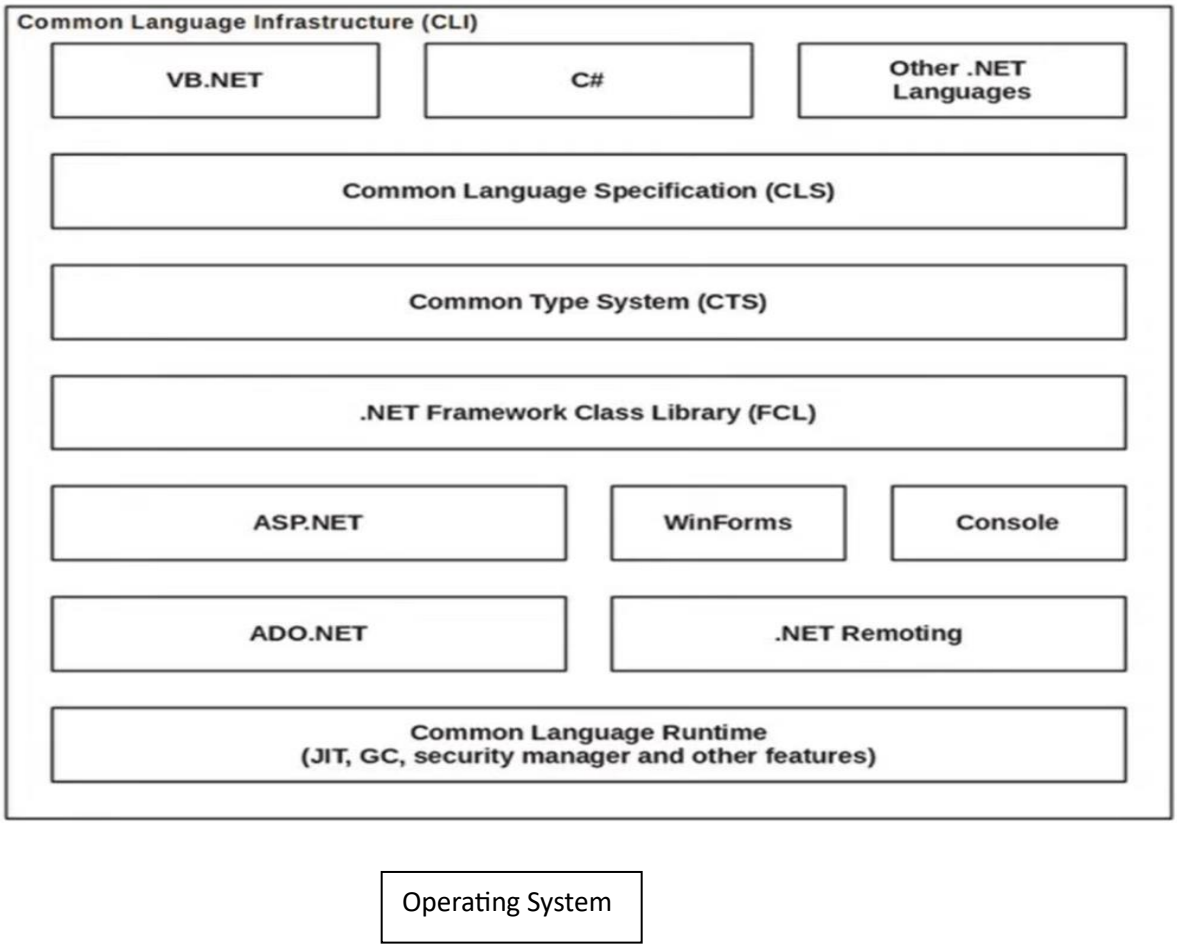
Unit1: .NET Framework, .NET Core and C# Language

- .NET Framework, .NET Core, Xamarin, and .NET Standard are related and overlapping platforms for developers used to build applications and services.

- .NET Framework is a development platform that includes a Common Language Runtime (CLR), which manages the execution of code, and a Base Class Library (BCL), which provides a rich library of classes to build applications from.
- Microsoft originally designed the .NET Framework to have the possibility of being cross-platform, but Microsoft put their implementation effort into making it work best with Windows.
- Since .NET Framework 4.5.2 it has been an official component of the Windows operating system.
- .NET Framework is installed on over one billion computers so it must change as little as possible. Even bug fixes can cause problems, so it is updated infrequently.
- It is a framework that compiles and executes programs written in different languages like C#, VB.Net etc.
- It is used to develop Windows Form-based applications, Web-based applications, and Web services.
- .NET is not a language, rather it provides runtime and a library for writing and executing in any compliant languages (e.g., C#, VB, F#, etc.)

Fig1. .NET Framework Architecture



.NET Framework Services

- Common Language Runtime
- Console App and CLI
- Windows Forms
- ASP.NET
- Web Forms
- Web Services
- ADO.NET for database handling

.NET Development

- Use of .NET supported languages for developing applications
    - -Common Language Specification (CLS)
    - -Common Type System (CTS)
    - - Standard class framework (BCL)
- Automatic memory management
- Consistent error handling and safer execution
- Potentially multi-platform

Deployment

- Removal of registration dependency
- Safety – fewer versioning problems
-

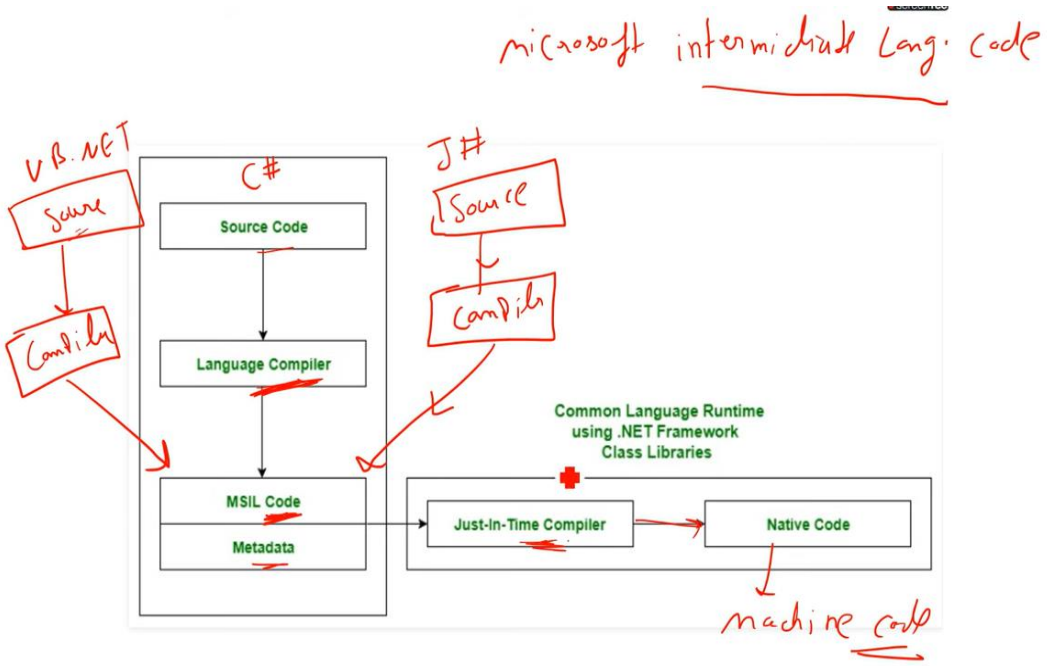
.NET Multiple Language Support

-Common Type System (CTS) is a rich type system built into the CLR

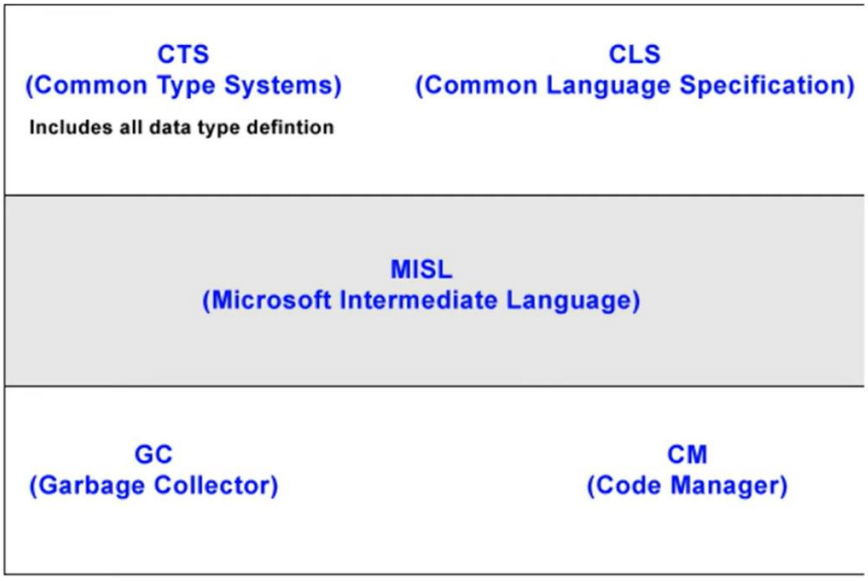
- – Implements various types (int, double, etc)
- – And operations on those types

• Common Language Specification (CLS) is a set of specifications that language and library designers need to follow, which ensure interoperability between languages

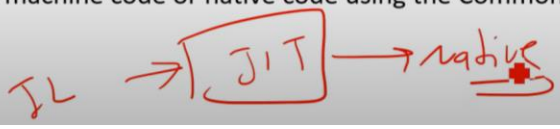
Fig2. Compilation and Execution of .NET Application



Architecture of CLR

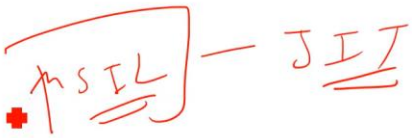


- Common Language Specification (CLS) : It is responsible for converting the different .NET programming language syntactical rules and regulations into CLR understandable format. Basically, it provides the Language Interoperability.
- Common Type System (CTS) :Every programming language has its own data type system, so CTS is responsible for understanding all the data type of system.
- Garbage Collection (GC): It is used to provide the *Automatic Memory Management* feature.
- Just In – Time Compiler (JIT) : It is responsible for converting the CIL(Common Intermediate Language) into machine code or native code using the Common Language Runtime environment.



.Net Framework Design Principle

1. **Interoperability**
2. **Portability**
3. **Security**
4. **Memory management**
5. **Simplified deployment**



Salient Features

- **Less Coding and Increased Reuse of Code**
- **Reliability**
- **Security**
- **Language Interoperability**
- **Deployment**

The Mono and Xamarin projects

- Mono is the original cross-platform implementation of .NET.
- Third parties developed a .NET Framework implementation named the Mono project.
- Mono is cross-platform, but it fell well behind the official implementation of .NET Framework.

- Mono collaborated with the Xamarin mobile platform as well as cross-platform game development platforms like Unity.
- Microsoft purchased Xamarin in 2016 and now gives away what used to be an expensive Xamarin extension for free with Visual Studio 2019.

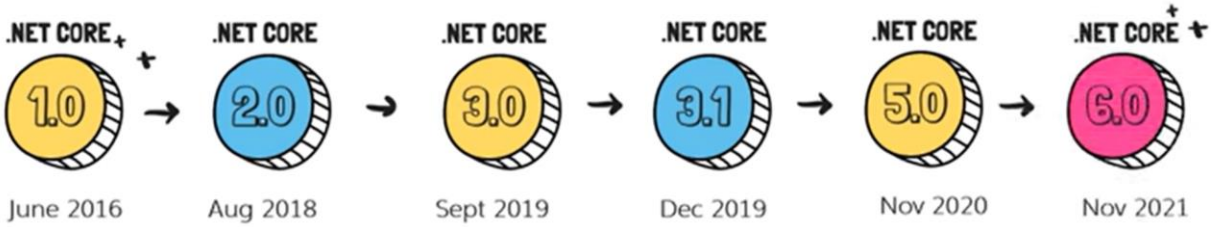
- Mono is the runtime used as part of Xamarin (mobile app development)
- Later in 2016, .NET Core used CoreCLR as the runtime, targeting at supporting cloud applications, including the largest services at Microsoft, and now is also being used for Windows desktop, IoT and machine learning applications.
- Now, .NET Core combines Mono runtimes and Xamarin for complete solutions.

There will be just one .NET going forward, and you will be able to use it to target Windows, Linux, macOS, iOS, Android, tvOS, watchOS and WebAssembly and more.

We will introduce new .NET APIs, runtime capabilities and language features as part of .NET 5.

.NET – A unified platform

From the inception of the .NET Core project, we've added around fifty thousand .NET Framework APIs to the platform. .NET Core 3.0 closes much of the remaining capability gap with .NET Framework 4.8,



Understanding .NET Core

We live in a truly cross-platform world where modern mobile and cloud development have made Windows, as an operating system, much less important. Because of that, Microsoft has been working on an effort to decouple .NET from its close ties with Windows. While rewriting .NET Framework to be truly cross platform, they've taken the opportunity to refactor and remove major parts that are no longer considered core.

This new product was branded .NET Core and includes a cross-platform implementation of the CLR known as CoreCLR and a streamlined library of classes known as CoreFX. .NET Core is fast moving and because it can be deployed side by side with an app, it can change frequently knowing those changes will not affect other .NET Core apps on the same machine. Improvements that Microsoft can make to .NET Core cannot be added to .NET Framework.

In May 2019, the .NET team announced that after .NET Core 3.0 is released in September 2019, .NET Core will be renamed .NET and the major version number will skip the number four to avoid confusion with .NET Framework 4.x. So, the next version of .NET Core will be .NET 5.0. After that, Microsoft plans on annual major version releases every November, rather like Apple does major version number releases of iOS every second week in September.

You can read more about Microsoft's positioning of .NET Core and .NET Framework at the following link: https://devblogs.microsoft.com/dotnet/update□on-net-core-3-0-and-net-framework-4-8/.

What is different about .NET Core?

.NET Core is smaller than the current version of .NET Framework due to the fact that legacy technologies have been removed. For example, Windows Forms and Windows Presentation Foundation (WPF) can be used to build graphical user interface (GUI) applications, but they are tightly bound to the Windows ecosystem, so they have been removed from .NET Core on macOS and Linux. One of the new features of .NET Core 3.0 is support for running old Windows Forms and WPF applications using the Windows Desktop Pack that is included with the Windows version of .NET Core 3.0 which is why it is bigger than the SDKs for macOS and Linux. You can make some small changes to your legacy Windows app if necessary, and then rebuild it for .NET Core to take advantage of new features and performance improvements. The latest technology used to build Windows apps is the Universal Windows Platform (UWP), which is built on a custom version of .NET Core. UWP is not part of .NET Core because it is not cross-platform. ASP.NET Web Forms and Windows Communication Foundation (WCF) are old web application and service technologies that fewer developers are choosing to use for new development projects today, so they have also been removed from .NET Core. Instead, developers prefer to use ASP.NET MVC

and ASP.NET Web API. These two technologies have been refactored and combined into a new product that runs on .NET Core, named ASP.NET Core. It has gained baggage over the years, so the cross-platform API has been slimmed down, will be given support for non-relational databases like Microsoft Azure Cosmos DB, and named Entity Framework Core.

In addition to removing large pieces from .NET Framework in order to make .NET Core, Microsoft has componentized the .NET Core into NuGet packages, those being small chunks of functionality that can be deployed independently. Microsoft's primary goal is not to make .NET Core smaller than .NET Framework. The goal is to componentize .NET Core to support modern technologies and to have fewer dependencies, so that deployment requires only those packages that your application needs.

.NET Standard

The situation with .NET in 2019 is that there are three forked .NET platforms controlled by Microsoft,

1. .NET Core: for cross-platform and new apps.
2. .NET Framework: for legacy apps.
3. Xamarin: for mobile apps.

Each has strengths and weaknesses because they are all designed for different scenarios. This has led to the problem that a developer must learn three platforms, each with annoying quirks and limitations. Because of that, Microsoft defined .NET Standard: a specification for a set of APIs that all .NET platforms can implement to indicate what level of compatibility they have. For example, basic support is indicated by a platform being compliant with .NET Standard 1.4. With .NET Standard 2.0 and later, Microsoft made all three platforms converge on a modern minimum standard, which makes it much easier for developers to share code between any flavor of .NET

We can summarize and compare .NET technologies in 2019, as shown in the following table:

Technology	Description	Host OSes
.NET Core	Modern feature set, full C# 8.0 support, port existing and create new Windows and Web apps and services.	Windows, macOS, Linux
.NET Framework	Legacy feature set, limited C# 8.0 support, maintain existing applications.	Windows only
Xamarin	Mobile apps only.	Android, iOS, macOS

By the end of 2020, Microsoft promises that there will be a single .NET platform instead of three. .NET 5.0 is planned to have a single BCL and two runtimes: one optimized for server or desktop scenarios like websites and Windows desktop apps based on the .NET Core runtime, and one optimized for mobile apps based on the Xamarin runtime.

To keep up to date with .NET, an excellent blog to subscribe to is the official .NET Blog written by the .NET engineering teams, and you can find it at the following link: https://blogs.msdn.microsoft.com/dotnet/

You can use the following links to read more details about the topics we've covered in this chapter:

- **Welcome to .NET Core**: `http://dotnet.github.io`
- **.NET Core Command-Line Interface (CLI) tool**: `https://aka.ms/dotnet-cli-docs`
- **.NET Core runtime, CoreCLR**: `https://github.com/dotnet/coreclr/`
- **.NET Core Roadmap**: `https://github.com/dotnet/core/blob/master/roadmap.md`
- **.NET Standard FAQ**: `https://github.com/dotnet/standard/blob/master/docs/faq.md`
- **Stack Overflow**: `http://stackoverflow.com/`
- **Google Advanced Search**: `http://www.google.com/advanced_search`
- **Microsoft Virtual Academy**: `https://mva.microsoft.com/`
- **Microsoft Channel 9: Developer Videos**: `https://channel9.msdn.com/`

C# grammar/Syntax

The grammar of C# includes statements and blocks. To document your code, you can use comments.

```
/* Multiline comments.

The first console program in C#

*/

using System; // a semicolon indicates the end of a statement namespace Basics

{

        class Program {

                static void Main(string[] args) { // the start of a block

                        Console.WriteLine("Hello World!"); // a statement

                        } // the end of a block

        }

}
```

The C# vocabulary is made up of keywords, symbol characters, and types. Some of the predefined, reserved keywords that you will see in this book include using, namespace, class, static, int, string, double, bool, if, switch, break, while, do, for, and foreach.

Nouns are types, fields, and variables

Verbs are methods

```
// outputs the greeting and a carriage-return

Console.WriteLine("Hello Ram");

// outputs a formatted number and date and a carriage-return

Console.WriteLine( "Temperature on {0:D} is {1}°C.", DateTime.Today, 23.4);
```

Working with variables

- All applications process data. Data comes in, data is processed, and then data goes out.
- Data usually comes into our program from files, databases, or user input, and it can be put temporarily into variables that will be stored in the memory of the running program.
- When the program ends, the data in memory is lost. Data is usually output to files and databases, or to the screen or a printer.
- When using variables, you should think about, firstly, how much space it takes in the memory, and, secondly, how fast it can be processed.
- We control this by picking an appropriate type.
- One can think of simple common types such as int and double as being different-sized storage boxes, where a smaller box would take less memory but may not be as fast at being processed; for example, adding 16-bit numbers might not be processed as fast as adding 64-bit numbers on a 64-bit operating system.

Most of the primitive types except string are value types, which means that they must have a value. You can determine the default value of a type using the default() operator. The string type is a reference type. This means that string variables contain the memory address of a value, not the value itself. A reference type variable can have a null value, which is a literal that indicates that the variable does not reference anything (yet). null is the default for all reference types

```
char userChoice = GetKeystroke(); // assigning from a function
```

Strings

• Literal string: Characters enclosed in double-quote characters. They can use escape characters like \t for tab.

• Verbatim string: A literal string prefixed with @ to disable escape characters so that a backslash is a backslash.

• Interpolated string: A literal string prefixed with $ to enable embedded formatted variables.

string[] names; // can reference any array of strings // allocating memory for four strings in an array names = new string[4];

// storing items at index positions

names[0] = "Kate"; names[1] = "Jack"; names[2] = "Rebecca"; names[3] = "Tom";

Declaring local variables

Local variables are declared inside methods, and they only exist during the execution of that method, and once the method returns, the memory allocated to any local variables is released. Strictly speaking, value types are released while reference types must wait for a garbage collection.

Making a value type nullable

By default, value types like int and DateTime must always have a value, hence their name. Sometimes, for example, when reading values stored in a database that allows empty, missing, or null values, it is convenient to allow a value type to be null, we call this a nullable value type.

The most significant change to the language in C# 8.0 is the introduction of nullable and non-nullable reference types. C# 8.0, reference types can be configured to no longer allow the null value by setting a file- or project-level option to enable this useful new feature. Since this is a big change for C#, Microsoft decided to make the feature opt-in.

To enable the feature at the project level, add the following to your project file:

<PropertyGroup>

<Nullable>enable</Nullable>

</PropertyGroup>

To disable the feature at the file level, add the following to the top of a code file:

#nullable disable

To enable the feature at the file level, add the following to the top of a code file:

#nullable enable

Console App and CLI

Console applications are text-based and are run at the command line. They typically perform simple tasks that need to be scripted, such as compiling a file or encrypting a section of a configuration file. Equally they can also have arguments passed to them to control their behavior.

Getting text input from the user

```
Console.Write("Type your first name and press ENTER: ");

string firstName = Console.ReadLine();

Console.Write("Type your age and press ENTER: ");

string age = Console.ReadLine();

Console.WriteLine(

 $"Hello {firstName}, you look good for {age}.")
```

Passing arguments to main() method from CLI

```
namespace Arguments

{

 class Program

 {

 static void Main(string[] args)

 {

 WriteLine($"There are {args.Length} arguments.");

 }

 }

}
```

Name space (logical grouping of classes)

The System.Console.WriteLine line tells the compiler to look for a method named WriteLine in a type named Console in a namespace named System. To simplify our code, the dotnet new

console command added a statement at the top of the code file to tell the compiler to always look in the System namespace for types that haven't been prefixed with their namespace, as shown in the following code:

using System;

We call this importing the namespace. The effect of importing a namespace is that all available types in that namespace will be available to your program without needing to enter the namespace prefix and will be seen in IntelliSense while you write code.

using System;

using static System.Console;

postfix operator and prefix unary operator

```
int c = 3;
 int d = ++c; // increment c before assigning it
WriteLine($"c is {c}, d is {d}");
```

Pattern matching with the if statement

```
// add and remove the "" to change the behavior
object o = "3";
int j = 4;
if(o is int i)
{
 WriteLine($"{i} x {j} = {i * j}");
}else
{
 WriteLine("o is not an int so it cannot multiply!");
}
```

Rest of the hands-on coding will continue in the Lab Session.