Accolite University, July 2020
.Net Framework Introduction
Assignment Submission

By: Ayush Malik

1. Demonstrate the process of conversion of Source code into the native machine code in .Net framework with the help of a flowchart.

The conversion of Source code into native machine code in .Net involves the following 2 stages:

1. Compiler time process.
2. Runtime process.

1. Compiler time process

- The .Net framework has one or more language compilers, such as Visual Basic, C#, Visual C++, JScript, or one of many third-party compilers such as an Eiffel, Perl, or COBOL compiler.
- Any one of the compilers translate your source code into Microsoft Intermediate Language (MSIL) code.
- For example, if you are using the C# programming language to develop an application, when you compile the application, the C# language compiler will convert your source code into Microsoft Intermediate Language (MSIL) code.
- In short, VB.NET, C# and other language compilers generate MSIL code. (In other words, compiling translates your source code into MSIL and generates the required metadata.)
- Currently "Microsoft Intermediate Language" (MSIL) code is also known as "Intermediate Language" (IL) Code or "Common Intermediate Language" (CIL) Code.

  SOURCE CODE -----.NET COMPILER------> BYTECODE (MSIL + METADATA)

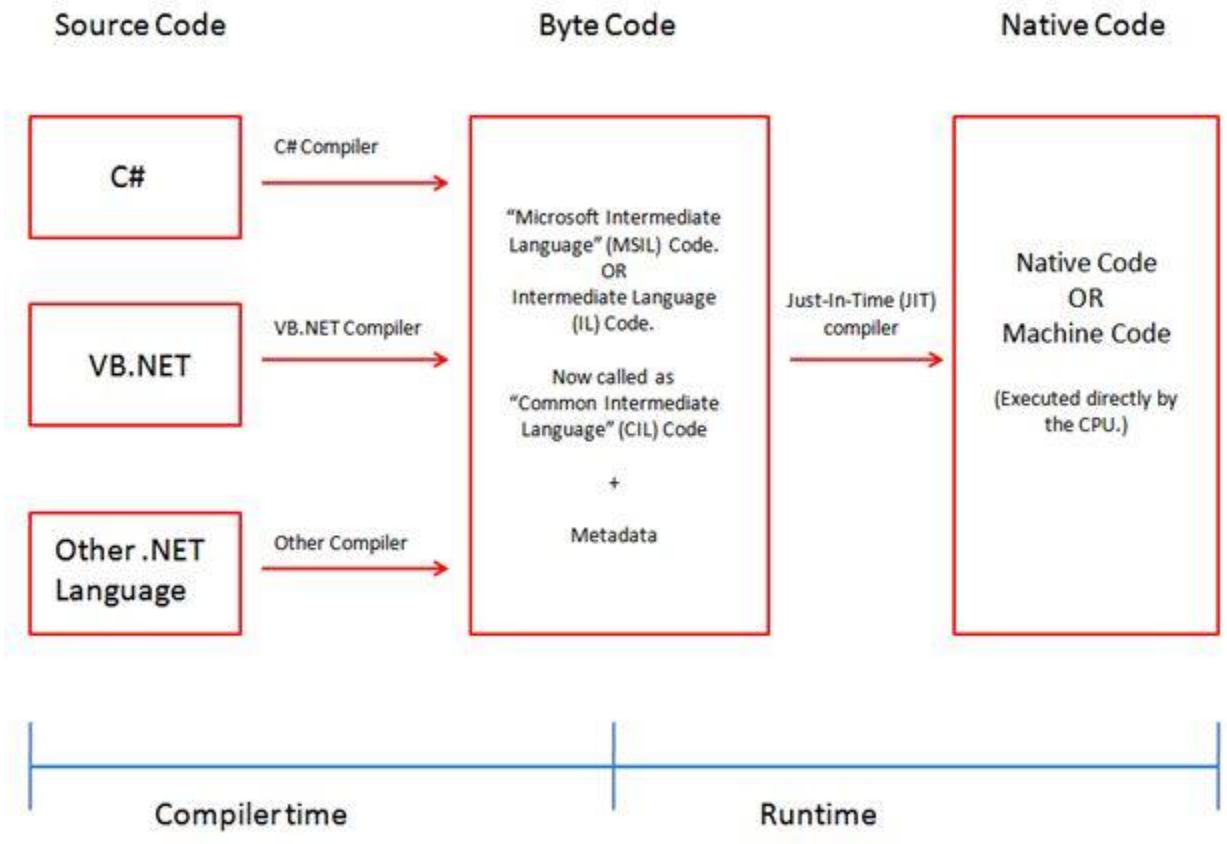
2. Runtime process.

- The Common Language Runtime (CLR) includes a JIT compiler for converting MSIL to native code.
- The JIT Compiler in CLR converts the MSIL code into native machine code that is then executed by the OS.
- Code whose execution is managed by a runtime is called managed code.

- During the runtime of a program the "Just in Time" (JIT) compiler of the Common Language Runtime (CLR) uses the Metadata and converts Microsoft Intermediate Language (MSIL) into native code.

  BYTE CODE (MSIL + METADATA) ----- Just-In-Time (JIT) compiler------> NATIVE CODE

The following flowchart explains the conversion of Source Code into Native machine Code in .Net Framework.


Source Code
Byte Code
Native Code
C#
C# Compiler
VB.NET
VB.NET Compiler
Other .NET Language
Other Compiler
"Microsoft Intermediate Language" (MSIL) Code.
OR
Intermediate Language (IL) Code.
Now called as "Common Intermediate Language" (CIL) Code
+
Metadata
Just-In-Time (JIT) compiler
Native Code
OR
Machine Code
(Executed directly by the CPU.)
Compilertime
Runtime

Conversion of Source Code into Native machine Code in .Net Framework

2. Explain in detail the CTS and how the .net framework implements CTS.

In Microsoft's .NET Framework, the Common Type System (CTS) is a standard that specifies how type definitions and specific values of types are represented in computer memory. It is intended to allow programs written in different programming languages to easily share information. The CTS describes the datatypes that can be used by managed code.

It 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.
- Provides a library that contains the primitive data types (such as Boolean, Byte, Char, Int32, and UInt64) used in application development.

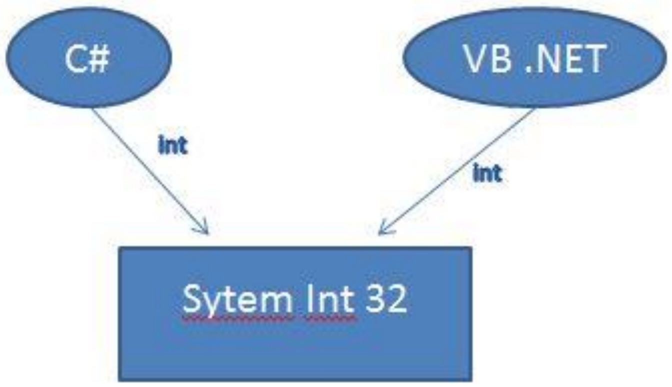
As already explained, the CTS standardizes the data types of all programming languages using .NET under the umbrella of .NET to a common data type for easy and smooth communication among these .NET languages.

For example, you might have some part of your code in VB, some portion in C++ and you prefer to integrate the code from both the languages in your application. In such cases, you might face problems with respect to data types.

If you have a Boolean variable for example, it will be represented as "Boolean" in VB.NET whereas it will be represented as "bool" in C#. Same data type is represented in two different languages in two different ways. All that you need is a common type system that has a single representation of data type irrespective of the language in which it is used.

This unique representation is provided by CTS which stands for Common Type System. CTS represents Boolean variables as System. Boolean which will be acceptable across languages and thereby it ensures successful integration.

To implement or see how CTS is converting the data type to a common data type, for example, when we declare an int type data type in C# and VB.Net then they are converted to int32. In other words, now both will have a common data type that provides flexible communication between these two languages.



A figure showing conversion to common data type in CTS

A .NET implementation is language agnostic. This doesn't just mean that a programmer can write their code in any language that can be compiled to IL. It also means that they need to be able to interact with code written in other languages that are able to be used on a .NET implementation. This is what the Common Type System (CTS) is in charge of doing

All types in .NET are either value types or reference types.

Reference types' objects are represented by a reference to the object's actual value; a reference here is similar to a pointer in C/C++. It simply refers to a memory location where the objects' values are. This has a profound impact on how these types are used. If you assign a reference type to a variable and then pass that variable into a method, for instance, any changes to the object will be reflected on the main object; there is no copying.

Value types are the opposite, where the objects are represented by their values. If you assign a value type to a variable, you are essentially copying a value of the object. The common type system in .NET supports the following five categories of types:

- Classes
- Structures
- Enumerations
- Interfaces
- Delegates

CTS also defines all other properties of the types, such as access modifiers, what are valid type members, how inheritance and overloading works and so on.

3. Name at least 3 runtime services provided by CLR and explain their role in .net framework.

3 of the many runtime services provided by CLR are:

- Code Management
- Base Class Library Support
- Garbage Collection

These 3 have been discussed individually below:

Code Management

Managed code is compiled for the .NET run-time environment. It runs in the Common Language Runtime (CLR), which is the heart of the .NET Framework. Managed applications written to take advantage of the features of the CLR perform more efficiently and safely, and take better advantage of developers' existing expertise in languages that support the .NET Framework.

Unmanaged code includes all application code written before the .NET Framework was introduced—this includes code written to use COM, native Win32, and Visual Basic 6. Because it does not run inside the .NET environment, unmanaged code cannot make use of any .NET managed facilities.

Key features available to managed code applications include:

- Performance benefits gained from executing all code in the CLR. Calling unmanaged code decreases performance because additional security checks are required. Other performance advantages are available through judicious use of the Just-In-Time compiler and NGEN utility.
- Automatic lifetime control of objects, which includes garbage collection and scalability features.
- Ease of deployment and the vastly improved versioning facilities-the end of "DLL hell".
- Built-in security by using code access security and avoiding buffer overruns.

Base Class Library Support

A .NET Framework library, BCL is the standard for the C# runtime library and one of the Common Language Infrastructure (CLI) standard libraries. BCL stands for Base class library also known as Class library (CL). It is the collection of reusable types that are closely integrated with CLR. BCL provides types representing the built-in CLI data types, basic file access, collections, custom attributes, formatting, security attributes, I/O streams, string manipulation, and more. It also helps in performing day to day operation e.g. dealing with string and primitive types, database connection, IO operations, etc.

Garbage Collection

In .NET framework, each and every type identifies some resources which are available for the program. To use these resources, the memory needs to be allocated which represents the type. The following steps are required to access a resource.

- Allocate the memory for the type that represents the resource (via new operator).
- Initialize the memory to set the initial state of the type and make the resource available (object construction).
- Use the resource.
- Tear down the resource.
- And free the memory. This is where the garbage collector comes into the picture.

When the application calls a new operator to create an object, there might not be enough space left in the managed heap to allocate the object. In case of insufficient space, CLR performs garbage collection, which works in the generations. CLR garbage collector is also known as Ephemeral Garbage Collector (Generation based GC). CLR GC makes the following assumptions about the code.

- The lifetime of the new object is short.
- The lifetime of an old object is likely to be longer.
- Collecting a portion of the memory (heap) is comparatively faster than collecting the whole heap.

.NET framework has 3 generations: Gen0, Gen1, and Gen2.Gen0 is the basic layer in which GC would like to perform the memory cleaning and the objects which are on Gen1 and Gen2 are considered to be in memory for a long time and have survived for more than one GC.

CLR kicks-in GC when Gen0 has filled its budget, this is considered to be the most common GC trigger, along with this the following are also responsible as GC triggers,

- The code explicitly calls System. GC’s static Collect method.
- Windows is reporting low memory conditions
- The CLR is shutting down normally, (not via an unhandled exception)
- The CLR is unloading an App Domain

4. What are the differences between Library vs DLL vs .Exe? Explain.

In this solution, first we will understand Library, DLL and .Exe individually and then understand the differences between them via comparative analysis.

Libraries (LIB) are used because we may have code that we want to use in many programs. For example, if we write a function that counts the number of characters in a string, that function will be useful in lots of programs. Once we get that function working correctly, we don't want to have to recompile the code every time we use it, so we put the executable code for that function in a library, and the linker can extract and insert the compiled code into your program. Static libraries are sometimes called 'archives' for this reason.

Dynamic libraries take this one step further. It seems wasteful to have multiple copies of the library functions taking up space in each of the programs. Rather than building the library code into your program when it is compiled, it can be run by mapping it into your program as it is loaded into memory. Multiple programs running at the same time that use the same functions can all share one copy, saving memory. We can load dynamic libraries only as needed, depending on the path through our code. This means we have to have a copy of the dynamic library installed on every machine our program runs on.

The term ‘EXE’ is a shortened version of the word executable as it identifies the file as a program. On the other hand, ‘DLL’ stands for Dynamic Link Library, which commonly contains functions and procedures that can be used by other programs. An EXE is an executable file and is not a supportive file rather itself an application. An EXE will contain an entry point (main function) so runs individually. When coding, we can either export our final project to either a DLL or an EXE. In the basest application package, we would find at least a single EXE file that may or may not be accompanied with one or more DLL files.

Differences between Library and DLL:

- A DLL is a library that contains functions that can be called by applications at run-time while LIB is a static library whose code needs to be called during the compilation.
- Using LIB would result in a single file that is considerably bigger while you end up with multiple smaller files with DLL.
- DLL’s are more reusable than LIBs when writing new versions or totally new applications.
- DLL’s are prone to versioning problems while LIB is not.

Differences between EXE and DLL:

- EXE is an extension used for executable files while DLL is the extension for a dynamic link library.
- An EXE file can be run independently while a DLL is used by other applications.

- An EXE file defines an entry point while a DLL does not.
- A DLL file can be reused by other applications while an EXE cannot.
- A DLL would share the same process and memory space of the calling application while an EXE creates its separate process and memory space.

5. How does CLR in .net ensure security and type safety? Explain.

Security

The common language runtime of the .NET Framework has its own secure execution model that isn't bound by the limitations of the operating system it's running on. In addition, the CLR enforces security policy based on where code is coming from rather than who the user is. This model is called code access security.

CLR implements its own secure execution model that is independent of the host platform. Beyond the benefits of bringing security to platforms that have never had it, this also is an opportunity to impose a more component-centric security model that takes into account the nature of dynamically composed systems.

The CLR implements a code access security model in which privileges are granted to code, not users. Upon loading a new assembly, the CLR gathers evidence about the code's origins. This evidence is associated with the in-memory representation of the assembly by the CLR and is used by the security system to determine what privileges to grant to the newly loaded code. This determination is made by running the evidence through a security policy. The security policy accepts evidence as input and produces a permission set as output. To avoid performance hits, security policy is typically not run until an explicit security demand is made.

Permission sets like those returned by policy are simply a collection of permissions. A permission is a right to perform some trusted operation. The CLR ships with a set of built-in permission types to protect the integrity of the system and the privacy of the user. However, this system is extensible and user-defined permission types can be transparently integrated into the model.

The determination of which code is assigned which permissions is called policy. The enforcement of this policy is done using a distinct set of mechanisms. Prior to executing a privileged operation, trusted code is expected to enforce the security policy by explicitly demanding that the callers have sufficient privileges to perform the operation.

Code access security requires an omniscient and omnipotent runtime. In particular, that means that calling code that is not written to a strict format can thwart the security system. To that end, the Common Language Infrastructure (CLI) categorizes code into two broad families: verifiable code and non-verifiable code. Verifiable code can be mathematically proven to adhere to the type-safe execution model that the CLI encourages. Visual Basic® .NET and C# produce verifiable code by default. To protect the integrity of the system, the ability to load non-verifiable code is itself a permission that must be explicitly granted to code through policy. The default policy that is installed with the CLR grants this permission only to code installed on a local file system.

Type Safety

In .Net type safety check happens both at Compile time and at runtime.

The compiler will do the compile-time type safety check and CLR will do the runtime safety check. At compile time, we get an error when a type instance is being assigned to an incompatible type; hence preventing an error at runtime. So, at compilation time itself, developers come to know such errors and code will be modified to correct the mistake.

Run time type safety ensures we don't get strange memory exceptions and inconsistent behaviour in the application. Type safety is provided by using the Common Type System (CTS) and the Common Language Specification (CLS) that are provided in the CLR to verify the types that are used in an application.

Type safety in .NET had been introduced to prevent the objects of one type from peeking into the memory assigned for the other object. Writing safe code also means to prevent data loss during conversion of one type to another.

Suppose, we have two types defined as in the following:

```
public class MyType
{
  public int Prop{ get; set;}
}
public class YourType
{
   public int Prop{get;set;}
   public int Prop1{get;set;}
}
```

Now let us create an object of MyType as in the following:

```
MyType obj = new MyType();
```

In memory obj would be referencing the 4 bytes of space. Let us assume that next to that part of memory is another string.

Now, we cast obj to YourType. Such an assignment is not at all possible in .NET because of compile time type checking and dynamic type checking at runtime using CLR.

If it would have been possible, the code would look something as in the following,

```
YourType obj1 = (YourType)obj; // Here we will get compile time error
```

If we assign Prop using obj1 reference, as shown below:

```
obj1.Prop = 10;
```

Which is fine but if we want to assign value to Prop1 as shown below:

```
obj1.Prop1 = 20;
```

That can lead to some unpredictable results and bugs since this part of memory could be used by the string. From this we can say that type safety comes to our rescue, when the compiler would not allow the casting that we have tried to do. This is a relief for the programmers with a background in unmanaged programming.

Let us consider one more scenario in which type safety comes to our rescue. If we are using out or ref with our method parameter and when calling the method our argument does not match the method's parameter, the compiler would not allow the code to compile.

Let's take an example for it. Here, we have user type defined as follows:

```
public class SomeType
{
}
```

There is a function that takes a parameter of type object as shown below:

```
private void ChangeValue(out object par)
{
   par = new String('x', 10);
}
```

Now, If we call the method ChangeValue as in the following that would not of course compile:

```
SomeType obj = new SomeType();
ChangeValue(out obj); //compile time error
```

If .NET would have allowed this code to execute, type safety could have easily been compromised here.

If the ChangeValue function had the parameter without the out keyword, there would not have been any compile time error, since the caller can have an argument type that derives from the parameter type.