

According to Microsoft .Net is a platform built on open internet protocols & standards with tools and services that meld computing and communication in new ways

It's an environment for developing and running software applications featuring ease of development of web based services, rich standard runtime services available to components written in variety of programming languages & provides inter language & inter machine interoperability

The .NET Framework is a new computing platform that simplifies application development in the highly distributed environment of the Internet. The .NET Framework is designed to fulfill the following objectives:

- Preserve consistency between objects that are stored and executed locally, objects that are Internet-distributed but are executed locally, and objects that are stored and executed remotely on a server.
- Avoid deployment and versioning conflicts. No DLL update upon deployment. Solves the DLL Hell Syndrome.
- Guarantee safe execution of code, even if written by unknown semi-trusted third party. Avoid the slowness of scripted or interpreted languages.
- Preserve development environment consistency between windows-based applications and web-based applications.

The .NET Framework includes:

- 1) Application services
- 2) BCL (Base Class Library)
- 3) CLR (Common Language Runtime)

TYPES OF MS.NET APPLICATION

1. Console Based Applications (e.g. Compiler)
2. Windows Application (WinForms)
3. Windows Services
4. ASP.NET Web Applications
5. ASP.NET Web Services
6. Remoting Application.
7. Mobile / Smart Device Applications

.NET Base Class Libraries (also called as Framework Class Libraries (FCL))

The .NET base class library is a collection of object-oriented types and interfaces that provide object models and services for many of the complex programming tasks you will face. Most of the types presented by the .NET base class library are fully extensible, allowing you to build types that incorporate your own functionality into your managed code. These class libraries are distributed with MS.NET Framework and works with any language under the common language runtime environment. Therefore if you are familiar with one .NET language then you can easily migrate to other .NET Languages
All the base class libraries are grouped under the root namespace **System**.

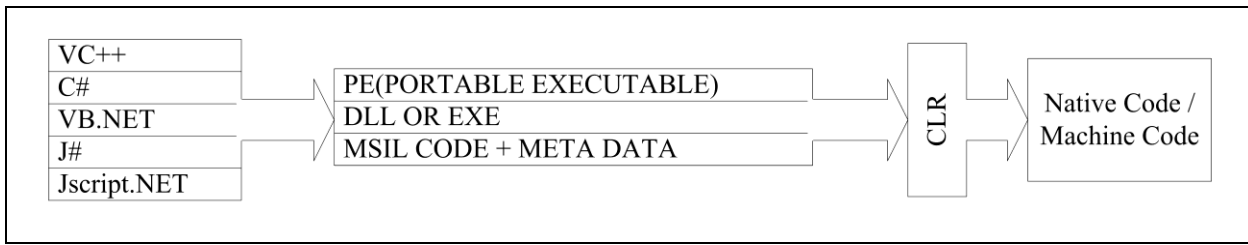
Namespace: A namespace is a logical collection of classes and other types with unique name. The structure of the namespace is like a tree where all the related classes are like leaves.

The most important namespaces in the .NET class library are:

- | | | |
|----------------------|-------------------|------------------------|
| • System | • System.Security | • System.Web.Services |
| • System.IO | • System.Net | • System.Windows.Forms |
| • System.Collections | • System.Data | • System.Drawing |
| • System.Threading | • System.XML | • System.Globaliztion |
| • System.Reflection | • System.Web | • System.Resources |

CLR (Common Language Runtime):

CLR is the foundation of .NET Framework. The common Language Runtime manages code at execution time. It does Memory management, thread management, and runs the code on different platforms (Client or Server). It enforces strict variable type definitions, security, and robustness.



CLR is a component divided in sub components which perform their own respective tasks. CLR as the name specifies provides a common runtime environment for different languages like VC++, C#, VB.NET, J# and JavaScript. The code written in these languages is compiled with their respective language compilers to give a common intermediate language called **MSIL (Microsoft Intermediate Language)** and Metadata. This files generated are called as **PE (Portable Executable)**.

CLR provides the following benefits for the application developers:

- Vastly simplified development.
- Seamless integration of the code written in various languages.
- Evidence-based security with code identity.
- Assembly-based deployment that eliminates DLL Hell.
- Side-by-side versioning of reusable components.
- Code reuse through implementation inheritance.
- Automatic object lifetime management.
- Self describing objects.

MSIL: MSIL is an intermediate instruction set which is processor and hardware independent. The source code when compiled gives MSIL which is an input to the operating system and with the help of CLR is converted into native code which is processor specific.

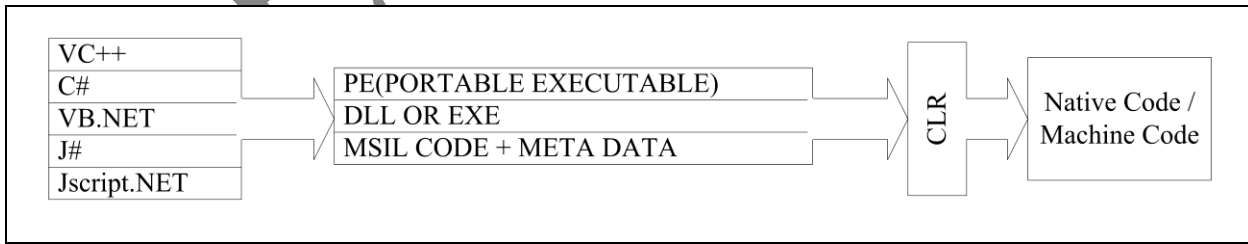
- It stands for Microsoft intermediate language.
- MSIL instructions are pure platform independent.
- MSIL is an intermediate instruction set which is processor and hardware independent.
- The source code when compiled gives MSIL which is an input to the operating system and with the help of CLR is converted into native code which is processor specific.

```
.assembly MyAssembly {}
.class MyApp
{
    method static void Main()
    {
        .entrypoint
        ldstr Hello, IL!"
        call void System.Console::WriteLine(class
                                     System.Object)

        ret
    }
}
```

You can write **IL programs** directly, for example:

Note: Just put this into a file called **hello.il**, and then run **ilasm hello.il**. An exe assembly will be generated.

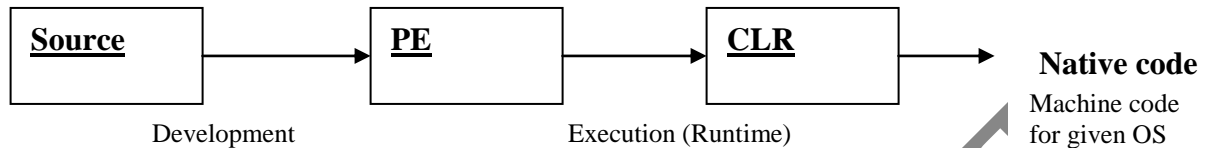


MSIL Assembler(Iiasm) :The MSIL Assembler generates a portable executable (PE) file from Microsoft intermediate language (MSIL).

MSIL DisAssembler(Ildasm): You can use the MSIL Disassembler (Ildasm.exe) to view Microsoft intermediate language (MSIL) information in a file. If the file being examined is an assembly, this information can include the assembly's attributes, as well as references to other modules and assemblies. This information can be helpful in determining whether a file is an assembly or part of an assembly, and whether the file has references to other modules or assemblies.

Microsoft supply a tool called **Ildasm**, which can be used to view the metadata and IL for an assembly. Source code can be reverse-engineered from IL, it is often relatively straightforward to regenerate high-level source (e.g. C#) from IL.

Portable Executable (PE) is a Microsoft Win32 compatible format file for .Net applications which contains the MSIL code and Metadata in binary form. It has the extension **.exe or .dll**. PE has COFF (Common Object File Format) specification



Managed Code: means that the complete life cycle and execution is managed by the .NET Common Language Runtime (CLR). The .NET CLR manages the memory on behalf of the managed code, performs garbage collection on the managed heap, perform assembly validation and assembly (component) resolution on behalf of the program. The CLR also maintains the security constraints applied to the managed code.

The managed execution process includes the following steps:

1. Choosing a Compiler: To obtain the benefits provided by the common language runtime, you must use one or more language compilers that target the runtime.
2. Compiling your code to MSIL: Compiling translates your source code into MSIL and generates the required metadata.
3. Compiling MSIL to native code: At execution time, a just-in-time (JIT) compiler translates the MSIL into native code. During this compilation, code must pass a verification process that examines the MSIL and metadata to find out whether the code can be determined to be type safe.
4. Running code: The common language runtime provides the infrastructure that enables execution to take place as well as a variety of services that can be used during execution.

Note: By **un-safe code**, it means that the managed program can access the memory address using pointers. There are two points to remember here

- Un-safe code is different from un-managed as it is still managed by the CLR
- You still can not perform pointer arithmetic in un-safe code.

Un-managed code runs outside the CLR control while the unsafe code runs inside the CLR's control. Both un-safe and un-managed codes may use pointers and direct memory addresses.

Metadata: Metadata is binary information describing your program that is stored either in a common language runtime portable executable (PE) file or in memory. When you compile your code into a PE file, metadata is inserted into one portion of the file, while your code is converted to Microsoft intermediate language (MSIL) and inserted into another portion of the file. Every type and member defined and referenced in a module or assembly is described within metadata. When code is executed, the runtime loads metadata into memory and references it to discover information about your code's classes, members, inheritance, and so on.

Metadata describes every type and member defined in your code in a language-neutral manner. Metadata stores the following information:

- a) Description of the assembly.
 - Identity (name, version, culture, public key).
 - The types that are exported.
 - Other assemblies that this assembly depends on.
 - Security permissions needed to run.
- b) Description of types.
 - Name, visibility, base class, and interfaces implemented.
 - Members (methods, fields, properties, events, nested types).
- c) Attributes.
 - Additional descriptive elements that modify types and members

To enable the runtime to provide services to managed code, language compilers must emit metadata that describes the types, members, and references in your code. Metadata is stored with the code; every loadable common language runtime portable executable (PE) file contains metadata. The runtime uses metadata to locate and load classes, lay out instances in memory, resolve method invocations, generate native code, enforce security, and set run-time context boundaries.

To enable the runtime to provide services to managed code, language compilers must emit metadata that describes the types, members, and references in your code. Metadata is stored with the code; every loadable common language runtime portable executable (PE) file contains metadata. The runtime uses metadata to locate and load classes, lay out instances in memory, resolve method invocations, generate native code, enforce security, and set run-time context boundaries.

CLR provides the following benefits for the application developers:

- Vastly simplified development.
- Seamless integration of the code written in various languages.
- Evidence-based security with code identity.
- Assembly-based deployment that eliminates DLL Hell.
- Side-by-side versioning of reusable components.
- Code reuse through implementation inheritance.
- Automatic object lifetime management.
- Self describing objects.

What is Type safety?

Defining the data types in a language to be specific and strict in their behavior.

Datatypes are different in different languages and hence cannot be compiled with a single compiler. In .net languages CLR takes the responsibility of verifying the typesafety of the code being executed both at compile time and runtime.

CTS (COMMON TYPE SYSTEM)

It is a specification where different datatypes are defined. This .net datatype collection is used by the language compilers to map the language datatypes to datatypes defined in CTS. This ensures the interoperability at runtime as the code written in one language when invoked from another language would not be misinterpreted the data because the calling language datatypes would map the same CTS datatypes to which the called language datatypes are mapped.

CTS provide a framework for cross-language integration and address a number of issues:

- Similar but subtly different, types (for example, Integer is 16 bits in VB6, but int in C++ is 32 bits; strings in VB6 are represented as BSTRs and in C++ as char pointers or a string class of some sort; and so on)
- Limited code reuse (for example, you can't define a new type in one language and impart into another language)
- Inconsistent object models.

CTS define how types are declared, used, and managed in the runtime, and is also an important part of the runtime's support for cross-language integration. The common type system performs the following functions:

- Establishes a framework that helps enable cross-language integration, type safety, and high performance code execution.
- Provides an object-oriented model that supports the complete implementation of many programming languages.
- Defines rules that languages must follow, which helps ensure that objects written in different languages can interact with each other.

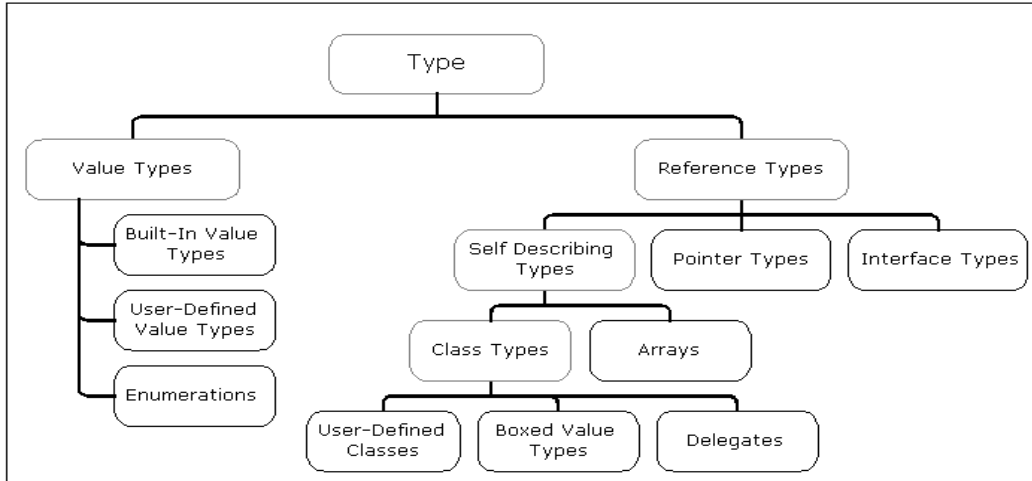
The common type system supports two general categories of types, each of which is further divided into subcategories:

Value Types: Value types directly contain their data, and instances of value types are either allocated on the stack or allocated inline in a structure. Value types can be built-in (implemented by the runtime), user-defined, or enumerations

Reference Types: Reference types store a reference to the value's memory address, and are allocated on the heap. Reference types can be self-describing types, pointer types, or interface types. The type of a reference type can be determined from values of self-describing types. Self-describing types are further split into arrays and class types. The class types are user-defined classes, boxed value types, and delegates.

Variables that are value types each have their own copy of the data, and therefore operations on one variable do not affect other variables. Variables that are reference types can refer to the same object; therefore, operations on one variable can affect the same object referred to by another variable.

All types derive from the System.Object base type.



COMMON LANGUAGE SPECIFICATION:

CLS = Common Language Specification. This is a subset of the CTS, which all .NET languages are expected to support. The idea is that any program, which uses CLS-compliant types, can interoperate with any .NET program written in any language.

The CLS is a set of constructs and constraints that serves as a guide for library writers and compiler writers. It allows libraries to be fully usable from any language supporting the CLS, and for those languages to integrate with each other. The CLS is a subset of the CTS. The CTS is also important to application developers who are writing code that will be used by other developers. When developers design publicly accessible APIs following the rules of the CLS, those APIs are easily used from all other programming languages that target the common language runtime.

In short, this allows very tight interoperability between different .NET languages, for example allowing a C# class to inherit from a VB class.

NOTE: CLS is not a part of CLR Specification.

JIT (JUST IN TIME COMPILER)

JIT compiler compiles the managed or MSIL code into native code. JIT compiles only the required code for execution i.e. as the code is being visited for execution and also the compiled code is cached so that if the block of code is called again the same cached copy is directly executed. Advantage of managed code being compiled in small required fractions is processor time.

There are three types of JIT compilers

- Standard JIT
- Econo JIT = Standard JIT – Caching (this is ideal for small devices like Pocket PC and Mobile Phones)
- Pre JIT

Standard JIT: This compiles a block of code as it is visited for execution. The compiled code is cached so that the subsequent call to the same block of code is directly reused.

Pre JIT: was planned in the initial draft of .NET but was never supported instead Microsoft provided a utility program called **NGen.exe** (Native Generator) here the complete MSIL code is compiled to native code for that machine and is stored in **GAC (Global Assembly Cache)** The advantage being the first time execution is also very fast because its already in precompiled form, but the output of the NGen is not portable and is always machine dependent.

Note: we don't have any facility to set our choice of JIT. For desktops Standard JIT is used and for Compact devices Econo JIT is used.

GARBAGE COLLECTION:

The variables in the application can be allocated memory either in Global memory, Stack memory or Heap memory.

All global variables are allocated memory when the program starts execution and would remain in memory for the lifetime of the application.

All local variables and parameters of a function are allocated memory when the function is called and they are deallocated memory automatically when the function returns.

Heap memory is used for all dynamic memory requirements i.e. variables like pointers for which until runtime we don't know the amount of memory required would be allocated memory from the heap. But the size of heap is limited and all allocations done on heap must be also deallocated when that pointer doesn't need the memory anymore. The deallocated heap memory then can be used for another pointer dynamic memory allocation.

Memory leakage: If some pointers are allocated memory on the heap and are never deallocated from the heap before the pointer itself goes out of scope then such memory can never be freed and would remain as Leakage from the heap as it cannot be allocated to another pointer.

In .Net the memory de-allocation in .Net is handled by **Garbage Collector**. It gets activated when the heap is full and there is a need for release of memory.

Garbage Collector a background thread running within the application / CLR and is responsible for destroying all the unreferenced objects (objects which are no longer in use). When garbage collector is initiated all other threads running within the application are temporarily suspended and hence the sooner the garbage collector finishes its job the better would be the performance of the application.

To optimize the work of garbage collector Microsoft has divided the heap memory into three equal parts called Generations: **GEN0, GEN1, and GEN2**. This is compact garbage collection. New objects are always created in GEN0 and are promoted to other generations based on their lifespan. Long lived objects would be destroyed last as these objects would move from GEN0 to GEN1 and from GEN1 to GEN2.

Note: In .net the address of object can changes at runtime as it goes from one generation to another.

In .Net the object once created are never guaranteed to be destroyed because they might get promoted to higher generations and GC would not visit those generations as memory is mostly available in lower generations itself.

CLASS LOADER

It is a sub component in CLR and is responsible for loading classes and other types as and when needed .It also loads the PE if it's not already loaded.