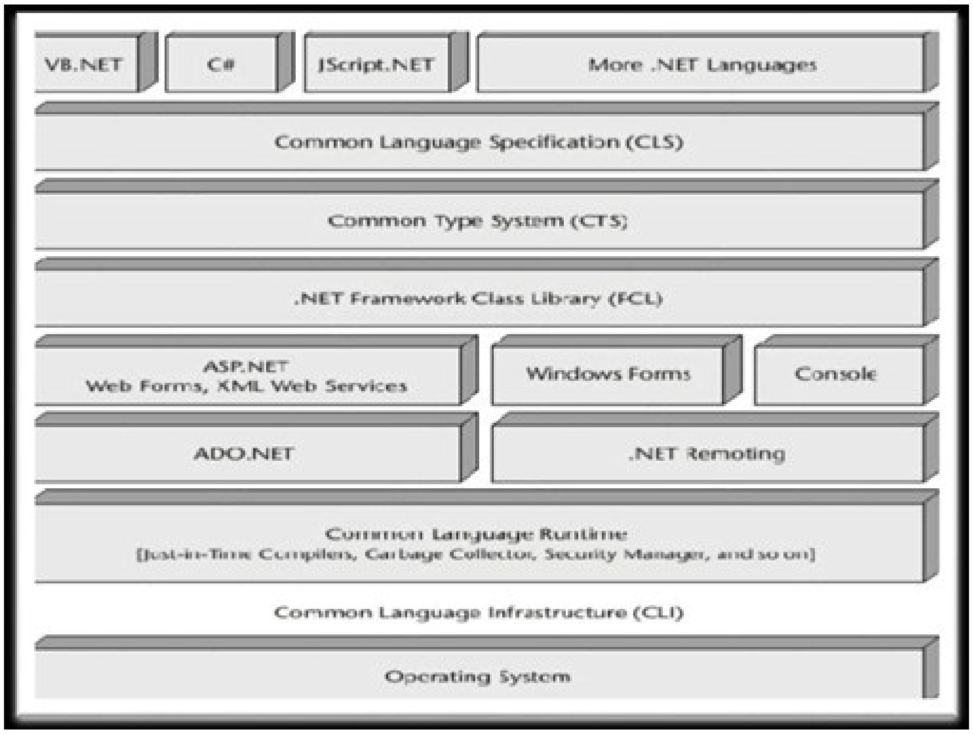
Microsoft .Net Framework

The Microsoft .Net Framework is a platform that provides tools and technologies needed to build Networked Applications as well as Distributed Web services and Web applications and support cross language development (sharing programs of one .Net compatible language with program written in other .Net compatible language).

The .NET framework consists of:

- The .NET programming languages (C#, VB.NET and others);
- An environment for the execution of managed code (CLR), which executes C# programs in a controlled manner;
- A set of development tools, such as the csc compiler, which turns C# programs into intermediate code (called MSIL) that the CLR can understand;
- A set of standard libraries, like ADO.NET, which allow access to databases (such as MS SQL Server or MySQL) and WCF which connects applications through standard communication frameworks and protocols like HTTP, JSON, SOAP and TCP sockets.

The following is a diagram depicting .Net Framework:


VB.NET
C#
JScript.NET
More .NET Languages
Common Language Specification (CLS)
Common Type System (CTS)
.NET Framework Class Library (FCL)
ASP.NET Web Forms, XML Web Services
Windows Forms
Console
ADO.NET
.NET Remoting
Common Language Runtime [Just-in-Time Compilers, Garbage Collector, Security Manager, and so on]
Common Language Infrastructure (CLI)
Operating System

The Framework Class Library (FCL)

The .Net Framework provides a huge Framework (or Base) Class Library (FCL) for common, usual tasks. FCL contains thousands of classes to provide access to Windows API and common functions like String Manipulation, Common Data Structures, IO, Streams, Threads, Security, Network Programming, Windows Programming, Web Programming, Data Access, etc. You can use the classes in FCL in your program just as you would use any other class. You can even apply inheritance and polymorphism to these classes.

CLR (Common Language Runtime)

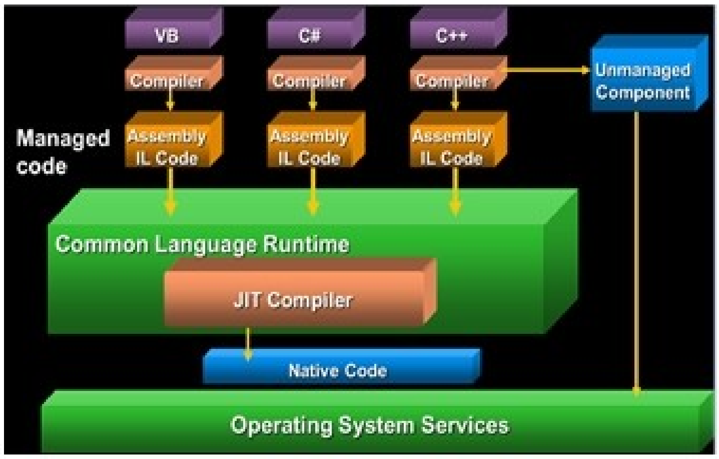
The Common Language Runtime (CLR) is the environment where all programs in .NET are run. It provides various services, like memory management and thread management. Programs that run in the CLR need not manage memory, as it is completely taken care of by the CLR, for example, when a program needs a block of memory, CLR provides the block and releases the block when program is done with the block. All programs targeted to .NET are converted to MSIL. Code running under the control of the CLR is often termed managed code.

Managed code is when all of the supported languages share a unified set of class libraries and can be encoded into an Intermediate Language (IL). A runtime-aware compiler compiles the IL into native executable code

within a managed execution environment that ensures type safety, array bound and index checking, exception handling, and garbage collection.
Generally, CLR address language interoperability, common classes for all languages, common types for all languages, assemblies, application domains etc

MSIL or IL

When we compile .Net Programs using any .Net compliant language (such as C#, VB.Net or C++.Net) the source code does not get converted into the executable binary code, but to an intermediate code known as MSIL which is interpreted by the Common Language Runtime. MSIL is operating system and hardware independent code. Upon program execution, this MSIL (intermediate code) is converted to binary executable code (native code). The following is the CLR Execution Model



Just In Time Compilers (JITers)

When our IL compiled code needs to be executed, the CLR invokes the JIT compiler, which compile the IL code to native executable code (.exe or .dll) that is designed for the specific machine and OS. JITers in many ways are different from traditional compilers as they compile the IL to native code only when desired; e.g., when a function is called, the IL of the function's body is converted to native code just in time. So, the part of code that is not used by that particular run is never converted to native code. If some IL code is converted to native code, then the next time it's needed, the CLR reuses the same (already compiled) copy without re-compiling.
When the code is executed, the platform-specific VES (Virtual Execution System) will compile the IL (CIL) to the machine language according to the specific hardware and operating system. .NET Framework comes with Native Image Generator (NGEN), which performs ahead-of-time compilation.

Common Language Infrastructure (CLI)

The Common Language Infrastructure (CLI) is an open specification developed by Microsoft and standardized by ISO and ECMA (European Computer Manufacturers Association) that describes executable code and a runtime environment that allow multiple high-level languages to be used on different computer platforms without being rewritten for specific architectures. The .NET Framework and the free and open source Mono and Portable.NET are implementations of the CLI.

The Common Type System (CTS)

.Net also defines a Common Type System (CTS). Like CLS, CTS is also a set of standards. CTS defines the basic data types that IL understands. Each .Net compliant language should map its data types to these standard data types. This makes it possible for the two languages to communicate with each other by passing/receiving parameters to and from each other. For example, CTS defines a type, Int32, an integral data type of 32 bits (4 bytes) which is mapped by C# through int and VB.Net through its Integer data type.

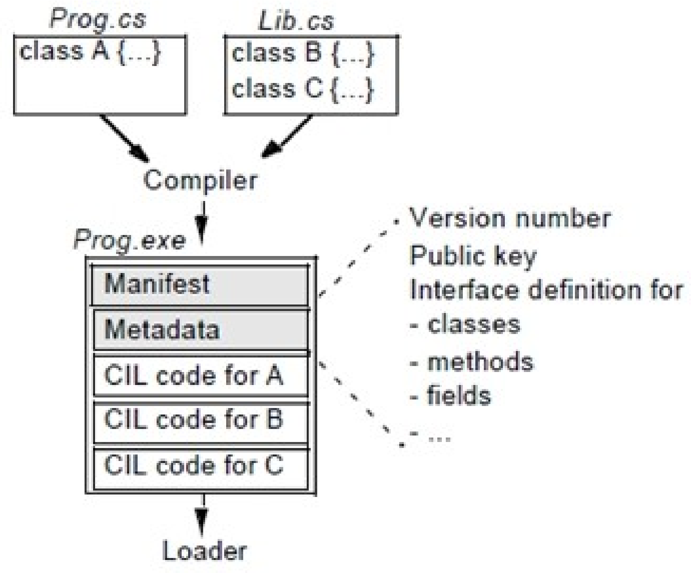
Common Language Specification

The Common Language Specification (CLS) works with the CTS to ensure language interoperability.

It is generally a set of minimum standards that all compilers targeting .NET must support.
E.g. IL is case-sensitive. Developers who work with case sensitive languages regularly take advantage of the flexibility this case sensitivity gives them when selecting variable names. Visual Basic .NET, however, is not case sensitive. The CLS works around this by indicating that CLS-compliant code should not expose any two names that differ only in their case. Therefore, Visual Basic .NET code can work with CLS-compliant code.

Assembly

An assembly is the logical unit that contains compiled code targeted at the .NET Framework. It consists of one or more files (dlls, exe's, html files etc.). Each assembly also contains metadata in addition to code. The metadata is used to provide version information along with a complete description of methods and types. An assembly also contains a manifest. The manifest includes identification information, public types and a list of other used assemblies. The manifest is part of the assembly, thus making the assembly self-describing. As all information is in the assembly itself, it is independent of registry.
In some cases an assembly is even made up of multiple files as shown below



Note that the same assembly structure is used for both executable code and library code.

Types of Assemblies

Assemblies come in two types: shared and private assemblies.

i). A private assembly is normally used by a single application, and is stored in the application's directory, or a sub-directory beneath. No registration is needed, and no installation program is required. When the component is removed, no registry cleanup is needed, and no uninstall program is required

ii). A shared assembly is intended to be used by multiple applications, and is normally stored in the global assembly cache (GAC), which is a central repository for assemblies. The GAC contains shared assemblies that are globally accessible to all .NET applications on the machine. (A shared assembly can also be stored outside the GAC, in which case each application must be pointed to its location via a codebase entry in the application's configuration file.)

Garbage Collection (GC)

CLR also contains the Garbage Collector (GC) that checks for un-referenced, dynamically allocated memory space. If it finds some data that is no longer referenced by any variable/reference, it re-claims it and returns it to the OS. The presence of a standard Garbage Collector frees the programmer from keeping track of dangling data.

APPLICATION OF .NET

The .NET Framework can be used to create many types of application

i). Console applications
ii). Windows applications

iii). Web application

i). Console applications

Console applications run from the command prompt. All written output from the application is shown in the command prompt window. The .NET Framework uses C# technology to create console applications.

ii). Windows applications

Windows applications are known in the .NET Framework as Windows Forms applications. These look like your normal applications that you run on the windows platform, they include buttons, text boxes, drop down list, graphics etc. The .NET Framework uses VB (visual basic technology) to create window applications.

iii).Web applications

Web applications appear on a web page, with the output appearing on the same page. The .NET Framework uses ASP technology to create web applications.

Introduction to C#

C# is a multi-paradigm programming language encompassing strong typing, imperative, declarative, functional, generic, object-oriented (class-based), and component-oriented programming disciplines. C# is one of the programming languages designed for the Common Language Infrastructure.
C# enables you to develop different types of applications such as Windows Based applications, Console Based applications, and Web Based applications. C# contains various features that make it similar to other programming languages such as C,C++, and java .However, there are some additional features in C# that make it different from other programming languages such as C++ and java.

C# Language Features

C# was developed as a language that would combine the best features of previously existing Web and Windows programming languages.

i). Modern: - C# has been based according to the current trend and is very powerful and simple for building interoperable, scalable, robust applications. C# is the best language for developing web application and components for the Microsoft .NET platform.

ii). Object Oriented: - C# is an object-oriented language. It supports all the basic object oriented language features such as: encapsulation, polymorphism, and inheritance.

iii). Simple and Flexible: - C# is as simple to use as Visual Basic, in that everything in C# represented as an object. All data type and components in C# are objects. Pointers are missing in C# and there is no usage of "::" or "->" operators

iv). Type safety: - C# is a type safe language. All variables and classes (including primitive type, such as integer, Boolean, and float) in C# are types derived from the object type. Arrays are zero base indexed and are bound checked. Overflow of types can also be checked.

v). Scalable: - .NET has introduced assemblies, which are self- describing by means of their manifest. Manifest establishes the assembly identity, version, culture etc. Assemblies need not to be register anywhere. To scale applications we delete the old files are deleted and updated with new ones.

vi). Language and Cross Platform Interoperability: - C#, as with all Microsoft .NET supported language,
shares a common .NET run-time library. The language compiler generates intermediate language (IL) code, which a .NET supported compiler can read with the help of the CLR. Therefore, you can use a C# assembly in VB.NET without any problem, and vice versa.

Application of C#

C# is modern and very power full language and is used in developing window, web and device application. The following are some of the applications of C#:

i). Developing console application

ii). Developing Windows Application

iii). Developing Web Application

iv). C# is used is ASP.NET so web application is very easy to develop using C# and Asp.NET.

v). C# is use to create web services.
vi). C# is use to create both web and window Component.
vii). Mobile Application is also develop in C#.
viii). C# is used for making games.

Namespaces in C#

A Namespace is simply a logical collection of related classes in C#. It is used to bundle related classes (like those related with database activity) in some named collection calling it a namespace (e.g., DataActivity). As C# does not allow two classes with the same name to be used in a program, the sole purpose of using namespaces is to prevent name conflicts. This may happen when you have a large number of classes, as is the case in the Framework Class Library (FCL). It is very much possible that our Connection Class in DataActivity conflicts with the Connection Class of InternetActivity. To avoid this, these classes are made part of their respective namespace. So the fully qualified name of these classes will be DataActivity.Connection and InternetActivity.Connection, hence resolving any ambiguity for the compiler.

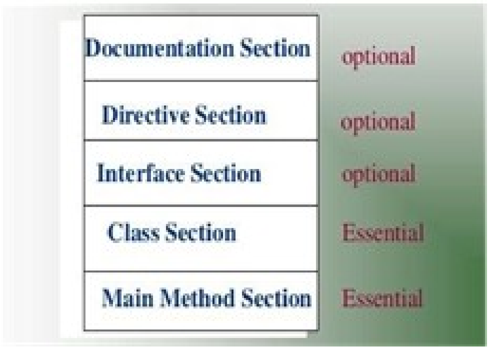
Standard Namespaces in .NET

In C#, you can use various namespaces provided by .NET framework to access classes contained in them. The various namespaces available in .Net framework that can be used in C# are as follow:-

- System: - It contains classes that allow you to perform basic operations such as mathematical operations and data conversion.
- System.IO:- It contains classes used for input and output operations for a file in C#.
- System.Net:- It contains classes that are used to define network protocols.
- System.Data: - It contains classes that are used to make ADO.NET data access architecture.
- System.Collections:- It contains classes that implement collection of objects such as list.
- System.Drawing:- It contains classes that are used to implement GUI functionalities.
- System.Web:- It contains classes that help implement HTTP protocol to access web pages.

Structure of C# program

A C# program contains multiple class definitions, which are the primary elements of the program. A C# program contains following sections:



- Documentation Section: - This section contains comments such as name of the program and date on which the programmer started creating the program. The documentation section must provide the information about the classes defined in the program.
- Directive Section: - This section includes all the namespaces that contain classes required for the execution of the program.
- Interface Section: - The Interface section contains the declarations of interfaces that are used in the program. Interfaces are used to implement the section of multiple inheritance.
- Main Method Section: - The Main method is an essential section of a C# program. It helps to creates of various classes of the program and establishes communication between them.

Generic Structure of C# Program

```
// A skeleton of a C# program
using System;
namespace YourNamespace
```

```
{
        class YourClass { }
        struct YourStruct { }
        interface    IYourInterface    {    }
        delegate    int    YourDelegate();
        enum YourEnum { }
        namespace
        YourNestedNamespace
        {
                struct YourStruct { }
        }
        class  YourMainClass  {  static  void  Main(string[]  args)  {  //Your
        program starts here... } }
}
```

Example 1:
Write a C# console application to display the word "Welcome to C# programming"

```
Using System;
class  HelloCSharp
{
        public   static   void   Main(string   []   args)
        {
                Console.WriteLine("Welcome to C# programming");
        }
}
```

Or

```
Using System;
class  HelloCSharp
{
        static   void   Main()
        {
                Console.WriteLine("Welcome to C# programming");
        }
}
```

Explanations

The using Keyword

The statement

```
using System;
```

is used in accessing members of the System namespace. So in the example, the class Console, which contains the method WriteLine belongs to the System namespace. Notice that it can only allow you to access the classes in the referenced namespace and not in its internal/child namespaces. Hence we might need to write

```
using System.Collections;
```

in order to access the classes defined in Collection namespace which is a sub/internal namespace of System namespace.
We could avoid the "using" statement by writing the complete path of the method:

```
System.Console.WriteLine("Welcome to C# programming");
```

The class Keyword

All of our C# programs contain at least one class. The Main() method resides in one of these classes. Classes are a combination of data (fields) and functions (methods) that can be performed on this data in order to achieve the solution to our problem. Classes in C# are defined using the class keyword followed by the name of class.

The Main() Method

The Main() method is defined as follows:

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

This is the standard signature of the Main method in C#. The Main method is the entry point of our program.
The Main method is designated as static as it will be called by the Common Language Runtime (CLR) without making any object of our HelloCSharp Class (which is the definition of static methods, fields and properties).
The method is also declared void as it does not return anything. Main is the (standard) name of this method, while string [] args is the list of parameters that can be passed to main while executing the program from command line.
One interesting point here is that it is legitimate to have multiple Main() methods in C# program. But, you have to explicitly identify which Main method is the entry point at the run-time. Java Programmers, take note that Main starts with capital 'M' and the return type is void.

Printing on the Console

The following statement prints Welcome to C# programming on the Console screen:

```
Console.WriteLine("Welcome to C# programming")
```

The WriteLine(), a static method of the Console class defined in the System namespace is called. This method takes a string (enclosed in double quotes) as its parameter and prints it on the Console window.
C#, like other Object Oriented languages, uses the dot (.) operator to access the member variables (fields) and methods of a class. Also, braces () are used to identify methods in the code and string literals are enclosed in double quotation marks ("). Lastly, each statement in C# (like C, C++ and Java) ends with a semicolon (;), also called the statement terminator.

Adding Comments in Program

Comments play very important role in the maintenance of programs. They are used to simplify readability and understanding code. All programs should have information such as implementation, details of class, and others. C# use three type of comments

1. Single line comment
   ```
   // this is single line comments
   ```
2. Multiline comments
   ```
    /* this is
        Example Of
        multi line Comments
   */
   ```
3. Documentation comments

C# introduces another kind of comment called 'documentation comments'. C# can use these to generate the documentation for your classes and program and make use of XML elements.

```
 /// These are documentation comment

Or

/**
These are documentation comment
*/
```

```
E.g
///<summary>
/// File: HelloWorld.cs
/// prints "Hello, World!”
///</summary>
```

Program Freeform Style

C# is a freeform language. We need not indent any lines to make the program work properly. C# does not care where on the line we begin coding. Although several alternative styles are possible to write codes of C# program.

System. Console.WriteLine (“Hello!”);

// can be written as:

System.Console.writeLine

(“Hello!”);

Example 2:

Write a C# program that ask the user their name and then greets the user using his/her name

```
Using System;
class HelloCSharp
{
     static void Main(string[] args)
     {
          Console.Write("Please enter your name: ");
          string name = Console.ReadLine();
          Console.WriteLine("Hello {0}, Good Luck in C#", name);
     }
}
```

Explanations

The substitution parameter {0} is used to state where in the line the data in the variable 'name' should be written when the WriteLine() method is called.

```
Console.WriteLine ("Hello {0}, Good Luck in C#", name);
```

When the compiler finds a substitution parameter, {n}, it replaces it with the (n+1)th variable following the string. The string is delimited with double quotation marks and each parameter is separated by a comma. Hence, in our case when the compiler finds {0}, it replaces it with (0+1)th, i.e., the 1st variable ('name') following the string. Alternatively, it can also be written as:

```
Console.WriteLine ("Hello " + name + ", Good Luck in C#");
```

removing the substitution parameter. Here we concatenate the strings together to form a message. (The first approach is similar to C's printf() function while the second is similar to Java's System.out.println() method)