Introduction to .NET Platform (C#)

C# and the .NET platform were first introduced to the world in 2002 and were intended to offer a much more powerful, more flexible, and simpler programming model than COM. The .NET Framework is a software platform for building systems on the Windows family of operating systems, as well as on numerous non-Microsoft operating systems such as macOS, iOS, Android, and various Unix/Linux distributions. To set the stage, here is a quick rundown of some core features provided courtesy of .NET:

• *Interoperability with existing code*: This is (of course) a good thing. Existing COM software can commingle (i.e., interop) with newer .NET software, and vice versa. As of .NET 4.0 onward, interoperability has been further simplified with the addition of the dynamic keyword.

• *Support for numerous programming languages:* .NET applications can be created using any number of programming languages (C#, Visual Basic, F#, and so on).

• *A common runtime engine shared by all .NET-aware languages:* One aspect of this engine is a well-defined set of types that each .NET-aware language understands.

• *Language integration:* .NET supports cross-language inheritance, cross-language exception handling, and cross-language debugging of code. For example, you can define a base class in C# and extend this type in Visual Basic.

• *A comprehensive base class library:* This library provides thousands of predefined types that allow you to build code libraries, simple terminal applications, graphical desktop applications, and enterprise-level web sites.

• *A simplified deployment model:* Unlike COM, .NET libraries are not registered into the system registry. Furthermore, the .NET platform allows multiple versions of the same .dll to exist in harmony on a single machine.

The key topics that make the major benefits, provided by .NET, possible are: the CLR, CTS and CLS.

Common Type System (CTS)

Another building block of the .NET platform is the **Common Type System (CTS)**. *The CTS specification fully describes all possible data types and all programming constructs supported by the runtime, specifies how these entities can interact with each other, and details how they are represented in the .NET metadata format.*

A given assembly may contain any number of distinct types. In the world of .NET, *type* is simply a general term used to refer to a member from the set {class, interface, structure, enumeration, delegate}. When you build solutions using a .NET-aware language, you will most likely interact with many of these types. For example, your assembly might define a single class that implements some number of interfaces. Perhaps one of the interface methods takes an enumeration type as an input parameter and returns a structure to the caller. So, *the CTS is a formal specification that documents how types must be defined in order to be hosted by the CLR*. Typically, the only individuals who are deeply concerned with the inner workings of the CTS are those building tools and/or compilers that target the .NET platform. It is important, however, for all .NET programmers to learn about how to work with the *five types defined by the CTS* in their language of choice.

Every .NET-aware language supports, at the least, the notion of a ***class type***, which is the cornerstone of object-oriented programming (OOP). A class may be composed of any number of members (such as constructors, properties, methods, and events) and data points (fields).

Interfaces are nothing more than a named collection of abstract member definitions, which may be supported (i.e., implemented) by a given class or structure. By convention, all .NET interfaces begin with a capital letter *I.*

The concept of a structure is also formalized under the CTS. A ***structure*** can be thought of as a lightweight class type having value-based semantics.

Enumerations are a handy programming construct that allow you to group name-value pairs. By default, the storage used to hold each item is a 32-bit integer; however, it is possible to alter this storage slot if need be (e.g., when programming for a low-memory device such as a mobile device).

Delegates are the .NET equivalent of a type-safe, C-style function pointer. The key difference is that a .NET delegate is a *class* that derives from System.MulticastDelegate, rather than a simple pointer to a raw memory address. Delegates are critical when you want to provide a way for one object to forward a call to another object and provide the foundation for the .NET event architecture.

The most types formalized by the CTS take any number of *members*. The CTS defines various adornments that may be associated with a given member. The final aspect of the CTS is that it establishes a well-defined set of fundamental data types. *Although a given language typically has a unique keyword used to declare a fundamental data type, all .NET language keywords ultimately resolve to the same CTS type defined in an assembly named mscorlib.dll.*

Common Language Specification (CLS)

Different languages express the same programming constructs in unique, language-specific terms. The minor syntactic variations are inconsequential in the eyes of the .NET runtime, given that the respective compilers (csc.exe or vbc.exe, in this case) emit a similar set of CIL instructions. However, languages can also differ with regard to their overall level of functionality. For example, a .NET language might or might not have a keyword to represent unsigned data and might or might not support pointer types. Given these possible variations, it would be ideal to have a baseline to which all .NET-aware languages are expected to conform. It is also important to understand that a given .NET-aware language might not support every feature defined by the CTS.

The **Common Language Specification (CLS***), is a related specification that defines a subset of common types and programming constructs that all .NET programming languages can agree on.* Thus, if you build .NET types that expose only CLS-compliant features, you can rest assured that all .NET-aware languages can consume them. Conversely, if you make use of a data type or programming construct that is outside of the bounds of the CLS, you cannot guarantee that every .NET programming language can interact with your .NET code library.

The CLS is a set of rules that describe in vivid detail the minimal and complete set of features a given .NET-aware compiler must support to produce code that can be hosted by the CLR, while at the same time be accessed in a uniform manner by all languages that target the .NET platform. In many ways, the CLS can be viewed as a subset of the full functionality defined by the CTS. Given this rule, you can (correctly) infer that the remaining rules of the CLS do not apply to the logic used to build the inner workings of a .NET type. The only aspects of a type that must conform to the CLS are the member definitions themselves (i.e., naming conventions, parameters, and return types).

Common Language Runtime (CLR)

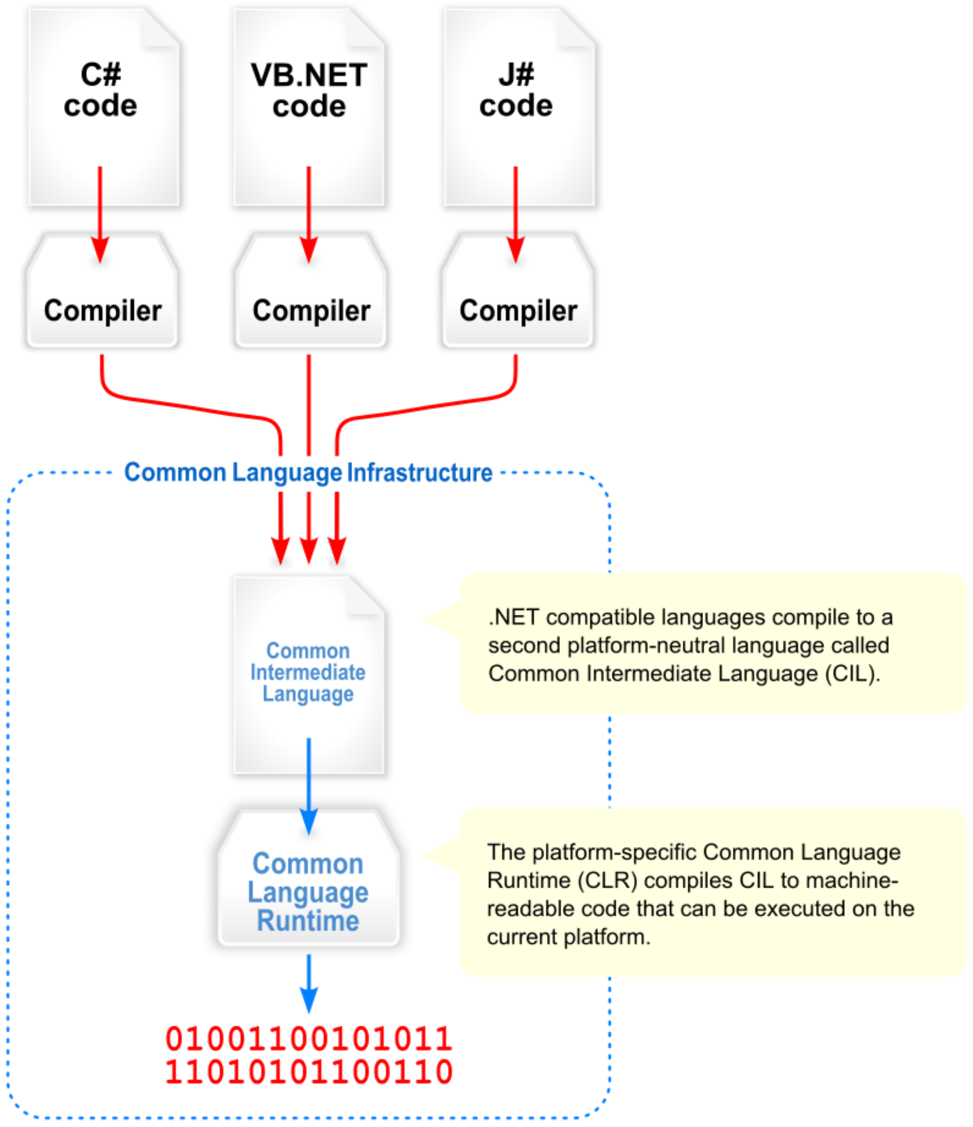
The term <u>runtime</u> can be understood as a collection of services that are required to execute a given compiled unit of code.

The .NET runtime provides a single, well-defined runtime layer that is shared by all languages and platforms that are .NET-aware.

The runtime layer is properly referred to as the **Common Language Runtime (CLR).** *The primary role of the CLR is to locate, load, and manage .NET objects on your behalf*. The CLR also takes care of a number of low-level details such as memory management, application hosting, coordinating threads, and performing basic security checks (among other low-level details).

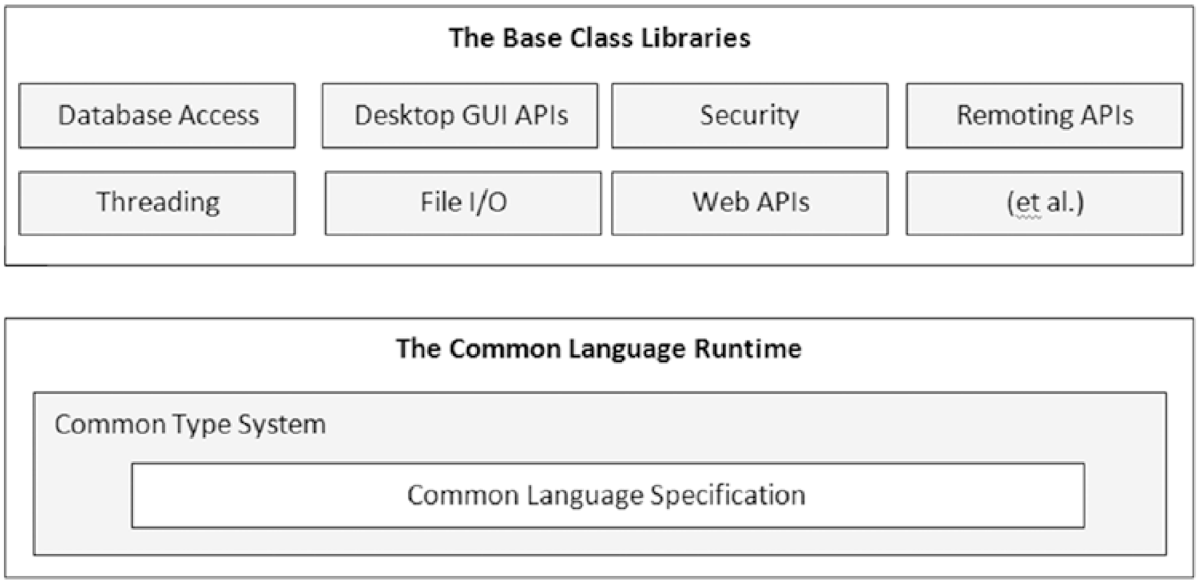
The crux of the CLR is physically represented by a library named mscoree.dll (aka the Common Object Runtime Execution Engine). When an assembly is referenced for use, *mscoree.dll* is loaded automatically, which in turn loads the required assembly into memory. The runtime engine is responsible for a number of tasks. First, it is the agent in charge of resolving the location of an assembly and finding the requested type within the binary by reading the contained metadata. The CLR then lays out the type in memory, compiles the associated CIL into platform-specific instructions, performs any necessary security checks, and then executes the code in question.

In addition to loading your custom assemblies and creating your custom types, the CLR will also interact with the types contained within the .NET base class libraries when required. Although the entire base class library has been broken into a number of discrete assemblies, the key assembly is mscorlib.dll, which contains a large number of core types that encapsulate a wide variety of common programming tasks, as well as the core data types used by all .NET languages. When you build .NET solutions, you automatically have access to this particular assembly.


C# code
VB.NET code
J# code
Compiler
Compiler
Compiler
Common Language Infrastructure
Common Intermediate Language
.NET compatible languages compile to a second platform-neutral language called Common Intermediate Language (CIL).
Common Language Runtime
The platform-specific Common Language Runtime (CLR) compiles CIL to machine-readable code that can be executed on the current platform.
01001100101011
11010101100110

Base Class Library (BCL)

In addition to the CLR, CTS, and CLS specifications, the .NET platform provides a **Base class library** that is available to all .NET programming languages. Not only does this base class library encapsulate various primitives such as threads, file input/output (I/O), graphical rendering systems, and interaction with various external hardware devices, but it also provides support for a number of services required by most real-world applications. *The base class libraries define* types that can be used to build any type of software application. For example, you can use ASP.NET to build web sites and REST services, WCF to build distributed systems, WPF to build desktop GUI applications, and so forth. As well, the base class libraries provide types to interact with the directory and file system on a given computer, communicate with relational databases (via ADO.NET), and so forth.


The Base Class Libraries
Database Access
Desktop GUI APIs
Security
Remoting APIs
Threading
File I/O
Web APIs
(et al.)
The Common Language Runtime
Common Type System
Common Language Specification