        break;
                }
                Console.WriteLine("Do you want to continue? y / n");
                string ynchoice = Console.ReadLine();
                if (ynchoice == "n")
                {
                    break;
                }
            }
            Console.ReadLine();
        }
    }
```

```
public class DatabaseFactory
{
    public Database GetSomeDatabase(int choice)
    {
        Database somobj = null;

        if (choice == 1)
        {
            somobj = new SQLServer();
        }
        else if (choice == 2)
        {
            somobj = new Oracle();
        }
        else
        {
            somobj = new MYSQL();
        }
        return somobj;
    }
}
//Sachin
public abstract class Database
{
    protected Logger logger = null;
    public Database()
    {
        logger = Logger.GetLogger();
    }
    protected abstract void DoInsert();
    protected abstract void DoUpdate();
    protected abstract void DoDelete();

    protected abstract string GetDatabaseName();

    public void Insert()
    {
        DoInsert();
        //string dbName = GetDatabaseName();
        //logger.Log("Insert happened in " + dbName + " Successfully!");

        logger.Log("Insert happened in  " + GetDatabaseName() + "  Successfully!");
    }

    public void Update()
    {
        DoUpdate();
        logger.Log("Update happened in  " + GetDatabaseName() + "  Successfully!");
    }

    public void Delete()
    {
        DoDelete();
        logger.Log("Delete happened in  " + GetDatabaseName() + "  Successfully!");
    }
}
//Amit
public class MYSQL : Database
{
    protected override string GetDatabaseName()
    {
        return "MYSQL";
    }
    protected override void DoDelete()
    {
```

```
            Console.WriteLine("MYSQL DELETE");//Imagine this is 100s of lines of code here..
        }
        protected override void DoInsert()
        {
            Console.WriteLine("MYSQL INSERT");
        }
        protected override void DoUpdate()
        {
            Console.WriteLine("MYSQL UPDATE");
        }
    }

    //Nilesh
    public class SQLServer : Database
    {
        protected override string GetDatabaseName()
        {
            return "SQL Server";
        }
        protected override void DoInsert()
        {
            Console.WriteLine("SQL Insert called");
        }
        protected override void DoUpdate()
        {
            Console.WriteLine("SQL Update called");
        }
        protected override void DoDelete()
        {
            Console.WriteLine("SQL Delete called");
        }
    }

    //Mahesh
    public class Oracle : Database
    {
        protected override string GetDatabaseName()
        {
            return "Oracle";
        }
        protected override void DoInsert()
        {
            Console.WriteLine("Oracle Insert called");
        }

        protected override void DoUpdate()
        {
            Console.WriteLine("Oracle Update called");
        }

        protected override void DoDelete()
        {
            Console.WriteLine("Oracle Delete called");
        }
    }
    public class Logger
    {
        private static Logger logger = new Logger();
        //private static Logger logger2 = new Logger();
        //since constructor is private ..we can call it only here!
        //You can decide ... how many objects are supposed to be created ....
        //like you want to offer single object to people
        //or pool of objects to people..
        private Logger()
        {
            Console.WriteLine("Logger Object Created...");
```

```
            //Here we can have Logger Related Initializations...
        }

        public static Logger GetLogger()
        {
            //Here can be some logic which based on some condition return
            //specific object from pool of Logger objects
            return logger;
        }

        public void Log(string message)
        {
    //You can log the message details into -- DB or File or EMail or anywhere client has asked for..
            //as of now for simulation we will put it on console..
          Console.WriteLine("---- Logged: " + message + " at " + DateTime.Now.ToString() + " ---");
        }
    }
} #endregion
```

----------%%%%----------



#Sealed (Final keyword in Java)

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

namespace DemoOOP4
{

    //Demo for Sealed
    class Program
    {
        static void Main(string[] args)
        {

        }
    }

    public class Button
    {
```

```
    }
    //Telerik button
    //To save the RADButton from getting inherited as a concern from
    //product based company ... we can make RADButton sealed
    //sealed in c# == final in Java
    public sealed class RADButton : Button
    {

    }

    //Making of our own button to save the cost
    //When the RADButton is sealed .. you can not inherit!!
    //public class OurButton : RADButton
    //{

    //}
}
-------------------------------------------------%%%%-------------------------------------
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DemoProperties
{

    class Program
    {
        static void Main(string[] args)
        {
            Employee emp = new Employee();
            //emp.No = -99;

            //Console.WriteLine(emp.No);
            Console.ReadLine();
        }
    }
    public class Employee
    {
        private int _No;
        private string _Name;
        private string _Address;

        public string Address
        {
            get { return _Address; }
            set { _Address = value; }
        }

        public string Name
        {
            get { return _Name; }
            set { _Name = value; }
        }

        public int No
        {
            get { return _No; }
            set { _No = value; }
        }
    }


    #region Getter / Setter in C# Way as Property
    //class Program
```

```
    //{
    //    static void Main(string[] args)
    //    {
    //        //Employee emp = new Employee();
    //        //emp.Set_No(80);
    //        //Console.WriteLine(emp.Get_No());

    //        Employee emp = new Employee();
    //        emp.No = -99;

    //        Console.WriteLine(emp.No);
    //        Console.ReadLine();
    //    }
    //}
    //public class Employee
    //{
    //    private int _No;

    //    public int No //here No is called as property
    //    {
    //        get
    //        {
    //            //We may want to return a different / manipulated value to end user from here..
    //            return _No;
    //        }
    //        set
    //        {
    //            //Validate the int value and if valid then only you set the value to _No
    //            if (value > 0)
    //            {
    //                _No = value;
    //            }
    //            else
    //            {
    //                _No = 0;
    //            }

    //        }
    //    }

    //}
    #endregion
}

#region Getter Setter in CPP and Java Way
//    public class Employee
//    {
//        private int _No;

//        public void Set_No(int no)
//        {
//            //Validate the int no and if valid then only you set the value to _No
//            _No = no;
//        }

//        public int Get_No()
//        {
//            //We may want to return a different / manipulated value to end user from here..
//            return _No;
//        }

//    }
//}
#endregion
```

Class — Constructor, Functions, Member

Inheritance

Interface

Abstract Class

Overriding : Virtual & Override

Abstract & Override

Logger - Singletone
- private constructor
- static method / member

Sealed (Final in Java)

Parameterised Contructors

Getter, Setters aka properties

Event

Delegate

Sarang is a software developer in an IT firm Microsoft.
Company has asked Sarang to develop a Notepad like Editor.
Just like any Editor would have Cut, Copy, Paste like formatting features ... This editor will also have the same.

The unique selling point (USP) of the editor is - this editor can do spellcheck for given word!!

Notepad is not offered without a spell check support as its USP of the editor.
So, when this edtor is given to any user; use can do by default english word check.

Keep in mind, english word check is a huge functionality in itself.. and can be treated as a seperate entity itself!

Now, when company would sell this editor, customers would demand other languages support for spell check.

Sarang wants to make classes / code in such a manner that -- English support is given free
Other languages support like Hindi, French, German can be developed separately by company and can be made available on demand against some payments

At the same Sarang want to ensure, that the spellcheck functionality is somone wants to develop & use for some languages -- they sould be free to do so without coming back Sarang / Microsoft

If you are sarang, how will you develop and structure the classes?

** Create a console app.