



Module "C#"

Submodule "C# Essentials"

Part 1

AGENDA

- 1 .NET Ecosystem (.NET Framework and .NET Core)
- 2 CLR. .NET model execution and JIT compiler. Managed code.
- 3 .NET versions overview
- 4 C# Versions overview (1.0 - 8.0)
- 5 CTS, value and reference types. Boxing/unboxing
- 6 C# operator basis

.NET Ecosystem



.NET Ecosystem

Portable Class Libraries

.NET Framework

.NET Core

.NET

February 13, 2002

Roslyn Compiler













.NET Standard

Base Class Library

.NET Ecosystem

Create a new project

Recent project templates

-  ASP.NET Web Application (.NET Framework) C# 
-  Console App (.NET Framework) C# 
-  Class Library (.NET Framework) C# 
-  Windows Forms App (.NET Framework) C# 
-  Empty Project (.NET Framework) C# 
-  Console App (.NET Core) C#
-  Class Library (.NET Standard) C#

Search for templates (Alt+S)



[Clear all](#)

C#

Windows

Console



Console App (.NET Core)

A project for creating a command-line application that can run on .NET Core on Windows, Linux and MacOS.

Console

C#

Linux

macOS

Windows



Console App (.NET Framework)

A project for creating a command-line application

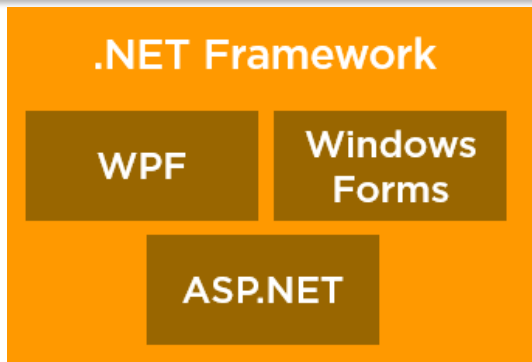
Console

C#

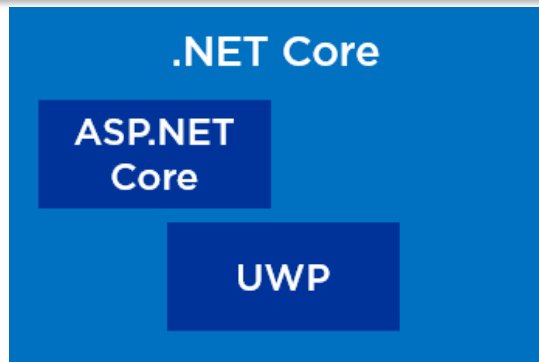
Windows

Not finding what you're looking for?
[Install more tools and features](#)

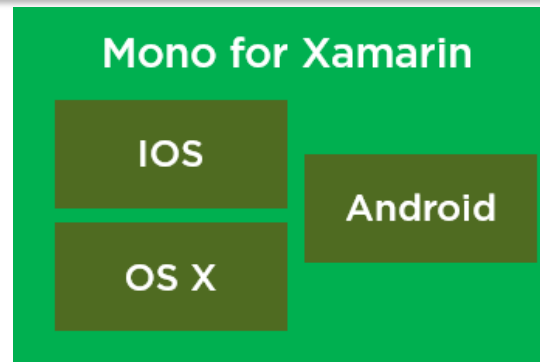
The components of the .NET Ecosystem



- Windows centric
- Windows specific APIs

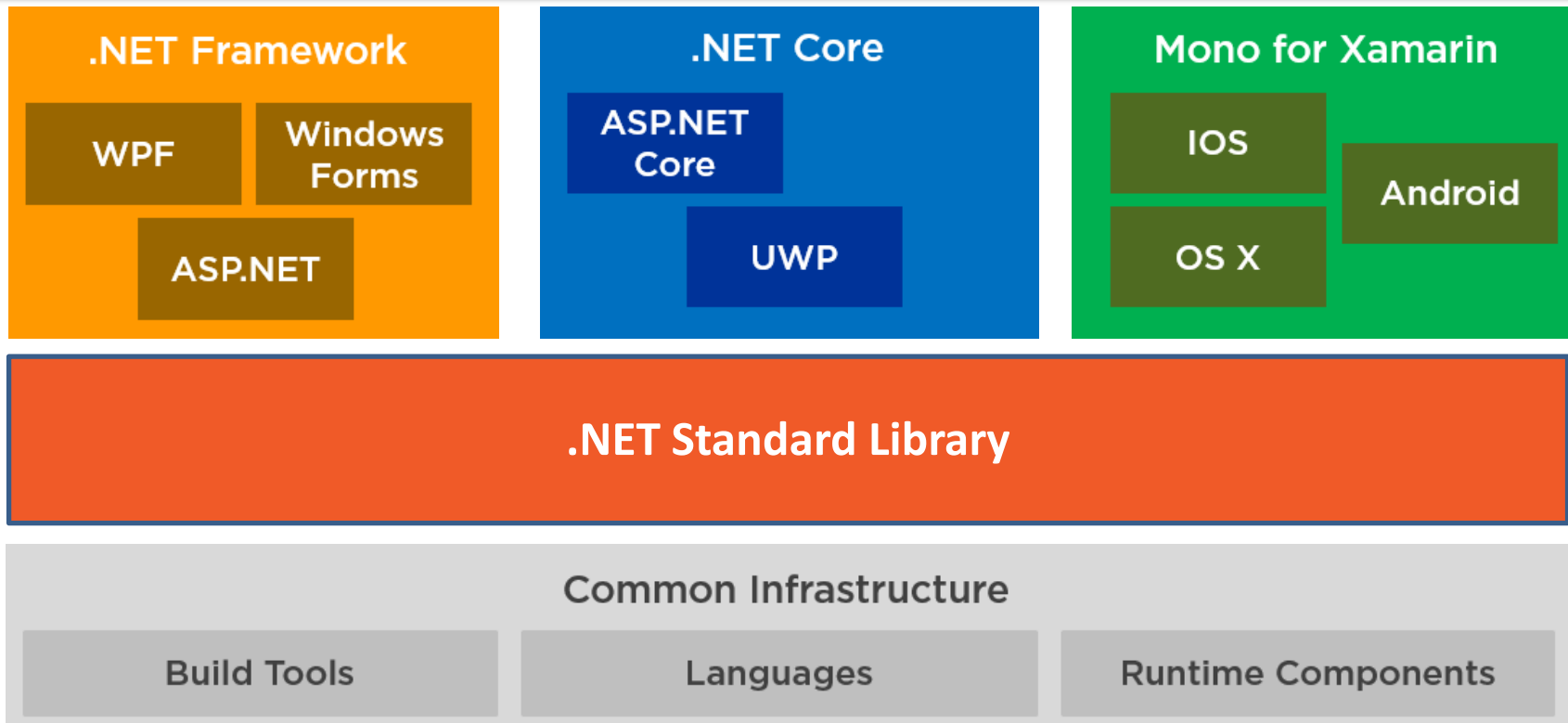


- Cross-platform
- Run side-by-side
- Performance



- Cross-platform
- Specific APIs for
 - IOS
 - Android
 - Xamarin.Mac

The components of the .NET Ecosystem

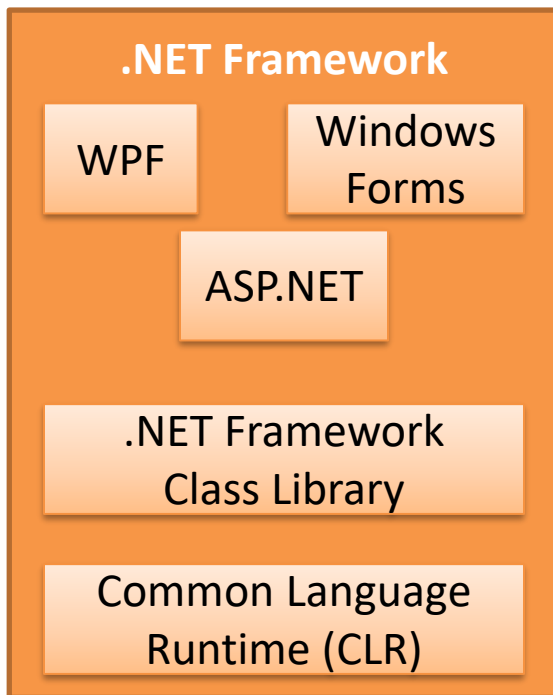


Runtimes in .NET Ecosystem

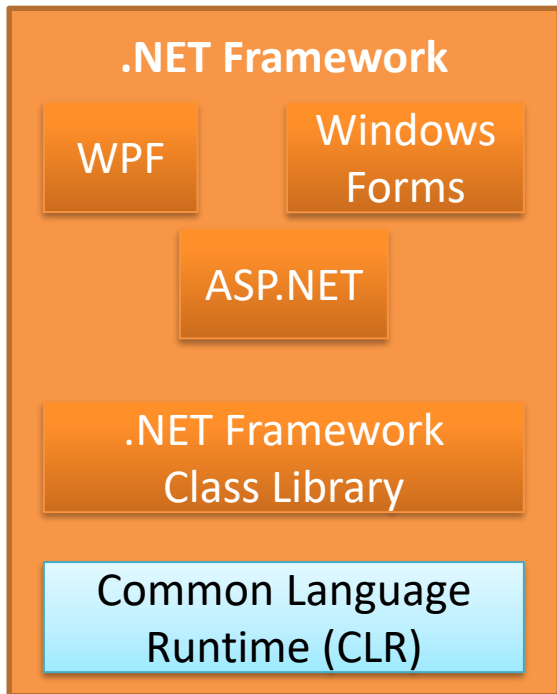
- .NET Framework
- .NET Core
 - UWP
- Mono for Xamarin

.NET Framework

<https://github.com/microsoft/referencesource>

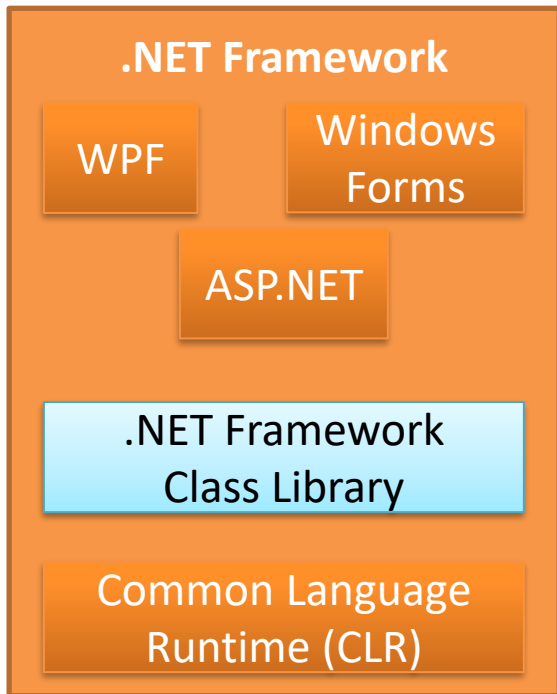


.NET Framework: CLR



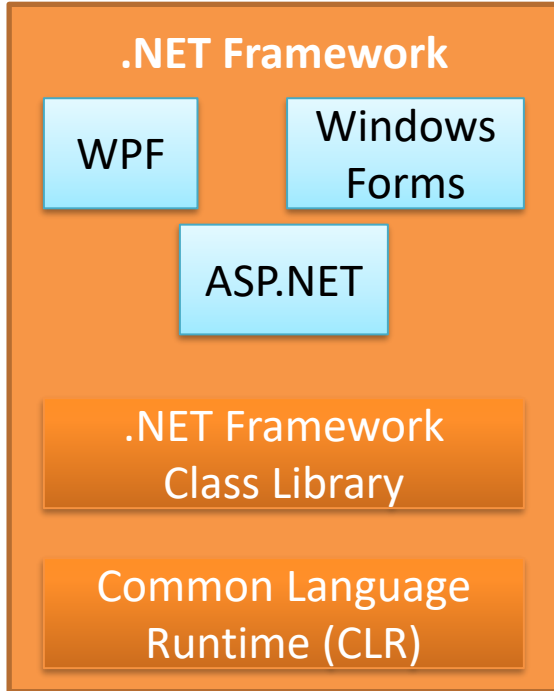
- Run code, Garbage Collection (GC)
- C#, Visual Basic .NET, F#

.NET Framework: Class Library



- Classes, interfaces and value types that provide capabilities

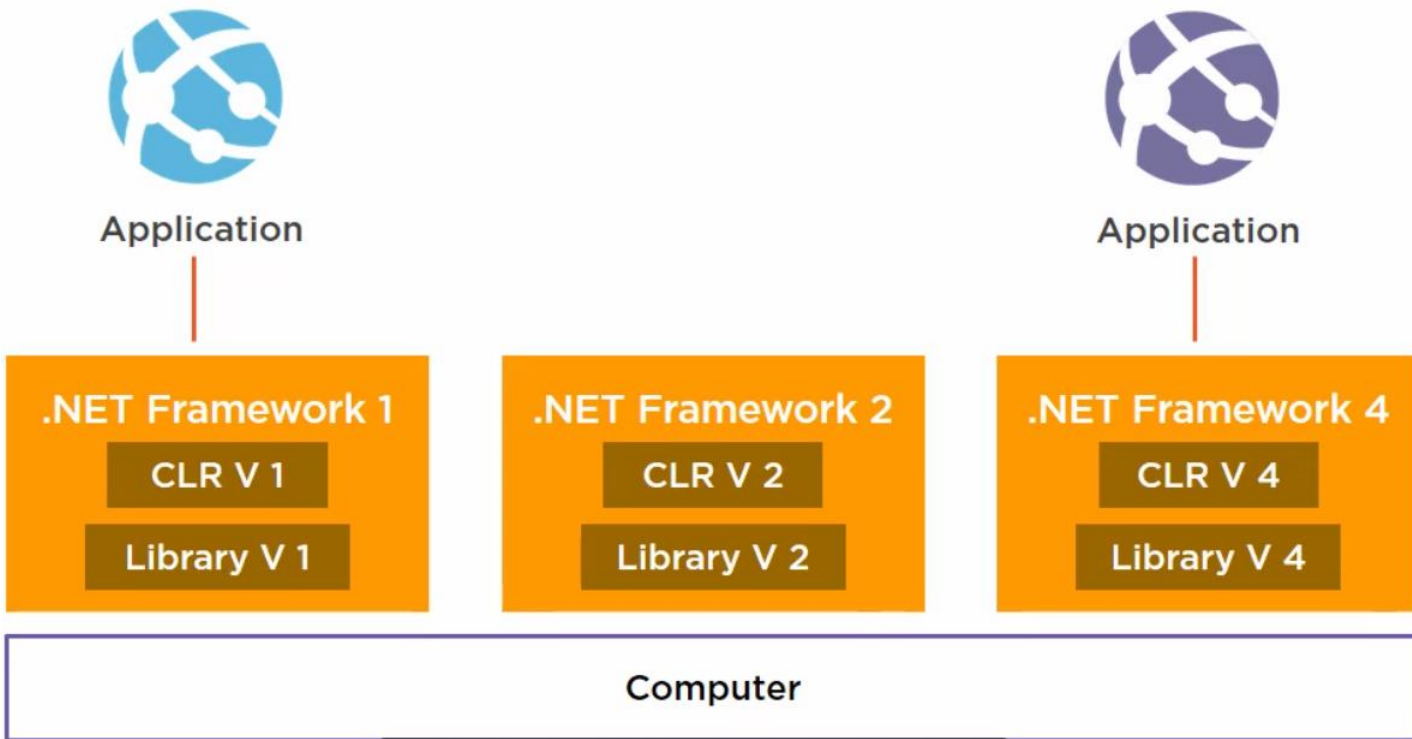
.NET Framework: Workloads



Workloads (application types):

- Console application
- Windows Communication Foundation (WCF)
- Windows Workflow Foundation (WF)
- Windows Presentation Foundation (WPF)
- Windows Forms
- ASP.NET
 - ASP.NET Forms, ASP.NET MVC, ASP.NET Web API
- Microsoft Azure (WebJobs, Cloud Services)
- ...

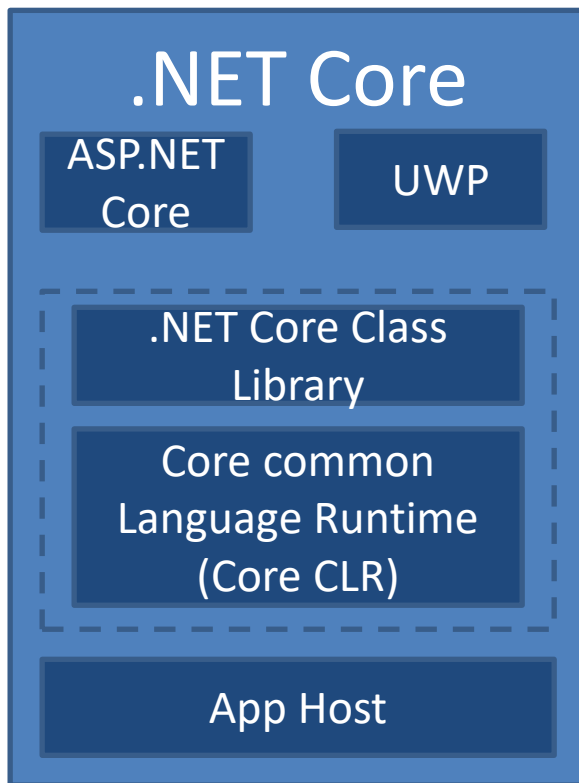
.NET Framework: different version side-by-side



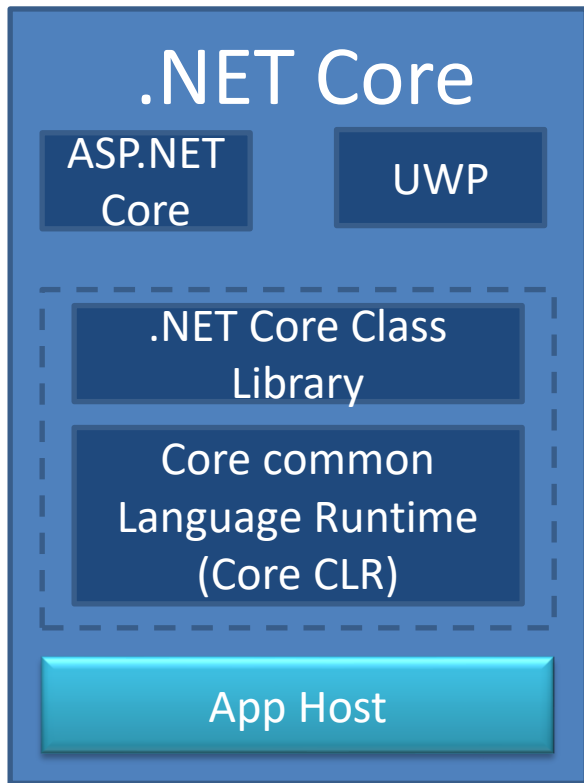
DEMO

.NET Core

- Released in 2016
- Open source:
<https://github.com/dotnet/core>

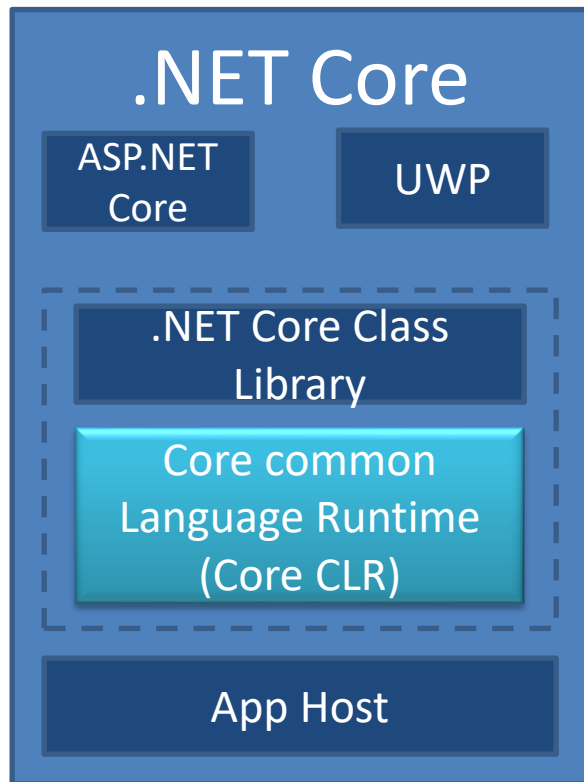


.NET Core components



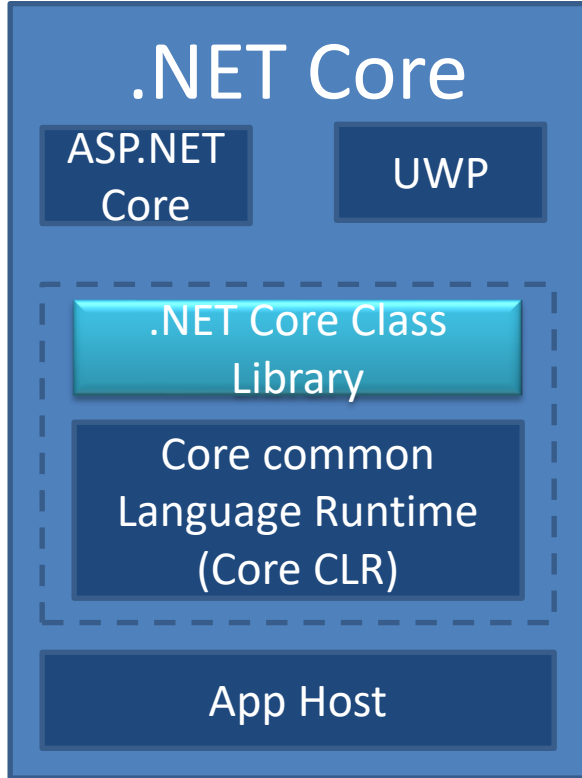
- Host the CoreCLR and launches the app

.NET Core components



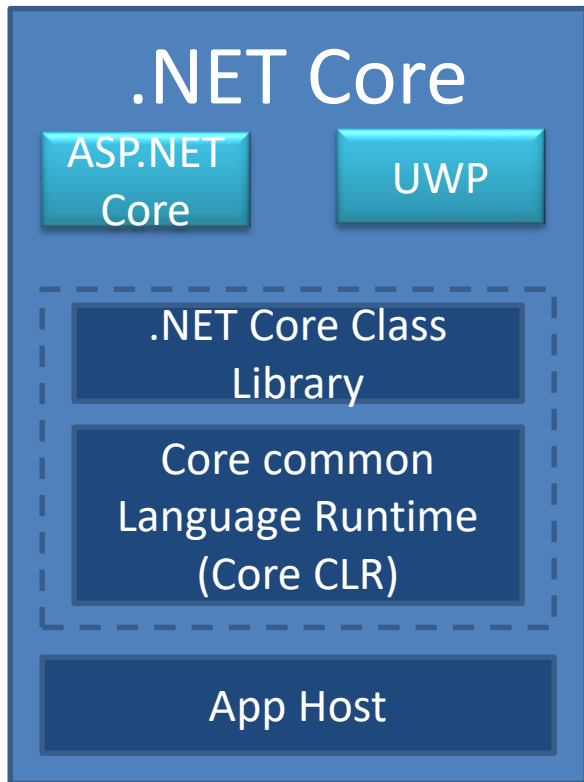
- Assembly loading, Garbage Collection
- C#, VB.NET, F#

.NET Core components



- Classes that provide capabilities
- Subset of the .NET Framework library

.NET Core components



Workloads (application types):

- Console application
- ASP.NET Core
 - MVC
 - API
- Universal Windows Platform Apps
- ...



Windows Client

Windows IoT

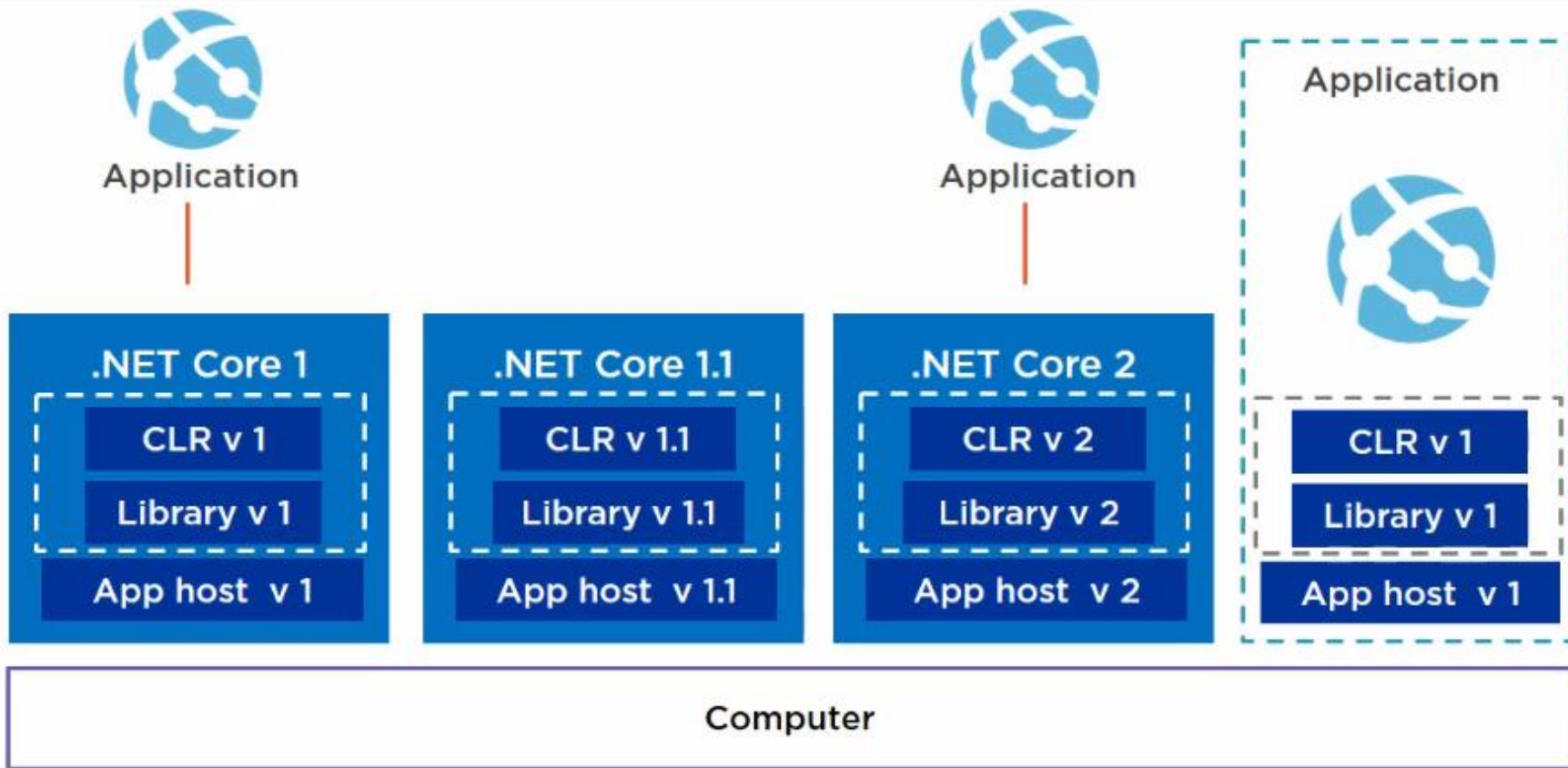
Linux

Windows Server

Max OS X

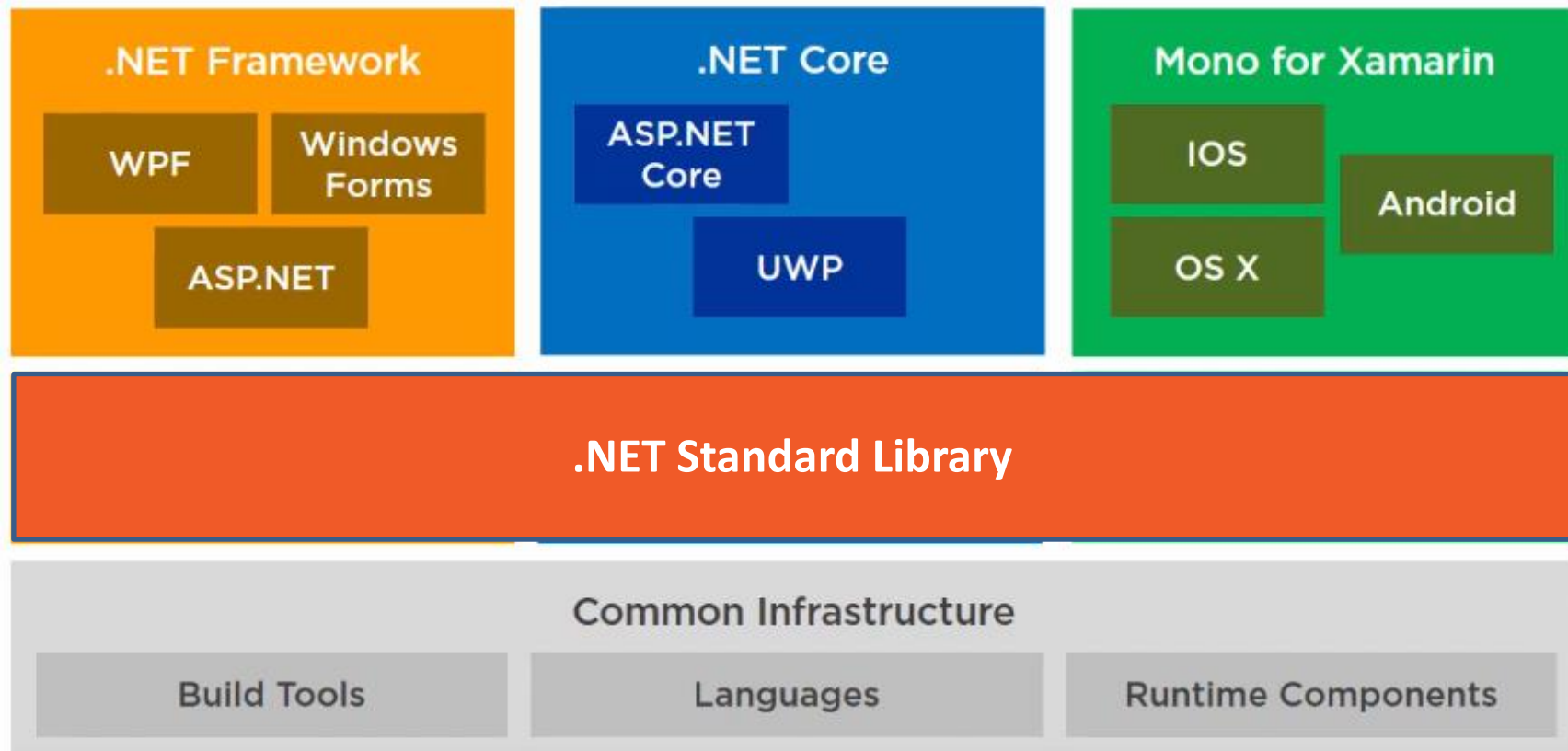
FreeBSD

.NET Core



DEMO

.NET Standard Library



.NET Standard Library

- Not something that you install
- Formal specification of .NET APIs
- Evolution of Portable Class Library (PCL)
- Runtimes implement .NET Standard
- Runtime version implement .NET standard version
 - .NET Framework 4.5 implements .NET standard ≤ 1.1

.NET Standard vs Portable Class Libraries

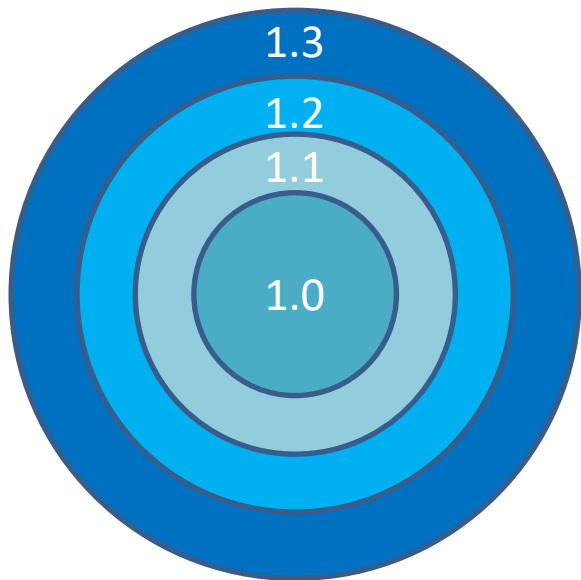
.NET Standard

- Curated set of APIs
- Is platform agnostic

Portable Class Libraries (PCL)

- APIs are defined by the platforms you target
- Targets limited amount of platforms

.NET Standard versioning

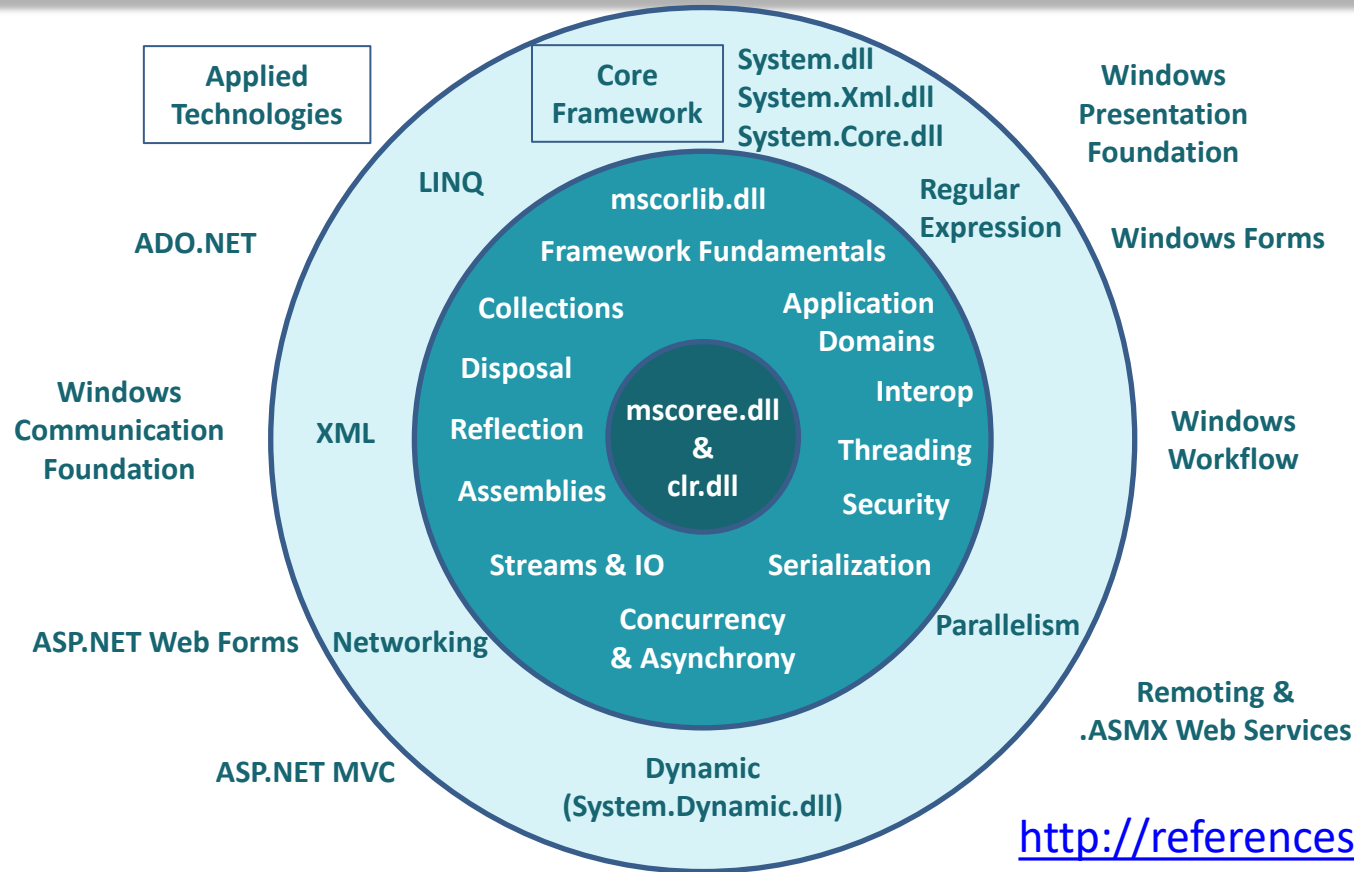


- Each version contains the APIs of the previous versions
- No breaking changes between version
- Once shipped, versions are frozen
- Specific .NET runtime version implement specific .NET Standard version

.NET API browser: <https://docs.microsoft.com/en-us/dotnet/api/>

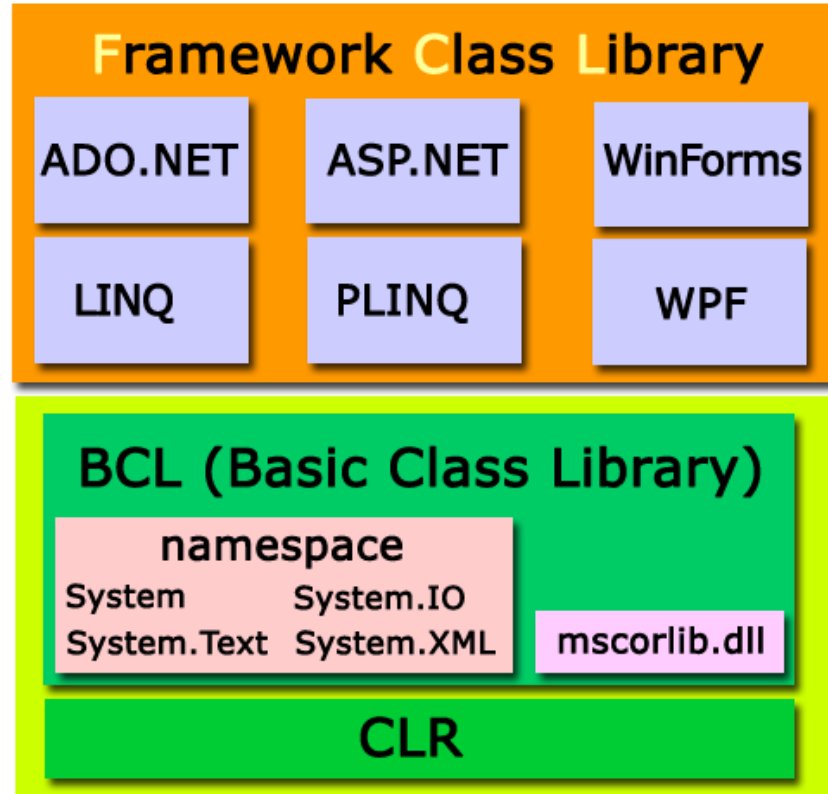
CLR. .NET model execution and JIT compiler

.NET Framework = CLR + Libraries (BCL, FCL)



<http://referencesource.microsoft.com/>

.NET Framework Platform = CLR + Libraries (BCL, FCL)



.NET Framework. CLR vs CLI

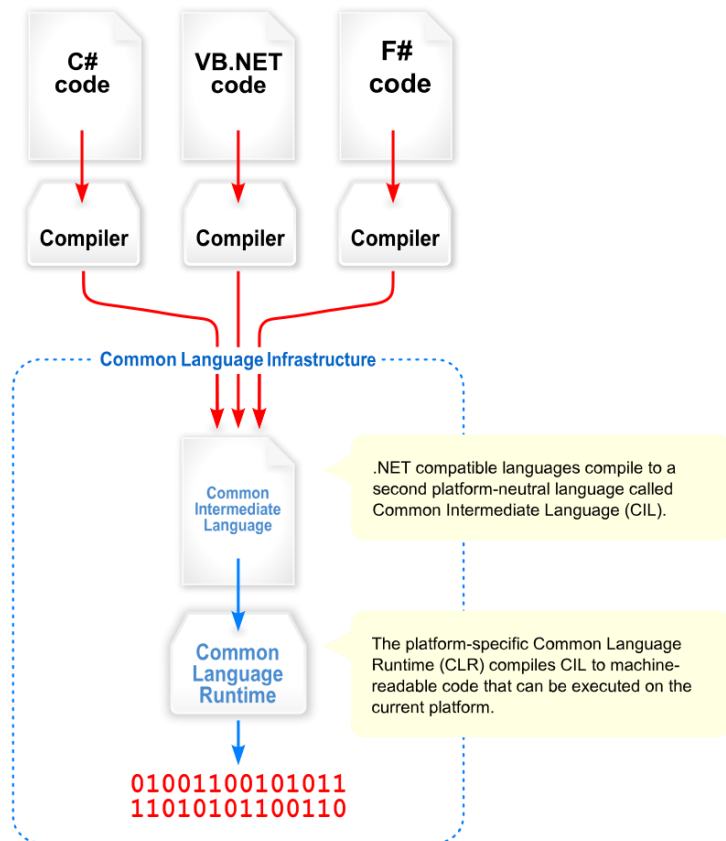
The **Common Language Infrastructure (CLI)** is an open specification (technical standard) developed by Microsoft and standardized by ISO and Ecma that describes executable code and a runtime environment that allows multiple high-level languages to be used on different computer platforms without being rewritten for specific architectures. This implies it is platform agnostic. The .NET Framework, .NET Core and Mono are implementations of the CLI.

ECMA-335: <https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-335.pdf>

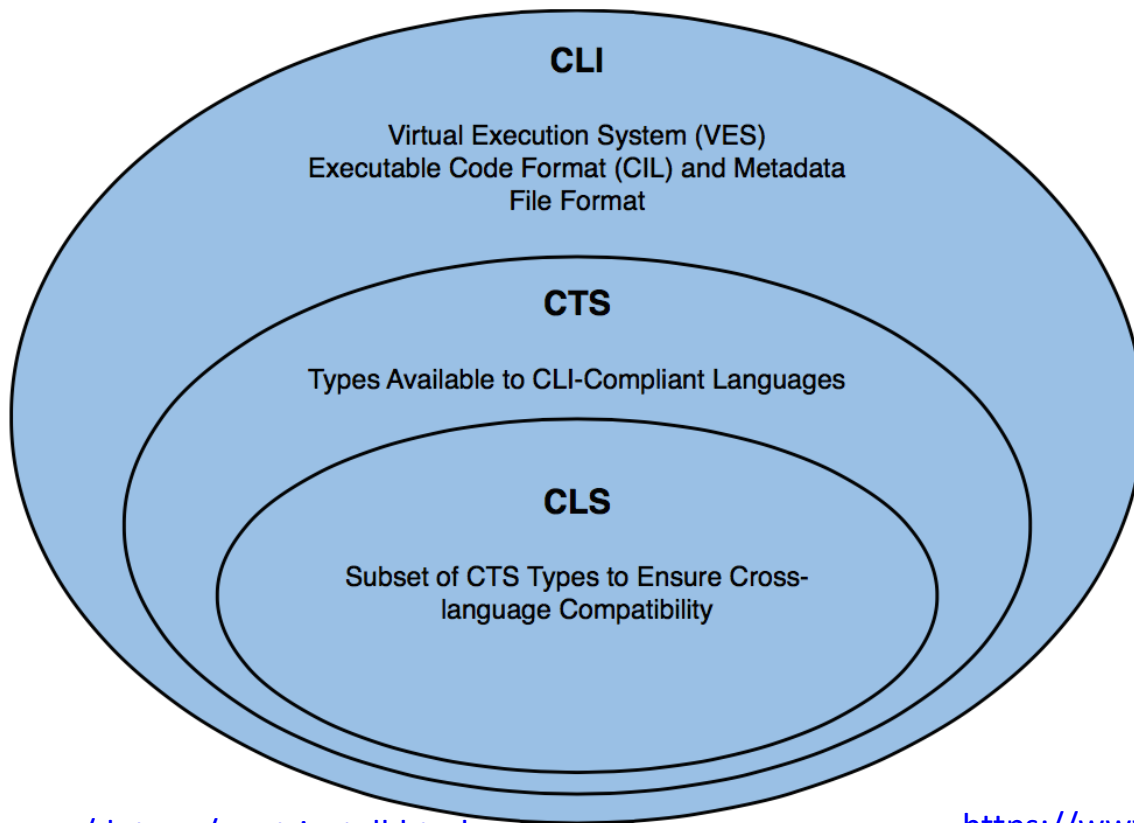
Among other things, the CLI specification describes the following four aspects:

- **The Common Type System (CTS)** - a set of data types and operations that are shared by all CTS-compliant programming languages.
- **The Metadata** - information about program structure is language-agnostic, so that it can be referenced between languages and tools, making it easy to work with code written in a language the developer is not using.
- **The Common Language Specification (CLS)** - a set of base rules to which any language targeting the CLI should conform in order to interoperate with other CLS-compliant languages. The CLS rules define a subset of the Common Type System.
- **The Virtual Execution System (VES)** - the VES loads and executes CLI-compatible programs, using the metadata to combine separately generated pieces of code at runtime

Visual overview of the Common Language Infrastructure



CLR vs CLI



<https://www.gnu.org/software/dotgnu/pnet-install.html>

<https://www.mono-project.com/>

Common Language Runtime (CLR)

The **Common Language Runtime (CLR)**, the virtual machine component of Microsoft .NET framework, manages the execution of .NET programs. **Just-in-time compilation** converts the managed code (compiled intermediate language code), into machine instructions which are then executed on the CPU of the computer. The CLR provides additional services including memory management, type safety, exception handling, garbage collection, security and thread management. All programs written for the .NET framework, regardless of programming language, are executed by the CLR. All versions of the .NET framework include CLR. The CLR team was started June 13, 1998.

CLR implements the **Virtual Execution System (VES)** as defined in the **Common Language Infrastructure (CLI) standard**, initially developed by Microsoft itself. A public standard defines the Common Language Infrastructure specification.

With Microsoft's move to .NET Core, the CLI VES implementation is known as **CoreCLR** instead of CLR.

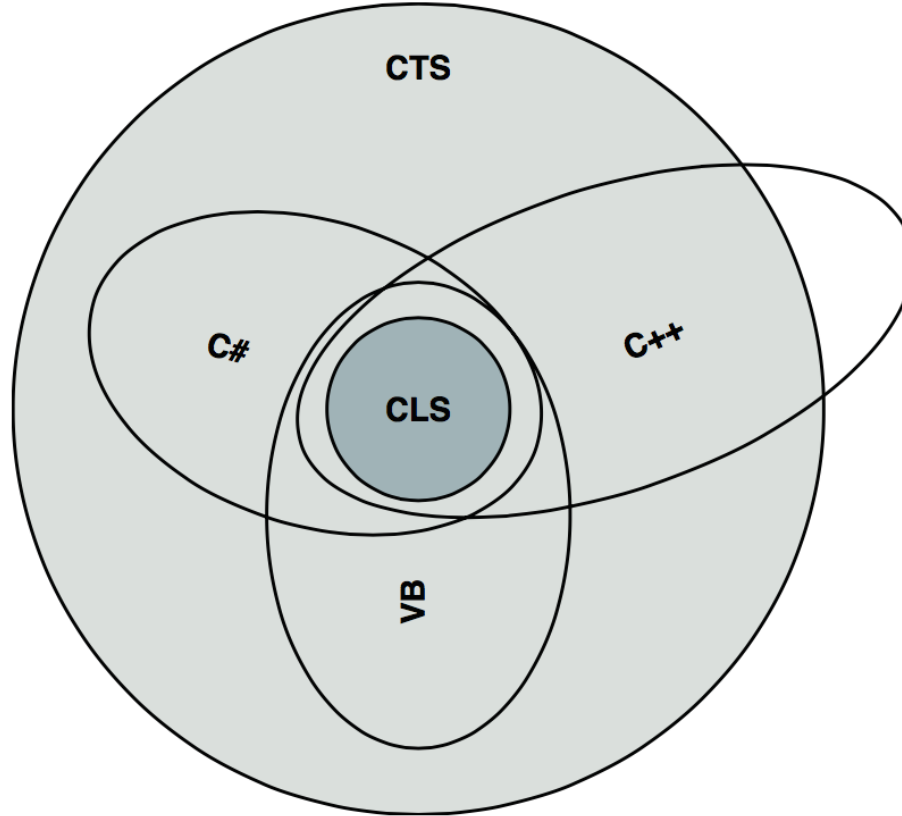
https://en.wikipedia.org/wiki/Common_Language_Runtime

Common Type System and Common Language Specification

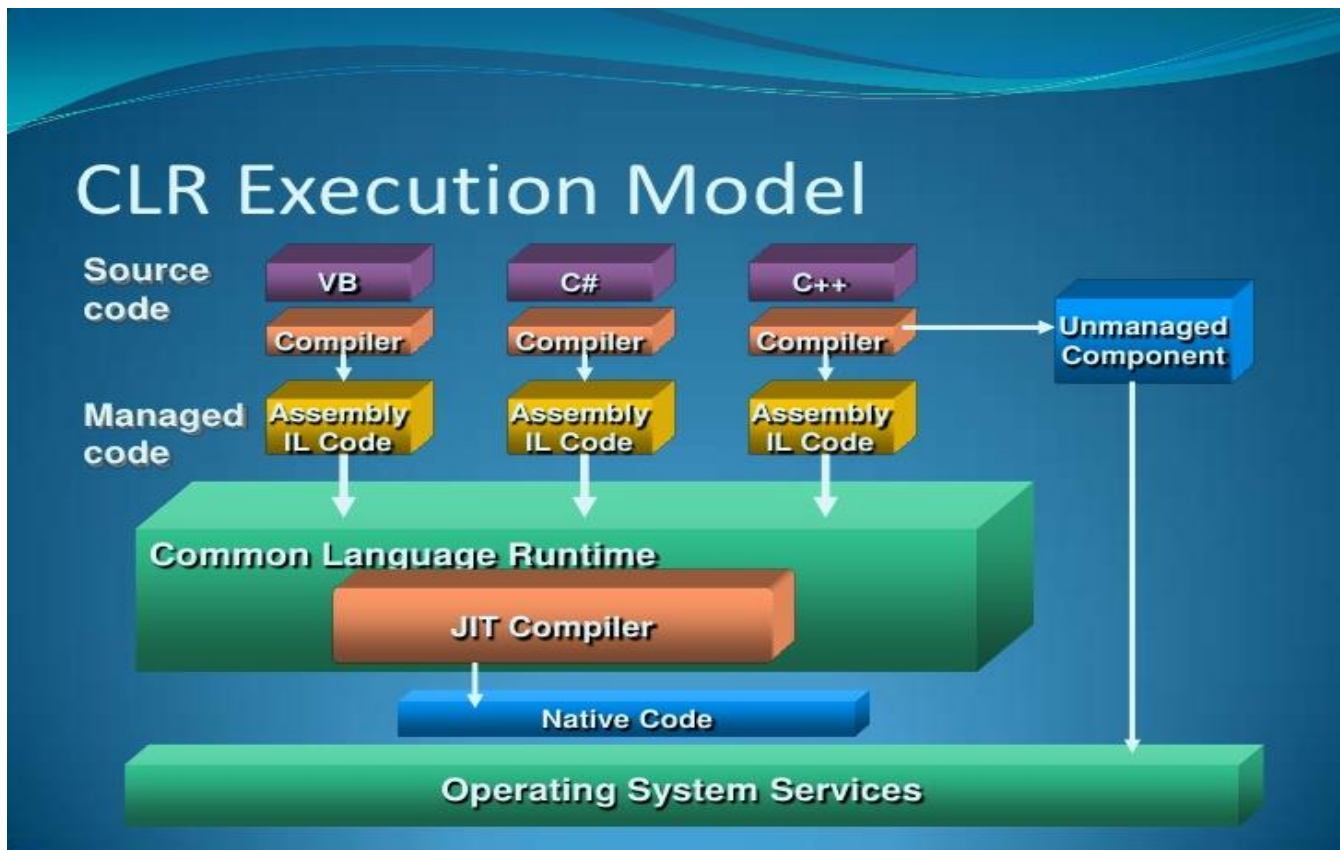
Common Type System (CTS) - defines how types are declared, used, and managed in the common language runtime, and is also an important part of the runtime's support for cross-language integration.

Common Language Specification (CLS) - is a subset of CTS, was designed to support language constructs commonly used by developers and to produce verifiable code, which allows all CLS-compliant languages to ensure the type safety of code.

Common Language Specification



Compilation in .NET



Let`s practice

1. Open Notepad
2. Create simple application

```
using System;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");
    }
}
```

3. Save file with .cs extension. (Hello.cs)
4. Run in a command line: csc Hello.cs

Compiler:

C:\Windows\Microsoft.NET\Framework64\v4.0.30319\csc

Common Intermediate Language

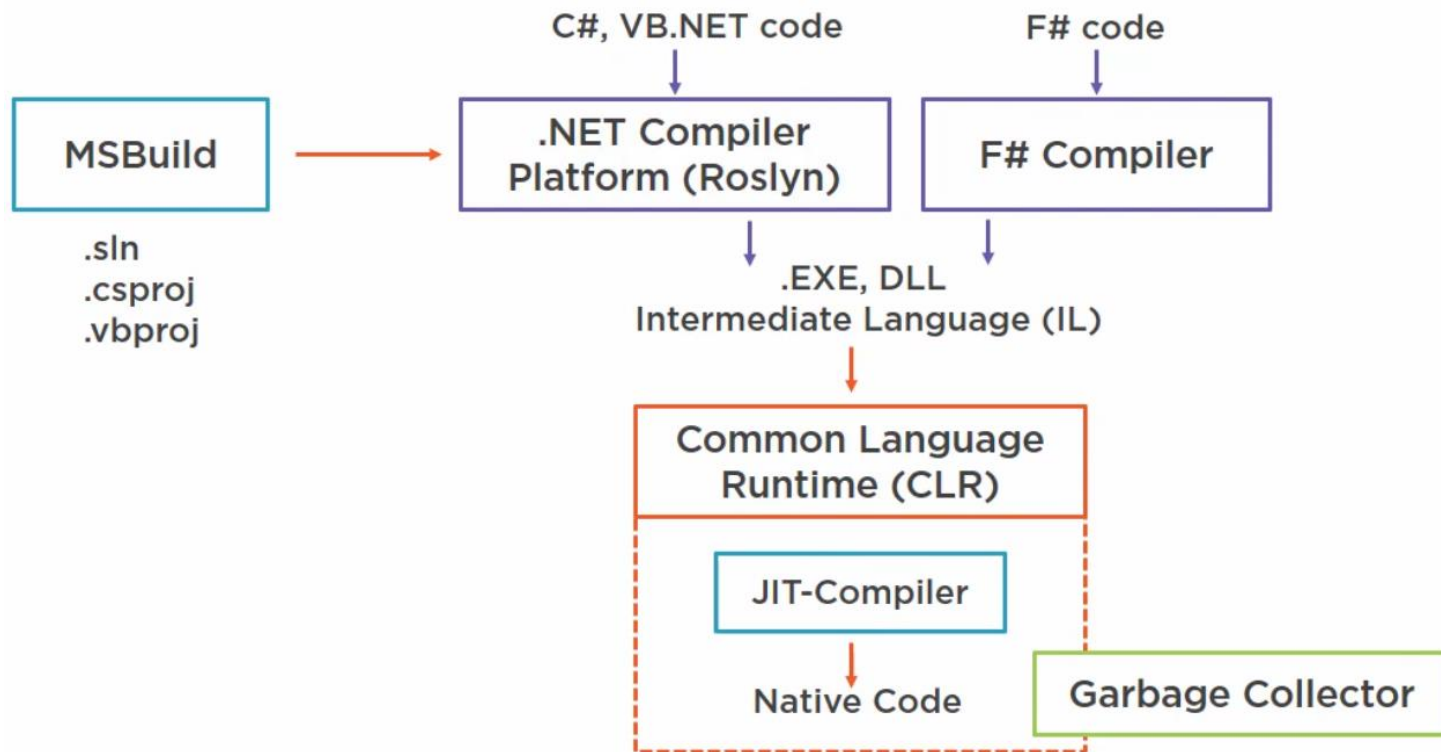
Common Intermediate Language (CIL), formerly called **Microsoft Intermediate Language (MSIL)** or **Intermediate Language (IL)**, is the intermediate language binary instruction set defined within the Common Language Infrastructure (CLI) specification. CIL instructions are executed by a CLI-compatible runtime environment such as the Common Language Runtime. Languages which target the CLI compile to CIL. CIL is object-oriented, stack-based bytecode. Runtimes typically just-in-time compile CIL instructions into native code.

Just-in-time compilation

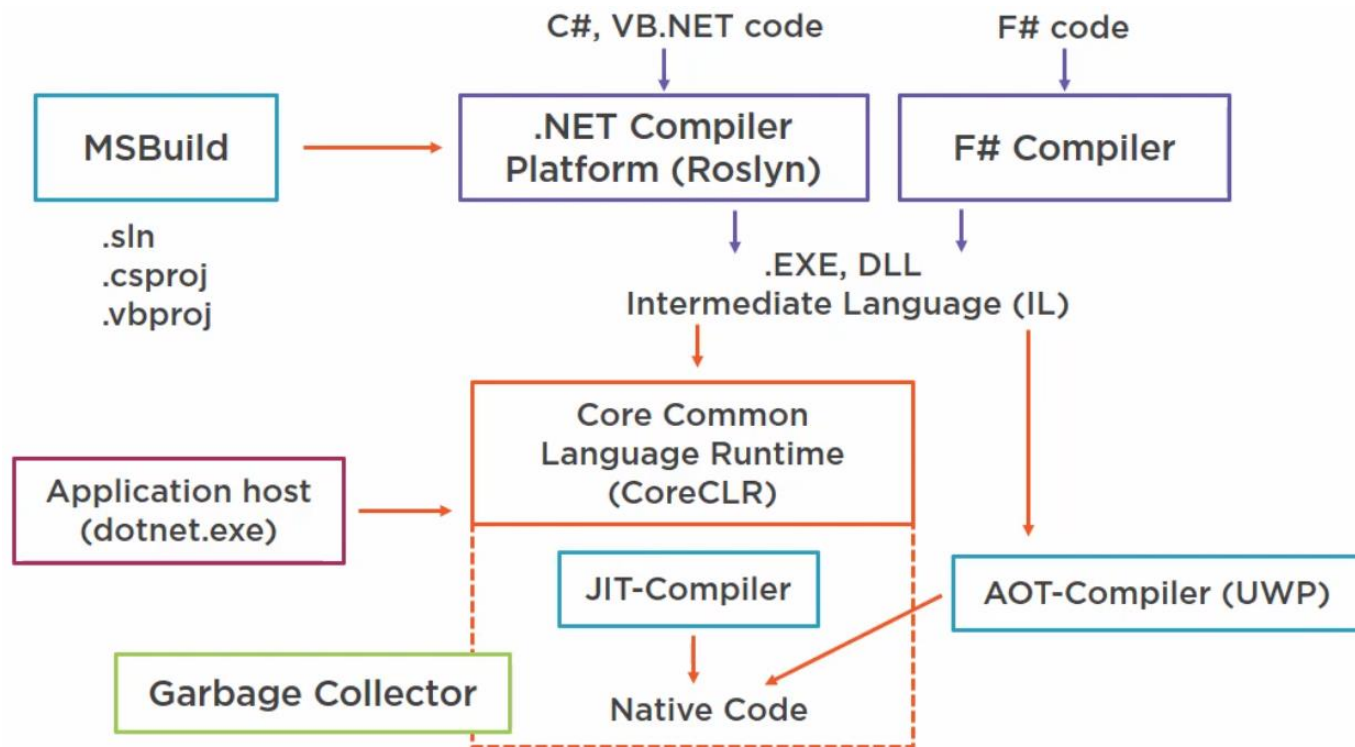
Just-in-time (JIT) compilation (also **dynamic translation** or **run-time compilations**) is a way of executing computer code that involves compilation during execution of a program – at run time – rather than before execution. Most often, this consists of source code or more commonly bytecode translation to machine code, which is then executed directly. A system implementing a JIT compiler typically continuously analyses the code being executed and identifies parts of the code where the speedup gained from compilation or recompilation would outweigh the overhead of compiling that code.

JIT compilation is a combination of the two traditional approaches to translation to machine code – **ahead-of-time compilation (AOT)**, and **interpretation** – and combines some advantages and drawbacks of both. Roughly, JIT compilation combines the speed of compiled code with the flexibility of interpretation, with the overhead of an interpreter and the additional overhead of compiling (not just interpreting). JIT compilation is a form of dynamic compilation and allows adaptive optimization such as dynamic recompilation and microarchitecture-specific speedups. Interpretation and JIT compilation are particularly suited for dynamic programming languages, as the runtime system can handle late-bound data types and enforce security guarantees.

Compilation in .NET Framework



Compilation in .NET Core



Let`s practice

1. Open Notepad
2. Create simple application

```
using System;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");
    }
}
```

3. Save file with .cs extension. (Hello.cs)
4. Run in a command line: csc Hello.cs

ILDASM:

C:\Program Files\Microsoft SDKs\Windows\vX.X\bin\ildasm.exe

Managed code vs Unmanaged code in .NET

Unmanaged Code

- Applications that are not under the control of the CLR are unmanaged
- The unsafe code or the unmanaged code is a code block that uses a pointer variable.
- The unsafe modifier allows pointer usage in unmanaged code.

Managed Code

Managed code is a code whose execution is managed by Common Language Runtime. It gets the managed code and compiles it into machine code. After that, the code is executed. The runtime here i.e. CLR provides automatic memory management, type safety, etc.

Managed code is written in high-level languages run on top of .NET. This can be C#, F#, etc. A code compiled in any of this language with their compilers, a machine code is not generated. However, you will get the Intermediate Language code, compiled and executed by runtime

C/C++ code, called "unmanaged code" do not have that privilege. The program is in binary that is loaded by the operating system into the memory. Rest, the programmer has to take care of.

Managed Code

Managed Code in Microsoft .Net Framework, is the code that has executed by the Common Language Runtime (CLR) environment. On the other hand Unmanaged Code is directly executed by the computer's CPU. Data types, error-handling mechanisms, creation and destruction rules, and design guidelines vary between managed and unmanaged object models.

The benefits of Managed Code include programmer's convenience and enhanced security. Managed code is designed to be more reliable and robust than unmanaged code , examples are Garbage Collection , Type Safety etc. The Managed Code running in a Common Language Runtime (CLR) cannot be accessed outside the runtime environment as well as cannot call directly from outside the runtime environment. This makes the programs more isolated and at the same time computers are more secure. Unmanaged Code can bypass the .NET Framework and make direct calls to the Operating System. Calling unmanaged code presents a major security risk.

.NET Garbage Collection

The .NET Framework provides a new mechanism for releasing unreferenced objects from the memory (that is we no longer needed that objects in the program), this process is called **Garbage Collection (GC)**. When a program creates an Object, the Object takes up the memory. Later when the program has no more references to that Object, the Object's memory becomes unreachable, but it is not immediately freed. The Garbage Collection checks to see if there are any Objects in the heap that are no longer being used by the application. If such Objects exist, then the memory used by these Objects can be reclaimed. So these unreferenced Objects should be removed from memory, then the other new Objects you create can find a place in the Heap.

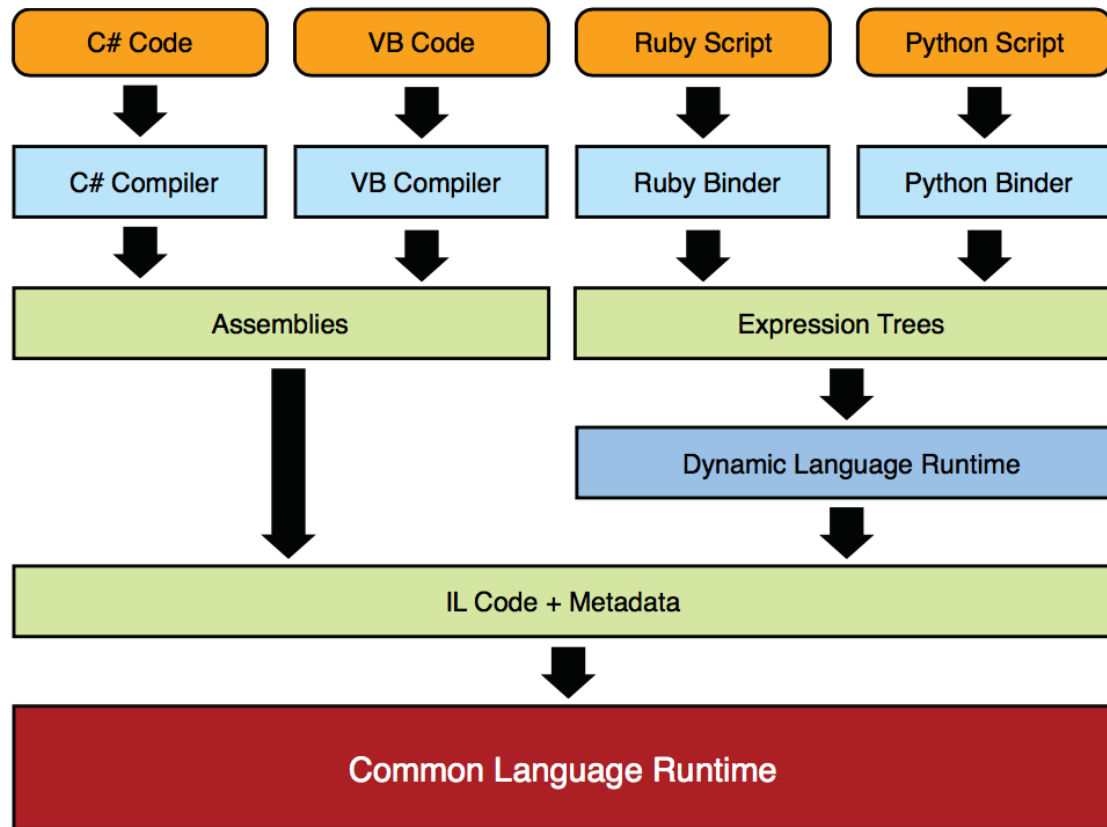
The reclaimed Objects have to be Finalized later. Finalization allows a resource to clean up after itself when it is being collected. This releasing of unreferenced Objects is happening automatically in .NET languages by the Garbage Collector (GC). The programming languages like C++, programmers are responsible for allocating memory for Objects they created in the application and reclaiming the memory when that Object is no longer needed for the program. In .NET languages there is a facility that we can call Garbage Collector (GC) explicitly in the program by calling `System.GC.Collect`.

Microsoft .NET Metadata

Metadata in .NET is binary information which describes the characteristics of a resource. This information include Description of the Assembly, Data Types and members with their declarations and implementations, references to other types and members , Security permissions etc. A module's metadata contains everything that needed to interact with another module.

During the compile time Metadata created with Microsoft Intermediate Language (MSIL) and stored in a file called a Manifest . Both Metadata and Microsoft Intermediate Language (MSIL) together wrapped in a Portable Executable (PE) file. During the runtime of a program Just In Time (JIT) compiler of the Common Language Runtime (CLR) uses the Metadata and converts Microsoft Intermediate Language (MSIL) into native code. When code is executed, the runtime loads metadata into memory and references it to discover information about your code's classes, members, inheritance, and so on. Moreover Metadata eliminating the need for Interface Definition Language (IDL) files, header files, or any external method of component reference.

Managed modules, MSIL code and Metadata



Microsoft .NET Assembly

Microsoft **.NET Assembly** is a logical unit of code, that contains code which the Common Language Runtime (CLR) executes. It is the smallest unit of deployment of a .NET application and it can be a **.dll** or an **exe**. Assembly is really a collection of types and resource information that are built to work together and form a logical unit of functionality. It includes both executable application files that you can run directly from Windows without the need for any other programs (.exe files), and libraries (.dll files) for use by other applications.

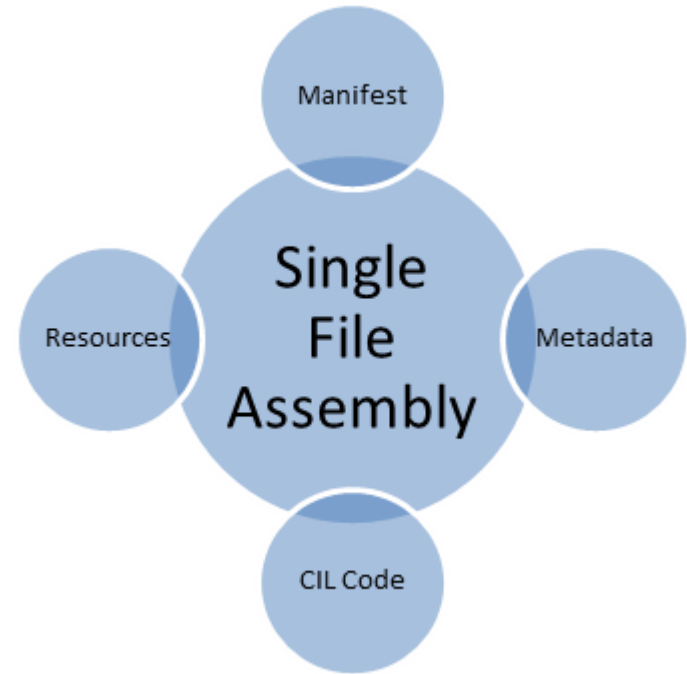
Assemblies are the building blocks of .NET Framework applications. During the compile time Metadata is created with Microsoft Intermediate Language (MSIL) and stored in a file called Assembly Manifest. Both Metadata and Microsoft Intermediate Language (MSIL) together wrapped in a Portable Executable (PE) file. Assembly Manifest contains information about itself. This information is called Assembly Manifest, it contains information about the members, types, references and all the other data that the runtime needs for execution.

Every Assembly you create contains one or more program files and a Manifest. There are two types of program files: Process Assemblies (EXE) and Library Assemblies (DLL). Each Assembly can have only one entry point (that is, DllMain, WinMain, or Main).

.NET Assembly Contents

A .NET static assembly can consist of following elements:

1. **Assembly Manifest** - The Metadata that describes the assembly and its contents.
2. **Type Metadata** - Defines all types, their properties and methods.
3. **MSIL** - Microsoft intermediate language
4. **A set of Resources** - All other resources like icons, images etc.



Only the assembly manifest is required, but either types or resources are needed to give the assembly in any meaningful functionality.

Types of Assembly

We can create two types of Assembly:

1. **Private Assembly** - is used only by a single application, and usually it is stored in that application's install directory
2. **Shared Assembly** - is one that can be referenced by more than one application.

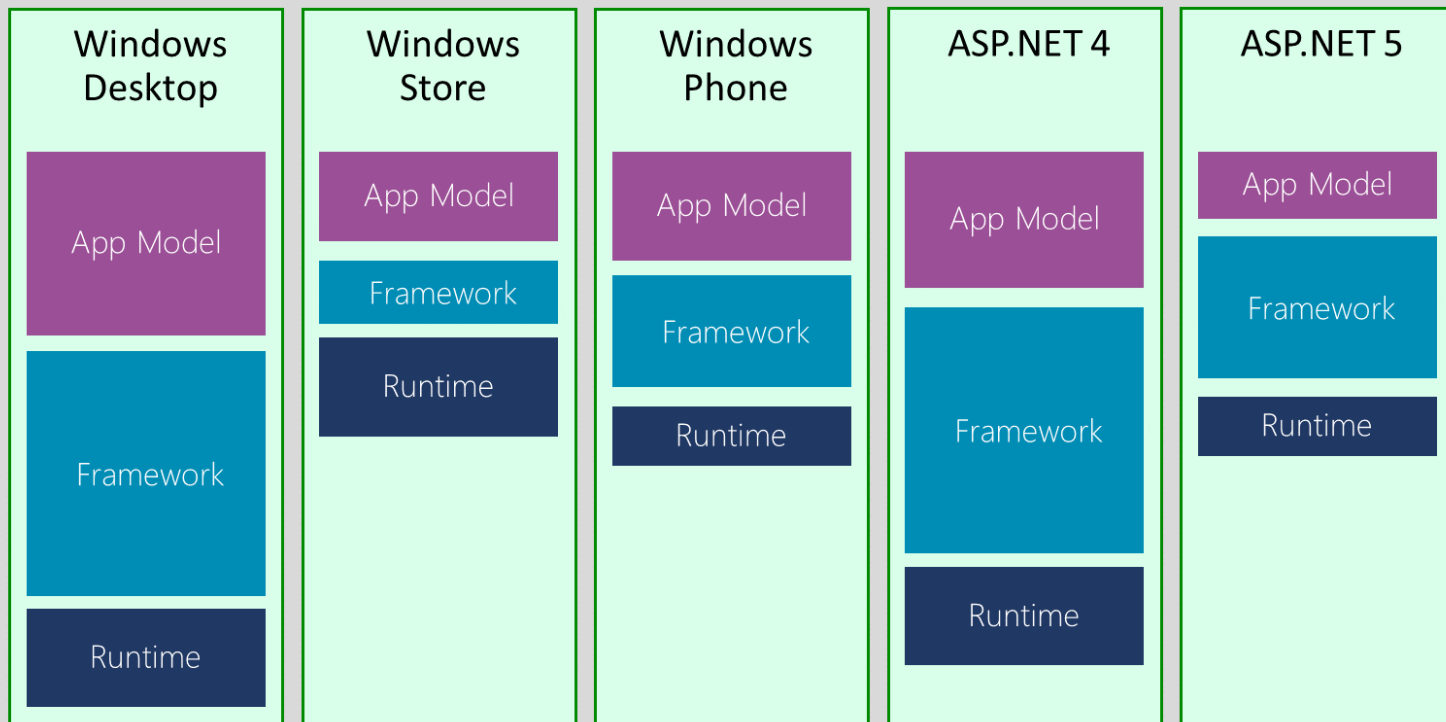
If multiple applications need to access an Assembly, we should add the Assembly to the **Global Assembly Cache (GAC)**.

There is also a third and least known type of an assembly: **Satellite Assembly**. A Satellite Assembly contains only static objects like images and other non-executable files required by the application.

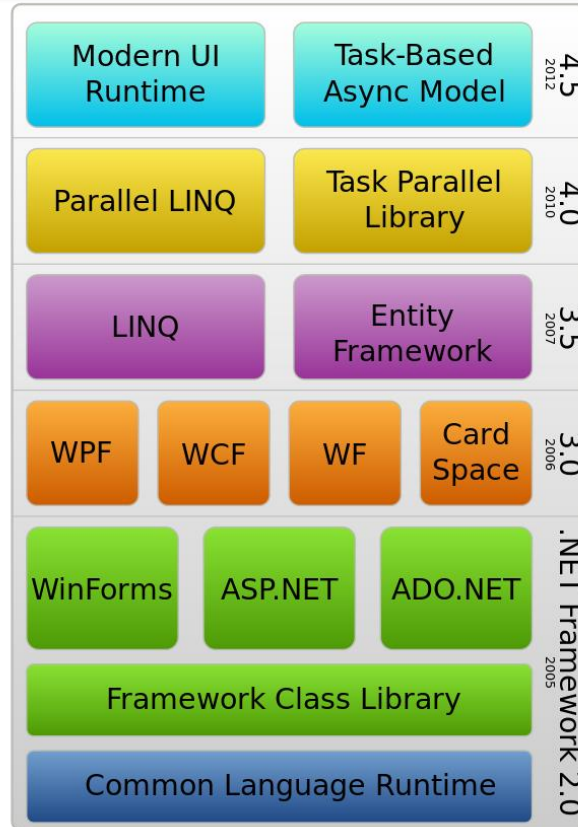
GAC is located in %windir%\assembly (for example, C:\WINDOWS\assembly) and it is a shared repository of libraries.

.NET versions overview

.NET



.NET Framework



.NET Framework history

.NET Framework	CLR	Release date	Visual Studio	Default build in Windows
1.0	1.0	2002-05-01	Visual Studio .NET	n/a
1.1	1.1	2003-04-01	Visual Studio .NET 2003	Windows Server 2003
2.0	2.0	2005-07-11	Visual Studio 2005	Windows Vista, Windows 7, Windows Server 2008 R2
3.0	2.0	2006-11-06	Visual Studio 2005 + extension	Windows Vista, Windows Server 2008, Windows 7, Windows Server 2008 R2
3.5	2.0	2007-11-09	Visual Studio 2008	Windows 7, Windows Server 2008 R2
4.0	4	2010-04-12	Visual Studio 2010	Windows 8, Windows Server 2012
4.5	4	2012-08-15	Visual Studio 2012	Windows 8, Windows Server 2012
4.5.1	4	2013-10-17	Visual Studio 2013	Windows 8.1, Windows Server 2012 R2
4.5.2	4	2014-05-05	n/a	n/a
4.6	4	2015-07-20	Visual Studio 2015	Windows 10
4.6.1	4	2015-11-17	Visual Studio 2015 Update 1	Windows 10 v1511
4.6.2	4	2016-07-20		Windows 10 v1607
4.7	4	2017-04-05	Visual Studio 2017	Windows 10 v1703
4.7.1	4	2017-10-17	Visual Studio 2017 v15.5	Windows 10 v1709, Windows Server 2016 (version 1709)
4.7.2	4	2018-04-30	Visual Studio 2017 v15.8	Windows 10 v1803
4.8	4	2019-04-18		Windows 10 v1903

.NET Framework and .NET Core

.NET 2015

.NET Framework



ASP.NET 5
ASP.NET 4.6
WPF
Windows Forms

.NET Core



ASP.NET 5
.NET Native



ASP.NET 5 for Mac and Linux

Common



Runtime

Next gen JIT
SIMD



Compilers

.NET Compiler Platform
Languages innovation



NuGet packages

.NET Core 5 Libraries
.NET Framework 4.6 Libraries

.NET Core

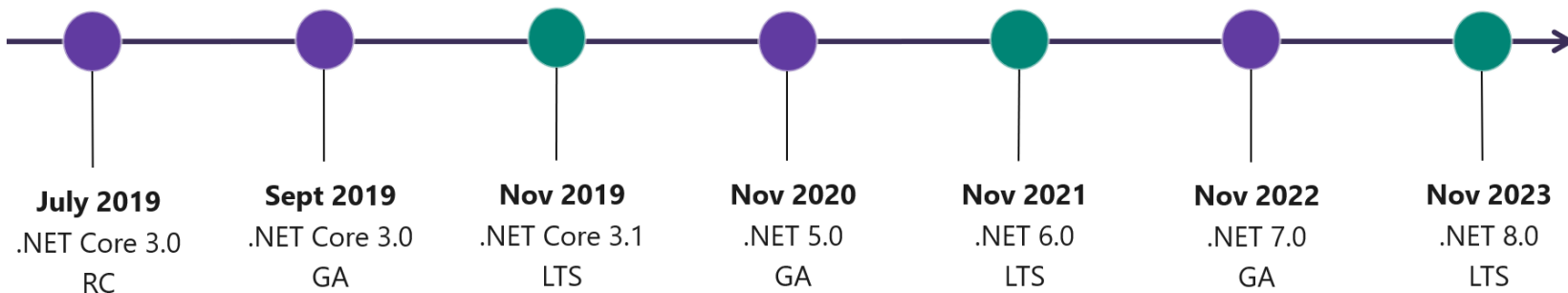
GIT: <https://github.com/dotnet>



.NET Core history

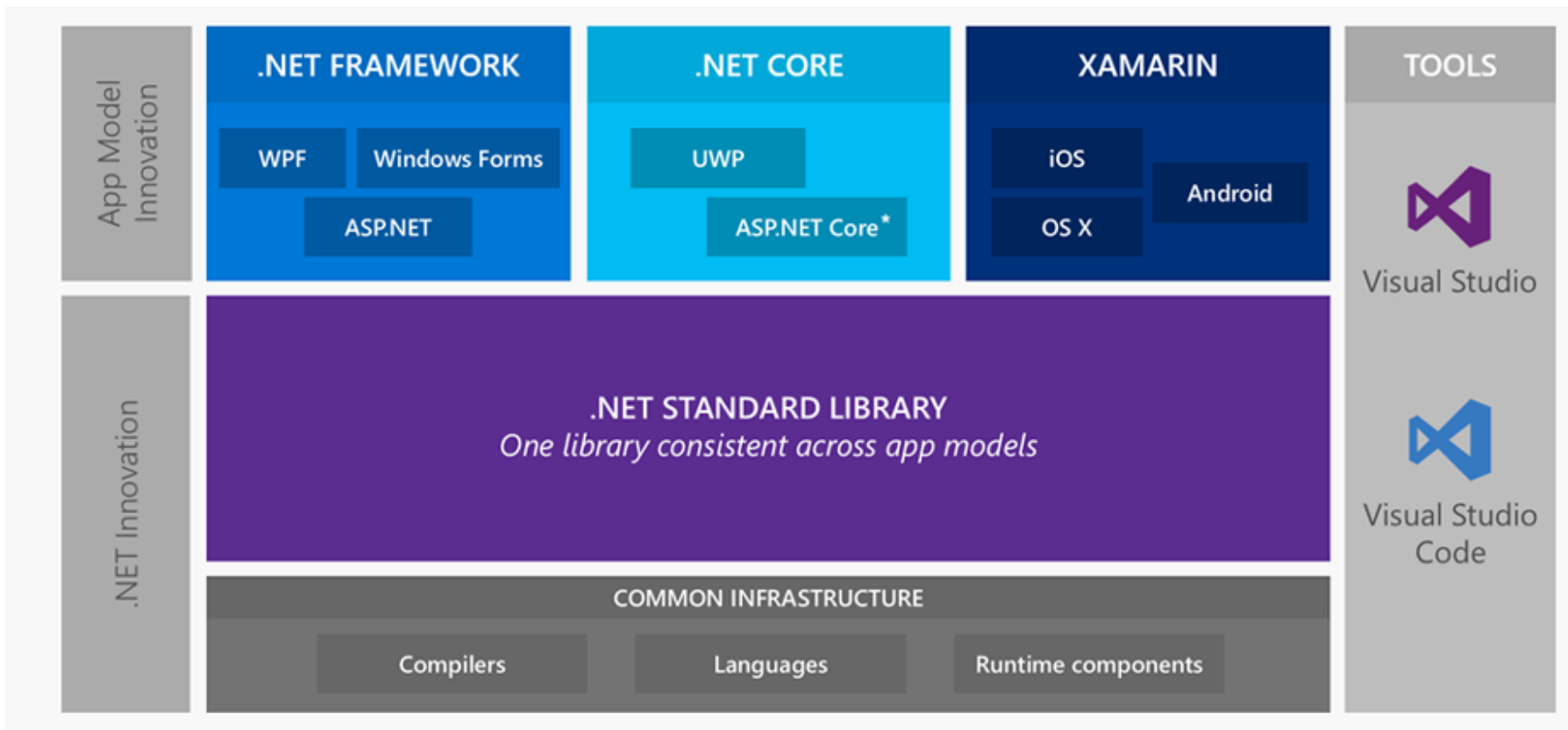
Version	Release date	Released with	Latest update	Latest update date	Support Ends
.NET Core 1.0	2016-06-27	Visual Studio 2015 Update 3	1.0.16	2019-05-14	June 27, 2019
.NET Core 1.1	2016-11-16	Visual Studio 2017 Version 15.0	1.1.13	2019-05-14	June 27, 2019
.NET Core 2.0	2017-08-14	Visual Studio 2017 Version 15.3	2.0.9	2018-07-10	October 1, 2018
.NET Core 2.1	2018-05-30	Visual Studio 2017 Version 15.7	2.1.19 (LTS)	2020-06-09	August 21, 2021
.NET Core 2.2	2018-12-04	Visual Studio 2019 Version 16.0	2.2.8	2019-11-19	December 23, 2019
.NET Core 3.0	2019-09-23	Visual Studio 2019 Version 16.3	3.0.3	2020-02-18	March 3, 2020
.NET Core 3.1	2019-12-03	Visual Studio 2019 Version 16.4	3.1.5 (LTS)	2020-06-09	December 3, 2022
.NET 5	2020-11 (projected)		5.0 Preview 5	2020-06-10	
.NET 6	2021-11 (projected)		(LTS)		
.NET 7	2022-11 (projected)				
.NET 8	2023-11 (projected)		(LTS)		

.NET Schedule



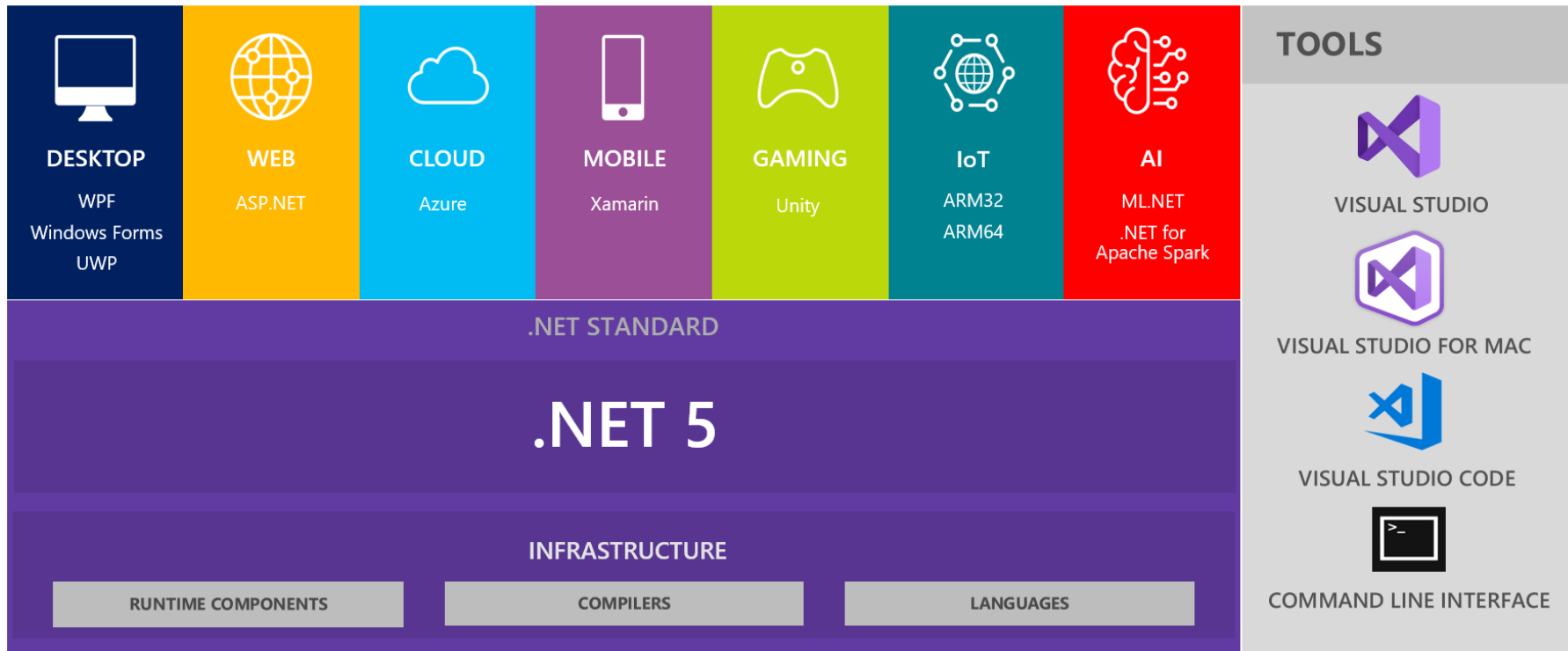
- .NET Core 3.0 release in September
- .NET Core 3.1 = Long Term Support (LTS)
- .NET 5.0 release in November 2020
- Major releases every year, LTS for even numbered releases
- Predictable schedule, minor releases if needed

.NET future innovation



.NET Core vNext = .NET 5

.NET – A unified platform



C# Versions overview (1.0 - 8.0)

[C# 1.0](#) (Visual Studio.NET)

[C# 2](#) (VS 2005)

[C# 3](#) (VS 2008)

[C# 4](#) (VS 2010)

[C# 5](#) (VS 2012)

[C# 6](#) (VS 2015)

[C# 7.0](#) (Visual Studio 2017)

[C# 7.1](#) (Visual Studio 2017 version 15.3)

[C# 7.2](#) (Visual Studio 2017 version 15.5)

C# 7.3 (Visual Studio 2017 version 15.7)

C# 8.0 (Visual Studio 2019 version 16.3, .NET Core 3.0)

<https://github.com/dotnet/csharplang/blob/master/Language-Version-History.md>

C# Version History

Version	.NET Framework	Visual Studio	Important Features
C# 1.0	.NET Framework 1.0/1.1	Visual Studio .NET 2002	•Basic features
C# 2.0	.NET Framework 2.0	Visual Studio 2005	•Generics •Partial types •Anonymous methods •Iterators •Nullable types •Private setters (properties) •Method group conversions (delegates) •Covariance and Contra-variance •Static classes
C# 3.0	.NET Framework 3.0/3.5	Visual Studio 2008	•Implicitly typed local variables •Object and collection initializers •Auto-Implemented properties •Anonymous types •Extension methods •Query expressions •Lambda expressions •Expression trees •Partial Methods

C# Version History

Version	.NET Framework	Visual Studio	Important Features
C# 4.0	.NET Framework 4.0	Visual Studio 2010	<ul style="list-style-type: none">•Dynamic binding (late binding)•Named and optional arguments•Generic co- and contravariance•Embedded interop types
C# 5.0	.NET Framework 4.5	Visual Studio 2012/2013	<ul style="list-style-type: none">•Async features•Caller information
C# 6.0	.NET Framework 4.6	Visual Studio 2013/2015	<ul style="list-style-type: none">•Expression Bodied Methods•Auto-property initializer•nameof Expression•Primary constructor•Await in catch block•Exception Filter•String Interpolation
C# 7.0	.NET Core 2.0	Visual Studio 2017	<ul style="list-style-type: none">•out variables•Tuples•Discards•Pattern Matching•Local functions•Generalized async return types•more..

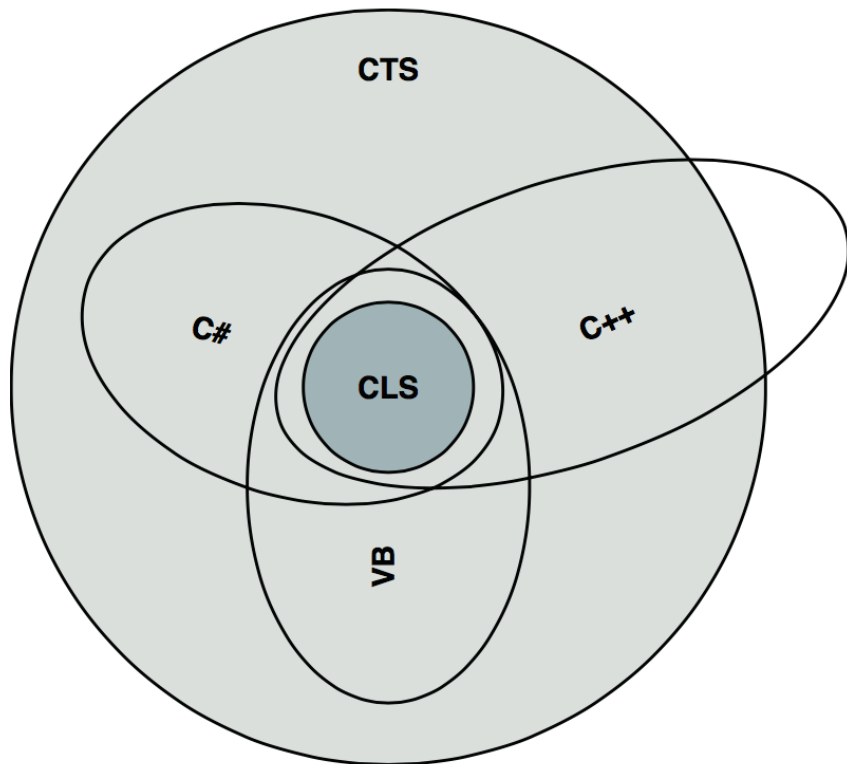
C# Version History

Version	.NET Framework	Visual Studio	Important Features
C# 8.0	.NET Core 3.0	Visual Studio 2019	<ul style="list-style-type: none">•Readonly members•Default interface methods•Using declarations•Static local functions•Disposable ref structs•Nullable reference types•more..

The compiler determines a default based on these rules:

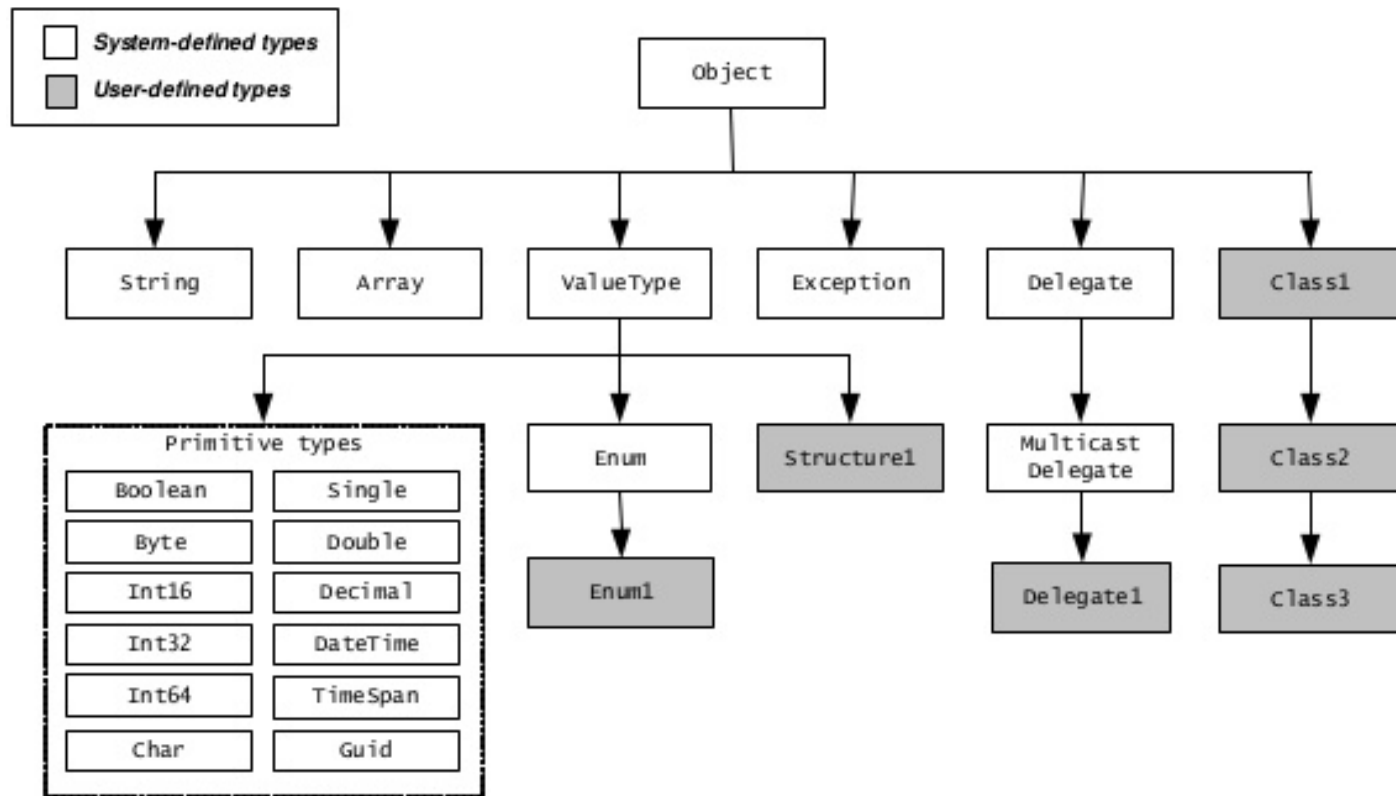
Target framework	version	C# language version default
.NET Core	3.x	C# 8.0
.NET Core	2.x	C# 7.3
.NET Standard	2.1	C# 8.0
.NET Standard	2.0	C# 7.3
.NET Standard	1.x	C# 7.3
.NET Framework	all	C# 7.3

CTS, value and reference types. Boxing/unboxing



Common Type System

Common Type System



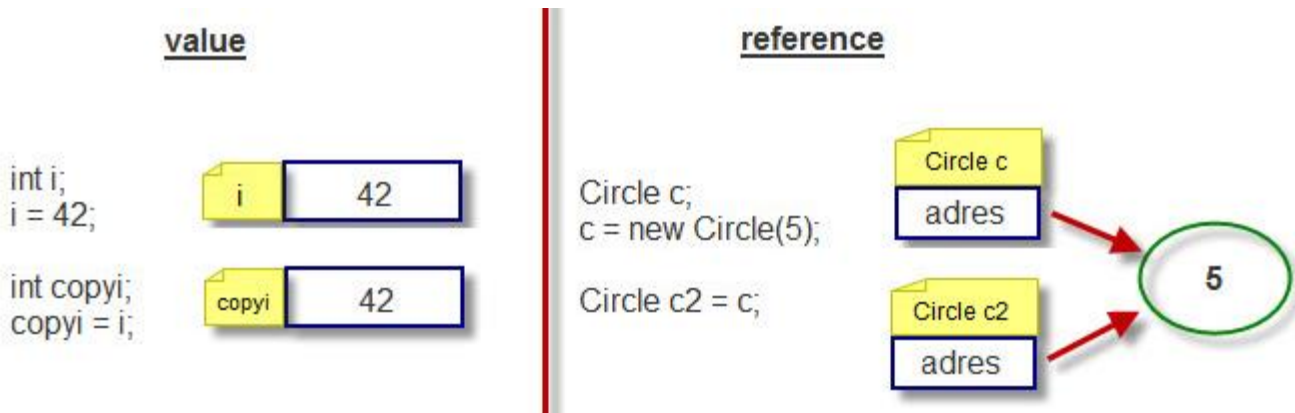
Types: Value vs. Reference

Value types and reference types are the two main categories of C# types.

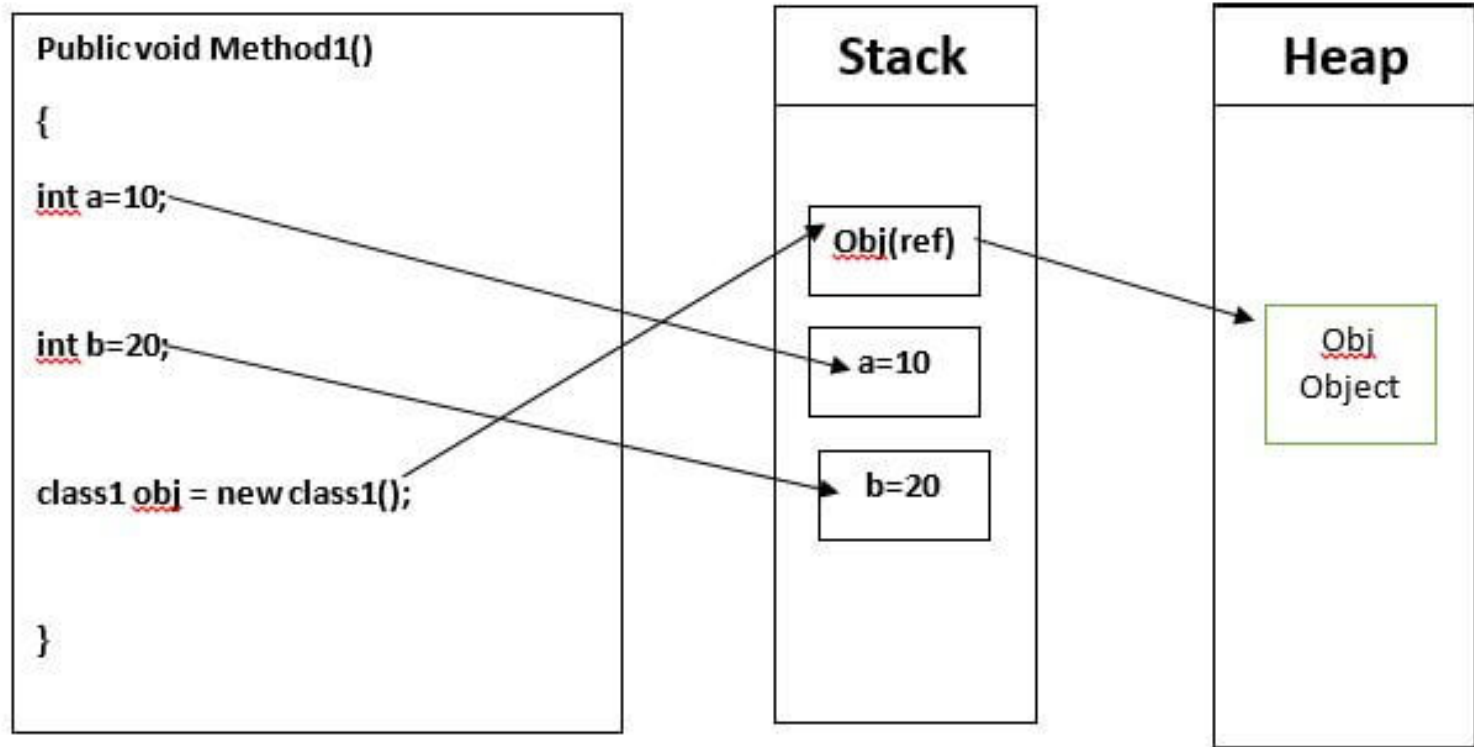
A variable of a value type contains an instance of the type.

This differs from a variable of a reference type, which contains a reference to an instance of the type.

By default, on assignment, passing an argument to a method, and returning a method result, variable values are copied. In the case of value-type variables, the corresponding type instances are copied.



Types: Stack or Heap



Value types

Kinds of value types:

- a structure type, which encapsulates data and related functionality
- an enumeration type, which is defined by a set of named constants and represents a choice or a combination of choices

A nullable value type $T?$ represents all values of its underlying value type T and an additional null value. You cannot assign null to a variable of a value type, unless it's a nullable value type.

Built-in value types (also known as simple types):

- Integral numeric types
- Floating-point numeric types
- bool that represents a Boolean value
- char that represents a Unicode UTF-16 character

Beginning with C# 7.0, C# supports value tuples. A value tuple is a value type, but not a simple type.

Value types: Built-in types

C# type keyword	.NET type
bool	System.Boolean
byte	System.Byte
sbyte	System.SByte
char	System.Char
decimal	System.Decimal
double	System.Double
float	System.Single

C# type keyword	.NET type
int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
short	System.Int16
Ushort	System.UInt16

Value types: Enumeration types

An enumeration type (or enum type) is a value type defined by a set of named constants of the underlying integral numeric type. To define an enumeration type, use the **enum** keyword and specify the names of enum members:

```
enum Season
{
    Spring,
    Summer,
    Autumn,
    Winter
}
```

```
enum ErrorCode : ushort
{
    None = 0,
    Unknown = 1,
    ConnectionLost = 100,
    OutlierReading = 200
}
```

You cannot define a method inside the definition of an enumeration type. To add functionality to an enumeration type, create an extension method.

Value types: Structure types

A structure type (or struct type) is a value type that can encapsulate data and related functionality.

```
public struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }
}
```

Structure types have *value semantics*.

```
public double X { get; }
public double Y { get; }
```

```
public override string ToString() => $"({X}, {Y})";
}
```


Value types: Structure types - Readonly struct

Beginning with C# 7.2, you use the readonly modifier to declare that a structure type is immutable:

```
Public readonly struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; }
    public double Y { get; }

    public override string ToString() => $"({X}, {Y})";
}
```

All data members of a readonly struct must be read-only as follows:

- Any field declaration must have the readonly modifier
- Any property, including auto-implemented ones, must be read-only

That guarantees that no member of a readonly struct modifies the state of the struct.

Value types: Nullable

A nullable value type $T?$ represents all values of its underlying value type T and an additional null value. For example, you can assign any of the following three values to a bool? variable: true, false, or null. An underlying value type T cannot be a nullable value type itself.

```
double? pi = 3.14;  
char? letter = 'a';
```

```
int m2 = 10;  
int? m = m2;
```

```
bool? flag = null;
```

```
// An array of a nullable value type:  
int?[] arr = new int?[10];
```

Reference types

Variables of reference types store references to their data (objects), while variables of value types directly contain their data. With reference types, two variables can reference the same object; therefore, operations on one variable can affect the object referenced by the other variable. With value types, each variable has its own copy of the data, and it is not possible for operations on one variable to affect the other (except in the case of in, ref and out parameter variables).

The following keywords are used to declare reference types:

- class
- interface
- delegate

C# also provides the following built-in reference types:

- dynamic
- object
- string

Built-in reference types: The object type

C# has a number of built-in reference types. They have keywords or operators that are synonyms for a type in the .NET library.

The object type

The object type is an alias for `System.Object` in .NET. In the unified type system of C#, all types, predefined and user-defined, reference types and value types, inherit directly or indirectly from `System.Object`.

You can assign values of any type to variables of type `object`. Any `object` variable can be assigned to its default value using the literal `null`.

When a variable of a value type is converted to `object`, it is said to be **boxed**. When a variable of type `object` is converted to a value type, it is said to be **unboxed**.

Built-in reference types: The string type

The string type represents a sequence of zero or more Unicode characters. string is an alias for System.String in .NET.

Although string is a reference type, the equality operators == and != are defined to compare the values of string objects, not references.

```
string a = "hello";  
string b = "h";  
// Append to contents of 'b'  
b += "ello"; Console.WriteLine(a == b);  
Console.WriteLine(object.ReferenceEquals(a, b));
```

Strings are *immutable* - the contents of a string object cannot be changed after the object is created, although the syntax makes it appear as if you can do this.

Built-in reference types: Nullable reference types

Nullable reference types are available beginning with C# 8.0, in code that has opted in to a *nullable aware context*. Nullable reference types, the null static analysis warnings, and the null-forgiving operator are optional language features. All are turned off by default. A *nullable context* is controlled at the project level using build settings, or in code using pragmas.

```
string notNull = "Hello";  
string? nullable = default;  
notNull = nullable!; // null forgiveness
```

Unmanaged type

A type is an **unmanaged type** if it's any of the following types:

- sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, or bool
- Any enum type
- Any pointer type
- Any user-defined struct type that contains fields of unmanaged types only and, in C# 7.3 and earlier, is not a constructed type (a type that includes at least one type argument)

Beginning with C# 7.3, you can use the unmanaged constraint to specify that a type parameter is a non-pointer, non-nullable unmanaged type.

Beginning with C# 8.0, a *constructed* struct type that contains fields of unmanaged types only is also unmanage

Boxing and Unboxing

Boxing is the process of converting a value type to the type object or to any interface type implemented by this value type.

```
int i = 123; // The following line boxes i.  
object o = i;
```

Boxing is used to store value types in the garbage-collected heap.

Boxing is an implicit conversion of a value type to the type object or to any interface type implemented by this value type.

Boxing a value type allocates an object instance on the heap and copies the value into the new object.

On the stack

i



int i=123;

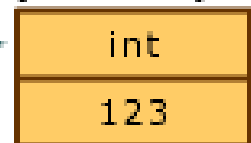
On the heap

o



object o=i;

(i boxed)



Boxing and Unboxing

Unboxing is an explicit conversion from the type object to a value type or from an interface type to a value type that implements the interface. An unboxing operation consists of:

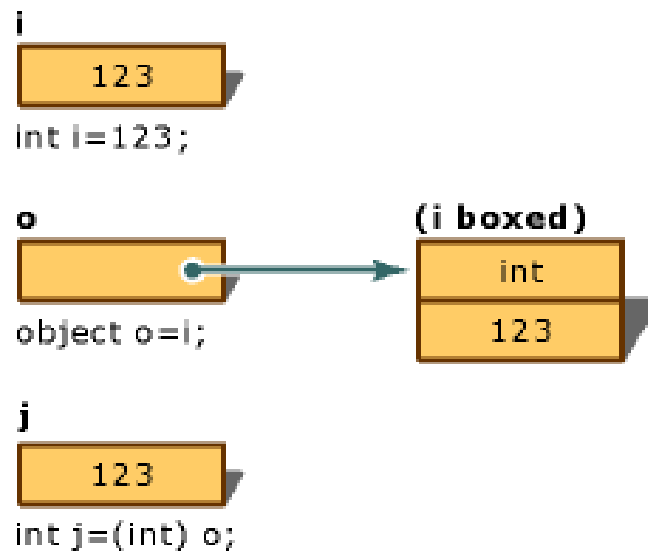
- Checking the object instance to make sure that it is a boxed value of the given value type.
- Copying the value from the instance into the value-type variable.

```
int i = 123; // a value type
object o = i; // boxing
int j = (int)o; // unboxing
```

For the unboxing of value types to succeed at run time, the item being unboxed must be a reference to an object that was previously created by boxing an instance of that value type.

Attempting to unbox null causes a `NullPointerException`.

Attempting to unbox a reference to an incompatible value type causes an `InvalidCastException`.



Boxing and Unboxing: .NET Performance Tips

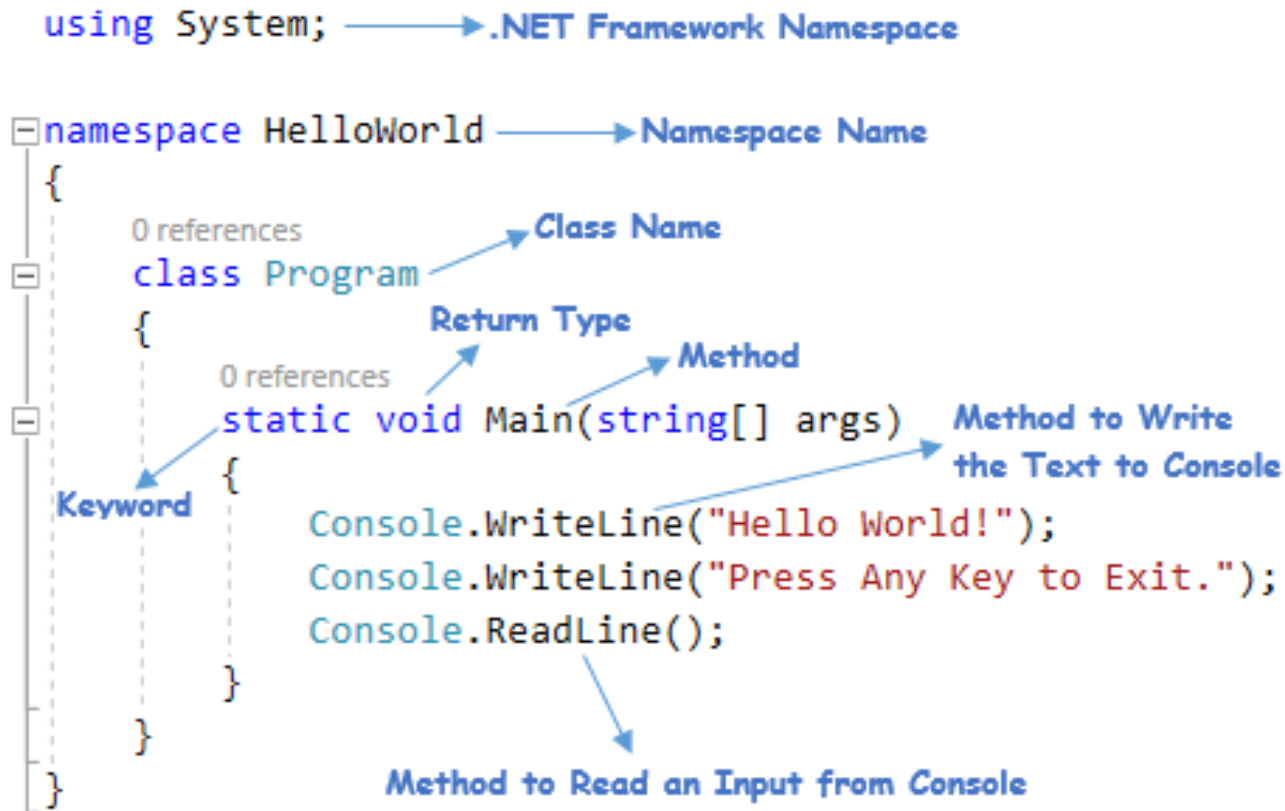
It is best to avoid using value types in situations where they must be boxed a high number of times, for example in non-generic collections classes such as `System.Collections.ArrayList`.

You can avoid boxing of value types by using generic collections such as `System.Collections.Generic.List<T>`.

Boxing and unboxing are computationally expensive processes. When a value type is boxed, an entirely new object must be created. This can take up to 20 times longer than a simple reference assignment. When unboxing, the casting process can take four times as long as an assignment.

C# operator basis

Program structure



C# Comments

- Single-line comments start with two forward slashes (//).

```
// This is a comment  
Console.WriteLine("Hello World!");
```

- C# Multi-line Comments

```
/* The code below will print the words Hello World  
to the screen, and it is amazing */  
Console.WriteLine("Hello World!");
```

Variables

<data_type> <variable_list>;

```
int i, j, k;  
char c, ch;           i = 100;  
float f, salary;  
double d;
```

<data_type> <variable_name> = value;

```
int d = 3, f = 5; /* initializing d and f. */  
byte z = 22; /* initializes z. */  
double pi = 3.14159; /* declares an approximation of pi. */  
char x = 'x'; /* the variable x has the value 'x'. */
```

Literals

The constants refer to fixed values that the program may not alter during its execution. These fixed values are also called **literals**. Constants can be of any of the basic data types like an integer constant, a floating constant, a character constant, or a string literal. There are also enumeration constants as well.

```
85 /* decimal */  
0x4b /* hexadecimal */  
30 /* int */  
30u /* unsigned int */  
30l /* long */  
30ul /* unsigned long */
```

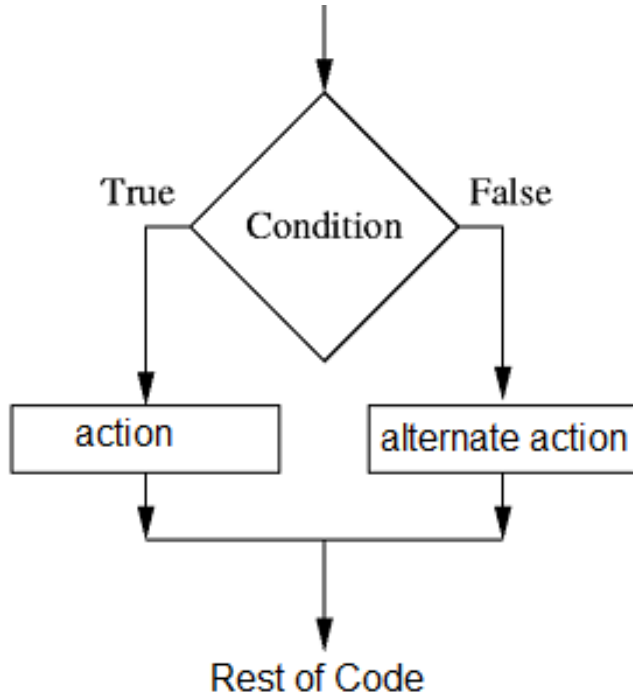
```
"hello, dear"  
"hello, \  
dear"  
"hello, " "d" "ear"  
@"hello dear"
```

```
3.14159 /* Legal */  
314159E-5F /* Legal */  
510E /* Illegal: incomplete exponent */  
210f /* Illegal: no decimal or exponent */  
.e55 /* Illegal: missing integer or fraction */
```

Defining Constants

```
const <data_type> <constant_name> = value;
```

C# if else statements

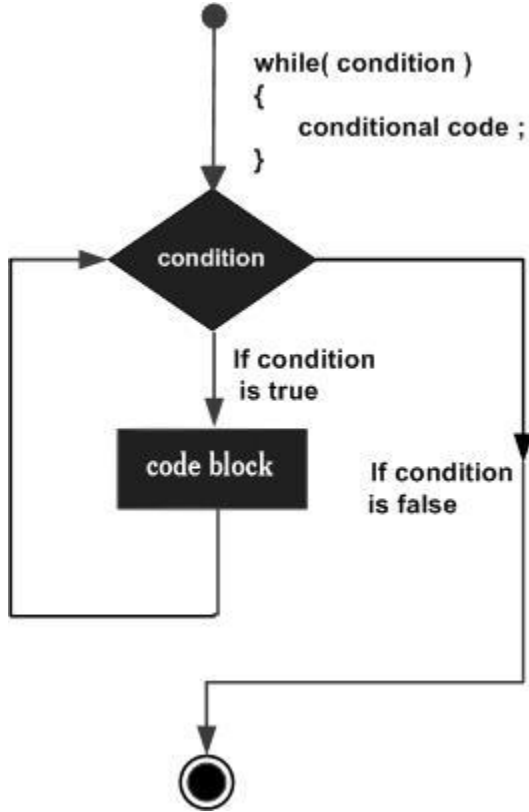


```
if (condition)  
    statement;
```

```
if (condition)  
    statement;  
else  
    statement;
```

```
if (condition)  
    statement;  
else if (condition)  
    statement;  
else  
    statement;
```

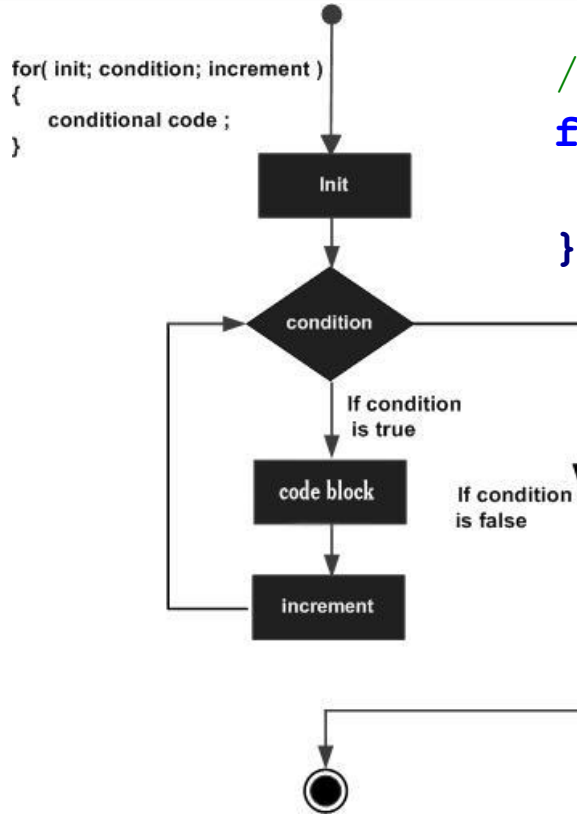

C# - Loops



```
while(condition) {  
    statement(s);  
}
```

```
/* while loop execution */  
while (a < 20) {  
    Console.WriteLine("value of a: {0}", a);  
    a++;  
}
```

C# - Loops

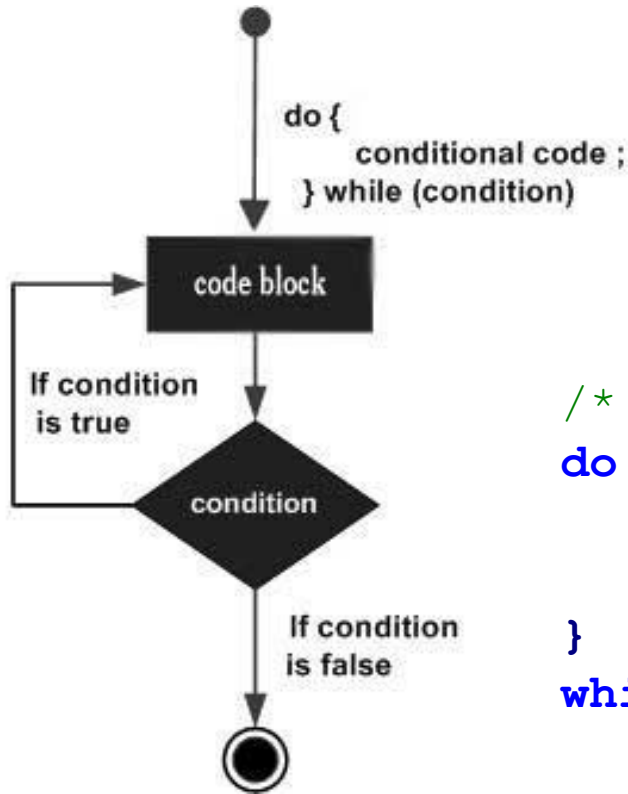


```
/* for loop execution */
```

```
for (int a = 10; a < 20; a = a + 1) {  
    Console.WriteLine("value of a: {0}", a);  
}
```

```
for ( init; condition; increment  
) {  
    statement(s);  
}
```

C# - Loops

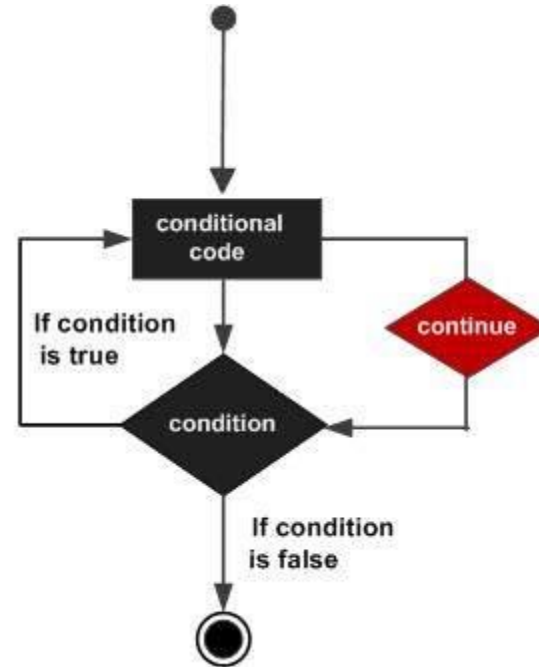
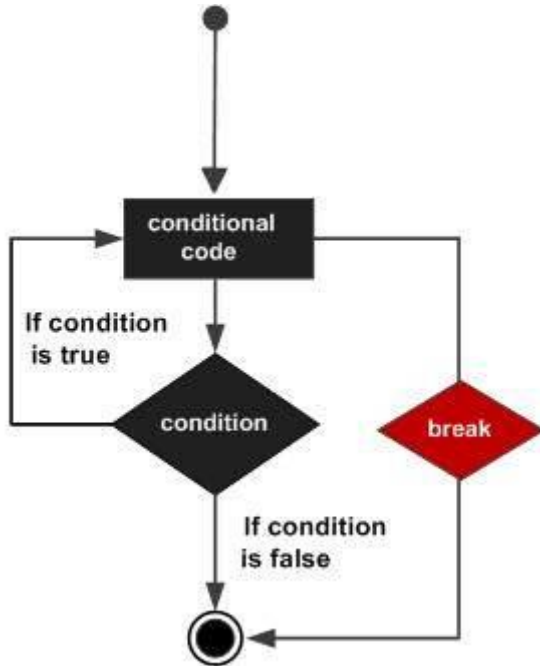


```
do {  
    statement(s);  
} while( condition );
```

```
/* do loop execution */
```

```
do {  
    Console.WriteLine("value of a: {0}", a);  
    a = a + 1;  
}  
while (a < 20);
```

C# - Loops: Break and Continue Statement

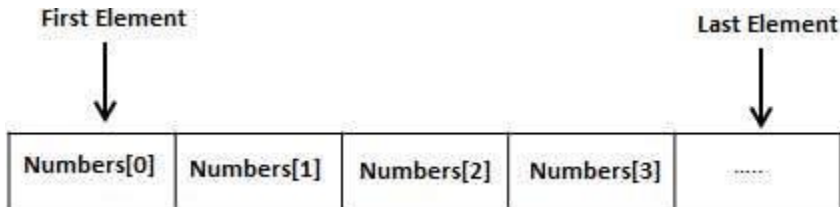


Switch Statements

```
switch(expression)
{
    case x:
        // code block
        break;
    case y:
        // code block
        break;
    default:
        // code block
        break;
}
```

```
int day = 4;
switch (day)
{
    case 1:
        Console.WriteLine("Monday");
        break;
    case 2:
        Console.WriteLine("Tuesday");
        break;
    case 3:
        Console.WriteLine("Wednesday");
        ;
        break;
    case 4:
        Console.WriteLine("Thursday");
        break;
    case 5:
        Console.WriteLine("Friday");
        break;
    case 6:
        Console.WriteLine("Saturday");
        break;
    case 7:
        Console.WriteLine("Sunday");
        break;
}
```

Arrays



```
datatype[] arrayName;
```

- *datatype* is used to specify the type of elements in the array.
- *[]* specifies the rank of the array. The rank specifies the size of the array.
- *arrayName* specifies the name of the array.

```
int[] nums2 = new int[4] { 1, 2, 3, 5 };  
int[] nums3 = new int[] { 1, 2, 3, 5 };  
int[] nums4 = new[] { 1, 2, 3, 5 };  
int[] nums5 = { 1, 2, 3, 5 };
```

.NET Online UA Training Course Feedback

I hope that you will find this material useful.

If you find errors or inaccuracies in this material or know how to improve it, please report on to the electronic address:

Oleksii_Leunenko@epam.com

With the note [.NET Online UA Training Course Feedback]

Thank you.

Q&A



DRIVEN



CANDID



CREATIVE



ORIGINAL



INTELLIGENT



EXPERT

UA .NET Online LAB