Unit 1

Introduction to .NET Framework

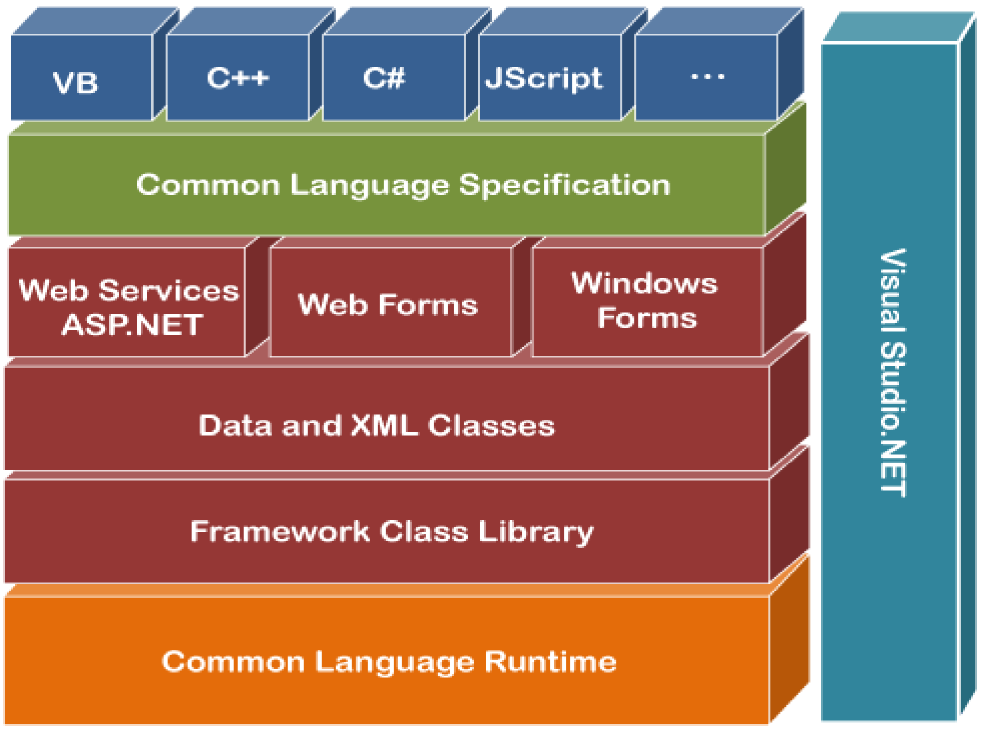
The .NET Framework is a software development platform that was introduced by Microsoft in the late 1990 under the NGWS. On 13 February 2002, Microsoft launched the first version of the .NET Framework, referred to as the .NET Framework 1.0.

In this section, we will understand the .NET Framework, characteristics, components, and its versions.

What is .NET Framework

It is a virtual machine that provide a common platform to run an application that was built using the different language such as C#, VB.NET, Visual Basic, etc. It is also used to create a form based, console-based, mobile and web-based application or services that are available in Microsoft environment. Furthermore, the .NET framework is a pure object oriented, that similar to the Java language. But it is not a platform independent as the Java. So, its application runs only to the windows platform.

The main objective of this framework is to develop an application that can run on the windows platform. The current version of the .Net framework is 4.8.



Components of .NET Framework

There are following components of .NET Framework:

1. CLR (Common Language Runtime)
2. CTS (Common Type System)
3. BCL (Base Class Library)
4. CLS (Common Language Specification)
5. FCL (Framework Class Library)
6. .NET Assemblies
7. XML Web Services
8. Window Services

CLR (common language runtime)

It is an important part of a .NET framework that works like a virtual component of the .NET Framework to executes the different languages program like c#, Visual Basic, etc. A CLR also helps to convert a source code into the byte code, and this byte code is known as CIL (Common Intermediate Language) or MSIL (Microsoft Intermediate Language). After converting into a byte code, a CLR uses a JIT compiler at run time that helps to convert a CIL or MSIL code into the machine or native code.

CTS (Common Type System)

It specifies a standard that represent what type of data and value can be defined and managed in computer memory at runtime. A CTS ensures that programming data defined in various languages should beinteract with each other to share information. For example, in C# we define data type as int, while in VB.NET we define integer as a data type.

BCL (Base Class Library)

The base class library has a rich collection of libraries features and functions that help to implement many programming languages in the .NET Framework, such as C #, F #, Visual C ++, and more. Furthermore, BCL divides into two parts:

1. **User defined class library**
    - **Assemblies -** It is the collection of small parts of deployment an application's part. It contains either the DLL (Dynamic Link Library) or exe (Executable) file.
        1. In LL, it uses code reusability, whereas in exe it contains only output file/ or application.
        2. DLL file can't be open, whereas exe file can be open.
        3. DLL file can't be run individually, whereas in exe, it can run individually.
        4. In DLL file, there is no main method, whereas exe file has main method.
2. **Predefined class library**
    - **Namespace -** It is the collection of predefined class and method that present in .Net. In other languages such as, C we used header files, in java we used package similarly we used "using system" in .NET, where using is a keyword and system is a namespace.

CLS (Common language Specification)

It is a subset of common type system (CTS) that defines a set of rules and regulations which should be followed by every language that comes under the .net framework. In other words, a CLS language should be cross-language integration or interoperability. For example, in C# and VB.NET language, the C# language terminate each statement with semicolon, whereas in VB.NET it is not end with semicolon, and when these statements execute in .NET Framework, it provides a common platform to interact and share information with each other.

Microsoft .NET Assemblies

A .NET assembly is the main building block of the .NET Framework. It is a small unit of code that contains a logical compiled code in the Common Language infrastructure (CLI), which is used for deployment, security and versioning. It defines in two parts (process) DLL and library (exe) assemblies. When the .NET program is compiled, it generates a metadata with Microsoft Intermediate Language, which is stored in a file called Assembly.

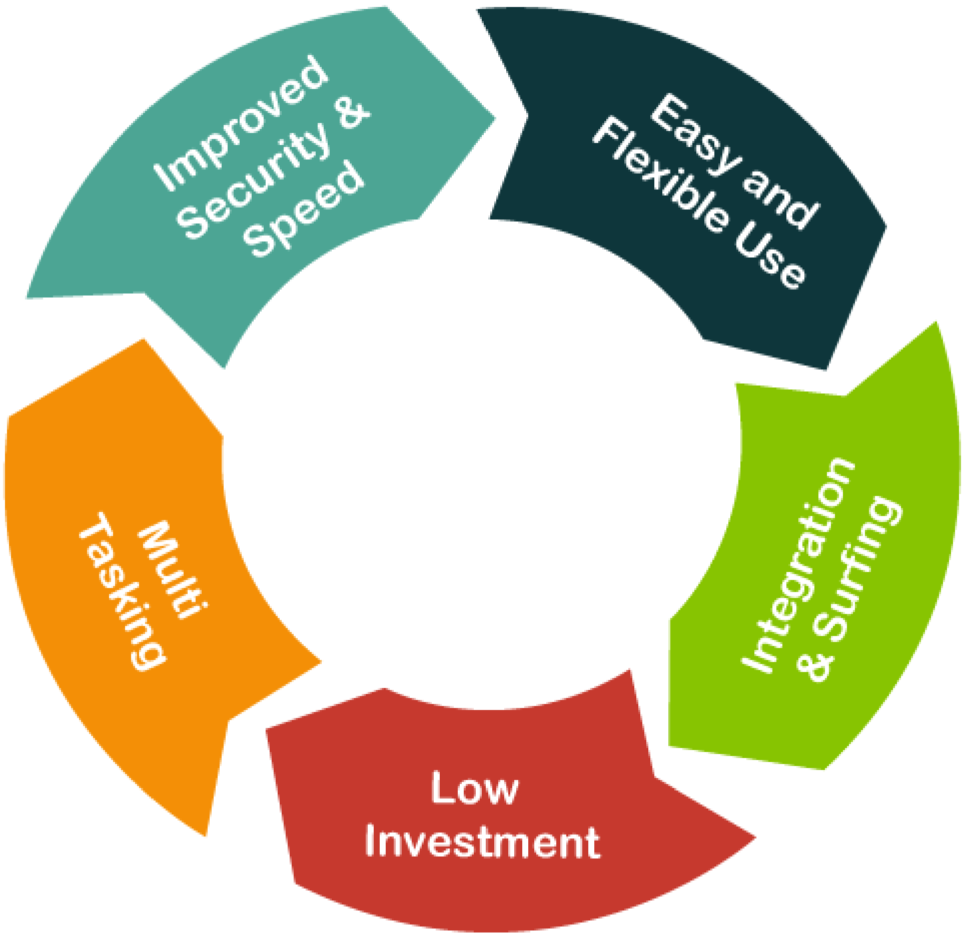
FCL (Framework Class Library)

It provides the various system functionality in the .NET Framework, that includes classes, interfaces and data types, etc. to create multiple functions and different types of application such as desktop, web, mobile application, etc. In other words, it can be defined as, it provides a base on which various applications, controls and components are built in .NET Framework.

Key Components of FCL

1. Object type
2. Implementation of data structure
3. Base data types
4. Garbage collection
5. Security and database connectivity
6. Creating common platform for window and web-based application

Characteristics of .NET Framework



1. CLR (Common Language Runtime)
2. Namespace - Predefined class and function
3. Metadata and Assemblies
4. Application domains
5. It helps to configure and deploy the .net application
6. It provides form and web-based services
7. NET and ASP.NET AJAX
8. LINQ
9. Security and Portability
10. Interoperability
11. It provides multiple environments for developing an application

.NET Framework Class Library

.NET Framework Class Library is the collection of classes, namespaces, interfaces and value types that are used for .NET applications.

It contains thousands of classes that supports the following functions.

- Base and user-defined data types
- Support for exceptions handling
- input/output and stream operations
- Communications with the underlying system
- Access to data
- Ability to create Windows-based GUI applications
- Ability to create web-client and server applications
- Support for creating web services

.NET Framework Class Library Namespaces

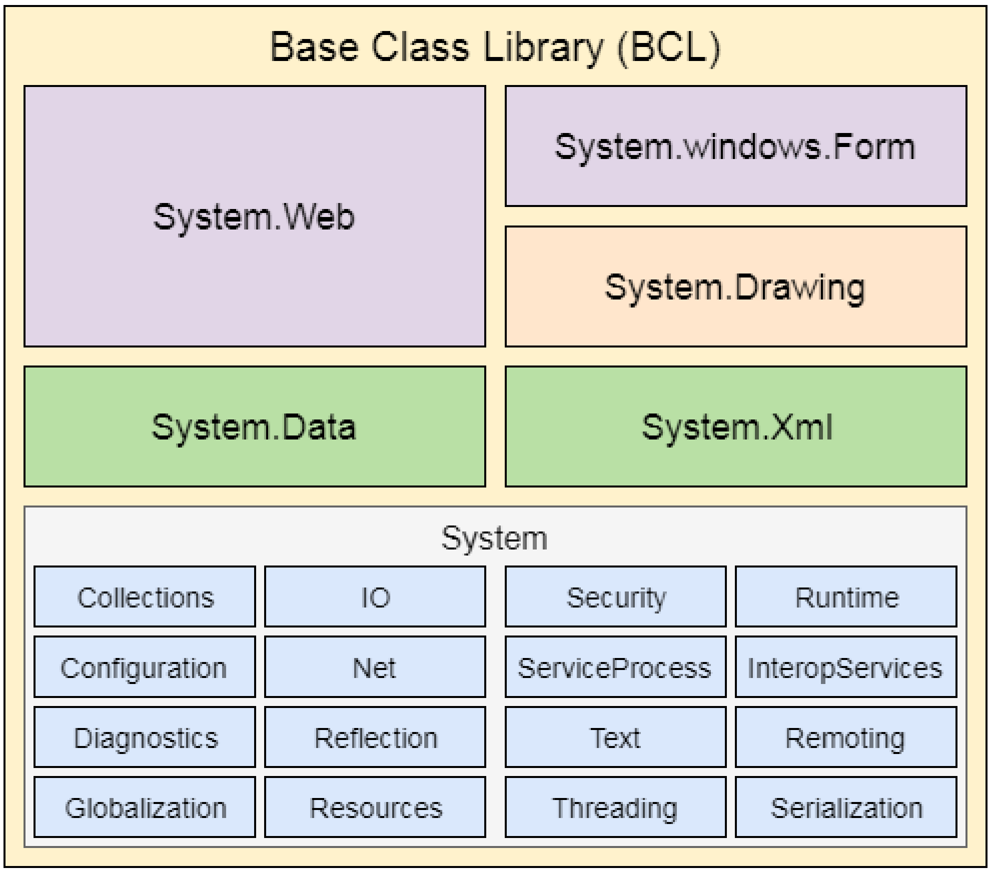
Following are the commonly used namespaces that contains useful classes and interfaces and defined in Framework Class Library.

Namespaces	Description
System	It includes all common datatypes, string values, arrays and methods for data conversion.
System.Data, System.Data.Common, System.Data.OleDb, System.Data.SqlClient, System.Data.SqlTypes	These are used to access a database, perform commands on a database and retrieve database.
System.IO, System.DirectoryServices, System.IO.IsolatedStorage	These are used to access, read and write files.
System.Diagnostics	It is used to debug and trace the execution of an application.
System.Net, System.Net.Sockets	These are used to communicate over the Internet when creating peer-to-peer applications.
System.Windows.Forms, System.Windows.Forms.Design	These namespaces are used to create Windows-based applications using Windows user interface components.

System.Web, System.WebCaching, System.Web.UI, System.Web.UI.Design, System.Web.UI.WebControls, System.Web.UI.HtmlControls, System.Web.Configuration, System.Web.Hosting, System.Web.Mail, System.Web.SessionState	These are used to create ASP. NET Web applications that run over the web.
System.Web.Services, System.Web.Services.Description, System.Web.Services.Configuration, System.Web.Services.Discovery, System.Web.Services.Protocols	These are used to create XML Web services and components that can be published over the web.
System.Security, System.Security.Permissions, System.Security.Policy, System.WebSecurity, System.Security.Cryptography	These are used for authentication, authorization, and encryption purpose.
System.Xml, System.Xml.Schema, System.Xml.Serialization, System.Xml.XPath, System.Xml.Xsl	These namespaces are used to create and access XML files.

.NET Framework Base Class Library

.NET Base Class Library is the sub part of the Framework that provides library support to Common Language Runtime to work properly. It includes the System namespace and core types of the .NET framework.



Components of .NET Framework

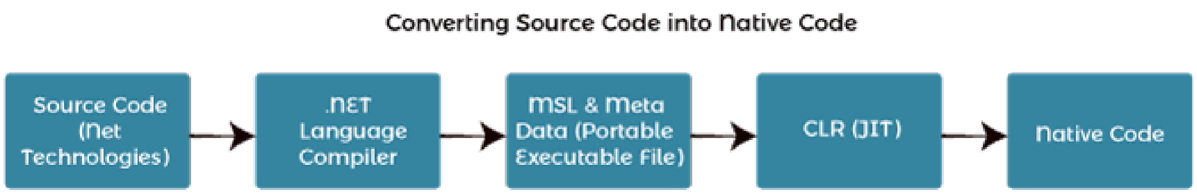
There are following components of .NET Framework:

1. CLR (Common Language Runtime)
2. CTS (Common Type System)
3. BCL (Base Class Library)
4. CLS (Common Language Specification)
5. FCL (Framework Class Library)
6. .NET Assemblies
7. XML Web Services
8. Window Services

.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.

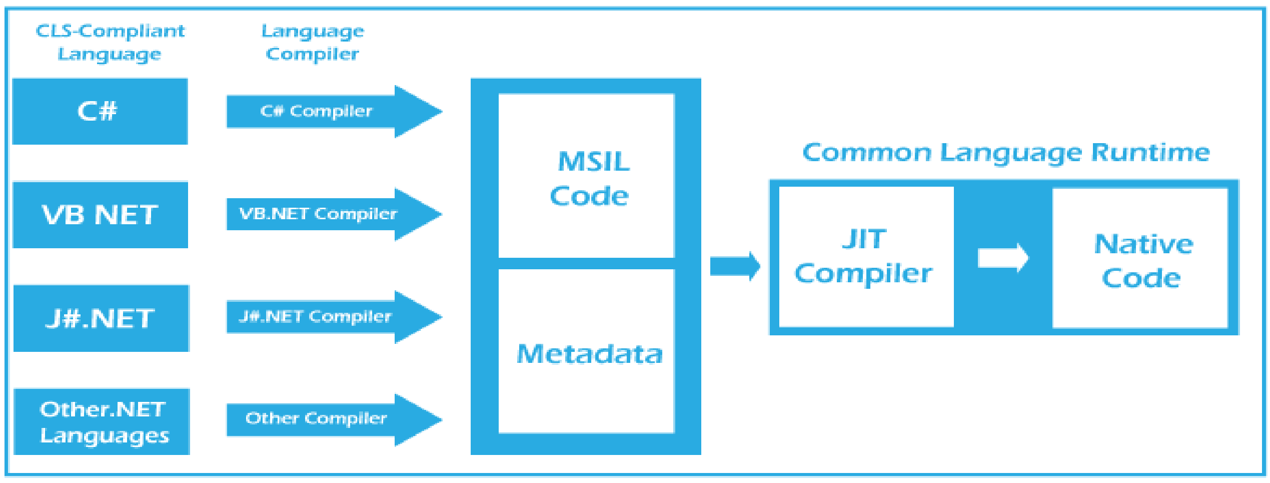


Converting Source Code into Native Code
Source Code (Net Technologies)
.NET Language Compiler
MSL & Meta Data (Portable Executable File)
CLR (JIT)
Native Code

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

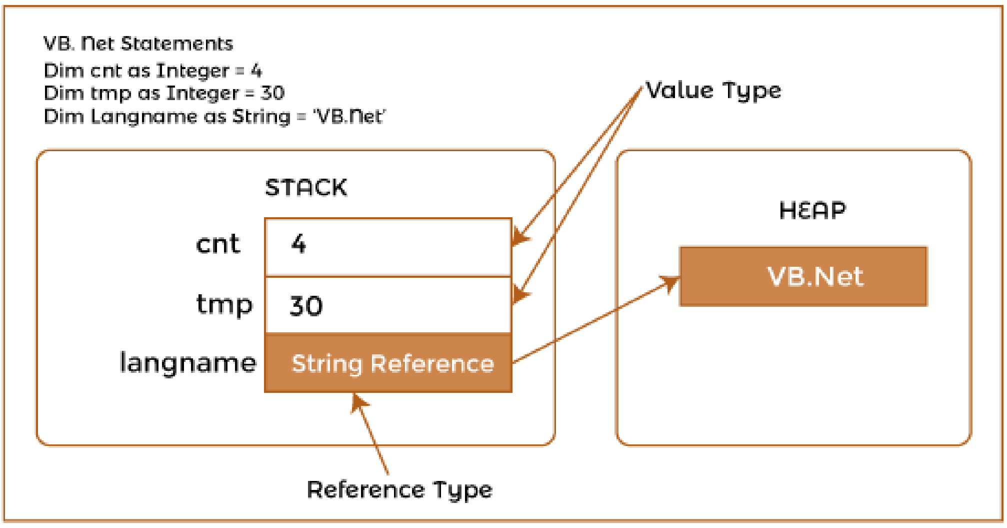
1. Common type system:

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.

- **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



VB. Net Statements

```
Dim cnt as Integer = 4
Dim tmp as Integer = 30
Dim Langname as String = 'VB.Net'
```

- **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 Functions

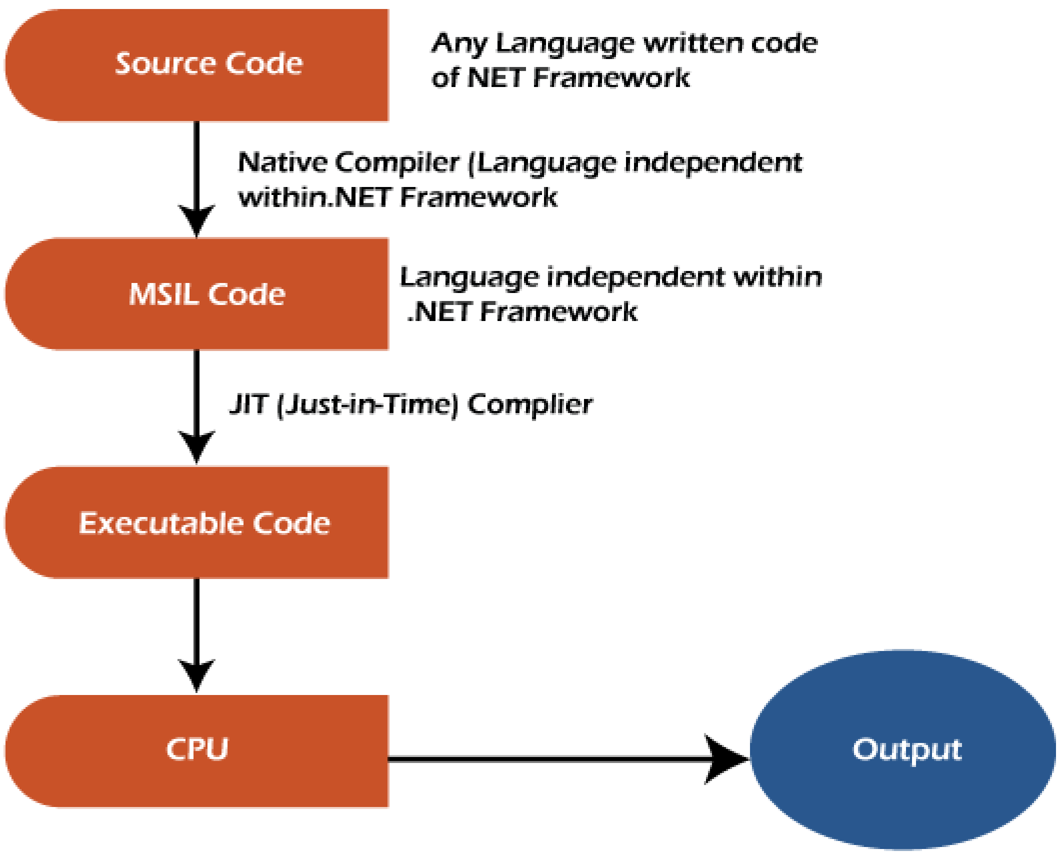
Following are the functions of the CLR.

- 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:

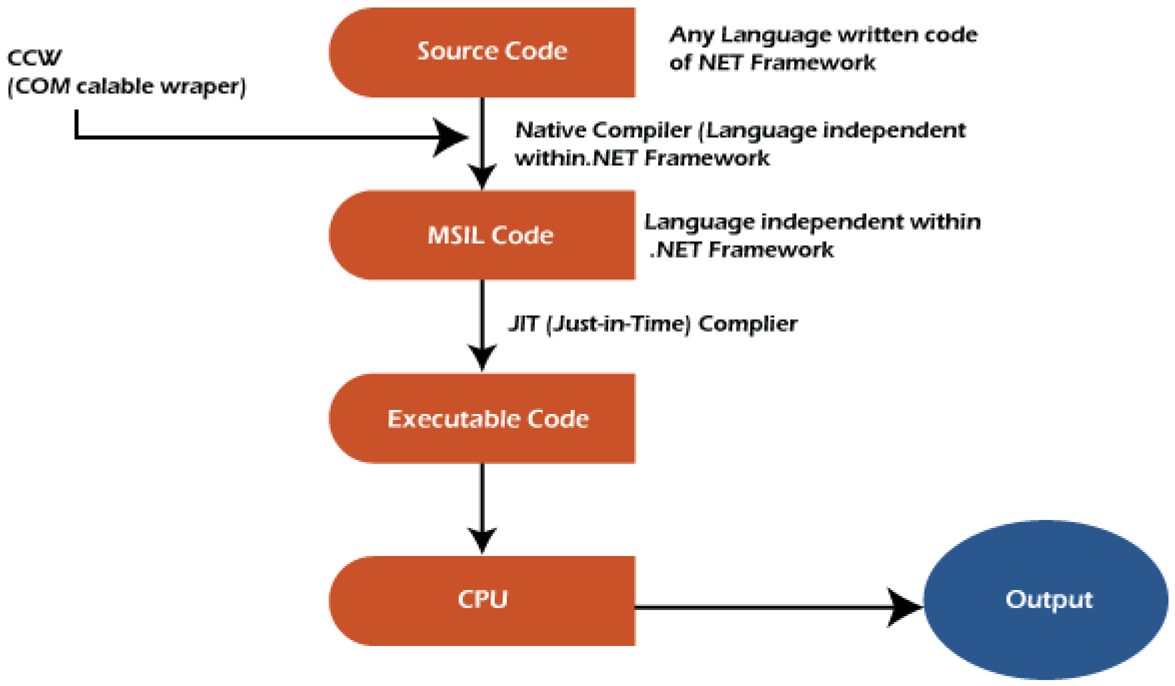
Any language that is written in the .NET framework is managed code. Managed code use CLR, which looks after your applications by managing memory, handling security, allowing cross-language debugging, etc. The process of managed code is shown in the figure:


Source Code
Any Language written code of NET Framework
Native Compiler (Language independent within.NET Framework
MSIL Code
Language independent within .NET Framework
JIT (Just-in-Time) Complier
Executable Code
CPU
Output

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:


CCW
(COM calable wraper)
Source Code
Any Language written code of NET Framework
Native Compiler (Language independent within.NET Framework
MSIL Code
Language independent within .NET Framework
JIT (Just-in-Time) Complier
Executable Code
CPU
Output

Unmanaged Code Execution Process

.NET CLR Versions

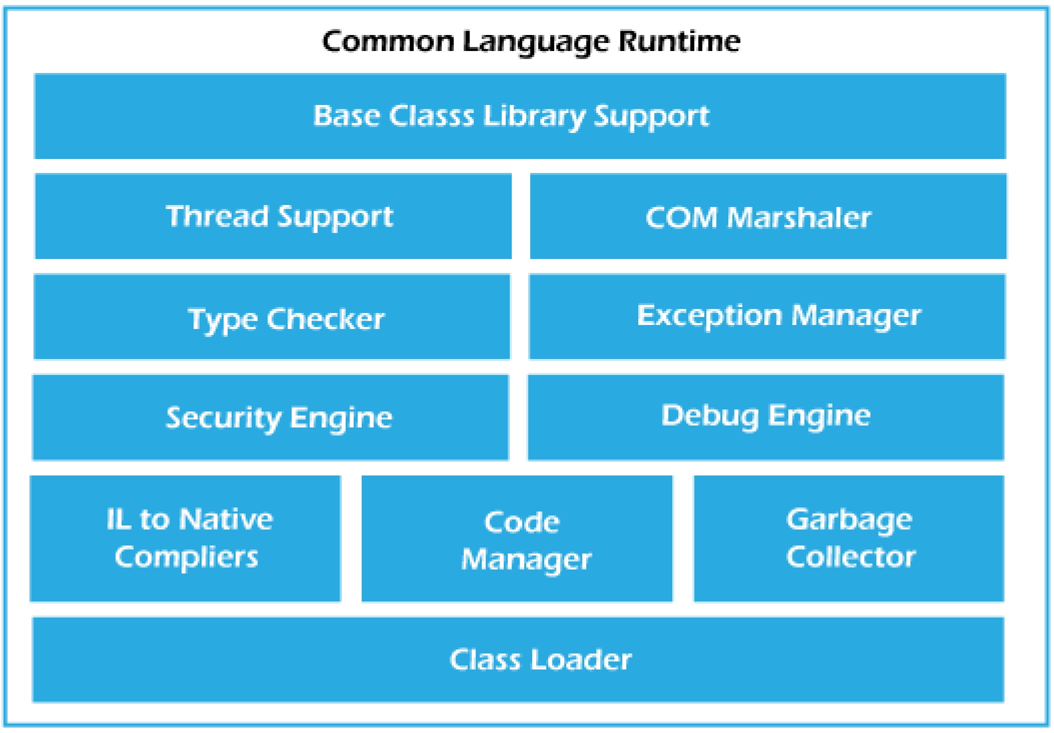
The CLR updates itself time to time to provide better performance.

.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

.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 run time.

DLL (Dynamic Link Library)

Dynamic-Link Library (DLL) is a shared library concept that was introduced in the Microsoft Windows Operating System. **DLLs** are collections of code, data, and resources that can be used by multiple applications simultaneously. They offer several advantages over static libraries, such as reduced memory consumption, faster startup time, and easier maintenance. In this article, we will see what a **DLL** is in C# and how it can be used to build modular and extensible applications.

What is a DLL?

A **DLL** is a binary file that has code and data that can be utilized by multiple applications at the same time. The **DLL** is loaded into the memory of each application that uses it, and the code and data can be accessed by those applications as if they were part of the application's code. This makes **DLLs** a powerful tool for building modular and extensible applications.

A **DLL** can contain any type of code or data that can be used by an application, including functions, classes, variables, and resources. When an application needs to use a **DLL**, it loads the **DLL** into memory and calls the functions or uses the data it contains. Once the application is done with the **DLL**, it can unload it from memory.

In C#, a **DLL** is a compiled assembly that contains **.NET** Framework code. It is created by compiling one or more C# source files into a **DLL** file. The **DLL** file can then be referenced by other C# projects, allowing them to use the code and data that it contains.

Advantages of Using DLLs in C#:

There are several advantages to using **DLLs** in C#:

Reusability:

DLLs allow code to be shared between multiple applications. This can help reduce development time and improve code maintainability.

Modularity:

DLLs allow code to be organized into separate modules, each of which can be loaded and unloaded independently. This can help reduce memory usage and improve application startup time.

Extensibility:

DLLs can be used to add functionality to an application without modifying its existing code. This can be useful for adding plugins or extensions to an application.

Versioning:

DLLs can be versioned, which allows different versions of a **DLL** to be used by different applications. This can help prevent compatibility issues between applications that use different versions of the same **DLL**.

Creating a DLL in C#:

Creating a **DLL** in C# is a straightforward process. Here are the steps involved:

1. Create a new C# project in Visual Studio.
2. Add the code that you want to include in the **DLL** to the project.
3. Build the project to create a **DLL**
4. Reference the **DLL** file from other C# projects that need to use the code that it contains.

Let's walk through these steps in more detail.

Step 1: Create a New C# project

To create a new C# project in Visual Studio, follow these steps:

- Open Visual Studio.
- From the File menu, select New > Project.
- In the New Project dialog box, select C# from the list of project types.
- Select Class Library from the list of the given templates.
- Choose a name and location for your project, and then click Create.

Step 2: Add the Code to the Project

Once you have created the project, you can add the code that you want to include in the **DLL**. This can include functions, classes, variables, and resources. Here is an example of a simple function that can be included in a DLL:

C# Code:

```
namespace MyLibrary
{
   public class MyFunctions
   {
      public static int Add(int a, int b)
      {
         return a + b;
      }
   }
}
```

Step 3: Build the project

Once you have added the code to the project, you can build the project to create a **DLL** file. To do this, follow these steps:

- From the Build menu, select Build Solution.
- Visual Studio will compile the code and create a **DLL** file in the output folder for the project. By default, this folder is located in the bin\Debug or bin\Release folder of the project directory, depending on the build configuration.

Step 4: Reference the DLL From Other C# projects

To use the code that you have included in the **DLL** from other C# projects, you need to reference the **DLL** file. To do this, follow these steps:

- Open the project that needs to use the code from the **DLL**.
- From the Solution Explorer, right-click on the project name and select Add Reference.
- In the Reference Manager dialog box, select Browse and navigate to the location of the **DLL**
- Select the **DLL** file and click Add.

The **DLL** file should now be listed in the References section of the project. You can now use the code that it contains in your project.

Using a DLL in C#:

Once you have created a **DLL** and referenced it from another C# project, you can use the code that it contains in your project. Below is an example of how to use the Add function that we created earlier:

C# Code:

```
using MyLibrary;

int result = MyFunctions.Add(2, 3);
Console.WriteLine(result); // Output: 5
```

In this example, we have included a using directive to import the *MyLibrary* namespace, which contains the *MyFunctions* class that we created earlier. We can then call the Add function from the *MyFunctions* class to add two integers together.

Conclusion:

DLLs are a powerful tool for building modular and extensible applications in C#. They allow code to be shared between multiple applications, organized into separate modules, and added to an application without modifying its existing code. Creating a **DLL** in C# is a straightforward process that involves creating a new project, adding code to the project, building the project, and referencing the **DLL** file from other C# projects. Once a **DLL** has been created and referenced, its code and data can be accessed by other applications as if they were part of the application's own code.

What is Meta Data?

Metadata is data about the data or documentation about the information which is required by the users. In data warehousing, metadata is one of the essential aspects.

Metadata includes the following:

1. The location and descriptions of warehouse systems and components.
2. Names, definitions, structures, and content of data-warehouse and end-users views.
3. Identification of authoritative data sources.
4. Integration and transformation rules used to populate data.
5. Integration and transformation rules used to deliver information to end-user analytical tools.
6. Subscription information for information delivery to analysis subscribers.
7. Metrics used to analyze warehouses usage and performance.
8. Security authorizations, access control list, etc.

Metadata is used for building, maintaining, managing, and using the data warehouses. Metadata allow users access to help understand the content and find data.

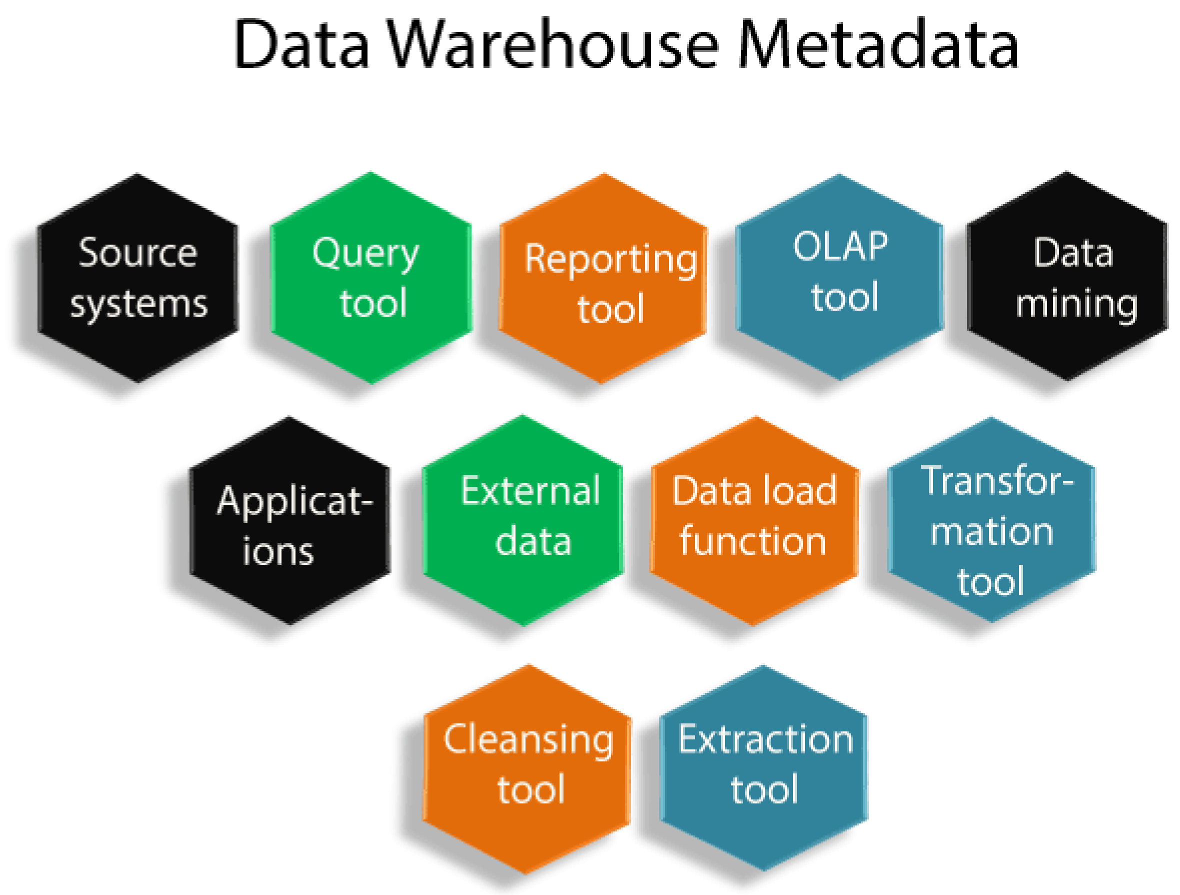
Several examples of metadata are:

1. A library catalog may be considered metadata. The directory metadata consists of several predefined components representing specific attributes of a resource, and each item can have one or more values. These components could be the name of the author, the name of the document, the publisher's name, the publication date, and the methods to which it belongs.
2. The table of content and the index in a book may be treated metadata for the book.
3. Suppose we say that a data item about a person is 80. This must be defined by noting that it is the person's weight and the unit is kilograms. Therefore, (weight, kilograms) is the metadata about the data is 80.
4. Another examples of metadata are data about the tables and figures in a report like this book. A table (which is a record) has a name (e.g., table titles), and there are column names of the tables that may be treated metadata. The figures also have titles or names.

Why is metadata necessary in a data warehouses?

- First, it acts as the glue that links all parts of the data warehouses.
- Next, it provides information about the contents and structures to the developers.
- Finally, it opens the doors to the end-users and makes the contents recognizable in their terms.

Metadata is Like a **Nerve Center**. Various processes during the building and administering of the data warehouse generate parts of the data warehouse metadata. Another uses parts of metadata generated by one process. In the data warehouse, metadata assumes a key position and enables communication among various methods. It acts as a nerve centre in the data warehouse.



Types of Metadata

Metadata in a data warehouse fall into three major parts:

- Operational Metadata
- Extraction and Transformation Metadata
- End-User Metadata

Operational Metadata

As we know, data for the data warehouse comes from various operational systems of the enterprise. These source systems include different data structures. The data elements selected for the data warehouse have various fields lengths and data types.

In selecting information from the source systems for the data warehouses, we divide records, combine factor of documents from different source files, and deal with multiple coding schemes and field lengths. When we deliver information to the end-users, we must be able to tie that back to the source data sets. Operational metadata contains all of this information about the operational data sources.

Extraction and Transformation Metadata

Extraction and transformation metadata include data about the removal of data from the source systems, namely, the extraction frequencies, extraction methods, and business rules for the data extraction. Also, this category of metadata contains information about all the data transformation that takes place in the data staging area.

End-User Metadata

The end-user metadata is the navigational map of the data warehouses. It enables the end-users to find data from the data warehouses. The end-user metadata allows the end-users to use their business terminology and look for the information in those ways in which they usually think of the business.

Metadata Interchange Initiative

The metadata interchange initiative was proposed to bring industry vendors and user together to address a variety of severe problems and issues concerning exchanging, sharing, and managing metadata. The goal of metadata interchange standard is to define an extensible mechanism that will allow the vendor to exchange standard metadata as well as carry along "proprietary" metadata. The founding members agreed on the following initial goals:

1. Creating a vendor-independent, industry-defined, and maintained standard access mechanisms and application programming interfaces (API) for metadata.
2. Enabling users to control and manage the access and manipulation of metadata in their unique environment through the use of interchange standards-compliant tools.
3. Users are allowed to build tools that meet their needs and also will enable them to adjust accordingly to those tools configurations.
4. Allowing individual tools to satisfy their metadata requirements freely and efficiently within the content of an interchange model.

5. Describing a simple, clean implementation infrastructure which will facilitate compliance and speed up adoption by minimizing the amount of modification.
6. To create a procedure and process not only for maintaining and establishing the interchange standard specification but also for updating and extending it over time.

Metadata Interchange Standard Framework

Interchange standard metadata model implementation assumes that the metadata itself may be stored in storage format of any type: ASCII files, relational tables, fixed or customized formats, etc.

It is a framework that is based on a framework that will translate an access request into the standard interchange index.

Several approaches have been proposed in metadata interchange coalition:

- Procedural Approach
- ASCII Batch Approach
- Hybrid Approach

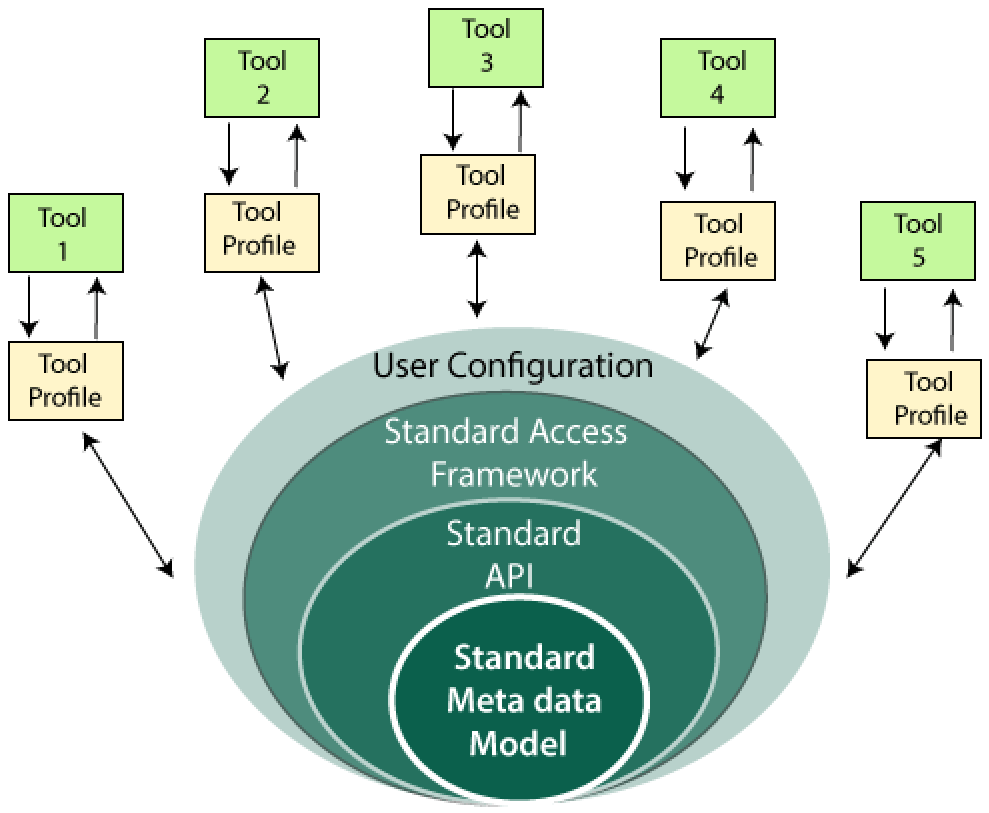
In a **procedural approach**, the communication with API is built into the tool. It enables the highest degree of flexibility.

In **ASCII Batch approach**, instead of relying on ASCII file format which contains information of various metadata items and standardized access requirements that make up the interchange standards metadata model.

In the **Hybrid approach**, it follows a data-driven model.

Components of Metadata Interchange Standard Frameworks

1) **Standard Metadata Model**: It refers to the ASCII file format, which is used to represent metadata that is being exchanged.



Metadata Interchange Standard Framework

2) The **standard access framework** that describes the minimum number of API functions.

3) Tool profile, which is provided by each tool vendor.

4) The user configuration is a file explaining the legal interchange paths for metadata in the user's environment.

Metadata Repository

The metadata itself is housed in and controlled by the metadata repository. The software of metadata repository management can be used to map the source data to the target database, integrate and transform the data, generate code for data transformation, and to move data to the warehouse.

Benefits of Metadata Repository

1. It provides a set of tools for enterprise-wide metadata management.

2. It eliminates and reduces inconsistency, redundancy, and underutilization.
3. It improves organization control, simplifies management, and accounting of information assets.
4. It increases coordination, understanding, identification, and utilization of information assets.
5. It enforces CASE development standards with the ability to share and reuse metadata.
6. It leverages investment in legacy systems and utilizes existing applications.
7. It provides a relational model for heterogeneous RDBMS to share information.
8. It gives useful data administration tool to manage corporate information assets with the data dictionary.
9. It increases reliability, control, and flexibility of the application development process.

Benefits of Dot Net Framework

C#'s design started on a blank page, taking into account the current state of programming languages. The designers drew on the strengths of existing languages and avoided their flaws. What emerged is a full-featured, object-oriented language of rare beauty and grace. Based on C, C++, and, most would say, Java, C# is a strongly typed language that C++ and Java developers will find very familiar. It is easy to learn and has none of the backward-compatibility issues of C or C++. Created for the .NET environment, C# has a thoroughly modern syntax with accessors (properties), a concept borrowed from COM, and new operators, such as implicit and explicit. While affording easy access to the framework's class library, it also allows direct memory and pointer access with the unsafe keyword. There is also talk of extending language support to templates, or parameterized types.

Among the stated design goals of the C# inventors, Anders Hejlsberg, Scott Wiltamuth, and Peter Golde, was the creation of a modern, simple, general-purpose programming language. They also intended to craft a language flexible enough to handle the contradictory demands of hosted and 10 C# Corner embedded systems running on distributed networks. As you become familiar with C#, you might agree that the inventors have more than succeeded.

You should be aware of an important limitation of C#: When exceptional or time-critical performance is required, it is probably wise to retreat to C or C++. In the C# specifications submitted to the ECMA, the section outlining C#'s design goals clearly states that C# is not intended to compete directly with C or assembly language on performance or executable file size.

Managed Code

Managed code is simply code written to execute in the .NET environment. Managed code offers a few benefits in terms of integrated security, type-safe code, and automatic memory allocation and deallocation.

In the .NET world, there is no delete keyword. No longer are hours, if not days, spent tracking down a stubborn memory leak. Memory allocation and deallocation is done via the common language runtime (CLR). This garbage collection is handled by the System.GC, which periodically walks the heap to clean up obsolete objects. Although you are not required to dereference objects with null, it is still good programming practice to do so; and it delivers added benefit in that the Garbage Collector (GC) is given early notice that the object is not needed.

Security is integrated into .NET from the ground up. With .NET comes support for role-based security and code access security. In traditional role-based security, authentication (the verification of a person or process) and authorization (a means of determining that a person or process is permitted to perform a requested process) are carried out by any one of many commonly used mechanisms, such as .NET Passport, NTLM, or an application-defined mechanism. Code access security takes a different approach. The idea behind this .NET innovation is that code itself can be malicious, as any victim of a virus or worm will attest. Code access security allows code behavior to be restricted by varying degrees, depending on the code's origin and identity.

Currently, most application boundaries are normally confined to the process level. Any communication outside of the process requires specialized handling such as a COM component. Application domains, a concept introduced in .NET, permit multiple applications to run in a single process. Before managed code runs, it goes through a verification procedure to establish that it will not access memory or perform an action that would cause the process to crash. If the code passes this verification, it is said to be type safe. Type-safe code allows the CLR to define application domains, which carry the same level of isolation and protection as a process. While not totally free of charge, interapplication communication, across application domains, avoids most of the performance penalties associated with interprocess communication.

Cross-Language Operation

The hurdle of calling methods across different languages is almost as high as the barrier to interaction between different operating systems. TCP/IP aided in communication between operating systems, just as COM components went a long way to facilitate cross-language conversations. But there were constraints. Visual Basic only has signed integer and long data types, while Unicode characters are not native to C++. In many instances, performance would suffer as data was converted or cast back and forth. In .NET, all compliant languages are equal; and, with Microsoft Intermediate Language (MSIL), all operating systems are potentially equal as well. Efforts toward platform independence are taking shape with the Mono Project, a .NET implementation on Linux. A .NET-compliant language is one that adheres to the Common Language Specification (CLS) and the Common Type System (CTS) laid out in the CLI. It is now possible to have C#, managed C++, and VB.NET code in the same module. Other compliant languages include J# (Java Sharp), Cobol, .NET and the .NET Framework 11 and Perl. MSIL has made programming languages transparent. Systems can be developed in the language or languages of choice.

Of course, becoming a compliant language might force some concessions to be made. Visual Basic appears to have mutated to such an extent that VB.NET could qualify as a new .NET language along with C#. C++ is confined to single inheritance. In all likelihood, other languages that adhere to the CLS will have minor restrictions placed upon them.

This newfound freedom may come at a cost. It brings to mind an old adage in the C world: C gives the developer plenty of freedom-enough to hang himself! In the same way, developing an application in multiple .NET languages sows the seeds for project failure. An application written using C#, VB.NET, J#, Cobol, and Perl requires an army of skill sets, rather than a cohesive team, to maintain the code. C# will likely be the language of choice for most projects, because of current and future features such as the unsafe keyword and possible support for templates. Other serious contenders are probably VB.NET and J#.

The Class Library

What can be said about the framework's class library? It is extensive and extensible. There is finally a consistent method to access system resources spanning the many .NET-compliant languages. Certainly, at first glance, the number of namespaces, classes, and interfaces can be daunting, and here seasoned C++ and Java developers will hit a learning curve. Developers migrating to the .NET Framework from C++ or Java should have an easier transition because they have already grasped object-oriented concepts. Those coming from a procedural language world, such as Visual Basic and Cobol, face a much sharper bend on the road to mastering the .NET Framework.

The class library is broken down into a hierarchy of namespaces. Each namespace's name describes its general purpose. Contained in each namespace are the classes and interfaces, which define specific behaviors and data that contribute to the namespace's functionality. The class library is not language specific and may used by any of .NET's compliant languages.

In conjunction with the CLR, the class library supports not only dynamic runtime binding but also dynamic compile-time binding. Although the term compile-time binding is open to debate, the practice isn't. If an object is not present, it is possible for an application to obtain the source code and compile it on the fly. Calling the compiler, in the System.CodeDom.Compiler namespace, with either a string or a file name returns a reference to the executable code. The results can then be loaded and the object's methods executed, thus giving a whole new meaning to late binding.

The class library is a comprehensive implementation that addresses 80 percent of problem domain needs with an out-of-the-box answer. But, as is always the case, a customized solution is the only alternative in some situations. In designing the class library, the developers paid a great deal of attention to extensibility. The library features a rich set of virtual and abstract classes and methods, as well as a wide range of interfaces to tackle any customization requirement.

Deployment

Windows application distribution has always been fraught with potential disaster. COM executables and DLLs have to be registered. Likely, every developer has had nightmares involving the Windows registry. Did I register everything? Are all the proxy/stubs accounted for? And that's only for local machine registry-in DCOM the proxy/stub has to be registered on both client and server machines. And if you're using a customized surrogate, you had better start writing an installation application. Microsoft describes this situation as "DLL Hell."

Updating shared DLLs is also a form of DLL Hell and commonly creates havoc. The newer DLL versions overwrite the current ones, or registration conflicts occur. The newer versions are backward-compatible with only some of the existing software. Consequently, some applications run normally while others inexplicably crash. Tracking down which DLL is the culprit and where the bug is can be a long, agonizing task. .NET eliminates the clash of old and new with assemblies.

An assembly, the name .NET uses for an executable or DLL, contains a manifest and metadata in addition to the compiled source code. Together, the manifest and metadata fully describe the contents of the assembly. Each assembly is a self-contained unit with compiled source code, a full description of the code and its dependencies, a version number, and security settings. Assemblies can be private or shared. In most cases, to install private assemblies, it is a simple matter of using XCOPY. To uninstall, use DEL. Shared assemblies require a bit more work and are discussed in detail in Chapter 26. Shared assemblies are installed in the global assembly cache (GAC).

Common Language Infrastructure

In previous sections, terms such as CLI and CTS have been bandied about without any explanation. Beginning with this section, we'll start to explore what makes .NET run. The CLI can be broken down into three main components:

Common Type System

The CTS lays out the basic data types in order for compiler designers to build a .NET-compliant language.

Common Language Specification

The CLS establishes the minimum set of rules to promote language interoperability. In conjunction with the CTS, it describes the contract all compilers must meet when targeting .NET.

Virtual Execution System

The VES is responsible for loading and running programs and providing services for those programs.

These three elements, and the use of metadata by the VES, constitute the CLI. Without putting too fine a point on it, the CLI can almost be thought of as specifications for a virtual operating system or machine. In the Windows world, the CLI comes to life in the CLR.