



Common Language Runtime

.NET CORE

Common Language Runtime (CLR) is a managed execution environment that is part of Microsoft's .NET framework. **CLR** manages the execution of programs written in different supported languages. **CLR** transforms source code into a form of bytecode known as **CIL** (**Common Intermediate Language**).

[HTTPS://WWW.TECHOPEDIA.COM/DEFINITION/5225/COMMON-LANGUAGE-RUNTIME-CLR#:~:TEXT=](https://www.techopedia.com/definition/5225/common-language-runtime-clr#:~:text=)

CLR (Common Language Runtime)

<https://docs.microsoft.com/en-us/dotnet/standard/clr>

<https://docs.microsoft.com/en-us/dotnet/framework/get-started/overview>

- The .NET Framework consists of the **Common Language Runtime** (CLR) and the **.NET Framework class library**.
- The **CLR** is the foundation for .NET Framework. It manages and runs the code and provides services like memory management, remoting, type enforcement (through the **CTS**), and security.

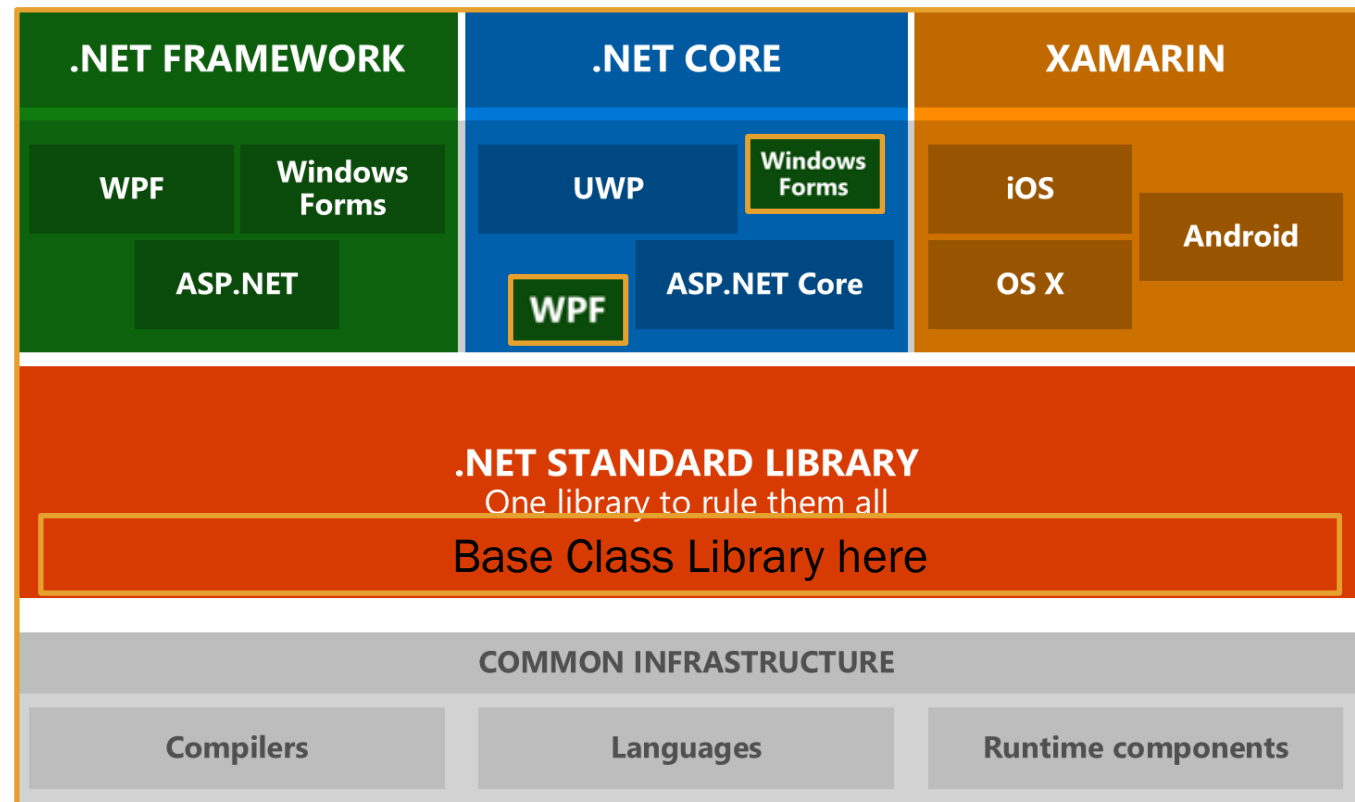
| Benefits of CLR: | | |
|-----------------------------------|--|-----------------------------------|
| cross-language integration | cross-language exception handling | enhanced security |
| versioning and deployment support | a simplified model for component interaction | debugging and profiling services. |

.NET Class Libraries - BCL (Base Class Library)

<https://docs.microsoft.com/en-us/dotnet/standard/clr>

BCL stands for *Base Class Library* (AKA, *Class library (CL)*). A .NET Framework library, **BCL** is the foundation for the C# runtime library and one of the *Common Language Infrastructure (CLI)* standard libraries.

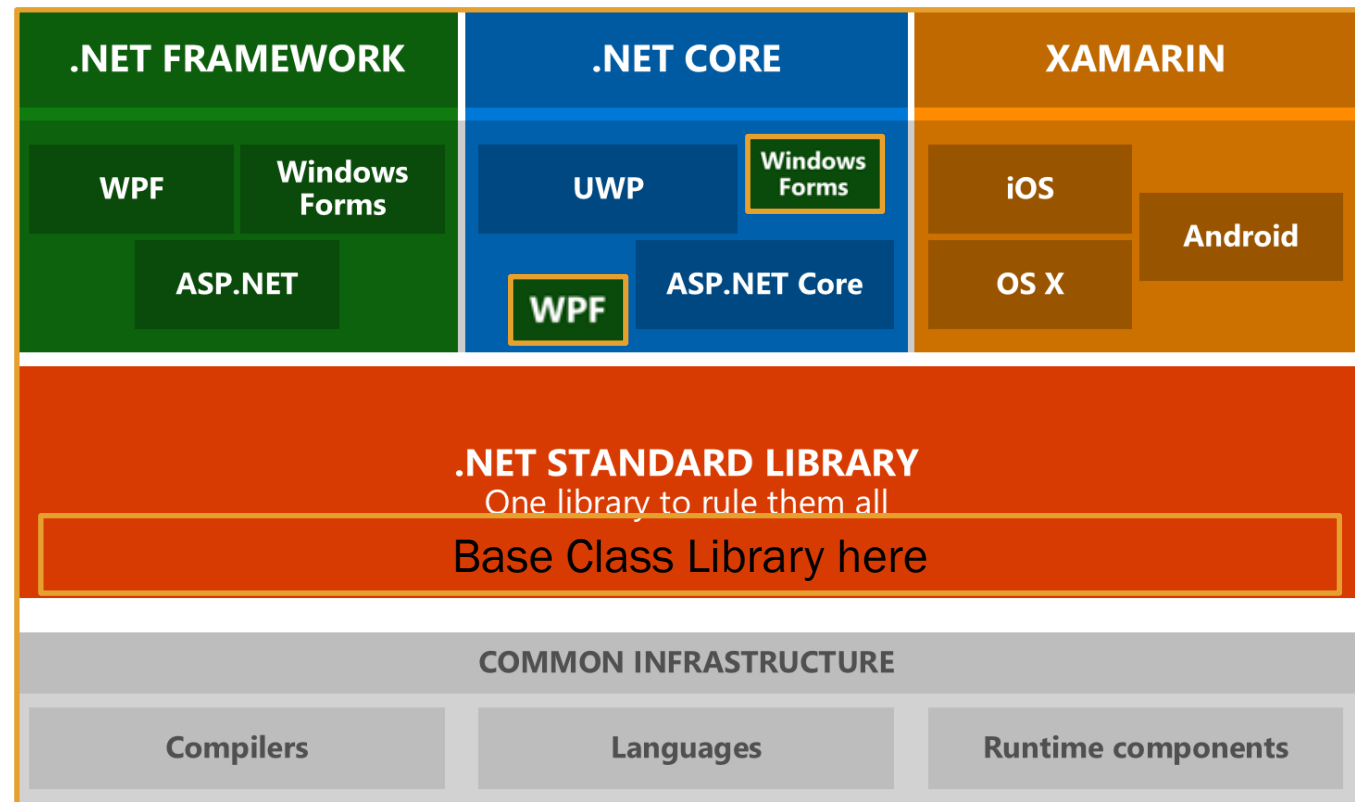
BCL provides the types that represent built-in CLI data types, basic file access, collections, custom attributes, formatting, security attributes, I/O streams, string manipulation, etc.



.NET Class Libraries

<https://docs.microsoft.com/en-us/dotnet/framework/get-started/overview>

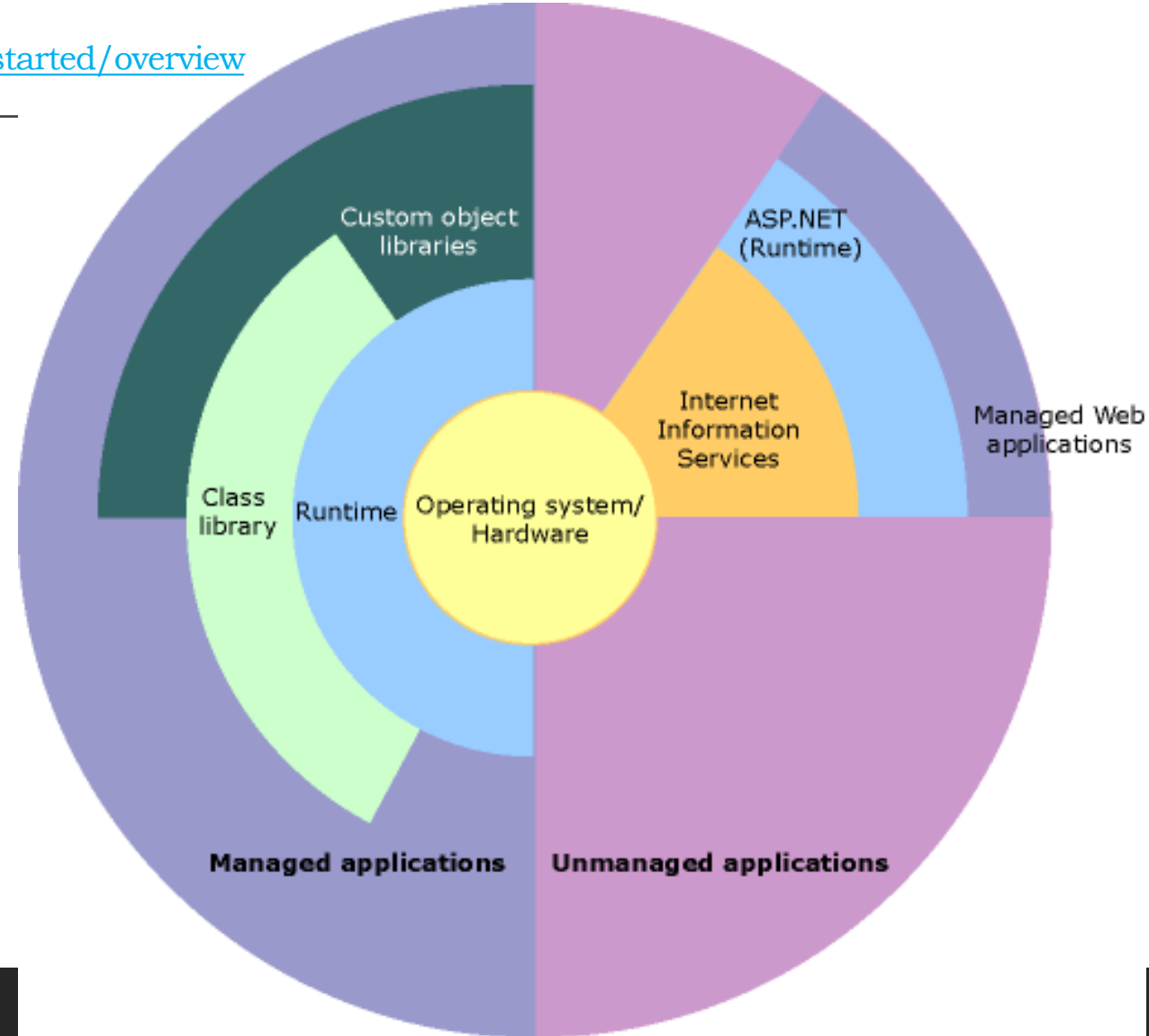
A ***class library*** is an object-oriented collection of reusable types that you can use to develop apps ranging from traditional command-line or graphical user interface (GUI) apps to apps based on the latest innovations provided by ASP.NET, such as XML Web services.



.NET CLR and Class Library Relationship

<https://docs.microsoft.com/en-us/dotnet/framework/get-started/overview>

This illustration shows the relationship of the ***Common Language Runtime*** and the class library to your apps and to the overall system. The illustration also shows how managed code operates within a larger architecture.



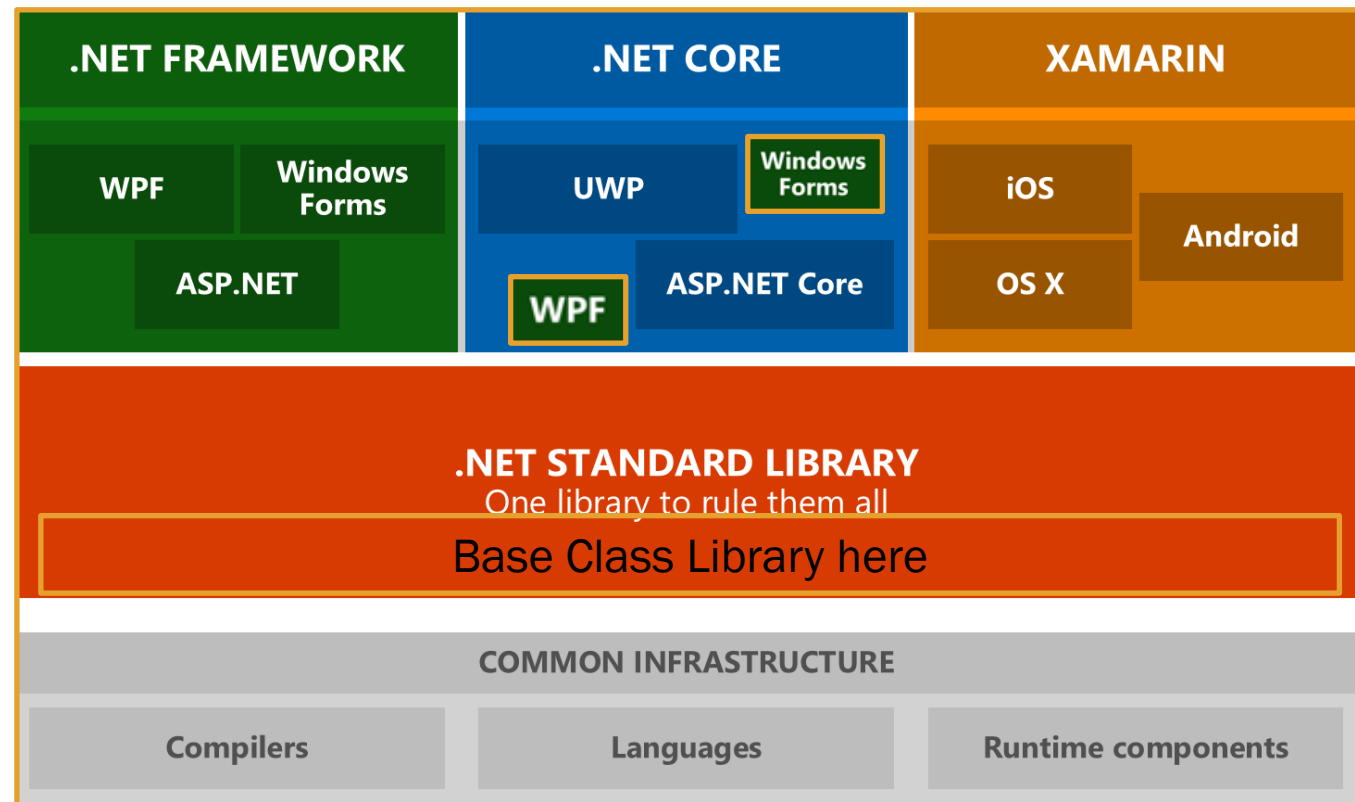
.NET CoreFX

<https://docs.microsoft.com/en-us/dotnet/standard/glossary#corefx>

Think of **CoreFX** as a fork of the .NET Framework **BCL**.

CoreFX is the foundational class library for **.NET Core**. It defines the types for primitives, collections, file systems, console, JSON, XML, async and many others.

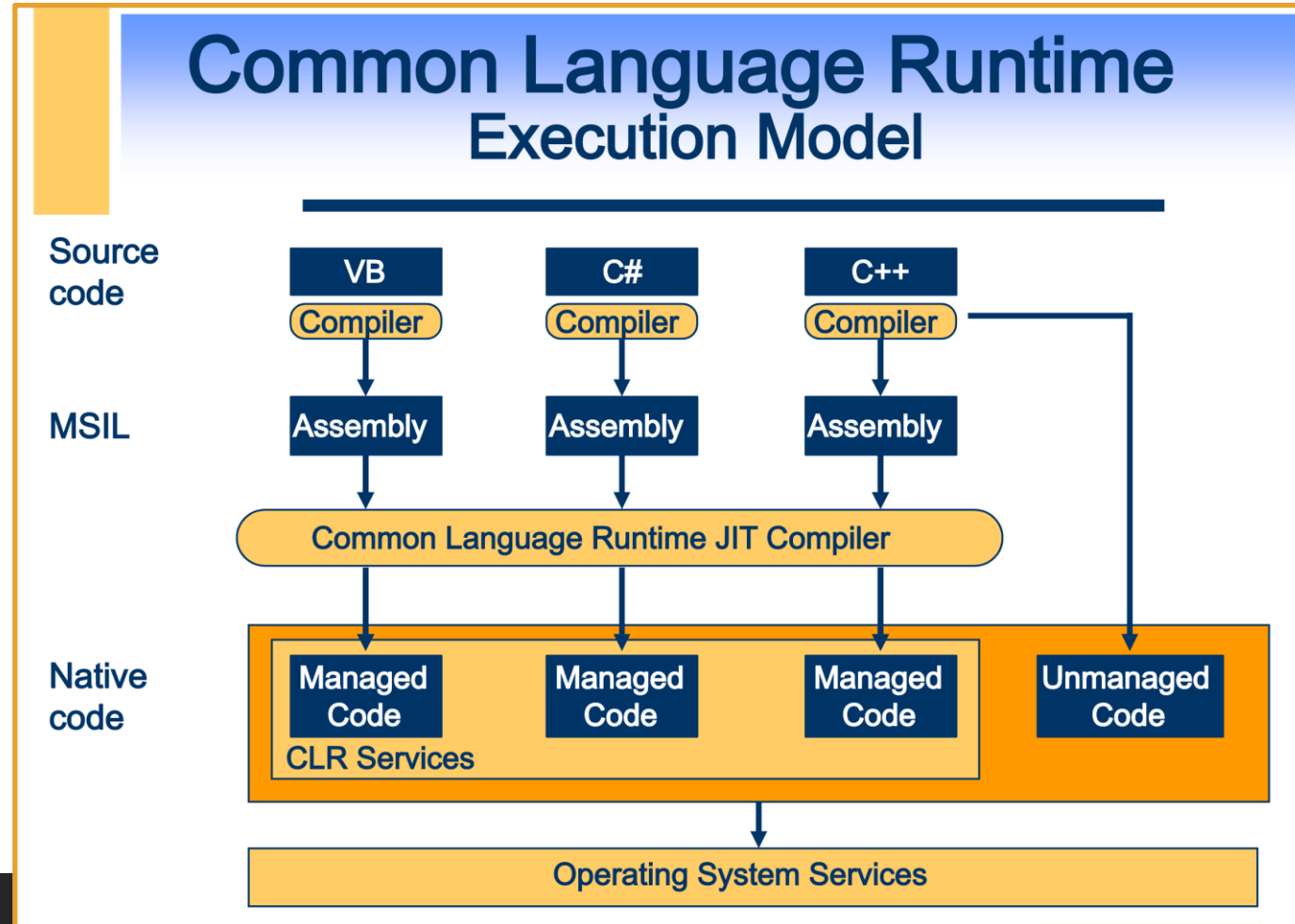
CoreFX makes up the **.NET Core Base Class Library (BCL)**.



Managed Code

<https://docs.microsoft.com/en-us/dotnet/standard/managed-code>

- Managed code is managed by the **Common Language Runtime (CLR)** at runtime.
- The **CLR** knows what your code is doing and can *manage* it.
- The **CLR** provides memory management (**GC**), security boundaries, type safety, etc.
- Managed code is written in a high-level language that can be run on top of .NET.
- Code is compiled into **Intermediate Language** code, which the **CLR** compiles and executes.
- The **CLR** manages the **Just-In-Time** compiling of code from **IL** to machine code that can be run on a **CPU**.



Unmanaged Code

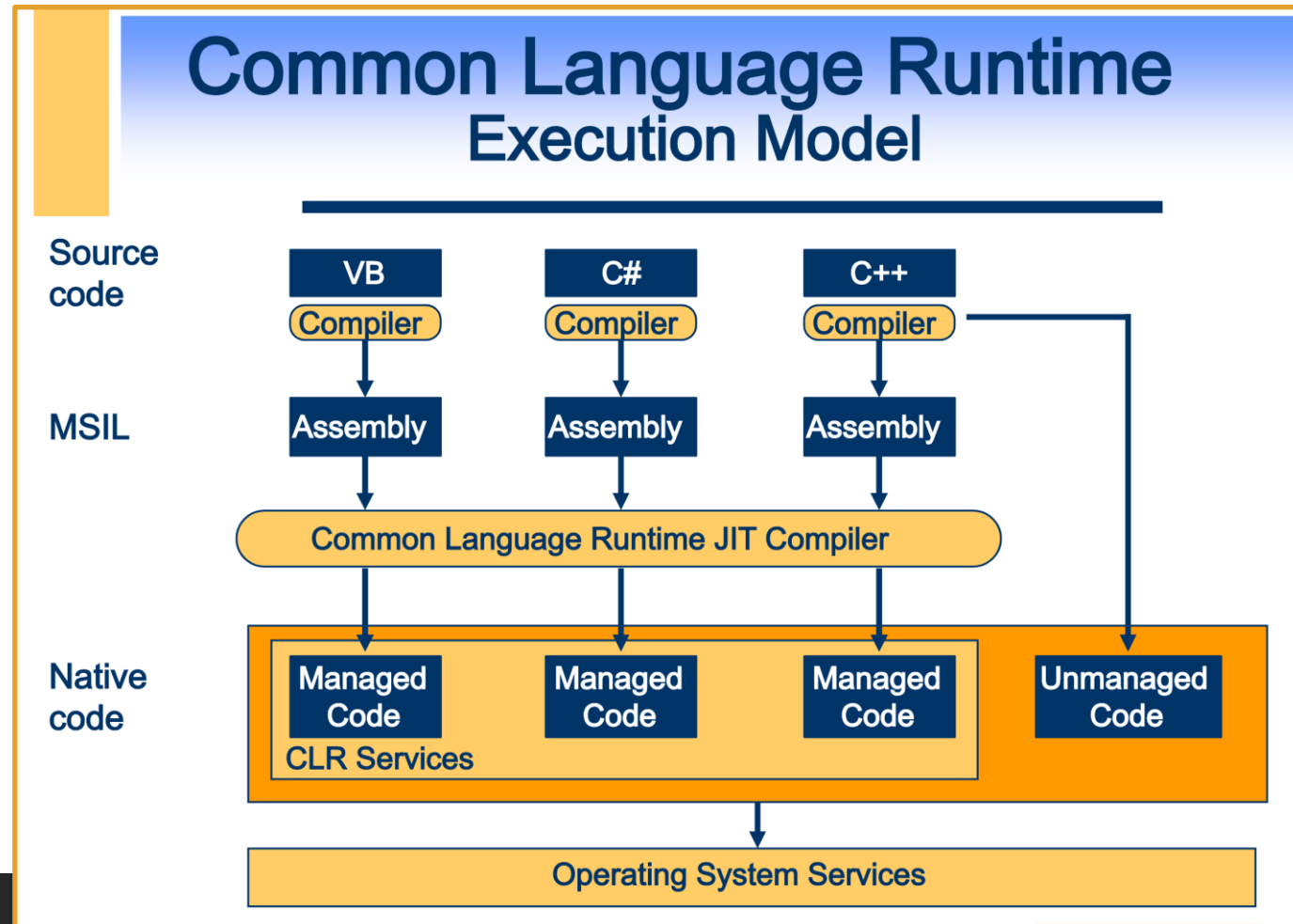
<https://docs.microsoft.com/en-us/dotnet/framework/interop/>

Code that runs outside the **CLR** is called Unmanaged Code.

The .NET Framework promotes interaction with COM components, COM+ services, external type libraries, and many operating system services.

Examples of Unmanaged Code:

- COM components,
- ActiveX interfaces,
- Windows API functions.



IDisposable Interface

<https://docs.microsoft.com/en-us/dotnet/api/system.idisposable?view=netframework-4.8>

- The **Garbage Collector (GC)** has no knowledge of unmanaged resources (open files and streams).
- **IDisposable** provides a method for releasing unmanaged resources.
- To use the **IDisposable** interface, call the object's **IDisposable.Dispose** implementation when finished using it.

```
// A base class that implements IDisposable.  
// By implementing IDisposable, you are announcing that  
// instances of this type allocate scarce resources.  
public class MyResource: IDisposable  
{  
    // Pointer to an external unmanaged resource
```

```
// Dispose managed resources.  
component.Dispose();
```

Using Block

<https://docs.microsoft.com/en-us/dotnet/api/system.idisposable?view=netframework-4.8>

If your language supports a construct such as the using statement in C#, you can use it instead of explicitly calling *IDisposable.Dispose* yourself.

```
public WordCount(string filename)
{
    if (! File.Exists(filename))
        throw new FileNotFoundException("The file does not exist.");

    this.filename = filename;
    string txt = String.Empty;
    using (StreamReader sr = new StreamReader(filename)) {
        txt = sr.ReadToEnd();
    }
    nWords = Regex.Matches(txt, pattern).Count;
}
```

The *using* statement is a syntactic convenience. At compile time, the language compiler implements the intermediate language (IL) for a try/finally block.

Using Block and IDisposable

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/using-statement>

Provides a convenient syntax that ensures the correct use of *IDisposable* objects.

```
using (var font1 = new Font("Arial", 10.0f))  
{  
    byte charset = font1.GdiCharSet;  
}
```

When the lifetime of an *IDisposable* object is limited to a single method, it should be declared and instantiated in a *using* statement. The *using* statement calls the *Dispose* method on the object and causes the object itself to go out of scope as soon as *.Dispose()* is called. Within the *using* block, the object is read-only and cannot be modified or reassigned.

Using Block

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/using-statement>

The *using* statement ensures that *.Dispose* is called even if an exception occurs within the *using* block. You can achieve the same result by putting the object inside a *try* block and then calling *.Dispose* in a finally block.

A *using* block is expanded to a *try/catch* block at compile time (note the extra curly braces to create the limited scope for the object).

```
{
    var font1 = new Font("Arial", 10.0f);
    try
    {
        byte charset = font1.GdiCharSet;
    }
    finally
    {
        if (font1 != null)
            ((IDisposable)font1).Dispose();
    }
}
```

```
using (var font1 = new Font("Arial", 10.0f))
{
    byte charset = font1.GdiCharSet;
}
```