**What is the common language runtime (CLR)?**

The common language runtime is the execution engine for .NET Framework applications.

It provides a number of services, including the following:

* Code management (loading and execution)
* Application memory isolation
* Verification of type safety
* Conversion of IL to native code
* Access to metadata (enhanced type information)
* Managing memory for managed objects
* Enforcement of code access security
* Exception handling, including cross-language exceptions
* Interoperation between managed code, COM objects, and pre-existing DLLs (unmanaged code and data)
* Automation of object layout
* Support for developer services (profiling, debugging, and so on)

#### What is the common type system (CTS)?

The common type system is a rich type system, built into the common language runtime, that supports the types and operations found in most programming languages. The common type system supports the complete implementation of a wide range of programming languages.

#### What is the Common Language Specification (CLS)?

The Common Language Specification 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 ***Common Language Specification is a subset of the common type system***. The Common Language Specification 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.

#### What is the Microsoft Intermediate Language (MSIL)?

MSIL is the CPU-independent instruction set into which .NET Framework programs are compiled. It contains instructions for loading, storing, initializing, and calling methods on objects.

Combined with metadata and the common type system, MSIL allows for true cross-language integration.

Prior to execution, MSIL is converted to machine code. It is not interpreted.

**What is an assembly?**

An assembly is the primary building block of a .NET Framework application. It is a collection of functionality that is built, versioned, and deployed as a single implementation unit (as one or more files). All managed types and resources are marked either as accessible only within their implementation unit, or as accessible by code outside that unit.

Assemblies are self-describing by means of their manifest, which is an integral part of every assembly. The manifest:

* Establishes the assembly identity (in the form of a text name), version, culture, and digital signature (if the assembly is to be shared across applications).
* Defines what files (by name and file hash) make up the assembly implementation.
* Specifies the types and resources that make up the assembly, including which are exported from the assembly.
* Itemizes the compile-time dependencies on other assemblies.
* Specifies the set of permissions required for the assembly to run properly.

This information is used at run time to resolve references, enforce version binding policy, and validate the integrity of loaded assemblies. The runtime can determine and locate the assembly for any running object, since every type is loaded in the context of an assembly. Assemblies are also the unit at which code access security permissions are applied. The identity evidence for each assembly is considered separately when determining what permissions to grant the code it contains.

The self-describing nature of assemblies also helps makes zero-impact install and XCOPY deployment feasible.

#### What are private assemblies and shared assemblies?

A private assembly is used only by a single application, and is stored in that application's install directory (or a subdirectory therein). A shared assembly is one that can be referenced by more than one application. In order to share an assembly, the assembly must be explicitly built for this purpose by giving it a cryptographically strong name (referred to as a strong name). By contrast, a private assembly name need only be unique within the application that uses it.

By making a distinction between private and shared assemblies, we introduce the notion of sharing as an explicit decision. Simply by deploying private assemblies to an application directory, you can guarantee that that application will run only with the bits it was built and deployed with. References to private assemblies will only be resolved locally to the private application directory.

There are several reasons you may elect to build and use shared assemblies, such as the ability to express version policy. The fact that shared assemblies have a cryptographically strong name means that only the author of the assembly has the key to produce a new version of that assembly. Thus, if you make a policy statement that says you want to accept a new version of an assembly, you can have some confidence that version updates will be controlled and verified by the author. Otherwise, you don't have to accept them.

For locally installed applications, a shared assembly is typically explicitly installed into the global assembly cache (a local cache of assemblies maintained by the .NET Framework). Key to the version management features of the .NET Framework is that downloaded code does not affect the execution of locally installed applications. Downloaded code is put in a special download cache and is not globally available on the machine even if some of the downloaded components are built as shared assemblies.

The classes that ship with the .NET Framework are all built as shared assemblies.

#### If I want to build a shared assembly, does that require the overhead of signing and managing key pairs?

Building a shared assembly does involve working with cryptographic keys. Only the public key is strictly needed when the assembly is being built. Compilers targeting the .NET Framework provide command line options (or use custom attributes) for supplying the public key when building the assembly. It is common to keep a copy of a common public key in a source database and point build scripts to this key. Before the assembly is shipped, the assembly must be fully signed with the corresponding private key. This is done using an SDK tool called SN.exe (Strong Name).

Strong name signing does not involve certificates like Authenticode does. There are no third party organizations involved, no fees to pay, and no certificate chains. In addition, the overhead for verifying a strong name is much less than it is for Authenticode. However, strong names do not make any statements about trusting a particular publisher. Strong names allow you to ensure that the contents of a given assembly haven't been tampered with, and that the assembly loaded on your behalf at run time comes from the same publisher as the one you developed against. But it makes no statement about whether you can trust the identity of that publisher.

#### What is the difference between a namespace and an assembly name?

A namespace is a logical naming scheme for types in which a simple type name, such as MyType, is preceded with a dot-separated hierarchical name. Such a naming scheme is completely under the control of the developer. For example, types MyCompany.FileAccess.A and MyCompany.FileAccess.B might be logically expected to have functionality related to file access. The .NET Framework uses a hierarchical naming scheme for grouping types into logical categories of related functionality, such as the Microsoft® ASP.NET application framework, or remoting functionality. Design tools can make use of namespaces to make it easier for developers to browse and reference types in their code. The concept of a namespace is not related to that of an assembly. A single assembly may contain types whose hierarchical names have different namespace roots, and a logical namespace root may span multiple assemblies. In the .NET Framework, a namespace is a logical design-time naming convenience, whereas an assembly establishes the name scope for types at run time.

#### I've written an assembly that I want to use in more than one application. Where do I deploy it?

Assemblies that are to be used by multiple applications (for example, shared assemblies) are deployed to the global assembly cache. In the prerelease and Beta builds, use the /i option to the GACUtil SDK tool to install an assembly into the cache:

**gacutil /i myDll.dll**

Windows Installer 2.0, which ships with Windows XP and Visual Studio .NET will be able to install assemblies into the global assembly cache.

#### What is an application domain?

An application domain (often AppDomain) is a virtual process that serves to isolate an application. All objects created within the same application scope (in other words, anywhere along the sequence of object activations beginning with the application entry point) are created within the same application domain. Multiple application domains can exist in a single operating system process, making them a lightweight means of application isolation.

An OS process provides isolation by having a distinct memory address space. While this is effective, it is also expensive, and does not scale to the numbers required for large web servers. The Common Language Runtime, on the other hand, enforces application isolation by managing the memory use of code running within the application domain. This ensures that it does not access memory outside the boundaries of the domain. It is important to note that only type-safe code can be managed in this way (the runtime cannot guarantee isolation when unsafe code is loaded in an application domain).

#### What is garbage collection?

Garbage collection is a mechanism that allows the computer to detect when an object can no longer be accessed. It then automatically releases the memory used by that object (as well as calling a clean-up routine, called a "finalizer," which is written by the user). Some garbage collectors, like the one used by .NET, compact memory and therefore decrease your program's working set.

<https://msdn.microsoft.com/en-us/library/ms973837.aspx>

<https://www.codeproject.com/articles/5362/garbage-collection-in-net-a-deeper-look-for-the-be>

#### Difference Between Finalize and Dispose Method

|  |  |
| --- | --- |
| Dispose | Finalize |
| It is used to free unmanaged resources like files, database connections etc. at any time. | It can be used to free unmanaged resources (when you implement it) like files, database connections etc. held by an object before that object is destroyed. |
| Explicitly, it is called by user code and the class which is implementing dispose method, must has to implement IDisposable interface. | Internally, it is called by Garbage Collector and cannot be called by user code. |
| It belongs to IDisposable interface. | It belongs to Object class. |
| It's implemented by implementing IDisposable interface Dispose() method. | It's implemented with the help of destructor in C++ & C#. |
| There is no performance costs associated with Dispose method. | There is performance costs associated with Finalize method since it doesn't clean the memory immediately and called by GC automatically. |

#### [What are the different types of JIT compilers?](http://www.dotnetfunda.com/interviews/exclusive/show/2496/what-are-the-different-types-of-jit-compilers" \o "What are the different types of JIT compilers?" \t "_blank)

1.Standard JIT.  
2.Econo JIT.  
3.Pre JIT.