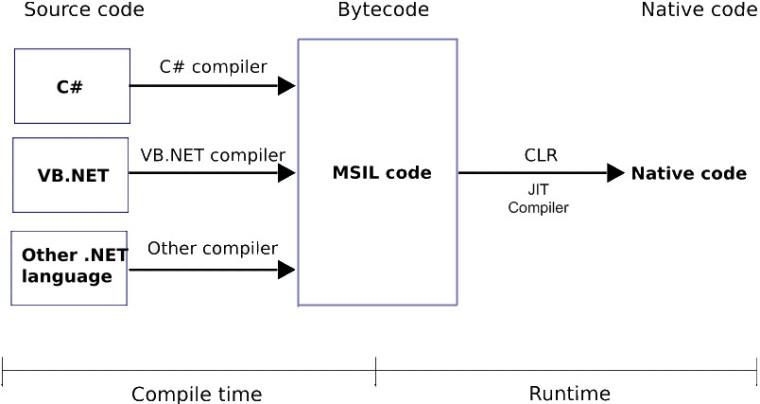
# Compile Time vs Runtime

Làm thế nào .NET có thể chạy được đoạn code C# (hay VB, F#) mà bạn viết ra? Để hiểu được điều này, chúng ta cần nắm rõ quy trình biên soạn code trong .NET.



Trong hình trên, **Compile Time** là quá trình Build, và **Runtime** là quá trình chạy (tức là tính từ khi bạn khởi động ứng dụng của mình).

Về cơ bản, khi bạn thực hiện lệnh build (trong Visual Studio, hay dotnet build bằng dòng lệnh) thì source code của bạn được chuyển hóa thành một dạng ngôn ngữ trung gian có tên là **MSIL** (Microsoft Intermediate Language). Khi ứng dụng được khởi chạy, thành phần Runtime - hay tên gọi riêng biệt trong .NET là **CLR** (Common Language Runtime) sẽ tiến hành dịch mã MSIL thành **Native code** (mã máy) để cho máy tính có thể thực thi. Quá trình này gọi là **JIT** (Just-In-Time) compilation.

Để hiểu rõ hơn về JIT, chúng ta hãy thử một thí nghiệm. Copy đoạn code sau vào file *Program.cs*trong project của bạn:

using System;

using System.Diagnostics;

namespace ConsoleApp

{

public class Program

{

public static void DoSomeCalculation()

{

string input="0";

for(int i=0; i<10; i++)

{

var converted = Convert.ToInt32(input);

converted++;

input = converted.ToString();

}

}

public static void Main(string[] args)

{

Stopwatch sw = new Stopwatch();

//first time

sw.Start();

DoSomeCalculation();

sw.Stop();

Console.WriteLine($"First calculation: {sw.ElapsedTicks} ticks");

//second time

sw.Restart();

DoSomeCalculation();

sw.Stop();

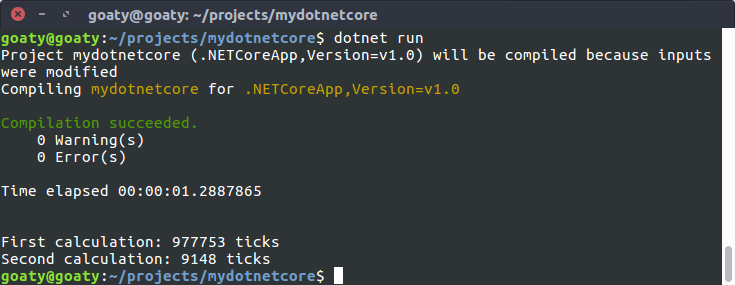
Console.WriteLine($"Second calculation: {sw.ElapsedTicks} ticks");

}

}

}

Chúng ta sẽ tiến hành gọi cùng 1 hàm 2 lần và đo xem thời gian chạy của 2 lần gọi là bao nhiêu. (10000 ticks = 1 ms). Tiến hành chạy dotnet run và xem kết quả.



Bạn có thể thấy rằng lần chạy đầu tiên lâu hơn lần thứ 2 rất nhiều. Tại sao vậy? Đó là do khi DoSomeCalculation() được gọi lần đầu, CLR tiến hành biên soạn (JIT) hàm này thành ngôn ngữ máy. Ở những lần chạy kế tiếp, DoSomeCalculation() không cần JIT lại (do đã được JIT xong), nên thời gian thực thi nhanh hơn nhiều. Điều này cũng lý giải tại sao trong một số phần mềm, thời gian thực hiện một chức năng nào đó lần đầu tiên thường chậm hơn so với các lần kế tiếp.

Tại sao lại cần có ngôn ngữ trung gian? Ngôn ngữ trung gian trong .NET khá gần với mã máy nhưng không chứa thông tin cụ thể về CPU. Việc giúp cho đoạn code trung gian của bạn có thể hoạt động trên nhiều loại CPU (64bit, 32bit), cũng như nhiều loại kiến trúc khác nhau (ARM, Intel…)

Trên thực tế một vài ngôn ngữ (JavaScript, Python…) không sử dụng đến ngôn ngữ trung gian: Source sẽ được dịch thẳng ra mã máy tại Runtime. Điểm lợi của việc này là quá trình build được đơn giản hóa, tuy nhiên hiệu năng sẽ bị hạn chế.

Ngoài việc biên dịch, môi trường hoạt động (Runtime) còn có những công dụng như:

* Tự động quản lý bộ nhớ. Khi làm việc với những ngôn ngữ bậc cao như C# hay Java, bạn không cần  giải phóng bộ nhớ bằng cách gọi free() như khi làm việc với C/C++. CLR bao gồm một công cụ dọn rác (**Garbage Collector** - GC) sẽ tự động giải phóng những phần bộ nhớ không được sử dụng
* Strong typings: CLR quản lý thông tin về các kiểu dữ liệu mà bạn sử dụng. Điều này giúp cho bạn có thể phân biệt được các định dạng thông tin của từng biến khác nhau (class, structure…)

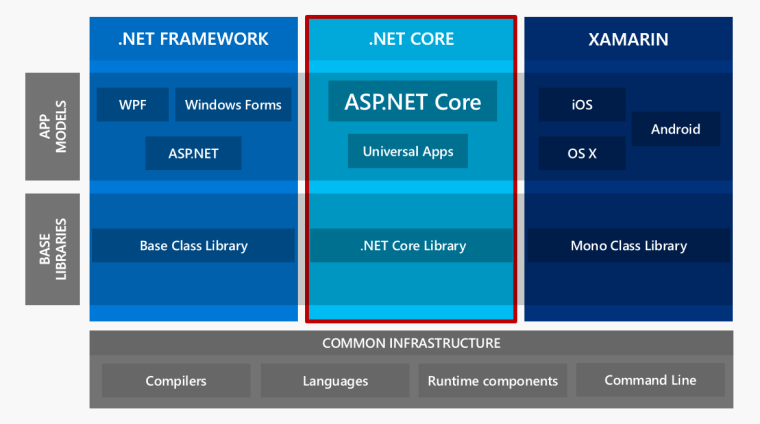
# .NET Framework, .NET Core, .NET Mono

Tại sao cần phải phân biệt chúng? Vì bạn cần phải hiểu rõ mình đang làm gì. VD: Nếu bạn có ý định chạy một Web Server trên Linux thì tuyệt đối không nên sử dụng Mono.

Về cơ bản, .NET Framework, .NET core và .NET Mono là ba phiên bản .NET khác nhau (có nghĩa là mỗi phiên bản có Runtime, Libraries và Toolings riêng).

Vậy tại sao lại có đến 3 phiên bản khác nhau?

* **.NET Framework** được Microsoft đưa ra chính thức từ năm 2002. .NET Framework chỉ hoạt động trên Windows. Những nền tảng ứng dụng như Winforms, WPF, ASP.NET (1-4) hoạt động dựa trên nó.
* **.NET Mono** là phiên bản cộng đồng nhằm mang .NET đến những nền tảng ngoài Windows. Nó được phát triển chủ yếu nhằm xây dựng những ứng dụng với giao diện người dùng và được sử dụng rất rộng rãi: Unity Game, Xamarin…
* Cho đến năm 2013, Microsoft định hướng đi đa nền tảng (Windows, Linux và cả MacOS) và phát triển **.NET Core**. Nó hiện được sử dụng trong các ứng dụng Universal Windows Platform (UWP) và ASP.NET Core.



**Vậy nên sử dụng .NET Framework, .NET Core hay .NET Mono?**

Điều đó tùy thuộc vào ứng dụng mà bạn có ý định phát triển. Đối với các ứng dụng Windows Desktop, .NET Framework là sự lựa chọn tốt nhất. Nếu bạn phát triển game dựa trên Unity, hay những ứng dụng di động với Xamarin, bạn sẽ sử dụng .NET Mono. Đối với các Web Server, bạn có thể sử dụng cả .NET Framework và .NET Core.

Tuyệt đối không nên dùng Mono để vận hành Web Server. Bộ máy dọn rác của Mono không được thiết kế để hoạt động với webserver và sẽ gây ra quá tải nhanh chóng.

Vậy nên lựa chọn .NET Framework hay .NET Core cho các Web Server? .NET Core chạy được đa nền tảng và có hiệu năng cao hơn. Nhược điểm duy nhất của nó là số lượng thư viện hỗ trợ vẫn còn hạn chế. .NET Framework có hệ sinh thái lớn hơn với nhiều các thư viện hỗ trợ hơn.

# Numerics

.NET Core supports the standard numeric integral and floating-point primitives. It also supports the following types:

## Integral Types

.NET Core supports both signed and unsigned integers of different ranges from 1 to 8 bytes in length.

The following table represents the integral types and their size:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Type** | **Signed/ Unsigned** | **Size (bytes)** | **Minimum Value** | **Maximum Value** |
| Byte | Unsigned | 1 | 0 | 255 |
| Int16 | Signed | 2 | −32,768 | 32,767 |
| Int32 | Signed | 4 | −2,147,483,648 | 2,147,483,647 |
| Int64 | Signed | 8 | −9,223,372,036,854,775,808 | 9,223,372,036,854,775,807 |
| SByte | Signed | 1 | -128 | 127 |
| UInt16 | Unsigned | 2 | 0 | 65,535 |
| UInt32 | Unsigned | 4 | 0 | 4,294,967,295 |
| UInt64 | Unsigned | 8 | 0 | 18,446,744,073,709,551,615 |

Each integral type supports a standard set of arithmetic, comparison, equality, explicit conversion, and implicit conversion operators.

You can also work with the individual bits in an integer value by using the System.BitConverter class.

## Floating-Point Types

.NET Core includes three primitive floating-point types, which are shown in the following table:

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Size (bytes)** | **Minimum Value** | **Maximum Value** |
| Double | 8 | −1.79769313486232e308 | 1.79769313486232e308 |
| Single | 4 | −3.402823e38 | 3.402823e38 |
| Decimal | 16 | −79,228,162,514,264,337,593,5 43,950,335 | 79,228,162,514,264,337,593,543,9 50,335 |

* Each floating-point type supports a standard set of arithmetic, comparison, equality, explicit conversion, and implicit conversion operators.
* You can also work with the individual bits in Double and Single values by using the BitConverter class.
* The Decimal structure has its own methods, Decimal.GetBits and Decimal.Decimal (Int32()), for working with a decimal value's individual bits, as well as its own set of methods for performing some additional mathematical operations.

## Big Integer

* System.Numerics.BigInteger is an immutable type that represents an arbitrarily large integer whose value in theory has no upper or lower bounds.
* The methods of the BigInteger type is closely parallel to those of the other integral types.

## Complex

* The System.Numerics.Complex type represents a complex number, i.e., a number with a real number part and an imaginary number part
* It supports a standard set of arithmetic, comparison, equality, explicit conversion, and implicit conversion operators, as well as mathematical, algebraic, and trigonometric methods.

## SIMD

* The Numerics namespace includes a set of SIMD-enabled vector types for .NET Core.
* SIMD allows some operations to be parallelized at the hardware level, which results in huge performance improvements in mathematical, scientific, and graphics apps that perform computations over vectors.
* The SIMD-enabled vector types in .NET Core include the following:
  + System.Numerics.Vector2, System.Numerics.Vector3, and System.Numerics.Vector4 types, which are 2-, 3-, and 4-dimensional vectors of type Single.
  + The Vector <T> structure that allows you to create a vector of any primitive numeric type. The primitive numeric types include all numeric types in the System namespace except for Decimal.
  + Two matrix types, System.Numerics.Matrix3×2, which represents a 3×2 matrix; and System.Numerics.Matrix4×4, which represents a 4×4 matrix.
  + The System.Numerics.Plane type, which represents a three-dimensional plane, and the System.Numerics.Quaternion type, which represents a vector that is used to encode three-dimensional physical rotations.